Escola Tècnica
Superior d'Enginyeria
Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Flow algorithms and their implementation

TREBALL FI DE GRAU

Grau en Enginyeria Informàtica

Autor: Máximo Pérez López

Tutor: Inge Li Gørtz, Salvador Boquera España

Curs 2020-2021

DTU

DTU Compute
Department of Applied Mathematics and Computer Science

Flow Algorithms and their implementation

Bachelor Thesis

Flow Algorithms and their implementation

Bachelor Thesis
June, 2021

By
Máximo Pérez López

Supervised by:

Inge Li Gørtz

Salvador España Boquera

Copyright: Reproduction of this publication in whole or in part must include the customary bibliographic citation, including author attribution, report title, etc.
Cover photo: Vibeke Hempler, 2012
Published by: DTU, Department of Applied Mathematics and Computer Science, Richard Petersens Plads, Building 324, 2800 Kgs. Lyngby Denmark
https://www.compute.dtu.dk/english

Abstract

The maximum flow problem is polynomially solvable, and multiple efficient algorithms have been discovered since the first one by Ford and Fulkerson. We study some of the most relevant algorithms that use augmenting paths, as well as several versions of push-relabel algorithms (FIFO and highest vertex selection). We analyse their worst-case complexity and their implementation details. Furthermore, we do a custom implementation of them in Java. We show an original way of selecting the highest active vertex in constant time, and also an implementation of the exact distance labelling feature for highest vertex selection. Finally, we test the algorithms against two different types of networks: sparse AK networks, that are shown to have a quadratic lower bound on the algorithms under study, and fully dense acyclic networks.

Acknowledgements

Máximo Pérez López, BSc Computer Engineering, UPV
Author of the thesis.

Inge Li Gørtz, Professor, Department of Applied Mathematics and Computer Science, DTU
Supervisor

Salvador España Boquera, Professor, Department of Information Systems and Computation (DSIC), UPV
Co-Supervisor

Contents

Abstract . . . ii
Acknowledgements . . . iii

1 Introduction 1
1.1 Motivation and objectives . . . 1
1.2 Sections . . . 1
1.3 State of the art . . . 1

2 Preface and notation 3
2.1 Graph, flow, maximum flow problem . . . 3
2.2 Degree, path, distance, cycle, connected graph . . . 4
2.3 Cuts in networks . . . 5
2.4 Complexity, sparse and dense graphs . . . 6

3 Augmenting path algorithms 7
3.1 Augmenting paths . . . 7
3.2 Residual Graph G_f . . . 8
3.3 Ford-Fulkerson algorithm . . . 8
3.4 Edmonds-Karp algorithm . . . 11
3.5 Scaling Edmonds-Karp algorithm . . . 12
3.6 Dinic's algorithm . . . 14

4 Implementation of Augmenting Paths 17
4.1 Graph implementation . . . 17
4.2 Edmonds-Karp implementation . . . 18
4.3 Scaling Edmonds-Karp implementation . . . 18
4.4 Dinic's algorithm implementation . . . 18

5 Preflow-push algorithms 23
5.1 Introduction . . . 23
5.2 Preflows . . . 23
5.3 Generic Preflow-Push Algorithm . . . 24
5.4 Improvements to the generic algorithm . . . 32
5.5 FIFO Algorithm . . . 36
5.6 Highest-Level algorithm . . . 38
5.7 Heuristics . . . 43

6 Implementation of push-relabel algorithms 45
6.1 Introduction . . . 45
6.2 Residual graph and Vertex structures . . . 45
6.3 Linked lists instead of sets . . . 45
6.4 Generic algorithm . . . 46
6.5 FIFO Algorithm . . . 46
6.6 Highest Vertex Algorithm . . . 46
6.7 Gap relabelling . . . 47
6.8 Global Relabelling . . . 48
6.9 Exact labelling . . . 48

6.10 Pseudo-codes for FIFO and Highest Vertex selection 49

7 Networks and results **55**
7.1 AK Networks 55
7.2 Acyclic networks 58
7.3 Computational results 58

Bibliography **63**

A Java code **65**

1 Introduction

The maximum flow problem is an optimization problem on a special kind of graphs called flow networks. Flow networks have a number associated to each edge, called a capacity, and the aim is to find how much *flow* can be pushed from a given source vertex *s* to a given sink vertex *t*. Intuitively, the maximum flow problem looks like a pipe routing problem: the edges in the graph can be thought of as pipes, and the capacity of each edge is how much flow of water can the pipe transport, let's say, in an hour. So, given a pipe network, we want to know how much water can we pump through the network from a point *s* to a point *t* in an hour. This will depend on the capacities of each edge and on the shape of the network.

1.1 Motivation and objectives

The maximum flow problem is one that has been extensively studied, both for its numerous applications and its theoretical interest. It has many different efficient algorithms that can be used to solve it, and research is still being carried out today, with some new theoretical results having been published in the last five years. However, despite the amount of theoretical algorithms, it is sometimes unclear whether all of these are efficient in practice. In some cases, a simpler algorithm that could perform badly in a worst case scenario can actually outperform some more complex algorithms in practical examples.

We seek two main objectives in this thesis. The first one is to provide a comprehensive and accessible explanation of push-relabel and augmenting path methods, covering both computational complexity analysis and implementation details. Specially, we seek to cover the most relevant push-relabel improvement heuristics, and explain the details of an exact labelling. The second aim is to implement the algorithms in Java and compare them against two different type of networks: AK networks, which have a quadratic complexity lower bound on all the algorithms that we study, and fully dense acyclic networks.

1.2 Sections

The thesis will be divided in seven sections. We will finish this introduction section with a discussion of the state-of-the-art in maximum flow algorithms. The second section will introduce the notation and basic definitions concerned with the maximum flow problem. The third section is devoted to describe and analyze theoretically the augmenting path algorithms. The fourth section will describe the relevant implementation decisions that were made in those algorithms. The fifth section will present and explain the push-relabel method theoretically, as well as their heuristics and improvements. The sixth section will deal with the details of the latter ones. And finally, the last section will describe the test networks and present and interpret the computational results achieved on them. In the appendix there will be links and information about the Java code that was implemented as part of the project.

1.3 State of the art

The maximum flow problem has been extensively studied since the first approach of Ford and Fulkerson [13] in the 1950s. Since then, Edmonds-Karp provided the first strongly polynomial algorithm [6] and Dinitz [5] presented an improvement to find all shortest paths of the same length at once. Karzanov presented the first $O(n^3)$ algorithm with the idea of preflows [11]. Building on that idea, Goldberg and Tarjan presented the push-relabel

algorithm [8] that we will study in detail. Since then, many more algorithms have been developed. As of 2019, the algorithm shown by Orlin and Gong [17] has the best asymptotic worst-case running time, of $O(nm \log n/(\log\log n + \log \frac{m}{n}))$.

2 Preface and notation

In this chapter we set straight the notation and the basic definitions that we will use throughout the whole project. The notation is inspired in that of Bondy Murty [15].

2.1 Graph, flow, maximum flow problem

- **Graph:** A graph G is a pair $G = (V, E)$ consisting of a finite set of vertices V, graphically represented as points, and a finite set of edges E where, graphically, each edge connects two vertices. These vertices are called the *endpoints* of the edge. If the edges have an orientation, we say that the graph is *directed*, otherwise we say that the graph is *undirected*. If two vertices are connected by an edge, we say that they are *adjacent*.

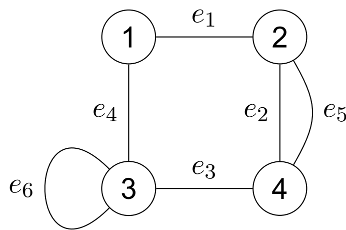


Figure 2.1: An undirected multigraph, with vertex set $V = \{1, 2, 3, 4\}$ and edge set $E = \{e_1, e_2, e_3, e_4, e_5, e_6\}$

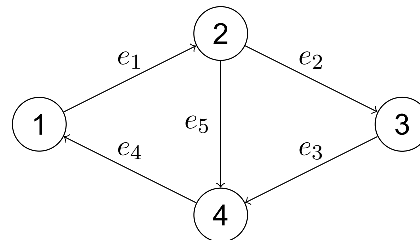


Figure 2.2: A directed simple graph, with vertex set $V = \{1, 2, 3, 4\}$ and edge set $E = \{e_1, e_2, e_3, e_4, e_5\}$

  If two or more edges can connect two given vertices (in the same direction if applicable), or the endpoints can be the same vertex (called a *loop*), the graph is called a *multigraph*, otherwise it is called a *simple graph*. Futhermore, if an undirected edge e connects two vertices x and y, we write $e = xy$. Otherwise, if the edge is directed from x to y, we write instead $e = (x, y)$, and we say that e is incident from x and incident to y.

  In this thesis we're interested in a special kind of directed graphs called *flow networks*.

- **Flow network:** A directed graph where each edge e_i is assigned a real, non-negative number c_i called the capacity. Two vertices are special: the source and the sink, denoted by s and t. In this thesis we will work only with integer capacities, and we will refer to a flow network just as a *network*.

- **s-t Flow:** An $s-t$ flow in a network is, mathematically, a function from the edge set to the non-negative reals $f : E \to \mathbb{R}_{\geq 0}$ that satisfies two constraints:

  – Capacity constraint: The flow of any edge cannot exceed its capacity:

$$f_e \leq c_e$$

– Conservation constraint: The sum of flows that go into a vertex $v \neq s, t$ equals the sum of flows coming out of the vertex:

$$\sum_{e \text{ into } v} f_e = \sum_{e \text{ out of } v} f_e$$

The sum of flows that go into a vertex will be called *incoming flow*, and the sum of flows going out will be called *outgoing flow*.

There may be flow going out of the source without coming in, and flow coming into t without coming out (hence the names source and sink).

We will only work with flows of integer values. In our pictures, an edge that has flow f and capacity c is written as "f/c":

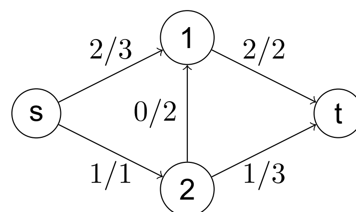


Figure 2.3: A flow network with an example flow.

– The **value of the flow** is defined as the sum of flows going out of the source, or what is the same, the sum of flows going into the sink. Note that there's a maximum flow value, that is bounded above by the capacities of the edges going out of s, but that can be lower depending on the other edges (as in the figure). The flow with all zero values is always valid. We denote the value of the flow f by $F(f)$.

– If an edge has flow equal to its capacity, $f_e = c_e$, then we say that the edge is *saturated*. Otherwise, the edge is *unsaturated*.

- **Maximum flow problem:** It is the problem of finding the maximum value of an $s-t$ flow in a given network with a given source and sink.

  Note that, for this problem, simple networks are enough to consider, since two edges that connect the same vertices can be exchanged for one edge with the sum of both capacities. Also, loops will be irrelevant to the maximum flow value.

2.2 Degree, path, distance, cycle, connected graph

- **Degree:** The degree of a vertex v, denoted $d(v)$, is the number of edges that have v as an endpoint. If the graph is directed, the number of edges incident from v is called the *in-degree* $d^-(v)$, and the number of edges incident to v is called the *out-degree* $d^+(v)$.

- **Path:** A *path* from a vertex v_0 to a different vertex v_k is an alternating sequence of vertices and edges, $P = v_0 e_1 v_1 \dots e_k v_k$ such that, for $1 \leq i \leq k$, v_{i-1} and v_i are the endpoints of e_i, and all the vertices are different.

  – If the edges are directed and all of them go from v_{i-1} to v_i, we call the path a *strongly directed* path. Otherwise, if there are edges that go in the reverse direction, we say that the path is *weakly directed*. The edges in the direction

from v_0 to v_k are called *forward* edges, and the others are called *backward* edges.

– We call k the length of the path, and denote it by $|P|$. Note that, because the path starts and ends with vertices, P has k edges and $k+1$ vertices. In a simple graph, if backward and forward edges don't lead to confusion, we may write $P = v_0v_1 \dots v_k$.

– A directed path from x to y will be denoted an $(x \to y)$-path, and the undirected version, (x, y)-path.

– The vertex v_0 is called the *start vertex* of the path, the vertex v_k is called the *end vertex*, and all others are called *internal vertices* of the path.

- **Cycle:** A cycle is a path where the first and the last vertex are the same: $v_0 = v_k$.

- **Distance:** The distance between two vertices x and y in a graph G, denoted $dist_G(x, y)$ or $dist(x, y)$ if there's no confusion, is the length of the shortest $(x \to y)$-path in G. By convention, $dist(x, x) = 0$.

- **Connected or disconnected graph:** We say that a graph is connected if, for any pair of vertices x and y, there exists a path from x to y in the graph. Otherwise, we say that the graph is disconnected. All flow networks are connected graphs.

- **Connected component:** A disconnected graph is formed by parts that are connected. We call each one of this parts a *connnected component*.

- **Tree and forest:** A tree is a connected graph with no cycles, acyclic. A forest is a disconnected graph where all the connected components are trees. We can have directed or undirected versions.

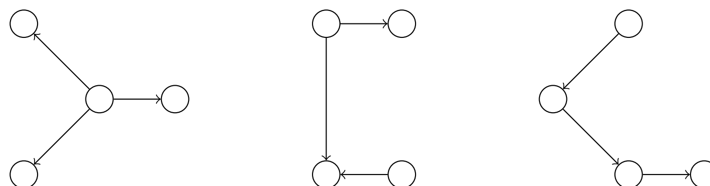


Figure 2.4: A directed forest with three connected components (trees). The connected component on the left is an ordinary tree, the middle one is a weakly directed path, and the one on the right is a strongly directed path.

2.3 Cuts in networks

- **s-t Cut:** Let $N = (V, E)$ be a network. Let $S \subset V$ be a non-empty subset of vertices where $s \in S$ and $t \notin S$, and let $\overline{S} = V \setminus S$. An *s-t cut* $K = (S, \overline{S})$ is the set of all the edges (x, y) where $x \in S$ and $y \in \overline{S}$. We will call them just cuts.

  – Note that the cut $(S, \overline{S})$ only considers edges that go from S to $\overline{S}$, but not the ones that go from $\overline{S}$ to S.

- **Capacity of a cut:** Let K be a cut in a network. The capacity of the cut, denoted as cap K, is the sum of the capacities of the edges in the cut. A cut in the network that has the minimum capacity possible is called a *minimum cut*.

- **Flow across a cut:** Let $K = (S, \overline{S})$ be a cut. The flow across the cut is the sum of flow values of the edges of the cut, minus the sum of flow values of all the edges

going from $\overline{S}$ to S.

2.4 Complexity, sparse and dense graphs

To analyse the theoretical complexity of an algorithm, we will use the standard big O notation. We will use the terms complexity or cost with the same meaning. Let us state the definition:

- **Big O Notation:** If $f(n)$ is the cost function of an algorithm, regardless if it denotes the time cost or the memory cost, and n is the size of the input, we say that $f(n) = O(g(n))$ if there exists a constant C such that, for any sufficiently large n ($n \geq n_0$):

$$f(n) \leq C \cdot g(n), \quad \text{for all } n \geq n_0.$$

- **Size of the input:** The input to the graph problems that we will look at is defined by two integers: m, the number of edges in the graph, and n, the number of vertices. The *size* of these two integers is given by the number of bits that it takes to represent them: log(m) and log(n).
  - Polynomial algorithm: An algorithm whose cost function is a polynomial of the size of the input. For example: $f(m,n) = 2mn = O(mn)$.
  - Pseudo-polynomial algorithm: An algorithm whose cost function depends on the numerical size of the input. For example, if U is the maximum capacity in the graph, $f(m,n,U) = U \cdot m = O(U \cdot m)$.

    This is a worse bound than a polynomial cost if U is very large (for example, 2^n), since it might cause the algorithm to run in exponential time in the *size* of the input.
  - Note: Because the flow networks that we will look at are simple graphs:

$$m \leq \frac{n^2 - n}{2} = O(n^2)$$

    And because the graphs are connected, a minimum of $n-1$ edges are always in the graph (See []). Except the case where $m = n - 1$, which is rare and will be explicitly stated if it happens, we will assume that $m \geq n$ and use it in our bounds. For example, $f(m,n) = m + n = O(m)$.
- **Sparse and dense graphs:** If in a certain kind of graph, as the graph gets bigger $m = O(n)$, we say that the graph is *sparse*. Otherwise, if $m = O(n^2)$ as the graph gets bigger, then we say that it is *dense*.
- **Logarithms:** All logarithms will be in base 2, unless explicitly noted.

3 Augmenting path algorithms

The first algorithms that were designed to solve the maximum flow problem rely on finding what are called ”Augmenting paths” in the network. In this section we will explain the most significant ones from a theoretical perspective, and in section 3 we will see their implementation in detail. The augmenting path description is based on Frank Nielsen’s book [16].

3.1 Augmenting paths

An augmenting path P is a weakly directed $(s \to t)$-path where the edges that are directed from s to t are not saturated, $f_e < c_e$, and the edges that are directed backward carry some positive flow, $f_e > 0$. The existence of an augmenting path in the graph implies that the current flow in the graph is not a maximum flow, because we can increase it along the path with the following operations:

1. Let δ be the minimum of the remaining capacities of the forward edges in the path, min$\{c_e - f_e \mid e \texttt{ is a forward edge}\}$, and the flow of the backward edges: min$\{f_e \mid e \texttt{ is a backwards edge}\}$

2. For each forward edge in P, increase the flow by δ. The capacity constraint is satisfied because, for each forward edge e, $f_e + \delta \leq c_e$.

3. For each backwards edge in P, decrease the flow by δ. The non-negative flow constraint is satisfied because $\delta \leq f_e$ for each backwards edge.

We can see that the conservation constraint is easily satisfied if the path consists of forward edges: the increment in incoming flow into a vertex is compensated by the increment in outgoing flow. Backward edges also work, but it is less obvious.

Let $e_k = (y, x)$ be a backwards edge with positive flow in a section of an augmenting path $P = e_{k-1} x e_k y e_{k+1}$. Let’s look at the conservation constraint formula for x:

$$\sum_{e \text{ into } x} f_e = \sum_{e \text{ out of } x} f_e$$

Since x is the end vertex of k, decreasing the flow in e_k causes the incoming flow in the left hand side to decrease. This decrease can be compensated by an increase in incoming flow (left hand side) or a decrease in the outgoing flow (right hand side). If e_{k-1} is a forward edge, the incoming flow will increase, and if it is a backwards edge, the outgoing flow will decrease. In both cases, the equality holds.

The argument for y is very similar: the outgoing flow decreases, but it is compensated by an increase in the outgoing flow if e_{k+1} is a forward edge, or by a decrease in incoming flow if e_{k+1} is a backwards edge. A picture of an augmenting path should clear all the doubts:

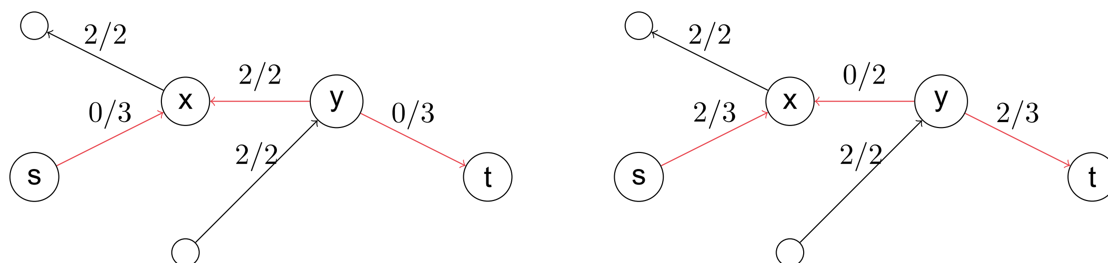


Figure 3.1: An augmenting path with a backwards edge before and after we have sent flow through it.

3.2 Residual Graph G_f

A concept very closely related to augmenting paths, and that will be used in the augmenting path algorithms, is the residual graph of G with respect to a flow f, G_f. To build this graph:

- G_f has the same vertices as G.
- For each edge $e = (x, y) \in E(G)$ of capacity c_e that has zero flow, $f_e = 0$, there's an edge $(x, y) \in E(G_f)$ of capacity c_e.
- For each edge $e = (x, y) \in E(G)$ of capacity c_e that has flow below the capacity, $0 < f_e < c_e$, there are two edges in G_f: One edge (x, y) with capacity $c_e - f_e$, that represents a possible forward edge in an augmenting path, and another edge (y, x) of capacity $c_e = f_e$, that represents a possible backwards edge.
- For each saturated edge $e = (x, y) \in E(G)$, $f_e = c_e$, there's only one edge (y, x) of capacity c_e.

The interest of the residual graph is that, whenever there is an augmenting $(s \to t)$-path of value δ in G, there is a normal, strongly directed $(s \to t)$-path P in G_f where

$$\delta = \text{min}\{c_e \mid e \in P\}.$$

This follows from the fact that all the forward, non-saturated edges, are in G_f with the capacity that would be used in an augmenting path, $c_e - f_e$. Furthermore, every time that there is an edge $e = (x, y)$ with flow f_e, a backwards edge $e' = (y, x)$ is in G_f with capacity $c_{e'} = f_e$ to be used in an augmenting path.

Another way to understand it is to consider $c_e - f_e$ as the *remaining capacity* of an edge e, and note that an edge $(u, v) \in E(G)$ is in G_f too if the remaining capacity of (u, v) is greater than zero, or if an edge (v, u) has positive flow.

In other words, the capacity c_e of an edge $e \in G_f$ denotes the amount of flow that can be pushed into that edge without violating the capacity constraint.

3.3 Ford-Fulkerson algorithm

The first algorithm that used augmenting paths was developed by Ford and Fulkerson [13] and bears their name. It starts with the zero flow and aims to increase the flow through

augmenting paths, until none are left.

Algorithm 1: Ford-Fulkerson algorithm

Result: A maximum flow f^*

1 Let f be the zero flow ($f_e = 0$ for all $e \in E$);
2 **while** *There exists an augmenting $(s \to t)$-path P with value δ* **do**
3   Send δ units of flow through P.
4 **end**

And that is it. A skeptical reader might have come up with a reasonable doubt: we have shown that augmenting paths increase the flow. But, when no more augmenting paths are left, can we be sure that we have reached the maximum flow in the network? Can there be another sophisticated method to increase the flow even further?

The answer is no, and a beautiful theorem called the *max-flow-min-cut* theorem can give us the reason. The following proofs are based on the ones in Bondy-Murty [15].

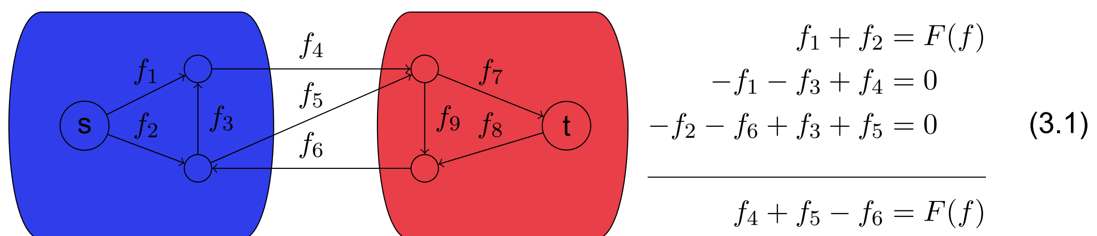
Theorem 1.1, max-flow-min-cut: In a flow network, the value of a maximum flow is always the same as the capacity of a minimum cut: $F(f^*) = \text{cap } K$.

The first part of the proof of this theorem is showing that the following statements hold:

- **Lemma 1.2:** The value of the flow across an $s - t$ cut $K = (S, \overline{S})$ equals the value of the flow $F(f)$.
- **Lemma 1.3:** The capacity of a cut is always bigger or equal than the value of the flow: $F(f) \leq \text{cap } K$.
- **Lemma 1.4:** Equality holds ($F(f) = \text{cap } K$) if all the edges of a minimum cut are saturated, and the edges that go from $\overline{S}$ to S have zero flow.

And the other part of the proof is arguing that, in any network, when there is a maximum flow of value $F(f)$, there is always a minimum cut of the same capacity. We will show this part using the Ford-Fulkerson method.

Proof of Lemma 1.2: It is best shown with an example. Take the conservation equations of the vertices in S, with the outgoing flows in the right hand side and the incoming flows in the right hand side too (with a negative sign), and add them:



When we add all the equations for vertices in S, all the flow that is generated and absorbed in vertices in S cancels. Because there's an excess of flow going out of S, this excess is transported to T, where the sink absorbs it. The flow might go back and forth, and so we have to subtract the flow that goes back from T to S again. This happens for any s-t-cut in the network.

Proof of Lemma 1.3: For every edge in a network, $f_e \leq c_e$. Therefore, for the edges of a cut $K = (S, \overline{S})$:

$$\sum_{e \in K} f_e \leq \sum_{e \in K} c_e = \text{cap } K.$$

The edges in the cut exclude the ones going from $\overline{S}$ to S. If we count those too, the value of the sum of the flows across the cut will be lower, since those flows are subtracted. Lemma 1.2 tells us that if we count the flow of the back edges too, we have the value of the flow in the network. Therefore,

$$F(f) \leq \sum_{e \in K} f_e$$

and

$$F(f) \leq \text{cap } K.$$

Proof of Lemma 1.4: If all the edges going from S to $\overline{S}$ are saturated ($f_e = c_e$) and the ones going in the other direction have flow 0:

$$F(f) = \sum_{e \in (S,\overline{S})} f_e + \sum_{e \in (\overline{S},S)} f_e = \sum_{e \in (S,\overline{S})} f_e + 0$$
$$= \sum_{e \in (S,\overline{S})} c_e = \text{cap } K$$

This will be a maximum flow, since Lemma 1.3 states that the value of the flow cannot be higher than the capacity of a cut.

Now, we will show how the Ford-Fulkerson's method ensures that the maximum flow value always equals the capacity of a minimum cut.

Theorem 1.5: If there are no more augmenting $(s \to t)$-paths in the network, then the flow value equals the capacity of a minimum cut.

Proof: Let's take a look at the set of vertices, call it R, that are reachable from the source with augmenting paths. t is not one of them, because there are no more augmenting $(s \to t)$-paths. Let $\overline{R}$ be $V \setminus R$. Any forward edge $e = (x, y)$ that goes from R to $\overline{R}$ has to be saturated, or else, there would be an augmenting path from s to y, and y wouldn't be in $\overline{R}$ by definition. But also any backward edge $e = (y, x)$ from $\overline{R}$ to R has zero flow, because otherwise there would be an augmenting path from s to y that has e as a backwards edge.

But now, $K = (R, \overline{R})$ is a cut in the network where all the edges in the cut are saturated, and all the edges from $\overline{R}$ to R have zero flow. Thus, Lemma 1.4 shows that $F(f) =$ cap K and $F(f)$ is the maximum flow value possible.

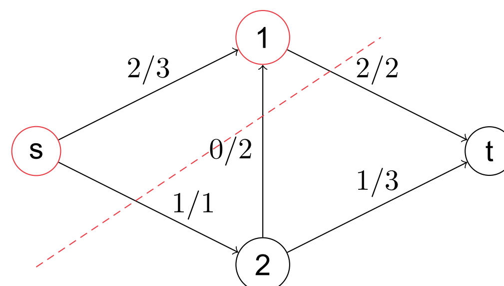


Figure 3.2: A minimum cut and a maximum flow. $R = \{s, 1\}$ is the set of vertices reachable from s with augmenting paths.

3.3.1 Complexity

Right now, we are not ready yet to analyse the computational complexity of Ford-Fulkerson's algorithm, because we have not specified how are we going to find the augmenting paths. It can be shown that it is possible to modify a graph search algorithm, like BFS or DFS (that we will discuss later), to find augmenting paths if we guide it through edges in G_f that have a remaining capacity greater than 0. This can be done in $O(m)$ time. Then, we will send the correct flow through the edges of the path. Because an $(s \to t)$-path has all the vertices distinct and there are n of them in the graph, at most we will have to change $n-1$ edges for each path. In total, for each augmenting path we use $O(m+n) = O(m)$ operations.

How many times do we have to send flow through an augmenting path? At least, each time we find an augmenting path we will increase the flow by 1, until we reach the value of the maximum flow, that we will call $F(f^*)$. Thus, the complexity is $O(F(f^*)m)$.

This is a quite bad bound, because the number of times that we have to scan through the edges is, at most, the numerical value of the maximum flow. If this value is very high, say a million, we would have to scan the edges at most a million times. The algorithm is pseudo-polynomial in this form. Fortunately, Edmonds and Karp [6] showed that we can do better than this.

3.4 Edmonds-Karp algorithm

The idea behind the Edmonds-Karp algorithm is to send flow through the shortest augmenting $(s \to t)$-path available at each step. They showed that, this way, the number of times that we will find augmenting $(s \to t)$-paths is bounded by m and n, and doesn't depend on the capacities of the edges. The algorithm is otherwise the same as the Ford-Fulkerson algorithm.

Algorithm 2: Edmonds-Karp algorithm

Result: A maximum flow $F(f^*)$

```
1 Let f be the zero flow (f_e = 0 for all e ∈ E);
2 while There exist augmenting (s → t)-paths do
3 |   Get a shortest augmenting (s → t)-path using BFS;
4 |   Send δ units of flow through P.
5 end
```

3.4.1 BFS: Finding shortest paths

To find the shortest augmenting $(s \to t)$-path in each step, we will find the shortest path in the residual graph G_f. We will perform what is called a *Breadth-First Search* from s to find the shortest path to t. A BFS looks at all the vertices adjacent to s and stores them in a queue Q. If it has not found t, it then looks at all the vertices adjacent to the ones in Q, ignoring the ones that it has already seen. It repeats this process in levels. It discovers all the vertices at a distance 1 from s, then all vertices at distance 2, etc., until it finds t. Then, it can backtrack to see the path that lead to t. More about this process on the implementation section, X.

As for the complexity of a BFS, it will scan in the worst case all the edges, m, and then it will backtrack along a path of length at most $n-1$. The complexity is then $O(m+n) = O(m)$.

3.4.2 Edmonds-Karp theorem

The importance of Edmonds-Karp algorithm is that it provides the first strongly polynomial bound for the maximum flow problem. The following proof of Edmonds-Karp's theorem is

based on the one on Frank Nielsen's book [16].

Theorem 2 (Edmonds-Karp): If we follow Ford-Fulkerson's algorithm, but choosing every time the shortest augmenting path available, then the algorithm will stop after having used, at most, $m \cdot n$ augmenting paths.

Proof: Let $P_1, ..., P_k$ be the augmenting $(s \to t)$-paths that the algorithm will find, in order.

Claim 1: $|P_i| \leq |P_{i+1}|$. To see this, consider an augmenting path P_i in the residual graph G_f and the flow that we can send through it, δ. As we have seen, δ is the minimum of the capacities of the edges in P_i in G_f. When we send flow through the path, at least one edge will be saturated, the edge with the minimum capacity in P_i. Let $e = (x, y)$ be this edge. Note that, because e lies on a shortest path, $dist_{G_f}(s, y) = dist_{G_f}(s, x) + 1$, and $|P_i| = dist_{G_f}(s, x) + 1 + dist_{G_f}(y, t)$.

When we saturate that edge, e is removed from the residual graph, and the reverse edge (y, x) is created. For this edge to be used again, there shall exist a $(s \to y)$-path in G_f and another $(x \to t)$-path in G_f. But, because $dist_{G_f}(s, y) > dist_{G_f}(s, x)$ and $dist_{G_f}(x, t) > dist_{G_f}(y, t)$, a path P' that contains (y, x) will be larger than P_i by 2 at least.

Now, look at a sequence of consecutive chosen paths by the algorithm $P_i, \ldots, P_j$ that have the same length. Since each of them saturates an edge that the next path with the same length cannot contain, all of these paths are different by at least one edge. There are m edges, so at most there are m different paths of the same length.

The length of the paths is bounded by the maximum length of an $(s \to t)$-path, that is $n - 1$ because there are n different vertices in the path. Therefore, the maximum amount of paths that we can find is $m \cdot n$.

Theorem 2.1: The complexity of Edmonds-Karp algorithm is $O(m^2 n)$.

Proof: If it takes $O(m)$ steps to find a shortest augmenting path, and there are at most mn augmenting paths, so the total complexity is $O(m^2 n)$.

3.5 Scaling Edmonds-Karp algorithm

There is a variant of the Edmonds-Karp algorithm that can improve the runtime in certain bad case scenarios. Its main idea is to divide the computation in *phases*, where each phase only considers the edges that have a remaining capacity bigger than some threshold. The explanation follows the lecture notes on *Algorithms and Data Structures 2* by Gørtz and Billie [9].

In this algorithm, the threshold is denoted by Δ, and starts as the biggest power of 2 that is less than or equal to the largest capacity out of s. This threshold is halved when the

phase ends, until the threshold is lowered to 1 and all augmenting paths can be used.

Algorithm 3: Scaling Edmonds-Karp algorithm

Result: A maximum flow $F(f^*)$

1 Let f be the zero flow ($f_e = 0$ for all $e \in E$);
2 Let C be the maximum capacity out of s;
3 Let $\Delta = \max\{2^x | 2^x \leq C\}$;
4 **while** $\Delta >= 1$ **do**
5   **while** *There exist augmenting* $(s \to t)$*-paths* P *where each* $e \in P$ *has capacity* $c_e \geq \Delta$ **do**
6   Get a shortest augmenting $(s \to t)$-path P using BFS;
7   Send δ units of flow through P.
8   **end**
9     $\Delta \leftarrow \Delta/2$;
10 **end**

3.5.1 Complexity

In scaling algorithms, the theoretical complexity depends on the capacities of the edges, in this case, on the largest capacity out of s, denoted C. This leads to a pseudo-polynomial algorithm. However, the bound that we will derive is rather loose, and we should expect it to perform better than the standard Edmonds-Karp in most cases.

Theorem 3: The complexity of the Scaling Edmonds-Karp algorithm is $O(m^2 \log C)$. Three lemmas will be used together to prove it:

Lemma 3.1: The number of scaling phases is $1 + \lfloor \log C \rfloor$.

The scaling phases go in order from $2^{\lfloor \log C \rfloor}$ to $2^2, 2^1, 2^0$. There are $1 + \lfloor \log C \rfloor$ of them.

Lemma 3.2: Let f be the flow when the Δ-scaling phase ends. Then, $F(f^*) < F(f) + m \cdot \Delta$.

To show this, let's look at the cut formed by the vertices that are reachable by augmenting paths with the Δ condition. Note that the forward edges will have a remaining capacity less than Δ, and the backward edges will have flow that is less than Δ:

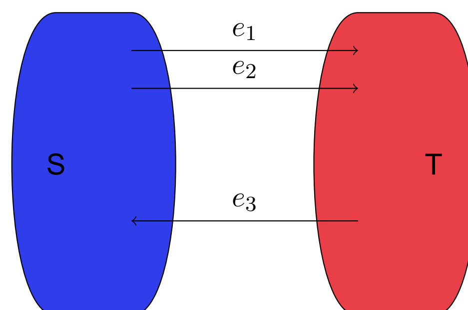


Figure 3.3: s-t cut at the end of phase Δ

$$\text{cap }(S,T) = c_{e_1} + c_{e_2}$$
$$F(f) = f_{e_1} + f_{e_2} - f_{e_3}$$
$$\text{cap }(S,T) - F(f) = c_{e_1} + c_{e_2} - f_{e_1} - f_{e_2} + f_{e_3} =$$
$$= c_{e_1} - f_{e_1} + c_{e_2} - f_{e_2} + f_{e_3}$$
$$(c_{e_1} - f_{e_1}) + (c_{e_2} - f_{e_2}) + f_{e_3} < \Delta + \Delta + \Delta = 3\Delta$$
$$\Rightarrow \text{cap }(S,T) - F(f) < m \cdot \Delta$$
$$\Rightarrow \text{cap }(S,T) < F(f) + m \cdot \Delta$$
$$\text{cap }(S,T) \geq F(f^*) \text{ always by Lemma 1.3:}$$
$$\Rightarrow F(f^*) \leq \text{cap }(S,T) < F(f) + m \cdot \Delta.$$

Lemma 3.3: The number of augmentations in each phase is, at most, $2m$.

We will prove it by induction. First, note that any augmentation in phase Δ will increase the flow by at least Δ, since all edges considered in that phase in the residual graph G_f have a capacity of at least Δ.

First phase: $\Delta = \max \{2^x | 2^x \leq C\}$. The claim is that we can use each edge that goes out of s only once in this phase.

Let's prove it by contradiction: suppose that we use one edge out of s, e, and the remaining capacity is greater or equal to Δ, so that it can be used again. Then:

$$c_e - \Delta \geq \Delta \Rightarrow c_e \geq 2\Delta.$$

But this can't happen, because then there exists a power of two greater than Δ and less than the greatest capacity out of s, namely 2Δ, which contradicts the choice of Δ. We have an upper bound on the number of augmentations of the first phase: m.

Arbitrary phase: Let f be the flow from the previous phase, where we used $\Delta' = 2\Delta$. Then, we apply Lemma 2:

$$F(f^*) < F(f) + m\Delta' \text{ (Scaling phase ended)}$$
$$F(f^*) < F(f) + 2m\Delta$$
$$\Rightarrow F(f^*) - F(f) < 2m\Delta$$

The "leftover" flow in this phase is less than $2m\Delta$, and since each augmentation increases the flow by Δ, we have less than $2m$ augmentations in this phase. This completes the proof of Lemma 3.

Then, it follows that there will be, at most, $O(m \cdot \log C)$ augmentations. If we perform a BFS to find each one, the complexity becomes $O(m^2 \log C)$.

3.6 Dinic's algorithm

Dinic's algorithm is a big improvement of the Edmonds-Karp algorithm, that has its insight in the Edmonds-Karp theorem. It was first developed by Dinitz [5]. The video by William Fiset [19] provides a very accessible explanation. We will describe it informally:

If we always follow the shortest augmenting paths, the ones with the same length l are used consecutively (Claim 1), and when an edge of them is saturated, it will not be used

again until all augmenting paths of that length are used. Therefore, we do not really need another BFS for finding the edges that form augmenting paths of length l, because all of them have already been discovered.

However, we do need another way of finding the $(s \to t)$-paths through those edges. But for that we will use a more efficient search method, the *Depth-First Search* or DFS. DFS does not necessarily find shortest paths, but if we guide it through the edges that we have discovered with a BFS, the paths will be short necessarily. This way, only one BFS is enough to discover all paths of the same length.

Algorithm 4: Dinic's algorithm

Result: A maximum flow $F(f^*)$

1 Let f be the zero flow ($f_e = 0$ for all $e \in E$);
2 **while** *There exist augmenting* $(s \to t)$*-paths* **do**
3   Perform a BFS search that stores the distance from s to each vertex;
4   Let l be the distance of t;
5   **while** *There exist augmenting* $(s \to t)$*-paths of length* l **do**
6   Get an augmenting path P with a DFS search using the vertices at distance l or less;
7   Send δ units of flow through P.
8   **end**
9 **end**

3.6.1 Depth-First Search

The DFS searches can be implemented in such a way that that we can find all the paths of a given length in $O(mn)$ time. To achieve this we will need an implementation that lets us scan sequentially the outgoing edges from each vertex u, and store a pointer to one of them. We will call this pointer $next(u)$, and at the beginning we will set all the pointers to the first edge of the sequence. We will only use edges that were found by the previous BFS search, that is edges that go from vertices with distance i to distance $i + 1$. We can also use this to avoid cycles.

To perform this search, we do a recursive strategy to find the edges of an augmenting path P. Starting from s, and recursively from any vertex u, we look at the endpoint vertex v of the edge $next(u)$. If $v = t$, we have found an $(s \to t)$-path and we add the edge $(u \to t)$ to the end of the augmenting path P. The call returns. If $v \neq t$, we call this same procedure on v.

If the procedure returns because it has found t, we add the edge $(u \to v)$ to the start of P. Otherwise, if the procedure returns without having found t, we ignore the $(u \to v)$ edge in future DFS searches by changing $next(u)$ to point at the next edge, and we continue recursing on the second adjacent vertex to u. We repeat the process until we have explored all the adjacent vertices to u. If all return without having found the sink, this call returns too.

3.6.2 Complexity

The key aspect to the DFS bound is that, whenever we find what is called a *dead end* (a path that does not lead to t), the vertices of the path are ignored by changing the pointer $next(v)$. Each time that a DFS search is performed, we ignore up to n vertices, and backtrack through a path of length at most n, therefore its complexity is $O(n)$.

As we explained before, when an augmenting path is used and an edge becomes saturated, it will not be used again until the next BFS. Therefore, we have to find up to m

augmenting paths with DFS searches between each BFS. The number of steps between each BFS is then $O(nm)$.

It is worth noting that these bounds represent extreme cases. If an augmenting $(s \to t)$-path of length $n-1$ were to exist, then there would be no possibility of finding dead ends in the graph (there would simply be no more vertices outside the path).

If the DFS search is implemented correctly, only $n-1$ BFS searches are needed to find all augmenting paths of all lengths. Therefore, the complexity of the whole algorithm lowers to $O(n(m+nm)) = O(n^2m)$.

4 Implementation of Augmenting Paths

In this chapter we will explain the details and design decisions that can be taken to implement the algorithms that we analysed in the last chapter. In the repository of the project there is a Java implementation of the algorithms following the guidelines that we will explain here. We will use pseudo-code to write the algorithms as in the previous section, but using a style closer to Java wherever it is necessary to get closer to the code.

4.1 Graph implementation

There are a few different ways to implement a graph structure in a computer program. We will use the traditional adjacency list implementation.

To represent the graph, we will have an array of size n, where each entry represents a vertex. The "name" of the vertex will be its index in the array. Each entry i will contain what is called an *adjacency list*, a list that contains the information of the edges incident from the vertex i. The array of adjacency lists will be denoted in the pseudo-code as $G_f.array$, and the adjacency list of vertex i as $G_f.array[i]$, of type List<Edge>.

Since we are trying to solve flow problems, we want to include in our representation the flow and capacity of an edge, and represent the residual graph, since it is easier to do searches in it. This complicates a bit the structure. The choice of the author was the following:

- For each edge of the graph, a residual edge backwards is always added, even if the edge has no flow. This is done because it is easier to ignore an edge with remaining capacity 0 than to add and to delete a residual edge dynamically. We should note that any runtime or space cost derived from this will only increase the total cost by a factor of 2 at most, oblivious to any asymptotic analysis.
- The information stored by each edge e, in an object of type Edge, is:
  - The end vertex index (endpoint): $e.v_{out}$.
  - Capacity of the edge: $e.capacity$.
  - Flow of the edge: $e.flow$.
  - Link to the correspondent backwards edge: $e.backwards$.

  Note that, although an edge only stores the end vertex by itself, it always has a link to the backwards edge, that points to the start vertex.
- The capacity of all backward edges is set to 0, and the flow values are stored in negative. This way we can distinguish a real edge from a backwards edge, that comes in handy sometimes. Furthermore, the *remaining capacity* of any edge can be calculated now as *capacity* - *flow*, and it will always be non-negative.

This way, we can represent the residual graph G_f of any network, and use any search method on it to find augmenting paths.

To identify each vertex in the algorithms we will use its adjacency list index in the array, that goes from 0 to n-1.

4.2 Edmonds-Karp implementation

Since the Edmonds-Karp algorithm can be seen as a particular implementation of Ford-Fulkerson's algorithm, we will use this section as an explanation for Ford-Fulkerson's algorithm too.

The Edmonds-Karp algorithm relies on BFS searches in the residual graph to find augmenting paths. It runs a while loop that keeps on finding paths with BFS in G_f until there are none left. After finding a shortest $(s \to t)$-path in G_f, the algorithm iterates through it to find the maximum possible flow δ that we can send through it, and iterates again to increase the flow of the edges of the path with δ. After this step is done, it goes back to the beginning of the while loop to find the next $(s \to t)$-path.

The only modification of the standard BFS search that we need is to check that every edge that we explore has a remaining capacity greater than 0. In our implementation of G_f we are keeping edges that do not have any remaining capacity, and these should be ignored by the algorithm.

4.2.1 Early stop

Because we are only interested in finding one $(s \to t)$-path in each BFS, we can stop the search procedure once we have found t, instead of letting it run until the queue is empty.

4.2.2 Fast queue

Usually we implement queues with linked lists, because they do not require a resize if we add many elements, as an array would. However, in our case we do know how many items will be put in the queue at maximum: in each BFS, we can visit at most n vertices. Therefore, we can implement the queue with an array of size n and with two pointers, one for the starting element of the queue and one for the end. The array implementation should be faster in practice than the linked list implementation.

4.3 Scaling Edmonds-Karp implementation

To implement the scaling Edmonds-Karp variant, we just need to modify line 9 of the BFS algorithm so that we can restrict ourselves to edges with more than Δ capacity. Then, all that is left is to wrap the Edmonds-Karp algorithm in a loop that starts with $\Delta = \max\{2^x | 2^x \leq C\}$, and halve it in every iteration, as explained in the theoretical section.

4.4 Dinic's algorithm implementation

To implement Dinic's algorithm, we need to perform BFS searches in the residual graph in order to find the shortest augmenting $(s \to t)$-paths, and then use DFS searches to send flow through them.

4.4.1 BFS

The only purpose now of the BFS is to find which vertices are part of shortest paths, and store this information so that the next DFS searches can go through them. The working principle is the same as before, but the information that we store will be different.

We will create an integer array $level[]$, that will store the distance of each vertex to the source. $level[s]$ is set to 0 at the beginning, and all the others to -1. We can use this array to check for visited vertices too. Whenever we find an adjacent vertex v to a vertex u, we can set the level of v to the level of u plus 1, if it had not been set before (if $level[v] == -1$). Furthermore, we do not need to store any backtrack information now, since the DFS will be in charge of finding the augmenting paths and sending flow through them.

```
Algorithm 5: Full Edmonds-Karp algorithm
Input: A residual graph G_f and s,t ∈ V
Result: An augmenting (s → t)-path in G_f
1  Let f be the zero flow (f_e = 0 for all e ∈ E);
2  boolean[] visited;
3  Backtrack[] back; //backtrack objects;
4  while BFS finds an augmenting (s → t)-path P do
5      //Find δ by backtracking from t;
6      δ =Integer.MAX_VALUE;
7      v = t;
8      while v ≠ s do
9          Backtrack b = back[v];
10         Edge e = back.edge;
11         δ = min{δ, e.capacity};
12         v = back.v_in;
13     end
14     //We have the bottleneck δ. Now, update the path;
15     v = t;
16     while v ≠ s do
17         Backtrack b = back[v];
18         Edge e = back.edge;
19         e.flow += δ;
20         e.backwards.flow -= δ;
21         v = back.v_in;
22     end
23 end
```

```
Algorithm 6: BFS algorithm for EdmondsKarp
Input: A residual graph G_f, its flow f, and s,t ∈ V
Result: An augmenting (s → t)-path in G_f, or false if there is none
1  Queue q;
2  initialize visited[n] to false;
3  visited[s] = true;
4  q ← s;
5  while q is not empty do
6      v ← q;
7      List<Edge> adj = G_f.array[v];
8      foreach edge e in adj do
9          if (remaining capacity of e ≥ 0) and (not visited[e.v_out]) then
10             Backtrack b = new Backtrack(v, e);
11             back[e.v_out] = b;
12             visited[e.v_out] = true;
13             if e.v_out == t then
14                 stop;
15             q ← e.v_out;
16     end
17 end
```

Early stop: An immediate improvement that we can implement is not to explore vertices that are at the distance of t or further. All those vertices would be part of augmenting paths of a greater length than the shortest one available, and we should use them later. A *max distance* value will be maintained, that is first set to the maximum distance possible, and then it is set to the distance of t once that is found (the level of t). All vertices in the queue that are at the same distance to the source as t will be ignored afterwards.

However, we cannot stop the algorithm once we have found t as we did before, since there may be several shortest augmenting paths to be found after the first one.

Algorithm 7: BFS for Dinic's algorithm

Input: A residual graph G_f, its flow f, and $s, t \in V$
Result: The array $level$ filled with the distance of each vertex to s

1 Queue q;
2 initialize $level[]$ to -1;
3 $level[s]$ = 0;
4 $max_distance$ = Integer.MAX_VALUE;
5 $q \leftarrow s$;
6 **while** *q is not empty* **do**
7 | $v \leftarrow q$;
8 | **if** $level[v] \geq max_distance$ **then**
9 | | continue;
10 | List<Edge> $adj = G_f.array[v]$;
11 | **foreach** *edge* e *in* adj **do**
12 | | **if** $($*remaining capacity of e* $\geq 0)$ *and* $(level[v_{out}] \neq -1)$ **then**
13 | | | $level[e.v_{out}]$ = $level[v] + 1$;
14 | | | $q \leftarrow e.v_{out}$;
15 | **end**
16 | **if** $e.v_{out} == t$ **then**
17 | | $max_distance = level[t]$;
18 **end**

4.4.2 DFS

DFS finds an $(s \to t)$-path with recursive calls, as explained in the theory section X. To store the pointers to the next edge we will use an array $next[]$, initialized to 0, that will indicate the index of the next edge of a vertex in its adjacency list. This edge should have remaining capacity greater than zero, as in the BFS search. Not only that, it should also make sure that it is on a shortest path. For that, we will check that the end vertex is at a level higher than the start vertex. The graph that only contains these valid edges is usually called the *level graph*.

If the DFS finds a dead end, that is a path that does not lead to t, it should backtrack and increase the $next$ pointer of the last vertex that it used, so that the vertex that was a dead end is not considered anymore. If all the vertices become dead ends, then there is no $(s \to t)$-path in the given level graph. Therefore, the next shortest augmenting path will

be longer than the ones found in this level graph, and the next BFS should be performed.

Algorithm 8: Recursive DFS for Dinic's algorithm, *dinicDFS(vertex u, bottleneck)*

Input: A residual graph G_f, its flow f, $level$ and $next$ arrays, previous $bottleneck$ in the path and the vertices $u, t \in V$

Result: The bottleneck of a $(u \rightarrow t)$-path P if there exists one, and the augmented flow in P

1 **if** $u == t$ **then**
2 │ return $bottleneck$;
3 List<Edge> $adj = G_f.array[u]$;
4 int $forward_flow = 0$;
5 **while** $next[u] \leq adj.size()$ *and* $forward_flow = 0$ **do**
6 │ Edge $e = adj.get(next[u])$;
7 │ **if** (*remaining capacity of e* ≥ 0) *and* ($level$[$e.v_{out}$] = $level[u] + 1$) **then**
8 │ │ $forward_flow = dinicDFS(e.v_{out}, \min\{bottleneck, c_e - f_e\})$;
9 │ │ **if** $forward_flow > 0$ **then**
10 │ │ │ $e.flow$ **+=** $forward_flow$;
11 │ │ │ $e.backwards.flow$ **-=** $forward_flow$;
12 │ │ │ **if** $e.flow == e.capacity$ **then**
13 │ │ │ │ //the edge is saturated;
14 │ │ │ │ $next[u]$**++**;
15 │ │ **else**
16 │ │ │ //v is a dead end;
17 │ │ │ $next[u]$**++**;
18 │ │ **end**
19 │ **else**
20 │ │ //Need to find the next edge;
21 │ │ $next[u]$**++**;
22 │ **end**
23 **end**
24 //If no path was found, we return the default value 0;
25 return $forward_flow$;

Algorithm 9: Dinic's full algorithm

Result: A maximum flow $F(f^*)$

1 int[] $level$;
2 int[] $next$;
3 int $max_flow = 0$;
4 Let f be the zero flow ($f_e = 0$ for all $e \in E$);
5 **while** *BFS fills level[] and* $level[t] \neq -1$ **do**
6 │ Let l be the distance of t;
7 │ initialize $next[]$ to 0;
8 │ int $flow =$Integer.MAX_VALUE;
9 │ **while** $flow > 0$ **do**
10 │ │ $flow = dinicDFS(s, Integer.MAX_VALUE)$;
11 │ │ max_flow **+=** $flow$;
12 │ **end**
13 **end**
14 return max_flow;

---

5 Preflow-push algorithms

5.1 Introduction

All the algorithms that we have described until now make use of augmenting paths to find what flow we can send through which edges. The working principle was to start with the zero flow, and increase the flow all that is possible while maintaining at all times the flow constraints: non-negativity, capacity and conservation.

But this is not the only way of solving the maximum flow problem. There is an entirely different paradigm that aims to start with a condition that guarantees the optimality of the solution, violating the conservation constraint, and works towards satisfying the conservation constraint. This is the **preflow-push** paradigm, pioneered by Andrew V. Goldberg and presented by him and Robert Tarjan in *A New Approach to the Maximum-Flow problem* [8]. It is based on Karzanov's idea of a preflow [11]. In theory and implementation it is more complicated than the augmenting paths methods, but it ultimately provides a better performance in all types of networks with the most sophisticated algorithms.

We will first describe what is a preflow, then the distance labelling that guarantees an optimal solution, and then we will present the different algorithms to reach a valid maximum flow. We will show their correctness and their worst-case computational complexity. The chapter follows the original paper [8], as well as the introduction given in *Algorithm Design* by Kleinberg and Tardos [12] and the insight of Tim Roughgarden's lectures [18].

5.2 Preflows

A *preflow* is a weaker version of a flow, that changes the conservation constraint to allow more flow to go into a vertex than the flow that comes out. The other properties are the same as the properties of a flow. This also leads to the concept of *excess* of a vertex:

- **Preflow conservation constraint:** The incoming flow into a vertex $v \neq s, t$ is always greater or equal to the outgoing flow:

$$\sum_{e \text{ into } v} f_e \geq \sum_{e \text{ out of } v} f_e$$

- **Excess of a vertex:** The excess of a vertex $v \neq s, t$, $\alpha(v)$, is defined as the difference between the incoming flow and the outgoing flow:

$$\alpha(v) = \sum_{e \text{ into } v} f_e - \sum_{e \text{ out of } v} f_e \geq 0$$

We shall note that a preflow is a valid flow when the excess of all the vertices are zero, because then the incoming flow into all vertices equals the outgoing flow. This situation satisfies the conservation constraint of a regular flow. Therefore, a preflow push algorithm will start with a specific initial preflow and work to clear all the excesses by pushing them to the sink, until there are no excesses left.

We will also define the residual graph G_f of a preflow f exactly as the one of a flow.

5.2.1 Distance labelling

The preflow push algorithms make use of a vertex labelling called *distance labelling*. The labelling is intended to provide a lower bound on the distance from any vertex to the sink following an augmenting path, to make sure that the excess of vertices is directed towards the sink. It also provides the necessary condition for the final flow to be optimal.

Valid distance labelling: It is a function from the vertices to the integers:

$$d : V(G) \to \mathbb{Z}_{\geq 0}$$

that satisfies the following constraints:

1. $d(s) = n$ and $d(t) = 0$.
2. For all edges $e = (u, v)$ in the graph, $d(u) \leq d(v) + 1$.

   In other words, v can be at most one step closer to the sink than u, or be the same distance or further away.

Lemma 5.0: If f is a preflow that is compatible with a valid distance labelling d, then there is no augmenting $(s \to t)$-path in the residual graph of the preflow G_f.

Proof: If the constraints of the distance labelling are maintained, for any strongly directed $(v_0 \to v_k)$-path P in G_f, whose vertices are $v_0, v_1, \ldots, v_k \in P$:

$$d(v_i) \leq d(v_{i+1}) + 1, \text{ by constraint number 2.}$$

An $(s \to t)$-path starts with $d(s) = n$ and has to reach $d(t) = 0$. Each time it traverses one edge of the path, $e = (v_i, v_{i+1})$, it can get at most one unit distance closer to the sink. However, the longest path in G_f has n vertices and $n - 1$ edges, so it can go at most $n - 1$ units of distance down. The minimum distance that the path can reach from s is $n - (n - 1) = 1 > 0$. It can never reach t, therefore there cannot exist an $(s \to t)$-path.

Note that this condition applies to any preflow, including a preflow with no excess (a valid flow). If we maintain the constraints of the distance labelling at all times and reach the conditions of a valid flow, then we will have a flow where there is no $(s \to t)$-path. Then, by Theorem 1.5 (explained in the augmenting paths chapter), the flow is maximum.

We will now explain a basic preflow-push algorithm that maintains a valid labelling at all times and terminates with a valid maximum flow. In this explanation it is intuitive to think of the distance labels as *heights*, and to think about the flow in the graph as a network of pipes. The aim is to push all the water that we can from the highest point, s, to the lowest point, t. The second constraint of the distance labelling can be understood as the fact that we cannot have a downwards slope that is too high, and the excess on a vertex can be thought of as a pool of excess water that has to be pushed down through the pipes at some point in the algorithm.

From now on, the words *height* and *distance* of a vertex will mean the same, namely the label of a vertex given by a valid distance labelling. For an edge (u, v) where $d(u) = d(v)+1$ we will say that the edge goes downwards, and otherwise that it is flat ($d(u) = d(v)$) or it goes upwards ($d(v) < d(u)$). For two vertices u and v where $d(u) > d(v)$, we will say that u is "higher up" than v.

5.3 Generic Preflow-Push Algorithm

5.3.1 Initialisation

The initialisation of the algorithm is a bit more involved than in augmenting path algorithms, since we have to start with a preflow that is consistent with a valid labelling. The common

way to do it is to set the heights of s and t to n and 0, respectively, and all the others to 0.

With this setup, the second constraint is violated: all the edges coming out of s have no flow, and they go from distance n to distance 0. Because they have no flow, their remaining capacity is greater than zero, and they are therefore contained in G_f. This violates the second constraint. To fix this, we "flood" the network by saturating those edges. This removes the forward edges in G_f that come out of s, and changes them to backward edges that go from a vertex at height 0 to the source, that is at height n. This satisfies all the constraints.

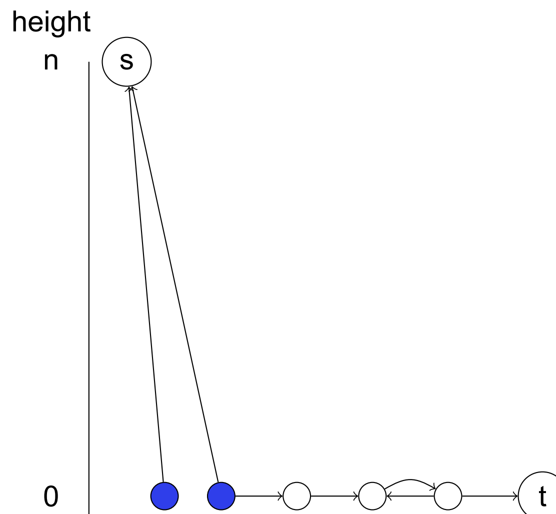


Figure 5.1: Sketch of what the initialisation looks like. The vertices with excess are displayed in blue. The edges are the ones present on G_f.

The vertices adjacent to s now have an excess. We need to push that excess to the sink. It might happen that not all the excess can be pushed to t, and in that case, we will push it back to s. The following algorithm accomplishes that.

5.3.2 Generic Push-Relabel Algorithm

The algorithm consists on applying two basic operations on vertices with excess: $push$ and $relabel$. For the algorithms that are based on these operations, we will use the names *preflow-push* and *push-relabel* indistinctly.

Push operation: On a vertex u with excess $\alpha(u)$, select an edge $e = (u, v) \in G_f$ where $d(u) = d(v) + 1$. Push on that edge the minimum between the excess and the remaining capacity of the edge: min$\{\alpha(u), c_e\}$.

Relabel operation: On a vertex u with excess $\alpha(u)$, increase the label of u from $d(u)$ to $d(u) + 1$.

The push-relabel algorithm is as follows:

Algorithm 10: Generic push-relabel algorithm

Input: A network $G = (V, E)$ with capacities on the edges, c_e for all $e \in E$
Result: The value of the maximum flow $F(f^*)$

```
1  Let d be a valid distance labelling;
2  //Initialisation;
3  d(s) = n and d(v) = 0 for all other vertices;
4  foreach Edge e = (s, v) coming out of s do
5  |  Push c_e units of flow on e;
6  end
7  //Algorithm;
8  while There is a vertex with excess u do
9  |  if there is an edge e = (u, v) with d(u) = d(v) + 1 then
10 |  |  push(u, v);
11 |  else
12 |  |  relabel(u);
13 |  end
14 end
15 //Get the value of the flow;
16 max_flow = 0;
17 foreach Edge e = (s, v) coming out of s do
18 |  max_flow+ = f_e;
19 end
20 return max_flow;
```

5.3.3 Correctness and termination

It requires some insight to understand why this simple algorithm works, why it terminates at all, and how it produces a maximum flow. We will follow the original proofs and lemmas of Goldberg's and Tarjan's paper [8] and the insight of Tim Roughgarden's lectures [18], to try to give an intuitive understanding of the correctness and termination of the algorithm.

Lemma 5.1: The algorithm maintains a valid distance labelling at all times.

Proof: We start with a valid labelling, that we explained in the initialisation section. s and t will never be relabelled, because the excess of a vertex was defined on vertices different than s and t, and therefore they will never be chosen for a relabel operation (the only one that changes labels).

Now, we argue for the other vertices. After the initialization, a vertex with excess is chosen and a push operation is tried first. If there is no applicable edge for it, then a relabel is done. The push operation does not change the labels, but it could create a residual edge that is not compatible with a distance labelling. However, the only edges that we push flow on go downwards, and the residual edge that is created will go in the reverse direction, upwards, so it will always be valid.

If there is no push on the vertex, then there are no edges in G_f going downwards from it. All the edges in G_f incident from u are flat or go upwards. After the relabel operation, all the flat edges will go only downwards by one and the rest will stay upwards or flat. Furthermore, any edges incident with u will continue to be valid: if $d(v) \leq d(u) + 1$, then $d(v) < (d(u) + 1) + 1$.

Both operations always maintain the distance labelling constraints, thus it is maintained

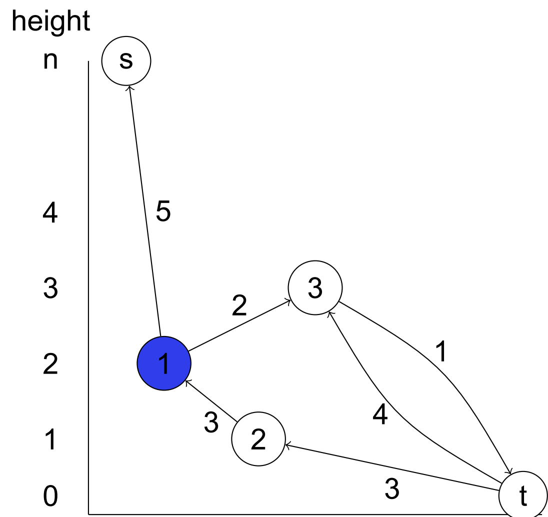


Figure 5.2: Relabel operation. Edge (1, 2) is saturated with 3 units of flow and does not appear in G_f anymore. All edges going downwards from 1 are saturated, therefore it can be relabelled without breaking the constraints, until height 4, where it will be able to push to vertex 3.

at all points.

This implies the following lemma:

Lemma 5.2: If the push-relabel algorithm terminates, it gives a maximum flow as an output.

Proof: The algorithm will only terminate when there are no more vertices with excess. This means that what is left in the graph is a flow. Because at that point the distance labelling constraints are satisfied (Lemma 5.1), Lemma 5.0 tells us that there is no $(s \to t)$-path in f, and that means by Theorem 1.5 that the flow is maximum.

We will now prove that the algorithm will always terminate, thus it will always give a maximum flow. First, we will need a key lemma:

Key Lemma 5.3: If a vertex v has excess, then there is a $(v \to s)$-path in G_f.

Intuitively, if a vertex has an excess, this had to come from s, and it is reasonable to think that it can be pushed back from where it came. Let's present a formal proof, this time from *Algorithm design* [12]:

Proof: Let A be the set of vertices v such that there is a $(v \to s)$-path in G_f, and let $B = V \setminus A$. We shall prove that all vertices with excess are in A. Now, consider the sum of excesses of vertices in B:

$$\sum_{v \in B} \alpha(v) = \sum_{v \in B} \Big[\Big(\sum_{e \text{ into } v} f_e \Big) - \Big(\sum_{e \text{ out of } v} f_e \Big) \Big] \geq 0$$

It is greater or equal to zero because the excesses are always positive. Now, let's analyse the terms of the sum for each edge in G:

- For an edge that has its both ends in B, its flow will be added and subtracted once, therefore it will not contribute to the sum.
- An edge that has its start vertex in A and its end vertex in B cannot carry flow, otherwise its backwards edge in G_f would exist, and the end vertex would belong to A.
- An edge that has its start vertex in B and its end vertex in A will appear in the sum of flows out of a vertex in B. Therefore, its value is subtracted. They are therefore the only edges that contribute to the sum.

$$\sum_{v \in B} \alpha(v) = \sum_{v \in B} \Big[- \Big(\sum_{e \text{ out of } v} f_e \Big) \Big] \geq 0$$

Since each of the excesses are positive or 0, but the sum has to be less than or equal to 0, the only possibility is that the excesses in B are 0. This means that all the excesses are in vertices in A.

The reason why this is a key lemma is because it means that, at any point in the algorithm, any excess can be pushed back to s. In fact, what will happen is that, when the excess that is pushed to the sink is the maximum possible, the distance labels will rise above n and the extra flow will be reverted to the source. The key lemma also allows us to put an upper bound on the height of a vertex:

Lemma 5.4: At any time during the execution of the algorithm, the label of a vertex does not decrease, and it cannot be higher than $2n - 1$.

Proof: For s and t it is trivial, because their labels do not change. The label of any other vertex is only changed when a relabelling happens, and this operation only increases it, so it cannot decrease.

For the upper bound, we will use the Key Lemma 5.3. A vertex is only relabelled when it has excess, and a vertex with excess has always a path to s in the residual graph. Let v be a vertex with excess and let P be the path in G_f that leads to s. This path has at most n vertices and $n - 1$ edges. The highest label that v can have is bounded because P cannot have edges that go downhill by more than 1 unit of distance. It has to reach the height of s, that is n. Therefore, as we reasoned before, the path can only go $n - 1$ units of distance down, and the highest distance possible for v is $n + (n - 1) = 2n - 1$.

This automatically sets an upper bound on the number of relabel operations that the algorithm performs. We will prove that the full algorithm stops by bounding the amount of relabel and push operations that will happen. As we showed with Lemma 5.2, at that point we will have reached a maximum flow. Let us start with bounding the number of relabel operations:

Number of Relabels, Lemma 5.5: The number of relabel operations is, at most, $(2n-1)$ per vertex and thus $(2n - 1)(n - 2) < 2n^2 = O(n^2)$ in total.

Proof: The relabel operations only happen in vertices different from s and t, thus in $n - 2$ different vertices. The labels of these vertices start at 0 and are increased by 1 with each relabel. By Lemma 5.4 they will reach at most the height $2n - 1$, so there will be at most

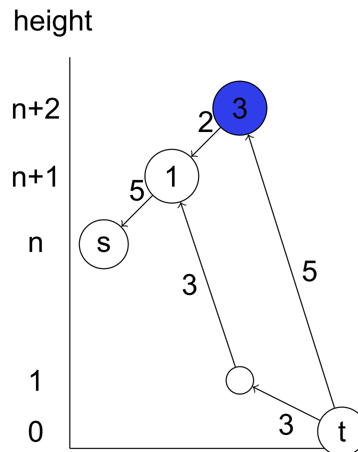


Figure 5.3: Effect of Key Lemma: all vertices with excess have a path to s that satisfies the constraints, and these constraints bound the maximum height. The excess that cannot be pushed to the sink will be pushed back to the source.

$2n-1$ relabel operations per vertex. The amount of operations is therefore $(2n-1)(n-2) < 2n^2 = O(n^2)$.

For bounding the number of pushes, we will distinguish between *saturating* and *non-saturating* pushes:

- **Saturating push:** a push that saturates the edge that it is applied on. It might or might not clear the excess of the vertex.
- **Non-saturating push:** a push that does not saturate the edge that it is applied on. This only happens if the vertex had less excess than the remaining capacity of the edge, and thus the excess of the vertex is cleared entirely.

Number of saturating pushes, Lemma 5.6: The number of saturating pushes is at most $2nm = O(nm)$ in the whole execution.

Proof: Let's analyse the number of saturating pushes in each edge, $e = (u, v)$. When a saturating push happens:

1. $d(u) = d(v) + 1$ by definition of the push operation.
2. After the saturating push, (u, v) disappears from G_f, because it becomes saturated.

The only way to recover the edge (u, v) on G_f is to push back some flow on the backwards edge (v, u), that is always present if (u, v) is saturated. However, a push on (v, u) will never happen until $d(v) = d(u) + 1$, so until at least two relabels on v must happen. Afterwards, another two relabels on u are needed to push back again on (u, v).

Because we have an upper bound on the number of relabels that can happen on u by Lemma 5.4, we have an upper bound on the number of saturating pushes on each edge that comes out of u, namely 1 every 2 relabels: $(2n - 1)/2 < n$. Therefore, for each edge (u, v) in G_f there are less than n saturating pushes, and because there are two edges in G_f for each one in G, there are less than $2n$ saturating pushes per edge of G. In total, $2nm = O(nm)$ saturating pushes in the whole algorithm.

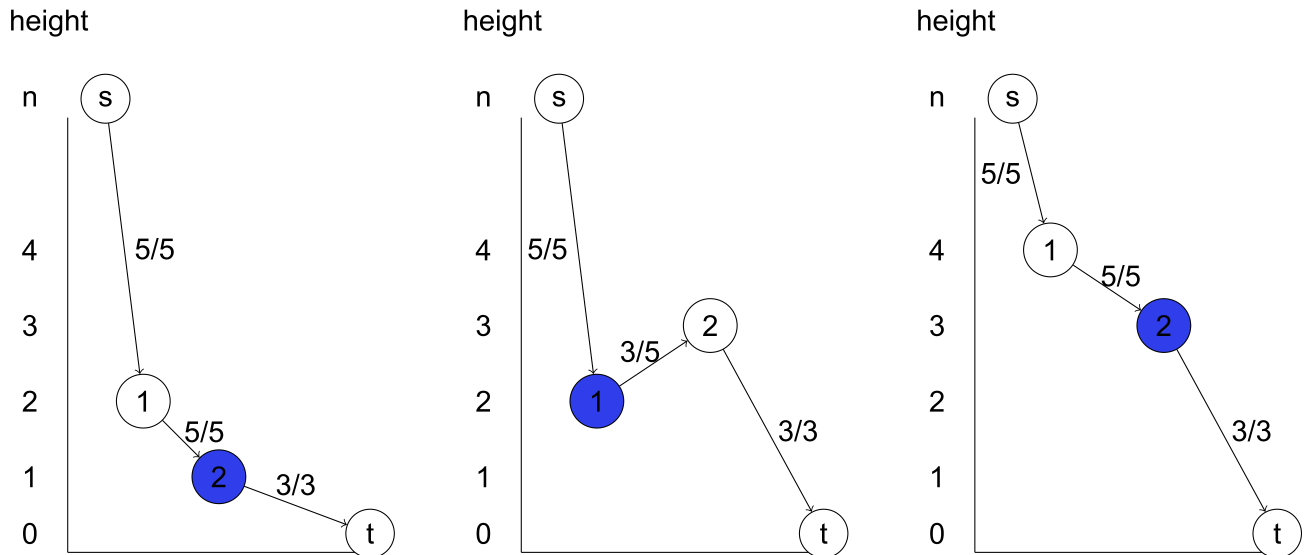


Figure 5.4: Two saturating pushes in the same edge require relabellings. In this case we display the relevant edges of G with their flow value.

Number of non-saturating pushes, Lemma 5.7: The number of non-saturating pushes is, at most, $4n^2m$.

To show this, we will use a proof method called *potential method*. We define a potential function, that is a non-negative function of the data of the algorithm, and then we analyse how it changes with each operation in the algorithm. Now that we have shown how many relabels and saturating pushes can happen at most in the whole algorithm, and we know some starting and ending conditions, we can use a potential method to somehow "isolate" the number of saturating pushes from the equation and count them too.

Proof: Let Φ be a potential function, and let $\Phi(i)$ be the value of the function at step i in the algorithm. We define Φ as the sum of the heights of the vertices with excess:

$$\Phi = \sum_{\{v|\alpha(v)>0\}} d(v)$$

At any point in the algorithm, $\Phi \geq 0$ because the excesses are always non-negative. Furthermore, at the start of the algorithm, $\Phi(0) = 0$ because the excesses are at vertices with height 0, and at the end $\Phi(i = end) = 0$ too, because no vertex has excess anymore.

Therefore, the number of increases and decreases of Φ along the execution of the algorithm have to be the same. We will now analyse what effect has each operation in Φ:

- Relabel: Since it is always performed on a vertex with excess that is contributing to the sum, it increases Φ by exactly 1.
- Saturating push on edge (u, v): It increases Φ by at most $d(v)$ if u does not clear its excess and v had no excess before, because after the push v will contribute to the sum. This increase can be of, at most, $d(v) = 2n - 1$. It can also decrease Φ by $d(u)$, at most $2n - 1$, if u clears its excess and v already had excess before.
- Non-saturating push on edge (u, v): It clears the excess of u, therefore $d(u)$ stops contributing to Φ. If v starts to have excess, the increase in potential will be one less

than the decrease because of u, since $d(v) < d(u)$. If v already had excess, the decrease in potential can be of up to $2n - 1$.

Since the non-saturating pushes only decrease the potential, we are interested on counting how many times can the potential rise, to bound how many times, at most, it has to decrease because of non-saturating pushes.

The number of relabels is at most $2n^2$, and the increase of Φ by them is then, at most, $2n^2$. The number of saturating pushes can be higher: $2nm$ at most, and each one can at most increase the potential by $2n-1$, so it can increase at most by $4n^2m$. This makes the number of non-saturating pushes be, at most, $4n^2m + 2n^2$ if each one decreases Φ by 1.

This bound is quite loose, however it is enough to prove that the algorithm always terminates in $O(n^2m)$ operations (it is the asymptotic bottleneck). This concludes the termination and correctness argument.

5.3.4 Complexity

The number of operations is not enough to determine the complexity of the algorithm, since we have not argued how complex it is to perform the operations, neither how to select a vertex with excess efficiently. In this section we will show how we can accomplish a true $O(n^2m)$ complexity for the generic algorithm, and in further sections we will show how this bound can be improved to $O(n^3)$ and $O(n^2m^{1/2})$ with different vertex selection patterns.

Vertex selection

To select a vertex with excess efficiently, we can maintain a set S that initially contains the adjacent vertices to s (minus t if it were adjacent), and later is updated in each push operation by adding and removing vertices. There are data structures to manage sets in constant or expected constant time, as we will see in the implementation section. Thus, the vertex selection can be done in constant time.

Pushes and current edges

A push in a vertex can be done in constant time, but to perform it, we must select an edge incident from the vertex. Checking all the edges incident with from a vertex each time that the vertex is selected for a basic operation would be inefficient. Fortunately, we can do better by storing a value called the *current edge* per vertex.

If we have an implementation that lets us search the edges adjacent to a vertex sequentially, like a list (described in the implementation section X), we can store a pointer or an index to one of them. We will call this index the *current edge*. We can set this index to the first edge in the sequence for each vertex at the start of the algorithm, and then increase it when the current edge is not eligible anymore for a push operation. When the edge list is scanned entirely, then the vertex has no more edges going downwards and we should apply a relabel operation, after which the current edge index will be set to the start again.

It might not be clear why this approach works. Let u be a vertex that has its current edge in an arbitrarily advanced position in the list. What if a vertex v that appears before the current edge in the list becomes available again for a push? A relabel of u could be made before this push operation is explored. But this situation cannot happen.

If the current edge is advanced past an adjacent vertex v it is because of one of these two situations:

1. $d(v) \geq d(u)$ and a push is not applicable.
2. The edge (u, v) is saturated.

If 1) is the case, then a push will not happen until u is relabelled because the label of v will not decrease. If 2) is the case, for (u, v) to be non-saturated again, a push on (v, u) has to happen. That demands not only a relabel of v, but also of u to be able to push again to v. As we explained when we discussed the number of saturating pushes, this will not happen until a relabel of u. Therefore, the current edge scannning strategy will always be correct.

The amount of work that is done scanning edge sequences will depend on how many relabels we have. We know that there will be, at most, $2n - 1$ relabels per vertex. This means that, for each vertex, we will perform $(2n-1) * deg^+(v)$ operations, where $deg^+(v)$ is the out-degree of the vertex. In total in the whole algorithm we will do the following operations:

$$\sum_{v \in V}(2n-1) * deg^+(v) = (2n-1) * deg^+(v_1) + \ldots + (2n-1) * deg^+(v_n)$$
$$= (2n-1)(deg^+(v_1) + \ldots + deg^+(v_n)) = (2n-1)(m) = O(nm).$$

Relabels and non-saturating pushes
A relabel consists on incrementing by one the label of a selected vertex, that can be done in constant time. There are $O(n^2)$ relabels during the algorithm (Lemma 5.5), which take less time than the previous bound on scanning the adjacent edges of the vertices. Therefore, the relabels are not a bottleneck on the algorithm.

This analysis concludes that the bottleneck of the algorithm is the number of non-saturating pushes, since there are $O(n^2m)$ of them. Each one can be done in constant time once we have found the edge, so the asymptotic complexity of the algorithm is $O(n^2m)$ in the end.

5.4 Improvements to the generic algorithm

The generic algorithm, as it was presented, has a very poor practical performance. However, we can tune it heavily to achieve both better asymptotic complexity and also practical speedup. In this section we present three basic improvements: a more accurate relabel, an early stop condition, and a way to keep the labels of the algorithm exact.

5.4.1 Maximum relabel

Sometimes, when relabelling a vertex we can increase its label by more than one. In fact, we can always increase it by this quantity:

$$d(u) \leftarrow \min\{d(v) + 1 | (u, v) \in E(G_f)\}$$

When a relabel is needed on u, all the edges (u, v) in G_f going out of u are flat or go uphill. Furthermore, we know that at least one of these edges exist: the vertex has excess, therefore has flow coming into it, and that means that there is an edge in G_f going out of u. This guarantees that the formula always increases the height of u. The maximum relabel that we can do is the relabel that ensures that any edge coming out from u to v satisfies the constraint: $d(u) \leq d(v) + 1$, and it will be satisfied when the least *steep* edge turns into a downwards edge, with $d(u) = d(v) + 1$.

If we implement this kind of relabel, each time that we perform a relabel on u we have to scan the edges going out of it, to take the minimum. Therefore, the complexity rises. However, it is not a bottleneck operation either:

Lemma 5.8: The improved relabel operations take at most $O(nm)$ steps during the whole execution of the algorithm.

Proof: As we did before, there will be $2n-1$ relabels per vertex at most, and each one will scan the $deg(v)$ edges that can come out of v:

$$\sum_{v \in V}(2n-1)*deg(v) = (2n-1)*deg(v_1)+\ldots+(2n-1)*deg(v_n)$$
$$= (2n-1)(deg(v_1)+\ldots+deg(v_n)) = (2n-1)(2m) = O(nm).$$

In practice, this kind of relabel should always be used, because it can avoid a lot of edge scannings between unnecessary relabels of the same vertex. However, for the following proofs of complexity we will use the naive relabel by 1 (unless stated otherwise), since it simplifies the explanations.

5.4.2 Early stop, phases

Sometimes we are not interested in getting the flow on each edge at the end of the algorithm, but rather just the value of the maximum flow or the capacity of a minimum cut. We can skip some steps in this algorithm to find a minimum cut before the full flow is computed. At this point, we can retrieve the value of the maximum flow or the capacity of the minimum cut by adding the flows that come into the sink. It is guaranteed that this situation happens when the vertices with excess are at height n or higher.

We will say that the algorithm has two phases: the first one goes from the beginning until all vertices with excess are at height n or higher, and the second phase goes from this point until the end. We present an adapted proof done by the author of the one shown in the preliminary paper by Goldberg and Tarjan [7]:

Lemma 5.9: When all the vertices with excess are at height n or higher (at the end of phase one), then the set of vertices $\overline{S}$ that have an augmenting path that reaches t, and $S = V \setminus S$ form a minimum s-t-cut $(S, \overline{S})$.

Proof: Lemma 5.0 ensures that there is no $(s \to t)$-path in G_f at any point in the algorithm, so $s \in S$. Furthermore, because there is always a path of length 0 from t to t, $t \in \overline{S}$ and $(S, \overline{S})$ is a cut. Lemma 5.0 can also be applied to all the vertices with excess: they are at height n or higher, thus an augmenting path cannot reach t, because any edge goes downwards at most 1 unit of distance in G_f. All the vertices with excess are in S. We will need to show two claims for the proof:

Claim 1: At the end of phase one, all edges from S to $\overline{S}$ are saturated, and the edges in the reverse direction have flow 0.

Proof by contradiction: If there existed a non-saturated edge (u, v) where $u \in S$ and $v \in \overline{S}$, then by definition of $\overline{S}$, there would be a $(u \to v)$-path in G_f connecting with a $(v \to t)$-path, and that contradicts $u \in S$.

Furthermore, if there were any edge (v, u) from $\overline{S}$ to S carrying positive flow, the reverse edge (u, v) from S to $\overline{S}$ would also have positive remaining capacity and would be in G_f, but this cannot happen as we showed above. Therefore, all edges from $\overline{S}$ to S carry no flow.

We cannot invoke the max-flow-min-cut theorem yet, because what we have in the network is a preflow and not a flow. However, the edges between S and $\overline{S}$ will not change until the end of the algorithm:

Claim 2: The flow of the edges in the cut $(S,\overline{S})$ and the ones in $(\overline{S},S)$ does not change anymore in the following execution of the algorithm (phase two).

Proof: Let $u \in S$ and $v \in \overline{S}$ be two adjacent vertices. The edge (u,v) is saturated as we showed before. For the flow between these two vertices to change, a push between (v,u) would be needed. But all the vertices with excess are in S, and $v \in \overline{S}$. Because no vertex with excess in S has a non-saturated edge going to $\overline{S}$, they cannot push any excess to v, and v will never be selected anymore for a basic operation. Therefore, a push in the edge (v,u) will never happen again, and the flow in that edge will stay at 0. This also applies for forward edges with zero flow between $\overline{S}$ and S.

At the end of the algorithm, we have a valid flow (Lemma 5.2) and the cut $(S,\overline{S})$ stayed in the same state as in the end of phase one. We can now invoke the max-flow-min-cut theorem and state that $(S,\overline{S})$ is a minimum cut, that can be found when all the vertices with excess are at height n or above.

5.4.3 Active vertices, phase one

From now on, we will define an *active* vertex as a vertex with excess and with distance label less than n: v active $\Leftrightarrow \alpha(v) > 0$ and $d(v) < n$. All of our following algorithms will only perform the phase one of the computation, and will select only active vertices to perform their operations. The algorithms will return the value of the flow, instead of the full flow. Goldberg and Tarjan [7] showed that the complexity of phase two is only $O(nm)$, and its computational study with Cherkassky [3] has shown that the phase is very fast compared to phase one in most networks. Therefore, we will focus on discussing phase one for the rest of the thesis.

5.4.4 Exact distance labels

We hinted before that the labels of the vertices are a lower bound of the distance of a vertex to the sink following a path in G_f. There is a way of keeping the distance labels exact at every time in the algorithm, which is not a bottleneck operation. The following operations should be used with a maximum relabel strategy.

In this section we will discuss which extra operations we need to perform and their correctness and complexity. In the implementation section X we will describe the implementation details.

Initial exact labelling

To start the algorithm with an initial exact labelling, we can perform a backwards BFS from the sink. This BFS search will discover all the vertices v that have an augmenting $(v \to t)$-path in G_f, and will store the distance of each vertex v to the sink, as we did with Dinic's algorithm. However, we can also start the algorithm with an inexact labelling (the basic one) and work towards the exact labelling as the algorithm advances.

What do we do with the vertices that do not have an augmenting path to t? It is a reasonable doubt that in another stage of the algorithm, they could be used. But it is not difficult to see that these vertices do not contribute to increase the flow. Let $\overline{S}$ be the set of vertices that are found with the BFS search, and $S = V \setminus \overline{S}$. $s \in S$ by Lemma 5.0, and $t \in \overline{S}$ because it always has a path of length 0 to itself. This makes $(S,\overline{S})$ an s-t-cut. All the edges in the cut are saturated, otherwise there would be an augmenting path from a vertex of S to t. Also, all the edges from $\overline{S}$ to S have flow 0, otherwise the backwards edge of one of these would also form an augmenting path from a vertex in S to t.

The following easy lemma is a statement that seems to be overlooked in the literature, but that it has a strong consequence that is used here and in the gap relabelling heuristic that we will see later. The proof is originally done by the author.

Lemma 5.10: Let $(S, \overline{S})$ be an s-t-cut with a preflow where all the edges from S to $\overline{S}$ are saturated, and all the edges from $\overline{S}$ to S have zero flow. Then, the sum of flows in the cut is greater than or equal to the value of the maximum flow.

Proof: The sum of the flows across the cut equals the capacity of the cut (we can use Lemma 1.4). If the cut is minimum, the max-flow-min-cut theorem (Theorem 1.5) states that it will be equal to the value of the maximum flow. Otherwise, if the cut is not minimum, its capacity is greater than the value of the maximum flow. The sum of flows across the cut is equal to its capacity, so the sum is greater than the value of the maximum flow.

This means that all the required flow is on the $\overline{S}$ side of the cut. Furthermore, because the cut is saturated, there is no way that the vertices in S could be used in a path to transport more flow: any edge that returned to $\overline{S}$ already has all the flow that it can carry. The important consequence is that the vertices in S are not useful anymore to compute the value of the maximum flow, and therefore they can be ignored. We can ignore them by raising their distance label to n, for example, and not consider them active anymore.

Keeping the labels exact

Now, to keep the distance labels exact throughout the whole algorithm, we need a ”stronger interpretation” [8] of what a current edge means. This time, a current edge should be unsaturated and point to a vertex that is one unit of distance closer to the sink. We can start the algorithm with an exact distance labelling, and set the current edge of all the vertices found to the edge on the shortest path to t.

Whenever a saturating push happens, before we select another vertex for a basic operation, we should update the current edge with the following operations:

- **Operation 1:** Scan the edge list for a new valid current edge.
- **Operation 2:** If operation 1 is not successful, before relabelling v, scan the vertices adjacent to v. If there is a vertex u whose current edge is (u, v), it will have to change. Relabel v and apply operation 1 (or 2 if applicable) on all vertices u that have to change the current edge.

Let's see why these operations keep the labels exact. Let v be the vertex where we apply the operation 1. If it is successful, we are done, because the vertex has found another path to t, of the same length, through the end vertex of its new current edge.

If not, we should perform operation 2 and relabel the vertex (we have shown before that no more edges will be available to v until a relabel happens, so we can perform the relabel immediately). When the maximum relabel is done, we set the label to one more than the adjacent vertex that is closer to the sink. The new distance label is exact.

Furthermore, operation 2 searches for all vertices u whose shortest path to t contained v, because they could increase their distance to t. Therefore, we must perform the operations recursively on all those vertices u too, after v is relabelled.

This recursive operations will not stop until a current edge path is formed to t. However, they should stop if a vertex reaches distance n or more, because we have shown that in that case, they will never be useful to push flow to the sink anymore (Lemma 5.9). In fact, the distance label that would have been chosen would have given a shortest path to s, that would have been used to push the excess flow back into the source.

Operation 1 is a part of the $O(nm)$ time required to scan the edge lists of vertices to apply push operations. Operation 2 can be done at the same time as we scan the edge list of a vertex to perform a maximum relabel. Therefore, keeping the labels exact will not rise

the asymptotic complexity of the algorithm, it will just change the order in which edge lists are scanned and relabels are performed.

We must be aware that these operations cause relabels on non-active vertices. The bound on the height of relabels still holds whenever we set the height to one more than another vertex: we are finding paths to t that have a bounded height. Furthermore, we only explored the exact distance labelling strategy for algorithms that aim to finish the phase one only: any vertex whose height rises above n will be ignored.

There is an edge case that we must be aware of: if we do not use an initial exact labelling, dead ends will not be discovered by a BFS, and can be problematic. A vertex in a dead end (that is, a path that does not reach t) that receives excess will return it all to the path where it came from. In the last push, it saturates a backwards edge and stays with zero excess. Because of the saturating push, the algorithm might try to relabel it. Because the vertex has no excess, we cannot guarantee anymore that there will be an edge in G_f coming out of it. The maximum relabel operation might be done over an empty set. In this case, we can just ignore the vertex (set the label to n) because we know that it lies on a dead end.

5.5 FIFO Algorithm

We will now present a variant of the generic algorithm that performs better than it, by using a FIFO (*First-in, first-out*) queue to select the active vertices. The correctness and termination analysis that we did before applies to any vertex selection strategy, therefore we have freedom to choose the strategy that we want without hurting the complexity bound, as long as the vertex selection is done reasonably fast and does not become a bottleneck of the algorithm.

In the generic algorithm, after each basic operation we selected a new vertex. However, here we will select one and apply pushes on it until a relabel has to be done, an operation that is called *discharge* in the literature [8]. To keep track of the active vertices, first we will push all the initial ones (the ones adjacent to s) to a FIFO queue Q. Then, we will poll one vertex from the queue and apply a discharge operation to it. If any push that is performed creates an active vertex, this one will be added to the rear of the queue. If the discharge on a vertex u ends and u still has excess, it will be added to the rear of the queue (if its new distance label is less than n). Otherwise, if u clears its excess, its discharge ends

and it is not added to the queue.

Algorithm 11: Discharge operation (without exact labelling)

Input: A network $G = (V, E)$ with capacities on the edges, a preflow f, a distance labelling d, and a queue of active vertices Q

```
1  //Discharge;
2  if Q is not empty then
3  |  Vertex u ← poll Q;
4  |  while α(u) > 0 and there is an edge e(u,v) with d(u) = d(v) + 1 do
5  |  |  boolean v_active = α(v) > 0 and d(v) < n;
6  |  |  push(u,v);
7  |  |  if not v_active then
8  |  |  |  add v to the rear of Q;
9  |  end
10 |  if α(u) > 0 then
11 |  |  relabel(u);
12 |  |  if d(u) < n then
13 |  |  |  add u to the rear of the queue;
```

A final note: if there is a push on vertex (u, v), then v for sure will have excess after the push. If it had excess before then it was already added to the queue and we do not need to add it again. Otherwise, we shall add it to the queue, since it became an active vertex.

5.5.1 Complexity

All the complexity bounds that we derived later are valid upper bounds for any variant of the generic algorithm that selects vertices in constant time. However, we may improve them for special cases that we can analyse, like this one. We will do a more careful analysis of the number of non-saturating pushes, that was the bottleneck in the generic analysis.

We will introduce the concept of *passes* through the queue. The first pass is defined as all the discharge operations that are performed on the vertices that were added during the initialisation. These vertices are the first ones to come in, so they will be the first ones to come out of the queue (FIFO order). Whatever vertices they add to the queue, they will be processed later. We then define the pass $i + 1$ of the queue as the consecutive discharge operations on the vertices that were added to the queue in the previous pass i.

The reason why the concept of pass is useful is because a pass consists at most of n vertices, and at most one non-saturating push happens for each vertex in a pass. When one happens, that vertex clears its excess and the discharge ends. Therefore, for each pass we will have at most n non-saturating pushes. This proves the following Lemma:

Lemma 5.11: There are n non-saturating pushes in each pass.

Lemma 5.12: The number of passes through the queue is less than $4n^2$.

Proof: To count the number of passes we will use again a potential method. We will distinguish the passes where the potential rises, the passes where it stays the same and the passes where it decreases. The sum of all of them will be the total number of passes.

Let $\Phi = \max\{d(v) | v \text{ is active}\}$ be a potential function. Φ is zero at the start and at the end of the algorithm: at the start all active vertices are at height zero, and at the end there are no active vertices.

- If Φ stays the same in a pass, then there is at least one vertex that underwent a relabel: all the vertices at the maximum height cleared their excess (that is why Φ did not increase), and a different vertex occupied the maximum height position, by a relabel.

- If Φ increases by Δ on a pass, some vertex had to have a relabel, since it is the only way that the labels increase.

Because there is a relabel at least in each pass where the potential increases or stays the same, the number of passes where the potential increases or stays the same is less than $2n^2$ (Lemma 5.5).

The potential starts and ends at zero, so the amount of passes where it decreases (at least by 1) has to be less than $2n^2$ too. Therefore, total number of passes is less than $4n^2 = O(n^2)$.

Theorem 5.13: The complexity of the FIFO push-relabel algorithm is $O(n^3)$.

Proof: There are $O(n^2)$ passes through the queue (Lemma 5.10) and there are at most n non-saturating pushes in each pass, therefore the amount of non-saturating pushes is $O(n^3)$, which is the bottleneck of the algorithm.

5.6 Highest-Level algorithm

There is a better selection rule than the FIFO one in terms of computational complexity: select every time the active vertex with a higher label. Cheriyan and Maheshwari [1] showed that the number of non-saturating pushes with this selection rule is $O(n^2 m^{1/2})$. We will present and explain the proof in ”Network Flows” by Ahuja, Magnanti and Orlin [14].

The algorithm also uses the discharge operation, and uses some sophisticated data structures to select the vertex with highest excess at each point. The implementation section will discuss how to do it in detail. We will show that the naive way to do it has a complexity higher than constant, but it is not a bottleneck for the algorithm. Furthermore, we will also show an original way of retrieving the highest active vertex in true constant time.

In this theory section, we will focus on counting the non-saturating pushes. The proof is very involved, and we will need a few more graph-theoretical concepts to explain it.

5.6.1 Additional concepts

- **Rooted tree:** A rooted tree is a tree where one vertex has been designed the root, and all the edges have a direction either *towards* or *away from* the root. We will focus on rooted trees whose edges are directed towards the root.

- **Descendants, ascendants:** In a rooted tree where the edges are directed towards the root, the descendants of a vertex v are all the vertices whose path to the root passes through v, including v. The ascendants of v are the vertices in the path that are closer to the root than v.

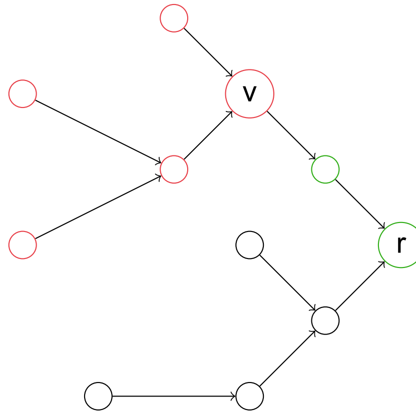


Figure 5.5: A rooted tree. The red vertices are the descendants of v, and the blue ones are the ascendants. r is the root.

5.6.2 Complexity

To analyse the complexity, we will assume that the distance labels of the vertices are kept exact during the algorithm, as we explained in section 5.2.

We will divide the algorithm in phases, like in the FIFO complexity argument, and we will distinguish between ”cheap” phases, where not too many non-saturating pushes happen, and ”expensive” phases, where most of the non-saturating pushes happen, using a parameter. We will use a sophisticated potential function for determining the number of non-saturating pushes in both types of phases, optimise the parameter that distinguishes cheap and expensive phases, and add them both up.

To set up the potential function, we need some definitions, some observations and some lemmas that stem from them:

Current forest: At every state of the algorithm, every vertex has a current edge. Denote the set of current edges by F. With the exact labelling there are no cycles of current edges, since each one goes to an end vertex that is a unit of distance lower to the starting vertex. Furthermore, some vertices will not have a current edge if they have not been relabelled yet. If we take the graph formed by the edges of F and its endpoints as vertices, it will be a *forest*, a possibly disconnected graph whose connected components are trees. We call F the current forest.

Lemma 6.0: Each tree in the current forest is a rooted tree.

Proof: There is only one current edge going out from each vertex, but there can be many arriving at it. For a vertex v in a tree, we can define the vertices incident with v as its descendants, and the endpoint of its current edge as its only ascendant. This way, we will end up with a rooted tree.

Label of descendants: The descendants of the root of a tree in F, except the root itself, all have a distance label higher than the root (because of the exact distance labelling).

Maximal active vertex: We define a maximal active vertex as an active vertex with no active descendants. In other words, the descendants of a maximal active vertex have no excess.

Set of maximal active vertices: Let H denote the set of maximal active vertices at each time in the algorithm.

Lemma 6.1: If two different vertices u and v are in H, they do not share any descendants.

Proof: To see this, remember that each tree in F is a rooted tree. By contradiction, pick a descendant $w \notin H$ of two maximal active vertices u and v in H. The maximal active vertices can always be found by traversing the ascendant current edges from w. Let u be the first maximal active vertex found in this way. v can be found by traversing more ascendant edges past u. But then, u would be a descendant of v, and that contradicts the definition of maximal active vertex.

Lemma 6.2: The highest level algorithm always pushes from a maximal excess vertex.

Proof: Let u be the active vertex with the highest level. Any vertex that is at the same distance as u is not one of its descendants, since the descendants of u have a higher label (except u itself). All vertices at a distance higher than u have no excess by definition of u. Therefore, any descendant of u has no excess, and u is a maximal excess vertex.

Now, we can introduce the phases of the algorithm:

Phase: Let $d_{max} = \max\{d(i) : i \text{ is an active vertex}\}$. We define a phase as the sequence of pushes during which d_{max} remains constant.

As we saw in the FIFO complexity section, d_{max} can be seen as a potential function. This time, instead of defining "passes" through the queue, we define "phases" in a different way. We shall count the number of phases:

- d_{max} only increases $O(n^2)$ times: the maximum distance can only increase with relabels, so it will only increase at most $2n^2$ times (Lemma 5.5).
- d_{max} is zero at the end of the algorithm, so it has to decrease $O(n^2)$ times too.
- This implies that the number of phases where d_{max} stays the same is $O(n^2)$ too.

Cheap and expensive phases: Let K be a parameter to optimise. We define a *cheap* phase as a phase that performs less than or equal to $2n/K$ non-saturating pushes. We say that the phase is *expensive* otherwise.

Now, we shall count the number of non-saturating pushes in cheap and expensive phases, with a potential function.

Potential function Φ: Let $\Phi : V \to \mathbb{Z}_{\geq 0}$ be a helper function defined as:

$$\Phi(v) = \max\{0, K + 1 - |D(v)|\}$$

And let $\Phi = \sum_{i \in H} \Phi(i)$ be a potential function.

We shall spend some time understanding the definition of the potential function Φ, first looking at the helper function $\Phi(v)$. It is a function of the descendants of a maximal active vertex v, where the term of the descendants $|D(v)|$ is subtracted. That means that it has its peak when $|D(v)|$ is as small as possible. This happens when v has no descendants

other than itself: $|D(v)| = 1$. The function has a minimum when $|D(v)|$ is greater or equal to $K + 1$. Therefore:

$$0 \leq \Phi(v) \leq K, \ \forall v \in H.$$

The potential function changes when H changes, or when the descendants of the vertices in H change. If H changes, Φ will change unless we add or delete a vertex v such that $\Phi(v) = 0$, or what is the same, $D(v) \geq K + 1$. On the other hand, if $D(v)$ changes it is because the current edge of a vertex v in H changed, and Φ probably will change depending on the amount of descendants that it has after the current edge change.

We have to study how the potential function changes across the algorithm. We shall discuss the action of a push or a relabel independently of the current edge creation that they may cause afterwards. Therefore, we divide the analysis in changes made by a non-saturating push, by a saturating push, by a relabel and by a current edge addition.

1. **Non-saturating pushes:** A non-saturated push is performed on a current edge. Let (u, v) be this edge. The push leaves it unsaturated, so the current edge will not change, and therefore F will remain the same.

   Note that $u \in D(v)$, thus v was not a maximal excess vertex, and also $|D(v)| > |D(u)|$. After the push, u becomes inactive, and v might or might not become a maximal active vertex.

   If $|D(u)| > K$, then u has too many descendants and is not contributing to the potential. Because v has more descendants, it will not contribute either, whether it becomes a maximal active vertex or not. On the other hand, if $|D(u)| \leq K$, it will decrease the potential when u clears its excess. The minimum decrease is of 1, if v becomes a maximal active vertex, and the maximum is of K, otherwise.

   Because we are looking to bound the non-saturating pushes, we will try to count which operations of the other types increase the potential, to see in the worst case how many non-saturating pushes will be needed to decrease it to zero, and also try to bound how many non-saturating pushes are performed that don't change the potential.

2. **Saturating pushes:** A saturating push on an edge (u, v) makes it disappear from the the current forest. Vertex v might or might not become a maximal excess vertex, and vertex u might or might not release all its excess.

   We're interested in seeing if the saturating pushes can increase the potential. If i does not release all its excess, $|D(i)|$ will keep on contributing to Φ. Note that $|D(i)|$ does not change in this operation. If j becomes a maximal excess vertex (it can happen because i is not a descendant of j anymore), Φ can rise by K at most (if $|D(j)| = 1$ after the push).

3. **Relabel:** If a relabel happens on vertex u, it is because it has no current edge any-more. Therefore, it is a root vertex in the current edge forest. For any descendant whose current edge had u as an endpoint, the current edge will become invalid with the relabel. These will be called to be relabelled later, by the exact labelling operation. But for this operation, Φ increases by at most K, because u loses all its descendants. The relabelling does not create new maximal excess vertices.

4. **Current edge creation:** This can happen after a saturating push, or after a relabel. When a new current edge appears, it does not create new maximal vertices with excess, but it might connect two vertices (u, v) that were both maximal active vertices

before, and that will make v not be maximal anymore. In that situation the potential would decrease, otherwise the potential does not change.

Now we got all the ingredients to count the non-saturating pushes in cheap and expensive phases. Note that, a priori, we don't know how many of the phases in the execution of the algorithm will be cheap and expensive. We just know that the sum of them includes all the pushes. Depending on the choice of K, there would be a lot of cheap phases (if K is big) or a lot of expensive phases (if K is small). Furthermore, not all the non-saturating pushes decrease Φ: only those of maximal active vertices that have less than or equal to K descendants.

Non-saturating pushes in cheap phases: By definition, they have less than or equal to $2n/K$ non-saturating pushes. These pushes might not change Φ at all: if there are very few pushes per phase, then all of them could be done on vertices with a lot of descendants. Therefore, we cannot conclude anything by the potential method in cheap phases. We can put a naive bound: there are $O(n^2)$ phases, so at most, $O(n^2) * (n/K) = O(n^3/K)$ non-saturating pushes can happen in cheap phases.

Non-saturating pushes in expensive phases: By definition, they have at least $2n/K$ non-saturating pushes. We will count first how many do not decrease the potential, and then how many do, with some lemmas.

Lemma 6.3: In a phase, only the vertices at height d_{max} can push flow.

Proof: The highest level algorithm always chooses the highest active vertex. By definition, this height is d_{max}, therefore no vertex higher up will be chosen. During this phase d_{max} remains constant, so a vertex with excess that is lower will never be chosen: the vertex at d_{max} will have preference.

Lemma 6.4: At any moment in the algorithm, the network can only contain at most n/K maximal excess vertices with more than K descendants.

Proof: The descendants of each maximal excess vertex are disjoint, so:

$$\text{\#max. excess vertices} \cdot (K+1) \leq n \Rightarrow \text{\#max. excess vertices} \leq n/(K+1) < n/K$$

Lemma 6.5: In an expensive phase, there can be at most n/K non-saturating pushes that do not change the potential Φ.

Proof: The non-saturating pushes that do not change the potential are pushes on maximal active vertices that have $K+1$ or more descendants. Lemma 6.3 states that all the pushes in a phase are done at the same height, d_{max}. When a non-saturating push happens on a maximal active vertex with $K+1$ descendants at height d_{max}, this vertex cannot be eligible for a push again in this phase, because for that to happen, another vertex would have to push flow to it. But this flow should be pushed from a higher distance than d_{max}, which will not happen in this phase. Therefore, each maximal active vertex with more than K descendants can only do one non-saturating push in this phase, and Lemma 6.4 states that there can be at most n/K of them.

Lemma 6.6: An expensive phase performs, at least, n/K non-saturating pushes that change the potential Φ.

Proof: By definition there are at least $2n/K$ non-saturating pushes in an expensive phase, and Lemma 6.5 tells us that at most n/K of them do not change Φ, therefore at least

another n/K of them must change Φ. The important thing is that this result states that the number of non-saturating pushes that change the potential in an expensive phase is, at least, the same as the number of non-saturating pushes that do not.

Lemma 6.7: The amount of non-saturating pushes in expensive phases is $O(nmK)$.

Proof: We have seen that the saturating pushes and the relabels can increase the potential, each one by at most K. There are at most $O(nm)$ saturating pushes in the algorithm (Lemma 5.6) and at most $O(n^2)$ relabels (Lemma 5.5), so in total the increase can be of $O(nm) \cdot K + O(n^2) \cdot K = O(nmK)$. Therefore, the maximum number of non-saturating pushes that change the potential must be $O(nmK)$. Because they are the same or more than the number of non-saturating pushes that do not change the potential, the total number of them is also $O(nmK)$.

Lemma 6.8: The optimal value of K is $n/m^{1/2}$, therefore the algorithm performs at most $O(n^2m^{1/2})$ non-saturating pushes.

Proof: The total number of non-saturating pushes is divided into the ones that are performed on cheap phases, and the ones on expensive phases, that is: $O(n^3/K)+O(nmK)$. One of this terms increases with K, and the other decreases with K. Note that this formula was found by analysing the algorithm in its worst-case scenario for any value of K, so any value that we choose will actually be a valid bound for it. However, we are interested in finding which is the ”best worst-case bound”, to find the true asymptotic complexity of the algorithm. Since this formula is simple enough, we can differentiate it and find the minimum value of K with basic calculus, by setting the derivative to 0 and solving:

$$\frac{\partial(n^3/K + nmK)}{\partial K} = \frac{-n^3}{K^2} + nm = 0 \rightarrow$$
$$\frac{n^3}{K^2} = nm \rightarrow n^3 = nmK^2 \rightarrow n^2 = mK^2$$
$$K^2 = \frac{n^2}{m} \rightarrow K = \frac{n}{m^{1/2}} \rightarrow$$
$$O\Big(\frac{n^3}{n/m^{1/2}}\Big) + O(nm(n/m^{1/2})) = O(n^2m^{1/2}) + O(n^2m^{1/2}) = O(n^2m^{1/2}).$$

5.7 Heuristics

There are two further improvements that we can add to the push-relabel algorithms to achieve great practical performance. These are the *global relabelling* update, described in the paper by Goldberg and Tarjan [8], and the *gap relabelling* update, discovered independently by Cherkassky and by Derigs and Meier [4]. The global relabelling update can be used instead of an exact distance labelling strategy, and the gap relabelling may be used alongside any of the algorithms and heuristics that we have described until now.

5.7.1 Global relabelling

Instead of keeping the labels exact at every time in the algorithm, we can choose to update them periodically by performing a backwards BFS search from the sink, just as we did with the initial exact labelling. Furthermore, it can identify vertices that are no more connected to the sink by augmenting paths, and that can be ignored (Lemma 5.10).

This heuristic requires the programmer to choose how often are these updates going to happen. The BFS is a computationally expensive operation in comparison to the rest of

the operations ($O(m)$), and performing too many can hurt the performance of the algorithm. Authors like Goldberg [8] and Cherkassky [3] suggested that the updates be performed after n relabellings. In its computational study, Cherkassky noted that the optimal relabelling frequency was different in different types of networks.

5.7.2 Gap relabelling

It was discovered independently by Cherkassky [2] and Derigs and Meier [4]. We will follow the explanation of Derigs and Meier. A *gap* is defined as a distance k such that no vertices have a distance equal to k, and there exists a vertex with a distance label higher than g.

Lemma 6.9: If $0 < k < n$ is a gap, then all vertices that have a distance label higher than k are useless to increase the value of the flow.

Proof: Take the set $S = \{v|d(v) > k\}$ of vertices that are above the gap k, and let $\overline{S} = V \setminus S$. We are in the same situation as in Lemma 5.10: $s \in S$, $t \in \overline{S}$, and $(S, \overline{S})$ is an s-t-cut. All the edges in the cut are saturated, because an edge from S to $\overline{S}$ cannot exist in G_f: it would jump downwards from $k+1$ to $k-1$. Also, all the edges from $\overline{S}$ to S have zero flow, because otherwise the backwards edge of one of these would also jump from distance $k+1$ to $k-1$. We invoke Lemma 5.10 to conclude that the vertices in S are useless to increase the value of the flow, and they can therefore be ignored.

We can check for gaps after every relabel operation, if we maintain information about the height of all vertices at all times. Note that this time we are required to keep track of all the vertices, not only the ones with excess. The implementation section will deal with the details of how to do it in the FIFO algorithm and the Highest Vertex algorithm. This heuristic can help to detect much quicker the minimum cut of phase one.

6 Implementation of push-relabel algorithms

6.1 Introduction

The implementation of the basic push-relabel algorithms is easy, but it has poor practical performance. To get a performance that can beat the augmenting path algorithms, we need to include the heuristics explained in the theory section, that can be very tricky to implement. We will show the details of each heuristic, and the pseudo-code of two of the most efficient algorithms that the author could implement, one with FIFO selection, and the other with highest vertex selection. The implementation of the other explored push-relabel variants are very similar to these.

6.2 Residual graph and Vertex structures

We can use the same residual graph as we used with augmenting path algorithms. There is more data to be stored per vertex in the push-relabel algorithms, and a vertex object per vertex is useful to store them. For an object *vertex*, we have the following fields:

- Index of the vertex: *vertex.v*. Set to the vertex identifier.
- Current edge: *vertex.currentEdge*, default 0.
- Label/distance/height: *vertex.height*, default 0.
- Excess: *vertex.excess*, default 0.
- List pointers (if necessary)

We can keep an array of these objects, and index it by the vertex identifier, *vertex.v*.

6.3 Linked lists instead of sets

In several sections of the algorithm we need to maintain subsets of vertices with a given property (for example, being active, or having a specific height). The operations we need on these subsets are insertion and deletion of specific vertices in constant time, retrieval of an arbitrary vertex from the set, and sometimes iteration in an unspecified order.

To support these operations in constant time we can use a hash set. Hash sets provide expected constant time for all of the operations, and they can have a good performance when a good hash function is used.

An alternative is to use linked lists. However, these have the problem of deletion: deleting an arbitrary value cannot be done in constant time unless we hold a pointer to the node of the list that holds the value. If we store such a pointer and the list is doubly linked, we can delete the value in constant time. At all times we must take extra care to dispose the pointers to the items that have been deleted from the list, to avoid memory leaks. Java's standard implementation of a linked list does not expose the nodes, and avoids this functionality. However, it is not difficult to do a custom implementation that supports it. In this text, a linked list that supports deletion of nodes will be called an *eraser linked list* (in reference to C++ lists, where deletion of nodes is possible).

The author chose to use eraser linked lists instead of hash sets in the implementation.

6.4 Generic algorithm

The only implementation decision out of the ordinary description of the algorithm is the choice of the set of active vertices. As discussed above, this can be done with a hash set or an eraser linked list. With an eraser linked list, every time that a vertex is added to the list of active vertices, a pointer should be kept in its object that points to its node on the list. When that vertex clears its excess, the vertex can be taken out of the active node list using its pointer. Afterwards, the pointer has to be disposed, for example setting it to $null$.

We will only implement the phase one of the algorithm. This means that, whenever a vertex is relabelled and its height becomes larger or equal to n, it will not be considered active anymore (as explained in the theory section).

6.5 FIFO Algorithm

The FIFO algorithm just requires the active vertex set to be implemented as a queue. We can use a regular linked list for this task, because we can poll a vertex from the front and add it to the rear of the list in constant time.

6.6 Highest Vertex Algorithm

This is the most involved algorithm in terms of implementation details. Keeping track of which is the highest active vertex at every point in the algorithm is not an easy task. The usual choice is to keep an array of size n of linked lists (we could also use hash sets), where the list at position i will hold the active vertices that are at height i.

The active vertices that are at the highest of heights have the same priority, therefore we can always choose to select to discharge the one that is at the start of the list. We do not need an eraser linked list for this data structure, and addition and deletion can be done in constant time.

However, we have a problem to keep track of which is the highest height at every moment, especially after having cleared the excess of the last vertex of the highest height. There are two approaches: the standard one keeps a pointer to the highest height, and increases and decreases it. Here is presented another different, original strategy: to use a stack that keeps which distances have vertices with excess. The index strategy is based on the text of *Network Flows* [14].

6.6.1 Index selection

We will keep an index, which we will call b, denoting the highest height where there is a vertex with excess. At the start of the algorithm, $b = 0$, and it is updated as following:

- When there is a relabel, since it is done in the highest vertex, the new height will be the highest one too. We set b to the new height found.

  However, if we wish to stop the algorithm at the end of phase one, we should not increase b if the new height is greater or equal to n.

- After the last vertex with excess in the highest height clears its excess, we cannot always know which one is the next lowest active vertex. We should iterate downwards in the array until we find the next non-empty linked list.

- When b reaches 0 and there are no active vertices at that height, there are no more active vertices and we can stop the algorithm.

Iterating downwards in the heights is not a constant operation for vertex selection. However, it is easy to see that across the whole algorithm, the time spent iterating is not a

bottleneck. Let $\Phi = b$ be a potential function. It is always non-negative, at the beginning of the algorithm it is 0 and at the end it is 0 too. b only increases due to relabel operations, and Lemma 5.5 tells us that the maximum increase is $2n - 1$ per vertex, so the total increase in b in the whole algorithm is $O(n^2)$. The total decrease must be $O(n^2)$ too, and thus it is not a bottleneck for the algorithm.

6.6.2 Stack selection

To select the highest vertex in real constant time, we can maintain a stack of integers that will contain the active height numbers, and no others, in increasing order. The top of the stack will always point to the highest active height. This way, when the last vertex at the maximum active height clears its excess, we just need to pop from the stack to know the next highest active height. This stack can be implemented with a linked list.

At the beginning of the algorithm, the stack just has a 0 on top because all the vertices with excess are at height 0. To keep the stack correct across the algorithm we need to make sure that, whenever a vertex with a given height becomes active, its height is added to the stack in order, and whenever the last vertex at one height clears its excess, this height is removed from the stack. The additional operations are:

- Just before a relabel is going to happen on the highest active vertex v, we check if v is the only remaining active vertex on its height. For that, we inspect the list of active vertices at height $d(v)$. If it is, when relabelled, there will be no more active vertices on that height, so we should pop the height of v from the stack. If the new height of v is below n, after the relabel we should add the new height to the top of the stack.
- When a push from a highest active vertex u to vertex v happens, the end vertex v might become active. The height of v is just one less than the highest one. We have to make sure that its height is present below the height of u in the stack. For that, we pop $d(u)$ from the stack, and we inspect if $d(v)$ is present next. If it is not, then we push $d(v)$ on the stack and push back $d(u)$ again.
- When a vertex becomes inactive after a push, we have to check if it was the last one remaining of its height. If it is, then we should pop its height from the stack. If not, we keep the stack as it is: there is another highest active vertex at its same height.

This way, we keep the stack consistent at every moment in the algorithm. Whenever we want to find the highest active height, we just have to inspect the top of the stack. When the last active vertex clears its excess, it will pop its height from the stack, and the stack will become empty. At this point, we can stop the algorithm.

6.7 Gap relabelling

Gap relabelling requires to keep track of the heights of the non-active vertices as well as the active vertices. We can implement it both with the FIFO selection and the highest vertex selection.

We will add the data structure that highest vertex selection uses to keep the active vertices, but for the non-active vertices too. In the FIFO algorithm, we can just use one structure because we do not need to distinguish between active and non-active vertices. In the highest vertex selection we need both, one for the active vertices as before, and another for the non-active vertices.

If we are using lists, the ones that FIFO uses, as well as the lists of non-active vertices in the case of highest vertex selection, must be eraser linked lists. In the FIFO case, we

need it when we might poll a vertex of the queue that is in an arbitrary position in one of the lists. In the highest vertex selection algorithm, when a push is done on a vertex without excess, it becomes active, and we must move it from the non-active list to the active list. In that case, we might have to take it out from the middle of the non-active list. We should handle the pointers to the lists as we did in the generic algorithm.

A gap is detected by checking, after a relabel of a vertex at height $d(u)$, if the lists of height $d(u)$ are empty. In the highest vertex selection case we have to check both active and non-active lists, and in the FIFO case we only need to check the general list of height $d(u)$. If a gap is detected, all the vertices above the gap should be deleted (their label raised to n), starting from the vertex that caused the gap. It must be noted that in the highest vertex case, all the vertices above the one that caused the gap are not active, and therefore we only need to check the non-active lists.

After the gap relabelling, the FIFO queue might contain a vertex whose label has been raised to n. Therefore, after polling any vertex from the queue, we must add a clause to check if it is still active. In the highest vertex case, the next highest vertex is now in a lower position, but this one will be found either by decreasing the pointer b as we usually did, or on the top of the stack. The stack was updated before the relabel and the pointer can be updated later (when iterating downwards), so we do not need to update anything.

It is also useful to keep track of the maximum height of a vertex, so that we do not have to scan all the heights above a gap, but only the ones that might have vertices. A value max_height can be used both in FIFO and in highest vertex selection as an upper bound (denoting the highest non-active height). max_height must be increased in each relabel, and can be set after a gap relabelling to the height just below the gap.

6.8 Global Relabelling

The global relabelling only consists of doing a BFS search backwards in G_f. For this, we only need to perform an ordinary BFS search from t, and consider an edge only if its reverse edge has a remaining capacity greater than 0. Furthermore, we should set the height of any vertex that was not discovered by the BFS to n, so that it is not considered active again.

A global relabelling can change completely the heights of the vertices, and this can affect to the data structures of the highest vertex selection and gap relabelling. The easiest move is to clear the data structures at the start, and fill them along the search, adding each vertex to the list of its new height. The FIFO queue does not need to be cleared, since it ignores heights; but we should add a clause to the FIFO algorithm to check if the vertex that is polled has been raised to n by a global relabelling.

We can set the max_height value of gap relabelling to the highest found height. In the highest vertex selection algorithm, if we are using the pointer b, we can increase every time we find a higher active vertex. In the stack case, because the vertices are found in increasing order of distance, we can fill the stack with the distances in the correct order as they are found.

6.9 Exact labelling

The exact labelling can be tricky to implement with a gap relabelling heuristic. We will present how to do it in the case of highest vertex selection with gap search, since it is where it is most efficient (see the results section X).

There are a few things to change:

1. Now, the discharge operation of a vertex should be extended until it finds a valid current edge, or it is relabelled. Clearance of excess is not a terminating condition anymore for the discharge operation.
2. Several relabelling operations may be made before the next discharge happens.
3. Relabels can happen now on non-active vertices. This has implications in the active vertex management that we had until now.

The discharge loop now will terminate when a relabel has to be done, or a valid current edge has found after clearing the excess of the vertex.

If a relabel is needed on a vertex u, the procedure now has to change. We will keep a FIFO queue of vertices to relabel, which we will call *relabel queue*. We add u at the start, and then, for each vertex on the queue, we will iterate the operations 1 and 2 discussed in the theory section.

For the operation 1, we just need to run a loop again through the edge list of the polled vertex v starting from its current edge, until we find a new valid one or the end is reached. If the end is reached, we pass to the operation 2.

In operation 2, we scan the edge list of u as we did before to find the new height. We can also set the new current edge properly at this time (as discussed in the theory section). At the same time, for each adjacent vertex v, we check if its height is one more than the height of u, and if its current edge points to u. We can check it by comparing it to the reverse edge of (u, v). If they coincide, we add v to the relabel queue.

Before the relabel, we would check if we had to pop the top of the stack in the stack version. Now, we need to do it only if the vertex to relabel is active, since we might be relabelling non-active vertices.

Then, we relabel the vertex. After it, we check for gaps as we used to do. If there is a gap we must clear the relabel queue, since all the vertices in the relabel queue are above the one that is being relabelled, and therefore have been raised to height n already.

And finally, if the relabel does not trigger a gap relabelling and it provides a label below n, we have to move it up in the height data structures, checking if the vertex belongs to the active or non-active vertices.

6.10 Pseudo-codes for FIFO and Highest Vertex selection

The biggest challenge that the author faced when implementing the algorithms was in the data structure management. The following pseudo-codes will hopefully help any reader to clarify when and how to update the data structures in the most complicated settings.

The code for FIFO uses global relabelling without gap relabelling and initial exact labelling, and the code for highest vertex selection uses gap relabelling, exact labelling, stack selection and basic initialisation. The basic initialisation is often, surprisingly, more efficient with gap search when using highest vertex selection.

In the code, a current edge is valid if it is not saturated and the start vertex is one height above the end vertex.

6.10.1 FIFO with global relabelling and initial exact labelling

```
1  //Initialisation
2  Initialise vertices[n] with each Vertex object;
3  Initialise FIFOQueue to an empty LinkedList<Vertex>;
4  vertices[s].height = n;
5  relabelCount = 0; //To execute global relabellings every n relabellings
6  for each ( edge out of s ) {
7      if (remaining capacity of edge >= 0) {
8          edge.flow = e.capacity;
9          edge.backwards.flow = -edge.capacity;
10         end_vertex = vertices[edge.v_out];
11         if(end_vertex is not s or t) {
12             if (end_vertex.excess == 0)
13                 add outVertex to FIFOQueue;
14             end_vertex.excess += edge.capacity;
15         }
16     }
17 }
18 globalRelabel(); //For the initial exact labelling
19 //Algorithm
20 while (FIFOQueue is not empty) {
21     vertex = FIFOQueue.poll();
22     if (vertex.height >= n) continue; //can happen due to global relabelling
23     adj = edge list of vertex;
24     //Discharge
25     while (vertex.height does not change and vertex.excess > 0) {
26         //Seek for push
27         while (vertex.current_edge is not at the end of adj
28                 and vertex.current_edge is not valid) {
29             current_edge = next edge in adj;
30         }
31         if (current_edge is valid) { //Push
32             delta = min(vertex.excess, remaining capacity of edge);
33             current_edge.flow += delta;
34             current_edge.backwards.flow -= delta;
35             if (end_vertex is not s or t) {
36                 if (end_vertex.excess == 0) {
37                     add end_vertex to FIFOQueue;
38                 }
39                 end_vertex.excess += delta;
40                 vertex.excess -= delta;
41             }
42         } else {
43             //There is no valid current edge: Relabel
44             newHeight = MAX_INT;
45             for each (edge in adj) {
46                 if (remaining capacity of edge > 0) {
47                     newHeight = min(newHeight, height of end vertex);
48                 }
49             } //Maximum relabel
```

```
50              vertex.height = newHeight + 1;
51              relabelCount++;
52              if (relabelCount % n == 0) { //Every n relabels
53                  globalRelabel();
54              }
55              relabel = true;
56          }
57      }
58      return calculateMaxFlowValue();
59  }
60
61  int calculateMaxFlowValue() {
62      max_flow = 0;
63      for each (edge out of t) {
64          forwards = edge.reverse_edge;
65          max_flow += forwards.flow;
66      }
67      return max_flow;
68  }
```

6.10.2 Highest vertex with exact labelling, gap relabelling and stack selection

Vertices now have an extra field called ”nonActivePointer” to store the pointer to its place in the non-active vertex list.

```
1
2   //Initialisation
3   Initialise vertices[n] with each Vertex object;
4   Initialise activeHeights[n] with n empty LinkedList<Vertex>;
5   Initialise nonActiveHeights[n] with n empty EraserLinkedList<Vertex>;
6   Initialise stack as an empty LinkedList<Integer>;
7   Initialise relabelQueue as an emtpy LinkedList<Integer>;
8   stack.push(0);
9   highestNonActiveHeight = 0;
10  for each (Vertex v except s) {
11      v.nonActivePointer = nonActiveHeights[0].addAndReturnPointer(v);
12  }
13  //From now on the above lines will be assumed when we add to,
14  //and remove from nonActiveHeights
15  vertices[s].height = n;
16  for each (edge out of s){
17      if (remaining capacity of edge >= 0) {
18          edge.flow = edge.capacity;
19          edge.backwards.flow = -edge.capacity;
20          end_vertex = vertices[edge.v_out];
21          if(end_vertex is not s or t){
22              if (end_vertex.excess == 0) {
23                  remove end_vertex from nonActiveHeights[0];
24                  add end_vertex to activeHeights[0];
25              }
26              end_vertex.excess += edge.capacity;
```

```
27          }
28      }
29  }
30  //Algorithm
31  while (stack is not empty) {
32      height = stack.peek(); //Does not remove from the stack
33      vertex = activeHeights[height].peek(); //Does not remove from the list
34      adj = edge list of vertex;
35      do {
36          //It starts from vertex.current_edge
37          while (vertex.current_edge is not at the end of adj
38                  and vertex.current_edge is not valid) {
39              vertex.current_edge = next edge in adj;
40          }
41          if (vertex.current_edge is valid and vertex.excess > 0) {
42              //Push is needed
43              delta = min(vertex.excess, remaining capacity of vertex.current_edge);
44              edge.flow += delta;
45              edge.backwards.flow -= delta;
46              w = vertices[vertex.current_edge.v_out];
47              if (w is not s or t) {
48                  if (w.excess == 0) { //w becomes active
49                      remove w from nonActiveHeights[w.height];
50                      add w to activeHeights[w.height];
51                      stack.pop(); //Removes from the top
52                      if (stack.peek() < w.height) {
53                          stack.push(w.height);
54                      }
55                      stack.push(height);
56                  }
57                  w.excess += delta;
58              }
59              vertex.excess -= delta;
60              if (vertex.excess == 0) { //vertex becomes inactive
61                  remove vertex from activeHeights[height];
62                  add vertex to nonActiveHeights[height];
63                  if (activeHeights[height] is empty) {
64                      stack.pop();
65                  }
66              }
67          } else { //Exact relabel
68              add vertex to relabelQueue;
69              while (relabelQueue is not empty) {
70                  vertex = relabelQueue.poll();
71                  //Operation 1: search for a valid current edge
72                  while (vertex.current_edge is not at the end of adj
73                          and vertex.current_edge is not valid) {
74                      vertex.current_edge = next edge in adj;
75                  }
76                  if (vertex.current_edge is valid) continue; //Nothing more to do
```

```
77                  //Operation 2: relabel, adding the needed vertices
78                  //to relabelQueue.
79                  new_height = MAX_INT;
80                  new_current_edge = 0;
81                  for each (edge out of vertex) {
82                      w = vertices[edge.v_out];
83                      if (remaining capacity of edge > 0 and
84                          w.height < new_height) {
85                          new_height = w.height;
86                          new_current_edge = edge; //We want the current edge
87                          //to be set too
88                      }
89                      //Now check if we have to add w to the relabelQueue
90                      if (w is not s or t and
91                          w.height == vertex.height + 1 and
92                          w.current_edge == edge.reverse_edge) {
93                              add w to relabelQueue;
94                      }
95                  }
96                  if (new_height == MAX_INT)
97                      new_height = n;
98                  else
99                      newHeight++;
100                 vertex.current_edge = new_current_edge;
101                 //We might have relabels with inactive vertices
102                 was_active = vertex.excess > 0;
103                 if (was_active) {
104                     activeHeights[height].pop();
105                     if (activeHeights[height].isEmpty()) {
106                         stack.pop();
107                     }
108                 } else {
109                     remove vertex from nonActiveHeights[height];
110                 }
111                 old_height = vertex.height;
112                 vertex.height = new_height;
113                 //Check for gaps
114                 if (activeHeights[old_height] and
115                         nonActiveHeights[old_height] are empty) {
116                     vertex.height = n;
117                     for each (Vertex w with
118                             old_height < w.height <= highestNonActiveHeight) {
119                         remove w from nonActiveHeights[w.height];
120                         w.height = n;
121                     }
122                     highestNonActiveHeight = old_height - 1;
123                     clear relabelQueue;
124                 } else if (new_height < n) {
125                     if (was_active) {
126                         add vertex to activeHeights[new_height];
```

```
127                         stack.push(new_height);
128                     } else {
129                         add vertex to nonActiveHeights[new_height];
130                     }
131                     highestNonActiveHeight = new_height;
132                 }
133             } //end of relabelQueue loop
134         }
135         //We might have to do more pushes, or find a valid current_edge
136     } while (vertex.height does not change and vertex.current_edge is not valid));
137 }
138 return calculateMaxFlowValue(); //Same as in FIFO version
```

7 Networks and results

We did a computational investigation to explore the practical efficiency of the algorithms that we discussed. We tested the following ones:

- Edmonds-Karp (EdmondsKarp)
- Scaling Edmonds-Karp (ScalingEK)
- Dinic's algorithm (Dinics)
- FIFO push-relabel with global relabelling (FIFO)
- FIFO push-relabel with global and gap relabelling (FIFO_Gap)
- Highest vertex push-relabel with global and gap relabelling, pointer selection (HV)
- Highest vertex push-relabel with exact labelling and gap relabelling, pointer selection (HV_Ex)
- Highest vertex push-relabel with global and gap relabelling, stack selection (HV_St)
- Highest vertex push-relabel with exact labelling and gap relabelling, stack selection (HV_Ex_St)

We used two different types of test networks. The first type are the AK networks described in the paper by Cherkassky and Goldberg [3], where they did a similar computational investigation as ours. The other networks are acyclic dense networks that were used in the famous DIMACS first competition [10].

7.1 AK Networks

AK networks were first described in the paper of Cherkassky and Goldberg [3]. They are designed to force a quadratic number of pushes with respect to the number of vertices of the network, both for FIFO and highest vertex selection, if the initial exact labelling is used. They also force a quadratic number of BFS phases in Dinic's algorithm. We have chosen them to compare the algorithms that we studied under a similar theoretical complexity lower bound. We use $\Omega(n)$ to refer to a lower asymptotic bound.

The original AK networks have two parallel modules, one that is hard for FIFO and highest vertex selection, called N1, and another one, called N2, that is hard for another type of vertex selection called WAVE [2]. Because we have not studied the WAVE implementation, we will only work with the N1 module, that has the following shape:

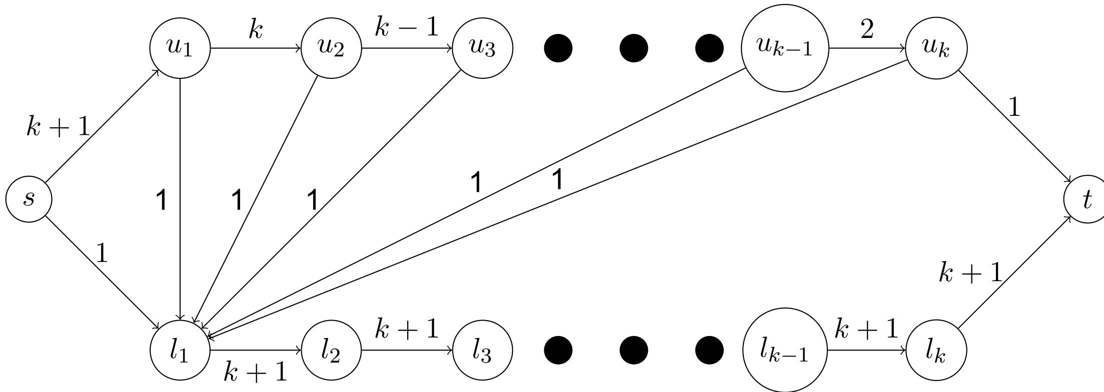


Figure 7.1: AK network, node names

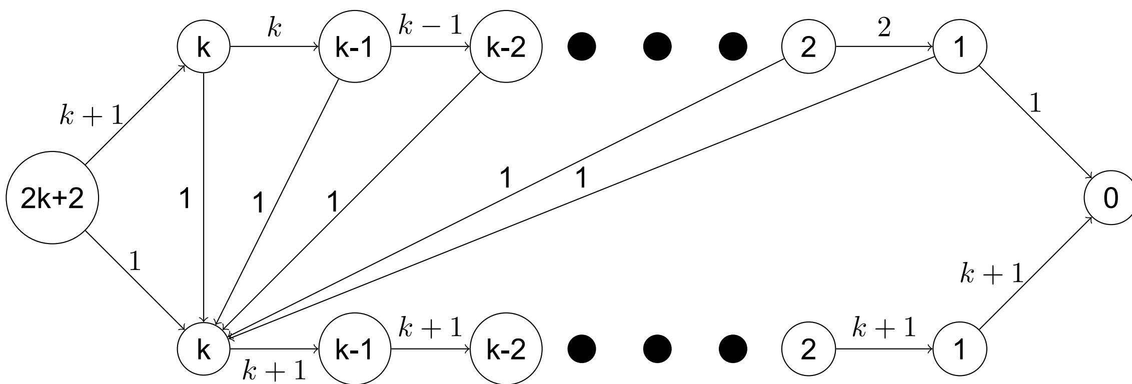


Figure 7.2: AK network, with the initial exact labelling shown on the vertices.

The size of the network is parameterised with the number of vertices in the upper path, from u_1 to u_k. The lower path has the same number of vertices, and so the total number of vertices is $2k + 2$. We shall show now how the different methods behave under this kind of network. All the explanations are based on the original paper by Cherkassky and Goldberg [3].

7.1.1 Complexity on highest vertex selection

To analyse the complexity of any highest vertex selection strategy for this network, we will define a *phase i* as the period from the time that (u_i, u_{i+1}) is first saturated to the time that (u_{i+1}, u_{i+2}) is first saturated. We shall show by induction that there are $k-1$ phases, and that in phase i there are at least $i + 1$ pushes.

Base case, phase 1: After the initialisation, u_1 has $k+1$ units of excess and l_1 has 1 unit. It does not matter if l_1 is chosen for discharge before or after u_1. When u_1 is chosen, it pushes k units of excess to u_2 directly because $d(u_2) = d(u_1) - 1$. It saturates the edge, and the phase starts with u_1 having 1 unit of excess. Then, u_1 is relabelled to $d(l_1)+1$ and pushes its last unit of excess (one push), and u_1 becomes inactive. Both u_2 and l_1 are active, but l_1 has a higher distance, so it is chosen for discharge and pushes its excess to l_2 (two pushes). After this, u_2 might be chosen for a push of $k-1$ units (three pushes) to u_3 and the next phase would start.

Hypothesis: In phase i, there are at least $i + 1$ pushes. Also, the vertex u_i is raised to distance $d(l_1) + 1$, and the lower vertices do not change their label. The phase finishes with a push from u_{i+1} to u_{i+2} that leaves u_{i+1} with one unit of excess.

Induction step: Consider phase $i + 1$. All vertices $u_1, \ldots, u_i$ have a distance label of $d(l_1)+1$, and all the vertices in the lower path have the same distance as in the beginning. Vertex u_{i+1} has 1 unit of excess, because the phase started with a saturating push from u_{i+1} to u_{i+2}, and u_{i+1} had $k-i+1$ units of excess. Now, it pushes its last excess to l_1 (first push) and becomes inactive. Vertex u_{i+2}, with distance label $k-1-i$, is lower than all the vertices $l_1, \ldots, l_{i+1}$. l_1 has one unit of excess, that will be pushed to l_2, and subsequently to $l_3, \ldots, li + 2$. At that point, $i+1$ pushes have happened on the lower path, plus the first push, giving $i + 2$ pushes in total.

In the last phase, u_k will push to t and the last unit of excess will flow through all the lower path until t. At that point, the algorithm will terminate. We have in total k phases:

$$\sum_{i=1}^{k}(i + 1) = \frac{1}{2}k(k + 3) = \Omega(k^2) \text{ pushes.}$$

Note that all the pushes that are performed on the lower path are non-saturating pushes, until the last phase.

7.1.2 Complexity on FIFO selection

To analyse the behaviour of the FIFO algorithm, we recover the concept of passes through the queue: pass $i + 1$ is defined as the sequence of consecutive pushes that happen on the vertices that were added on pass i, and the first pass is the sequence of pushes on the active vertices that were added during the initialisation.

To see the quadratic behaviour, we will focus on the first k passes. Let p be an integer between 1 and $k/2$. By induction, we will show that after $2p$ passes, the FIFO queue contains $p+2$ vertices. The trace of the algorithm differs if the initialisation first pushes on u_1 or on l_1. We show the case when u_1 is first in the queue, the other one is very similar.

Base case: $p = 1$. The two first passes are as follows. First, u_1 pushes to u_2 and adds it to the queue, and then it is relabelled and pushes to l_1 its last unit of excess. Then l_1 pushes two units of excess to l_2 and adds it to the queue. In the second pass, u_2 pushes to u_3 and to l_1 and adds them to the queue, and then l_2 pushes to l_3 and adds it too. The queue after the two first passes is $[u_3, l_1, l_3]$, which has $p + 2 = 3$ vertices.

Hypothesis: After 2p passes, we have p+2 vertices in the queue, which are

$$[u_{2p+1}, l_1, l_3, \ldots, l_{2p+1}].$$

Induction step: Consider the queue after 2p passes. u_{2p+1} pushes to $u_{2(p+1)}$ and to l_1 and adds $u_{2(p+1)}$ to the queue. The other vertices on the lower path push to their neighbors, and the queue ends up in this shape: $[u_{2(p+1)}, l_2, l_4, \ldots, l_{2(p+1)}]$. After this pass, $u_{2(p+1)}$ pushes to $u_{2(p+1)+1}$ and to l_1, and adds both to the queue. The even indexed vertices in the lower path push to their neighbors, and the queue finishes in the form: $[u_{2(p+1)+1}, l_1, l_3, \ldots, l_{2(p+1)+1}]$, which is what we wanted to show.

We have a linear number of vertices in each pass and there is a linear number of passes, therefore the total number of pushes is quadratic in k.

If l_1 was chosen as first vertex instead of u_1, it is possible to show with a similar argument as above that the order of the vertices in the queue is altered, but the growth of the queue is the same.

7.1.3 Complexity on Dinic's algorithm

When we run Dinic's algorithm on this network, we are forced to do $k+1$ BFS phases. Consider the first BFS: the upper path and the lower path are the shortest $(s \to t)$-paths, and we can only send 1 unit of flow through them. The edges $(u_k \to t)$ and $(s \to l_1)$ become saturated. The reverse edges of them are never contained in a shortest augmenting path, and therefore the forward edges will stay saturated until the end.

Now, the only shortest paths that exist in the algorithm go from s to the lower path, then to the lower path via an edge of capacity 1, and then to the sink traversing the entire lower path. Each time that one of these paths is used, the edge that connects the upper and lower path is saturated: first from u_1 to l_1, then from u_2 to l_2, and so on. Furthermore, the paths get longer and longer, and a BFS is needed after one augmenting path is used. Therefore, we have k BFS phases more, and the complexity goes to $\Omega(km)$.

7.2 Acyclic networks

The acyclic networks are the same that were used on the first DIMACS competition [10]. They are fully dense networks, and so they contain many more edges than the previous networks. The generator was taken from the repository of the DIMACS project.

7.3 Computational results

7.3.1 AK networks

All push-relabel algorithms are run with the initial exact labelling.

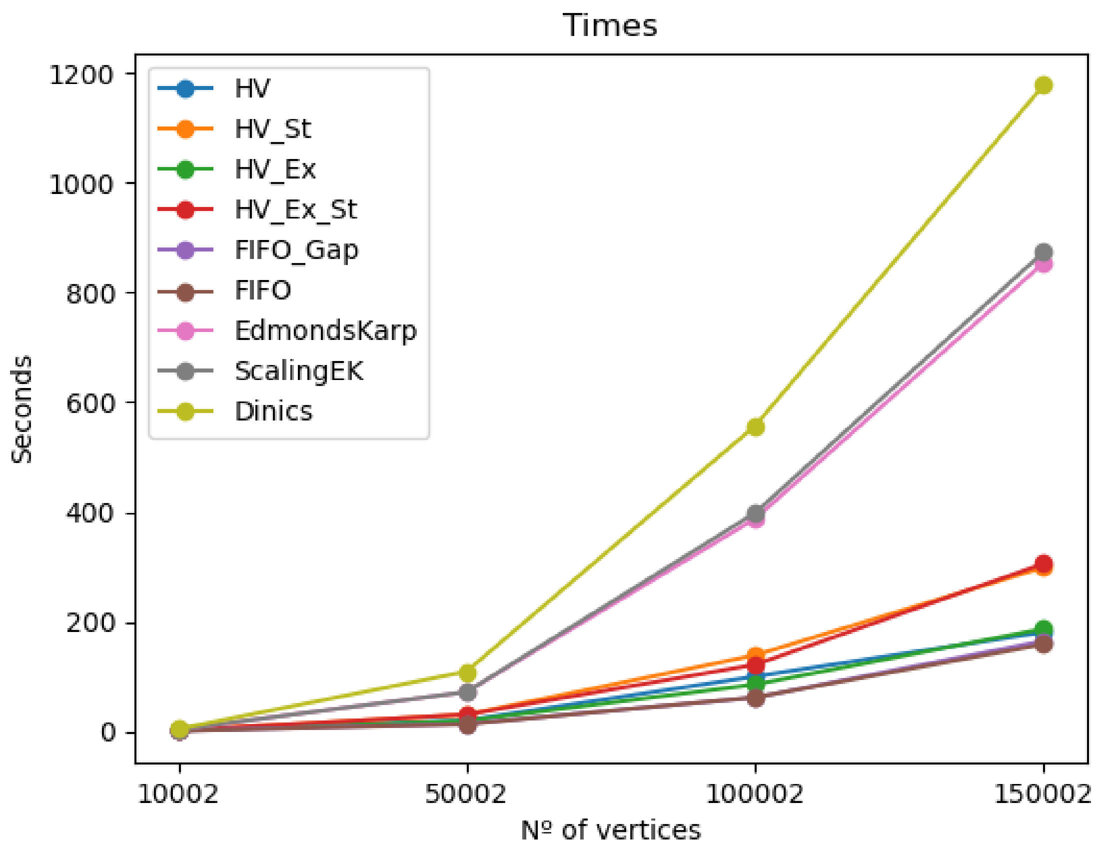


Figure 7.3: Runtimes of all the algorithms under AK networks in seconds.

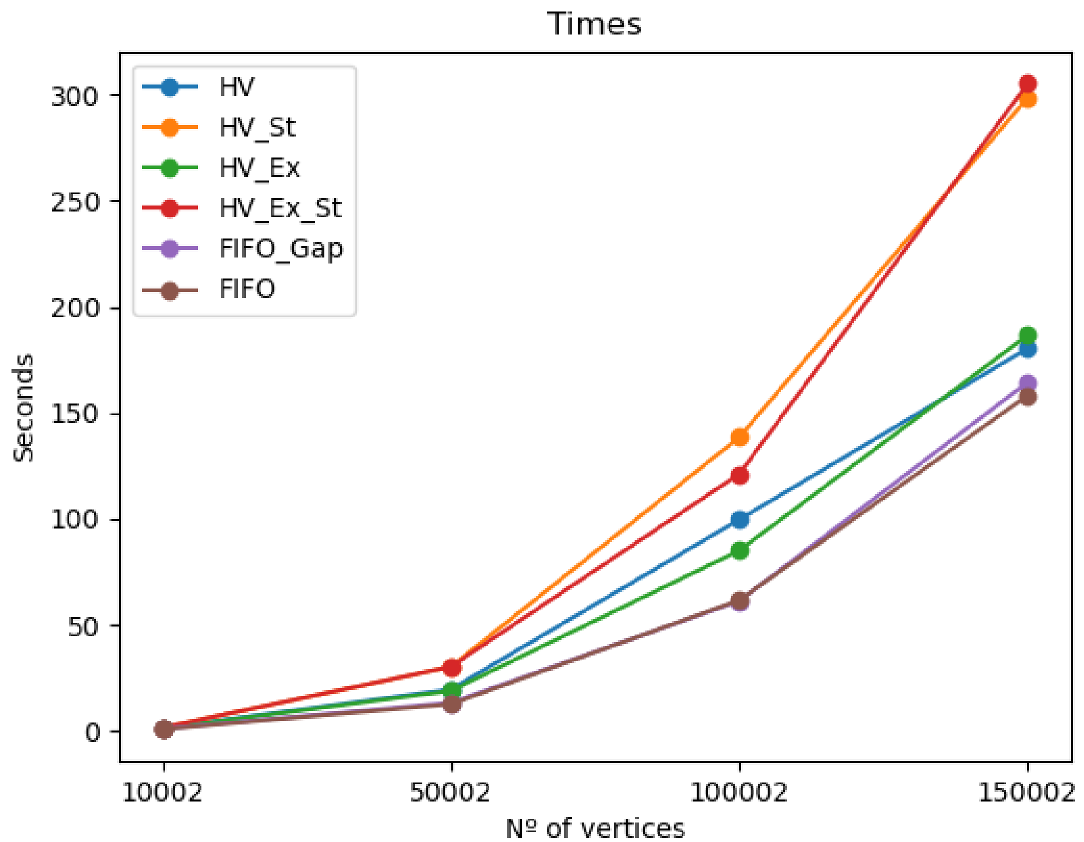


Figure 7.4: Runtimes of push-relabel algorithms only, in seconds.

From these plots we can see that the algorithms show indeed a roughly quadratic behaviour with respect to the number of vertices of the AK network, and that the push-relabel algorithms are still superior to all of the augmenting path methods.

It is interesting to see how Dinics algorithm performs worse than Edmonds Karp, when both find the same augmenting paths (both find the same shortest augmenting paths). This might just be due to how we have implemented Dinic's algorithm: after a BFS search, only the distance information is kept, and then a DFS search is carried out to find augmenting paths. Because there is only one augmenting path per BFS search, the DFS phase is only wasting time: we could have found the augmenting path if we had stored the backtrack information of the BFS. This could be an improvement to our implementation of Dinic's algorithm, that would be good in practice, and that would not hurt the performance of the algorithm in other cases.

Secondly, in the figure that focuses on the push-relabel implementations, we can see that the two FIFO implementations beat the highest vertex selection algorithms in all cases (although not by much). This differs from the results of the original authors of the AK networks, which also tested implementations of FIFO, FIFO_Gap and HV [3]. This might hint that our implementation of the highest vertex can be tweaked and perfected even more. Another, perhaps less exciting explanation, is that it could be due to the change of programming language.

We also see that the exact labelling versions of the algorithms do not improve or worsen

the performance, but surprisingly, the stack version of the algorithm displays a way worse performance than its counterpart, the pointer version. The difference is big enough to be impressive, but we shouldn't be surprised that the stack version performed worse in general. The times that the index selection has to iterate through the array are limited when we use gap relabelling, and even when its necessary to do it, the algorithm just has to iterate through a contiguous array in memory, which is a very fast operation on modern computers. On the other hand, managing a sparse memory data structure like a linked list could have a penalty, even if the time spent on it is asymptotically negligible.

7.3.2 Acyclic dense networks

These acyclic fully dense networks are crafted to have a number of edges that scales quadratically with the number of vertices. Edmonds-Karp's algorithm, and its scaling version, suffer particularly with these networks:

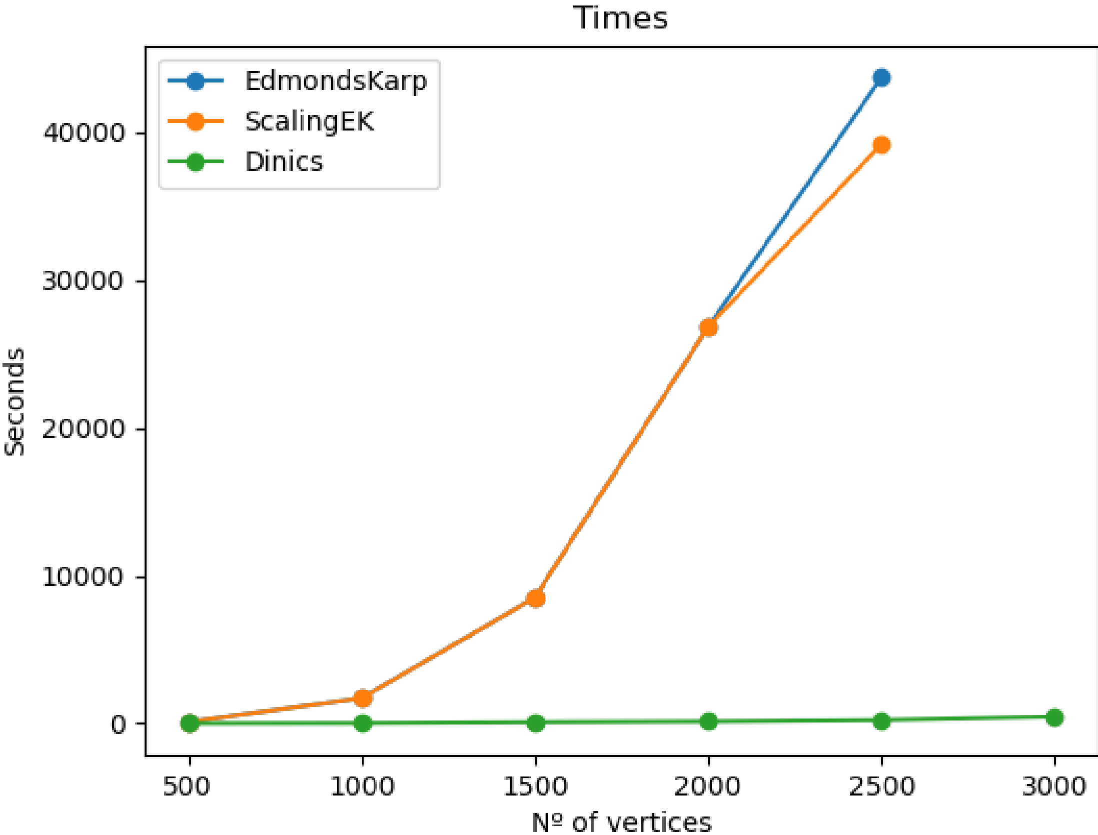


Figure 7.5: The three augmenting path algorithms on fully dense networks.

We can see that Dinic's better worst-case bound really pays off in comparison to the other augmenting path algorithms. However, it is still far from achieving the runtime of push-relabel algorithms:

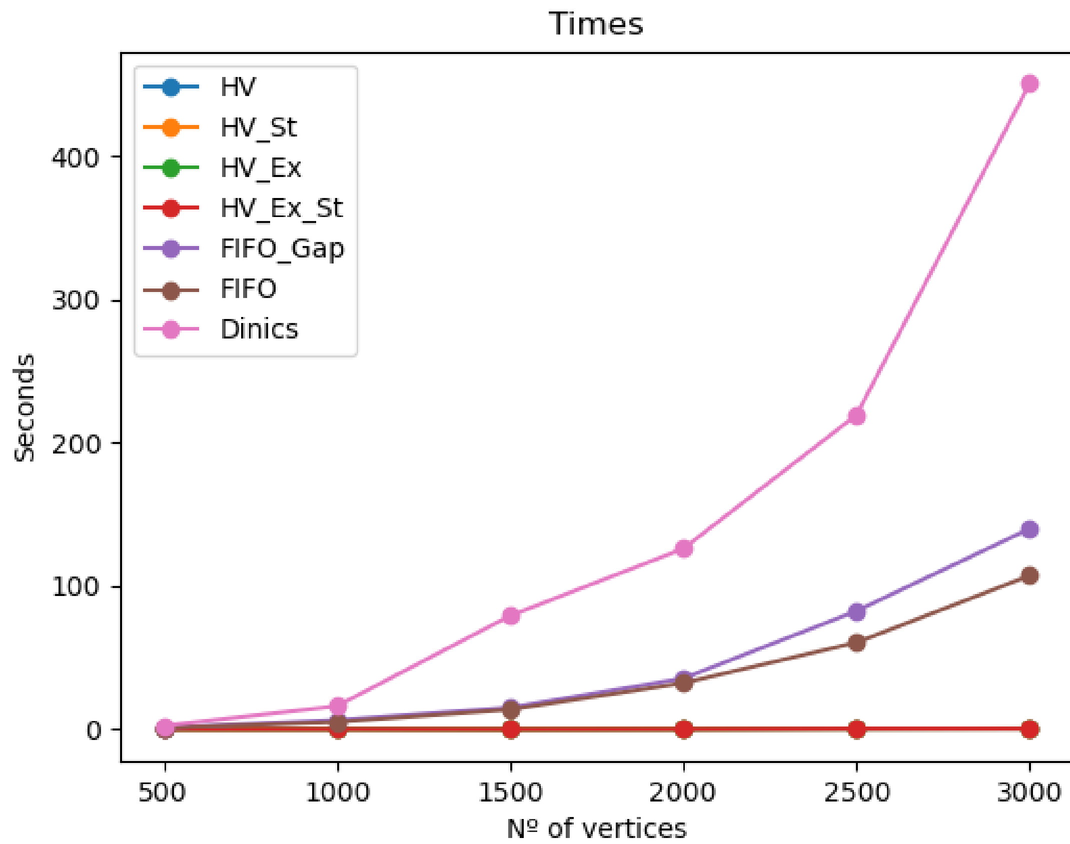


Figure 7.6: Push-relabel and Dinic's algorithms, on fully dense networks.

And among the push-relabel algorithms, all highest vertex strategies defeat undoubtedly the FIFO strategy, so much that we cannot distinguish them in the above plot. Below we have a comparison between the highest vertex algorithms:

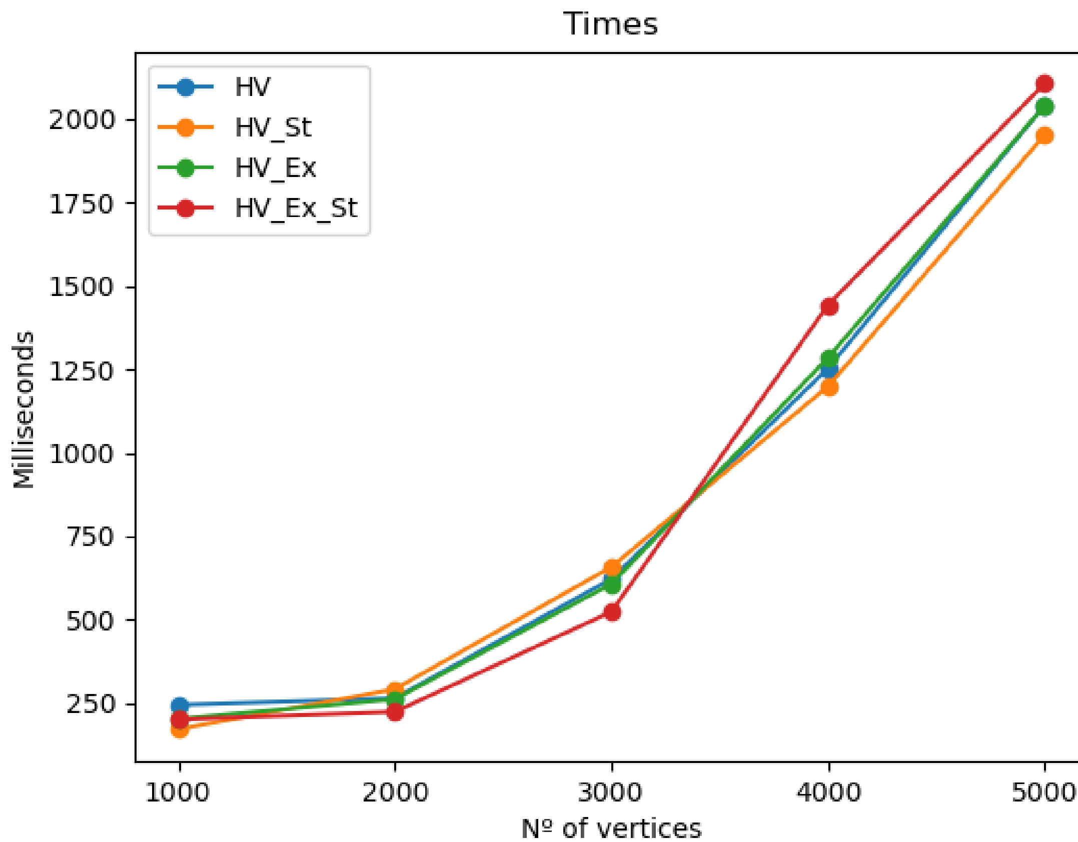


Figure 7.7: Highest vertex push relabel algorithms on fully dense networks.

It is very interesting to see that the stack version of the algorithms is the best with exact relabelling up until 3000 vertices, and afterwards the dominating one is the global relabelling with stack. However, all four versions display the same asymptotic behaviour, and the difference between them is in terms of milliseconds.

If we were to compare the worst-case bound for both FIFO and highest vertex selection in fully dense networks, we would get the same result: $m = O(n^2)$, so in these networks $O(n^2m^{1/2}) = O(n^3)$. However, we can see that the complexity of the highest vertex method in these fully dense networks is far below the worst case bound that we derived from it. In fact, from the above plot all highest vertex algorithms show an approximate linear growth after 3000 vertices.

Bibliography

[1] J. Cheriyan and S. N. Maheshwari. “Analysis of Preflow Push Algorithms for Maximum Network Flow”. In: *SIAM Journal on Computing* 18.6 (Dec. 1989), pp. 1057–1086. DOI: 10.1137/0218072. URL: https://doi.org/10.1137/0218072.

[2] B. V. Cherkassky. *A Fast Algorithm for Computing Maximum Flow in a Network, in Selected Topics in Discrete Mathematics: Proceedings of the Moscow Discrete Mathematics Seminar, 1972–1990*. en. Vol. 158. American Mathematical Society Translations: Series 2. American Mathematical Society, Feb. 1994. ISBN: 9780821875094. DOI: http://dx.doi.org/10.1090/trans2/158. URL: http://www.ams.org/trans2/158 (visited on 06/18/2021).

[3] B. V. Cherkassky and A. V. Goldberg. *On Implementing the Push–Relabel Method for the Maximum Flow Problem*. 1997.

[4] U. Derigs and W. Meier. “Implementing Goldberg’s max-flow-algorithm ? A computational investigation”. In: *ZOR Zeitschrift f□r Operations Research Methods and Models of Operations Research* 33.6 (Nov. 1989), pp. 383–403. DOI: 10.1007/bf01415937. URL: https://doi.org/10.1007/bf01415937.

[5] E. A. Dinitz. “Algorithm for Solution of a Problem of Maximum Flow in Networks with Power Estimation”. In: *Soviet Math. Dokl.* 11 (1970), pp. 1277–1280.

[6] J. Edmonds and R.M. Karp. “Theoretical improvements in algorithmic efficiency for network flow problems”. In: *J. Assoc. Comput. Mach.* 19 (1972), pp. 248–264.

[7] Andrew V. Goldberg and Robert E. Tarjan. “A new approach to the maximum flow problem”. In: *Proceedings of the 18th ACM Symposium on Theory of Computing* (1986), pp. 136–146.

[8] Andrew V. Goldberg and Robert E. Tarjan. “A new approach to the maximum flow problem”. In: *JOURNAL OF THE ACM* 35.4 (1988), pp. 921–940.

[9] Philip Bille Inge Li Gørtz. *Algorithms and Data Structures 2 at DTU, lecture notes*. 2020.

[10] David S. Johnson, Catherine C. McGeoch, and DIMACS (Group), eds. *Network flows and matching: first DIMACS implementation challenge*. DIMACS series in discrete mathematics and theoretical computer science v. 12. Providence, R.I: American Mathematical Society, 1993. ISBN: 9780821865989.

[11] Alexander Karzanov. “Determining the maximal flow in a network by the method of preflows”. In: *Doklady Mathematics* 15 (Feb. 1974), pp. 434–437.

[12] Jon Kleinberg and Éva Tardos. *Algorithm Design*. Pearson Education, 2006. ISBN: 978-0321295354.

[13] Jr. L. R. Ford and D. R. Fulkerson. “Maximal flow through a network”. In: *Canadian Journal of Mathematics* 8 (1956), pp. 399–404.

[14] Ravindra K. Ahuja; Thomas L. Magnanti and James B. Orlin. *Network Flows: Theory, Algorithms and Applications*. 1993. ISBN: 978-0136175490.

[15] John Adrian Bondy; U.S.R. Murty. *Graph Theory With Applications*. ISBN: 978-0444194510.

[16] Frank Nielsen. *Graph Theory - Algorithms and Netowrks*. Department of Mathematics, Technical University of Denmark, 1996.

[17] James B. Orlin and Xiao-Yue Gong. *A Fast Max Flow Algorithm*. 2019. arXiv: 1910.04848 [cs.DS].

[18] Tim Roughgarden. *A Second Course in Algorithms (Stanford CS261, Winter 2016) - YouTube*. URL: https://www.youtube.com/playlist?list=PLEGCF-WLh2RJh2yDxlJJjnKswWdoO8gAc (visited on 06/18/2021).
[19] WilliamFiset. *Dinic's Algorithm | Network Flow | Graph Theory*. en. URL: https://www.youtube.com/watch?v=M6cm8UeeziI (visited on 06/18/2021).

A Java code

The author's code can be found in the repository for "Algorithm Compare": https://github.com/MrMCMax/algorithm_compare. It has as a dependency the more general data structures repository: https://github.com/MrMCMax/data_structures, that contains several data structures that the author has been compiling over the years. The project uses mainly the graph classes and the special data structures like the eraser linked list.

The *algorithm_compare* project has a very basic interface to run some test networks with a selection of algorithms, in the *Launcher* app. The code for the algorithms is found under *src/main/java/algorithm_compare/logic/algorithms*. It has many versions of the algorithms that were discussed in this thesis. The versions that are revised and that we used are in the following classes:

- Edmonds-Karp: EdmondsKarp.java
- Scaling Edmonds-Karp: ScalingEdmondsKarp.java
- Dinic's algorithm: DinicsAlgorithm.java
- FIFO push-relabel with global relabelling: FIFOPushRelabelVertexGlR2.java
- FIFO push-relabel with global and gap relabelling: FIFOPR_Gl_Gap3.java
- Highest vertex push-relabel with global and gap relabelling, pointer selection: HighestVertexGapRelabelling3.java
- Highest vertex push-relabel with exact labelling and gap relabelling, pointer selection: HV_Gap_Gl_True_Exact_Stack.java
- Highest vertex push-relabel with global and gap relabelling, stack selection: HighestVertexGapRelabellingStack.java
- Highest vertex push-relabel with exact labelling and gap relabelling, stack selection: HV_Gap_Gl_True_Exact_Stack.java

None of these push-relabel algorithms, except *FIFOPushRelabelVertexGlR2*, use the initial exact labelling. In the folder there can be found versions of them with the same name and "_Init" at the end, that denote the versions with an initial exact labelling.

Technical
University of
Denmark

Richard Petersens Plads, Building 324
2800 Kgs. Lyngby
Tlf. 4525 1700

https://www.compute.dtu.dk/english