NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

RAFAEL TAVARES BORRALHO

BSc in Computer Science and Engineering

BEYOND COMPLEXITY

AN EXPERIMENTAL STUDY OF MAXIMUM FLOW ALGORITHMS

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
December, 2024

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

BEYOND COMPLEXITY

AN EXPERIMENTAL STUDY OF MAXIMUM FLOW ALGORITHMS

RAFAEL TAVARES BORRALHO

BSc in Computer Science and Engineering

Adviser: Margarida Paula Neves Mamede
Assistant Professor, NOVA FCT

Examination Committee

Chair: Joaquim Francisco Ferreira da Silva
Assistant Professor, NOVA FCT

Rapporteur: João Pedro Lebre Magalhães Pereira
Assistant Professor, Universidade de Évora

Adviser: Margarida Paula Neves Mamede
Assistant Professor, NOVA FCT

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
December, 2024

Beyond Complexity
An Experimental Study of Maximum Flow Algorithms

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v7.1.5) [23].

Acknowledgements

First of all, I would like to express my appreciation to my adviser, Professor Margarida Mamede, for providing all the insights and knowledge on the subject matter, which have greatly impacted this work for the best. Her encouragement and calmness have guided me throughout this last year. I would also like to express my gratitude to the NOVA School of Science and Technology and to the Computer Science Department for making this possible.

And, on a more personal note, my most sincere thanks to Marta, for putting up with everything and for always being ready to gift me a little surprise. For always helping me justify impulsive purchases in Decathlon and taking the time to play beach tennis in a garden in the middle of Sintra.

Queria também agradecer à minha família por serem uma fonte de apoio constante. Por todas as discussões incansáveis, os jantares fora para desanuviar e por todos os pequenos-almoços na Graça.

Abstract

The maximum flow problem was formulated in 1954 and consists of taking the maximum amount of something from one point to another through a set of connections, whilst always respecting each connection's restrictions. It is easy to think about real-life situations in which we could apply maximum flow algorithms such as transportation networks and water distribution. Thus, this problem has become very popular with applications in numerous domains.

Since its creation, many researchers have devoted their efforts to trying to come up with different approaches to get to a solution that is better than the already existing ones. Although all the algorithms covered in this dissertation presented, at the time they were published, an improvement to the theoretical time complexity, this does not always mean they ran faster in all conditions. Therefore, one of the purposes of this dissertation is to better understand the differences between the theoretical and the experimental results. To achieve that, we empirically tested the Edmonds-Karp, the Dinic's, recursive and iterative versions, and two variants of the push-relabel algorithm, one with first in first out (FIFO) and another with the highest label protocol. These five algorithms were tested on several networks with different characteristics, including both artificial and real-life scenarios. Furthermore, another contribution of this dissertation is the simplification and explanation of some procedures of the Chen et al.'s algorithm, a recent work that achieved an incredibly low time complexity.

Keywords: Maximum flow, Algorithms, Time complexity, Experimental analysis, Comparative analysis

Resumo

O problema do fluxo máximo foi formulado em 1954 e consiste em levar a quantidade máxima de algo de um ponto para outro através de um conjunto de conexões, respeitando sempre as restrições de cada conexão. É fácil identificar situações da vida real às quais é possível aplicar algoritmos de fluxo máximo, como as redes de transporte e de distribuição de água. Deste modo, este problema tornou-se muito popular, com aplicações em diversos domínios.

Desde a sua criação, muitos investigadores têm dedicado os seus esforços a tentar encontrar diferentes abordagens para chegar a uma solução que seja melhor do que as já existentes. Embora todos os algoritmos abordados nesta dissertação tenham apresentado, na altura da sua publicação, uma melhoria teórica em termos de complexidade temporal, tal nem sempre significa um menor tempo de execução em todas as condições. Assim, um dos objetivos desta dissertação é compreender melhor as diferenças entre os resultados teóricos e os experimentais. Para este efeito, testámos empiricamente os algoritmos de Edmonds-Karp, de Dinic, versão recursiva e iterativa, e duas variantes do algoritmo *push-relabel*, uma com FIFO e outra com o protocolo da etiqueta mais alta. Estes cinco algoritmos foram testados em várias redes com diferentes características, incluindo cenários artificiais e reais. Ademais, outra contribuição desta dissertação é a simplificação e explicação de alguns procedimentos do algoritmo de Chen et al., um trabalho recente que atingiu uma complexidade temporal incrivelmente baixa.

Palavras-chave: Fluxo máximo, Algoritmos, Complexidade temporal, Análise experimental, Análise comparativa

Contents

List of Figures

List of Tables

Acronyms

BFS breadth-first search *(pp. 9–12, 14, 19, 20, 31, 42, 43)*

DFS depth-first search *(pp. 9, 11, 14, 15, 20)*
DinicIt Dinic's iterative algorithm *(pp. xi, 28, 30–32, 35–38, 40–43, 59, 61, 65, 66, 72–74)*
DinicRec Dinic's recursive algorithm *(pp. xi, 28–32, 34–38, 40–42, 59, 61, 65, 66, 72–74)*

EK Edmonds-Karp algorithm *(pp. xi, 28, 30–33, 35–43, 59, 61, 65, 66, 72–74)*

FEP forward edge probability *(pp. xi, xiii, 25–28, 32–36, 69–71)*
FIFO first in first out *(pp. v, vii, xv, 16–18, 20–22, 28, 53)*

PRFIFO push-relabel with FIFO *(pp. 28, 33, 36–43, 46, 59, 61, 62, 65–68, 72–77)*
PRHL push-relabel with highest label protocol *(pp. 28, 30, 33, 36–43, 46, 59, 61, 62, 65–68, 72–77)*

RPD relative performance degradation *(pp. xi, 28, 36, 37, 41–43)*

SEP side edge probability *(pp. xi, xiii, 25–28, 32–36, 69–71)*

1 Introduction

1.1 Context

The maximum flow problem is still to this day a meaningful topic with numerous purposes in many different areas. From transportation to water distribution, the maximum flow problem can be very useful in any domain that requires managing flows through some type of channels making good use of the already existing facilities. Essentially, consider a graph $G = (V, E)$ in which V is the set of entities, which are usually called vertices or nodes in graph theory, and E represents the set of relations between entities, often called edges. The edges have a capacity, i.e. a maximum amount of flow that they can take. If we consider two special vertices, source and sink, the maximum flow problem is nothing more than, while respecting all edges' capacities, finding one way of transporting the maximum possible amount of flow from the source vertex to the sink. Its popularity comes from the similarity with many real-life problems that humankind constantly faces.

1.2 Goals and Main Contributions

This thesis aims to study different maximum flow algorithms and compare some of them based on their performance. From old and relatively simple algorithms such as the Edmonds-Karp [13] and the Dinic's [12] to one of the most complex and modern ones, the Chen et al.'s algorithm [9, 5, 7], that takes advantage of more recent discoveries to achieve a better value for its theoretical time complexity.

An additional challenge for this thesis is to understand why the Edmonds-Karp and the Dinic's algorithms are still much more notorious and often are the ones being taught to students and used in many real-life cases. This is because, even though all algorithms have a time complexity associated, that does not tell us the full story of it since that only represents the theoretical running time in the worst case. Sometimes it is the case that, when compared empirically, a supposedly slower algorithm outruns others that may have much lower running time upper bounds.

In order to accomplish this, we will compare empirically the Edmonds-Karp, the

Dinic's and the push-relabel algorithms to better comprehend the differences between theoretical and practical running times. We will do this by generating artificial networks as well as using real-life data as a means to derive conclusions from them.

This dissertation differentiates from many other similar empirical comparisons since, apart from the empirical comparisons that were done, we try to suggest a logical explanation to the most intriguing results. Some of the highlight findings were the push-relabel common efficiency issue of flow entering a temporary loop, and each algorithm's specific network preferences. Furthermore, we explain and break down some of the sections of the first deterministic version of the Chen et al.'s algorithm [5, 6], which was just released in 2023.

1.3 Document Structure

This document is split into five chapters. Besides this first one, Chapter 2 discusses the history of the maximum flow problem as well as the details of some specific algorithms on which the dissertation will focus more. It starts with the definition of general concepts, Section 2.1, used to explain the maximum flow problem and concepts that are commonly used in the studied algorithms. Following it, there is Section 2.2, which is focused on the study of the Edmonds-Karp [13], the Dinic's [12] and the push-relabel [17], as well as detailed pseudocode for each of these algorithms so that the reader can look up all procedures if needed. It also gives an introduction to the Chen et al.'s algorithm [8, 5] and explains how the maximum flow problem is reduced to a more general one. Furthermore, in Section 2.2.6 we take a glance at the history of the maximum flow problem, from its creation until today. We look into the continuous improvements made through new and different solutions and how these impacted the running time needed to solve this problem. Finally, Section 2.3 concludes Chapter 2 with previous works which had a partially similar approach to this dissertation. The results that these works got are also discussed as well as the method that each work used to get them.

Chapter 3 talks about the experimental tests that were made. More specifically, in Section 3.1, we discuss and justify the choice of data structures used in the implementation of the networks. Section 3.2 reveals the networks that were used for testing and the aim of each of them. It also explains what tests were made, as well as how and why they were done that way. Section 3.3 discusses the results that were obtained while comparing the five implemented algorithms. The primary objective of this section is to analyze all the data produced as a result of testing on the artificial and real-life networks. We also present the most significant findings and contributions of this dissertation to the maximum flow problem and their implications. Finally, in Chapter 3, there is Section 3.4, which explains the efficiency issue of the push-relabel algorithm that caused it to perform poorer than expected. We also talk about what we believe could be done to improve it.

In Chapter 4 we give an overview of how the Chen et al.'s algorithm manages to compute maximum flows in such a different manner, compared to the remaining algorithms.

Additionally, we also provide an explanation of the data structure used in their algorithm and a general description for the first step of the algorithm.

Moreover, in Chapter 5 we conclude this dissertation by talking about the overall work done and the main contributions of it. We also provide some future work suggestions that could be done as an extension of some topics covered here. While this research has important highlights, it also opens doors for future works to continue working on them, as there are some specific topics that, due to time limitations, were not possible to accomplish. Therefore, the suggestions we make are aligned with this dissertation and are pieces of work that we believe are worth more investigation since they are of great interest and importance to this area.

2

The Maximum Flow Problem

2.1 Concepts

2.1.1 Maximum Flow Problem

Below, we define the most important concepts for this thesis [18, 17]. Definitions like network, flow network and the whole concept of flow are essential for one to be able to understand what a maximum flow truly is. We will use Figure 2.1 as an example to illustrate the concepts. Note that, in the figure, each edge has a respective value representing its capacity.

Network: A network is represented as a pair $G = (V, E)$. V is a finite set of vertices, and E is a finite set of pairs of distinct vertices called edges. For simplicity, we denote $|V| = n$ and $|E| = m$. There are two special vertices called source and sink that are respectively identified as s and t. The remaining vertices are called internal. Furthermore, every edge (u, v) has an associated capacity represented by $c(u, v)$.

In the context of this thesis we will only consider weighted and directed networks in which, for every vertex v, there is a path from s to t that goes through v. Moreover, it is assumed all edges have non-negative capacities and that there are not two edges with the same origin and destination vertices. All the networks considered share a characteristic, which is $m \geq n - 1$. This is true due to the guarantee that all vertices belong to a path from the source to the sink.

Flow Network: A flow network is a network usually generated from another network $G = (V, E)$, as defined above. It can be represented as $G' = (V, E')$ as the set of vertices remains untouched from the original network. However, for every existing edge $e = (u, v) \in E$ there must also be an edge $e' = (v, u)$. If $(v, u) \notin E$ we must add it to the flow network with $c(v, u) = 0$. Thus, we can say that $E \subseteq E'$.

An example of a flow network generated from another network can be seen in Figure 2.1.

Flow: In a flow network $G' = (V, E')$, a flow is a function $f : E' \to \mathbb{R}$ that complies with the following rules:

- Capacity: $-c(u, v) \leq f(u, v) \leq c(u, v), \forall (u, v) \in E'$

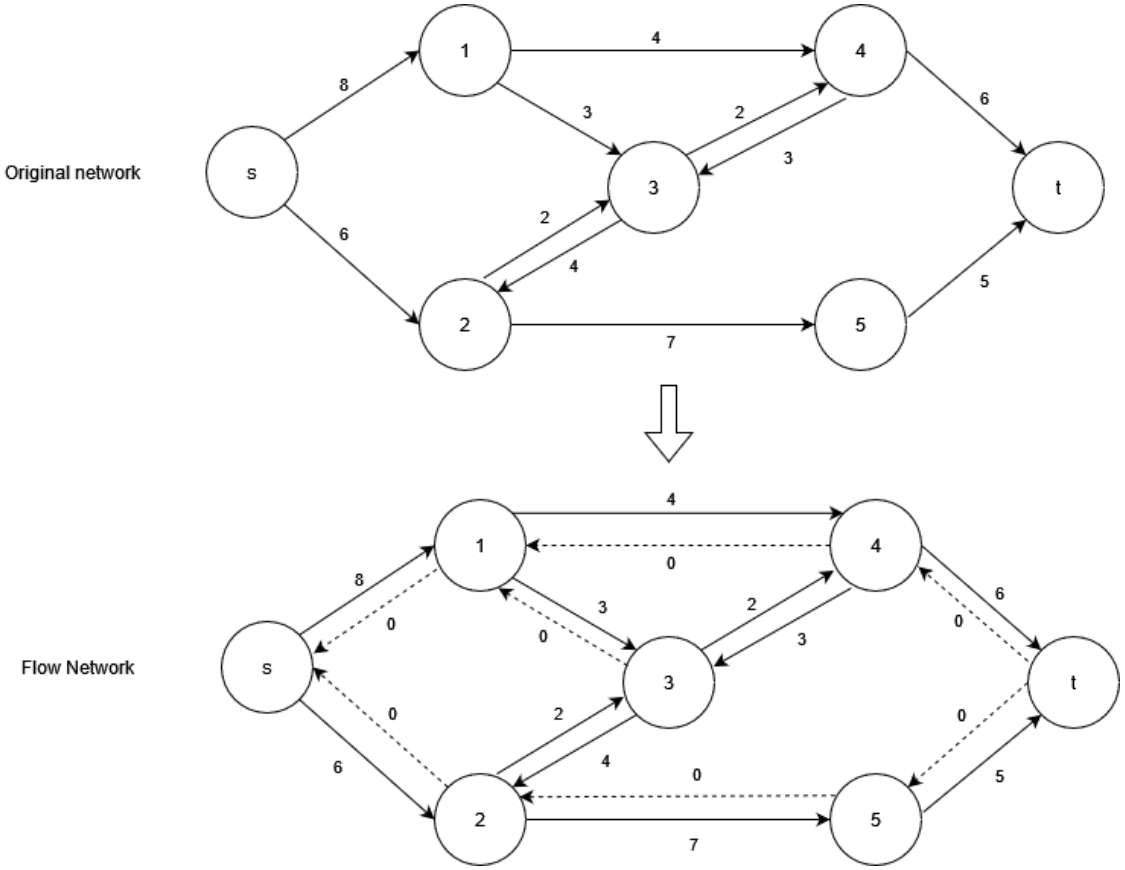


Figure 2.1: Example of flow network - In this figure, the number on each edge represents its capacity. The vertices s and t represent respectively the source and sink vertices and the internal vertices are represented as numbers with no specific order. Edges represented by dotted arrows indicate that they did not exist on the original network and were only added on the creation of the flow network.

- Symmetry: $f(u,v) = -f(v,u), \forall (u,v) \in E'$
- Conservation: $\forall u \in V \setminus \{s,t\} : \sum_{(u,v)\in E'} f(u,v) = 0$

The domain of this function f is commonly defined as $V \times V$. In that case, the vertex pairs that do not form an edge in E' get a value of 0.

The flow value $|f|$ represents the amount of flow that comes from the source and arrives at the sink. Therefore it can be computed either as:

$$|f| = \sum_{(s,v)\in E'} f(s,v)$$

or

$$|f| = \sum_{(v,t)\in E'} f(v,t)$$

Maximum Flow: A maximum flow of a network is a flow with the largest possible flow value. There may be several flows that are maxima.

Maximum Flow Problem: The maximum flow problem is the problem of finding, in a flow network, a maximum flow. Nowadays this is achievable with a number of different algorithms.

2.1.2 Other Concepts

With the maximum flow problem defined, we will now define concepts that are necessary for the study of algorithms [18, 17, 11]. Now using Figure 2.2, in which we use the notation flow/capacity for each edge.

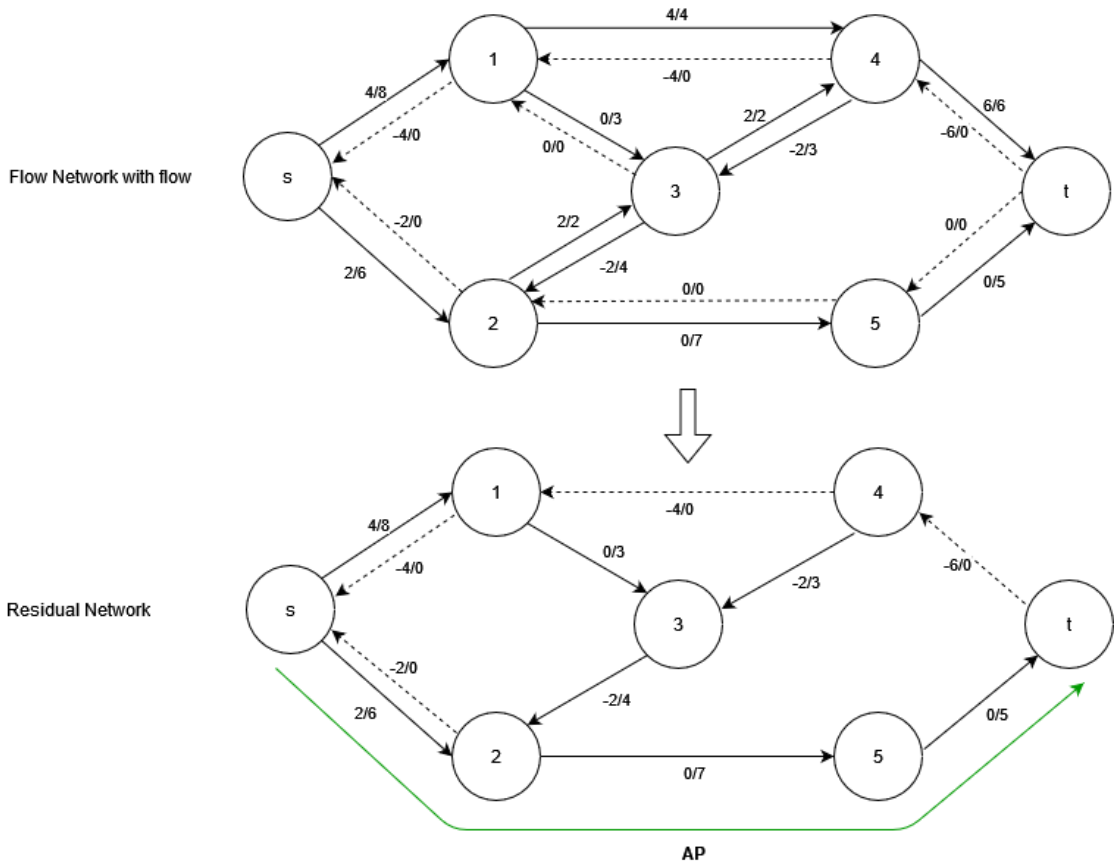


Figure 2.2: Example of flow and residual network - In this figure, we represent networks with flow and therefore use, for each edge, the format flow/capacity.

Residual Capacity: It refers to an edge $e = (u, v)$ and it is the difference between the edge's capacity and flow. Therefore this value is used to represent how much flow can still be taken through a given edge. The residual capacity can be represented as $r(u, v) = c(u, v) - f(u, v)$. For instance, in Figure 2.2, edge $(s, 1)$ has a residual capacity of $8 - 4 = 4$.

Residual Network: A residual network is described as a way of representing a network $G = (V, E)$ containing only the edges in which the residual capacity is greater than zero. This can be defined as $R = (V, E_r)$, with $E_r = \{(u, v) : (u, v) \in E \wedge r(u, v) > 0\}$.

Augmenting Path: An augmenting path is any path from the source to the sink vertex on a given residual network. This way we ensure that only edges with residual capacity greater than zero are taken. If we take the residual network depicted in Figure 2.2, there is an augmenting path AP following the edges $(s, 2), (2, 5), (5, t)$.

Blocking Flow: A blocking flow is a flow which, for every s to t path saturates at least one edge. This means that for every path from s to t in the flow network, the residual capacity of at least one edge of the path is zero. Therefore, there are no augmenting paths in a blocking flow.

Bottleneck: When talking about maximum flow algorithms, a bottleneck edge is an edge with the smallest residual capacity of a given augmenting path and thus preventing more flow from being taken. If we take the augmenting path AP in Figure 2.2, its bottleneck edge would be $(s, 2)$ given that $\min\{6-2, 7-0, 5-0\} = 4$.

2.2 Algorithms

In this section, we will analyze a few algorithms. In all of them except for the last one, from Chen et al. [6, 5], we start by creating a flow network by adding the missing edges. Additionally, after creating the flow network, those same algorithms create an initial flow by assigning 0 to all edges. These two initial steps are implemented in Algorithm 1.

Algorithm 1 $buildNetworkFlow(network)$

```
for ∀(u, v) ∈ E do
  if (v, u) ∉ E then
    add edge (v, u) to E with c(v, u) = 0
  end if
end for
for ∀(u, v) ∈ E do
  f(u, v) = 0
end for
//returns the initial flow
return f
```

2.2.1 Ford-Fulkerson Method

2.2.1.1 Overview

The Ford-Fulkerson [14] is usually referred to as a method instead of an algorithm since it only states the approach one should take in a general way and it does not fully specify how all its details should be handled. Thus, it can vary from one implementation to another.

The idea of this method can be explained very simply. As long as there is at least one augmenting path, we send the maximum possible flow through it. When the point on which there is no augmenting path available is reached, the method is over [22]. This method can be simply put as:

1. Build the network flow and create initial flow
2. Find any augmenting path
3. Update the flow of the edges that were used
4. Repeat steps 2-3 until there is not an available augmenting path.

2.2.1.2 Complexities

As this method does not specify the way an augmenting path p should be obtained, there is more than one way to do this (for example, breadth-first search (BFS) and depth-first search (DFS)) and its running time depends on that.

If we assume our network's edges have integer capacities, on each new iteration we are guaranteed that the flow will be increased by at least one. If we think about this thoroughly, we can recognize that in the worst case, each iteration will increase the flow value by one and thus the method will have $|f|$ iterations, with $|f|$ representing the maximum flow value. Finding an augmenting path, if using BFS or DFS, will take at most $O(m+n)$. Since we have for our networks that $m \geq n-1$, we can simplify it to $O(m)$ and thus the time complexity is $O(|f| \cdot m)$ [15].

2.2.2 Edmonds-Karp Algorithm

2.2.2.1 Overview

The Edmonds-Karp algorithm [13] was introduced by Jack Edmonds and Richard Karp in 1972 and it is still today one of the most known algorithms to solve the maximum flow problem. This algorithm implements the previously seen Ford-Fulkerson method by taking its main idea of finding as many augmenting paths as possible. However, they chose to do this in a slightly different manner. Instead of simply keep looking for any augmenting path as it was stated in the Ford-Fulkerson method, they figured it would be more beneficial to look for the shortest augmenting paths. This turned out to be a great idea as it sets an upper bound for the number of iterations of $O(nm)$.

The main ideas of this algorithm (see Algorithm 2) can be described as:

1. Build the network flow and create initial flow (Algorithm 1)
2. Find a shortest augmenting path using BFS (Algorithm 3)
3. Update the flow of the edges that were used (Algorithm 4)
4. Repeat steps 2-3 until there is not an available augmenting path.

Algorithm 2 $edmondsKarp(network)$

```
//Creates the network flow and the initial flow
f = buildNetworkFlow(network)
while findAugmentingPath(network, f) > 0 do
  continue;
end while
//returns the max flow
return f
```

Algorithm 3 $findAugmentingPath(network, f)$ via BFS

```
Create empty queue to perform BFS
Create array prev to hold previous edge with size n
Create array bottleneck to hold bottleneck values with size n
Create boolean array visited with size n and initialize all cells with false
visited[s] = true
queue.enqueue(s)
bottleneck[s] = ∞
while ¬queue.isEmpty() do
  currNode = queue.dequeue()
  for ∀(currNode, v) ∈ E do
    if r(currNode, v) > 0 ∧ ¬visited[v] then
      prev[v] = (currNode, v)
      bottleneck[v] = min{r(currNode, v), bottleneck[currNode]}
      if v == t then
        updateEdges(network, f, prev, bottleneck[t])
        return bottleneck[t]
      end if
      queue.enqueue(v)
      visited[v] = true
    end if
  end for
end while
return 0
```

2.2.2.2 Complexities

The upper hand that the Edmonds-Karp has on Ford-Fulkerson is its time complexity as it is independent of the maximum flow of the network and can run on polynomial time even in the worst case. This algorithm has at most $O(nm)$ iterations of the outer loop, see Algorithm 2, i.e., there will be at most $O(nm)$ flow augmentations. On each iteration, in order to find a new shortest augmenting path, which can be seen in Algorithm 3, BFS is used, which has a time complexity of $O(n + m)$. However, this can be simplified to $O(m)$ since, on the networks we are working with, $m \geq n - 1$. Multiplying it, we get the final time complexity of the algorithm $O(nm^2)$ [13, 22].

Algorithm 4 $updateEdges(network, f, prev, bottleneck)$

```
currNode = t
while currNode ≠ s do
  currEdge = prev[currNode]
  f(currEdge) = f(currEdge) + bottleneck
  f(reverse(currEdge)) = f(reverse(currEdge)) − bottleneck
  currNode = currEdge.origin
end while
```

2.2.3 Dinic's Algorithm

Algorithm 5 $dinic(network)$

```
//Creates the network flow and the initial flow
f = buildNetworkFlow(network)
while true do
  labels = labelNodes(network, f)
  if labels[t] == ∞ then
    //returns the max flow
    return f
  end if
  if is recursive version then
    Create array starts with size n and initialize all cells with 0
    findAllPaths(network, f, labels, starts)
  else
    findAllPaths(network, f, labels)
  end if
end while
```

Originally invented in 1969 and later published in 1970 by the Israeli Yefim Dinitz [12], this is a strong polynomial algorithm implemented over the generic Ford-Fulkerson method to compute maximum flows. It makes use of both BFS, to label the vertices by their levels, and DFS to find new augmenting paths, as well as concepts such as blocking flow and layered network. To fully grasp the idea of a layered network, we must first understand how to determine the level of a vertex v, denoted by $d(v)$. The level of a vertex is given by the number of edges along the shortest paths from the source to that specific vertex in a given residual network. With this concept in mind, a layered network is a subnetwork of the residual network, composed of all edges (u, v) in the flow network such that $d(v) = d(u) + 1$, in order to only keep edges that intuitively seem to take the flow one step closer to the sink. In Figure 2.3 there are two different examples of the transformation from a flow network to a layered network. Note that, in the figure, the level of a vertex in a layered network is represented on the vertex itself. Finally, the idea of a blocking flow is essential to understand when we should redo the layered network as it indicates the absence of more augmenting paths.

We implemented two versions of this algorithm. The first one being a recursive

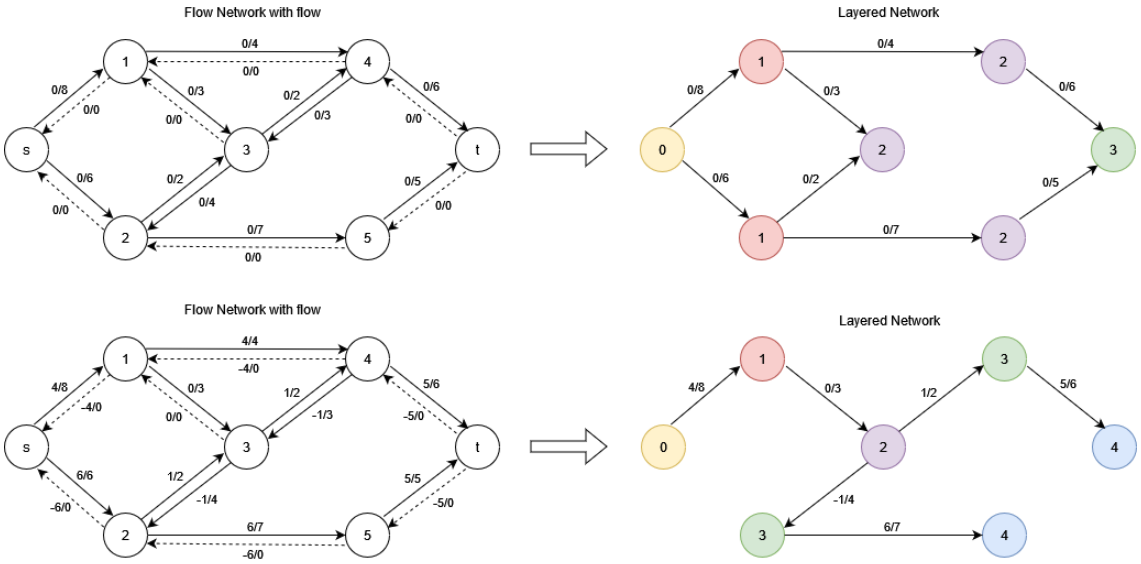


Figure 2.3: Network to layered network - In this figure, we show the creation of a layered network given a flow network. On the layered network, the numbers present on each vertex v represent the distance from s, i.e. the minimum number of edges that need to be taken to go from s to v in the corresponding residual network.

Algorithm 6 $labelNodes(network, f)$ via BFS

Create empty *queue* to perform BFS
Create boolean array *visited* with size n and initialize all cells with false
Create array *labels* with size n
$queue.enqueue(s)$
$visited[s] = true$
$labels[s] = 0$
$labels[t] = \infty$
while $\neg queue.isEmpty()$ **do**
  $currNode = queue.dequeue()$
  for $\forall(currNode, v) \in E$ **do**
    if $r(currNode, v) > 0 \wedge \neg visited[v]$ **then**
      $queue.enqueue(v)$
      $visited[v] = true$
      $labels[v] = labels[currNode] + 1$
    end if
  end for
end while
return *labels*

version with some intelligence added to it and the second one is an iterative version. The intelligence added to the classic recursive algorithm is essentially remembering previous paths already explored to avoid wasting time re-calculating them only to get to an edge with no residual capacity left. By doing this, we are potentially saving our algorithm from considering numerous edges that would not be useful since they were already explored. This is implemented with the help of an array *starts* that holds a number for each vertex that indicates which outgoing edges have not been explored yet. These two variants vary on the exploration of paths and thus we have two different implementations of the method that is used to explore paths. The recursive method can be seen on Algorithm 7, and the iterative version is on Algorithm 9. Keep in mind that we assume an implementation of the network using adjacency lists (see Section 3.1 for the graph implementation discussion) and thus, for a given vertex v, the expression $network[v]$ retrieves the list of edges whose origin is v. Therefore, the condition $starts[currNode] < network[currNode].size()$ present in Algorithm 8 checks if there are still outgoing edges from v that are yet to be explored, and $network[currNode][starts[v]]$ takes the first outgoing edge from v that still needs to be searched. This also explains why we increment $starts[v]$, meaning a new outgoing edge was explored and the incremented number now indicates the next edge to be searched.

Note that in the description of the algorithm, we talk about the creation of a layered network as it is great to visualize the algorithm. However, in practice, it is not necessary to create a new network since it is possible to validate which edges make progress through the network only using the vertices' levels. Therefore, in Algorithm 6, we only label the vertices by their levels and use the labels to identify whether an edge can be used or not.

Algorithm 7 $findAllPathsRec(network, f, labels, starts)$ via DFS (recursive version)

```
Create boolean array visited with size n
Create array bottleneck to hold bottleneck values with size n
hasBlockingFlow = false
while ¬hasBlockingFlow do
  Initialize all cells from visited with false
  if ¬findPathRec(network, f, s, visited, starts, bottleneck, labels)
    (see Algorithm 8) then
    hasBlockingFlow = true
  end if
end while
```

Algorithm 8 $findPathRec(network, f, currNode, visited, starts, bottleneck, labels)$

```
visited[currNode] = true
while starts[currNode] < network[currNode].size() do
  (currNode, v) = network[currNode][starts[currNode]]
  if ¬visited[v] ∧ r((currNode, v)) > 0 ∧ labels[currNode] == labels[v] − 1 then
    bottleneck[v] = min{r(currNode, v), bottleneck[currNode]}
    if v == t ∨ findPathRec(network, f, v, visited, starts, bottleneck, labels) then
      f((currNode, v)) = f(currNode, v) + bottleneck[t]
      f(reverse((currNode, v))) = f(reverse((currNode, v))) − bottleneck[t]
      return true
    end if
  end if
  starts[currNode] = starts[currNode] + 1
end while
return false
```

The Dinic's algorithm (see Algorithm 5) works as follows:

1. Build the network flow and create initial flow (Algorithm 1)

2. Label the vertices by their levels and create the layered network (Algorithm 6)

3. a) Find an augmenting path using DFS on the layered network (Algorithm 9 or 8)

   b) Update the flow of the edges that were used (Algorithm 4 or 8)

   c) Repeat steps (a) and (b) until a blocking flow is reached (Algorithm 9 or 7)

4. Repeat steps 2-3 until the sink is never reached on the layered network

2.2.3.1 Complexities

The Dinic's algorithm time complexity is $O(n^2m)$. This is a result of a pre-processing stage, which consists of building the layered network in time $O(m+n)$ as it requires a BFS traversal, which later helps get the complexity of each iteration (step 3) down to $O(nm)$. The number of iterations needed is established as follows. Each time a layered network is created, the number of edges on the shortest augmenting path strictly increases. Since the longest possible augmenting path has a length of $n-1$, we get an upper bound for the number of iterations of $O(n)$. This results in the overall time complexity of $O(n(m+n+nm))$ which can be simplified to $O(n^2m)$.

Unlike the Edmonds-Karp, this algorithm grows with the square of the number of vertices. This makes it a more effective algorithm for more dense networks [19, 3].

Algorithm 9 $findAllPathsIt(network, f, labels)$ via DFS (iterative version)

```
hasBlockingFlow = false
Create empty stack to perform DFS
Create boolean array visited with size n
Create array prev to hold previous edge with size n
Create array bottleneck to hold bottleneck values with size n
while ¬hasBlockingFlow do
  Initialize all cells from visited with false
  stack.push(s)
  bottleneck[s] = ∞
  bottleneck[t] = −1
  while ¬stack.isEmpty() do
    currNode = stack.pop()
    if ¬visited[currNode] then
      visited[currNode] = true
      for ∀(currNode, v) ∈ E do
        if ¬visited[v] ∧ r(currNode, v) > 0 ∧ labels[currNode] == labels[v] − 1 then
          prev[v] = (currNode, v)
          bottleneck[v] = min{r(currNode, v), bottleneck[currNode]}
          if v == t then
            updateEdges(network, f, prev, bottleneck[t])
            stack.makeEmpty()
          else
            stack.push(v)
          end if
        end if
      end for
    end if
  end while
  if bottleneck[t] == −1 then
    hasBlockingFlow = true
  end if
end while
```

2.2.4 Push-relabel Method

2.2.4.1 Overview

The push-relabel algorithm [17] was officially created by Goldberg and Tarjan in October 1988 and its name is inspired by the two essential operations push and relabel. As the name of the paper that officially announced this algorithm states, this was a new approach to the maximum flow problem. It uses an idea of height in order to guide the flow into the sink, which works as follows. Each vertex v has a respective height label, given by $h(v)$, and a vertex can only push flow to another if its height exceeds that of the latter. This concept can be visualized if we imagine that edges and vertices are respectively represented as water pipes and joints. This way, only edges $(u, v) \in E$ such that $h(u) > h(v)$ and $r(u, v) > 0$ can push flow. These are called admissible edges.

Algorithm 10 $pushRelabel(network)$

```
//Creates the network flow and the initial flow
f = buildNetworkFlow(network)
Create empty queue activeNodes with size n
Create array heights with size n
Create array excess with size n
heights[s] = n
excess[s] = 0
for ∀v ∈ V \ {s} do
    heights[v] = 0
    excess[v] = 0
end for
for ∀(s, v) ∈ E do
    f(s, v) = c(s, v)
    f(v, s) = −c(s, v)
    excess[v] = c(s, v)
    if v ≠ t then
        activeNodes.enqueue(v)
    end if
end for
while ¬activeNodes.isEmpty() do
    currNode = removeActiveNode(activeNodes, heights)
    discharge(network, f, currNode, activeNodes, heights, excess)
end while
//returns the max flow
return f
```

Algorithm 11 $removeActiveNode(activeNodes, heights)$

```
if protocol is FIFO then
    Remove and return from activeNodes the vertex that first entered the list
else
    Remove and return from activeNodes a vertex with the highest value on heights
end if
```

In the first stage of the algorithm (which can be seen in more detail in Algorithm 10), it maintains a preflow that can temporarily break the flow conservation constraint. As it only satisfies the relaxed version that requires all vertices to have a non-negative excessive flow. Therefore the concept of excessive flow is required, which indicates the amount of flow that has arrived at a given vertex v but has not yet left, given by $e(v)$.

Initially, the preflow is initialized as well as the initial values for the vertices. Firstly, the source's height should be equal to the number of vertices on the network and the remaining vertices should have a height label of 0. Furthermore, all vertices start with an initial excessive flow of 0. The flow of the edges starting from the source should be set equal to their respective capacity values.

In the second stage, the mentioned preflow will be transformed into a maximum

Algorithm 12 $discharge(network, f, v, activeNodes, heights, excess)$

while $excess[v] > 0$ **do**
  for $\forall (v, w) \in E$ **do**
    if $heights[v] > heights[w] \wedge r(v, w) > 0$ **then**
      $push(network, f, v, w, activeNodes, excess)$
      if $excess[v] == 0$ **then**
        return;
      end if
    end if
  end for
  $relabel(network, f, v, heights)$
end while

Algorithm 13 $relabel(network, f, v, heights)$

$heights[v] = \min\{heights[w] + 1 : (v, w) \in E \wedge r(v, w) > 0\}$

flow by repeatedly removing the excessive flow from the active vertices, which are vertices that are not the source nor the sink and their excess flow is greater than zero, i.e. $v \in V \setminus \{s, t\} : e(v) > 0$. The algorithm ends when all the internal vertices have no excessive flow.

Algorithm 14 $push(network, f, u, v, activeNodes, excess)$

$\delta = \min\{r(u, v), excess[u]\}$
$f(u, v) = f(u, v) + \delta$
$f(v, u) = f(v, u) - \delta$
$excess[u] = excess[u] - \delta$
$excess[v] = excess[v] + \delta$
if $v \neq s \wedge v \neq t \wedge \neg activeNodes.contains(v)$ **then**
  $activeNodes.enqueue(v)$
end if

The process of removing excessive flow from a vertex is called discharging (Algorithm 12) and it makes use of both push (Algorithm 14) and relabel (Algorithm 13) operations. In a simple scenario, pushing flow through one or more admissible edges to adjacent vertices (push operation) would do it. However, it is not always that simple. The relabeling operation might be necessary in a circumstance where a vertex has no admissible outgoing edges due to its low height. In that case, we relabel its height (relabel operation) making it larger and causing at least one edge to turn into an admissible one through which some excessive flow can be pushed into [11].

It is important to note that the order in which the vertices to discharge are chosen depends on the implementation. In this dissertation, we have two different variants. A first one using a FIFO protocol and the second one choosing vertices with the highest label first. Whereas the first protocol is implemented using a simple FIFO queue, the highest label protocol uses a priority queue that keeps the queue in decreasing order of label value.

Therefore the only difference between these two implementations is the order in which we choose the vertices that need to be discharged (see Algorithm 11).

In general, this algorithm runs as described (see Algorithm 10):

1. Build the network flow and create initial flow (Algorithm 1)
2. Initialize all vertices and the edges incident on the source with the initial values
3. Vertices with excessive flow push their flow (Algorithm 14) through admissible edges to adjacent vertices. If this cannot be done due to height levels, the vertex's height is relabeled (Algorithm 13). This step is implemented in Algorithm 12
4. Repeat step 3 until there are no more internal vertices with excessive flow

2.2.4.2 Complexities

The complexity of the push-relabel algorithm depends on the implementation. One of the factors that affects this is the order in which we choose to manage the active vertices. One way to do it is using a FIFO queue to hold the active vertices, which results in a time complexity of $O(n^3)$. Another studied option is selecting the active vertices in decreasing order of their height labels, which gets a running time of $O(n^2\sqrt{m})$ [17, 11].

2.2.5 Chen et al.'s Algorithm

The Chen et al.'s algorithm [6, 5] is a minimum cost flow algorithm, released in 2023. This was an improvement from the 2022 algorithm [9, 10, 8], by avoiding the randomness previously used and becoming fully deterministic. This work combined various ideas from previous works as a way to get, as the authors claim, a deterministic almost linear time solution for the maximum flow problem. Even though this algorithm solves the minimum cost flow problem, the maximum flow can be reduced to the former. To comprehend the reduction, first we must understand what the minimum cost flow problem consists of.

We are given a directed graph $G = (V, E)$ with a set of vertices V and a set of edges E, with $|E| = m$. Each edge (u, v) has its lower and upper capacities, respectively denoted as $c^-(u, v)$ and $c^+(u, v)$, for which $c^-(u, v) \leq c^+(u, v)$ and both are greater than or equal to 0. There is also a cost $cost(u, v)$ associated to each edge (u, v). For the vertices, each vertex v has a demand $d(v)$, which may be negative, zero, or positive. This problem has some constraints that must be followed. The first one being defined as $\sum_{v \in V} d(v) = 0$, meaning the graph, as a whole, must not lose or gain flow. Furthermore, a feasible flow $f : E \to \mathbb{R}$ is a flow for which, for any edge (u, v), we have that $c^-(u, v) \leq f(u, v) \leq c^+(u, v)$ and, for a vertex v, $d(v) = \sum_{(v,u) \in E} f(v, u) - \sum_{(w,v) \in E} f(w, v)$. Finally, the cost of a given edge denotes the cost of pushing one unit of flow through it. Hence, the minimum cost flow problem is finding a feasible flow that minimizes the following expression: $\sum_{(u,v) \in E} f(u, v) \cdot c(u, v)$.

With this in mind, the minimum cost flow problem consists of a more general problem than the maximum flow one. Therefore, a reduction from the latter to the former is

possible, as it will be explained. Considering a maximum flow problem in a directed graph $G = (V, E)$, with capacities c, and source and sink s and t, accordingly. We define a new graph $G' = (V, E')$, with the edge set $E' = E \cup \{(t, s)\}$. Additionally, for all edges in E, we define the lower and upper capacities as $c^-(u, v) = 0$ and $c^+(u, v) = c(u, v)$, and costs as $cost(u, v) = 0$. For the edge (t, s), we have $c^-(t, s) = 0$, $c^+(t, s) = m \cdot \max_{(u,v)\in E} c(u, v)$ and $cost(t, s) = -1$. Finally, all vertices' demands are equal to 0.

Note that an essential detail of this reduction are the edge costs. All edges have a cost equal to 0 except for (t, s), which has a negative cost. This way, the only possible approach to decrease the total cost of the flow is by pushing flow through (t, s), encouraging the algorithm to push as much units of flow as possible. We ensure the added edge between the sink and the source is not a bottleneck for the maximum flow by assigning it an upper capacity that is greater than or equal to the maximum possible amount of flow reaching the sink [5, 21].

While we have formulated the problem, and we know the algorithm is encouraged to get the maximum flow value to the sink vertex in order to then reduce the cost of the flow, it still has to be computed. The algorithm starts with an initial feasible flow that is, with high probability, not optimal. From here, it looks for cycles that are capable of reducing a given ratio, which we talk more about in Chapter 4.

We look at more aspects of this algorithm in Chapter 4. There, we discuss and focus on some parts of the algorithm. We specifically talk about the data structure used, the creation of a tree with essential characteristics and, in general, how the algorithm is able to get to an optimal flow.

2.2.5.1 Complexities

The authors of this algorithm were able to minimize the efforts to get a time complexity of $O(m^{1+o(1)})$. Note that $o(1)$ represents a function that approaches zero. As that happens, this complexity tends to $O(m)$ and, therefore it is said to achieve an almost linear running time.

2.2.6 Algorithms Discussion

The maximum flow problem was formulated exactly seventy years ago and it became a fundamental optimization challenge in graph theory. Here, we cover some of the most important and impactful algorithms.

The first algorithm to solve this problem appeared three years after its formulation and its authors were Ford and Fulkerson [14] with a method that repeatedly finds augmenting paths and sends the maximum possible flow amount through them. This step was not fully specified though and thus allowed researchers to develop variants of it. The Edmonds-Karp algorithm [13] being one of them, used BFS with the idea of always finding a shortest augmenting path. The motive behind this was that they noticed that by getting shortest

augmenting paths, the maximum number of iterations would be reduced from $O(|f|)$ to $O(nm)$, therefore making it independent from the maximum flow value.

However, not all variants were that simple. Yefim Dinitz came up with Dinic's algorithm [12] in which it first creates a subnetwork of the residual network where it only keeps unsaturated edges that seem to be making progress and get closer to the sink vertex, making certain that only shortest augmenting paths could be found. Only then it runs DFS. Although this step could theoretically be implemented using any search algorithm, DFS is exceptionally effective due to its natural backtrack ability when reaching dead ends and, by exploring depth-first, it takes advantage of the extra step of creating a layered network. An algorithm like BFS would instead try to explore every possible path, which is not useful since a layered network is created to guarantee whatever augmenting path found is one with the shortest length. This way, Dinitz was able to reduce the running time to $O(n^2m)$.

After Dinitz, in 1974 Karzanov [20] introduced and used the concept of preflows to get to a running time of $O(n^3)$ [3].

Later, Gali and Naamad revisited Dinic's algorithm and optimized it essentially by remembering paths to prevent rediscovering them multiple times [16]. They were able to reduce Dinic's running time from the original $O(n^2m)$ to $O(nm \log^2 n)$ becoming, at the time, the best algorithm for finding maximum flows on sparse graphs. Four years later, in 1983, Sleator and Tarjan proposed a data structure for dynamic trees [25] and were able to get the time complexity to $O(nm \log n)$. This was a significant improvement when compared to Gali and Naamad's work [16] since it beats it by a factor of $\log n$.

Goldberg and Tarjan revealed to the world the push-relabel algorithm in 1988 [17]. They used Karzanov's concept of preflow which allows the flow to temporarily break the conservation rule and it is gradually converted into a maximum flow. They also used a concept of distance labels that would help guide the flow toward the sink. This algorithm though does not specify every detail of it and thus the running time depends on the chosen implementation. For instance, the protocol to choose the order in which the active vertices are managed can vary. If the active vertices are chosen by the highest label we get a running time of $O(n^2\sqrt{m})$, and if done simply using a FIFO protocol the running time goes to $O(n^3)$. Additionally, they also presented a solution similar to the above FIFO protocol, except they used a dynamic tree that is capable of reducing the algorithm's time complexity to $O(nm \log \frac{n^2}{m})$.

In between the years 1988 and 2022, numerous new algorithms were developed trying to combine and get the best from the already known strategies like the preflow, layered network, dynamic trees and many more.

Finally, in 2022, Li Chen, Rasmus Kyng, Yang Liu, Richard Peng, Maximilian Gutemberg and Sushant Sachdeva combined efforts and were able to get to a new solution in almost linear time [9, 10, 8]. This was possible by using L1 interior point methods to limit the number of outer iterations to $O(m^{1+o(1)})$ and a sophisticated recursive data structure that can be efficiently managed.

One year later, Jan van den Brand , Li Chen, Rasmus Kyng, Yang Liu, Richard Peng, Maximilian Gutemberg, Sushant Sachdeva and Aaron Sidford [6, 5] used a new data structure, which led to a deterministic algorithm, on the contrary to the 2022 version. An improvement on this version was later released by Li Chen, Rasmus Kyng, Yang Liu, Simon Meierhans and Maximilian Gutemberg [7].

We can compare the running times of the algorithms referred in this work in Table 2.1.

Table 2.1: Algorithms' time complexities - n and m stand for the number of vertices and edges, respectively.

Algorithms	Variants	Time Complexity
Edmonds-Karp	-	$O(nm^2)$
Dinic	-	$O(n^2m)$
Karzanov	-	$O(n^3)$
Gali and Naamad	-	$O(nm\log^2 n)$
Sleator and Tarjan	-	$O(nm\log n)$
Push-relabel	FIFO	$O(n^3)$
	Highest Label	$O(n^2\sqrt{m})$
	w/ dynamic tree	$O(nm\log\frac{n^2}{m})$
Chen et al.	2022	$O(m^{1+o(1)})$
	2023	$O(m^{1+o(1)})$

2.3 Related Works

Previous works comparable to this one have already been written. Many algorithms and their variants have been tested in numerous different types of graphs. Each specific algorithm has its strengths and weaknesses, meaning some types of graphs may suit an algorithm better than others due to its characteristics. Therefore, a well-built comparison study might use many network generators that are capable of building different networks varying the number of vertices and edges as well as their structure, for instance, its density and number of augmenting paths. This way one can understand what kinds of networks work best for each algorithm, and thus infer why that happens and what are the main factors that cause it.

On [15], there were ten different types of graphs and 25 algorithm implementations. The network generators used build many types of graphs like semi-randomized graphs, graphs that maximize the amount of existing augmenting paths, etc. Each algorithm was tested on every type of graph and its size was exponentially increased in order to understand how its performance varies with increasing overall graph size. Furthermore, in the algorithms considered are the Edmonds-Karp [13], the Dinic's [12], Karzanov [20], Gali and Naamad [16], Sleator and Tarjan [25], and many more. Some algorithms represent variants of a base one. This helps the authors to better derive what is the empirical impact that such changes have on running time and handling different characteristics. It can also then be concluded if such a change truly achieved what was expected from it.

The authors drew many conclusions from this comparison. A very clear conclusion was that, for almost all kinds of graphs, the Dinic's algorithm performed better than Edmonds-Karp. In fact, the Dinic's achieved rather good running times, "generally beats the push-relabel algorithms without heuristics, and many of the heuristics as well" [15]. They found that the Dinic's starts losing some ground whenever it is presented with graphs in which the extra step of building a layered network is not compensated because each contains very few augmenting paths, making it unproductive to take extra time to build it.

Moreover, many variants of the push-relabel algorithms have been part of this study. What they found was that, even though the base algorithm performs already pretty well, global relabeling can still be a very good help in reducing the time depending on the graphs. However, maybe unexpectedly, the usage of dynamic trees, used to maintain a set of vertex-disjoint trees, tends to worsen the running time of the algorithms and thus, is not justified in practical terms.

Another computational study that showed interesting insights is [3]. This study made use of two different types of graphs, random layered networks and random grid networks. In random layered networks the vertices are distributed between layers of vertices and the edges are randomly added. In random grid networks, "vertices are arranged in a grid and each vertex is connected to its neighbor in the same and the next grid" [3]. Here, the authors found that in the tests made to the Dinic's, the number of layered networks built to complete the algorithm were roughly $n^{0.7}$. Moreover, as expected due to its complexity $O(n^2m)$, running time tends to rise with increasing density of the network. For push-relabel algorithms, the highest label variant was the one performing the best empirically. Additionally, this paper agrees that dynamic trees, although reducing the theoretical running time, in practice slow down the algorithm. This is explained as dynamic trees offer a more efficient push operation and in exchange, the relabel operation gets slower. According to [3], relabels tend to grow faster and thus their usage is in most cases not helpful.

Also, [4] compared maximum flow algorithms. However, it is important to note that this study was made with the purpose of energy minimization in vision and thus the algorithms were tested in accordance with its purpose. The authors chose to focus on four algorithms, three of which the Dinic's and two variants of the push-relabel, highest label and FIFO, and the fourth one being an algorithm focused on energy minimization in vision. The results shown were a product of testing in the context of "image restoration, stereo, and segmentation" [4]. These give an advantage to the vision-focused algorithm developed by the article's authors. However, if we compare the three algorithms that are general maximum flow problem algorithms, it seems like push-relabel using a FIFO queue was the one with the overall best performance. The Dinic's was the one showing the worst results.

3

Experimental Tests

3.1 Graph Implementation Discussion

To begin this chapter with, we start by remembering and introducing some terms. The number of vertices and number of edges, just like in the remaining chapters of this dissertation, are represented as n and m, respectively. Moreover, in this chapter we introduce $d(v)$, which stands for the outdegree of a vertex v, i.e, the number of edges with its origin in vertex v. Note that, due to the characteristics of the graphs used, $d(v) \leq n$, thus, $\Theta(d(v))$ is also $O(n)$.

There are different ways to represent and store a network in a computer. This is mainly done using adjacency lists, adjacency matrices and edge lists. We represent vertices as non-negative, consecutive integers, starting at 0 and finishing at $n - 1$. Adjacency lists consist of having an array in which each cell represents a vertex. Each cell stores a list with the outgoing edges from the corresponding vertex. The adjacency matrix, as the name suggests, represents a network via a squared matrix of order equal to the number of vertices in the graph. Therefore, $matrix[1][3]$ will either store the edge that goes from vertex 1 to 3, or its absence. The last out of these three is an edge list. This is the simplest way of storing a graph. By simply storing every edge in an array and the total number of vertices, we have all the information that we need about the network. Moreover, since all implemented algorithms build a flow network, each edge keeps a pointer to its reverse edge, to avoid searching for a specific edge very often.

All these possibilities represent and store everything needed about a network. However, all of them have their strengths and weaknesses. Thus, in order to later compare empirically all the algorithms in a fair way, we must first assess, for each algorithm, what is the data structure that better combines with its interests. To accomplish this, we need to identify what operations each algorithm uses and how frequently it runs them.

The main operation used by all the algorithms is getting the outgoing edges of a given vertex and, hence, it is essential that that is done efficiently. Furthermore, one less frequently used operation is searching for a specific edge in the graph. The complexities of the different structures for these operations can be seen in Table 3.1.

Table 3.1: Data structures' time complexities - n, m and $d(v)$ stand, respectively, for the number of vertices, number of edges and the outdegree of a vertex v.

Data Structure	Get Outgoing Edges from v	Find Edge	Space Complexity
Edge List	$\Theta(m)$	$O(m)$	$\Theta(m)$
Adjacency Lists	$\Theta(d(v))$	$O(d(v))$	$\Theta(n+m)$
Adjacency Matrix	$\Theta(n)$	$\Theta(1)$	$\Theta(n^2)$

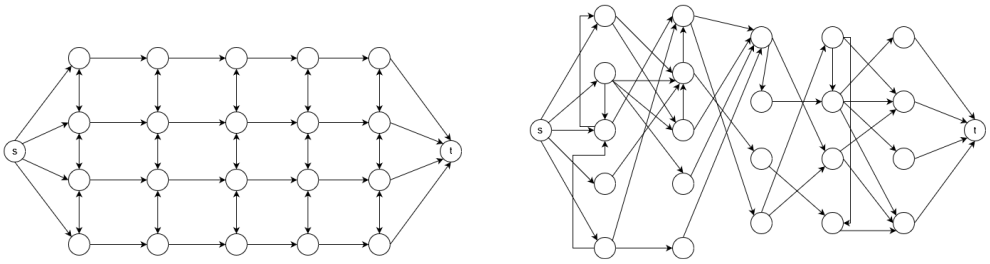
With this information in mind, edge lists are very inefficient on running time since their time complexities for both used operations are linear on the number of edges in the graph. This makes edge lists a poor choice for a maximum flow algorithm and its needs. An adjacency matrix is especially good at finding a specific edge, achieving it in constant time. Nevertheless, it is not great at finding the outgoing edges of a vertex considering it has to traverse through all the vertices to know if there is an edge between them. The last of these three options are adjacency lists, which are the most efficient when it comes to retrieving the outgoing edges of a vertex, being able to get them in $\Theta(d(v))$. Finding a specific edge is generally faster taking up to $O(d(v))$ in the worst case.

When it comes to space complexity, edge lists are the best option since they store the essential only, i.e, the set of edges only. Adjacency lists come in second since these use slightly more space in order to get more efficient searches. And, ultimately, adjacency matrices are by far the worst, allocating space to every combination of two vertices, even if there is no edge in between them. Although that provides an ideal data structure to search for specific edges, we will see that it does not fit our algorithms and the way we measure the running time.

Considering all the advantages and disadvantages explained, the data structure used for all algorithms in this dissertation is the (array of) adjacency lists as it is decent space-wise and the fastest one for the algorithms' needs. As said before, the most used operation is, undoubtedly, to retrieve the outgoing edges of a vertex and that is one of this data structure's strengths. Although this structure is not as great at finding a specific edge, which is a needed operation, that is only used for building the flow network, which is built only once every run. Since this procedure is used by all five algorithms in the exact same way, the time it takes is not accounted for the results, making this decision much easier. Furthermore, the lists, in the adjacency lists, that store the outgoing edges from a vertex are implemented as arrays.

3.2 Test Set

The tests made involve three different types of networks. The first two types are grid and random layered networks, both taken from [3] and slightly adapted. An example of these two types of networks can be visualized in Figures 3.1a and 3.1b for grid and random layered networks, respectively. The third type is real-life networks, which were all taken from [24]. Note that we use the terms number of columns for both grid and random



(a) Grid Network

(b) Random Layered Network with SEP = 0.1 and FEP = 0.3

Figure 3.1: Example of grid and random layered networks generated by their algorithms

layered networks, the number of rows is used only to talk about grid networks, and FEP and SEP only apply to random layered networks and are explained later in this chapter. The number of columns c is given by the number of columns between the source and sink vertices. For instance, the grid and random layered networks in Figure 3.1 have 5 columns. Similarly, the number of rows r is given by the number of rows that the graph has in total. r would be equal to 4 in the grid network in Figure 3.1a.

Grid networks are very structural and symmetric. The source is located behind the first column and the sink is in front of the last column, and all the columns have the same number of rows. The source vertex has edges to every vertex in the first column and all the vertices in the last column have outgoing edges to the sink. All the vertices between the source and the sink have outgoing edges to the vertex in front and to their left and right neighbors. The vertices that are in the first and last rows obviously have only one side neighbor. This kind of network is illustrated in Figure 3.1a and the number of edges can be easily computed from the number of columns c and number of rows r. Although in the implementation we actually provide number of columns and number of vertices, we can easily get the number of rows, which is defined as $r = \frac{n-2}{c}$. With the help of these parameters, we get to the total number of edges $m = 3cr + r - 2c$. This type of network was generated with random capacities in the interval $[6, 105]$.

Random layered networks can be less structured and have columns with different numbers of rows, which is why we do not use the term as a metric. As said before, the number of rows from one column to another can differ, as it is computed as follows: the first $(n-2) \bmod c$ columns have $\lfloor \frac{n-2}{c} \rfloor + 1$ rows and the remaining columns have $\lfloor \frac{n-2}{c} \rfloor$ rows. In these graphs, however, there are two attributes, FEP (forward edge probability) and SEP (side edge probability). These parameters respectively define the likelihood of, when creating a graph, adding an edge from each vertex to each of the vertices in the column in front and in the same column. Since these values represent a probability, they must be between 0 and 1. For instance, a value of 0 for SEP means that there is not going to be any edge whose origin and destination vertices are on the same column and, inversely, a value of 1 would mean every vertex is connected to all other vertices in the same column. The values for FEP work in the same way but relative to vertices in the column in front.

However, we want to assure that, for every vertex v, there is a path from the source to the sink that goes through v. Therefore, despite the value used for FEP, every vertex is the origin of an edge whose destination is a vertex from the column in front. Consequently, there is an edge from each vertex on the last column to the sink. By generating networks this way, we are able to produce more unpredictable and less structured networks to test the algorithms with distinct graphs' characteristics. We can also see an example of this type of network in Figure 3.1b, where we used values of 0.3 for FEP, and 0.1 for SEP. Similarly to grid networks the capacities in random layered networks are randomly assigned in the interval $[6, 105]$.

For the real-life networks, we used 8 different graphs. Unfortunately, these networks have very few to no information at all, therefore it is hard to know what exactly does the network represent. However, according to their categories and names, we believe seven of them represent the network of roads of certain countries or areas, and the eighth graph represents the network of some of the flights in the USA, which is the only weighted graph. Although we cannot be sure of what these weights represent, we decided to use them as capacities to make the most out of it. These capacities are in the interval $[9, 5396]$, with the mean value being around 720. For the remaining seven networks, due to the lack of information, we could either randomize the capacities for each edge or we could use unit capacities, meaning all edges have capacity equal to 1. Therefore, and to keep the network's reality property as much as possible, we used unit capacities. These networks vary from a few thousands to millions of vertices, which can be seen in Tables 3.3 or I.1 for more metrics of the networks. Furthermore, these networks do not have a predefined source and sink vertices. To address this problem we use five different pairs of source and sink vertices for each network. The first pair of source and sink is always the combination of vertex 0 and the largest vertex that there is, and the remaining four pairs were chosen randomly.

To test all the different algorithms and be able to take valuable insights from them, it is essential that we have a great variety of tests with various sizes and characteristics. To accomplish this, we produced numerous different grid as well as random layered networks. The grid networks produced vary on number of vertices, number of columns and number of edges. However, note that only number of vertices and columns are deliberately changed, as the number of edges is defined based on the former attributes, as seen before. Number of vertices go from 10 002 to 100 002. The number of edges, being heavily dependent on the number of vertices, go from around 30 000 to 300 000 and, lastly, the number of columns take the values 20, 50, 80, 100 and 125. We start by using 10 002 vertices and increase this number by 10 000 until it reaches 100 002. Meanwhile, we vary the number of columns for each different number of vertices. Since we use ten different values for number of vertices and five values for number of columns, we end up with a total of fifty different graphs. All the metrics from the generated grid networks can be seen in Tables I.4 and I.5.

Since the random layered networks have two additional parameters, being those the

FEP and SEP, there are a lot more combinations to test and thus the number of networks generated is larger. In this type of network there are four parameters that need to be tested for different values: the number of vertices, number of columns, FEP and SEP. The values chosen for number of vertices are between 2 000 and 10 000, increasing by 2 000. The values used for number of columns are 20, 50 and 80 and both FEP and SEP take values 0, 0.05, and 0.1. Since we have five values for number of vertices, three for number of columns and three values for each forward and side edge probabilities, we end up with 135 different networks. The sizes of the graphs produced are significantly smaller, compared to the grid networks, because the number of edges can reach very large values, starting at roughly 2 700 and going up to almost one million edges, taking the algorithms more work and time to work it out. Just like the other types of networks, all random layered networks are in Tables I.10, I.11 and I.12 in Annex I.

Although we think it would be quite valuable to test for larger graphs and in general, for a larger range of values, this was not possible since that would require a lot more time dedicated to only running these algorithms. However, we believe the tests made are acceptable and produced interesting and insightful results.

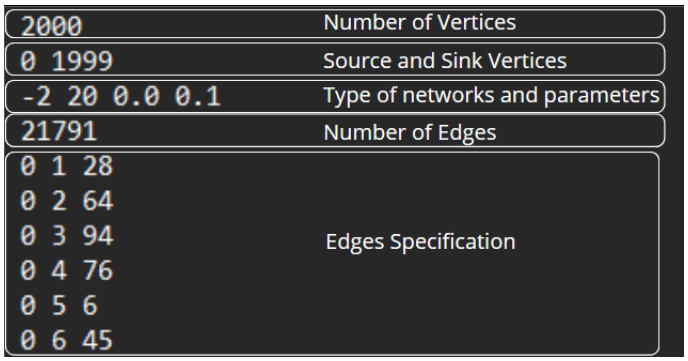


Figure 3.2: Input file example for a random layered network

In order to generate a network or to get it from an outer source, we must first specify the input file that the program requires to process the graph correctly. For all types of networks, we specify all the common values that we find important. These are the number of vertices, number of edges and source and sink vertices, followed by all the existing edges. These can be described as <origin vertex> <destin vertex> <edge capacity> (see edges specification in Figure 3.2). However, because each type of network has its own parameters, we also identify what is the type of network represented and the additional parameters required. We code the type of network with a number, with -2 representing a random layered network, a grid network being coded with -1, and a real-life network is identified with a 0. In grid networks and random layered networks, we additionally indicate the number of columns. Moreover, random layered networks also specify the FEP and SEP values. Real-life networks show only the parameters common to every type of network, which are the number of vertices and edges, the source and sink vertices, and the type of network it represents. An example of an input file of a random layered network

can be seen in Figure 3.2. The input files between different types of networks only vary on the third line, which indicates the type of network, as well as its own specific parameters. Since the example given represents a random layered network, the parameters beyond the type of network are the number of columns, FEP and SEP, in this order.

3.3 Results and Discussion

We start the results section by detailing the hardware and software specifications used to obtain the results discussed in this section. On the hardware side, the computer used is equipped with an Intel Core i7-8565U with 4 cores, 8 threads, and a base clock speed of 1.80GHz. It has a RAM of 8GB with a speed of 2400MHz. Moreover, it has a 256GB SSD. In the software side, it has Windows 11 installed and all tests were ran in IntelliJ IDEA 2023.2.2. with JDK 17.0.7. Additionally, the configuration "-Xmx4098m" was added in order to increase the maximum heap size of JVM to 4098MB. This was done to avoid running out of memory when performing the tests.

Note that the algorithms were ran three times on each network and the results used to build charts were the average of the three samples. This was done as a way to smooth out some fluctuations that cannot be controlled and to get more reliable results with less noise. We measure the variance between each of the three runs that each algorithm completes on every network with the coefficient of variation, which is essentially a relative standard deviation. The values of the coefficient of variation over all algorithms and networks were, on average, 8.6%.

When analyzing the results produced, there will be used some terms that are important to understand. The simpler terms are number of vertices, number of edges and the running times of each of the five implemented algorithms, which are always represented in milliseconds. We introduce a new parameter used to analyze results, called relative performance degradation, which indicates how much worse an algorithm performed, in percentage, when compared to the algorithm that performed best. For instance, if a given algorithm has a 20% RPD (relative performance degradation), it means that its running time was 20% slower than the best running time out of the five algorithms. To avoid repeating terms that are used very often, as in number of vertices, number of edges, number of columns, forward edge probability, side edge probability and relative performance degradation, we may use n, m, c, FEP, SEP, and RPD, respectively. Lastly, before looking at the actual results, we note that each algorithm is always coded by the same color. The color coding is done in a logical way, grouping similar algorithms with similar colors. Therefore, we represent the EK as orange, the DinicRec and DinicIt as dark and light blue, respectively, and the push-relabel algorithms are coded with purple and pink for the push-relabel with FIFO (PRFIFO) and the push-relabel with highest label protocol (PRHL), accordingly. Lastly, to easily identify each network from all the others, we have a parameter "NetId" which is the network identifier. It identifies each network by starting with a letter which codes its type: "R" for real life networks, "G" for grid, and

"L" for random layered networks. This is followed by a unique number for each type of network, which is ordered by increasing number of edges. So, for instance, G6 denotes the sixth grid network in increasing order of number of edges and is completely unrelated to R6 or L6.

We first look at overall results, splitting these in types of networks: grid networks, random layered networks and real-life networks. The goal here is to assess how the algorithms performed in relation to one another in each type of network, and to study how the parameters impacted each algorithm. With this done for all three types of networks, we explore each algorithm more thoroughly and try to suggest why it behaves and gets affected the way it does. We put extra effort into highlighting and, if possible, explaining any unexpected outcomes that we might spot.

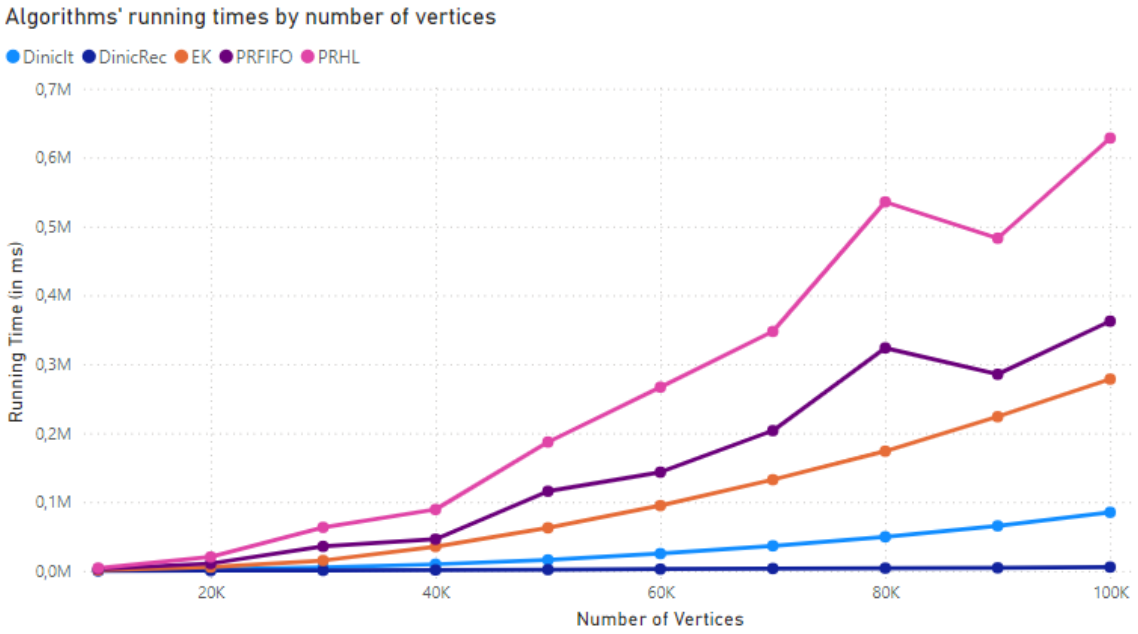


Figure 3.3: Algorithms' running times by number of vertices for grid networks

We start with the results from the grid networks, for which the running times are all displayed in Tables I.6 and I.7, and other interesting metrics in Tables I.8 and I.9. Now, looking at Figure 3.3, there is a graphic that shows the running times by the number of vertices, which, in this case, is a good way of assessing a graph's size and complexity, as m is practically directly proportional to n and thus we avoid showing a similar chart for number of edges as it would display almost the exact same trends.

We can clearly see that the two variants from the Dinic's algorithm are the ones performing best, independent of graph size. The DinicRec performs especially well, having the fastest running time for all numbers of vertices. This difference in performance, particularly when compared to the iterative version, is likely explained by the intelligence added of remembering previous paths and, therefore, avoid repeating bad decisions. Furthermore, both push-relabel algorithms performed surprisingly bad when considering their time complexity, being the ones performing the worst by a relatively big margin.

Moreover, both display a very similar dip in networks with 90 000 vertices. Although we were not able to understand why they performed substantially better when compared to the remaining networks, we still consider it a very intriguing result. The PRHL was especially slow as we can clearly notice in the chart, running consistently much slower than all other algorithms. Lastly, the EK performed in between the Dinic's and the push-relabel algorithms. This disadvantage toward the Dinic's makes sense when looking at their time complexities (see Table 2.1) and taking into consideration that we are only testing with graphs that have much more edges than vertices.

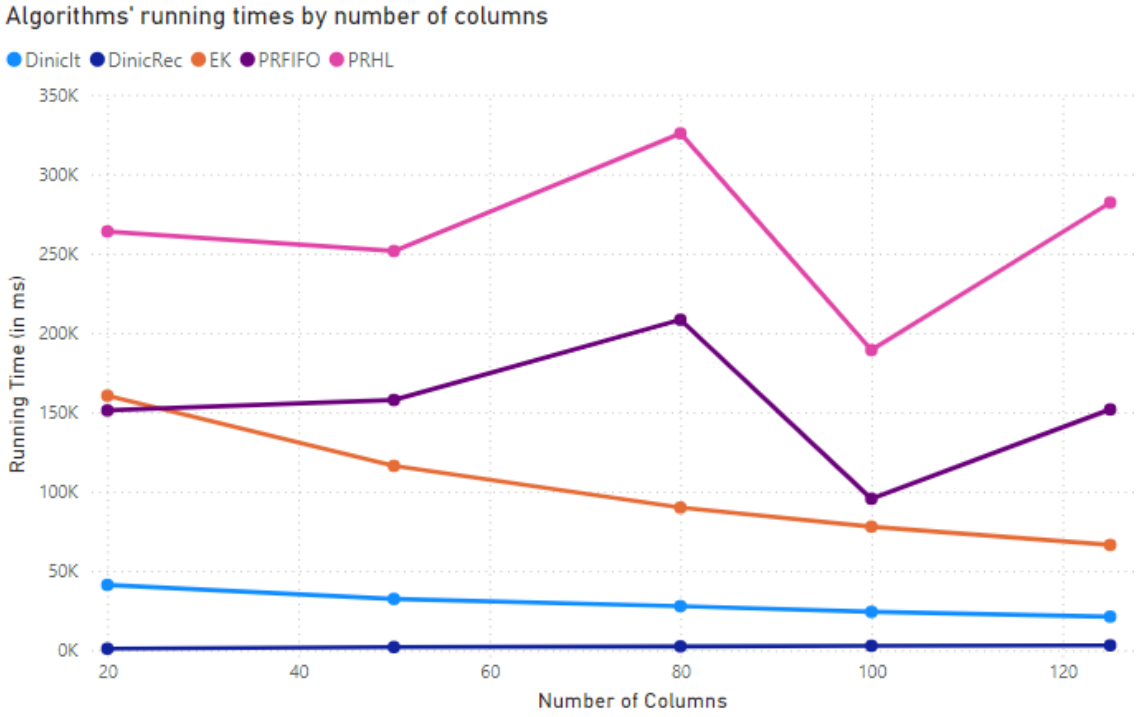


Figure 3.4: Algorithms' running times by number of columns for grid networks

Now, looking at Figure 3.4, we can observe how the number of columns affects the running time of each algorithm. It is important to note that, for each value of number of columns used, the number of vertices remains exactly the same and the number of edges stays within a small interval as well, thus not significantly affecting our results. With both push-relabel algorithms displaying a very similar and unpredictable curve, it is hard to say whether the number of columns has a big impact on their performance. However, the remaining three algorithms show clarifying results. The EK undoubtedly shows that it performs better with bigger numbers of columns. In the same way, the DinicIt gets much better results with increasing numbers of columns. Although this is hard to notice, due to the big range of values of the graph, the DinicIt displays a running time of around 41 000 milliseconds in networks with 20 columns and goes all the way down to, roughly, 21 000 milliseconds in networks with 125 columns. Lastly, and opposite to others, the DinicRec shows a worse time performance for larger numbers of columns. Going from around 1 000 to 3 000 milliseconds on the defined range. Due to its distinct effect on each algorithm, the

number of columns seems to be an interesting parameter that benefits most algorithms but hindrances others, depending on their characteristics.

We now know the evident preferences of the DinicIt, DinicRec and the EK as they all got very clear results. According to the way these algorithms work, we would like to suggest a reason for these three algorithms' preferences, since their time complexities are not able to explain it. Starting with the EK, which displayed a decrease in running time of almost 60%, we believe the results are explained by the way it searches for paths, with a BFS. In a wider grid network (fewer columns), the BFS will explore all the vertices and most of them will not be part of the shortest augmenting path chosen. Therefore, they will mandatorily be explored in at least one more iteration, thus, we could say the first time they were explored did not help much. In a narrower and longer network (more columns) though, the number of vertices that is explored and will be used in the augmenting path is larger since the average shortest augmenting path will also be larger. This makes it more efficient although they both have pretty much the same value of n and m. The fact that the DinicIt and DinicRec have opposite preferences is very interesting since we are talking about very similar algorithms. However, as we have said in Section 2.2.3, their main difference other than one having been implemented recursively and the other iteratively, is the fact that the DinicRec remembers the paths that it has explored previously to prevent repeating them. Therefore, we believe the difference in their preference must have something to do with this. This piece of added intelligence gives an advantage to the DinicRec, especially in networks where there are a lot of branching paths. While the DinicIt would waste a lot of time revisiting paths, the DinicRec remembers which paths it has explored and thus it skips them. Hence, the better performance in the wider networks. Contrarily, the DinicIt prefers the narrower graphs, since it will waste less time revisiting already explored paths. Although this might not be the only cause, we believe it is a strong one that makes a difference.

Looking at Figure 3.5, we are able to see how each of the displayed attributes change and their correlations over all grid networks ordered by increasing number of edges. First we see how the graph metrics relate to one another. Intuitively, the number of columns and average augmenting path length change in a very similar manner, which is inversely proportional to the number of augmenting paths found, which makes sense. We now spot that the DinicIt and, especially, the EK, heavily depend on the number of augmenting paths found, showing the same trends. The DinicRec, in contrast, has an opposite tendency and seems to follow the trends of the average augmenting path length much better. Note that we plotted the number of edges as way to show that there are metrics that can have more influence than the number of edges and the overall size of the network. These results corroborate what we suggested before, for Figure 3.4.

We now start looking and analyzing results from the random layered networks. These results are all available in Tables I.13, I.14 and I.15 for the algorithms' running times and in Tables I.16, I.17, I.18 for other metrics. In Figure 3.6 we show the way running times change when dealing with different numbers of vertices. When comparing with Figure 3.3,

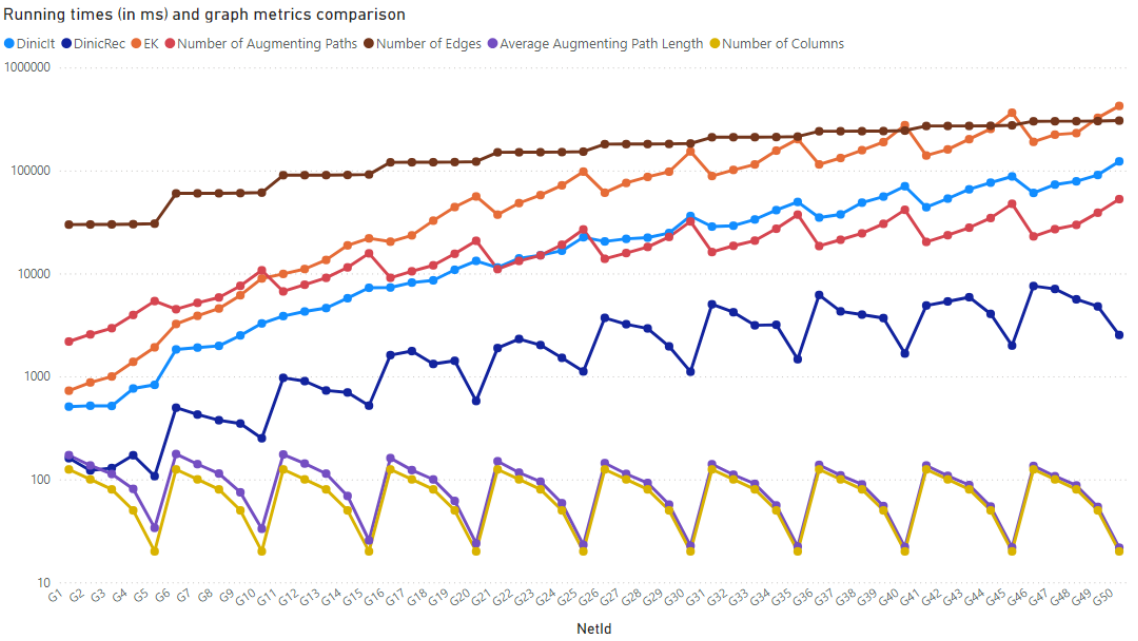


Figure 3.5: EK, DinicRec and DinicIt running times and grid networks' metrics

the running times increase much faster, percentage wise. However this is likely due to the way the random layered networks are generated. In this type of network, due to the use of parameters like FEP and SEP, the number of edges, on average, is more of a fixed percentage of number of vertices, making m rise much faster. When comparing the

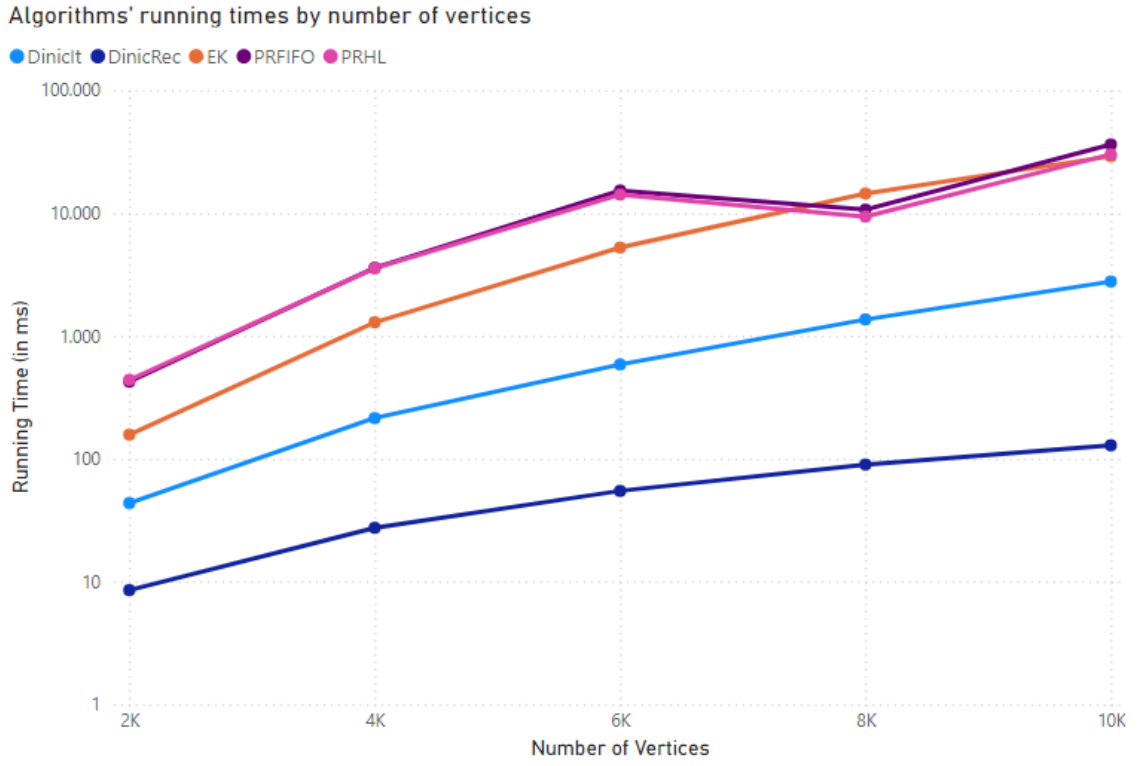


Figure 3.6: Algorithms' running times by number of vertices for random layered networks

algorithms with one another, we see that their hierarchy is pretty much the same, with the only spotted difference not being statistically significant.

Unlike on grid networks, random layered networks vary their number of edges based on a few other factors other than the number of vertices and number of columns. As both FEP and SEP are the attributes that define the number of edges relative to the number of rows and columns, as we have seen before, these are capable of having a very strong influence in the number of edges a network holds. Therefore, in Figure 3.7, we see how the running times are affected by different numbers of edges. The number of edges that each network has is distinct from all others due to FEP and SEP and the fact that these use randomness when applied. Thus, looking at a graphic with so many different values of m, though many of them are almost equal, would make analyzing and taking conclusions harder since the running times would be very volatile. Therefore, we grouped them in ten groups, going from the 10% of networks with least edges, to the 10% of networks with most edges, and used the average of the running times on each group of networks to analyze the algorithms' performance.

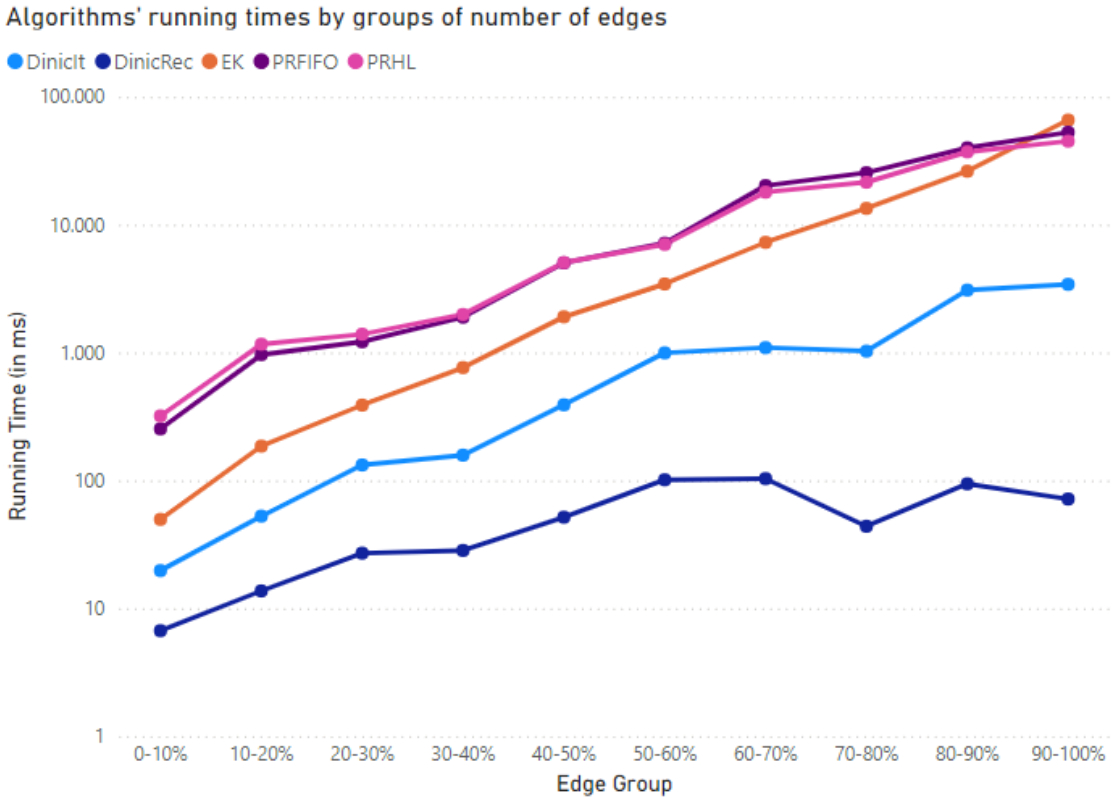


Figure 3.7: Algorithms' running times by number of edges for random layered networks

Just like expected and explained by the algorithms' time complexity, in general, the running times, in Figure 3.7 get progressively slower with the increase in edges. However, there are a few unexpected results that are worth taking time to understand. Whilst the PRFIFO, PRHL and the EK show a very consistent upwards trend, both Dinic's algorithms have a relatively similar dip for the networks in 70-80%. Mind that, the 60-70% networks

average a number of edges equal to approximately 118 000 and the 70-80% have an average of almost 175 000. We controlled the average values of FEP and SEP to see if they could be the reason of this time reduction. In that dip, the average of n is roughly the same and thus, what caused the number of edges to increase was a slightly larger value of FEP, around 0.01 increment, which, as we will analyze later in Figure 3.9, may be one of the preferences of these algorithms. Lastly, the DinicRec also has another dip when comparing the 80-90% networks with the 90-100%. Once again, upon a deeper examination, we see that while the value of SEP remains the same, FEP sees an increase of 0.01, which might be significant enough to explain this change in the running time. In this specific case, the number of edges increase from 287 000 to 608 000, showing just how much of an impact the FEP and SEP can have.

Figure 3.8 shows the average running times of the algorithms for each of the values chosen for number of columns in random layered networks, which are 20, 50 and 80. However, note that in random layered networks, while the number of vertices is not altered by the number of columns, the number of edges is (see in Figure 3.11). From the networks used for testing, a network with 20 columns has, on average, roughly 216 000 edges, and around 56 000 for networks with 80 columns. This is important to keep in mind since the number of columns is not the only parameter that is being modified and that also has an impact on the results.

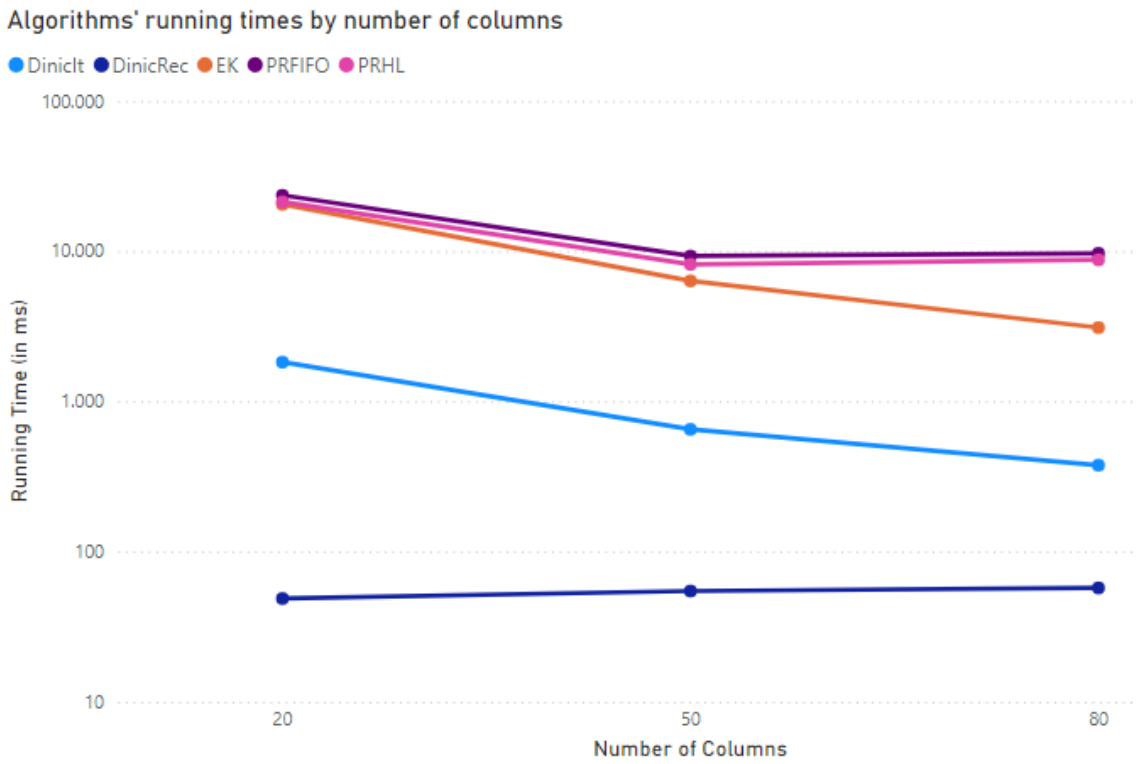


Figure 3.8: Algorithms' running times by number of columns for random layered networks

Now that we know this, we can analyze the results shown in Figure 3.8. Even though the number of edges decreases with the increase in number of columns, interestingly,

that does not dramatically change what we analyzed in Figure 3.4 as both push-relabel algorithms display an inconclusive outcome by reducing their running times significantly from 20 to 50 columns, but getting worse results again when changing to 80 columns. Just like before, the EK and the DinicIt, show that more columns, and in this case, fewer edges too, have a very positive impact on their performance, consistently decreasing their running time. The DinicRec, although the number of edges decreases, still gets worse results for larger numbers of columns, taking respectively 48 and 57 milliseconds to solve networks with 20 and 80 columns, on average. So, when comparing this with the same chart from grid networks (Figure 3.4), in which the number of edges remains always the same, we see that, although the values are slightly different, the tendencies are fairly the same. The difference seen between the two graphics is likely due to the variation in number of edges and, therefore, the charts from random layered and grid networks are consistent with each other.

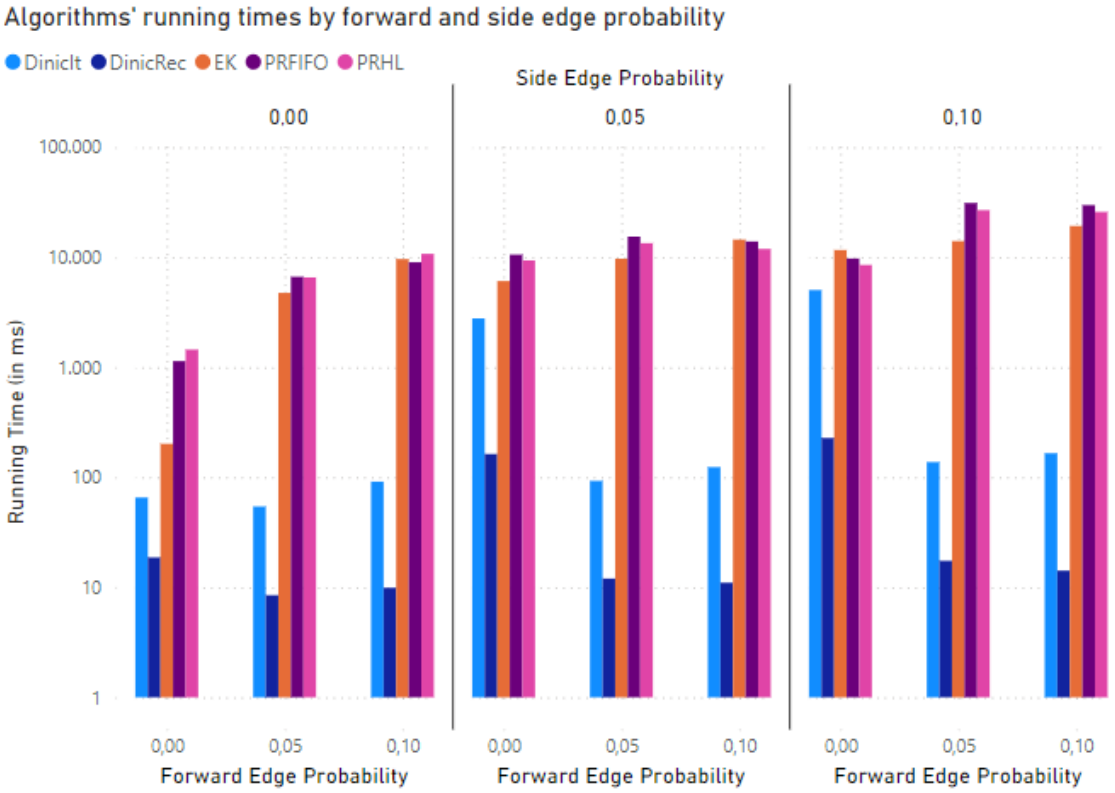


Figure 3.9: Column chart over algorithms' running times by FEP and SEP - The number of edges for each combination of FEP and SEP values can be seen in Table 3.2.

Figure 3.9 helps us understand how the parameters FEP and SEP affect running times. I.e, how the number of edges from one column to the next one (edges that make progress toward the sink), and edges in which the origin and destination vertices are in the same column, influence the algorithms tested. Note that the numbers of vertices do not change with different values of FEP and SEP. Recalling these parameters, we know that, the larger the values used, the more edges the graph has (see in Table 3.2). With this in mind, we can

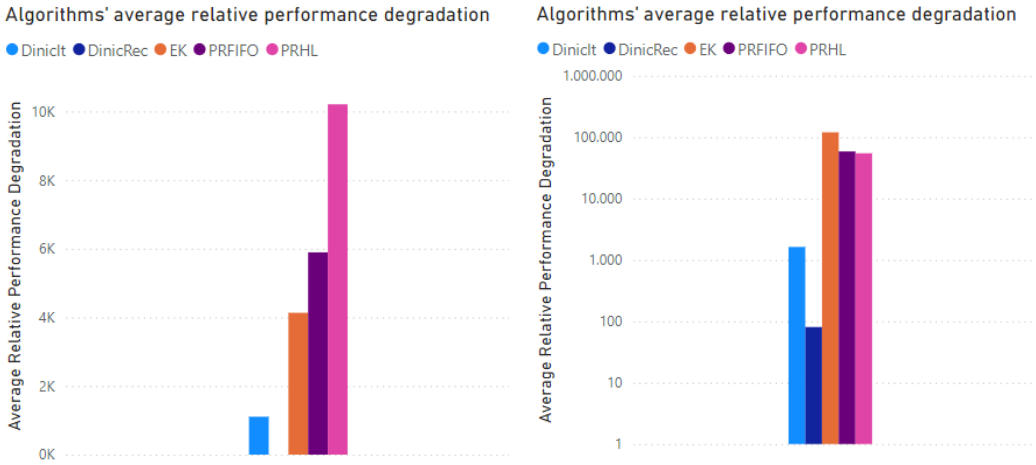
Table 3.2: Average number of edges for each combination of FEP and SEP values

FEP \SEP	0.00	0.05	0.1
0.00	8 293	66 356	126 545
0.05	58 827	118 925	178 843
0.1	116 626	176 883	237 082

see that they have a strong impact on some specific algorithms' performance, producing results that would be, according to the algorithms' time complexity, very strange. EK seems to be the less affected by the tested parameters. We extract this by noticing that its running time varies with the value of m and not so much with FEP and SEP. Here, again, both push-relabel variants perform very similarly, showing a preference for greater values of FEP and smaller values of SEP. This conclusion can be taken because, as it can be seen in Table 3.2, in some combinations of values of FEP and SEP the number of edges is approximately the same. And, if we compare these combinations as a way to try to isolate the parameters in study, we spot that the push-relabel algorithms always get worse running times on the configurations with the larger values of SEP and smaller values of FEP. Analyzing now the Dinic's algorithms, and starting with the iterative version, it seems to have an obvious preference for a FEP different than 0, as for all three values of SEP, performance gets significantly better once it gets to 0.05, despite the rise in number of edges. Then, once it goes to 0.1, it loses some performance, but that might be explained due to the increase in number of edges. The DinicRec shares a lot of preferences with the DinicIt except, when changing FEP from 0.05 to 0.1, the former either gets approximately the same running time or, in most cases, reduces it, which means that the increase in FEP makes up for the increase in number of edges that, by itself, is not helpful.

Although a graphic just like the one depicted on Figure 3.5 for random layered networks could be interesting, this is not possible due to having too many networks. However, we use the Power BI tool to help finding the key influencers of each algorithm. What this analysis tells us is that both Dinic variants are very hard to predict and that there are no significant relationships between the time taken and the graph metrics. They often have very sudden changes in running time for no apparent reason. For the EK, the key influencers indicated are the number of edges and the number of augmenting paths, both proportional to the running time. Just like in grid networks, it is clear that the time taken by the EK is very influenced by the number of augmenting paths.

Now, looking at Figure 3.10, we see how each algorithm compares to the best running time on both grid and random layered networks. In Figure 3.10a, the first thing that stands out is the fact that the DinicRec has an average RPD equal to 0%, meaning it was the fastest running algorithm in all grid networks tested. DinicIt comes second with an average of around 1 100%, indicating its running time is, on average, twelve times slower than the DinicRec. The third fastest algorithm is the EK with a value of around 4 000%, then the PRFIFO with roughly 6 000% and, sitting last, the PRHL, being on average, more than 100 times slower than the DinicRec.



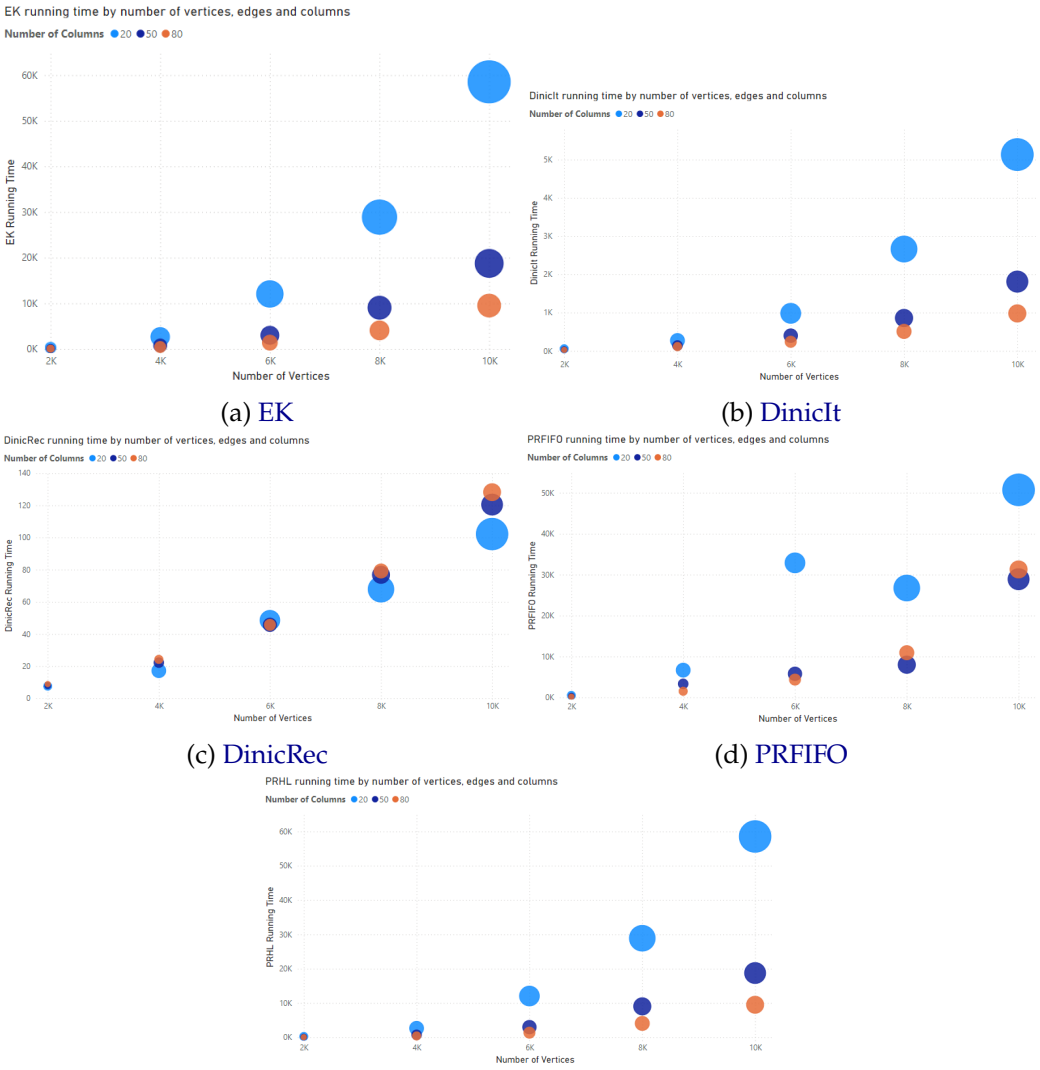
(a) Grid Networks (b) Random Layered Networks

Figure 3.10: Average RPD on grid and random layered networks

We see that, for random layered networks, in Figure 3.10b, the hierarchy and values change significantly. Here, the PRFIFO was able to get the fastest running time in 36 out of the 135 tests made, with DinicRec getting the fastest time on the remaining tests. The DinicRec was then, on average, 80% slower than the fastest one, and the DinicIt came second again with a RPD of approximately 1 600%. Both push-relabel algorithms performed very similarly with the PRFIFO having roughly 58 000% and the PRHL with around 55 000%. Upon a more detailed look, we see that, although the PRFIFO is the fastest in around 27% of the random layered networks, it then performed very poorly on others, making its average RPD rise to values larger than the PRHL's. The EK performed extremely slow, showing a value of almost 120 000% and a much weaker and slower performance than what it showed for grid networks.

In general, the average values of the RPD on random layered networks were much greater. This could be explained due to the higher variance that this type of network has, since it can reach very large numbers of edges and it is overall less structured than grid networks.

Now that we have compared the algorithms' running times on artificial networks, we focus on each algorithm, individually. In Figure 3.11 we have five scatter charts that code the number of edges as the size of each point and the number of columns with different colors. There, we see how each algorithm is influenced by number of vertices, edges and columns, simultaneously. For instance, if we look at Figure 3.11a we see that, in general, EK has an evident preference for larger numbers of columns and smaller numbers of edges, displaying very consistent results over all values. This preference makes sense when looking at the time complexities table (see Table 2.1), and is clearly shared with the DinicIt and the PRHL. However, the DinicRec and the PRFIFO display data that is not so clear. Starting with the PRFIFO, although it looks like it also has a tendency to achieve better results for larger numbers of columns and smaller numbers of edges, it gets rather



(a) EK (b) DinicIt

(c) DinicRec (d) PRFIFO

(e) PRHL

Figure 3.11: Algorithms' running times by number of vertices for random layered networks - The number of edges is coded in the circles' sizes and the color codes the number of columns.

unclear for networks with 8 000 and 10 000 vertices, where this tendency is inverted for values of 50 and 80 columns. Finally, the DinicRec, although more subtle, seems like it has a positive bias toward smaller numbers of columns and, oddly enough, larger numbers of edges. This does not necessarily mean that the algorithm performs better for networks with more edges. It could, and it probably is the case, that the structure of the graph fits the algorithm better and that makes up for the increase in edges.

We now look at the real-life networks which are composed of a total of eight different networks, which, just like for the other types of networks, all results are displayed in Annex I, in Tables I.2 and I.3 for running times and other kinds of metrics, respectively. However, as we have explained before, for each real-life network, we use five distinct combinations of source and sink vertices. Thus, we end up with forty network instances.

We would like to add that, out of the forty instances, in three of them we could only run the PRFIFO and the PRHL once instead of the three times that were ran on all the other tests. This was due to the extremely long time it took those algorithms to find a maximum flow, taking more than one day for each run to complete. However, we believe we can still work with these values as the coefficient of variation, between three runs, for the push-relabel algorithms on real-life networks is, on average, 15%. We consider this value is good enough that we can accept the running time of a single run as a likely good approximation of its true value.

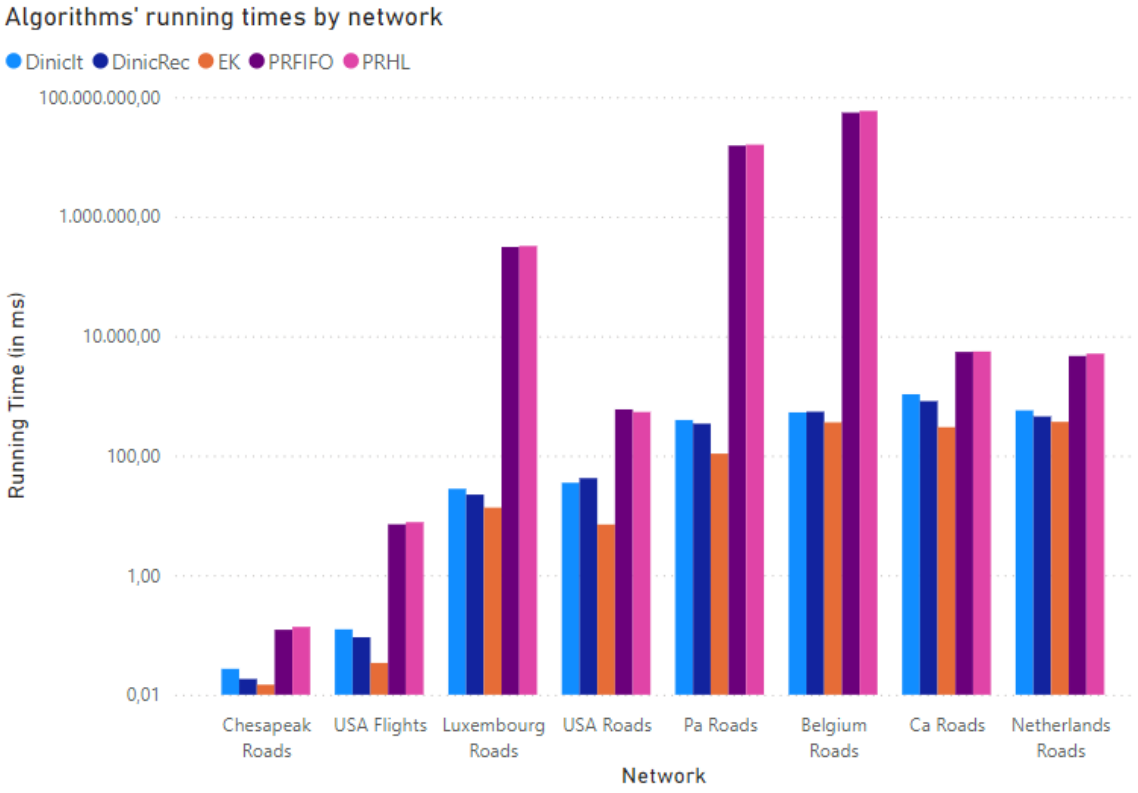


Figure 3.12: Algorithms' running times by real-life networks

Starting with Figure 3.12, which displays the running times that each algorithm took to find a maximum flow on each real-life network. Note that the networks, on the X-axis, are ordered by increasing number of vertices, as it can be observed in Table 3.3. As we can see, the EK and both Dinic's algorithms are fairly consistent with the network size. And, even though the Dinic's algorithms improve their performance from Ca Roads to the Netherlands Roads, it is important to note that there is actually a slight decrease in the number of edges. Generally they all get better times for smaller networks and worse times for larger ones. And, if we ignore the Luxembourg, Pa and Belgium roads we can say the same thing about the remaining two algorithms, the PRFIFO and the PRHL. However, the results shown are very intriguing, since these two algorithms display extremely poor running times for five specific network instances that do not have any particularity captured in number of vertices or edges. To better understand where this is

Table 3.3: Number of vertices and edges for each real-life network

	Number of vertices	Number of Edges
Chesapeak Roads	39	340
USA Flights	332	4 252
Luxembourg Roads	114 599	239 332
USA Roads	126 146	323 900
Pa Roads	1 087 562	3 083 028
Belgium Roads	1 441 295	3 099 940
Ca Roads	1 957 027	5 520 776
Netherlands Roads	2 216 688	4 882 476

coming from, we have to look at more specific running times. We see in Tables I.2 that, on both Luxembourg, Belgium and Pa roads, we have specific source and sink combinations that skyrocket the running times by a huge amount. This happens with two instances of the Luxembourg networks (R11 and R13), once in the Pa roads (R21) and twice on the Belgium roads (R27 and R28), which explains the larger average running time on those exact three networks. We are talking about an increase in time from a few thousand milliseconds to, at most, more than one million. As this only happens in some source and sink combinations, it is likely that it depends on their values, since the whole algorithm can also change. However, as these networks are extremely large, it is very hard to assess why this happens in some instances and not in others, which could give us good insights about the involved algorithms.

Something to consider is also how many times each algorithm was the fastest to solve the problem, which we refer to as best run wins. While we had DinicRec getting the best run win for every grid network and the random layered networks being won by only the DinicRec and the PRFIFO, the real-life networks offer a lot more variety, as can be seen in Figure 3.13. All of the five algorithms except the DinicIt got at least two best run wins, showing that we probably have a decent diversity of network characteristics. What stands out the most is the fact that the EK dominated this type of network, getting 32 best run wins out of the 40 possible. The PRFIFO is in second place, getting 4 best run wins and the PRHL and DinicRec both have 2 wins.

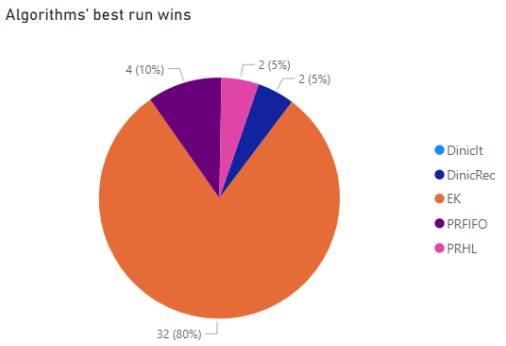


Figure 3.13: Best run wins of each algorithm in real-life networks

Looking at Figure 3.14 we compare each algorithm's RPD, on average. This is interesting to see because it shows that the number of best run wins or the RPD alone cannot tell the whole story. Both these statistics complement each other to give us more information. As expected, the EK, which had 80% of the best run wins, is clearly the algorithm with the least RPD, with around 25%. Then, very close, in second and third come the DinicRec and DinicIt with values of approximately 600% and 700%, respectively. The push-relabel algorithms, in terms of RPD, performed the worst, with both variants being around 30 000 times slower than the correspondent fastest algorithm, on average. However, note that the five network instances where the push-relabel algorithms take really long, inflate their values by a lot. If we do not consider those instances, we have the PRFIFO and PRHL with a RPD of around 9 000, meaning they are both roughly 90 times slower. Therefore, we can conclude that those five instances alone represent that huge increase in their RPD. Additionally, we see that, the push-relabel algorithms, which have six best run wins in total, perform way worse than the Dinic's algorithms, that have less best run wins. Furthermore, we would like to highlight the fact that even though the DinicIt is not the fastest algorithm in any of the networks, it still shows decent results. What this tells us is that both push-relabel algorithms are much more inconsistent and volatile, performing very well in some networks and terribly slow in others. While the remaining algorithms seem to be much more consistent, very rarely performing significantly worse than their usual.

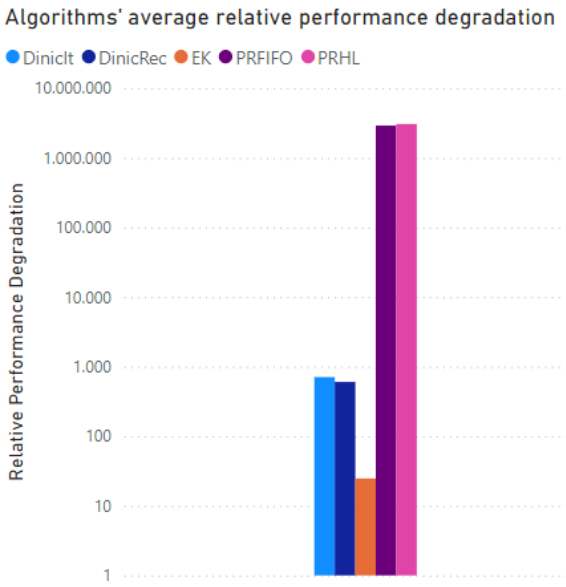


Figure 3.14: Average RPD on real-life networks

Since the results from the real-life networks are quite different than the ones we got from our artificial generated networks, we would like to try to understand and suggest why this occurred. Since the EK performed so well, an algorithm that is very impacted by the numbers of augmenting paths, we try to look into that. Therefore, we notice in Table I.3 that the number of augmenting paths in all those networks is, at most, 11, with most being

between 1 and 2. Regarding our previous analysis, this obviously works in favor of the EK. To ensure this is one of the factors, we compute, for the EK, DinicIt and DinicRec, the average number of augmenting paths of the networks for which each algorithm had a RPD of, at most, 20%. The results are consistent with what we suggested, since the EK has an average of 1.72, 2.5 for the DinicIt and 2.33 for the DinicRec. We clearly show that the EK takes advantage of the small numbers of augmenting paths in the real-life networks. Additionally, the real life networks tend to be less structured and more complex, which results in augmenting paths with different lengths from one another. While this metric may not mean much to the EK, to both Dinic variants, it means additional iterations of labeling the vertices, which requires an extra BFS per iteration.

Over the artificial networks, which are more structured and predictable, the DinicRec is the clear winner out of the five implemented algorithms, having the fastest running time in all grid networks and in 99 out of the 135 random layered networks. Surely, remembering previously explored paths and avoid repeating them has helped it achieve great times. However, over the real-life networks the winner is EK, getting the best run win on 80% of the networks.

So, closing the results, both Dinic's algorithms were very good on artificial conditions as the extra step of labeling the vertices usually proves to be beneficial. However, this may not happen in real-life cases with more unpredictable and complex networks, which explains why these algorithms did not perform so well in the real-life scenarios. The EK showed decent results, although very far from the Dinic's in the grid networks. Over the random layered networks, it performed quite poorly, displaying the worst RPD of all algorithms. Nevertheless, it performed extremely good in the real-life networks, taking advantage of variable conditions. Finally, both push-relabel algorithms were slower than expected, with highly inconsistent results. This led us to trying to understand what the issue is, which we discuss in detail in Section 3.4. However, they seemed to be able to achieve some interesting results on specific networks. Especially PRFIFO, that got 36 best run wins in the random layered networks and four in the real-life ones. Although PRHL usually showed similar running time trends and curves as the PRFIFO, most of the times it performed slightly worse.

Other previous articles have done similar jobs as the one done in this dissertation, the job of comparing empirically some of the most known maximum flow algorithms to put their theoretical time complexity to the test. However, as expected, not all works get the exact same results. Therefore, we now discuss some of the most significant differences and similarities between the results found in this and the works done by other authors.

Starting off with a master thesis from 2014 [15], they found that the empirical time complexity of the EK is, instead of the known $O(nm^2)$, best described as a function of the number of edges and the number of augmenting paths. This is consistent with the result we got that reveals that the EK gets better results with narrower and longer networks as they have less augmenting paths to find. Additionally, between the Dinic's, the EK, and the push-relabel algorithms, their results showed that the Dinic's was the one performing

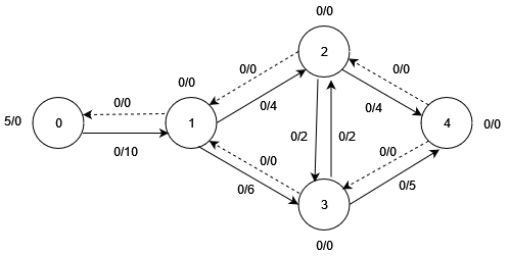
best in most cases and the push relabel being the worst. However, this work also tested the push-relabel using a global relabeling heuristic which would, from time to time, update the label of all vertices by running two BFS, the first one from the sink and the second one from the source. The vertices found in the first BFS are vertices that can still push flow toward the sink and their heights are updated to their distance to the sink. The second BFS run is made from the source and only considers vertices that were not visited in the first one. The heights of the vertices found are updated to their distance to the source plus n. The introduction of this global relabeling heuristic showed substantially better results, supporting what we believe the issue with the push-relabel is (see Section 3.4). Thus, the results on this work are similar to the ones that were produced in this dissertation and, the big improvement with the introduction of a global relabeling heuristic enhances what we explain and suggest in Section 3.4.

In another article, Ahuja et al. [3], after testing ten algorithms, found that the PRHL was the fastest out of the ten. It is important to note that the Dinic's algorithm was part of these ten algorithms but the EK was not. The tests were ran on networks that are very similar to the grid and random layered networks that we used. This is a major difference when compared to our work since our results led us to much different conclusions. In our results, even if we consider the DinicIt, as it does not have the added intelligence to it, it still performed much better than the PRHL did, contrarily to this work from Ahuja et al.

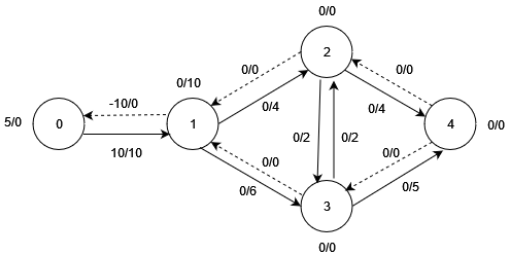
Yuri Boykov and Vladimir Kolmogorov, in [4], also tested and compared the PRFIFO, PRHL and the Dinic's algorithm in the context of energy minimization in vision. They found that the Dinic's algorithm was clearly the weakest algorithm among these three. While the PRFIFO easily outperformed the Dinic's on every case tested, although the PRHL managed to do it in the great majority of cases, it still got slightly worse results in a few cases. When comparing both push-relabel algorithms, the winner is the PRFIFO, which showed great consistency by getting a lower running time in most scenarios, even when compared to the PRHL, and the approximate same running time on the remaining cases. Comparing our results with this work's, we can see some similarities but also some significant differences. Our results agree that between the two tested push-relabel algorithms the PRFIFO was, in general, the best one. However, the fact that both push-relabel variants undoubtedly topped the Dinic's could not be further from our conclusion. As we have concluded, any of the two Dinic's algorithms implemented by us are easily a better option over the two push-relabels. This is obvious when we compare their consistencies and RPD.

3.4 Push-relabel Efficiency Issue

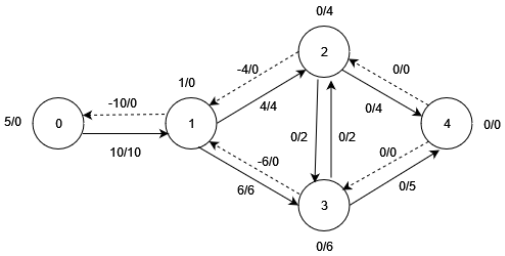
Both variants implemented from the push-relabel algorithm performed much poorer than what we expected. Papers like the one from Ahuja et al. [3, 4], that made similar comparisons concluded that the push-relabel algorithm was the one performing the best. Therefore, we explored why this happened and found a scenario in which this algorithm



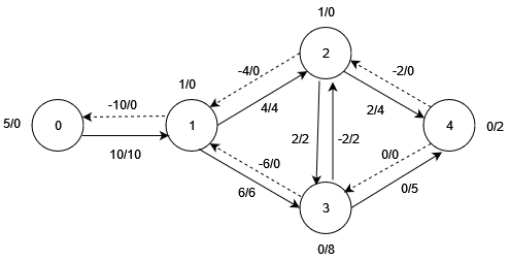
(a) Initial flow



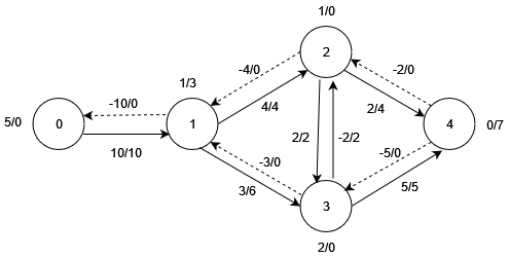
(b) First step



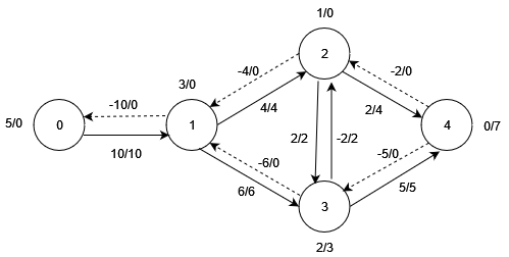
(c) Discharging vertex 1



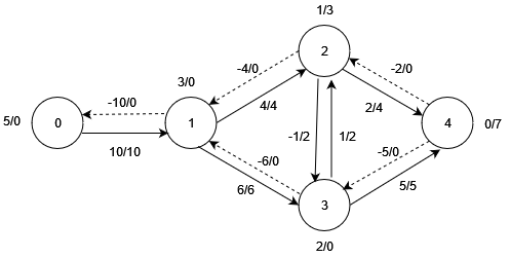
(d) Discharging vertex 2



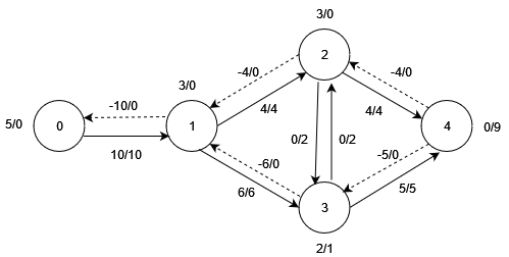
(e) Discharging vertex 3



(f) Discharging vertex 1



(g) Discharging vertex 3

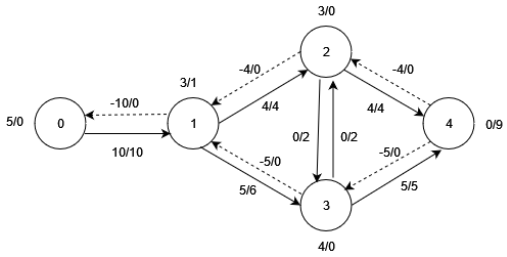


(h) Discharging vertex 2

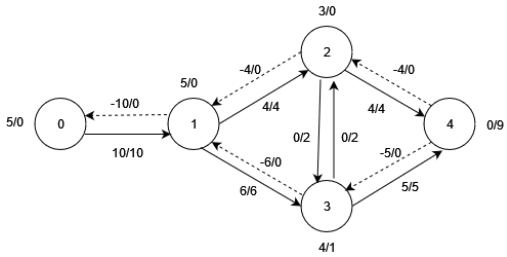
Figure 3.15: Push-relabel worst case (part 1/2)

easily falls into, which significantly harms its running time. This is represented in Figures 3.15 and 3.16. Both these figures are composed of several subfigures to show the evolution of the flow, to understand the origin and cause of this common harmful scenario. In the figures, we represent each vertex as an integer with two labels associated, indicating the height and excess of the vertex in the format height/excess. All edges have corresponding flow and capacity in the format flow/capacity, and dotted arrows indicate that the edge was added in the flow network creation. Note that, for this example's purpose, when in doubt between two or more vertices because the criteria used is tied, we always choose in increasing order of the vertices' identification. For instance, when discharging a vertex, we may have to choose between two vertices to push to. In this situation, we would choose the vertex with the smallest identification number.

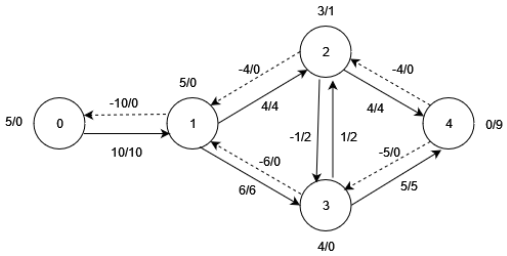
Essentially, and as we can see in Figure 3.15, we have an initial network flow (Figure 3.15a) and the algorithm runs from there until the moment depicted in Figure 3.15h, where it has reached a point in which it cannot take more flow into the sink as all edges that go to the sink have a residual capacity equal to 0. From Figure 3.15h on, any steps taken are simply worsening the running time with no benefits as it is not possible to increase the flow value any more. However, this step is part of the algorithm and, ideally,



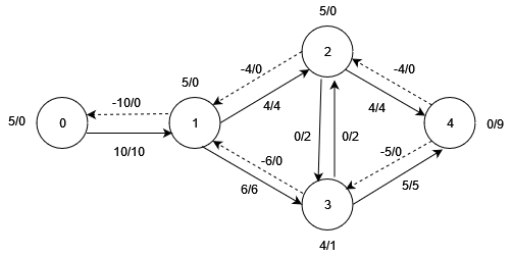
(a) Discharging vertex 3



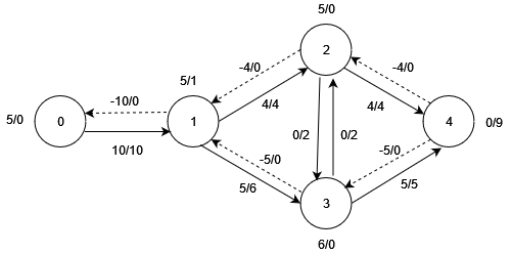
(b) Discharging vertex 1



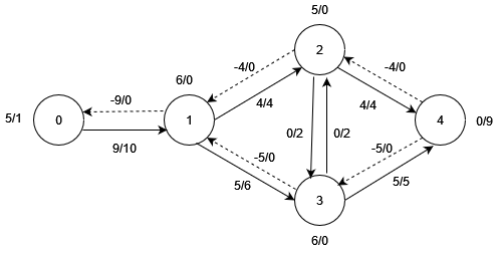
(c) Discharging vertex 3



(d) Discharging vertex 2



(e) Discharging vertex 3



(f) Discharging vertex 1 - Final state

Figure 3.16: Push-relabel worst case (part 2/2)

would not take long to route the excess flow in the network back to the source. As we proceed with the algorithm, in Figure 3.16, the algorithm ends once there are no vertices with excess flow greater than zero, except for the source and sink. However, before that happens, the remaining flow will be pushed between vertices 1, 2 and 3, until these reach a large enough height so that the flow is routed to the source, which only happens in Figure 3.16f.

This way we understand that in this scenario, and many similar ones, the algorithm performs far from ideal. The problem here is that the excess flow will only be routed back to the source once it is in vertex 1, and vertex 2 has a height greater than or equal to source's height, making it choose the source as the destination of the excess flow. Until then, the algorithm aimlessly routes flow from one vertex to another, just for the sake of increasing vertices' heights. Since this algorithm attributes the height label of the source vertex relative to the number of vertices there are, it is easy to conclude that a problem similar to this, except with source's height equal to a very large number, would be proportionally worse. Note that, this example was specifically designed to work the same way with both variants of push-relabel studied, therefore it will have the same outcome using the PRFIFO and PRHL variants.

Summing up, this happens whenever some flow gets stuck in a scenario in which it has to push flow from one vertex to another with the only purpose being increasing the vertices' heights to ultimately allow the flow to reach the source. We believe it would be interesting to try to implement and test a push-relabel algorithm with a global relabeling heuristic, in order to see if that would reduce the damage that this does. Although this is extra work for the algorithm, it periodically updates the height labels of all vertices as a way to avoid unnecessary steps. As a way to check this theory, we monitored the number of pushes and relabels that were performed on each network. To analyze more deeply, note that the push relabel algorithm has a maximum number of relabels of $(2n-1)(n-2)$, which is just under $2n^2$. This is explained because every vertex except for the source and sink ($n-2$ vertices) may be relabeled to, at most, $2n-1$. If we take, for instance, the belgium roads network, with 1 441 295 vertices, the maximum number of relabels would be 4.1×10^{12}. And, as we can see, on networks R27 and R28 (see Table I.3), the number of relabels performed was close to that number. This tells us that the algorithm relabeled almost every vertex to its maximum label, showing very low efficiency [11, 17].

4

Chen et al.'s Algorithm

The way this algorithm works is by reducing the initial network's size while preserving the properties needed to solve the problem. To dive deeper, first we must understand the data structure used, which also explains how this algorithm manages to reduce the network size recursively. The graph reduction is done using two steps, vertex sparsification and edge sparsification. These steps mainly reduce the number of vertices and number of edges in the graph, accordingly.

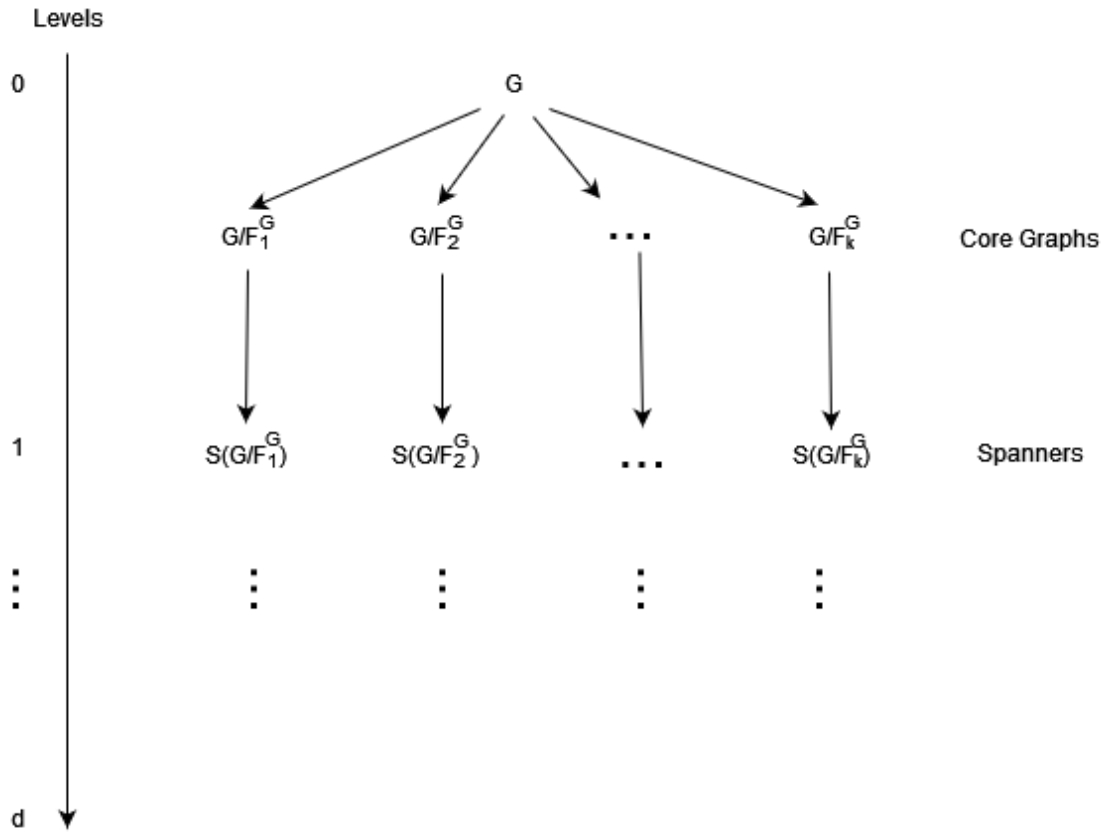


Figure 4.1: Data structure used on Chen et al.'s algorithm - This figure was adapted from a video[1]made by one of the authors of the algorithm.

Consider a spanning tree T of a weighted graph G, with weights w. The stretch of an edge (u, v) of G is defined as how much larger the sum of the weights of the edges between u and v in T are, when compared to the weight of the edge, $w(u, v)$. Hence, the total stretch of the spanning tree T is defined as the sum of stretches of all edges in G. A low stretch spanning tree denotes a spanning tree with a small total stretch.

Therefore, as can be seen in Figure 4.1, we start with an initial weighted graph G. A low stretch spanning tree of G is denoted as T^G. The algorithm then builds k forests $F_1^G, ...F_k^G$ of T^G. It starts by recursing on the forest F_1^G, building the core graph G/F_1^G during the vertex sparsification step, which we will look at later. We then reduce its number of edges in a way that we get a spanner graph $S(G/F_1^G)$ of the core graph, which is also said G_1 and, is a subgraph that preserves, up to a small factor, the lengths of essential shortest paths in the original graph. Just like represented in the figure, at this point we are on level one of recursion, meaning we have recursed once. This graph reduction cycle is repeated for the spanner obtained $S(G/F_1^G)$, or G_1, and called recursively until level d, where the network G_d is small enough that computing the maximum flow is not too complex. Note that Figure 4.1 represents the data structure as if we recurse on all forests simultaneously, however, this is just for illustration purposes as this would be too expensive to maintain. As previously mentioned, we recurse on one forest at a time, starting from $F_1^{G_j}$, with $1 \leq j \leq d$ representing the level of recursion, since that is enough to find a solution and it is much cheaper. So, we would recurse on $F_1^{G_j}$ and build $G_j/F_1^{G_j}$ and $S(G_j/F_1^{G_j})$. The remaining core and spanner graphs would not be initially built [2].

As we briefly described before, the vertex sparsification phase consists of building k forests $F_1^{G_j}, ..., F_k^{G_j}$ of T^{G_j} and, recursing on one of them, for instance $F_i^{G_j}$, with $1 \leq i \leq k$, generating a core graph $G_j/F_i^{G_j}$. A core graph $G_j/F_i^{G_j}$ is essentially produced by contracting each tree in the forest $F_i^{G_j}$ into a single vertex in G_j, simultaneously reducing the number of vertices and edges. Although it also reduces the number of edges, it is not enough. Hence, the algorithm performs the edge sparsification step, which gives a graph with less edges that approximately preserves the distances between vertices.

Now that we understand in a general way the data structure used, we can comprehend why it is used and the purpose of its characteristics. We have an initial weighted graph $G = (V, E)$ with gradients g and lengths l, where $g : E \to \mathbb{R}$ and $l : E \to \mathbb{R}$ are functions that depend on the flow. Moreover, each edge (u, v) has a cost, a lower capacity and an upper capacity, respectively denoted as $cost(u, v)$, $c^-(u, v)$ and $c^+(u, v)$. On each level of recursion, starting from level zero, we recurse on one forest at a time, always starting from the first one, building its correspondent core and spanner graphs recursively. We then check if we find a cycle Δ, in the chain composed by all recursively produced graphs $G, G_1, ...G_d$, that minimizes the following expression $\frac{|g \cdot \Delta|}{l \cdot \Delta}$. This ratio measures the gradient g per unit length l of the cycle Δ. If needed, we may perform a shift to $F_2^{G_j}$ and rebuild the chain from the level j downwards, now recursing on $F_2^{G_j}$ and thus on $G_j/F_2^{G_j}$.

[1]The video can be found here: https://youtu.be/mriENW-oVSk?si=a2LJ30kxZnStcZpM

Shift and rebuild are operations of the data structure which, respectively, shift the forest that is being recursed on in a specific level and, rebuild the data structure chain from a certain level down. These operations are usually used together since, when a shift is performed, a rebuild is done by recursing on the newly picked forest to get the new chain $G, G_1, ...G_d$.

The process used to compute a low stretch spanning tree from a weighted graph (the weights represent the edges' lengths) is the petal-decomposition [1], which manages to build a spanning tree while controlling its total stretch. We adopt the notations of $E(V)$ and $G(V)$ from the original article, which accordingly denote all edges whose both endpoints are in V and the graph with vertices V and all edges $E(V)$. Furthermore, we represent a shortest path between vertices u and v as $P_{u,v}$, and the distance between them as $d(u, v)$, which refers to the sum of weights of the edges involved in that path. This work is able to get an average total stretch of $O(\log n \log\log n)$ in time $O(m \log n \log\log n)$.

As the name suggests, it uses a process to divide the network into clusters, also called petals, which connected form a spanning tree. The process starts on an arbitrary vertex x_0 and it groups vertices into petals based on their distance to the vertex x_0 and specific shortest paths. When all petals are created, the remaining vertices are all grouped to become part of the central petal, the stigma, whose central vertex is x_0. Once the network is split into petals, the process is recursively ran within each petal until it is split into single vertices, to ensure that the tree spans all vertices. In order to dive deeper, we must know that each petal X_i has three parameters: a central vertex x_i, a target vertex t_i, and a radius r_i. More specifically, the central vertex x_i is always in the path from the target t_i to x_0, for every petal X_i. The radius r_i is what defines the distance of the highway of the petal, which denotes the path P_{x_i,t_i}. The edges contained in a highway see their weights get reduced by half, as a way to encourage its use.

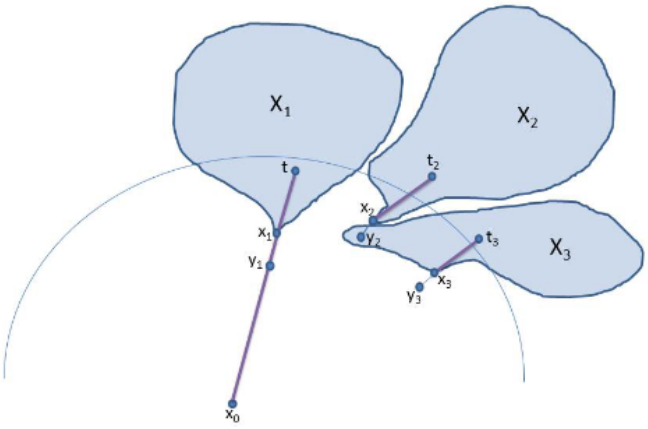
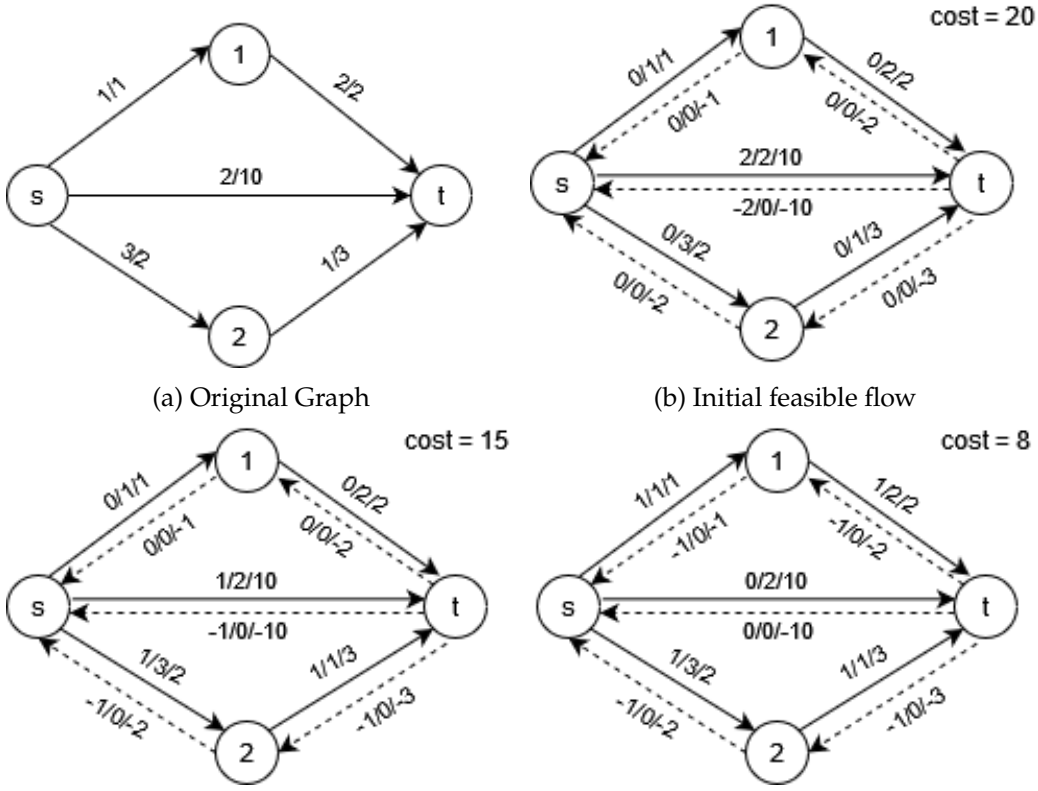


Figure 4.2: Petal decomposition example - For a petal i, X_i, x_i, t_i and y_i, respectively represent the petal identification, the central vertex, the target and the neighbor vertices. The thicker lines represent the highways of each petal, while the thinner ones represent the portal edges. This figure was borrowed from the original article [1].

Once we get to the end of a decomposition with s petals, they are all connected by adding the edges $(y_1, x_1), ..., (y_s, x_s)$. y_i is what the authors call a neighbor of the petal X_i, which is defined as the nearest vertex from x_i, outside of the petal, on the path P_{x_i,x_0}. An edge (y_i, x_i) is called a portal edge. By doing the decomposition in petals recursively, we ensure all vertices are connected on the last iteration where each petal represents a single vertex. In Figure 4.2, there is an example from the original article of a petal decomposition with the corresponding central, target and neighbor vertices represented respectively as x_i, t_i and y_i. The X_i notation identifies each petal [1].



(a) Original Graph

(b) Initial feasible flow

(c) First cycle: $(t, s), (s, 2), (2, t)$

(d) Second cycle: $(t, s), (s, 1), (1, t)$

Figure 4.3: Example of reducing flow cost with cycles - In this example we want to route 2 units of flow from s to t with minimum cost on the graph in Figure 4.3a. Figure 4.3b represents the initial feasible flow on a flow network, and Figures 4.3c and 4.3d represent the updates made to get an optimal flow. Mind that the vertices are identified with a number or s or t for source and sink, respectively. In the original graph, in Figure 4.3a, each edge displays an upper capacity and a cost represented in the format upper capacity/cost and in the remaining figures, representing the flow network, each edge displays the format flow/upper capacity/cost. The lower capacity of every edge is equal to 0.

Finally, we talk about how this algorithm manages to start with an initial feasible flow, which is probably not optimal, and progressively improve it until it meets its criteria. To understand, we use a simplified version that tries to get two units of flow from the source s to the sink t while minimizing the flow cost instead of a more complex ratio. We use Figure 4.3 to demonstrate this process. It starts with the graph in Figure 4.3a. The first

flow found in the network flow is not optimal (see Figure 4.3b). However, it then looks for cycles that are capable of reducing the total flow cost. As it can be seen, in Figure 4.3c a cycle $(t, s), (s, 2), (2, t)$ is found, which when applied, reduces the total cost by 5 units. The reduction in cost is explained by rerouting a unit of flow that was taking the (s, t) edge (cost equals 10) to the path $(s, 2), (2, t)$ (cost equals 5). In Figure 4.3d there is one more cycle that manages to reduce the cost by 7 units, rerouting one unit of flow taking the (s, t) edge (cost equals 10) to the path $(s, 1), (1, t)$ (cost equals 3), which is much cheaper. Note that in this example, the edges introduced in the network flow have a cost symmetric to their reverse edge as they are used as undo edges, to reroute flow.

5

Conclusions and Future Work

In this dissertation, we have implemented and tested five algorithms. These were the Edmonds-Karp [13], two variants of the Dinic's algorithm [12], a classic recursive and an iterative version, and two variants of the push-relabel [17]. We implemented a first variant with a FIFO protocol and another one with highest label. The algorithms were tested in distinct characteristic networks as a way of trying to take valuable insights. We used three different types of networks, with two of them being artificial networks generated by us, giving us the option to try specific configurations to dig deeper into our analysis. And lastly, we used real-life networks that we got from the following network repository: https://networkrepository.com/ [24]. Moreover, we have studied and explained some sections of the Chen et al.'s algorithm [5, 6] in an attempt to simplify the understanding of this very complex algorithm.

The results that were obtained reveal that, in fact, there are a lot of factors and particularities of each algorithm that are not accounted for and are beyond the theoretical time complexity of an algorithm. More specifically, we now have a much deeper comprehension of each of the five algorithms as we got to know some of their preferences. The highlight that stands out the most is that both push-relabel variants were the worst performing algorithms, contrary to what we expected due to their time complexity. However, we believe we understood and were able to clearly explain why this occurred, revealing a rather common scenario of this algorithm which can harm its performance massively. Another insight highlight was each algorithm's preference for either wider or narrower networks according to the running times, which was possible by monitoring the running times by number of columns of the networks generated by us.

This dissertation contributes to a deeper comprehension of the algorithms studied. In specific, the differences between the time complexity of a given algorithm and the running time that it actually takes to find a maximum flow. Furthermore, it gives an introduction to the Chen et al.'s algorithm for someone who wants to dive deeper into it. It is not always easy to understand the concepts used and the details of all procedures and therefore, this dissertation gives a simplified way to get a first grasp of some parts of it, and of the data structure used. It can also be used to build some intuition to better understand the

algorithm.

While this dissertation has provided some good insights over maximum flow algorithms, unfortunately, due to time constraints, it was not possible to complete all the work that we would have liked and believe that would have brought a lot of value to this work. Therefore, there remain many opportunities in this area to investigate more and to build on findings and new directions such as some of the work done in this dissertation.

We believe that some of the work that was done here has its importance and value to the area and, therefore is worth being continued and built on by future works. We are talking about some of the highlights and main contributions given by this dissertation. The biggest work that would be suggested for future works would be adding on to the work done of simplifying and breaking down each complex procedure used in the Chen et al.'s algorithm. The paper presenting this algorithm is written in a very math-focused and technical way, with the main purpose of it being to clearly prove that each procedure used is in fact feasible and is aligned with the record time complexity given by the authors. This, combined with its highly sophisticated core content, produced a paper that is not straightforward to understand and hard to translate into code. Considering all these factors, a future work that continues clarifying this algorithm is of great importance to this field as it will ease the access of the algorithm to many other people. Furthermore, the efficiency issue with the push-relabel could also be a topic of future work, trying to understand exactly what conditions provoke it.

Bibliography

[1] I. Abraham and O. Neiman. "Using Petal-Decompositions to Build a Low Stretch Spanning Tree". In: *SIAM Journal on Computing* 48.2 (2019), pp. 227–248. DOI: 10.1137/17M1115575. eprint: https://doi.org/10.1137/17M1115575. URL: https://doi.org/10.1137/17M1115575 (cit. on pp. 49, 50).

[2] R. Ahmed et al. "Graph spanners: A tutorial review". In: *Computer Science Review* 37 (2020), p. 100253 (cit. on p. 48).

[3] R. K. Ahuja et al. "Computational investigations of maximum flow algorithms". In: *European Journal of Operational Research* 97.3 (1997), pp. 509–542 (cit. on pp. 14, 20, 22, 24, 43).

[4] Y. Boykov and V. Kolmogorov. "An experimental comparison of min-cut/max- flow algorithms for energy minimization in vision". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26.9 (2004), pp. 1124–1137. DOI: 10.1109/TPAMI.2004.60 (cit. on pp. 22, 43).

[5] J. van den Brand et al. *A Deterministic Almost-Linear Time Algorithm for Minimum-Cost Flow*. 2023. arXiv: 2309.16629 [cs.DS] (cit. on pp. 1, 2, 8, 18, 19, 21, 53).

[6] J. van den Brand et al. "A deterministic almost-linear time algorithm for minimum-cost flow". In: *2023 IEEE 64th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE. 2023, pp. 503–514 (cit. on pp. 2, 8, 18, 21, 53).

[7] L. Chen et al. "Almost-Linear Time Algorithms for Incremental Graphs: Cycle Detection, SCCs, *s-t* Shortest Path, and Minimum-Cost Flow". In: *arXiv preprint arXiv:2311.18295* (2023) (cit. on pp. 1, 21).

[8] L. Chen et al. "Almost-Linear-Time Algorithms for Maximum Flow and Minimum-Cost Flow". In: *Commun. ACM* 66.12 (2023-11), pp. 85–92. ISSN: 0001-0782. DOI: 10.1145/3610940. URL: https://doi.org/10.1145/3610940 (cit. on pp. 2, 18, 20).

[9] L. Chen et al. *Maximum Flow and Minimum-Cost Flow in Almost-Linear Time*. 2022. arXiv: 2203.00671 [cs.DS] (cit. on pp. 1, 18, 20).

[10] L. Chen et al. "Maximum Flow and Minimum-Cost Flow in Almost-Linear Time". In: *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*. 2022, pp. 612–623. DOI: 10.1109/FOCS54457.2022.00064 (cit. on pp. 18, 20).

[11] B. V. Cherkassky and A. V. Goldberg. "On implementing the push—relabel method for the maximum flow problem". In: *Algorithmica* 19 (1997), pp. 390–410 (cit. on pp. 7, 17, 18, 46).

[12] Y. Dinitz. "Dinitz'algorithm: The original version and Even's version". In: *Theoretical Computer Science: Essays in Memory of Shimon Even*. Springer, 2006, pp. 218–240 (cit. on pp. 1, 2, 11, 20, 21, 53).

[13] J. Edmonds and R. M. Karp. "Theoretical improvements in algorithmic efficiency for network flow problems". In: *Journal of the ACM (JACM)* 19.2 (1972), pp. 248–264 (cit. on pp. 1, 2, 9, 10, 19, 21, 53).

[14] L. R. Ford and D. R. Fulkerson. "A simple algorithm for finding maximal network flows and an application to the Hitchcock problem". In: *Canadian journal of Mathematics* 9 (1957), pp. 210–218 (cit. on pp. 8, 19).

[15] J. M. Friis. "An experimental comparison of max flow algorithms". MA thesis. Aarhus Universitet, Datalogisk Institut, 2014 (cit. on pp. 9, 21, 22, 42).

[16] Z. Galil and A. Naamad. "Network flow and generalized path compression". In: *Proceedings of the eleventh annual ACM symposium on Theory of computing*. 1979, pp. 13–26 (cit. on pp. 20, 21).

[17] A. V. Goldberg and R. E. Tarjan. "A new approach to the maximum-flow problem". In: *Journal of the ACM (JACM)* 35.4 (1988), pp. 921–940 (cit. on pp. 2, 5, 7, 15, 18, 20, 46, 53).

[18] A. V. Goldberg and R. E. Tarjan. "Efficient maximum flow algorithms". In: *Communications of the ACM* 57.8 (2014), pp. 82–89 (cit. on pp. 5, 7).

[19] D. Goldfarb and M. D. Grigoriadis. "A computational comparison of the dinic and network simplex methods for maximum flow". In: *Annals of Operations Research* 13.1 (1988), pp. 81–123 (cit. on p. 14).

[20] A. Karzanov. "Determining the maximal flow in a network by the method of preflows". In: *Doklady Mathematics* 15 (1974-02), pp. 434–437 (cit. on pp. 20, 21).

[21] P. Kovács. "Minimum-cost flow algorithms: an experimental evaluation". In: *Optimization Methods and Software* 30.1 (2015), pp. 94–127 (cit. on p. 19).

[22] P. Lammich and S. R. Sefidgar. "Formalizing the edmonds-karp algorithm". In: *Interactive Theorem Proving: 7th International Conference, ITP 2016, Nancy, France, August 22-25, 2016, Proceedings 7*. Springer. 2016, pp. 219–234 (cit. on pp. 8, 10).

[23] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/main/template.pdf (cit. on p. i).

[24] R. A. Rossi and N. K. Ahmed. "The Network Data Repository with Interactive Graph Analytics and Visualization". In: *AAAI*. 2015. URL: https://networkrepository.com (cit. on pp. 24, 53).

[25] D. D. Sleator and R. Endre Tarjan. "A data structure for dynamic trees". In: *Journal of Computer and System Sciences* 26.3 (1983), pp. 362–391. ISSN: 0022-0000. DOI: https://doi.org/10.1016/0022-0000(83)90006-5. URL: https://www.sciencedirect.com/science/article/pii/0022000083900065 (cit. on pp. 20, 21).

I

Data Set and Experimental Results

In this annex there are multiple tables displaying all the data and results that were obtained in this work. For each of the three types of networks, we have three different tables exhibiting the metrics of the networks used in testing, the algorithms' running times (in ms) and the other algorithms' metrics, in this order.

In these tables, we use some abbreviations. "NetId" stands for the network identifier, which identifies each network by starting with a letter that codes the type of network: "R" for real life networks, "G" for grid, and "L" for random layered networks, followed by a unique number, which is ordered by increasing number of edges. We also use the n, m and c notations, respectively for the number of vertices, number of edges and number of columns. The "Max Flow" column indicates the maximum flow value of the network with the source and sink vertices used.

In the tables with other algorithms' metrics, there are columns with the indication of the push-relabel variant, PRFIFO or PRHL, followed by either letter "R" or "P". Letter "R" indicates that the column has the number of relabels executed by the according variant and the letter "P" indicates the number of pushes done. Moreover, there is also "# APs" and "AP Av. Length", which respectively denote the total number of augmenting paths found and their average length. Note that, because the EK, DinicIt and the DinicRec all find shortest augmenting paths, the number of augmenting paths found and their average lengths are all very similar to one another, oftentimes being the same. In our results, due to the way they were implemented, the EK and DinicRec always share the same numbers and the DinicIt may differ slightly in some cases. Therefore, the values shown are those from the EK and DinicRec.

It is also important to say that, for real-life networks, we omit the source and sink vertices since that information would not add much value to the reader. However, that is the reason why some networks have the same values for all remaining metrics but may still display much different results.

Table I.1: Real-life networks' metrics

NetId	Network Name	n	m	Max Flow
R1	Chesapeak Roads	39	340	11
R2	Chesapeak Roads	39	340	4
R3	Chesapeak Roads	39	340	5
R4	Chesapeak Roads	39	340	4
R5	Chesapeak Roads	39	340	4
R6	USA Flights	332	4 252	13
R7	USA Flights	332	4 252	1 637
R8	USA Flights	332	4 252	642
R9	USA Flights	332	4 252	174
R10	USA Flights	332	4 252	208
R11	Luxembourg Roads	114 599	239 332	1
R12	Luxembourg Roads	114 599	239 332	2
R13	Luxembourg Roads	114 599	239 332	1
R14	Luxembourg Roads	114 599	239 332	2
R15	Luxembourg Roads	114 599	239 332	2
R16	USA Roads	126 146	323 900	1
R17	USA Roads	126 146	323 900	2
R18	USA Roads	126 146	323 900	2
R19	USA Roads	126 146	323 900	1
R20	USA Roads	126 146	323 900	2
R21	Pa Roads	1 087 562	3 083 028	1
R22	Pa Roads	1 087 562	3 083 028	1
R23	Pa Roads	1 087 562	3 083 028	3
R24	Pa Roads	1 087 562	3 083 028	1
R25	Pa Roads	1 087 562	3 083 028	1
R26	Belgium Roads	1 441 295	3 099 940	2
R27	Belgium Roads	1 441 295	3 099 940	1
R28	Belgium Roads	1 441 295	3 099 940	1
R29	Belgium Roads	1 441 295	3 099 940	2
R30	Belgium Roads	1 441 295	3 099 940	1
R31	Netherlands Roads	2 216 688	4 882 476	2
R32	Netherlands Roads	2 216 688	4 882 476	1
R33	Netherlands Roads	2 216 688	4 882 476	1
R34	Netherlands Roads	2 216 688	4 882 476	1
R35	Netherlands Roads	2 216 688	4 882 476	1
R36	Ca Roads	1 957 027	5 520 776	3
R37	Ca Roads	1 957 027	5 520 776	3
R38	Ca Roads	1 957 027	5 520 776	1
R39	Ca Roads	1 957 027	5 520 776	1
R40	Ca Roads	1 957 027	5 520 776	2

Table I.2: Algorithms' running times (in ms) for real-life networks

NetId	EK	DinicIt	DinicRec	PRFIFO	PRHL
R1	0.02	0.04	0.02	0.01	0.09
R2	0.01	0.02	0.02	0.01	0.03
R3	0.01	0.03	0.02	0.47	0.47
R4	0.02	0.02	0.02	0.02	0.08
R5	0.01	0.03	0.02	0.01	0.01
R6	0.05	0.08	0.08	18.57	24.13
R7	0.03	0.15	0.13	0.07	0.08
R8	0.03	0.19	0.07	0.04	0.04
R9	0.01	0.03	0.05	0.08	0.09
R10	0.05	0.17	0.12	14.91	14.15
R11	10.36	21.41	20.75	7.23E+05	7.43E+05
R12	21.86	22.34	23.42	47.61	5.82
R13	14.05	22.19	27.16	8.25E+05	8.66E+05
R14	9.10	26.99	18.73	20.83	21.10
R15	12.21	46.37	21.30	12.81	11.49
R16	15.93	18.36	25.20	1 732.72	1 585.80
R17	1.73	45.28	54.19	326.41	314.92
R18	4.87	45.10	56.87	474.49	466.43
R19	11.22	24.41	31.69	381.40	310.67
R20	1.45	42.72	41.74	9.31	9.68
R21	238.70	621.61	548.41	7.70E+07	7.98E+07
R22	2.32	330.74	260.97	2 571.53	2902.25
R23	170.49	650.19	479.86	43.54	51.45
R24	96.14	220.09	175.96	3 547.82	3 776.75
R25	32.37	149.69	248.12	1.51E+04	1.51E+04
R26	455.63	610.38	615.13	608.08	785.98
R27	412.22	606.82	699.62	1.46E+08	1.59E+08
R28	359.09	588.96	638.30	1.28E+08	1.34E+08
R29	386.74	605.70	475.17	137.84	228.48
R30	185.74	234.49	288.55	1369.64	1 228.89
R31	870.42	923.66	632.60	5 441.22	6 002.25
R32	316.21	419.41	468.39	4 852.04	5 474.77
R33	98.71	509.74	292.64	1 553.32	1 556.55
R34	195.54	415.29	461.42	2 108.20	2 710.49
R35	353.93	586.86	407.78	8 332.08	9 671.76
R36	397.13	1 842.44	1 205.91	2 041.63	2 084.78
R37	516.60	1 504.92	1 133.53	4 237.68	4 742.23
R38	200.42	604.37	320.00	7 221.14	7 483.61
R39	198.10	655.36	536.29	1.15E+04	1.21E+04
R40	183.15	718.69	885.91	858.42	1 208.48

Table I.3: Other algorithms' metrics for real-life networks

NetId	PRFIFO R	PRFIFO P	PRHL R	PRHL P	# APs	AP Av. Length
R1	35	51	38	52	11	2.2
R2	32	42	31	43	4	2.8
R3	1 240	1 467	1 221	1 458	5	2.4
R4	52	69	54	81	4	2.5
R5	28	37	37	54	4	2.8
R6	7.7E+04	1.3E+05	7.8E+04	1.2E+05	1	5.0
R7	442	711	411	762	5	3.8
R8	288	390	268	379	6	3.7
R9	527	726	493	713	1	3.0
R10	7.9E+04	1.1E+05	8.1E+04	1.1E+05	2	3.5
R11	1.2E+10	1.3E+10	1.2E+10	1.3E+10	1	125.0
R12	3.5E+05	3.7E+05	8.9E+04	9.3E+04	2	634.0
R13	1.2E+10	1.3E+10	1.2E+10	1.3E+10	1	192.0
R14	3.8E+05	4.1E+05	3.3E+05	3.5E+05	2	630.5
R15	1.0E+05	1.1E+05	1.5E+05	1.6E+05	2	800.0
R16	1.4E+07	1.8E+07	1.4E+07	1.8E+07	1	476.0
R17	2.2E+06	2.8E+06	2.2E+06	2.8E+06	2	56.0
R18	4.1E+06	5.1E+06	4.1E+06	5.2E+06	2	134.5
R19	3.2E+06	4.0E+06	3.2E+06	4.0E+06	1	379.0
R20	1.1E+05	1.4E+05	1.2E+05	1.5E+05	2	66.5
R21	8.4E+11	1.2E+12	8.4E+11	1.2E+12	1	68.0
R22	2.0E+07	2.6E+07	2.0E+07	2.6E+07	1	35.0
R23	2.7E+05	3.4E+05	3.0E+05	3.8E+05	3	340.7
R24	3.0E+07	4.0E+07	3.0E+07	4.0E+07	1	215.0
R25	1.3E+08	1.7E+08	1.3E+08	1.7E+08	1	149.0
R26	5.0E+06	5.5E+06	5.7E+06	6.2E+06	2	819.0
R27	1.9E+12	2.1E+12	1.9E+12	2.1E+12	1	523.0
R28	1.9E+12	2.1E+12	1.9E+12	2.1E+12	1	369.0
R29	1.0E+06	1.1E+06	1.6E+06	1.7E+06	2	753.0
R30	1.6E+07	1.7E+07	1.6E+07	1.7E+07	1	762.0
R31	5.5E+07	6.4E+07	5.4E+07	6.1E+07	2	1 353.0
R32	5.3E+07	5.9E+07	5.3E+07	5.9E+07	1	880.0
R33	1.6E+07	1.8E+07	1.6E+07	1.8E+07	1	603.0
R34	2.9E+07	3.0E+07	2.9E+07	3.0E+07	1	1 178.0
R35	1.3E+08	1.4E+08	1.3E+08	1.4E+08	1	1 713.0
R36	1.3E+07	1.8E+07	1.2E+07	1.6E+07	3	186.7
R37	3.1E+07	4.1E+07	2.9E+07	3.9E+07	3	269.0
R38	6.1E+07	8.2E+07	6.1E+07	8.2E+07	1	404.0
R39	1.0E+08	1.4E+08	1.0E+08	1.4E+08	1	310.0
R40	6.1E+06	8.1E+06	6.2E+06	8.3E+06	2	159.0

Table I.4: Grid networks' metrics (part 1/2)

NetId	n	m	c	Max Flow
G1	10 002	29 830	125	2 987
G2	10 002	29 900	100	3 686
G3	10 002	29 965	80	4 723
G4	10 002	30 100	50	7 950
G5	10 002	30 460	20	20 092
G6	20 002	59 910	125	6 114
G7	20 002	60 000	100	7 609
G8	20 002	60 090	80	9 496
G9	20 002	60 300	50	15 492
G10	20 002	60 960	20	40 025
G11	30 002	89 990	125	9 082
G12	30 002	90 100	100	11 445
G13	30 002	90 215	80	14 563
G14	30 002	90 500	50	23 411
G15	30 002	91 460	20	59 863
G16	40 002	120 070	125	12 313
G17	40 002	120 200	100	15 178
G18	40 002	120 340	80	19 204
G19	40 002	120 700	50	31 579
G20	40 002	121 960	20	79 067
G21	50 002	150 150	125	15 003
G22	50 002	150 300	100	19 246
G23	50 002	150 465	80	24 117
G24	50 002	150 900	50	38 677
G25	50 002	152 460	20	100 382
G26	60 002	180 230	125	18 766
G27	60 002	180 400	100	23 060
G28	60 002	180 590	80	28 910
G29	60 002	181 100	50	46 339
G30	60 002	182 960	20	120 152
G31	70 002	210 310	125	21 841
G32	70 002	210 500	100	27 088
G33	70 002	210 715	80	33 560
G34	70 002	211 300	50	54 923
G35	70 002	213 460	20	139 614
G36	80 002	240 390	125	24 851
G37	80 002	240 600	100	30 817
G38	80 002	240 840	80	39 059
G39	80 002	241 500	50	62 001
G40	80 002	243 960	20	157 946
G41	90 002	270 470	125	27 433
G42	90 002	270 700	100	34 356
G43	90 002	270 965	80	44 088
G44	90 002	271 700	50	69 806
G45	90 002	274 460	20	179 361

Table I.5: Grid networks' metrics (part 2/2)

NetId	n	m	c	Max Flow
G46	100 002	300 550	125	30 934
G47	100 002	300 800	100	39 007
G48	100 002	301 090	80	47 580
G49	100 002	301 900	50	78 470
G50	100 002	304 960	20	199 716

Table I.6: Algorithms' running times (in ms) for grid networks (part 1/2)

NetId	EK	DinicIt	DinicRec	PRFIFO	PRHL
G1	726.6	508.8	160.7	3 428.0	4 715.2
G2	874.0	519.2	122.2	627.9	933.3
G3	998.8	516.6	128.9	2 135.5	4 471.4
G4	1 386.3	763.5	171.6	4 637.1	5 917.1
G5	1 916.3	828.3	107.6	3 315.3	4 030.1
G6	3 230.2	1 829.2	497.0	4 127.8	6 518.5
G7	3 877.1	1 905.8	426.5	1.25E+04	2.81E+04
G8	4 567.3	1 980.7	375.6	3 907.1	6 745.9
G9	6 140.5	2 502.6	348.8	1.79E+04	3.25E+04
G10	8 939.8	3 272.3	250.9	1.51E+04	2.69E+04
G11	9 899.7	3 856.5	969.1	4.76E+04	7.33E+04
G12	1.10E+04	4 272.7	900.6	3.95E+04	8.00E+04
G13	1.35E+04	4 612.9	730.1	1.79E+04	3.22E+04
G14	1.87E+04	5 748.6	698.7	3.64E+04	6.13E+04
G15	2.19E+04	7 256.7	520.4	3.62E+04	6.90E+04
G16	2.04E+04	7 306.4	1 611.6	3.57E+04	7.23E+04
G17	2.34E+04	8 156.5	1 762.8	4.77E+04	1.08E+05
G18	3.26E+04	8 578.0	1 321.8	6.74E+04	1.14E+05
G19	4.41E+04	1.09E+04	1 418.0	2.69E+04	5.42E+04
G20	5.59E+04	1.33E+04	577.4	5.25E+04	9.70E+04
G21	3.72E+04	1.14E+04	1 886.5	2.07E+05	2.86E+05
G22	4.82E+04	1.40E+04	2 312.3	5.21E+04	1.14E+05
G23	5.77E+04	1.51E+04	2 017.0	1.74E+05	2.62E+05
G24	7.17E+04	1.66E+04	1 516.1	3.06E+04	6.99E+04
G25	9.75E+04	2.25E+04	1 118.3	1.14E+05	2.03E+05
G26	6.09E+04	2.05E+04	3 706.0	1.50E+05	3.04E+05
G27	7.57E+04	2.17E+04	3 215.3	7.43E+04	1.91E+05
G28	8.66E+04	2.23E+04	2 925.4	1.45E+05	2.73E+05
G29	9.74E+04	2.47E+04	1 960.1	1.85E+05	2.96E+05
G30	1.53E+05	3.62E+04	1 110.7	1.63E+05	2.69E+05
G31	8.84E+04	2.85E+04	5 016.1	1.90E+05	4.14E+05
G32	1.01E+05	2.91E+04	4 201.1	1.10E+05	1.90E+05
G33	1.15E+05	3.34E+04	3 136.3	3.58E+05	4.77E+05
G34	1.56E+05	4.12E+04	3 174.0	1.90E+05	3.47E+05
G35	2.01E+05	4.95E+04	1 469.6	1.69E+05	3.10E+05
G36	1.15E+05	3.49E+04	6 184.5	2.80E+05	5.08E+05
G37	1.32E+05	3.75E+04	4 287.0	2.75E+05	5.07E+05
G38	1.58E+05	4.87E+04	3 984.7	4.74E+05	6.91E+05
G39	1.89E+05	5.58E+04	3 687.8	3.23E+05	4.81E+05
G40	2.76E+05	7.03E+04	1 668.7	2.65E+05	4.91E+05
G41	1.40E+05	4.40E+04	4 885.6	1.17E+05	2.93E+05
G42	1.60E+05	5.33E+04	5 350.1	9.89E+04	2.08E+05
G43	2.01E+05	6.56E+04	5 880.6	5.09E+05	8.25E+05
G44	2.54E+05	7.62E+04	4 051.0	4.46E+05	6.18E+05
G45	3.64E+05	8.76E+04	1 993.2	2.56E+05	4.71E+05

Table I.7: Algorithms' running times (in ms) for grid networks (part 2/2)

NetId	EK	DinicIt	DinicRec	PRFIFO	PRHL
G46	1.90E+05	6.06E+04	7 549.2	4.82E+05	8.58E+05
G47	2.23E+05	7.30E+04	7 069.0	2.45E+05	4.65E+05
G48	2.31E+05	7.85E+04	5 610.3	3.32E+05	5.70E+05
G49	3.25E+05	9.04E+04	4 782.2	3.17E+05	5.51E+05
G50	4.24E+05	1.23E+05	2 527.0	4.37E+05	6.97E+05

Table I.8: Other algorithms' metrics for grid networks (part 1/2)

NetId	PRFIFO R	PRFIFO P	PRHL R	PRHL P	# APs	AP Av. Length
G1	3.34E+07	6.47E+07	4.08E+07	6.19E+07	2 182	172.0
G2	6.72E+06	1.24E+07	8.67E+06	1.17E+07	2 559	136.6
G3	2.02E+07	4.14E+07	2.72E+07	3.79E+07	2 941	112.8
G4	5.35E+07	8.61E+07	7.06E+07	8.57E+07	3 952	80.9
G5	3.22E+07	5.85E+07	4.24E+07	5.64E+07	5 393	33.8
G6	4.22E+07	8.37E+07	5.40E+07	7.75E+07	4 493	176.4
G7	1.19E+08	2.19E+08	1.57E+08	2.08E+08	5 186	140.7
G8	3.54E+07	7.21E+07	4.17E+07	6.88E+07	5 847	114.0
G9	1.51E+08	2.96E+08	2.22E+08	2.76E+08	7 588	74.9
G10	1.33E+08	2.51E+08	1.86E+08	2.41E+08	10 732	33.1
G11	4.50E+08	7.43E+08	6.15E+08	7.23E+08	6 693	174.6
G12	3.69E+08	6.31E+08	4.97E+08	6.14E+08	7 774	142.5
G13	1.50E+08	3.02E+08	1.96E+08	2.80E+08	9 066	113.8
G14	2.99E+08	5.86E+08	4.46E+08	5.47E+08	11 452	69.0
G15	2.87E+08	5.61E+08	3.77E+08	5.34E+08	15 682	25.6
G16	3.30E+08	6.12E+08	4.76E+08	5.71E+08	9 079	161.1
G17	4.42E+08	8.12E+08	6.20E+08	7.80E+08	10 495	123.2
G18	5.04E+08	1.01E+09	7.40E+08	9.37E+08	11 993	99.9
G19	2.37E+08	4.61E+08	3.12E+08	4.44E+08	15 527	61.8
G20	4.17E+08	8.23E+08	5.35E+08	7.73E+08	20 762	23.9
G21	1.14E+09	2.29E+09	1.73E+09	2.07E+09	11 022	150.1
G22	4.18E+08	8.46E+08	5.95E+08	8.01E+08	13 241	116.4
G23	1.10E+09	2.10E+09	1.62E+09	1.99E+09	14 993	95.1
G24	3.09E+08	5.47E+08	3.67E+08	5.30E+08	19 013	58.8
G25	8.11E+08	1.55E+09	1.08E+09	1.49E+09	26 765	23.1
G26	1.28E+09	2.27E+09	1.77E+09	2.22E+09	13 886	143.9
G27	6.62E+08	1.26E+09	9.23E+08	1.21E+09	15 786	113.1
G28	1.41E+09	2.34E+09	1.65E+09	2.23E+09	18 100	92.4
G29	1.11E+09	2.26E+09	1.57E+09	2.10E+09	22 640	57.0
G30	1.05E+09	2.05E+09	1.37E+09	1.96E+09	32 008	22.6
G31	1.46E+09	2.70E+09	2.21E+09	2.61E+09	16 146	140.4
G32	8.65E+08	1.69E+09	1.29E+09	1.59E+09	18 547	111.1
G33	1.77E+09	3.69E+09	2.89E+09	3.38E+09	20 845	90.5
G34	1.29E+09	2.53E+09	1.93E+09	2.39E+09	27 192	55.9
G35	1.19E+09	2.31E+09	1.54E+09	2.21E+09	37 276	22.3
G36	2.45E+09	3.56E+09	2.89E+09	3.52E+09	18 458	138.3
G37	1.70E+09	3.31E+09	2.48E+09	3.12E+09	21 248	109.4
G38	2.49E+09	4.90E+09	3.97E+09	4.63E+09	24 510	89.4
G39	1.96E+09	3.92E+09	2.74E+09	3.68E+09	30 363	55.1
G40	1.73E+09	3.38E+09	2.28E+09	3.21E+09	41 526	22.0
G41	9.66E+08	1.94E+09	1.22E+09	1.82E+09	20 256	136.5
G42	8.07E+08	1.66E+09	1.17E+09	1.54E+09	23 524	108.3
G43	3.63E+09	5.96E+09	4.55E+09	5.77E+09	27 847	88.2
G44	2.53E+09	5.13E+09	3.72E+09	4.81E+09	34 546	54.5
G45	1.73E+09	3.35E+09	2.20E+09	3.19E+09	47 519	21.8

Table I.9: Other algorithms' metrics for grid networks (part 2/2)

NetId	PRFIFO R	PRFIFO P	PRHL R	PRHL P	# APs	AP Av. Length
G46	2.82E+09	5.62E+09	4.51E+09	5.26E+09	22 917	135.1
G47	2.60E+09	3.99E+09	3.08E+09	3.88E+09	26 927	107.3
G48	2.06E+09	4.28E+09	2.98E+09	3.94E+09	29 656	87.4
G49	2.07E+09	4.06E+09	2.89E+09	3.87E+09	38 896	54.1
G50	2.57E+09	5.05E+09	3.33E+09	4.80E+09	52 715	21.7

Table I.10: Random layered networks' metrics (part 1/3)

NetId	n	m	c	FEP	SEP	Max Flow
L1	2 000	2 723	50	0	0	531
L2	2 000	2 744	80	0	0	344
L3	2 000	2 779	20	0	0	1 447
L4	2 000	3 512	80	0.05	0	624
L5	2 000	4 471	50	0.05	0	1 671
L6	2 000	4 735	80	0	0.05	626
L7	2 000	5 201	80	0.1	0	951
L8	4 000	5 487	80	0	0	553
L9	4 000	5 500	50	0	0	1 031
L10	2 000	5 571	80	0.05	0.05	646
L11	4 000	5 580	20	0	0	2 584
L12	2 000	6 047	50	0	0.05	1 214
L13	2 000	6 885	80	0	0.1	726
L14	2 000	7 604	80	0.1	0.05	1 225
L15	2 000	7 942	80	0.05	0.1	1 158
L16	2 000	8 071	50	0.1	0	1 929
L17	6 000	8 210	80	0	0	934
L18	2 000	8 214	50	0.05	0.05	1 719
L19	6 000	8 274	50	0	0	1 521
L20	6 000	8 389	20	0	0	4 317
L21	2 000	9 762	20	0.05	0	5 601
L22	2 000	9 815	50	0	0.1	1 601
L23	2 000	9 975	80	0.1	0.1	1 482
L24	4 000	10 416	80	0.05	0	2 111
L25	8 000	11 033	80	0	0	1 284
L26	8 000	11 047	50	0	0	2 078
L27	8 000	11 189	20	0	0	6 056
L28	2 000	11 738	50	0.1	0.05	2 042
L29	2 000	11 764	20	0	0.05	4 996
L30	2 000	11 976	50	0.05	0.1	1 998
L31	10 000	13 688	80	0	0	1 626
L32	10 000	13 790	50	0	0	2 717
L33	10 000	13 970	20	0	0	6 944
L34	4 000	14 170	80	0	0.05	1 812
L35	2 000	15 655	50	0.1	0.1	1 904
L36	4 000	15 836	50	0.05	0	4 382
L37	2 000	19 206	20	0.1	0	5 369
L38	2 000	19 472	20	0.05	0.05	5 653
L39	4 000	19 717	80	0.1	0	2 513
L40	4 000	19 778	50	0	0.05	3 569
L41	4 000	20 259	80	0.05	0.05	2 208
L42	2 000	21 791	20	0	0.1	5 187
L43	6 000	22 842	80	0.05	0	3 679
L44	4 000	23 521	80	0	0.1	2 156
L45	6 000	28 341	80	0	0.05	3 211

Table I.11: Random layered networks' metrics (part 2/3)

NetId	n	m	c	FEP	SEP	Max Flow
L46	2 000	29 347	20	0.05	0.1	5 548
L47	2 000	29 355	20	0.1	0.05	5 301
L48	4 000	29 565	80	0.1	0.05	2 540
L49	4 000	29 718	80	0.05	0.1	2 608
L50	4 000	31 369	50	0.1	0	4 471
L51	4 000	31 397	50	0.05	0.05	3 900
L52	6 000	35 585	50	0.05	0	6 604
L53	4 000	35 681	50	0	0.1	3 852
L54	4 000	38 486	20	0.05	0	10 897
L55	2 000	39 204	20	0.1	0.1	5 216
L56	4 000	39 286	80	0.1	0.1	2 776
L57	8 000	39 536	80	0.05	0	5 410
L58	6 000	41 781	50	0	0.05	5 653
L59	4 000	44 070	20	0	0.05	10 473
L60	6 000	44 362	80	0.1	0	3 879
L61	6 000	44 561	80	0.05	0.05	4 071
L62	4 000	47 303	50	0.1	0.05	4 093
L63	4 000	47 364	50	0.05	0.1	4 412
L64	8 000	47 799	80	0	0.05	4 552
L65	6 000	50 696	80	0	0.1	3 489
L66	10 000	61 992	80	0.05	0	6 935
L67	4 000	63 221	50	0.1	0.1	3 921
L68	8 000	63 341	50	0.05	0	8 390
L69	6 000	66 381	80	0.1	0.05	3 986
L70	6 000	67 032	80	0.05	0.1	3 845
L71	6 000	70 615	50	0.1	0	6 611
L72	6 000	71 282	50	0.05	0.05	6 471
L73	8 000	71 848	50	0	0.05	8 140
L74	10 000	72 006	80	0	0.05	6 052
L75	4 000	76 081	20	0.1	0	10 397
L76	6 000	77 475	50	0	0.1	6 176
L77	4 000	78 217	20	0.05	0.05	11 061
L78	8 000	78 381	80	0.1	0	5 408
L79	8 000	79 199	80	0.05	0.05	5 291
L80	4 000	83 878	20	0	0.1	10 319
L81	6 000	86 729	20	0.05	0	16 498
L82	8 000	87 252	80	0	0.1	4 781
L83	6 000	89 058	80	0.1	0.1	4 013
L84	6 000	95 620	20	0	0.05	15 975
L85	10 000	98 334	50	0.05	0	10 765
L86	6 000	105 648	50	0.1	0.05	6 374
L87	6 000	106 377	50	0.05	0.1	6 048
L88	10 000	109 960	50	0	0.05	10 077
L89	4 000	115 383	20	0.1	0.05	11 558
L90	4 000	118 116	20	0.05	0.1	11 104

Table I.12: Random layered networks' metrics (part 3/3)

NetId	n	m	c	FEP	SEP	Max Flow
L91	8 000	118 857	80	0.1	0.05	5 392
L92	8 000	119 197	80	0.05	0.1	5 032
L93	10 000	123 685	80	0.1	0	6 913
L94	10 000	123 981	80	0.05	0.05	6 486
L95	8 000	126 128	50	0.1	0	8 451
L96	8 000	126 843	50	0.05	0.05	8 984
L97	10 000	134 054	80	0	0.1	6 266
L98	8 000	135 572	50	0	0.1	8 052
L99	6 000	141 586	50	0.1	0.1	6 515
L100	8 000	152 725	20	0.05	0	21 442
L101	4 000	154 967	20	0.1	0.1	10 461
L102	8 000	157 911	80	0.1	0.1	5 474
L103	8 000	167 086	20	0	0.05	21 366
L104	6 000	171 362	20	0.1	0	16 393
L105	6 000	175 259	20	0.05	0.05	16 189
L106	10 000	184 947	80	0.1	0.05	6 722
L107	6 000	185 723	20	0	0.1	15 805
L108	10 000	185 730	80	0.05	0.1	7 177
L109	8 000	189 321	50	0.1	0.05	8 678
L110	8 000	189 462	50	0.05	0.1	7 989
L111	10 000	195 900	50	0.1	0	10 503
L112	10 000	197 378	50	0.05	0.05	10 720
L113	10 000	209 428	50	0	0.1	10 345
L114	10 000	238 852	20	0.05	0	26 218
L115	10 000	246 805	80	0.1	0.1	6 921
L116	8 000	252 538	50	0.1	0.1	9 047
L117	10 000	260 344	20	0	0.05	26 280
L118	6 000	260 843	20	0.1	0.05	16 189
L119	6 000	265 042	20	0.05	0.1	16 148
L120	10 000	295 965	50	0.1	0.05	10 523
L121	10 000	298 026	50	0.05	0.1	11 043
L122	8 000	303 719	20	0.1	0	21 349
L123	8 000	313 043	20	0.05	0.05	22 196
L124	8 000	327 033	20	0	0.1	21 329
L125	6 000	350 229	20	0.1	0.1	17 223
L126	10 000	395 041	50	0.1	0.1	10 928
L127	8 000	464 274	20	0.1	0.05	21 632
L128	8 000	471 737	20	0.05	0.1	22 054
L129	10 000	475 607	20	0.1	0	27 034
L130	10 000	489 210	20	0.05	0.05	27 807
L131	10 000	509 378	20	0	0.1	26 057
L132	8 000	624 351	20	0.1	0.1	21 627
L133	10 000	726 068	20	0.1	0.05	27 498
L134	10 000	735 582	20	0.05	0.1	27 113
L135	10 000	976 411	20	0.1	0.1	27 951

Table I.13: Algorithms' running times (in ms) for random layered networks (part 1/3)

NetId	EK	DinicIt	DinicRec	PRFIFO	PRHL
L1	13.8	4.6	2.6	8.3	14.6
L2	14.2	6.0	4.9	71.9	102.4
L3	17.3	5.8	2.7	48.6	62.1
L4	31.5	9.4	3.8	115.6	178.7
L5	78.6	17.6	4.8	126.7	156.6
L6	39.3	29.1	12.8	19.4	31.2
L7	63.6	9.3	2.4	354.7	384.8
L8	41.5	7.5	5.4	704.5	918.5
L9	49.3	12.7	3.7	647.9	902.7
L10	41.2	19.1	5.6	107.9	147.5
L11	72.6	22.0	5.9	478.8	641.5
L12	78.6	56.2	15.6	230.2	273.5
L13	56.7	47.3	18.5	135.4	179.8
L14	97.1	29.8	4.8	500.2	486.9
L15	97.3	65.4	19.6	376.4	442.7
L16	125.7	6.7	1.8	0.6	5.7
L17	110.5	28.1	11.2	211.3	315.3
L18	103.4	35.0	6.4	556.4	571.1
L19	151.4	53.5	25.6	1 288.5	1 603.8
L20	210.5	77.2	14.8	1 342.1	1 629.1
L21	200.2	7.0	1.2	512.7	565.0
L22	128.6	102.0	25.7	480.7	529.9
L23	134.7	48.6	7.9	1.3	5.1
L24	303.1	30.9	6.5	1 459.8	1 776.4
L25	216.9	78.4	27.3	3 234.0	3 994.3
L26	285.4	79.4	21.8	1 454.1	1 917.4
L27	390.1	113.5	19.6	1 896.5	2 253.4
L28	141.8	11.3	2.3	696.9	685.8
L29	285.8	188.4	25.8	354.2	391.0
L30	155.6	80.7	9.5	708.7	722.2
L31	371.9	128.9	49.2	1 054.6	1 559.9
L32	488.5	158.6	51.3	1 690.5	2 268.2
L33	589.8	205.5	33.8	2 929.8	3 525.5
L34	327.9	234.0	72.1	1 589.2	1 794.0
L35	153.2	11.2	1.9	0.6	4.0
L36	594.0	21.8	4.9	1 819.2	2 119.5
L37	293.7	8.7	0.8	0.4	2.3
L38	279.9	12.3	1.8	0.4	3.3
L39	475.7	24.8	5.7	2 203.9	2 126.3
L40	647.0	432.9	81.1	1 562.8	1 752.5
L41	393.0	78.9	10.0	2 553.9	2 594.1
L42	385.3	264.7	30.3	558.0	645.1
L43	954.0	41.4	8.7	4 159.7	4 701.8
L44	491.3	391.5	90.7	1 779.6	1 977.5
L45	1 049.2	723.8	163.4	2 753.8	3 056.4

Table I.14: Algorithms' running times (in ms) for random layered networks (part 2/3)

NetId	EK	DinicIt	DinicRec	PRFIFO	PRHL
L46	329.2	11.1	2.0	0.7	4.7
L47	349.3	11.4	1.6	1 359.7	1 369.0
L48	560.7	31.4	6.6	3 358.9	3 301.1
L49	548.3	190.5	15.7	2.6	30.8
L50	834.6	27.2	4.9	2 355.2	2 452.0
L51	671.1	35.9	6.8	3 515.7	3 503.3
L52	1 716.6	39.1	6.6	1.5	21.5
L53	881.7	627.9	79.6	2 744.0	2 897.1
L54	1 171.8	22.1	3.1	2 679.4	2 759.5
L55	411.3	16.9	1.9	1 782.2	1 749.5
L56	724.1	39.5	5.5	1.3	11.7
L57	2 203.0	63.8	13.9	8 471.2	9 084.8
L58	1 843.4	1 319.9	131.9	6 282.0	6 571.0
L59	1 633.3	999.4	52.4	1 326.7	1 474.2
L60	1 440.6	47.4	7.9	1.2	33.7
L61	1 366.7	74.4	10.3	2.7	19.2
L62	932.5	33.8	5.0	4 822.7	4 623.5
L63	922.6	50.0	7.3	4 605.9	4 469.4
L64	2 300.9	1 500.3	262.4	3 128.8	3 381.9
L65	1 459.7	1 034.5	179.4	6 102.4	6 326.3
L66	4 313.3	96.6	17.0	1.74E+04	1.70E+04
L67	1 283.6	43.6	6.6	7 856.4	7 143.2
L68	3 496.7	57.2	9.1	1.4	26.3
L69	1 850.9	64.0	10.2	1.06E+04	1.07E+04
L70	1 640.4	84.7	9.0	2.7	33.2
L71	2 732.4	57.5	8.4	7 724.8	7 062.9
L72	2 657.0	62.1	8.9	1.24E+04	1.17E+04
L73	4 445.7	2 574.4	251.8	6 924.7	7 674.8
L74	4 432.9	2 745.8	383.7	1.52E+04	1.69E+04
L75	2 171.2	32.6	2.9	0.8	7.3
L76	2 896.2	1 815.8	197.6	2 813.2	3 066.1
L77	2 239.2	34.7	5.5	1.0	9.1
L78	3 433.8	79.9	11.1	1.5	34.5
L79	3 342.4	94.4	12.7	2.9	36.5
L80	2 628.3	1 183.2	60.1	7 293.9	6 887.0
L81	4 746.7	50.8	7.0	9 778.0	8 659.1
L82	3 784.4	2 390.2	329.6	1.51E+04	1.49E+04
L83	2 605.4	74.2	11.2	1.55E+04	1.32E+04
L84	6 117.5	2 749.2	127.2	7 860.0	8 254.0
L85	8 462.8	98.1	12.6	2.7	60.1
L86	4 565.6	67.2	8.7	1.6	33.2
L87	3 978.9	77.4	15.3	2.16E+04	1.74E+04
L88	9 915.6	4 942.0	364.1	3 882.0	4 543.6
L89	4 246.7	51.5	7.3	1.37E+04	1.31E+04
L90	3 894.8	48.3	7.6	1.35E+04	1.26E+04

Table I.15: Algorithms' running times (in ms) for random layered networks (part 3/3)

NetId	EK	DinicIt	DinicRec	PRFIFO	PRHL
L91	6 359.9	117.9	17.7	3.60E+04	2.78E+04
L92	5 516.8	131.5	23.7	3.25E+04	2.77E+04
L93	1.01E+04	154.9	22.3	2.77E+04	2.93E+04
L94	8 662.6	145.0	29.0	5.12E+04	4.23E+04
L95	8 649.3	104.8	19.3	2.21E+04	2.05E+04
L96	8 994.9	109.7	17.7	4.00E+04	3.43E+04
L97	9 221.1	4 882.8	570.4	1.99E+04	2.10E+04
L98	9 481.0	4 393.8	332.3	1 515.5	1 643.7
L99	6 482.0	117.6	8.7	2.1	35.3
L100	1.42E+04	81.6	7.7	2.5	23.0
L101	5 864.2	65.4	10.1	2.11E+04	1.81E+04
L102	9 513.1	154.1	14.4	3.2	51.1
L103	1.93E+04	7 371.1	210.9	1.03E+04	1.13E+04
L104	1.20E+04	97.7	12.1	2.20E+04	1.93E+04
L105	1.09E+04	93.4	15.6	3.89E+04	3.49E+04
L106	1.43E+04	214.5	26.7	7.83E+04	6.35E+04
L107	1.58E+04	5 339.9	205.8	3.45E+04	2.27E+04
L108	1.50E+04	228.2	36.8	7.08E+04	6.01E+04
L109	1.41E+04	143.5	12.5	3.1	48.1
L110	1.25E+04	113.4	13.9	3.6	37.4
L111	1.90E+04	182.6	14.2	3.1	76.8
L112	1.92E+04	181.2	24.5	8.17E+04	7.14E+04
L113	2.12E+04	1.00E+04	550.9	1.39E+04	1.52E+04
L114	2.87E+04	175.6	20.0	5.32E+04	5.12E+04
L115	1.91E+04	240.3	19.0	4.0	97.8
L116	1.94E+04	178.8	13.1	3.7	61.9
L117	3.87E+04	1.58E+04	283.2	9.70E+04	7.30E+04
L118	1.71E+04	130.5	16.4	5.93E+04	5.26E+04
L119	1.72E+04	120.0	17.3	5.37E+04	4.97E+04
L120	2.55E+04	219.8	15.8	3.8	67.4
L121	2.82E+04	216.9	17.3	5.0	84.3
L122	2.68E+04	193.7	20.7	5.06E+04	7.96E+04
L123	2.81E+04	137.4	10.6	4.2	37.3
L124	3.64E+04	1.52E+04	279.1	4 685.0	5 541.5
L125	2.45E+04	198.2	20.8	6.86E+04	6.62E+04
L126	3.69E+04	308.2	33.5	1.59E+05	1.27E+05
L127	4.34E+04	259.2	11.9	4.6	60.1
L128	4.07E+04	197.7	13.0	5.0	48.7
L129	5.63E+04	340.5	13.8	9.9	47.1
L130	5.86E+04	276.9	14.7	6.1	65.1
L131	6.98E+04	2.78E+04	462.9	3.44E+04	2.41E+04
L132	5.07E+04	386.1	37.3	1.73E+05	1.53E+05
L133	8.35E+04	469.1	17.6	6.4	90.2
L134	7.96E+04	445.5	52.9	2.69E+05	2.29E+05
L135	1.11E+05	597.6	20.5	7.9	83.3

Table I.16: Other algorithms' metrics for random layered networks (part 1/3)

NetId	PRFIFO R	PRFIFO P	PRHL R	PRHL P	# APs	AP Av. Length
L1	1.23E+05	1.67E+05	1.43E+05	1.78E+05	191	55.2
L2	8.58E+05	1.39E+06	8.88E+05	1.38E+06	169	88.6
L3	6.25E+05	9.20E+05	7.15E+05	8.97E+05	270	23.8
L4	1.35E+06	1.95E+06	1.21E+06	2.00E+06	365	84.8
L5	9.43E+05	1.73E+06	1.06E+06	1.68E+06	770	54.2
L6	1.95E+05	2.95E+05	2.12E+05	3.28E+05	390	102.5
L7	2.71E+06	4.21E+06	3.02E+06	4.02E+06	595	81.6
L8	7.39E+06	1.18E+07	7.81E+06	1.16E+07	280	83.4
L9	6.38E+06	1.07E+07	7.15E+06	1.04E+07	329	52.8
L10	9.20E+05	1.32E+06	9.69E+05	1.21E+06	379	86.7
L11	4.90E+06	7.69E+06	5.49E+06	7.51E+06	489	23.1
L12	1.76E+06	2.45E+06	1.77E+06	2.35E+06	656	72.5
L13	1.01E+06	1.40E+06	1.04E+06	1.36E+06	479	108.9
L14	3.51E+06	4.25E+06	3.50E+06	4.04E+06	759	82.4
L15	2.58E+06	3.27E+06	2.59E+06	3.16E+06	782	100.5
L16	3 465.0	5 201.0	2.83E+04	3.97E+04	962	51.0
L17	2.67E+06	4.08E+06	2.99E+06	4.02E+06	469	85.5
L18	3.50E+06	4.41E+06	3.49E+06	4.07E+06	792	53.6
L19	1.33E+07	2.08E+07	1.53E+07	2.05E+07	599	59.2
L20	1.32E+07	2.04E+07	1.49E+07	2.01E+07	885	25.5
L21	2.08E+06	5.06E+06	2.26E+06	4.10E+06	1 407	21.0
L22	2.90E+06	3.68E+06	2.88E+06	3.48E+06	937	78.7
L23	8 533.0	1.20E+04	2.59E+04	3.49E+04	953	84.2
L24	8.73E+06	1.71E+07	1.28E+07	1.58E+07	1 323	81.4
L25	3.32E+07	5.28E+07	3.68E+07	5.26E+07	671	87.5
L26	1.39E+07	2.24E+07	1.59E+07	2.19E+07	855	56.4
L27	1.75E+07	2.76E+07	2.05E+07	2.69E+07	1 186	24.0
L28	3.73E+06	4.51E+06	3.72E+06	4.09E+06	970	51.0
L29	1.90E+06	2.46E+06	1.90E+06	2.25E+06	1 830	39.4
L30	3.64E+06	4.55E+06	3.65E+06	4.08E+06	968	55.8
L31	9.95E+06	1.57E+07	1.15E+07	1.59E+07	868	92.1
L32	1.64E+07	2.58E+07	1.91E+07	2.54E+07	1 067	60.4
L33	2.61E+07	4.11E+07	3.02E+07	3.95E+07	1 383	25.1
L34	1.08E+07	1.37E+07	1.08E+07	1.34E+07	1 279	122.9
L35	2 844.0	4 368.0	1.70E+04	2.16E+04	942	51.1
L36	8.51E+06	1.83E+07	1.25E+07	1.61E+07	2 179	51.0
L37	1 555.0	3 461.0	7.85E+03	1.15E+04	1 461	21.0
L38	2 152.0	4 010.0	1.20E+04	1.69E+04	1 433	21.1
L39	9.02E+06	2.02E+07	9.79E+06	1.67E+07	1 690	81.0
L40	9.12E+06	1.07E+07	9.11E+06	1.07E+07	2 175	86.9
L41	1.45E+07	1.76E+07	1.45E+07	1.61E+07	1 350	81.5
L42	2.06E+06	2.57E+06	2.06E+06	2.66E+06	1 781	34.7
L43	2.17E+07	3.97E+07	2.84E+07	3.63E+07	2 386	81.0
L44	9.38E+06	1.16E+07	9.37E+06	1.08E+07	1 618	124.1
L45	1.61E+07	1.99E+07	1.60E+07	1.95E+07	2 391	132.0

Table I.17: Other algorithms' metrics for random layered networks (part 2/3)

NetId	PRFIFO R	PRFIFO P	PRHL R	PRHL P	# APs	AP Av. Length
L46	1 905.0	3 766.0	1.10E+04	1.53E+04	1 391	21.0
L47	3.87E+06	4.85E+06	3.88E+06	4.30E+06	1 430	21.0
L48	1.53E+07	1.76E+07	1.52E+07	1.63E+07	1 693	81.0
L49	1.17E+04	1.64E+04	1.01E+05	1.22E+05	1 626	82.9
L50	8.60E+06	1.77E+07	8.74E+06	1.72E+07	2 383	51.0
L51	1.52E+07	1.78E+07	1.52E+07	1.63E+07	1 941	51.1
L52	7 352.0	1.27E+04	5.18E+04	7.03E+04	3 403	51.0
L53	1.15E+07	1.36E+07	1.15E+07	1.34E+07	2 365	80.9
L54	8.23E+06	1.79E+07	8.25E+06	1.70E+07	2 896	21.0
L55	3.93E+06	4.86E+06	3.93E+06	4.28E+06	1 398	21.0
L56	5 539.0	8 939.0	4.35E+04	5.38E+04	1 880	81.0
L57	3.56E+07	7.48E+07	4.39E+07	6.59E+07	3 646	81.0
L58	2.85E+07	3.51E+07	2.84E+07	3.24E+07	3 519	83.6
L59	4.50E+06	6.00E+06	4.50E+06	5.60E+06	3 687	35.1
L60	5 642.0	1.02E+04	1.23E+05	1.49E+05	2 672	81.0
L61	1.21E+04	1.80E+04	7.91E+04	9.84E+04	2 630	81.1
L62	1.55E+07	1.93E+07	1.54E+07	1.63E+07	2 109	51.0
L63	1.55E+07	1.64E+07	1.55E+07	1.63E+07	2 155	51.1
L64	1.62E+07	2.10E+07	1.63E+07	1.87E+07	3 462	132.0
L65	2.56E+07	3.12E+07	2.55E+07	2.98E+07	2 655	124.7
L66	5.30E+07	1.37E+08	6.12E+07	1.03E+08	4 700	81.0
L67	1.57E+07	1.98E+07	1.57E+07	1.64E+07	2 034	51.0
L68	6 870.0	1.30E+04	8.61E+04	1.10E+05	4 337	51.0
L69	3.49E+07	4.13E+07	3.47E+07	3.67E+07	2 721	81.0
L70	1.04E+04	1.56E+04	1.13E+05	1.34E+05	2 507	81.1
L71	1.84E+07	4.40E+07	1.91E+07	3.79E+07	3 597	51.0
L72	3.51E+07	4.22E+07	3.50E+07	3.65E+07	3 310	51.0
L73	2.71E+07	3.30E+07	2.70E+07	3.15E+07	5 106	85.0
L74	6.77E+07	8.22E+07	6.75E+07	7.61E+07	4 577	130.4
L75	2 012.0	5 726.0	1.63E+04	2.25E+04	2 917	21.0
L76	8.87E+06	1.12E+07	8.87E+06	1.15E+07	3 850	80.9
L77	2 764.0	6 505.0	2.07E+04	2.78E+04	2 979	21.0
L78	6 155.0	1.25E+04	1.05E+05	1.29E+05	3 781	81.0
L79	1.12E+04	1.78E+04	1.20E+05	1.44E+05	3 453	81.0
L80	1.47E+07	1.77E+07	1.47E+07	1.72E+07	3 433	32.2
L81	1.81E+07	4.66E+07	1.85E+07	3.78E+07	4 482	21.0
L82	5.09E+07	6.18E+07	5.09E+07	5.46E+07	3 624	124.0
L83	3.53E+07	3.87E+07	3.53E+07	3.69E+07	2 777	81.0
L84	2.06E+07	2.56E+07	2.06E+07	2.42E+07	5 559	34.2
L85	7 734.0	1.62E+04	1.35E+05	1.71E+05	5 712	51.0
L86	3 375.0	8 257.0	6.00E+04	7.17E+04	3 374	51.0
L87	3.55E+07	4.26E+07	3.54E+07	3.68E+07	3 046	51.0
L88	1.32E+07	1.68E+07	1.32E+07	1.55E+07	6 214	81.8
L89	1.56E+07	1.86E+07	1.56E+07	1.71E+07	3 229	21.0
L90	1.59E+07	1.92E+07	1.59E+07	1.73E+07	3 023	21.0

Table I.18: Other algorithms' metrics for random layered networks (part 3/3)

NetId	PRFIFO R	PRFIFO P	PRHL R	PRHL P	# APs	AP Av. Length
L91	6.21E+07	6.84E+07	6.21E+07	6.44E+07	3 744	81.0
L92	6.29E+07	6.96E+07	6.27E+07	6.45E+07	3 349	81.0
L93	5.07E+07	1.24E+08	5.61E+07	1.05E+08	4 897	81.0
L94	9.76E+07	1.11E+08	9.74E+07	1.00E+08	4 507	81.0
L95	3.24E+07	8.20E+07	3.47E+07	6.68E+07	4 549	51.0
L96	6.28E+07	6.70E+07	6.28E+07	6.55E+07	4 719	51.0
L97	5.69E+07	6.74E+07	5.69E+07	6.38E+07	4 713	124.5
L98	3.80E+06	5.77E+06	3.82E+06	5.17E+06	4 940	78.4
L99	3 276.0	8 349.0	4.82E+04	5.72E+04	3 438	51.0
L100	4 293.0	1.22E+04	3.28E+04	4.68E+04	5 967	21.0
L101	1.58E+07	2.13E+07	1.58E+07	1.68E+07	2 867	21.0
L102	4 988.0	1.15E+04	9.01E+04	1.06E+05	3 840	81.0
L103	2.16E+07	2.91E+07	2.17E+07	2.52E+07	7 248	33.2
L104	1.81E+07	6.28E+07	1.84E+07	3.71E+07	4 591	21.0
L105	3.56E+07	4.49E+07	3.55E+07	3.71E+07	4 407	21.0
L106	9.76E+07	1.16E+08	9.73E+07	1.02E+08	4 749	81.0
L107	3.33E+07	3.97E+07	3.34E+07	3.84E+07	5 525	32.8
L108	9.88E+07	1.09E+08	9.86E+07	1.02E+08	4 914	81.0
L109	3 909.0	1.08E+04	6.71E+04	7.96E+04	4 726	51.0
L110	5 235.0	1.13E+04	5.28E+04	6.44E+04	4 145	51.0
L111	4 794.0	1.35E+04	9.41E+04	1.17E+05	5 753	51.0
L112	9.87E+07	1.15E+08	9.85E+07	1.02E+08	5 683	51.0
L113	3.10E+07	3.61E+07	3.10E+07	3.40E+07	6 305	78.8
L114	5.02E+07	1.57E+08	5.22E+07	1.01E+08	7 353	21.0
L115	5 542.0	1.40E+04	1.20E+05	1.36E+05	4 818	81.0
L116	4 166.0	1.16E+04	7.26E+04	8.53E+04	4 962	51.0
L117	9.22E+07	1.18E+08	9.24E+07	1.05E+08	8 803	32.4
L118	3.54E+07	4.70E+07	3.53E+07	3.68E+07	4 537	21.0
L119	3.57E+07	4.01E+07	3.57E+07	3.78E+07	4 385	21.0
L120	4 364.0	1.28E+04	6.69E+04	7.92E+04	5 658	51.0
L121	6 253.0	1.49E+04	9.13E+04	1.09E+05	5 823	51.0
L122	3.21E+07	1.21E+08	5.68E+07	6.38E+07	6 077	21.0
L123	4 342.0	1.23E+04	3.49E+04	4.66E+04	6 191	21.0
L124	6.34E+06	9.45E+06	6.34E+06	1.02E+07	7 255	32.5
L125	3.55E+07	3.74E+07	3.56E+07	3.68E+07	4 787	21.0
L126	9.91E+07	1.18E+08	9.88E+07	1.04E+08	6 061	51.0
L127	3 502.0	1.13E+04	3.35E+04	4.51E+04	6 122	21.0
L128	4 038.0	1.22E+04	3.02E+04	4.14E+04	6 138	21.0
L129	3 865.0	1.46E+04	2.80E+04	4.10E+04	7 788	21.0
L130	5 093.0	1.54E+04	4.46E+04	5.86E+04	7 792	21.0
L131	2.46E+07	3.56E+07	2.47E+07	3.58E+07	8 943	32.3
L132	6.32E+07	8.33E+07	6.33E+07	6.99E+07	6 186	21.0
L133	3 808.0	1.41E+04	3.97E+04	5.42E+04	7 866	21.0
L134	9.92E+07	1.27E+08	9.93E+07	1.09E+08	7 553	21.0
L135	3 883.0	1.52E+04	3.23E+04	4.53E+04	8 036	21.0