UCL Université catholique de Louvain

École polytechnique de Louvain (EPL)

# Study and Analysis of Network Flows

## Augmenting Path and Preflow-push algorithms

Dissertation presented by
**Denis GENON , Victor VELGHE**

for obtaining the Master's degree in
**Sciences Informatiques**

Supervisor
**Yves DEVILLE**

Readers
**François AUBRY, Jean-Charles DELVENNE**

Academic year 2015-2016

## Acknowledgment

We would like to thank our supervisor Pr. Deville for his releveant remarks and wise advices. Furthermore, we thank him for teaching the course Artificial Intelligence which made us discover our passion for algorithms.

We would also thank François Aubry, PhD student, for his support, patience, enthusiasm and bright ideas. His guindance helped us for our research and writing this master thesis. He's the reason we produced an accomplished work.

Our thanks also goes to Pr. Jean-Charles Delvenne to accept reading our master thesis.

We are also grateful to our wonderful famillies and friends, always behind us and supporting us.

# Contents

# Introduction

The maximum flow problem is an optimization problem which takes place in the graph theory. This problem is noteworthy by the long succession of research contributions that have improved on the worst-case complexity of the best known algorithms. Among those best known algorithms, we can cite, in the order of creation, the Ford-Fulkerson algorithm (1955), the blocking flow of Dinitz (1970), the Edmonds-Karp algorithm (1972), the push-relabel algorithm of Goldberg and Tarjan (1986) and the binary blocking flow of Goldberg and Rao (1997).

The goal of this master thesis is to do an analysis of how the augmenting path algorithms and the preflow-push algorithm, two families of maximum flow algorithm, perform in different families of graphs. We also studied which data structure was the most suited to represent a graph. Another goal of this work is to present an open-source and modulable Java implementation of these algorithms. With this implementation, we want to allow other developers to try our implementation and extend it.

Our work is divided into several parts. First, we introduce the problem studied and we define the notations that we will use throughout this work. Then we will define the algorithms and data structures that we used for our analysis. After, we show the improvements on these algorithms to make them more efficient. Then will come the part where we explain our choices and the structure of the implementation. Finally, the experimental analysis will conclude our work.

# Chapter 1

# The Maximum Flow Problem

The maximum flow problem can be stated as follows: in a capacitated network, we need to push as much flow as we can between two specials vertices, the source and the sink. The two constraints are that we cannot exceed the capacity of any edge and a vertex cannot hold flow. A common analogy is a water distribution in a country. The vertices can be sources of water, cities needing water or just some transfer nodes. The edges can be viewed as pipes with a maximal volume of water per second (or capacity). The flow is the amount of water flowing through the pipes. The goal is to find the maximum flow of water that can be send from a source of water to a needing city given the capacities of each pipes. As we need to find the maximum flow that can be pushed, this an optimization problem. We will see in the section Applications 1.4 that the maximum flow problem is important because it can be used to express and resolve a wide variety of different kinds of problems.

To solve this problem, two families of algorithms have been developped these last decades. The first one is the family of the augmenting path algorithms. The aim of thoses algorithms is to find an augmenting path in the residual network until it becomes impossible. We will return on these principles later. The second family is the family of preflow-push algorithms. The aim of those is to flood the network to quickly find the maximum flow.

## 1.1 Notations and Definitions

In this section, we will set the conventions that we will use along this master thesis. This problem is based on the graph theory, we will then use the definitions of this theory to frame the problem.

### 1.1.1 Capacitated network

**Definition 1.** A **directed graph** $G = (V, E)$ consists of a set $V$ of vertices and a set $E$ of edges. We assume, without loss of generality, that $V = \{0, 1, ..., n-1\}$, where $n$ is the number of vertices. The edges are represented as an ordered pair of its vertices and is, $E \subseteq V \times V$.

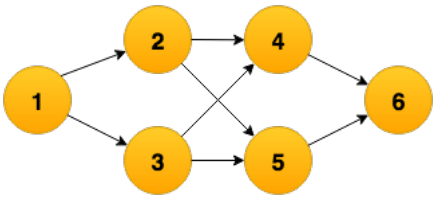



Figure 1.1: A directed graph.

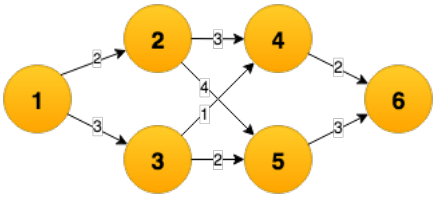



Figure 1.2: A capacitated network.

Figure 1.1 gives us an example of a directed graph.

**Definition 2.** A **capacitated network** is a directed graph whose edges have a numerical value associated to it. In the case of the maximum flow problem, this value represents the capacity of the edge. The **capacity** of an edge is a mapping $c : E \to \mathbb{R}^+$, and represents the maximum amount of flow that can pass through the edge. The capacity of the edge $(i, j)$ is denoted $c_{ij}$.

The figure 1.2 gives us an example of a capacitated network.

In this master thesis, everything that will be mentionned by the terms *graph*, *network*, ... will refer to a *capacitated network*.

### 1.1.2 Source and sink vertices

As said earlier, the maximum flow problem distinguishes two specials nodes in the problem: the source and the sink.

**Definition 3.** A **source** is an arbitrary vertice $s$ which possesses at least one outgoing edge.

$$s \in V \text{ such that } \exists (s, k) \in E \text{ with } k \in V$$

**Definition 4.** A **sink** is an arbitrary vertice $t$ which possesses at least one incoming edge.

$$t \in V \text{ such that } \exists (k, t) \in E \text{ with } k \in V$$

In this master thesis, we will use the letters $s$ and $t$ to represent the source and the sink, respectively.

### 1.1.3 Flow

**Definition 5.** A **flow** is a mapping $f : E \to \mathbb{R}^+$. The flow between $i$ and $j$ is denoted by $f_{ij}$ and is subject to two constraints:

**Capacity constraint** $f_{ij} \leq c_{ij}$, for each $(i,j) \in E$;

**Flow conservation constraint** $\sum_{(i,j)\in E} f_{ij} = \sum_{(j,i)\in E} f_{ji}$ , for each $j \in V \setminus \{s,t\}$.

**Definition 6.** The **value of a flow** is defined by $|f| = \sum_{(s,j)\in E} f_{sj}$ or $|f| = \sum_{(i,t)\in E} f_{it}$. It represents the amount of flow passing from the source to the sink. $|f|$ can be equivalently stated as the amount of flow leaving $s$ or entering $t$.

### 1.1.4 Residual network

To resolve the maximum flow problem, it is convenient to use another representation of the capacitated network. To be able to find the maximum flow, several algorithms update the current flow in an incremental process. Then, it will be useful to have a representation of the graph which gives the amount of remaining flow which can be added to the current flow through an edge. This representation is called *residual network* [AMO93].

**Definition 7.** Given a flow $f$ in a graph $G$, the **residual capacity** $c_f(i,j)$ is defined as $c_f(i,j) = c_{ij} - f_{ij}$.

**Definition 8.** Given a flow $f$ in graph G, the **residual network** $G_f$ is the directed network (defined on the same set of vertices) with all edges of positive residual capacity, each one labeled by its residual capacity.

## 1.2 Assumptions

All along this master thesis, we will consider a capacitated network $G = (V, E)$ with a non negative capacity $c_{ij}$ associated with each edge $(i,j) \in E$.

All capacities are positives integers. With irrational flow values, the flow might not even converge towards the maximum flow [Zwi95].

There is no incoming edges to the source and no outgoing edges from the sink.

The network is connected. This assumption is important to ensure that there is a path bewteen the source and the sink. If no such path exist, the flow is simply 0.

The network does not contain any path from the source to the sink composed only of infinite capacity edges. The reason of this is that if a such path exist, we can send an infinite amount of flow along this path, and therefore the maximum flow value cannot be bounded.

The network does not allow multiples edges bewteen two same vertices. This assumption is not requiered if we consider that the capacities of those edges add up. This assumption allows us to keep the representation of the problem simple.

## 1.3 Problem statement

We have now defined all terms we will use and made all the assumptions needed. Thus we can now state the problem formally. The maximum flow problem is to maximize $|f|$, that is, to route as much flow as possible from s to t. The flow f must satisfy the capacity constraint and the flow conservation constraint at all vertices (except s and t). We can state the problem formally as follows.

$$\begin{aligned}
&\text{Maximize} \quad \sum_{(s,i)\in E} f_{si} \\
&\text{subject to} \\
&\qquad \sum_{(i,j)\in E} f_{ij} = \sum_{(j,i)\in E} f_{ji} \quad \text{for all } i \in V \setminus \{s,t\} \qquad (1.1) \\
&\text{and} \\
&\qquad 0 \leq f_{ij} \leq c_{ij} \qquad \text{for each } (i,j) \in E \\
&\qquad f_{ij} \text{ is an integer.}
\end{aligned}$$

## 1.4 Applications

As we said earlier, the maximum flow problem arise in a wide variety of situations and in several forms. The maximum flow problem is often a subproblem of other more difficult network problems. In this section, we will describe one application of the maximum flow problem [AMO93]. Several other applications can be found on the literature or on the web [Way01].

### 1.4.1 Problem of representatives

#### Problem Statement

A city has $c$ citizens $C_1, C_2, C_3, ..., C_c$; $a$ associations $A_1, A_2, A_3, ..., A_a$ and $p$ political parties $P_1, P_2, P_3, ..., P_p$. Each resident is member of at least one association and belongs to one political party. Each association must elect one of its members to represent it on the governement so that the number of council member of the political party $P_k$ is at most $u_k$. The task is to know if an arrangement is possible in regards to that property.

#### Solution

We can transform this problem to a directed network as shown in the figure 1.3. In the network, we added a source node and a sink node. There is an arc from the source node to each association node with an unitary capacity. If a citizen is member of an association, there is an arc from the

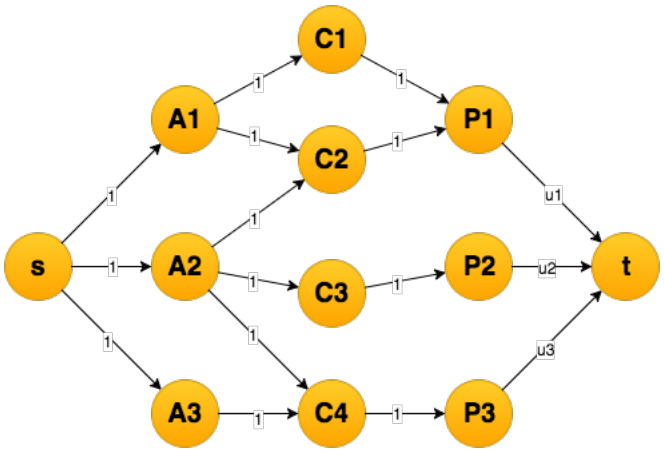



Figure 1.3: A directed network representing the problem of representatives.

assocation node $A_i$ to the citizen node $C_j$ with an unitary capacity. If a citizen is member of a political party, there is an arc from the citizen node $C_i$ to the political party node $P_j$ with an unitary capacity. Finally, each political party nodes have an outgoing arc to the sink, with a capacity of $u_k$ (defined in the problem).

If the maximum flow of this directed network is equals to the number of associations ($a$), we can say that the city has a balanced council; otherwise, it does not. This is because if the value of the maximum flow is not $a$, it means that at least one association is not represented.

This type of model has applications in several ressource assignment settings.

# Chapter 2

# Existing Algorithms

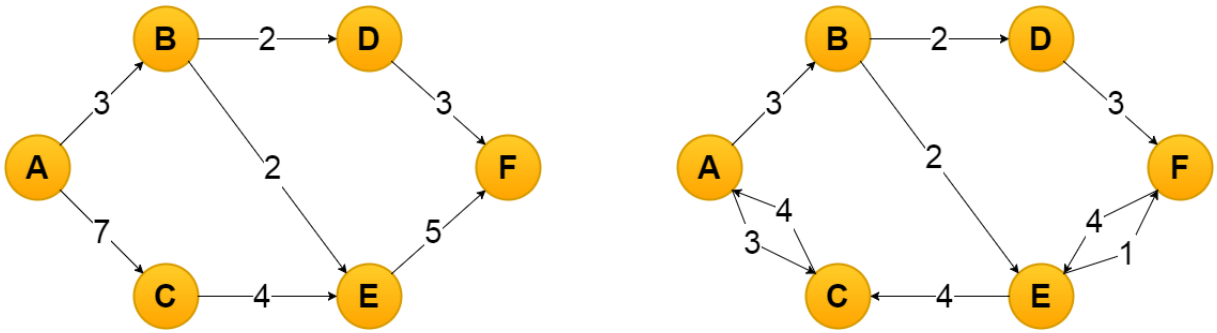
## 2.1 Augmenting path algorithms

The idea behind the augmenting path algorithms is as follows : as long as there is a path from the source to the sink in the residual network, we send flow along this path. This process is repeated until there is no more path from the source to the sink.

A directed path from the source to the sink in the *residual network* is called *augmenting path*.

When an *augmenting path* is found, we send a flow equivalent to the minimum capacity of any edge in the path. We update the *residual network* by decreasing capacities in forward edges and increasing capacities in backward edges. Then we look for a new augmenting path in the new *residual network*.

Here is a graph with its *residual network* after sending 4 units of flow through the *augmenting path* A-C-E-F :



The residual network is essential to be able to backtrack. For instance, after sending 4 units of flow through the path A-B-E-F in the graph $G1$, we obtain the graph $G2$. On this graph, it is not possible to find a new path from the source A to the sink F. But on its residual graph $G3$, we can send 2 units of flow through A-C-E-B-D-F to obtain the maximal flow represented in the graph $G4$.

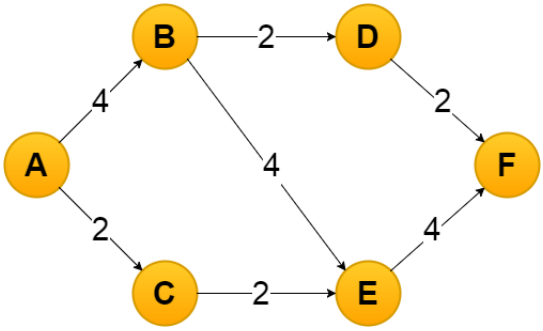



Figure 2.1: Graph $G1$.

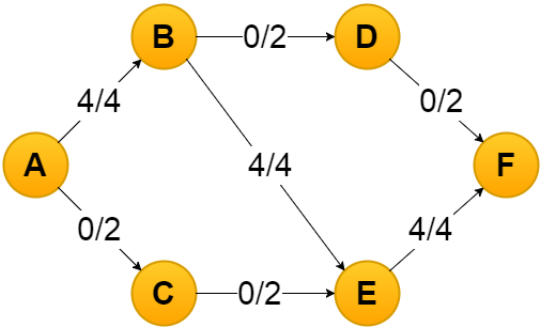



Figure 2.2: Graph $G2$.

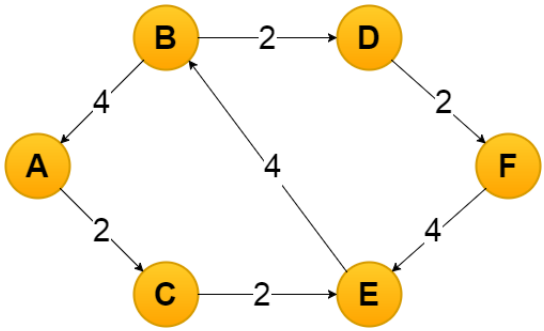



Figure 2.3: Residual graph $G3$.

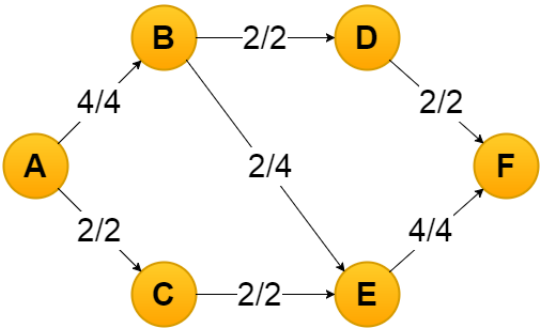



Figure 2.4: Graph $G4$.

The pseudo-code of the augmenting path algorithm is given here :

**while** $G_f$ *contains a directed path from vertex s to vertex t* **do**
    identify an augmenting path P from vertex s to vertex t;
    $\delta = \min\{c_{ij} : (i,j) \in P\}$;
    augment $\delta$ units of flow along P and update $G_f$;
**end**

**Algorithm 1:** Augmenting path algorithm.

### 2.1.1 Ford-Fulkerson and Edmonds-Karp

There are two main augmenting path algorithms, Ford-Fulkerson (published in 1956) and Edmonds-Karp (published in 1972). They instantiate in their own way the augmenting path algorithm given that the unique difference between both is the way of looking for an *augmenting path* in the *residual network*.

Ford-Fulkerson uses a depth-first search, looking for long paths [Sch15a]. The flow is thus potentially sent over a large number of edges but a small quantity. Indeed, a long path has a higher chance to contain at least one edge with a low capacity.

On the other hand, Edmonds-Karp uses a breadth-first search to look for the shortest path (in terms of number of edges) [Way15]. A bigger quantity of flow is thus sent on a smaller number of edges.

### 2.1.2 Complexities

The max flow problem, being a problem of complexity class P, can be solved in polynomial time. When the capacities are integers, the remaining time of Ford-Fulkerson is bounded by $O(|E| \cdot |V| \cdot U)$ and Edmonds-Karp by $O(|V| \cdot |E|^2)$, where $|E|$ is the number of edges in the graph, $|V|$ is the number of vertices and $U$ is the maximum capacity of the graph.

**Ford-Fulkerson** Each augmenting path can be found in $O(|E|)$ thanks to the depth-first search algorithm and in the worst case, the flow will increase by 1. So the time complexity is $O(|E| \cdot |f|)$, $|f|$ being the value of the maximal flow.

This complexity is expressed in terms of the final result, we can reformulate it. Indeed, the maximal flow cannot be greater than $|V| \cdot U$. The time complexity of Ford-Fulkerson is then $O(|E| \cdot |V| \cdot U)$.

Ford-Fulkerson has a pseudo-polynomial time complexity. Indeed, $O(|E| \cdot |V| \cdot U)$ is polynomial in the numeric value of the input but is exponential in the length of the input. The input takes $log(U)$ bits to represent $U$ [Sch15a].

**Edmonds-Karp** The breadth-first search assures us that after each iteration, the length of the augmenting path can't decrease [Kem04]. We also know that there is a maximum of $|E|$ path of the same length. So we conclude that the length of the augmenting path can stay the same for at most $|E|$ iterations before increasing.

We know that the length of the augmenting path is between 1 and $|V| - 1$. The length of the augmenting path increase by a least 1 so there is a maximum of $|V|$ possible increases. Thus there are at most $|V| \cdot |E|$ iterations.

Each augmenting path can be found in $O(|E|)$. The time complexity of Edmonds-Karp is then $O(|V| \cdot |E|^2)$ [Way15]. This upper bound complexity is very severe. Indeed, it is rare to find graphs with such features.

## 2.2 Preflow-push algorithms

### 2.2.1 Introduction

The drawback of the augmenting path algorithms is the operation of sending flow along a path. The time complexity of this operation in the worst case is $O(n)$, where $n$ is the number of vertices in the augmenting path. One example of this can be seen in the figure 2.5. With augmenting path algorithms, only one unit of flow can be pushed at a time on the edge $(12, 13)$ in the network. In this case, a better solution is to push directly from the vertice 12, 10 units of flow to the sink. This is the idea behind preflow-push algorithms [GT88].

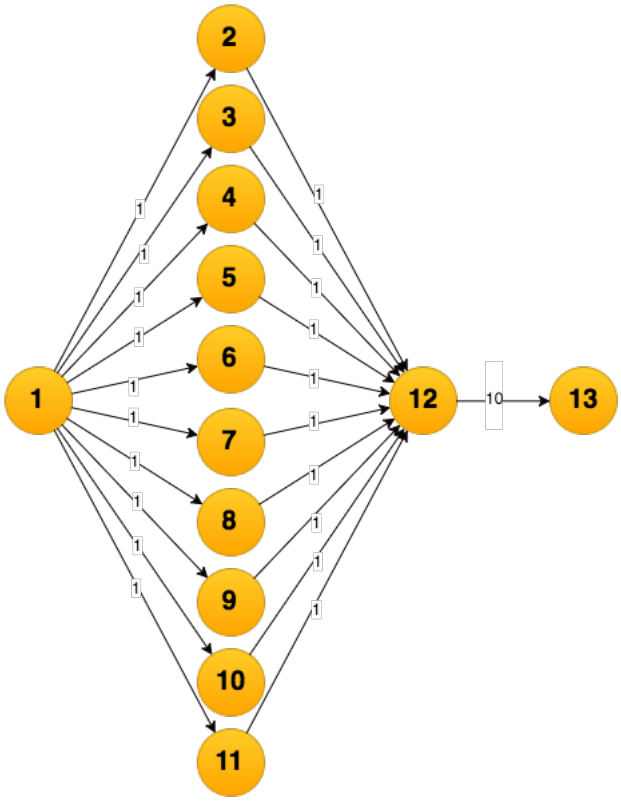



Figure 2.5: Extreme case for augmenting path algorithms.

The preflow-push algorithms don't push flow on augmenting path, they push flow on individual vertices. Because of this, preflow-push algorithms does not respect the flow conservation constraint. Indeed, these algorithms permit the flow entering a vertex to exceed the flow leaving the vertex. We call this flow a preflow.

**Definition 9.** A **preflow** is a function $x : E \to \mathbb{R}$ that satisfies the capacity constraint and the following relaxation of the flow conservation constraint:

$$\sum_{(j,i)\in E} x_{ji} - \sum_{(i,j)\in E} x_{ij} \geq 0 \text{ for each } i \in V \setminus \{s,t\}.$$

**Definition 10.** The **excess** of each vertex $i \in V$ is

$$e(i) = \sum_{(j,i)\in E} x_{ji} - \sum_{(i,j)\in E} x_{ij}.$$

We call a vertex with positive excess as an **active vertex**. The fact that there are active vertices means that the current solution is not feasible [CSRL01]. Thus, the goal of the algorithm is to remove the excess from those vertices. The intuition is to push the flow to the vertex closer to the sink. Each vertex has its own height label which is, in the beginning of the algorithm, the distance between the vertex and the sink. The algorithm only push a flow from a vertex to another if the first one is an active vertex and if his height label is greater than the height label of the other vertex. A popular analogy is that we can only push flow downhill. The difference between the two heights label must be exaclty one. When we can push a flow from a active vertex to another vertex, we call the edge linking the two an **admissible edge**. This operation is called a **push operation**. If such scenario cannot be applied, we need to relabel the height label of the vertex. This operation is called the **relabel operation**. The algorithm terminates when there is no more active vertex. The pseudo code of this algorithm is given in Algorithm 2 [AMO93].

```
preprocess;
while the network contains an active vertex do
    select an active vertex i;
    push/relabel(i);
end
```

**Algorithm 2:** Generic preflow-push algorithm.

```
x ← 0;
compute height labels d(i);
x_sj ← c_sj for each arc (s, j) ∈ E(s);
d(s) ← v;
```

**Algorithm 3:** Preprocess.

### 2.2.2 Complexity

To compute the complexity of the generic preflow-push algorithm we need to distinguish three kinds of operations: relabels, saturing pushes, non-saturing pushes.

**if** *the network contains an admissible edge* $(i, j)$ **then**
  push $\delta \leftarrow min\{e(i), r_{ij}\}$ units of flow from $i$ to $j$;
**end**
**else**
  replace $d(i)$ by $min\{d(j) + 1 : (i, j) \in E(i) \text{ and } r_{ij} > 0\}$;
**end**

**Algorithm 4:** Push/Relabel($i$).

- There is a possibility of maximum $|V|^2$ relabel operations. This is because the maximum height is $2 \cdot |V| - 1$ [AMO93] and we can increase the label height at minimum one per operation. We know that there is $|V| - 2$ vertices which can be relabeled (the sink and source vertices can't be relabeled). Then, the maximum number of relabelling is $(2 \cdot |V| - 1) \cdot (|V| - 2) = O(|V|^2)$.

- The number of saturing pushes per edge is $O(|V|)$. To make a saturing push on a certain edge again, the label height of the destination vertex must be augmented of 2. Then there is $O(\frac{2 \cdot |V| - 1}{2}) = O(|V|)$ saturing push per edge: the number of maximum saturing push is $O(|V| \cdot |E|)$.

- The number of non-saturing pushes is computed with $\phi$, the sum of the height labels of the actives vertices. The relabel increases $\phi$ at maximum $(2 \cdot |V| - 1) \cdot (|V| - 2)$ (the number of vertices times the maximum height label). The saturing push increase $\phi$ at maximum $(2 \cdot |V| - 1) \cdot (2 \cdot |V| \cdot |E|)$. As $\phi$ must be equals to zero at the end of the algorithm, we need at most $(2 \cdot |V| - 1) \cdot (|V| - 2) + (2 \cdot |V| - 1) \cdot (2 \cdot |V| \cdot |E|) = O(|V|^2 \cdot |E|)$ non-saturing push to have $\phi = 0$.

We have $O(|V|^2)$ relabel operations, $O(|V| \cdot |E|)$ saturing pushes and $O(|V|^2 \cdot |E|)$ non-saturing pushes. Thus, the generic preflow-push algorithm runs in $O(|V|^2) + O(|V| \cdot |E|) + O(|V|^2 \cdot |E|) = O(|V|^2 \cdot |E|)$.

## 2.3 Difficult cases

### 2.3.1 In Edmonds-Karp algorithm

In Figure 2.6, we show a graph where the Edmonds-Karp algorithm performs worse than the Preflow-push algorithm. The Edmonds-Karp algorithm will find $n + 1$ augmenting paths ($s$-$t$, $s$-1-$t$, $s$-1-2-$t$, ..., $s$-1-...-$n$-$t$). It will perfom slower than the Preflow-push algorithm because it will push $n$ times on the edge $(s, 1)$, $n - 1$ times on the edge $(1, 2)$, ... The Preflow-push will be faster, because he will only push once on each edges.

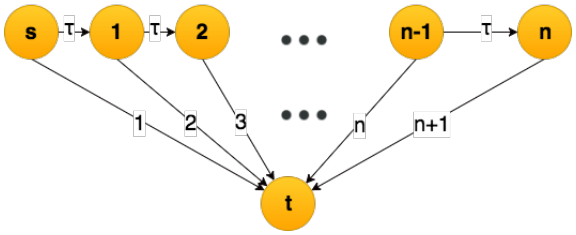



Figure 2.6: Difficult case for the Edmonds-Karp algorithm. $\tau$ is equal to $(\sum_{i=1}^{n+1} i) - 1$.

### 2.3.2 In Preflow-push algorithms

#### The ping-pong effect

The ping-pong effect is a phenomen that occurs in preflow-push algorithm that considerably slow it down. To describe the ping-pong effect [CGI08], we will execute the preflow-push algorithm on the network in Figure 2.7.

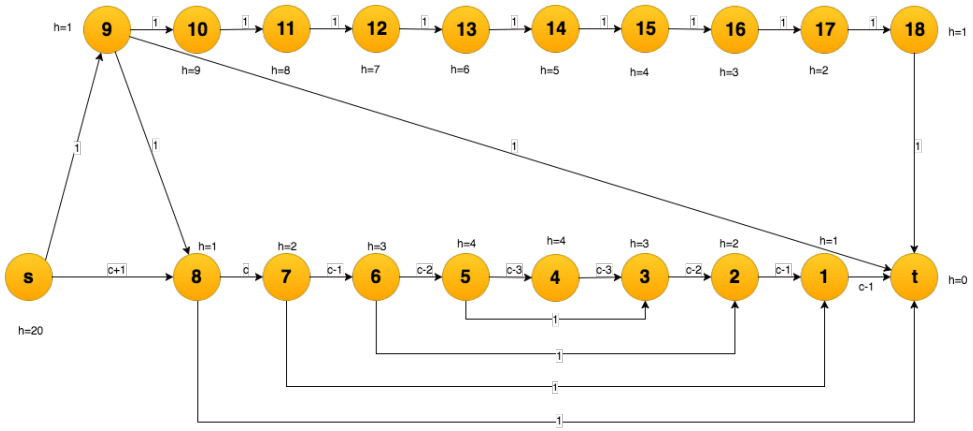



Figure 2.7: The initial graph. c is an integer greater than 4.

After the preprocess, we push a flow along the path 8-7-6-5-4-3-2-1-t. To be able to do this push, the vertices must be relabeled. We obtain the residual network given in the Figure 2.8 with the corresponding labels for each vertices. The vertex 1 is active. The algorithm relabels it with the height 3. Then, the flow is pushed along the path from the vertex 1 to the vertex 8, and relabel the height labels of the visited vertices. After thoses push and relabel operations, we obtain the residual network described in the Figure 2.9.

The algorithm will now select the vertex 8, relabels it to the height label 5 and push the excess flow from 8 to 7. The vertex 7 is selected, his height label modified to 6 and the excess flow is pushed from 7 to 6. The algorithm repeats those operations for each vertices along the path 6-5-4-3-2-1. We obtain now the residual network described in the figue 2.10.

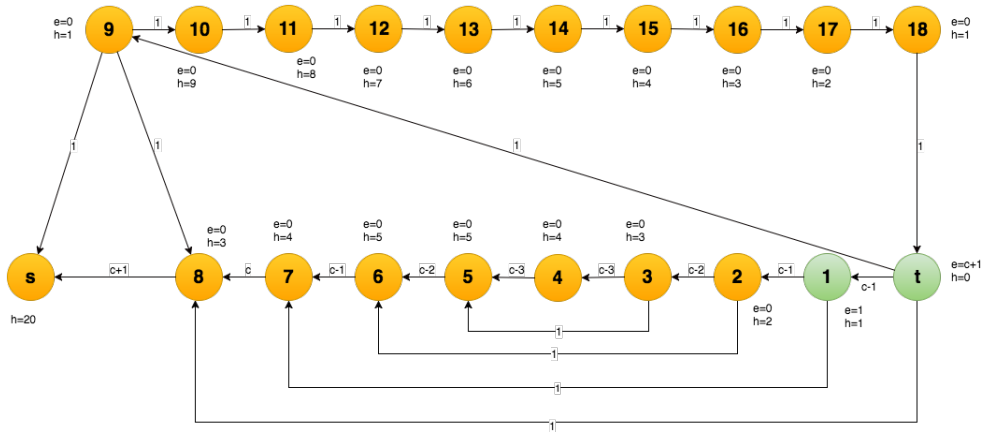



Figure 2.8: The residual network after some iterations. The only active node is node 1.

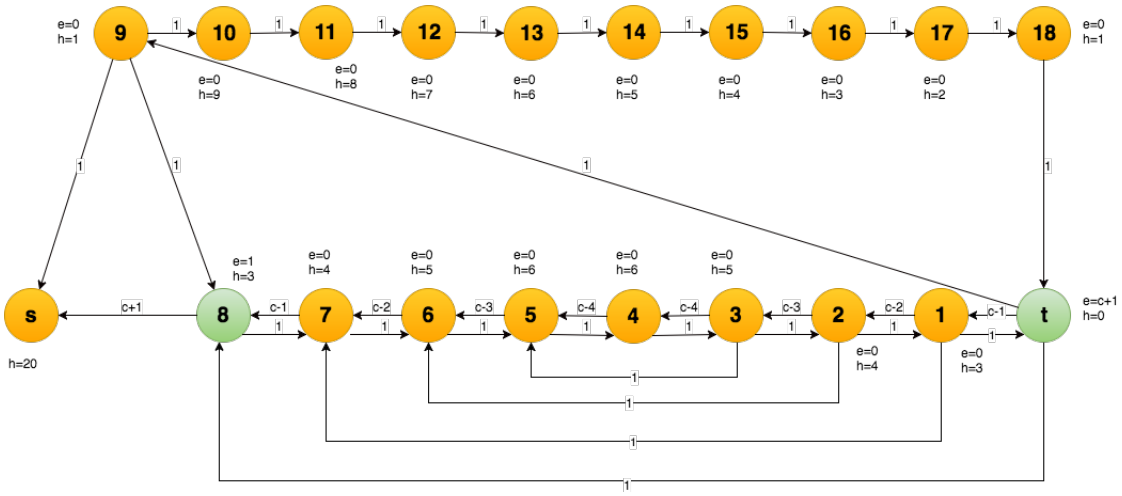



Figure 2.9: The only active node is node 8.

We obtain a residual network that look like the residual network in the figure 2.8. The only differences are the height label of the bottom vertices. The algorithm will relabel and push the vertices along the path 1-2-3-4-5-6-7-8. We have now the residual network described in the figure 2.11.

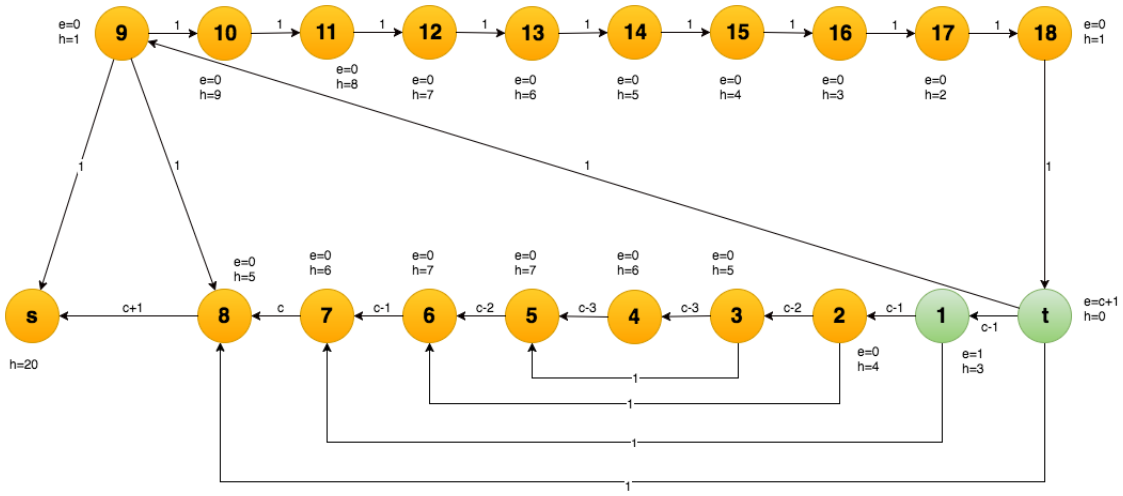



Figure 2.10: Node 1 is active again.

This round trip between the vertex 1 and the vertex 8 is called the ping-pong effect and is the main drawback of the preflow-push algorithms. This flow exchange terminates when the height label of the vertex 8 become greater than the height label of the source.

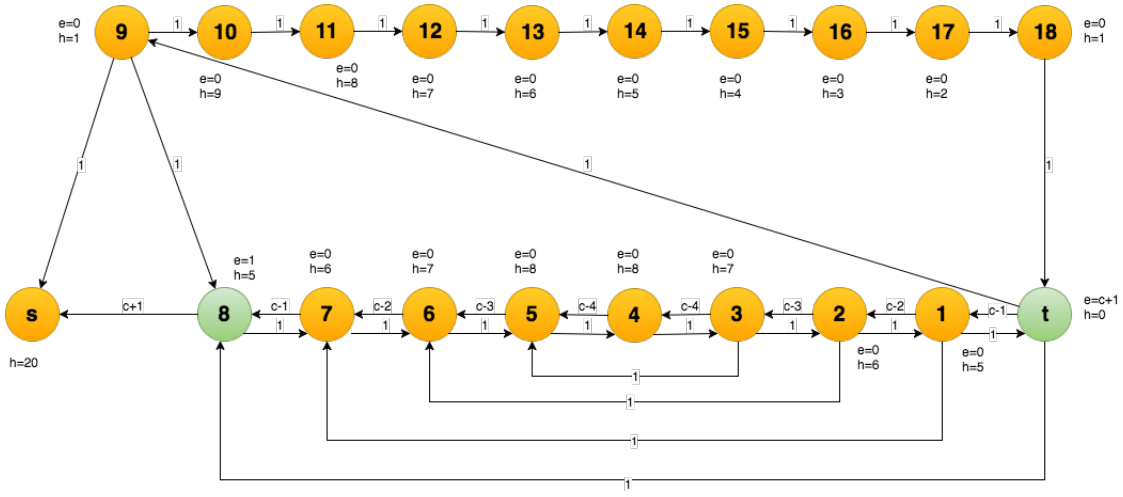



Figure 2.11: Node 8 is active again

### Pratical case

In this section, we will present you a simple case (Table 2.1) where the Preflow-push algorithm perfom very differently between two similar cases. The only difference, as shown in the state 1, is that the capacity of the upper edges of the graph differ by one. In the easy case, the flow is push from $s$ to $t$, then $s$ to 1, then 1 to 2 and so on. When the vertex 3 become active (with a excedent of 9) it push 4 units of flow to the sink (state 2). The vertex 3 is still active and will

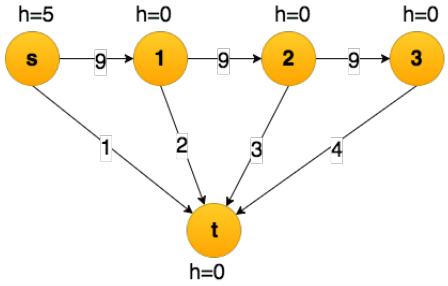
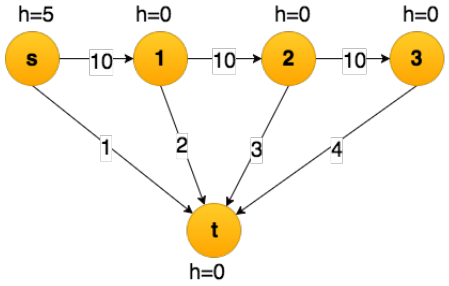
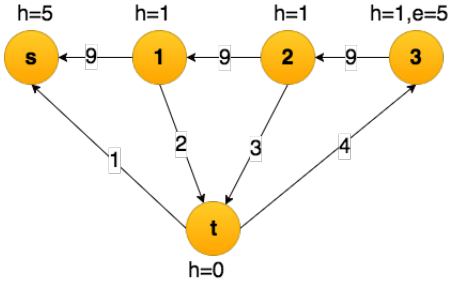
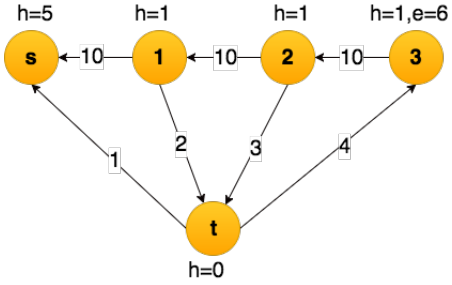
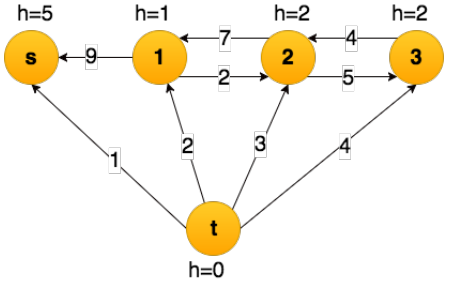
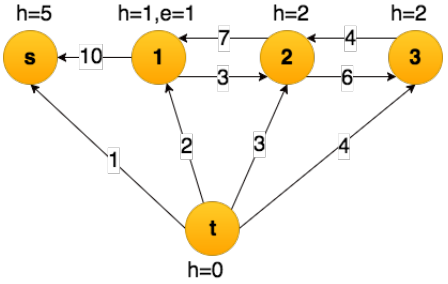
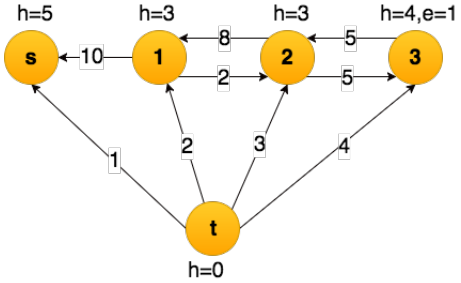
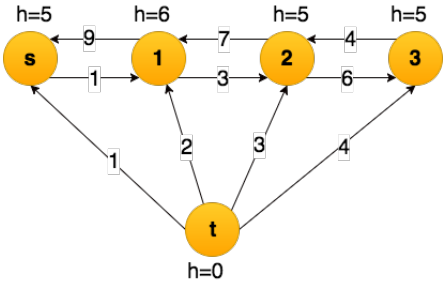
| State | Easy case | Difficult case |
|---|---|---|
| State 1 |  |  |
| State 2 |  |  |
| State 3 |  |  |
| State 4 | Solved |  |
| State 5 | Solved |  |

Table 2.1: Two similar cases with different resolutions.

push his flow to 2 after being relabeled. The vertex 2 will do the same to the vertex 1. We are in the state 3 and no more vertices are actives, thus the algorithm terminates.
In the second case, the three firsts states are the same except for the capacity of the upper edges. This cause an excedent of one unit in the vertex 1. This active vertex prevent the algorithm to terminate. The algorithm then push this excedent to the vertex 2 after relabeling the vertex 1. The vertex 2 will do the same to the vertex 3. We are now in state 3 and this ping pong effect will occur till the height label of the vertex 1 is bigger by one than the height label of the source. When this is the case, the vertex 1 push the excedent to the source and no more vertices are actives: the algorithm terminates. The longer the upper path is, the more time it takes to bring the extra unit of flow back to the source.

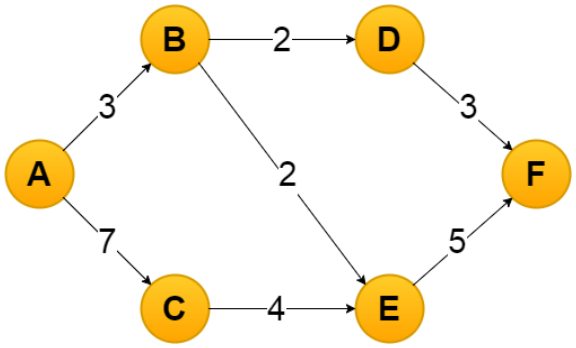
This simple case demonstrate how the Preflow-push algorithm perfom differently on graphs with small differences.

# Chapter 3

# Data Structures

For this master thesis, we want to be able to analyse the difference between several data structures. Moreover, the traditional adjacency matrix [Sch15a] is not adapted for the big graphs. Indeed the neighbourhood of a vertex is obtained in $O(|V|)$ given that it is necessary to go through the entire line. Furthermore, when the graph is not dense, a big part of the used memory is useless.

As a reminder, an adjacency matrix is a two dimension array, where each line represents a starting vertex and each column represents a ending vertex. The values represent the capacity between both vertices, 0 meaning that there are no edge between them.




| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| A | 0 | 3 | 7 | 0 | 0 | 0 |
| B | 0 | 0 | 0 | 2 | 2 | 0 |
| C | 0 | 0 | 0 | 0 | 4 | 0 |
| D | 0 | 0 | 0 | 0 | 0 | 3 |
| E | 0 | 0 | 0 | 0 | 0 | 5 |
| F | 0 | 0 | 0 | 0 | 0 | 0 |

Figure 3.1: A graph with its adjacency matrix.

We thus decide to use an other structure;

With the same principle as the adjacency matrix, we use an array where each line represents the neighbours of a vertex. For example, the first line contains the information on the neighbours of the first vertex. But contrary to the adjacency matrix, we do not use an array to represent neighbours but a different structure requiring less memory space.

We used five different data structures : hash map, tree map, simple linked list and two

home-made structures based on the sparse set, split array and sparse map.

Each vertex will have its structure, storing which vertices are neighbours and what are the capacities of the edges to these vertices. If we had to represent a graph with 10 vertices, we would have a array of 10, for example, hash maps. Each hash map representing the neighbourhood of a single vertex.

## 3.1 Data structures

### 3.1.1 Hash Map

A hash map is an unordered associative array, which associates a key with a value. It contains a single array of buckets, where the values are stored. A hash function converts the key into an index, which represents the bucket where the record (key/value) is stored [Dro08].

Ideally, the hash function assigns to every key a different bucket but it is possible to have several keys giving the same hash code. This is called a *collision*. One bucket can thus contain several records.

The *load factor* is the number of records divided by the number of buckets. If the load factor is too high, the hash map will be slower. But having a too low load factor does not save search time, it just uses some memory pointlessly. To keep the load factor to a defined value (eg between $\frac{2}{3}$ and $\frac{3}{4}$), we must, when inserting new records, resize the hash map.

In our case, the key is the id of the nearby vertex and the value is the capacity of the edge.

### 3.1.2 Tree Map

A tree map, implemented by a Red-Black tree in Java[1], is a self-balancing binary search tree. In addition to the restrictions imposed by the binary search tree, which is to have for each vertex, the *left* sub-tree containing only lower keys and the *right* sub-tree only higher keys, the Red-Black tree respects four other conditions thanks to an additional information, the color of a vertex :

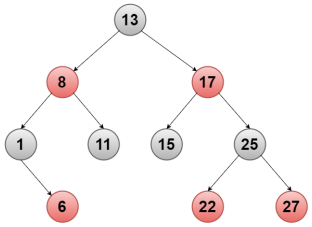


Figure 3.2: A Red-Black tree.

- A vertex is either red or black
- The root is black
- The parent of a red vertex is black
- For each leaf, the path to the root contains the same number of black vertices

These constraints implies an important property of the Red-Black trees : the longest possible path from a root to a leaf can be only twice as long as the smallest possible. We thus have an almost balanced tree. This ensures the fundamental operations to be performed in $O(log\ n)$.

[1] https://docs.oracle.com/javase/7/docs/api/java/util/TreeMap.html

### 3.1.3 Simple Linked List

The simple linked list is one of the simplest data structures. Each node consists of two fields, an object and a reference to the next node. It is unordered given that the put operation is systematically made on the head of the list.

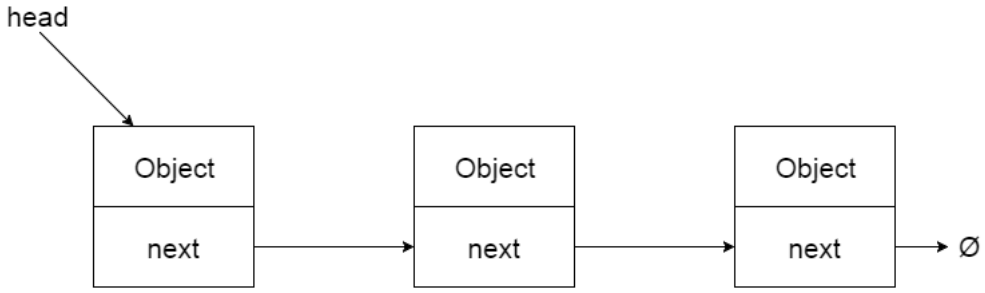



Figure 3.3: A simple linked list.

### 3.1.4 Home-made structures

We have implemented two home-made structures based on the sparse set. A sparse set is composed by two arrays, *dom* and *map*. *map* contains the position of each element in *dom* which contains the data. So $dom[map[i]]$ contains the data related to $i$. An integer value *split* represents the separation between current elements and removed elements [Sch15b]. For instance, if the sparse set represents the neighbours of a vertex, *dom* will contains all possible neighbouring vertices (outgoing and incoming edges). All vertices on the left part of *split* are current neighbours while all vertices on the right part aren't.

Here is an example of sparse set :

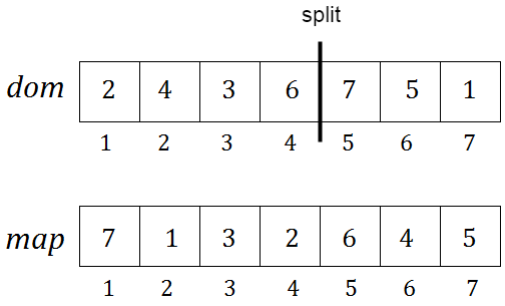



Figure 3.4: A sparse set.

We cannot use this structure like that because the neighbourhood of a vertex doesn't form a suite of index. For example, if a vertex had three neighbours 8, 72 and 13, we can't directly

know which vertex is contained in which index. So we implemented a first data structure, the split array, which is a simplified version of the sparse set without *map*. We also implemented an other structure, the sparse map, which is a sparse set with a hash map instead of an array for *map*, permitting to associate a vertex with its position in *dom*.

### Split array

A split array is composed only by the *dom* array which contains all possible neighbouring vertices. The array is divided into two parts, one with the current neighbour vertices and the other one with the vertices which are not, or no more, neighbours. An integer value *split* indicates the position of the array's separation.

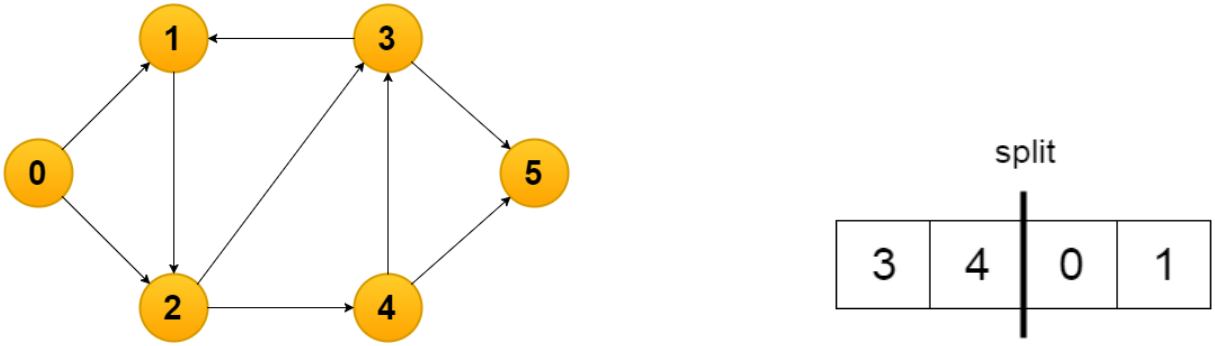



Figure 3.5: A graph with the split array of vertex 2.

To add a vertex (for example after sending units of flow on the path 0-1-2-4-5, an edge is created from 2 to 1 and 1 becomes a current neighbour of 2), we need to go through the right part of *dom* to find the future neighbour vertex, exchange its place with the vertex on the right of *split* and increase *split* by 1.

To remove a vertex, it is the reverse operation. We need to go through the left part of *dom* to find the neighbour vertex, exchange its place with the vertex on the left of *split* and decrease *split* by 1.

### Sparse map

A sparse map is a sparse set with a hash map instead of an array for *map*. The hash map associate each vertex with its position in *dom*.

To add or remove a vertex, it is the same as for the split array but we don't need to go through *dom* because we can obtain the position of a vertex in *dom* thanks to the hash map, saving valuable time. In compensation, we need to update the hash map so it remains consistent. We simply need to swap the values of the two vertices which were exchanged in *dom*.

### 3.1.5 Complexities

We use 5 functions on our data structures :

**entrySet** which is used to obtain the set of neighbours

**get/set** which, respectively, returns or modifies a capacity

**add/remove** which is used to add or to remove a vertex

This table below represents the complexities with the notation big O, which means "in the worst case". Although for the tree map, the simple linked list and the split array, the worst case is not too penalizing but that is not the case for the hash map and thus for the sparse map. Indeed, we obtain $O(n)$ if every elements of the hash map are in the same bucket. However, with a correct hash function, we can expected $O(1)$ [Dro08].

| | **Hash map** | **Tree map** | **Simple linked list** | **Split array** | **Sparse map** |
|---|---|---|---|---|---|
| entrySet | $O(b)$ | $O(n)$ | $O(n)$ | $O(n)$ | $O(n)$ |
| get/set | $O(n)$ | $O(log\ n)$ | $O(n)$ | $O(n)$ | $O(n)$ |
| put | $O(n)$ | $O(log\ n)$ | $O(1)$ | $O(n^-)$ | $O(n)$ |
| remove | $O(n)$ | $O(log\ n)$ | $O(n)$ | $O(n)$ | $O(n)$ |

With $n$ the number of outgoing edges of a vertex, $n^-$ the number of incoming edges and $b$ the number of buckets. So, $n$ is also the number of neighbours.

# Chapter 4

# Improvements of Existing Algorithms

In this chapter, we will describe some improvements found on the literature [EK72, AMO93] that can be done on the algorithms presented in Chapter 2.

## 4.1 Ford-Fulkerson Scaling

The Ford-Fulkerson algorithm has a pseudo-polynomial time complexity, making it very slow when the maximum capacity of the edges is high. But there is a variant of this algorithm which has a polynomial time complexity, the Ford-Fulkerson with scaling. The scaling consists in looking for an augmenting path which has a large enough residual capacity. To do it, we use $G_\Delta$ which is $G$ with only the edges having a capacity greater than or equal to $\Delta$ [Sch15a].

As long as there is an augmenting path in $G_\Delta$, we send flow through it. This is a $\Delta$-scaling phase. When a $\Delta$-scaling phase is ended, which means that there are no more augmenting paths in $G_\Delta$, we divide $\Delta$ by 2 and begin the next $\Delta$-scaling phase, looking for new augmenting paths in $G_\Delta$. Initially $\Delta = U$, with $U$ the maximum capacity of the graph.

### 4.1.1 Complexity

To compute the complexity of the Ford-Fulkerson Scaling algorithm, we need to know how many $\Delta$-scaling phases are possible, how many augmenting paths can be discovered at most in a $\Delta$-scaling phase and how an augmenting path is found [Way15].

- There is at most $O(log(U))$ $\Delta$-scaling phases because initially $\Delta = U = 2^{log(U)}$ and after each phase, $\Delta = \frac{\Delta}{2}$.
- There is at most $O(|E|)$ augmenting paths in each $\Delta$-scaling phase. To prove it, let $f$ be the flow at the end of the $\Delta$-scaling phase and $f^*$ be the maximal flow. At the end of a $\Delta$-scaling phase, the total flow which we can add to $f$ to obtain $f^*$ is less than or equal

to $|E| \cdot \Delta$ so $f^* - f \leq |E| \cdot \Delta$. Since we know that in the next $\Delta$-scaling phase, $\Delta' = \frac{\Delta}{2}$, there will be a maximum of $2|E|$ augmentations during the next phase [Sch15a].

- We know that each augmenting path can be found in $O(|E|)$.

The Ford-Fulkerson Scaling algorithm is thus bounded by $O(|E|^2 \cdot log(U))$.

## 4.2 Preflow-push heuristics

In Section 2.2, we defined the generic preflow-push algorithm. In this algorithm, we do not make a particular choice when it comes to select the next operation. In this section, we will define two heuristics that aim to reduce the number of non-saturing pushes which is the bottleneck of the generic preflow-push algorithm [AMO93].

### 4.2.1 FIFO heuristic

This algorithm examines the active vertices in the first-in, first-out (FIFO) order. The set of the active vertices is now a queue. The algorithm will always choose the first vertex from the queue while there are active vertices. The algorithm terminates when the queue is empty.

#### Complexity

To analyze the complexity of the FIFO preflow-push algorithm, we must define the concept of a *vertex examination*. We call vertex examination the sequence of operations that make an active vertex inactive or relabeled. For example, the algorithm could perfom several saturing pushes, leaving the first vertex of the queue active. Then the algorithm could make a non-saturing push or could relabel the vertex. We refer to this sequence of operations as a vertex examination.

We will partition the total number of vertex examinations into phases. The first phase consists of the vertex examinations of the vertices that becomes actives in the preprocess operation (the neighbors of the source). After all those specific vertices have been examinated, we enter in the second phase. The second phase consists of the vertex examinations of the vertices that are in the queue at this moment (i.e. when the vertices of the first phase has been all examinated). And so on.

To bound the number of phases in the algorithm, we define the potential function $\phi = max\{h(i) : i \text{ is active}\}$ where $h(i)$ is the height of the vertex $i$.

During a phase, the algorithm can perform at most one relabel operation. In the first case, the excess of every vertex that was active at the beginning of the phase moves to vertices with smaller height labels and $\phi$ decreases by at least one unit. In the second case, when the algorithm performs at least one relabel operation during a phase, $\phi$ might increase by as much as the maximum increase in any height label. As we know that the maximum number of relabel operation is $2 \cdot |V|^2$ (because each label can increases at most $2 \cdot |V|$ times), the total increase in $\phi$ over all phases is at most $2 \cdot |V|^2$.

With these two cases, we can bound the total number of phases by $2 \cdot |V|^2 + |V|$. The FIFO preflow-push algorithm thus runs in $O(|V|^3)$.

### 4.2.2 Highest label heuristic

This algorithm always pushes flow from the active vertex with the highest height label.

#### Complexity

It is fearly easy to develop an $O(|V|^3)$ bound for this algorithm. We define the function $h^* = max\{h(i) : i \text{ is active}\}$. First, the algorithm examines the active vertices with distance labels equal to $h^*$ and pushes flow to active vertices with distance labels equal to $h^* - 1$ and these vertices, in turn, push flow to active vertices with distance labels equal to $h^* - 2$, and so on. These operations stop when there is no more active vertices or the algorithm relabels a vertex. If the algorithm relabel a vertex, these operations are repeated. In the worst case, the algorithm makes $|V| - 1$ vertex examinations and then relabel a vertex. As we know that the maximum of relabel operations is $2 \cdot |V|^2$, the highest label preflow-push algorithm runs in $O(|V|^3)$.

However, this bound is rather loose and can be improved by a more clever analysis. We will first define some concepts and functions used to compute the complexity of the algorithm.

We define the *set of admissible arcs* as the set, at some point of the execution of the algorithm, of all the arcs $(u, v)$ such that $h(u) = h(v) + 1$. This set forms a *forest* (a set of trees) because it has at most $n - 1$ arcs, at most one incoming arc per vertex, and does not contain any cycle. The root of each of these trees is the vertex without any incoming arc.

For any vertex $i \in V$, we denote $D(i)$ the set of descendants of that vertex in the set of admissible arcs. The height label of the descendants of any vertex $i$ will always be higher than $h(i)$.

An active vertex with no active descendants (other than itself) will be called a *maximal active vertex*. We denote the set of the maximal active vertices $H$.

We define the potential function $\phi = \sum_{i \in H} \phi(i)$ with $\phi(i) = max\{0, K + 1 - |D(i)|\}$ where $K$ is a constant that we will define later. For any vertex $i$, $\phi(i)$ is at most $K$ because $|D(i)| \geq 1$.

We will now see how each operations performed by the algorithm will change the value of $\phi$. First, a non-saturing push on the arc $(i, j)$ will make $i$ inactive and $j$ might become a new maximal active vertex. Since $|D(j)| > |D(i)|$, this push increase $\phi(i) + \phi(j)$ by at least one if $|D(i)| \leq K$. When a saturing push occurs on the arc $(i, j)$, this arc becomes inadmissible and will be no more in the current forest (the set of admissible arcs). The vertex $i$ is no more a maximal active vertex and $j$ might become a new maximal vertex. This operation increases $\phi$ up to $K$ units. Let's consider now a relabel operation on the vertex $i$. This vertex has no admissible arcs because we relabeled it. Thus, this vertex is a root vertex in the current forest and it has no active proper descendants. After the relabel operation, all the incoming arcs at vertex $i$ become inadmissible. Therefore, all the current arc entering vertex $i$ will no

longer belong to the current forest. The relabel operation decreases the number of descendants of $i$ by one: $\phi$ increases by at most $K$. The last operation is to introduce new arcs in the current forest. This does not create new maximal active vertices and might remove maximal active vertices and increase the number of descendants of some vertices. Thus, $\phi$ does not increase.

To compute the worst-case, we define $h_{max} = max\{h(i) : i \text{ is active}\}$. A phase is the sequence of pushes during which $d_{max}$ remains unchanged. There are $O(|V|^2)$ phases because there can't be more than $2 \cdot |V|^2$ relabel operations. We distinguish two types of phases: *cheap* phases and *expensive* phases. A phase is cheap when it performs at most $\frac{2 \cdot |V|}{K}$ non-saturing pushes and expensive otherwise. The number of non-saturing pushes in cheap phases is at most $O(|V|^2 \cdot \frac{2 \cdot |V|}{K}) = O(\frac{|V|^3}{K})$.

By definition, an expensive phase performs at least $\frac{2 \cdot |V|}{K}$ non-saturing pushes. Since the network can contain at most $\frac{|V|}{K}$ vertices with $K$ descendants or more, at least $\frac{|V|}{K}$ non-saturing pushes must be from vertices with fewer than $K$ descendants. As we said earlier, the total increase of $\phi$ due to saturing pushes and relabels is at most $O(|V| \cdot |E| \cdot K)$. The algorithm perform $O(|V| \cdot |E| \cdot K)$ non-saturing pushes in expensive phases.

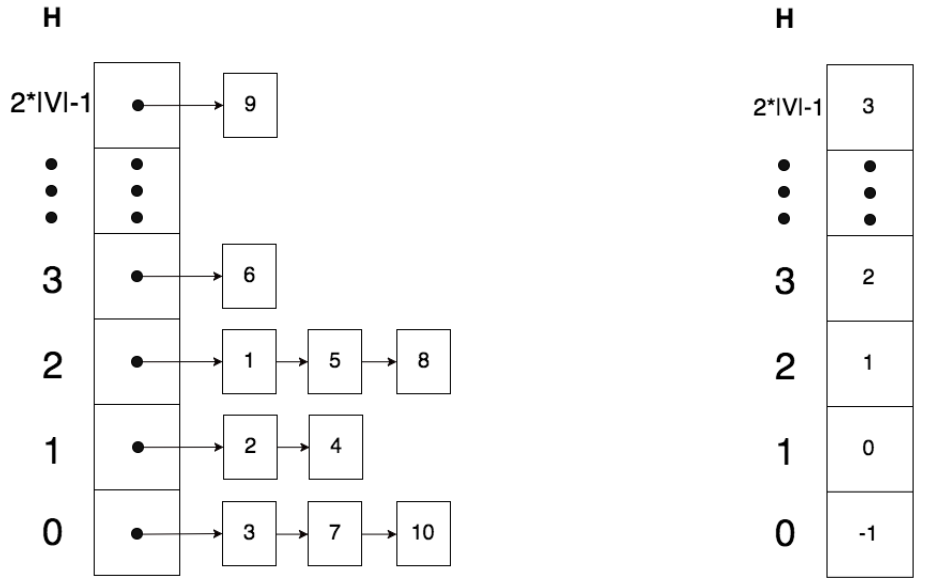
By balancing the complexity of each case, we obtain the optimal value of $K$ when both are equal: $\frac{|V|^3}{K} = |V| \cdot |E| \cdot K$ or $K = \frac{|V|}{\sqrt{|E|}}$. We have now the number of non-saturing pushes: $O(|V|^2 \cdot \sqrt{|E|})$ which is the complexity of our algorithm.

Textbooks do not provide a description of the data structure used to enforce the highest label heuristic. We implemented a custom data structure based on the idea of François Aubry, PhD student at UCL. We therefore show how to implement such a data structure with $O(1)$ operation.

**The ActiveSet**

The data structure is divided in two parts: the ActiveSet and the next array. The ActiveSet, as shown on Figure 4.1a, is a array of linked lists of size $|V| \cdot 2$. The ActiveSet can be defined as follows: $activeSet(k) = \{i : i \text{ is active and } h(i) = k\}\ \forall\ 0 \leq k < |V| \cdot 2$. When a vertex becomes active, it is added to the linked list at the index of the array corresponding to his height. The next array is an integer array of size $|V| \cdot 2$ where all of his element are pointers to the next non empty list. For example, let's take the case of the Figure 4.1, where there is no active vertices with heights between 3 and $2 \cdot |V| - 1$ (all active vertices are shown in the figure). The first (top) element of the data structure is 9 and the second is 6. In the next array, the value at the index $h(9)$ must be equals to $h(6)$ and so on. Note that the minimum height of the active vertex must point to $-1$ because there is no following elements. A variable is used to point at the top of the ActiveSet. Here is the operations needed:

**getTop** Give the top element of the ActiveSet. It is used when the highest label preflow-push algorithm must examine a vertex;




(a) The ActiveSet (b) The next array

Figure 4.1: The data structure

**add** When a vertex become active, the vertex is added to the ActiveSet;

**updateTop** Used when the top vertex is relabeled;

**removeTop** Used when the top vertex is no longer active (i.e. when there is a non-saturing push);

**isEmpty** Used to know when the ActiveSet is empty. When the ActiveSet is empty, the algorithm terminates.

## 4.3 No initialization of height in Preflow-push algorithms

In the Preflow-push algorithm, there is an initialization phase, called *preprocess*, used to compute the height label of the vertices and performs a saturing push on all the edges leave the source. The computation of the height labels can be performed in $O(|V|)$ with a simple breath first search from the sink (see Algorithm 5). This computation performs $|V|$ relabelings. But what happens if we do not compute the height labels before (i.e. all the height label will be equals to zero) ? In fact, the number of relabeling will be the same. If we do not compute the height labels before, any active vertices will be relabeled before to be able to make a first push because the active vertex will be at height zero and his neighbours too. From that, to case can arise: first, the push is non-saturing. It means that the vertex is no longer active, and the new active vertex need to be relabeled in order to be able to make a push from it. So there is no more relabeling operations than if we compute the height labels in the preprocess. In the second

case, the push is saturing: the vertex is still active. In this case, this vertex can push directly into an other neighbour because is height label has already been set at one unit higher than his neighbours. In these two cases, there is not unnecessary relabeling operation. In term of relabeling, there is no gain to compute the height labels in the preprocess.

```
parents ← fill(−1);
Queue q ← new Queue;
q.Enqueue(sink);
sink.h = 0;
while q is not empty do
    u ← q.Dequeue();
    for every vertex v adjacent of u do
        if v.h equals to 0 then
            parents[v] ← u ;
            v.h ← parents[v].h + 1 ;
            q.Enqueue(v);
        end
    end
end
```

**Algorithm 5:** The computation of the height labels.

However, not making this computation could prevent the highest label heuristic (Section 4.2.2) to perfoms at his best. We will analyze this in the experimental analysis (Chapter 6).

# Chapter 5

# Implementation

In this chapter, we will explain our implementation choices and the tools we used to carry out our project. Our main goal was to release a maintainable open source library so that other developers could use it and contribute to it. Our choices were mainly guided by this goal.

## 5.1 Langage choice


Figure 5.1: Java Logo

In order to implement the different flow algorithms presented, we had to choose a programming language. We decided to develop the different algorithms with **Java**[1]. Java is a programming language created in 1995 by Sun Microsystems. Java is designed to run acress multiple operating systems, including Linux, Mac OS X and Windows. Java is known to be flexible, scalable and maintainable. Java is an object-oriented programming language, which permits the development of modulable applications via systems like encapsulation, composition, inheritance, and delegation. We choose Java because of the followings reasons:

- Java is very popular. This is a key point because we want that the librairies we developped were accessible to as many. Another consequence of this is that Java is well documented.
- Java is modulable. The OO Java paradigm is useful to develop this project were different algorithms and different data structures must be implemented. Concepts like inheritance, interfaces are very useful to achieve this.
- Java is fast. Many optimizations have improved the performance of the JVM over time such as the Just-in-time compiling. Some of our instance are very big, so we need a language able to solve them quickly.
- We are accustomed to using java. A good knowledge of the language is required to make correct implementations.

[1] https://community.oracle.com/community/java

## 5.2 Structure

Our project is divided in two core packages: *models* and *solvers*. The first one contains all the representations of the network we used. The second contains all the different algorithms we implemented. There is two others packages: *objects* which contains implementation of some data structures and *results* which contains instance generators and results analyzers.

### 5.2.1 Package: Models

This package contains all the network representation we implemented: *SplitArrayGraph*, *SparseMapGraph*, *TreeMapGraph*, *LinkedListGrah* and *HashMapGraph*. As we can see in the Figure 5.2, all these class implements the interface *Graph* which define the behaviour of the object that represent the network. Here is the method which must be implemented when creating a network representation in our project:

`parse(file_path)` parse the file given in the constructor argument. The parse method must take into account if the graph is directed or not;

`getV(), getE()` return respectively the number of vertices and the number of edges;

`getVertex(id)` return the vertex with the corresponding id;

`getVertices()` return the set of vertices;

`getAdjacents(id)` return the neighbours of the vertex *id*;

`removeEdge(id1, id2)` remove the edge between *id1* and *id2*;

`addEdge(id1, id2, c)` add an edge from *id1* to *id2* with the capacity *c*;

`getCapacity(id1, id2)` return the capacity of the edge between *id1* and *id2*;

`setCapacity(id1, id2, c)` set the capacity of the edge between *id1* and *id2* at *c*;

`getAdjacentsSize(id)` return the number of neighbours of the vertex *id*;

`getMaxCapacity()` return the biggest capacity in the network: used to initalize the scaling phase.

We defined the abstract class *SimpleGraph* which contains all the common code between all the network representations we implemented.

### 5.2.2 Package: Solvers

This package contains all the solvers we implemented. As we said in the Chapter 2, there is two family of algorithm to solve the maximum flow problem: the augmenting path algorithms and the preflow-push algorithms. The different algorithms in the augmenting path have a lot in common. For example, the difference between the Ford-Fulkerson algorithm and the Edmonds-Karp algorithm is how the algorithm find an augmenting path. As we can see in the Figure 5.3,


Figure 5.2: UML diagram of the package models.

we used the strength of inheritance to avoid duplicate code. On the other part, the preflow-push algorithm have a lot of differences in their implementations, we decided split them in three classes.

The solvers have a constructor overloading, which permits to define any vertex as the *source* and the *sink*.

### 5.2.3 Package: Objects

This package contains several data structures:

`Vertex.java` used to represent a vertex;

`Arc.java` used to represent an arc;

`Edge.java` used to represent an edge. The difference between an arc and an edge is that the arc does not have the information about the origin vertex;

`SimpleLinkedList.java` implementation of a linked list;

`Node.java` represent a node in a linked list;

`SplitArray.java` implementation of the split array;

`SparseMap.java` implementation of the sparse map;

`ActiveSet.java` data structure used to select the highest label in the highest label preflow-push algorithm.

### 5.2.4 Package: Results

This small package contains four classes. The first are the instances generators: *InstanceGeneratorDensity.java* and *InstanceGeneratorSize.java*. The first class generates a graph class with different densities and the second with different sizes. The two last classes are *ResultsConverterByDensity.java* and *ResultsConverterBySize.java*. They are used to analyze our results.

### 5.2.5 Conclusion

The main advantage of our structure is how easy is to add different data structures or solvers. All you need is to implement the interfaces and add your own class in the correct package. Then, the solvers can be call with different data structures with a few line of codes as we can see in the Figure 5.4. Another advantage is that our code can be used as a library, without even looking at the code of it.

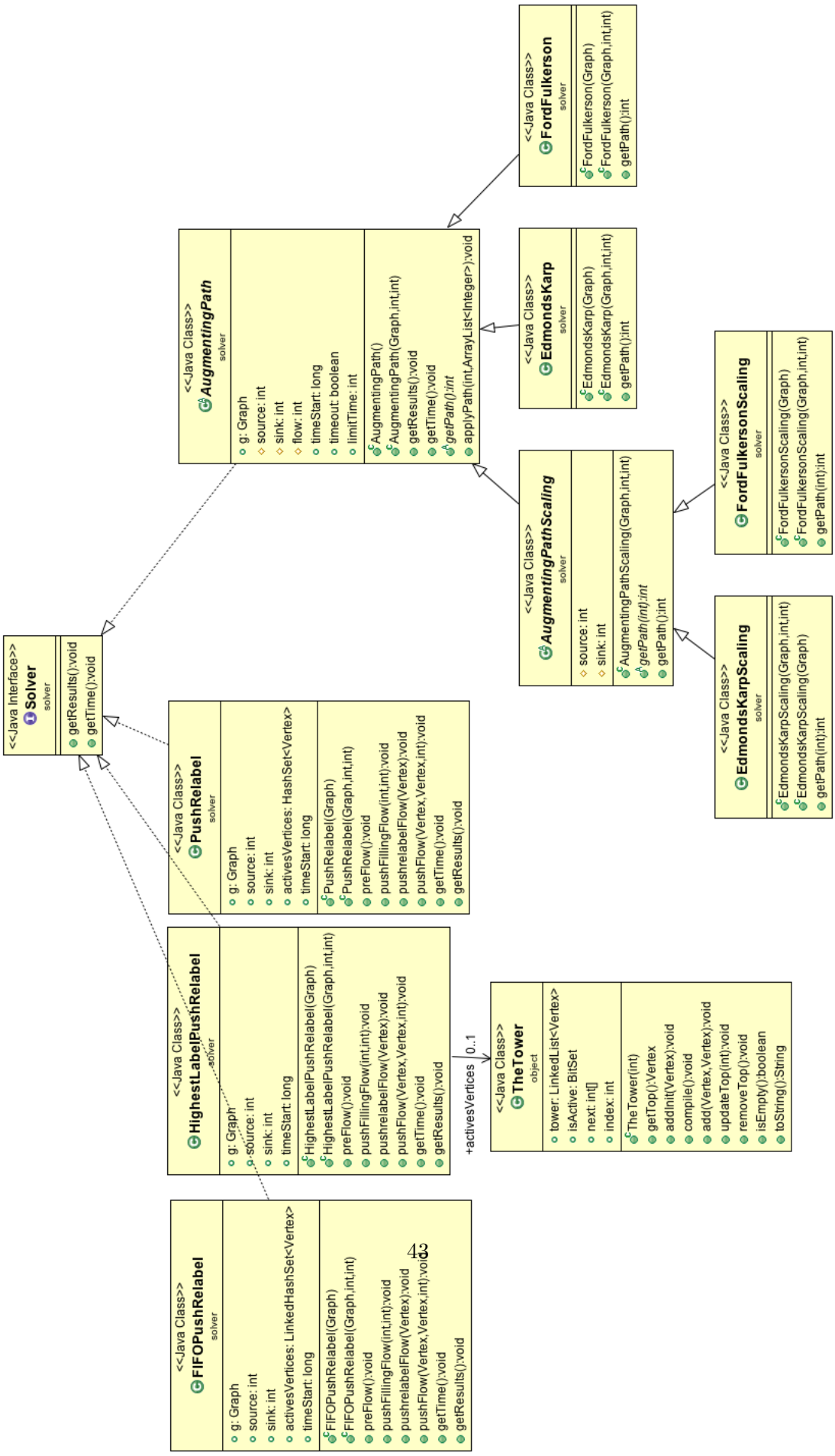





Figure 5.3: UML diagram of the package solver.

```
Graph sm = new SparseMapGraph("file_path", true);
Solver ek = new EdmondsKarp(sm);
ek.getResults();
Solver pr = new PushRelabel(sm);
pr.getResults();
```

Figure 5.4: Example of use.

## 5.3 Collecting and displaying results

In order to gather the different run times of all the algorithm we implemented on all the instances we generated, we decided to automate the launch of the different algorithms. We decided to make a script in **Python**[2] because it was the easiest way to do it. The Python script launch the different algorithms with the different data structes on all the instance, and store the computations time collected on a text file. A Java program is then used to classify the run times by solver and by instance and store these results in a csv file. When we have the run times in the right form, we use the library **ggplot2**[3] of the **R**[4] language to plot the different graphs. We choosed this library because the graphs rendered are very customizable and very nice.

All our scripts Python and R are available on the Git (see Section 5.4.1). You can also easily test your own instances thanks to the file BatchMain in the package flowAlgorithm of our Java code. It takes 5 arguments :

**args 0** represents the solver. 'FF' for Ford-Fulkerson with scaling, 'EK' for Edmonds-Karp, 'PR' for Generic Push-Relabel, 'FPR' for FIFO Push-Relabel and 'HLPR' for Highest Label Push-Relabel.

**args 1** represents the data structure. 'LL' for *linkedlist*, 'HM' for *hashmap*, 'TM' for *treemap*, 'SA' for *splitarray* and 'SM' for *sparsemap*.

**args 2** represents the path of the instance file structured as well : the first line contains "$|V|$ $|E|$" and each other line represents an edge "idvertex1 idvertex2 capacity". For example "0 1 10" represents an edge from 0 to 1 with a capacity of 10.

**args 3** 'true' for directed, 'false' for undirected graph.

**args 4** 'r' to obtain all results, empty to obtain the run time in ms.

[2] https://www.python.org
[3] http://ggplot2.org
[4] https://www.r-project.org

## 5.4 Tools

### 5.4.1 Version control system

Figure 5.5: Git Logo

Using a versioning tool appeared to us from the beginning as an evidence. Our main needs were to share the code we did between us and to version it. We decided to use **Git**[5] because we already both master it and it is one of the most popular versioning tool. We used **Github**[6] to host our application. With Github, it is fearly easy to make our project open source and knowing that this platform is very popular among developers, our project can benefit from a certain visibility. Our repository can be found at the following url: `https://github.com/denisgenon/flow_algorithm`.

### 5.4.2 Management application

Figure 5.6: Trello Logo

Even if we were only two working on the project, we judged useful to have project management tool to assist us. This kind of tools permits us to separate the tasks to make, to write somewhere our ideas, to fix deadlines and to see the evolution of our project. We decided to use **Trello**[7] because we were already used to it. Trello uses the kanban paradigm for managing projects. Our project were represented as a board, which contains columns corresponding to some states (backlog, nice to have, in progress, ...). Each column contains lists of cards which were our tasks. In this way, we could follow the flow of a feature from idea to implementation.

[5] https://git-scm.com
[6] https://github.com
[7] https://trello.com

# Chapter 6

# Experimental Analysis

For the analysis of our algorithms and data structures, we decided to analyze the run time on three types of instances :

**Density variation instances :** Graphs having a fixed number of vertices ($|V| = 1000$) and a density of edges ranging from 5 to 100%. The maximum capacity of an edge is 10000.

**Size variation instances :** Graphs having a density of edges fixed (10%) and a number of vertices varying from $|V| = 1000$ to $|V| = 5000$. The maximum capacity of an edge is 10000.

**Matching problem instances :** Graphs having a source $s$, a sink $t$ and two groups of 500 vertices $R1$ and $R2$. 500 edges connect $s$ to all vertices in $R1$ and 500 other edges connect all vertices in $R2$ to $t$. All other possible edges can only go from $R1$ to $R2$. The graphs have a density of connectivity between $R1$ and $R2$ ranging from 5 to 100%. The capacity of all edges is 1. This type of graph is often used in the maximum flow problem.

The Figure 6.1 represents a matching problem instance with a density of connectivity between $R1$ and $R2$ equal to 100%.

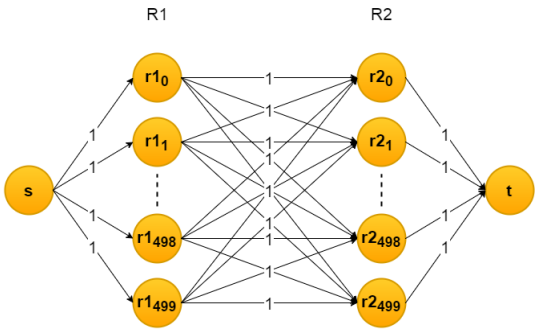



Figure 6.1: Matching problem instance with a density of edges equal to 100%.

## 6.1 Instances generation

To obtain the necessary instances, we implemented instances generators which respects the characteristics of the network graphs (a connected graph where each vertex has at least one incoming and one outgoing edge).

### 6.1.1 Density variation instances

For the density variation instances, we first create a minimal connected graph. To do this, let *Connected* be the set of the connected vertices, *DoubleConnected* the set of vertex having at least one incoming and one outgoing edge, *Edges* the set of edges and *AllEdges* the set of all possible edges. Initially, *Connected* contains the vertex 0, *DoubleConnected* and *Edges* are empty and *AllEdges* contains all possible edges. When we want to add a vertex $v$ to the graph, we take a random vertex $r$ from *Connected*, add the edge ($v$,$r$) in *Edges*, add $v$ in *Connected*, remove the edges ($v$,$r$) and ($r$,$v$) from *AllEdges* and add $r$ in *DoubleConnected*. After adding our 1000 vertices, we have a connected graph where each vertex has one incoming edge and sometimes at least one outgoing edge (all vertices in *DoubleConnected*).

For each vertex not present in *DoubleConnected*, we take a random vertex from *Connected*, add the edge between them in *Edges* and remove this edge and its opposite from *AllEdges*. For this step, to avoid adding an edge (or its opposite) which is already present in the graph, we check if the new edge and its opposite are not present in *Edges*. We thus have a connected graph where each vertex has one incoming edge and at least one outgoing edge.

We then add the necessary number of edges to obtain the desired density. We take a random edge from *AllEdges*, add it to *Edges* and remove it and its opposite from *AllEdges*. A first graph is generated when we have a density of 5%. We add it edges to obtain a density of 10% and generate a second graph. And so on up to 100%. At the end, a density variation class is composed by twenty graphs where each graph generated before an other one is a sub-graph of the latter. With $|V| = 1000$, a complete graph has $\frac{(|V|-1)(|V|)}{2} = 499500$ edges.

The Algorithm 6 represents the density variation class generator.

We generate 10 classes of this type.

### 6.1.2 Size variation instances

For the size variation instances, we use the same technique as for the density variation instances without the *AllEdges* set. Indeed, it is not necessary to generate all possible edges for graphs with a 10% of edge density. Especially when we know that a complete graph with $|V| = 5000$ has 12497500 edges. To know if an edge (or its opposite) is already present in the graph, we check if it is contained in *Edges*.

We generate a network graph with $|V| = 1000$ and an edge density of 10%. We add it 500 vertices and the corresponding edges to respect the characteristics of the network graphs. We add then the necessary edges to keep a 10% edge density and generate this new graph. And so on until $|V| = 5000$. At the end, a size variation class is composed by nine graphs where each graph generated before an other one is a sub-graph of the latter.

We generate 10 classes of this type.

**Local variables :**
*Connected* : set of connected vertices
*DoubleConnected* : set of vertex having at least one incoming and one outgoing edge
*Edges* : set of edges on the graph
*AllEdges* : set of all possible edges on the graph
*Connected* $\leftarrow \{0\}$, *DoubleConnected* $\leftarrow \emptyset$, *Edges* $\leftarrow \emptyset$, *AllEdges* $\leftarrow \emptyset$;
**for** $u \leftarrow 0$ **to** $|V| - 1$ **do**
  **for** $v \leftarrow 0$ **to** $|V| - 1$ **do**
    **if** $u \neq v$ **then**
      add ($u$,$v$) in *AllEdges*;
    **end**
  **end**
**end**
**for** $v \leftarrow 1$ **to** $|V| - 1$ **do**
  $r \leftarrow$ random vertex in *Connected*;
  add ($v$,$r$) to *Edges*;
  add $v$ in *Connected*;
  remove ($v$,$r$) and ($r$,$v$) from *AllEdges*;
  add $r$ in *DoubleConnected*;
**end**
**for** $v \leftarrow 0$ **to** $|V| - 1$ **do**
  **if** *!(DoubleConnected contains v)* **then**
    $r \leftarrow$ random vertex in *Connected*;
    **if** *Edges contains (v,r) || contains (r,v)* **then**
      $v = v - 1$;
    **else**
      add ($v$,$r$) to *Edges*;
      remove ($v$,$r$) and ($r$,$v$) from *AllEdges*;
      add $v$ in *DoubleConnected*;
    **end**
  **end**
**end**
**for** *prct* $\leftarrow 5$ **to** $100$ ***by*** $5$ **do**
  **while** $\frac{|Edges|-(V-1)}{\frac{V\cdot(V-1)}{2}-(V-1)} \cdot 100 < prct$ **do**
    $e \leftarrow$ random edge in *AllEdges*;
    add $e$ to *Edges*;
    remove $e$ and reverse $e$ from *AllEdges*;
  **end**
  generate graph with *Edges*;
**end**

**Algorithm 6:** Density variation class generator

## 6.2 Push-Relabel

In this section, we will analyze how the different heuristics performs in the push-relabel algorithm. We also tried each heuristic with and without the height label initialization phase (as explained in Chapter 4) to show if this phase is useful in practice.

### 6.2.1 Height label initialization

To analyze the effects of the height label initialization, we decided to compute the number of operations performed. In the preflow-push algorithms, they are three differents operations: the relabelling, the saturing push and the non-saturing push.

#### Relabelling

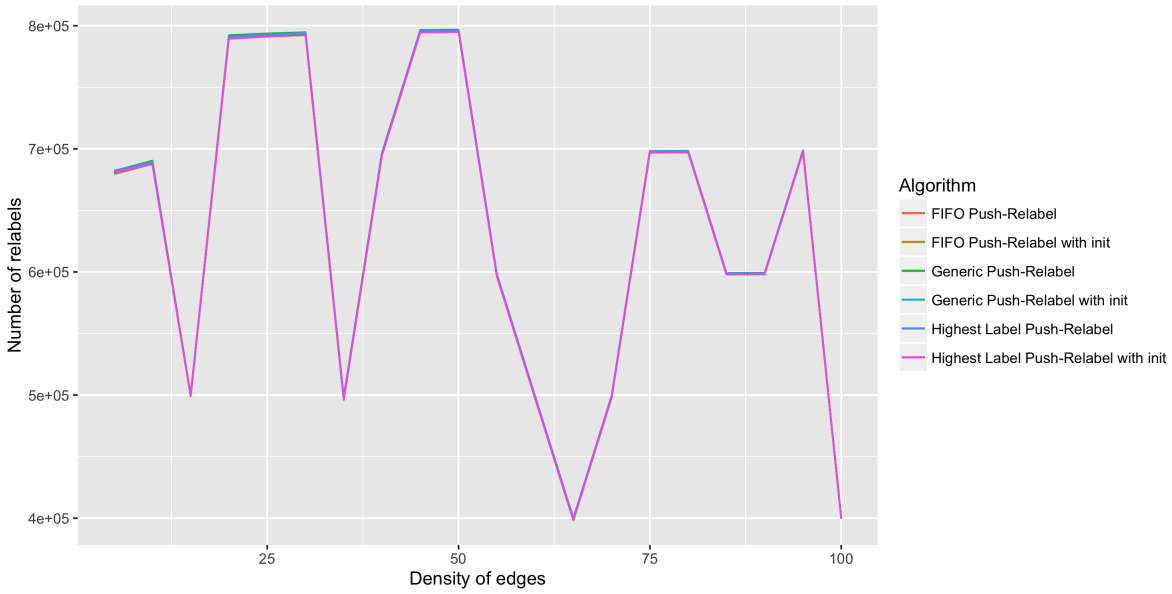



Figure 6.2: The average number of relabels in density variation classes. All the algorithms have the same count of relabeling operation.

We can observe in Figure 6.2 what we said in Section 4.3 in terms of relabels. The number of relabels is not affected by the fact we initialize or not the height label in the preflow phase of the algorithm. The reason of this is explained in Section 4.3.

### Saturing push

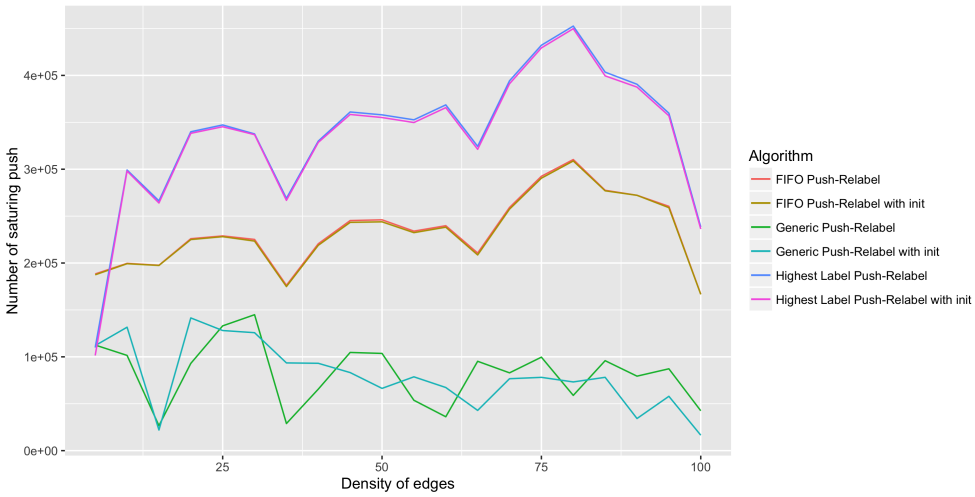



Figure 6.3: The mean number of saturing pushes in density variation classes.

We can observe in Figure 6.3 that for each heuristic, the initialization is slightly beneficial in term of the number of saturing pushes. The highest label heuristic perfoms more saturing pushes than the FIFO heuritic, and the generic algorithm performs less saturing pushes than the two others.

### Non-saturing push

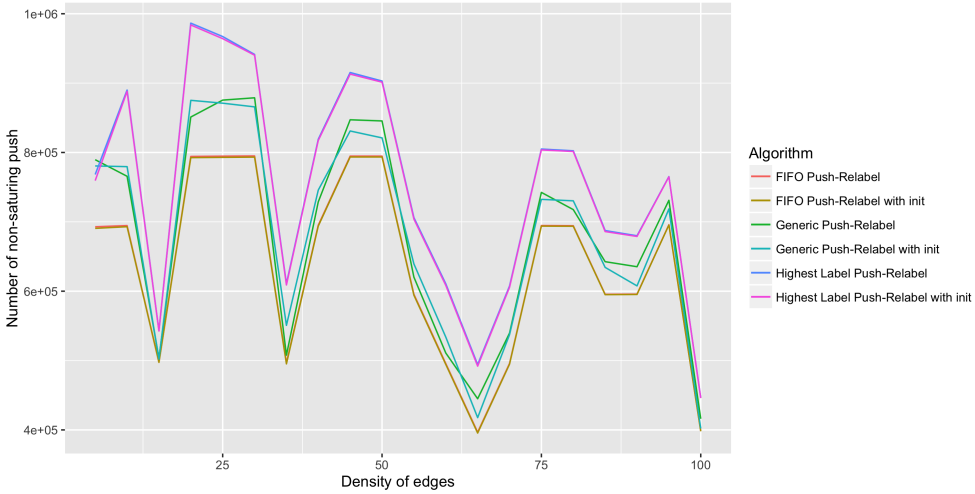



Figure 6.4: The mean number of non-saturing pushes in density variation classes.

As for the saturing push, the number of non-saturing pushes slightly decreases when we do an initialization of the height labels. The highest label heuristic decreases the number of non-saturing pushes compared to the generic algorithm. The FIFO heuristic tends to augment this number of pushes.

### 6.2.2 Best Push-Relabel

In this section, we will compare the run time of each algorithm on different kinds of instances.

#### Density variation instances

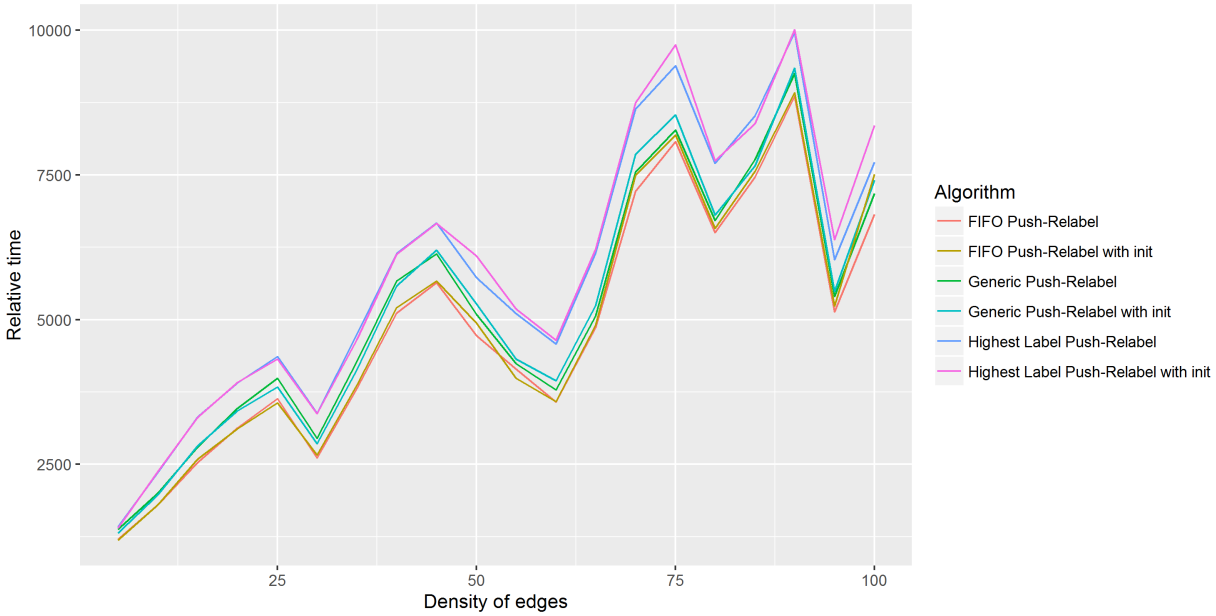



Figure 6.5: The run time of the different push relabels on all density variation classes.

We can see here that the *FIFOPushRelabel* is the fastest. For each heuristic, it is interesting to see that the initialization phase does not make the resolution of the problem faster. This is because the computation of the initialization phase does not benefit to resolve the problem faster.

### Size variation instances

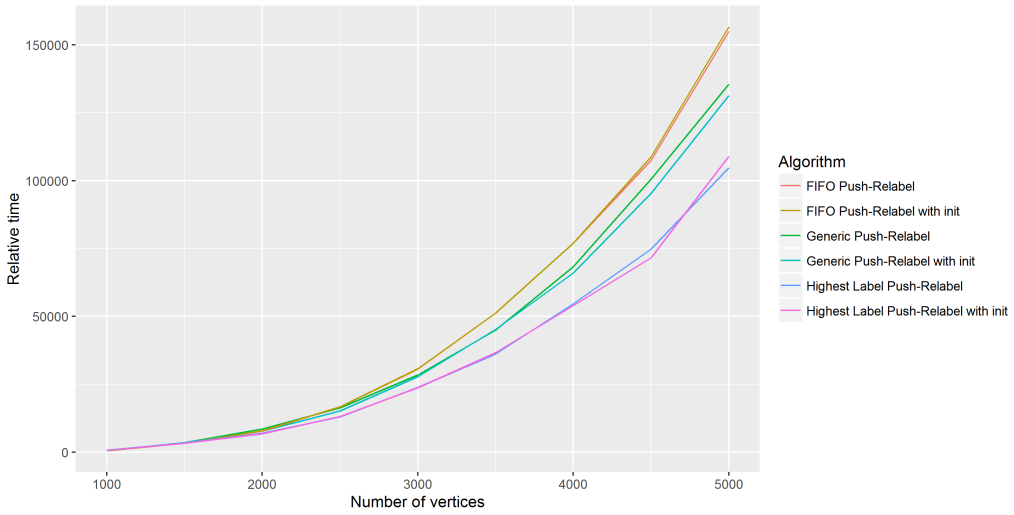



Figure 6.6: The run time of the different push relabels on all size variation classes.

When changing the size of the network, we can see that the highest label heuristic performs the best. The initialization of the algorithm does not seem to change a lot the computation time of each heuristic. In this case, the highest label is heuristic faster.

### Matching problem instances

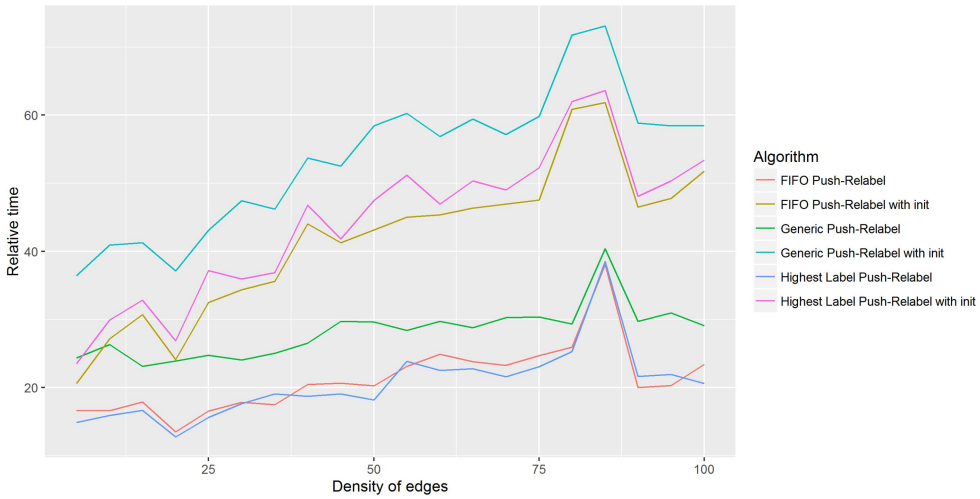



Figure 6.7: The execution time of the different push relabels on the matching problem classes.

Here, we can observe that the initialization phase makes each heuristic slower. It is because the problem here is simpler and very quick to solve, thus, the initialization phase takes a more important place in the result. Here, the high label heuristic is the best as well.

## 6.3 Data structures

In this section, we would like to determine which data structure is the most adapted for our algorithms. We therefore analyzed experimentally the run time of each algorithm based on 5 data structures defined on the Chapter 3.

### 6.3.1 Push-Relabel

The Figure 6.8, 6.9 and 6.10 represents the average run time of all data structures with Highest Label Push-Relabel. They were computed on, respectively, the density variation, the size variation and the matching problem instances.

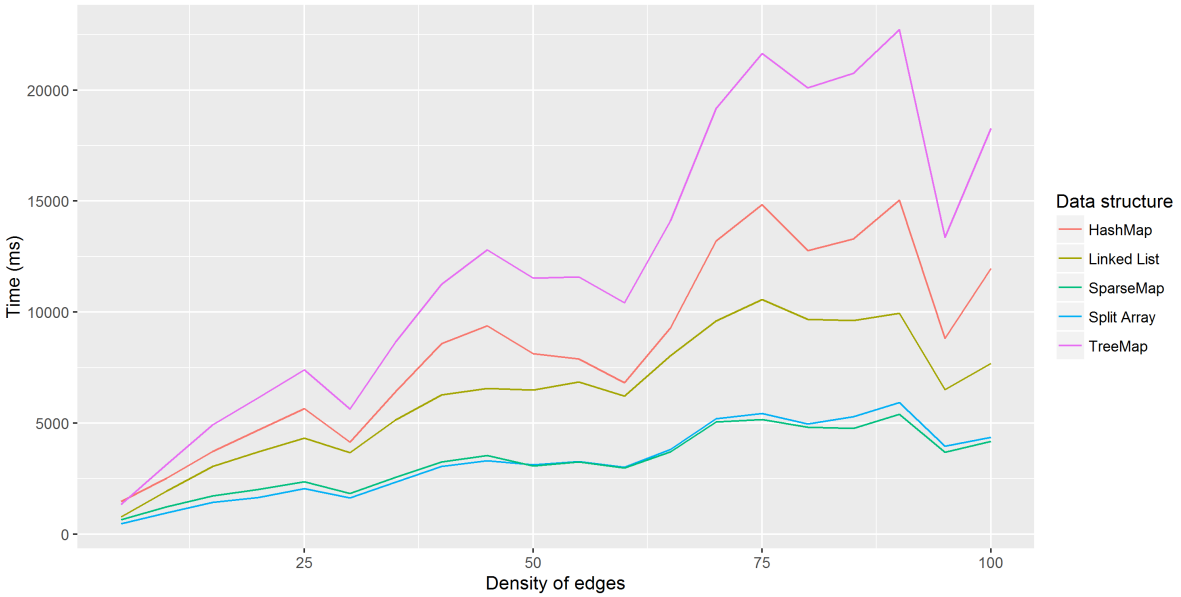



Figure 6.8: Average run time of all data structures with Highest Label Push-Relabel on density variation classes.

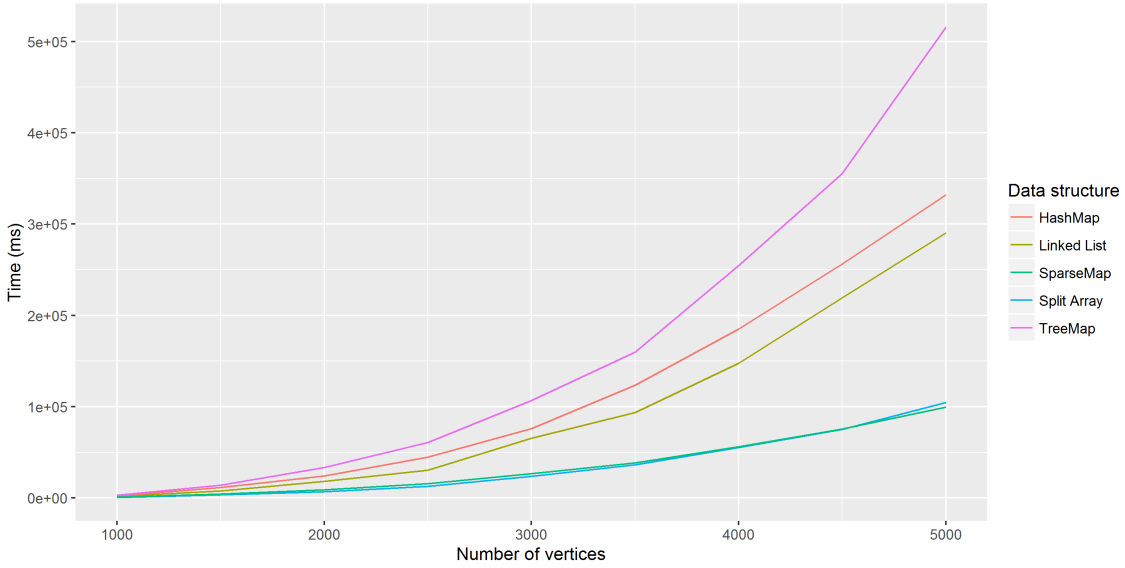



Figure 6.9: Average run time of all data structures with Highest Label Push-Relabel on size variation classes.

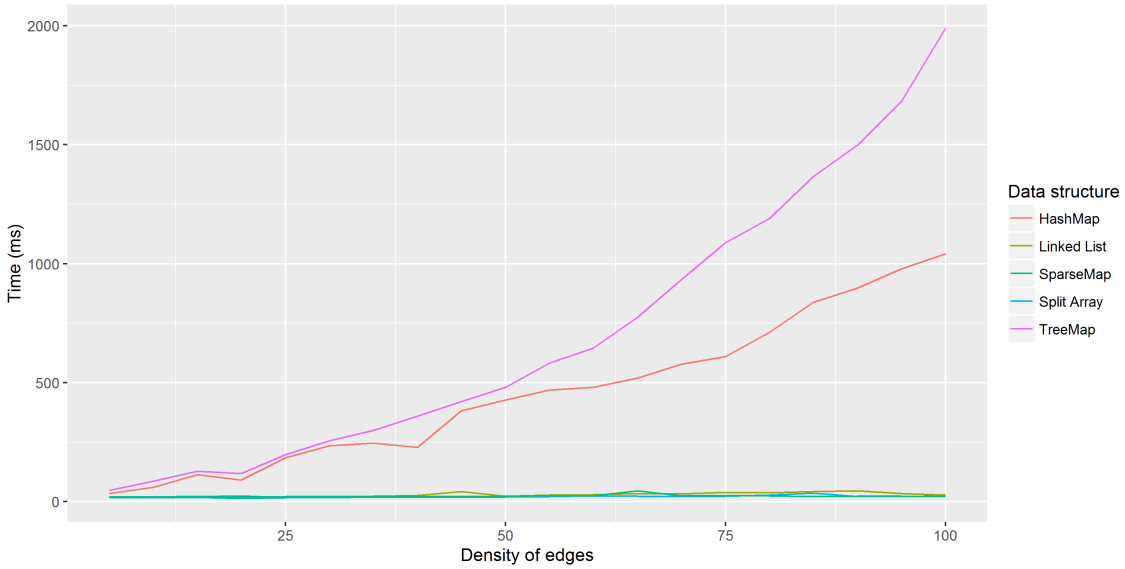



Figure 6.10: Average run time of all data structures with Highest Label Push-Relabel on matching problem classes.

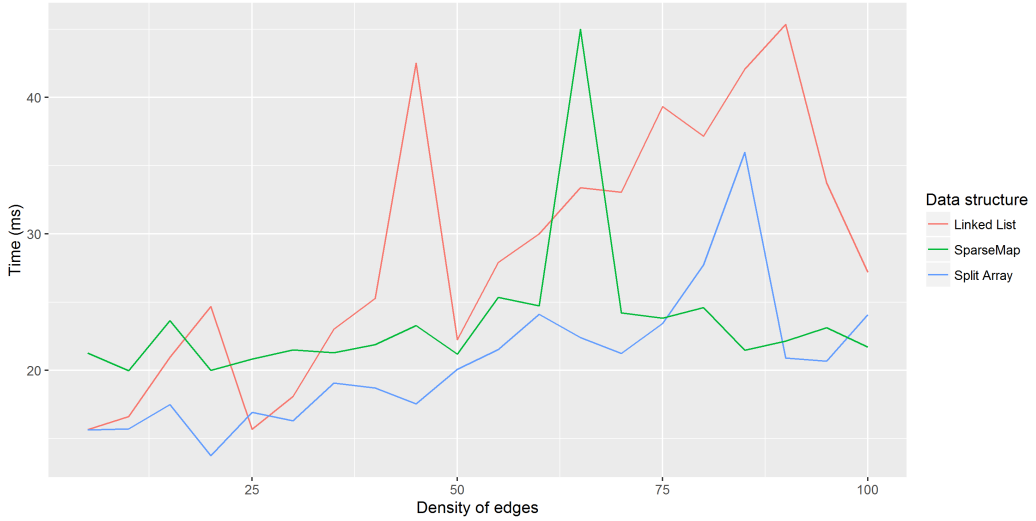



Figure 6.11: Average run time of all data structures without the *hashmap* and the *treemap* with Highest Label Push-Relabel on matching problem instances.

On the 3 type of instances, we notice that the *hashmap* and the *treemap* offer very poor performances. For density and size variation instances, structures based on the *sparseset* stand out from others by their good performances. The *linkedlist* being between map based structures and *sparseset* based structures. To be able to decide between the *splitarray* and the *sparsemap*, we need to look at the Figure 6.11, which represents the average run time of the *splitarray*, the *sparsemap* and the *linkedlist* on matching problem instances.

The most adapted data structure for Push-Relabel algorithms is the *splitarray*.

### Profiler

We have profiled our code with all data structures on a complete density instance graph with Highest Label Push-Relabel. The Figure 6.12 represents the number of invocations and the total run time of the function *getAdjacents*, which is the function that take the most time. Those results highlight the good performances of the *splitarray* and the *sparsemap*.

| | **Hash map** | **Tree map** | **Simple linked list** | **Split array** | **Sparse map** |
|---|---|---|---|---|---|
| getAdjacents | 21.940 | 21.940 | 17.555 | 18.980 | 19.513 |
| total time (ms) | 208 | 311 | 120 | 80 | 30 |

Figure 6.12: The number of invocation of the function *getAdjacents* and its total time with Highest Label Push Relabel.

### 6.3.2 Edmonds-Karp

The Figure 6.13, 6.14 and 6.15 represent the average run time of all data structures with Edmonds-Karp. They were computed on, respectively, the density variation, the size variation and the matching problem instances.

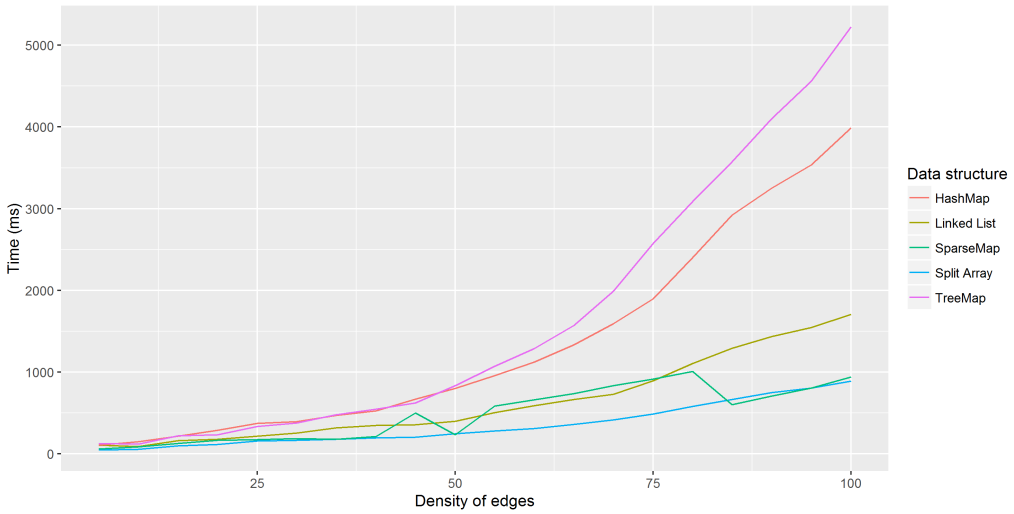



Figure 6.13: Average run time of all data structures with Edmonds-Karp on density variation classes.

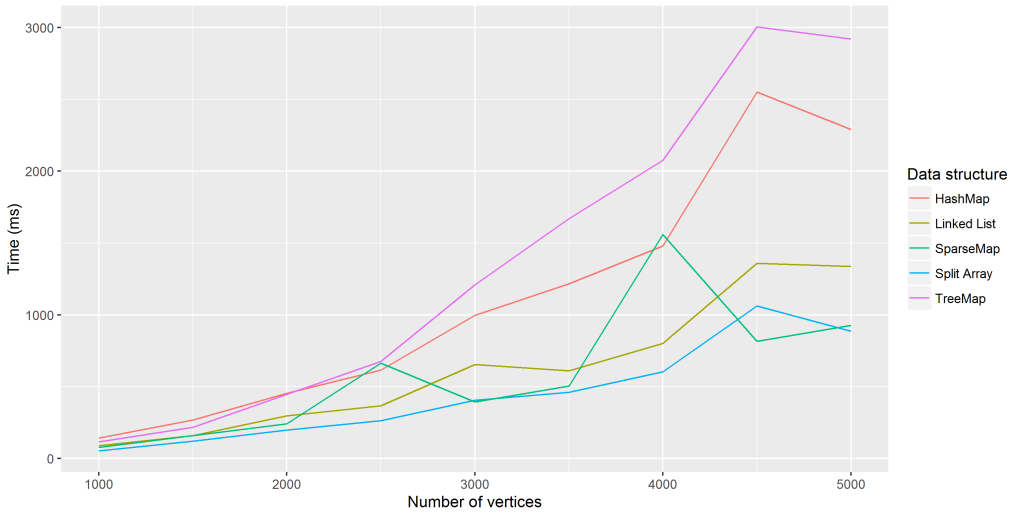



Figure 6.14: Average run time of all data structures with Edmonds-Karp on size variation classes.

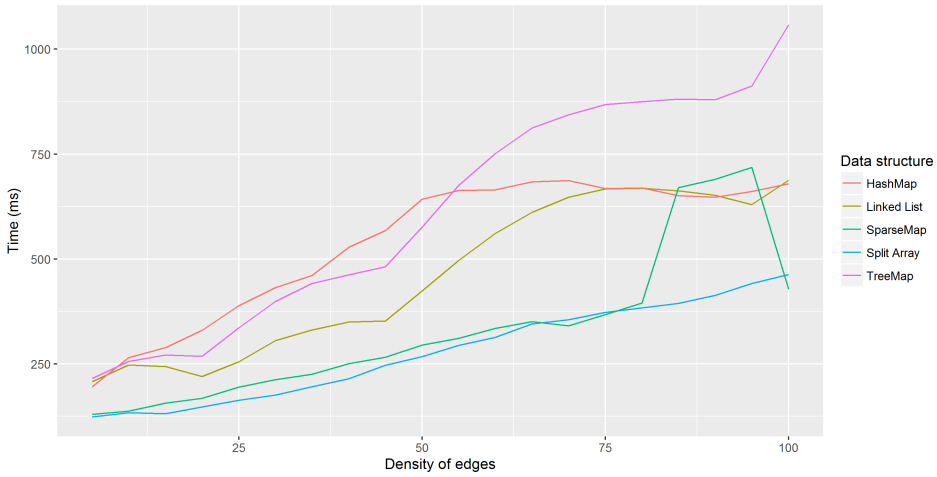



Figure 6.15: Average run time of all data structures with Edmonds-Karp on matching problem classes.

As for the Push-Relabel, the structures based on maps are the least efficient while the structures based on the *sparseset* provide better performances than the others. Nevertheless, the difference between the *splitarray* and the *sparsemap* is more pronounced. We can conclude that the most appropriate data structure for Edmonds-Karp is the *splitarray*.

**Profiler**

After profiled our code on a complete density variation graph, several observations can be made. First, we explain the poor performances of the *hashmap* and the *treemap* thanks to the Figure 6.16 which represents the number of invocations of the function *getAdjacents*.

| | **Hash map** | **Tree map** | **Simple linked list** | **Split array** | **Sparse map** |
|---|---|---|---|---|---|
| getAdjacents | 217.142 | 426.317 | 31.465 | 56.844 | 184.623 |
| total time (ms) | 1.933 | 7.714 | 322 | 163 | 1.092 |

Figure 6.16: The number of invocations of the function *getAdjacents* and its total time with Edmonds-Karp.

The *linkedlist* is slower than the *sparsemap* and the *splitarray* because its function *getCapacity* is very slow. That is what we can see in the Figure 6.17 which represents the number of invocation and the total time of the function *getCapacity*.

### 6.3.3 Ford-Fulkerson with scaling

The Figure 6.18, 6.19 and 6.20 represents the average run time of all data structures with Ford-Fulkerson with scaling. They were computed on, respectively, the density variation, the size variation and the matching problem instances.

| | Simple linked list | Split array | Sparse map |
|---|---|---|---|
| getCapacity | 220.029 | 279.480 | 783.293 |
| total time (ms) | 271 | 90 | 32 |

Figure 6.17: The number of invocations of the function *getCapacity* and its total time with Edmonds-Karp.

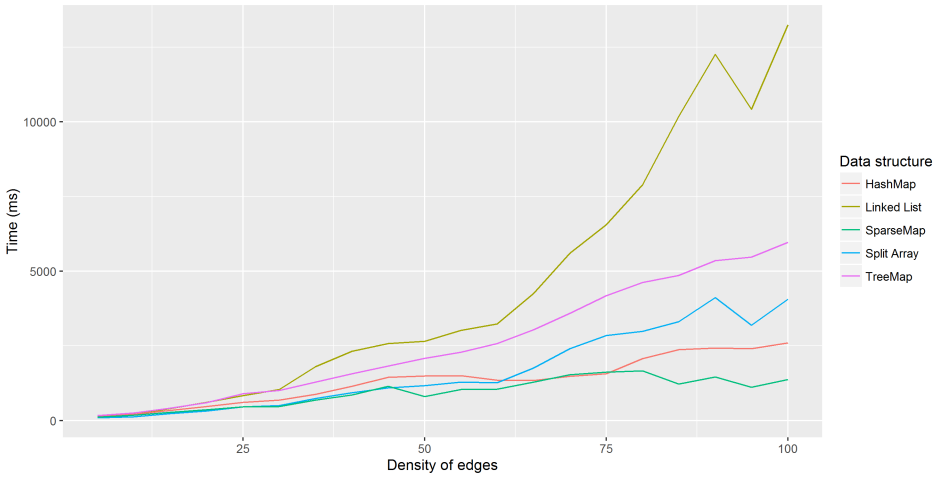



Figure 6.18: Average run time of all data structures with Ford-Fulkerson with scaling on density variation classes.

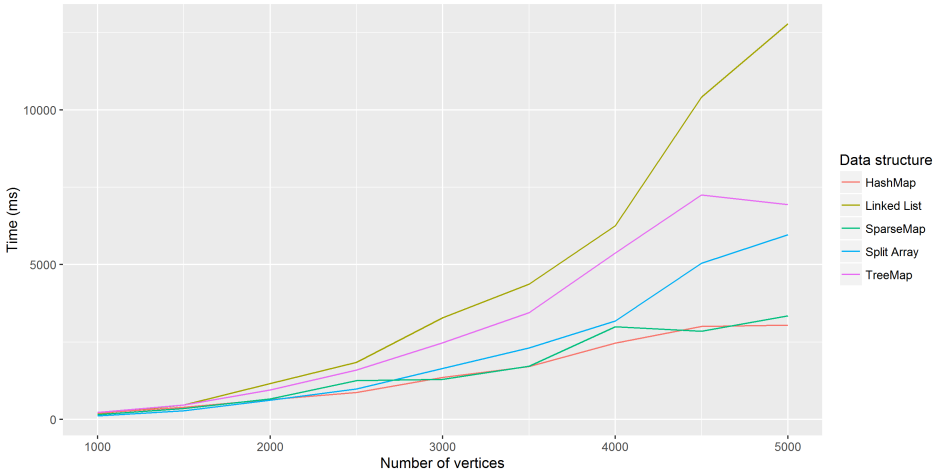



Figure 6.19: Average run time of all data structures with Ford-Fulkerson with scaling on size variation classes.

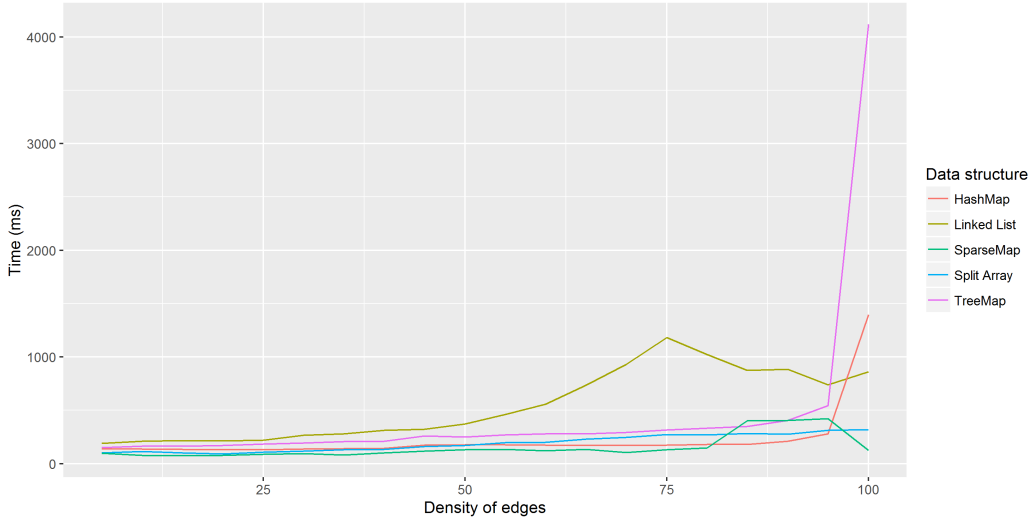



Figure 6.20: Average run time of all data structures with Ford-Fulkerson with scaling on matching problem classes.

Those figures highlight the poor performance of the *linkedlist* while the *hashmap* seems to be most appropriated for Ford-Fulkerson with scaling than for the other algorithms. We nevertheless note that the map based structures explode with a high density of edges for the matching problem instances. The structure based on the *sparseset* offer, as always, good performances.

The *sparsemap* is the most adapted data structure for Ford-Fulkerson with scaling.

### Profiler

When we look to the Figure 6.21, which represents the total time of the function *getCapacity* and its number of invocations, we understand why the *linkedlist* is so slow with Ford-Fulkerson with scaling. This figure explains also why the *splitarray* and the *treemap* are not well adapted to this algorithm, its function *getCapacity* is too slow.

The *hashmap* has a very fast function *getCapacity* but its function *getAdjacents* take too much time compared to the *sparsemap* (1146 ms for the *hashmap* and 31 ms for the *sparsemap*). This is why the *sparsemap* is the most adapted data structure for Ford-Fulkerson with scaling.

| | **Hash map** | **Tree map** | **Simple linked list** | **Split array** | **Sparse map** |
|---|---|---|---|---|---|
| getCapacity | 58.718.016 | 58.493.923 | 6.319.627 | 7.515.168 | 7.604.521 |
| total time (ms) | 1 | 2.020 | 7.489 | 1.411 | 209 |

Figure 6.21: The number of invocations of the function *getCapacity* and its total time with Ford-Fulkerson with scaling.

## 6.4 Behaviors

### 6.4.1 Edmonds-Karp

#### Density variation instances

One of the most blatant observations on Edmonds-Karp is that it is very regular, what we can observe in the Figure 6.22, which represents the run time on each density variation instance, with its best data structure, the *splitarray*. Indeed, Edmonds-Karp solves the maximum flow problem on complete graphs with $|V| = 1000$ with a run time ranging from 750 to 1000 ms.

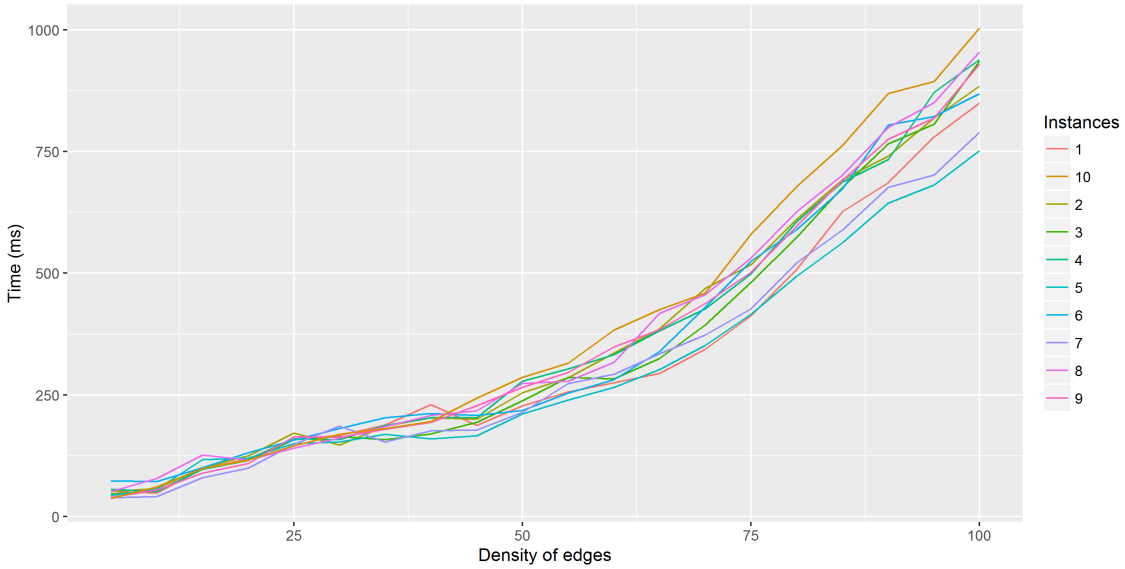



Figure 6.22: Run time of Edmonds-Karp on all density variation classes with the *splitarray*.

#### Size variation instances

The Figure 6.23 represents the run time of Edmonds-Karp on all size variation instances with the *splitarray*. Edmonds-Karp is regular with a run time ranging from 750 to 1000 ms to solve the maximum flow problem on graphs with $|V| = 5000$ and a density of edges equal to 10% ($|E| = 1249750$).

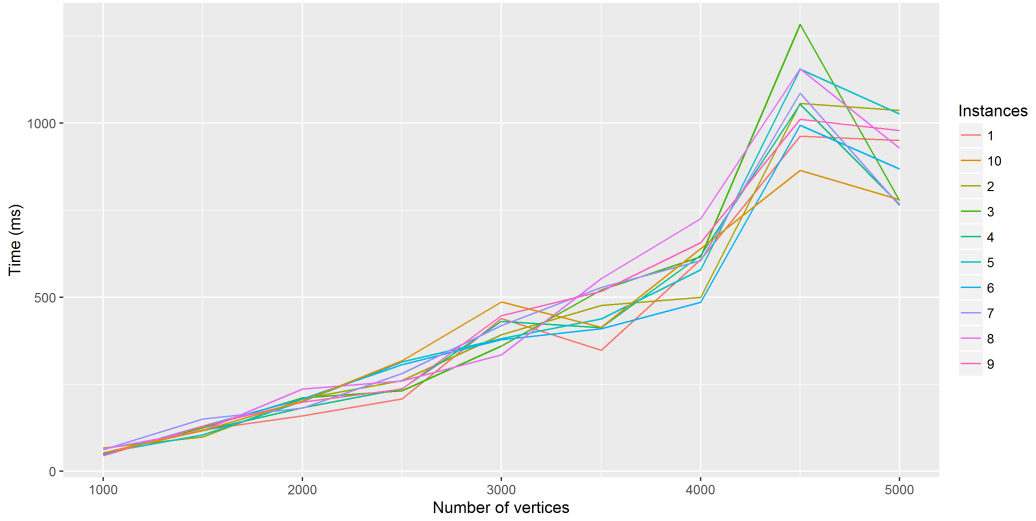



Figure 6.23: Run time of Edmonds-Karp on all size variation classes with the *splitarray*.

### Matching problem instances

As usual, Edmonds-Karp remains very regular. We can see that on the Figure 6.24 which represents the run time of Edmonds-Karp on all matching problem instances. It takes an average of 460ms to solve a matching problem instance with a maximum density of edges.

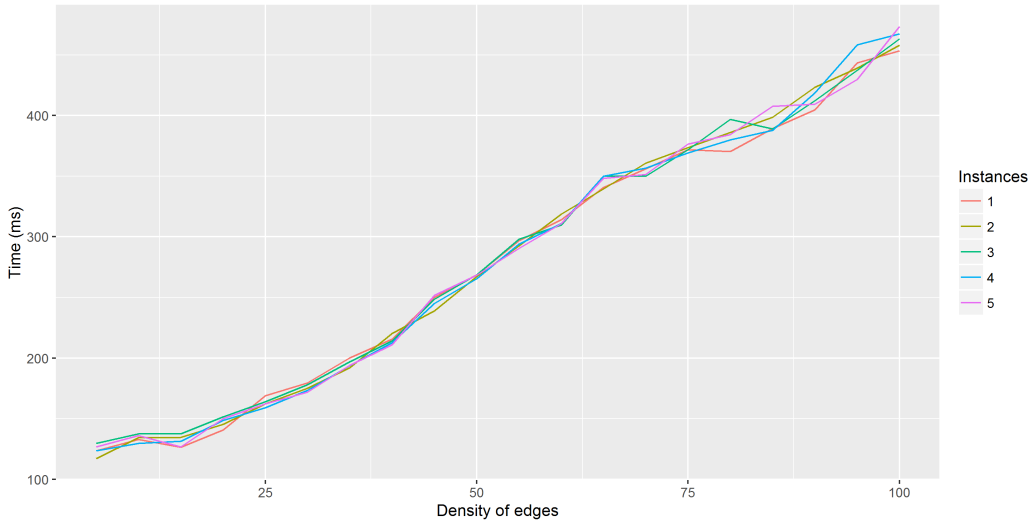



Figure 6.24: Run time of Edmonds-Karp on all matching problem classes with the *splitarray*.

### 6.4.2 Ford-Fulkerson with scaling

#### Density variation instances

The Figure 6.25 represents the run time on each density variation instance, with the *sparsemap*. Although less regular than Edmonds-Karp, Ford-Fulkerson with scaling remains stable with a run time ranging from 500 to 2500 ms to solve the maximum flow problem on complete graphs with $|V| = 1000$.

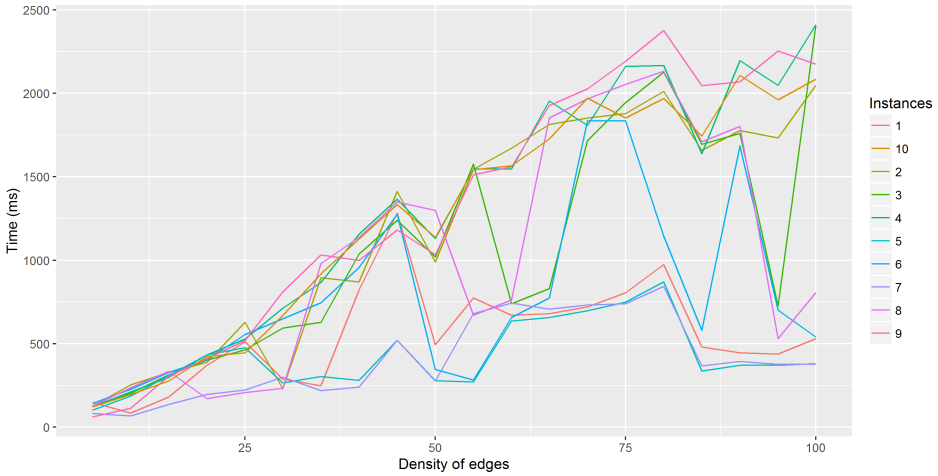



Figure 6.25: Run time of Ford-Fulkerson with scaling on all density variation classes with the *sparsemap*.

This graph also highlights the presence of two type of graphs on this type of instances. Indeed, we observe that the run time of instances 2, 4, 9 and 10 are similar. The same clustering is observable with the instances 1, 5 and 7.

#### Size variation instances

As we can see on the Figure 6.26 which represents the run time of Ford-Fulkerson with scaling on all size variation instances, it is quite regular but its performances are slightly worse than Edmonds-Karp. Indeed, Ford-Fulkerson with scaling solve the maximum flow problem on graphs with $|V| = 5000$ and a density of edges equal to 10% with a run time ranging from 3000 to 4000 ms.

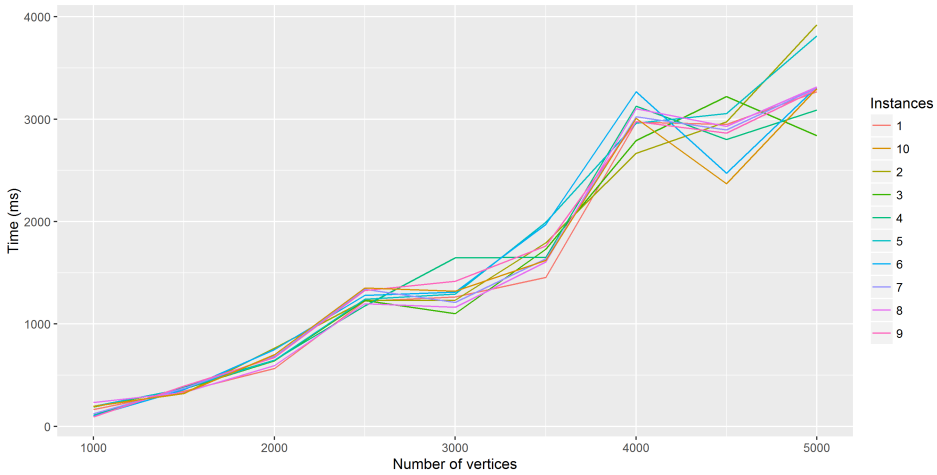



Figure 6.26: Run time of Ford-Fulkerson with scaling on all size variation classes with the *sparsemap*.

### Matching problem instances

The Figure 6.27 show that Ford-Fulkerson with scaling has a constant run time to solve this kind of instance. The spike present at the end of the graph is due to its data structure, the *sparsemap*. We also found this behaviour in the Figures 6.15 and 6.20, but we haven't found an explaination about it. It takes an average of 120 ms to solve a matching problem instance with a maximum density of edges.

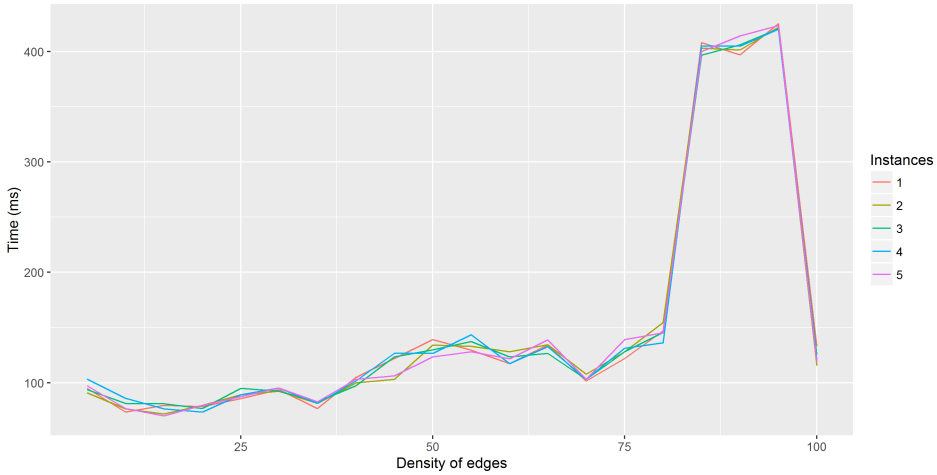



Figure 6.27: Run time of Ford-Fulkerson with scaling on all matching problem classes with the *sparsemap*.

### 6.4.3 Push-Relabel

#### Density variation instances

When we look at the results of the FIFO Push-Relabel, an observation is quite obvious : it is not regular at all. As shown in Figure 6.28, the run time on one density variation instance can vary from less than 100 ms to more than 5000 ms.

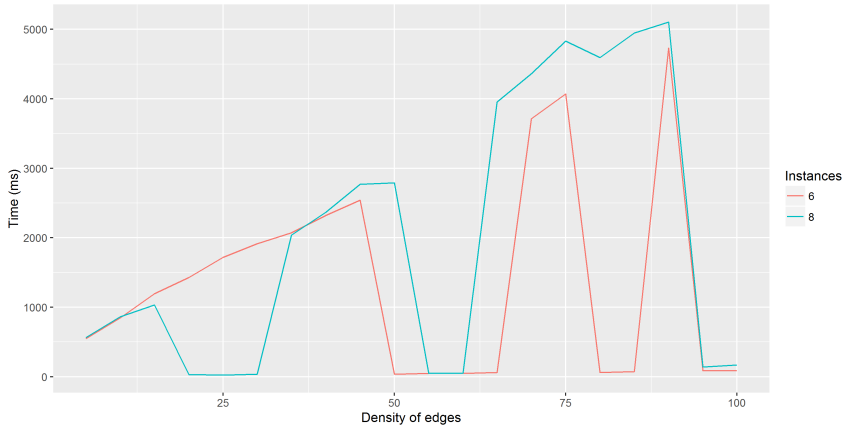



Figure 6.28: Run time of FIFO Push-Relabel on the density variation class 6 and 8 with the *splitarray*.

It may nevertheless have a stable run time but once very high and once extremely low. That is what we can observe in Figure 6.29 and Figure 6.30.

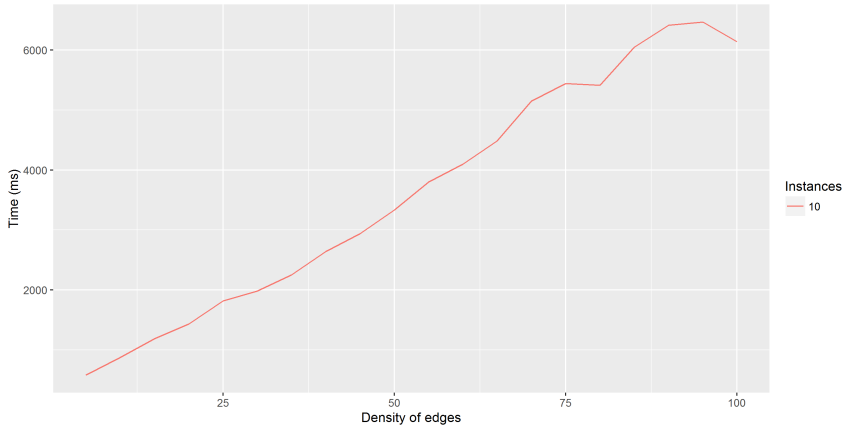



Figure 6.29: Run time of FIFO Push-Relabel on the density variation class 10 with the *splitarray*.

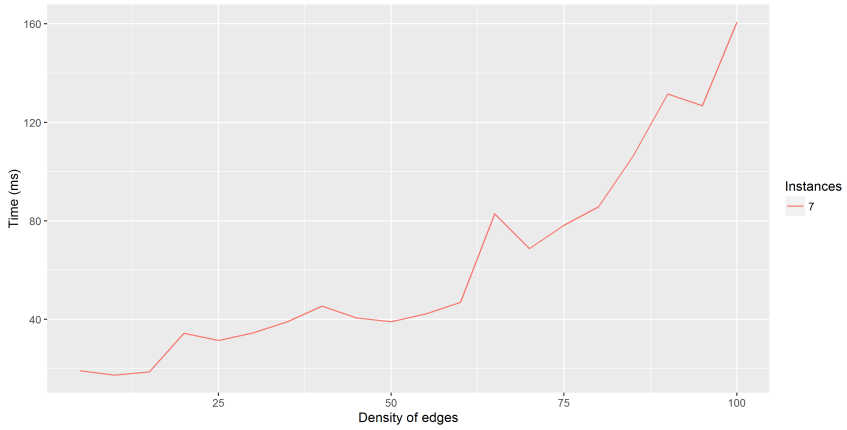



Figure 6.30: Run time of FIFO Push-Relabel on the density variation class 7 with the *splitarray*.

When displaying the run time of each density variation instance with its best data structure, we obtain a very disparate graph, as we can see in Figure 6.31. Indeed, the FIFO Push-Relabel solve the maximum flow problem on complete graphs with $|V| = 1000$ with a run time ranging from 100 ms to 6700 ms.

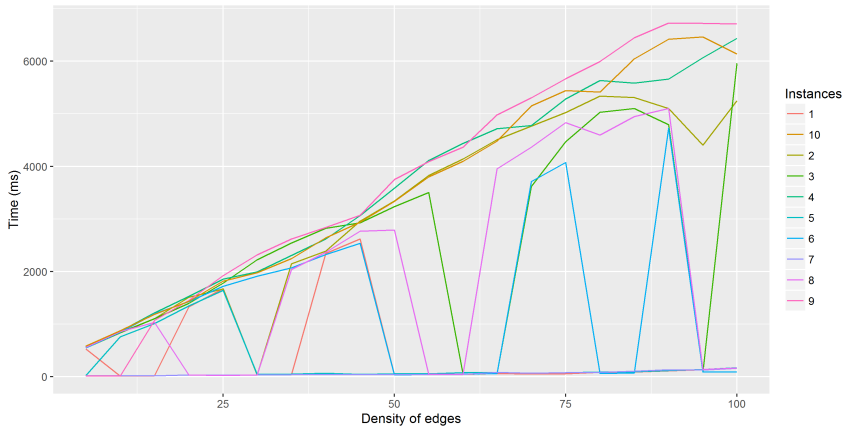



Figure 6.31: Run time of FIFO Push-Relabel on all density variation classes with the *splitarray*.

As for Ford-Fulkerson with scaling with this type on instances, we note that the instance 2, 4, 9 and 10 have high and similar run time.

### Size variation instances

Contrary to the results obtained on the density variation instances, the Highest Label Push-Relabel is regular but it offers catastrophic performances. Indeed, it has a run time ranging

from 85000 to 125000 ms to solve the maximum flow problem on graphs with $|V| = 5000$ and a density of edges equal to 10%. It is 1000 times slower than Edmonds-Karp.

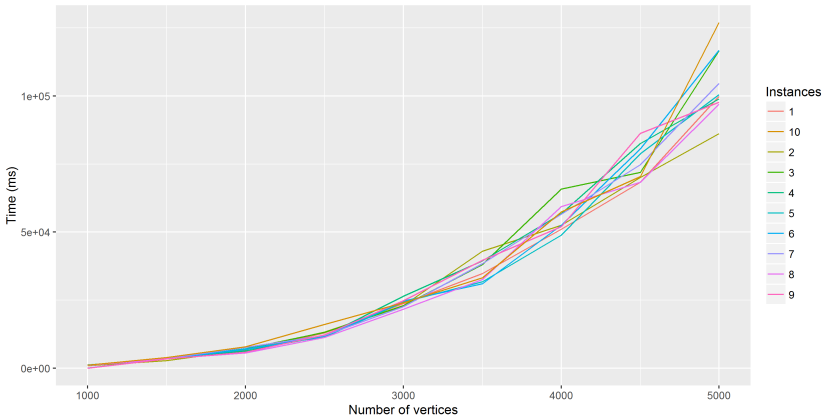



Figure 6.32: Run time of Highest Label Push-Relabel on all size variation classes with the *splitarray*.

### Matching problem instances

The Highest Label Push-Relabel excels in this type of instances. Indeed, it has a regular run time ranging from 20 to 30 ms to solve a matching problem instance with a maximum density of edges.

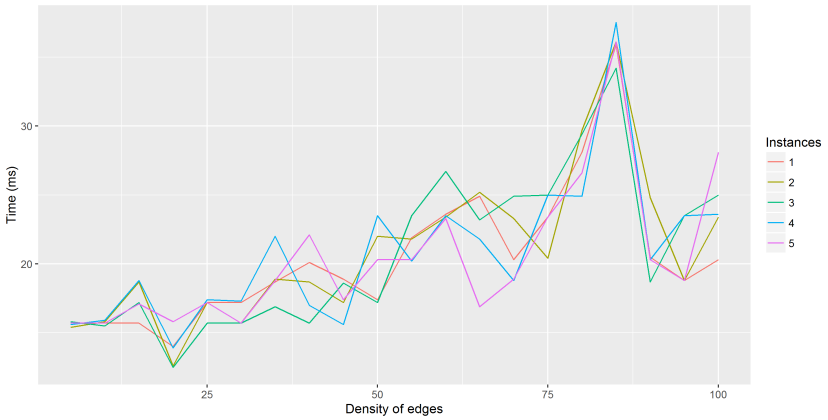



Figure 6.33: Run time of Highest Label Push-Relabel on all matching problem classes with the *splitarray*.

## 6.5 Comparison

### 6.5.1 Density variation instances

The Figure 6.34 represents, for each algorithm, the average run time on all density variation instances with its best data structure. As we can see, Edmonds-Karp seems to be the most appropriated algorithm to solve the maximum flow problem with graphs having a high-density of edges.

The complexity of FIFO Push-Relabel ($O(|V|^3)$) depend less on the number of edges than the complexity of Edmonds-Karp ($O(|V| \cdot |E|^2)$) or Ford-Fulkerson with scaling ($O(|E|^2 \cdot log(U))$). It should thus have the best performances but unfortunately, due to its irregularity, its average run time is high.

We believe that these results are due to two things. First, the worst case graph for Edmonds-Karp (a graph where each edge is used in each augmenting path of a different length), which defined its O complexity, is not possible to obtain with the generation of a density variation instance. Its complexity is a loose bound on this kind of graphs. Secondly, we saw that the Push-Relabel has a drawback, the ping pong effect. We think that the density variation instances, due to their random aspect, are favorable to the appearance of structure which allow the ping pong effect.

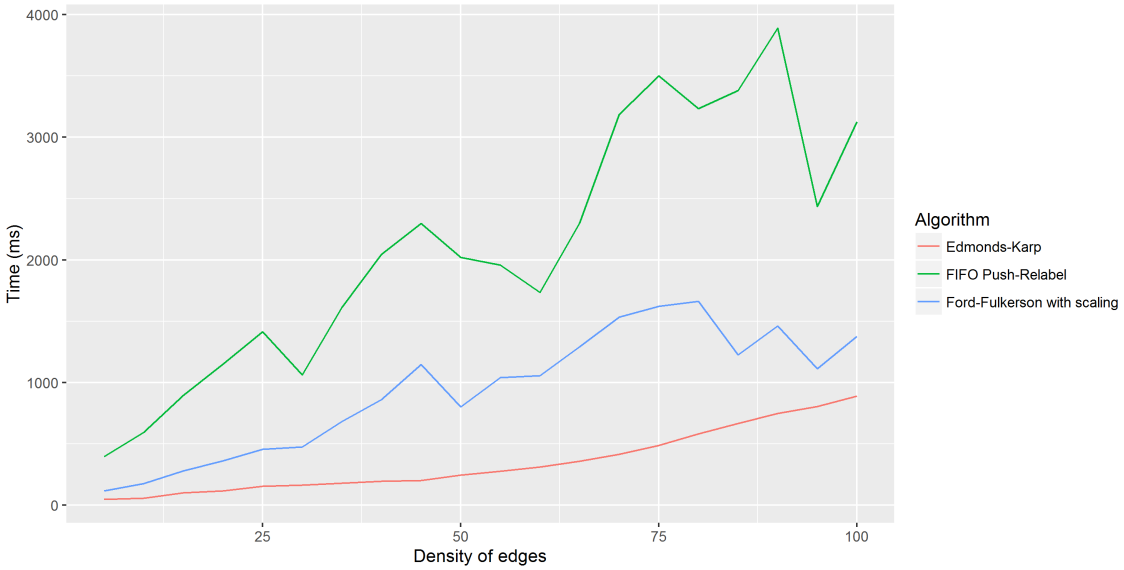



Figure 6.34: Average run time of the three algorithms on all density variation classes.

### 6.5.2 Size variation instances

The Figure 6.35 represents, for each algorithm, the average run time on all size variation instances with its best data structure. In this figure, it is clear that the augmenting path algorithms offer incomparable performances with Highest Label Push-Relabel. Edmonds-Karp is the best algorithm to solve the maximum flow problem on graphs with low density of edges and a large number of vertices.

It can be explained by their complexity. Indeed, the preflow-push algorithms ($O(|V|^2 \cdot \sqrt{|E|}$ for the Highest Label heuristic) are more dependant on the number of vertices than the augmenting path algorithms ($O(|V| \cdot |E|^2)$ for Edmonds-Karp).

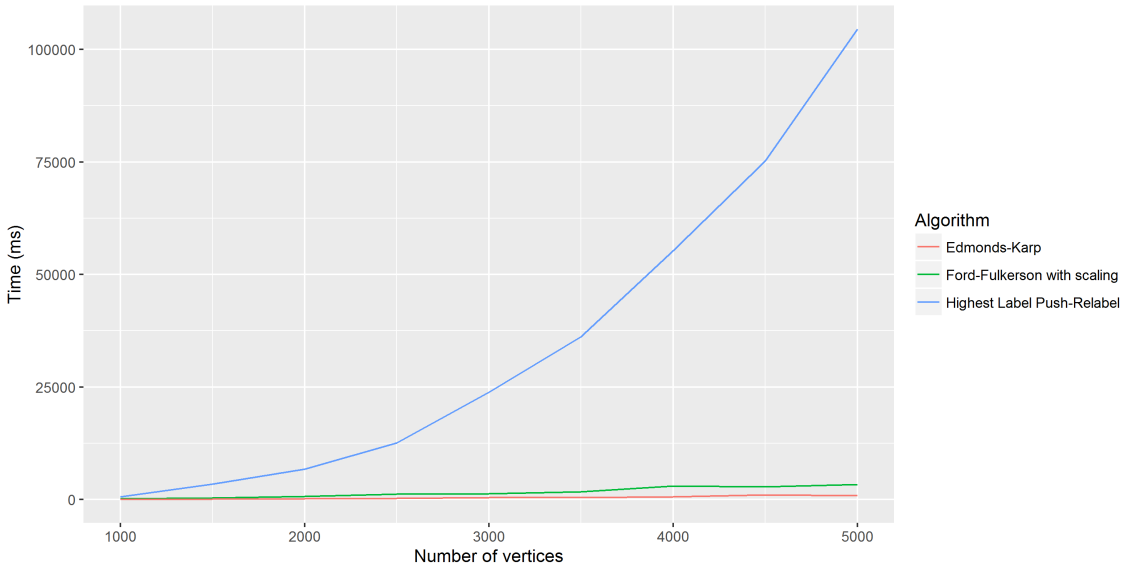



Figure 6.35: Average run time of the three algorithms on all size variation classes.

### 6.5.3 Matching problem instances

The Figure 6.36 represents, for each algorithm, the average run time on all matching problem instances with its best data structure. We notice that the run time of Highest Label Push-Relabel remains constant during the addition of edges while the run time of Edmonds-Karp increases. It is a logical behaviour in view of the complexities.

When we analyze the structure of this type of graphs, we notice the Edmonds-Karp is not adapted. Indeed, it will look for a huge number of augmenting path due to the backward edges from $R2$ to $R1$ created after a push on edge from $R1$ to $R2$. While Highest Label Push-Relabel visits once each vertex, not taking into account the number of edges.

Highest Label Push-Relabel is undoubtedly the most appropriated algorithm to solve this type of instance.

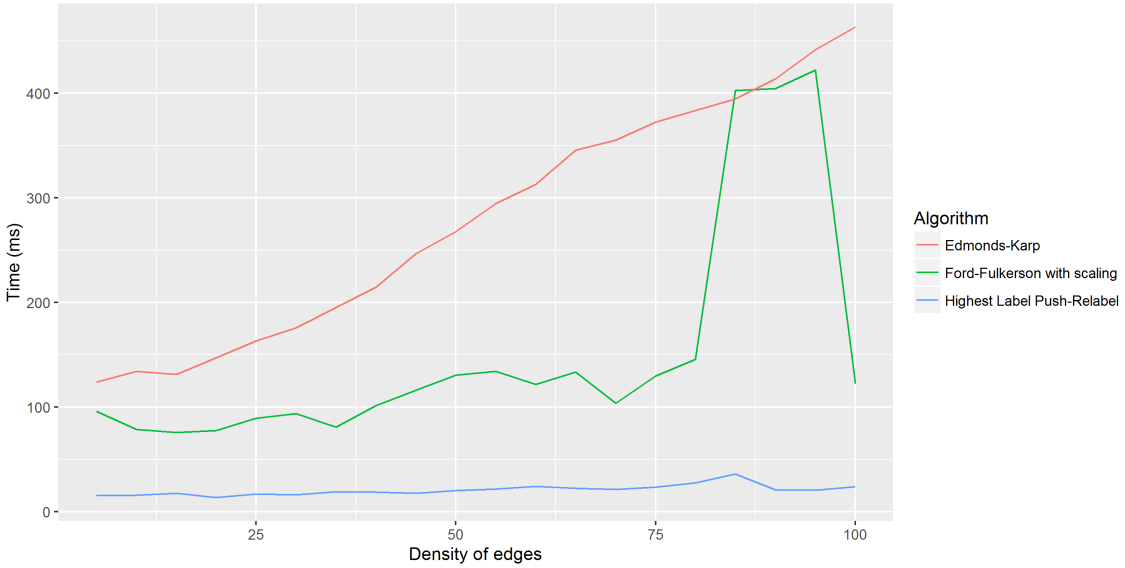



Figure 6.36: Average run time of the three algorithms on all matching problem classes.

# Conclusion

After all our work, we can draw several conclusions. First, which data structure should we use for each algorithm? Clearly, in view of the results, the data structures based on the sparse set are most suitable for maximum flow algorithms. Indeed, we have, in our analysis, focused attention on the importance of the functions *getAdjacents* (which return the set of neighbours contained in the structure) and *getCapacity* (which return the edge's capacity to a neighbour). These are the two major functions used in these algorithms. Ford-Fulkerson with scaling prefer a data structure with fast function *getCapacity* since it makes tremendous amount of calls to it due to the scaling. Edmonds-Karp and the preflow-push algorithms behave in a similar way from the point of view of the data structure, they are more efficient with a structure adapted to the function *getAdjacents*.

Depending on a graph, which algorithm should we use? One of the first conclusion is that in most cases, if you have to choose among the augmenting path algorithms, select Edmonds-Karp. Its performances are better than Ford-Fulkerson with scaling. We believe that the heuristic used in Edmonds-Karp, which is to send first the flow on the shortest augmenting path, is suitable and effective for solving the maximum flow problem. What about preflow-push algorithms? Our master thesis highlighted two things, the preflow-push algorithms can be extremely fast but are not regular at all. Indeed, if the graph allows the ping pong effect, Edmonds-Karp will perform much better than Push-Relabel. So if your graph has a random structure, we recommend to use Edmonds-Karp while if you know that the ping pong effect is not possible on your graph, Push-Relabel offer performances that Edmonds-Karp can not compete.

# Bibliography


[AMO93] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[CGI08] Raffaele Cerulli, Monica Gentili, and A Iossa. Efficient preflow push algorithms. *Computers & Operations Research*, 35(8):2694–2708, 2008.

[CSRL01] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[Dro08] Adam Drozdek. *Data Structures and Algorithms in Java*. Delmar Learning, 3rd edition, 2008.

[EK72] Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2):248–264, April 1972.

[GT88] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35(4):921–940, October 1988.

[Kem04] David Kempe. SC570: Analysis of Algorithm - Edmonds/Karp Algorithm. University of South California, USA, 2004.

[Sch15a] Pierre Schaus. LINGI2266: Advanced Algorithms for Optimization - Lecture 5 : Network flows. Université Catholique de Louvain, Belgium, 2015.

[Sch15b] Pierre Schaus. LINGI2266: Advanced Algorithms for Optimization - Lecture 8 : Constraint Programming Part 1. Université Catholique de Louvain, Belgium, 2015.

[Way01] Kevin Wayne. CS423: Theory of algorithms - Max Flows Applications. Princeton University, USA, 2001.

[Way15] Kevin Wayne. 2WO08: Graphs and Algorithms - Lecture 7 : Network Flow I. Eindhoven University of Technology, Netherlands, 2015.

[Zwi95] Uri Zwick. The smallest networks on which the ford-fulkerson maximum flow procedure may fail to terminate. *Theoretical Computer Science*, 148(1):165 – 170, 1995.

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve **www.uclouvain.be/epl**