
UCL

**Université
catholique
de Louvain**

UNIVERSITE CATHOLIQUE DE LOUVAIN

ECOLE POLYTECHNIQUE DE LOUVAIN



Study and Analysis of Networks Flows

Supervisor: YVES DEVILLE
Readers: FRANÇOIS AUBRY
JEAN-CHARLES DELVENNE

Thesis submitted for the Master's degree
in computer science and engineering
options: Artificial Intelligence
by DENIS GENON & VICTOR VELGHE

Louvain-la-Neuve
Academic year 2015-2016

Abstract

Acknowledgment

Contents

1	The Maximum Flow Problem	9
2	Existing Algorithms	11
2.1	Augmenting path algorithms	11
2.1.1	Introduction	11
2.1.2	Ford-Fulkerson and Edmonds-Karp	12
2.1.3	Complexities	12
2.2	Pre-flow algorithms	13
3	Data Structures	15
3.1	Introduction	15
3.2	Data structures	15
3.2.1	Hash Map	15
3.2.2	Tree Map	16
3.2.3	Simple Linked List	16
3.2.4	Split Array	16
3.2.5	Complexities	16
4	Improvements of Existing Algorithms	17
5	Implementation	19
6	Experimental Analysis	21
7	Conclusion	23
	Bibliography	23

Chapter 1

The Maximum Flow Problem

Chapter 2

Existing Algorithms

2.1 Augmenting path algorithms

2.1.1 Introduction

The idea behind the augmenting path algorithms is as follows : As long as there is a path from the source to the sink, we send flow along this path. And so on until there is no more path from the source to the sink.

An available path from the source to the sink is called *augmenting path* and to find it, we use the *residual graph*. A *residual graph* is a double oriented graph with the available capacities. For instance here is a graph with its *residual graph*:

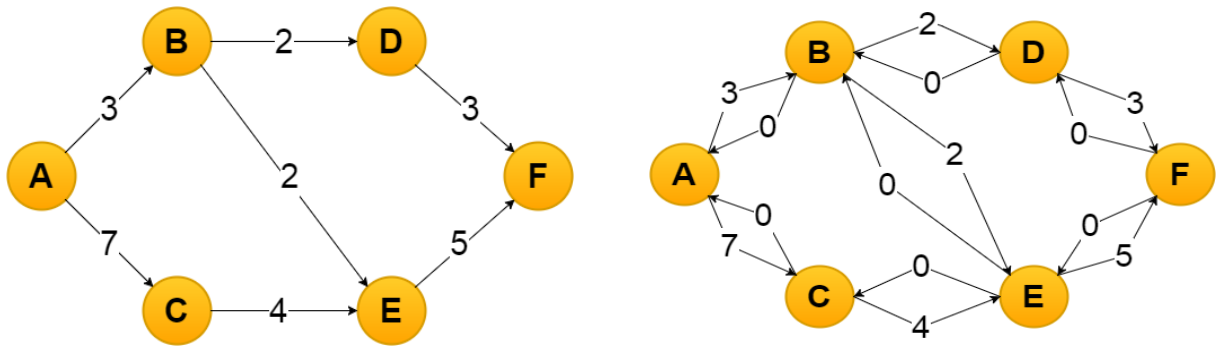


Figure 2.1: A graph with its residual graph

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

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

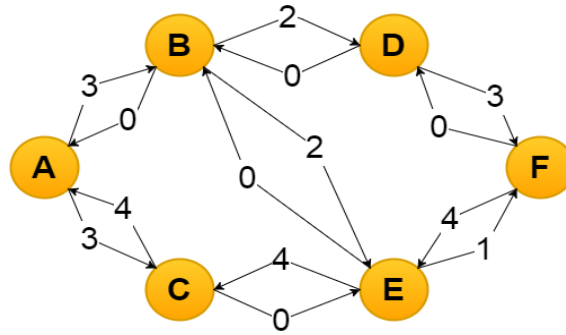


Figure 2.2: Residual graph

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

```

Look for an augmenting path;
while There is an augmenting path do
    | Send flow through this path;
    | Update the residual graph;
    | Look for an new augmenting path;
end

```

2.1.2 Ford-Fulkerson and Edmonds-Karp

There are two main augmenting path algorithms, Ford-Fulkerson (published in 1956) and Edmonds-Karp (published in 1972). The second being a variant of the first one. Indeed, the unique difference between both is the way of looking for an *augmenting path* in the *residual graph*.

Ford-Fulkerson uses a depth-first search while Edmonds-Karp uses a breadth-first search.

2.1.3 Complexities

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

Ford-Fulkerson is in $O(E * f)$ because each augmenting path can be found in $O(E)$ and in the worst case, the flow will increase by 1.

Edmonds-Karp is in $O(V * E^2)$ because the breadth-first search assures us that after each iteration, the length of the augmenting path can't decrease. Furthermore, the length of the augmenting path can stay the same for at most E iterations before increasing. We also know that the length of the augmenting path is between 1 and $V - 1$. Thus there are at most $V * E$ iterations and each augmenting path can be found in $O(E)$.

2.2 Pre-flow algorithms

Chapter 3

Data Structures

3.1 Introduction

For this thesis, we wanted to be able to analyse the difference between several data structures. Moreover, willing to work with big graph, the traditional adjacency matrix became too heavy. So we decide to use a structure defined below.

Like for the adjacency matrix, we use a array where each row represents the neighbours of a node. For example, the first row contains the information on the neighbours of the node 0. But contrary to the adjacency matrix, we do not use a array to represent neighbours but a different structure requiring less memory space.

We used four various data structures : Hash Map, Tree Map, Simple Linked List and one home-made structure, Split Array. Each node will have its structure, storing which nodes are neighbours and what are the capacities of the edges of these nodes. If we had to represent a graph with 10 nodes, we would have a array of 10, for example, Hash Map. Every Hash Map representing the neighbourhood of a single node.

3.2 Data structures

3.2.1 Hash Map

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

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

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

TODO Je met un exemple de hash map?

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

3.2.2 Tree Map

3.2.3 Simple Linked List

3.2.4 Split Array

3.2.5 Complexities

Chapter 4

Improvements of Existing Algorithms

Chapter 5

Implementation

Chapter 6

Experimental Analysis

Chapter 7

Conclusion

List of Figures

2.1	A graph with its residual graph	11
2.2	Residual graph	12

List of Tables