SAPIENZA
UNIVERSITÀ DI ROMA

Achieving Max Flow in Strongly Polynomial Time for Sparse Networks: Beyond the Edmonds-Karp Algorithm

Faculty of Information Engineering, Computer Science and Statistics
Bachelor's Degree in Computer Science

Armando Coppola
ID number 2003964

Advisor
Prof. Paul Joseph Wollan

Academic Year 2023/2024

Achieving Max Flow in Strongly Polynomial Time for Sparse Networks:Beyond the Edmonds-Karp Algorithm

Bachelor's Thesis. Sapienza University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: ArmandoCoppola24@gmail.com

«Computer Science is no more about computers
than astronomy is about telescopes»
— Edsger W. Dijkstra

Abstract

The following report is aimed at analyzing the evolution of solutions to the max-flow problem in networks. It will clearly and thoroughly demonstrate how improvements can be made upon the Edmonds-Karp algorithm.

Finding the maximum flow that can be routed through a network posed a significant challenge for decades, since the first solution was introduced by L. R. Ford Jr. and D. R. Fulkerson in 1956. Their algorithm runs in $O(mf)$ time were f represents the maximum flow in the network.

Given the rapid growth and dynamic nature of connections (especially virtual ones), determining the maximum flow of a network is a crucial problem. If solved efficiently, it allows for traffic optimization within the network, minimizing both waste and slowdowns.

For example, if we consider the internet, it consists of numerous connected devices that continuously receive and forward packets to other devices. Naturally, each device has different performance capabilities and can handle varying amounts of data flow. By identifying the maximum flow capacity, we can determine how fast information can travel from one point to another within the network. This allows us to maximize resource utilization without exceeding limits, which could otherwise cause bottlenecks.

Furthermore, since network topologies are highly dynamic, with many nodes and branches, quickly identifying the maximum routable data flow becomes crucial to ensure that information transfer is correct and efficient.

This thesis will focus on the $O(nm)$ solution for sparse graphs. Computational cost becomes particularly challenging when there are few edges, as the costs of other phases of the solution tend to increase.

An algorithm with $O(nm)$ complexity has existed since 1992 the King-Goldberg-Tarjan algorithm by V. King, S. Rao, and R. Tarjan (which is not covered in this thesis). However, this solution only achieves the declared complexity on relatively dense graphs. By contrast, Orlin's 2013 solution achieves $O(nm)$ complexity specifically for sparse graphs.

In conclusion, we will show that it is possible to find the maximum flow for any type of graph. By reconstructing all intermediate algorithms, from Edmonds-Karp to the most recent ones, even a less experienced reader will be able to understand the final solution while learning all the necessary concepts.

The algorithms are presented in chronological order, starting with Dinic's algorithm, continuing with Goldberg-Rao, and culminating with Orlin's most recent and effective solution. However, before delving into these algorithms, a chapter is dedicated to explaining some preliminary graph theory concepts that are essential for understanding how the solutions work.

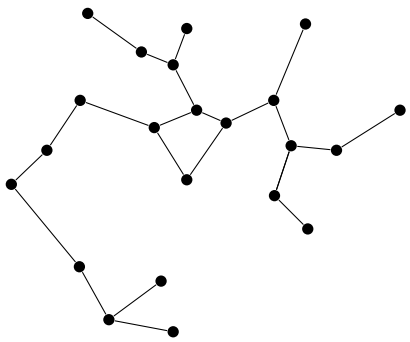
Contents

Chapter 1

Introduction and preliminaries

Graph theory is the branch of computer science which focuses on the study of graphs. More broadly, it can be seen as the study of connections, since a graph represents the most general form of 'objects' and how they can be linked together.

The objects are referred to as nodes and are represented by points. These points are connected by edges, which are depicted as lines.



Definition 1.1 (Graph)**.** A graph $G = (N, E)$ is a structure composed of a set of nodes N and a set of edges E that connect two nodes in N

$$(i, j) \in E \implies i, j \in N$$

The set of nodes in a graph G is denoted by $N(G)$, and similarly, the set of edges is denoted by $E(G)$. Often, the following labels will be used to indicate the cardinality of these sets:

$$\begin{aligned} n &= |N(G)| \\ m &= |E(G)| \end{aligned}$$

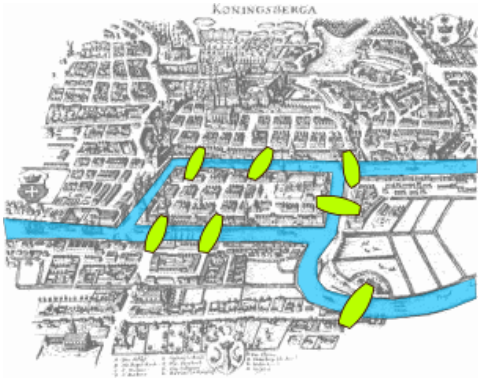
Various types of graphs exist in theory, just as various types of connections exist in the real world. An edge can be:

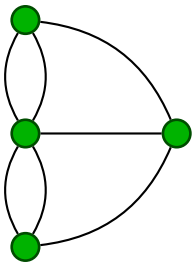
- **directed**, with an assigned direction to each connection
- **undirected**, indicating that each connection can be traversed in both directions.

There are also **simple graphs**, where no more than one edge connects the same two nodes and no edges start and end at the same node, and **multigraphs**, where such parallel edges are allowed.

Sometimes it is easy to see how something can be represented as a graph, other times the representation is more abstract.

Graph theory originated in 1736 with the Swiss mathematician Leonhard Euler. Euler wanted to know if it was possible to cross all the bridges of the city of Königsberg in a single walk, without crossing the same bridge twice.





Definition 1.2. A **walk** is an ordered sequence of edges where each edge starts from the node where the previous one ends (except for the first edge, of course).

A **path** is a walk in which no node is visited more than once.

$$P := \{(n_0, n_1), (n_1, n_2), ..., (n_{k-1}, n_k)\} \implies \nexists (a, b), (c, d) \in P | a = c \vee b = d$$

A **circuit** is path that begins and ends at the same vertex.

In this case, the bridges can be represented as a multigraph where two distinct edges connect the same nodes.

Definition 1.3 (Degree of a node)**.** The degree of a node refers to the number of edges incident to that node.

In directed graphs, we can also distinguish between the out-degree and in-degree of a node, depending on the direction of the edges adjacent to it.

In the end, Euler concluded that, for a graph to have an Eulerian path, it must have zero or two nodes with an odd degree, while for an Eulerian circuit, all nodes must have an even degree. With this, he demonstrated that the bridges of Königsberg do not have an Eulerian path, paving the way for a deeper study of graphs.

1.1 Networks and flows

Nowadays, connections are expanding at an ever-increasing pace, forming networks that are becoming larger and more complex, whether physical or virtual. For this reason, from this point forward, we will focus on a specific type of graph: the network. We will analyze its properties and challenges in order to address various problems as efficiently as possible.

Definition 1.4 (Network)**.** In graph theory a network is a structure composed of a graph $G = (N, E)$ and a function $u : E \to \mathbb{N}^+ \cup \{+\infty\}$ which denotes the capacity of each edge.

$$u(i, j) = \text{capacity of the edge } (i, j)$$

we will denote the capacity $u(i, j)$ *below with the abbreviation* u_{ij}

When we refer to the capacity of an edge, we are talking about the maximum amount of flow that can pass through it.

Observation 1.1 (Integer capacities)**.** *The condition that the capacities are integers doesn't restrict the problem since we can always scale.*

If the capacities $U := \{u_1, u_2, ..., u_k\} \wedge U \subseteq \mathbb{Q}$ *then we can multiply all the capacities by* $\gcd(u_1, u_2, ..., u_k)$ *so all the values are now integers.*

For real numbers, we can use a rational approximation so the solution is the same.

In real-world applications, *flow* can represent different things, such as the transfer of data in a network, the flow of water through a pipe, or even traffic flow on a road. The concept is versatile and applies to a wide range of scenarios depending on the context.

In each network there exists two special nodes, s *the source* and t *the sink*. The network aims to send a certain **flow** from the source to the sink. The edges in a network are directed, and except for s and t, for all (i, j) in a network G, $(j, i) \in E(G)$. We assume that the source s has no incoming edges, just as the sink t has no outgoing edges. When representing two nodes in a network where the flow can only be routed in one direction, we can set the capacity of the flow in the opposite direction to zero. Note that $u(i, j)$ refers specifically to the capacity of the edge from i to j, and the capacity from j to i may be different.

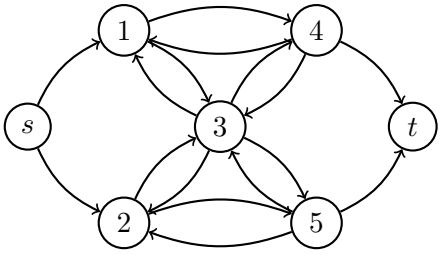


Figure 1.1. A classic example of a network

Definition 1.5 ($U_{min},\ U_{max}$)**.** In each network we define:

- U_{min}: the smallest non zero capacity associated to an edge:

$$U_{min} = u_{ij} | (i,j) \in E \ \wedge \ u_{ij} > 0 \ \wedge \ \nexists (k,l) \in E : 0 < u_{kl} < u_{ij}$$

- U_{max}: the largest finite capacity

$$U_{max} = u_{ij} | (i,j) \in E \wedge u_{ij} \neq +\infty \wedge$$
$$\nexists (k,l) \in E : u_{ij} < u_{jl} < +\infty$$

Moreover, we divide the edges into two categories:
External Arcs $:= \{(x,y) | (x,y) \in E \wedge (x = s \vee y = t)\}$
Internal Arcs $:= \{(x,y) | (x,y) \in E \wedge x \neq s \wedge y \neq t\}$ i.e. $E \setminus External\ edges$

For simplicity, we assume that for each internal edge $(i,j) \in E$ there exists the edge $(j,i) \in E$.
The same is true for any internal node for which there always exists an edge that links it with s and t, even if it has zero capacity.

$$\forall i \in N \implies \{(s,i),(i,t)\} \subseteq E$$

We can now give a formal definition of flow and the properties which it must respect.

Definition 1.6 (Flow)**.** We define the flow as the function $f : E \to \mathbb{R}_+ \cup \{0\}$ which associates to each edge the amount of flow that is routed through ti. A flow function must satisfy the **flow conservation rule**:

$$\sum_{j:(i,j)\in E} f_{ij} - \sum_{j:(j,i)\in E} f_{ji} = 0 \quad \forall i \in N \setminus \{s,t\}$$

that ensures that there is no loss of flow as there are no unexpected entries.

We call a flow *feasible* if it respects the **capacity constraint**:

$$\forall (i,j) \in E,\ f_{ij} \leq u_{ij}$$

Definition 1.7 (Flow value)**.** The **value of a flow** is given by the sum of all the outgoing edges of s (or by the sum of all the incoming edges of t; it is the same)

$$val(f) = \sum_{\forall (s,x)\in E(G)} f_{sx} = \sum_{\forall (y,t)\in E(G)} f_{yt}$$

Definition 1.8 (Residual capacity)**.** The residual capacity of an edge (i,j) means the amount of flow which can be routed through this edge before we saturate it.

$$r_{ij} = u_{ij} + f_{ji} - f_{ij}$$

When we talk about residual capacity according to different flows we could also use the notation:

$$u_f(i,j)$$

that means the residual capacity of the edge (i,j) which has routed the flow f.

We will often talk later about the residual function or the array of residual capacities, in fact we are referring to any function or structure that associates each arc with its residual capacity.

Definition 1.9 (Residual Graph)**.** Given a network G and a flow f, we can define a residual graph as follows

$$G[r] := (N(\mathcal{N}), \{(i,j)|(i,j) \in E(\mathcal{N}) \wedge r_{ij} > 0\})$$

The notation $G[r]$ refers to a graph designed from the residual capacity function r. We will refer to the residual graph also using the notation G_f that underlines the representation of the original network under the effect of the routed flow f

Definition 1.10 (s-t Cut)**.** Given a network G we define an *s-t* cut on G as a partition into two subsets (S,T) such that:

1. $s \in S$
2. $t \in T$
3. $S \cap T = \varnothing$
4. $S \cup T = N$

The *cutting capacity* is defined as:

$$u(S,T) = \sum_{i \in S \wedge j \in T} u_{ij}$$

the *residual of a cut* is defined as:

$$r(S,T) = \sum_{i \in S \wedge j \in T} r_{ij}$$

Lemma 1.1 (Max residual flow min residual cut)**.** *Given a residual graph $G[r]$ and a cut (S,T) then $r(S,T)$ represents the upper bound of the flow from $s \to t$. In particular, the maximum increase in flow with respect to r is the smallest residual capacity of an s-t cut.*

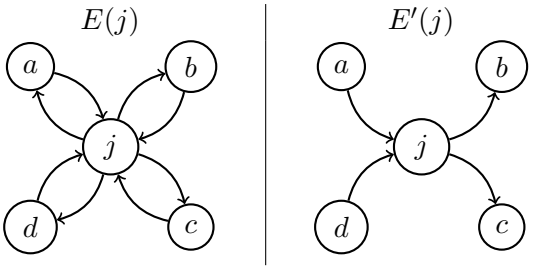
Proof. Omitted.

The lemma states that the problem of finding the maximum flow on a network is **dual** to that of finding a minimum capacity cut on the same network since this will represent the bottleneck that acts as an upper bound to the increase in flow.

Definition 1.11. (Anti-symmetric subset) Let $E(j)$ be de set of edges incident to a node j, we define the *Anti-symmetric* subset of j as:

$$E'(j) := \{(x,y)|(x,y) \in E(j) \wedge (x,y) \in E'(j) \iff (y,x) \notin E'(j)\}$$

Example:



Lemma 1.2 (Anti-symmetry lemma)**.** *Given $E'(j)$ an anti-symmetric subset of $E(j)$ and a flow f on G with $r = r[f]$ then it holds that:*

$$\sum_{(i,j)\in E'(j)} r_{ij} - \sum_{(j,i)\in E'(j)} r_{ji} = \sum_{(i,j)\in E'(j)} u_{ij} - \sum_{(j,i)\in E'(j)} u_{ji}$$

Proof.

$$\sum_{(i,j)\in E'(j)} r_{ij} - \sum_{(j,i)\in E'(j)} r_{ji} - \sum_{(i,j)\in E'(j)} u_{ij} + \sum_{(j,i)\in E'(j)} u_{ji} = 0 \implies$$

$$\sum_{(i,j)\in E'(j)} (u_{ij} - r_{ij}) + \sum_{(j,i)\in E'(j)} (u_{ji} - r_{ji}) = 0$$

since $r_{ij} = u_{ij} - f_{ji} + f_{ij} \implies u_{ij} - r_{ij} = f_{ji} - f_{ij}$

$$\sum_{(i,j)\in E'(j)} (f_{ji} - f_{ij}) + \sum_{(j,i)\in E'(j)} (f_{ij} - f_{ji}) = \sum_{(i,j)\in E(j)} (f_{ji} - f_{ij}) = 0$$

so we deduce the conservation flow constraint.

□

1.2 Flow decomposition and transfer

Definition 1.12 (Flow decompositon)**.** Given f an s-t flow on a network $\mathcal{N}$, we define a *flow-decomposition* as a collection of s-t directed path

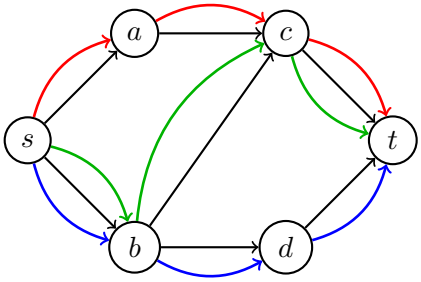
$$P_1, ..., P_k \quad where\ k < m$$

To each path P_i there is a corresponding value $\phi_i \in \mathbb{N}^+ | \phi_i > 0$ which is the value of the path flow.

In a flow decomposition the following rules must be respected:

1. $\forall P_i, P_j,\ |P_i \cap P_j| \neq |P_i| \wedge |P_i \cap P_j| \neq |P_i|$ So each path in the decomposition must differ for at least one edge
2. $val(f) = \sum_{i=1}^{k} \phi_i$

Note that the maximum number of path in our decomposition for any flow is m sonce the capacities are integer.



Example of a decomposed flow

Now that we have established what it means to decompose a flow, we can talk about capacity transfer

Definition 1.13 (Tranfer)**.** Given an edge $(i, j) \in E$ and a *path* P $i \to j$ with $|P| \geq 2$, **to transfer** δ unity of capacity from P to (i, j) means subtracting δ unity of residual capacity from each edge in P and incrementing the (i, j) residual capacity of the same δ unity

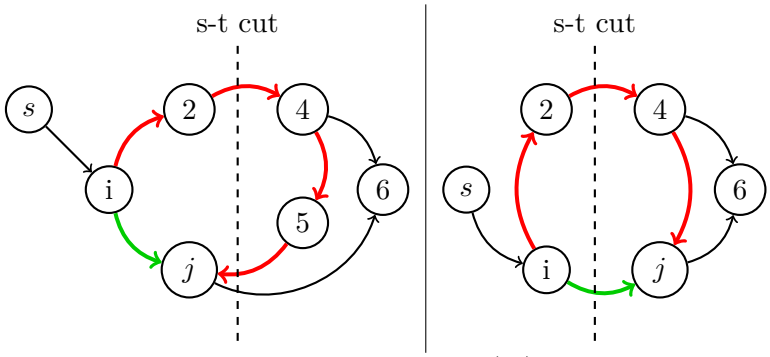
Lemma 1.3 (Capacity tranfer lemma)**.** *Let P be path in G from node i to node j and let (S, T) be an s-t cut. If we transfer δ capacity from the path P to the edge (i, j) and r and r' are respectively the residual capacity of the network before and after the transfer then is true that:*

$$r'(S, T) \leq r(S, T)$$

Proof. The proof is trivial if $i, j \in S \vee i, j \in T$ since $u'(P) \leq u(P) \implies u'(S, T) \leq u(S, T)$.
Otherwise if $i \in S \wedge j \in T$, if we consider $(l, k) \in P$ such that $l \in S \wedge k \in T$ it holds that:

$$u'(S, T) - u(S, T) \leq (u'_{kl} + u'_{ij}) - (u_{kl} + u_{ij}) = -\delta + \delta = 0$$

□



The path P in red, the edge (i,j) in green

We conclude that transferring capacity from a path to an edge doesn't increase the maximum routable flow in a network

1.3 Admissible Graphs

We usually think about graphs composed of nodes linked by edges and measure the distance between two nodes i and j as the minimum number of edges on any path from i to j. This is true just because we don't specify the length of an edge and assume that it is one. Instead, we can specify the length of each edge and still define the distance from a specified node. To do this, we have to define some additional properties which will allow us to achieve our goal. First of all we establish what a valid distance labeling is:

Definition 1.14 (Valid distance labeling)**.** Let N be a network, f a feasible flow on N and l a function that takes as input an edge in G and returns its *length*.
The **distance function** $d : N(G) \to \mathbb{N}$ is **valid** with respect to the residual graph $G[r]$ if it satisfies the following properties

1. $d(t) = 0$
2. $d(i) \leq d(j) + l((i, j))$

Observation 1.2 (Valid distance label property)**.** *A valid distance label d preserves the following properties:*

1. *$d(i)$ is a lower bound of the length of the shortest path from $i \to t$ in the residual graph*
2. *$d(s) \geq n \implies \nexists$ a path P in $G[r]$ from s to t*

Another way to phrase the second property that a valid distance label mast respect $(d(i) \leq d(j) + l((i, j)))$ is that:

$$\neg(d(v) > d(w) + l(v, w))$$

This means that there cannot exist a node i which is more distant from t than any node j adjacent to i, plus the length of the edge (i, j).

Definition 1.15 (Admissible graph)**.** Let G be a network with a feasible flow f, a valid distance label $d : N(G_f) \to \mathbb{N}$ and a length function l. A *residual arc* is a **admissible arc** if it satisfies:

$$d(v) = d(w) + l(v, w) \quad \forall (v, w) \in E(G(f))$$

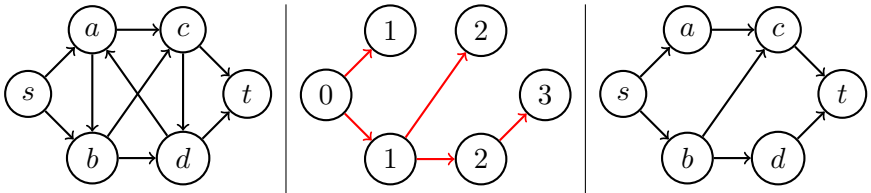
i.e.

$$d(v) > d(w) \vee (d(v) = d(w) \wedge l(v, w) = 0)$$

The **admissible graph** is the graph formed by all admissible arcs. We will represent the admissible graph with the notation $A(f, l, d)$ or just $A(f, d)$ if the length function is trivial.

Observation 1.3. *Let G_f be a residual graph, let B_s be an arborescence given by a BFS on the graph G_f from node s and let A be the admissible graph of G_f then*

$$E(B_s) \subsetneq E(A)$$



Original graph | Arborescence | Admissible graph

Given the distance label definition, we can recall the notions about s-t cut to define the **canonical cut**

Definition 1.16 (Canonical cut)**.** Given a network $\mathcal{N}$ and a distance label d on $\mathcal{N}$, a canonical cut is defined by a partition made as follows

$$(S_k, T_k) = (S_k := \{v \in V(\mathcal{N}) | d(v) \geq k\},\ T_k := V(\mathcal{N}) \setminus S_k)$$

1.4 Finding the max flow

To find the maximum flow in a network in polynomial, we can use the **Edmonds-Karp algorithm**[EK72].

This algorithm is an improvement of the Ford-Fulkerson[FF56] method since it's identical to that algorithm except that the Edmonds-Karp algorithm finds a shortest path from the source to the sink P and augments the flow on the edges of the path by the value x s.t.

$$x = \min_{\forall (i,j) \in P} r_{ij}$$

In this way at each increment at least one edge is deleted from the residual graph and in at most $O(nm)$ increments the algorithm terminates. Since we need $O(n+m)$ time to use the BFS to find the shortest s-t path and another $O(n)$ time to augment the flow in this path, Edmonds-Karp algorithm take $O(nm^2)$ time to find a maximum flow in any network.

Up to here, all notations that we need to recognize a network and its properties were given. The Edmonds-Karp algorithm represents the first step in a series of improvements that will lead us to find the max flow in $O(nm)$. From here on, each algorithm will bring a modification of the previous one while preserving the original intuition. The last algorithm shows how to reach the desired cost even for sparse graphs.

Chapter 2

Dinic's Algorithm

2.1 Introduction

Yefim Dinitz designed this algorithm in 1969 and published[Din06] in 1970. The algorithm builds upon the Edmonds-Karp algorithm, but instead of increasing the flow on just one shortest path $P_{s\to t}$, it increases the flow on all $s \to t$ paths of the same length as the shortest one. This significantly reduces the number of BFS executions required. To simplify the process, the BFS returns a distance label function d, which is used to compute the admissible graph, in which all paths from s to t have the same length $= d(s)$. It is within this graph that the flow is calculated.

2.2 Blocking flows

Since on each path $P_{s\to t} \in A(f, d)$, a flow of value δ is routed

$$\delta = \min_{(i,j)\in A(f,d)} r(i, j)$$

all $P_{s\to t} \in A(f, d)$ will have at least one saturated edge at the end of the increment, and thus, by the end of the increment, there will no longer be a path from s to t. Such a flow is defined as a **blocking flow**.

Definition 2.1 (Blocking flow)**.** A **blocking flow** refers to a flow in a network that saturates at least one edge on every possible path from s to t.

Observation 2.1 (Max flow $\implies$ blocking flow)**.** *In the admissible graph, the max flow is a blocking flow, but the reverse implication is not true.*

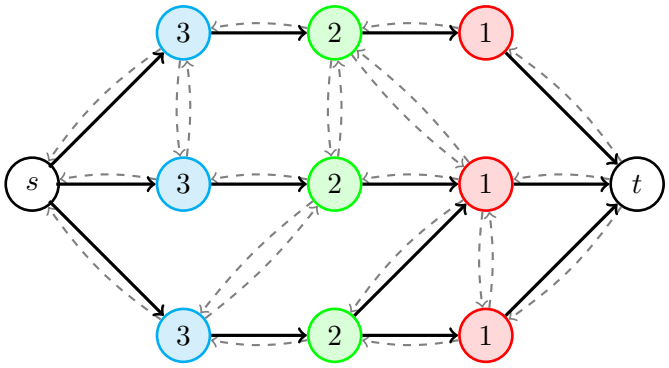
2.3 The algorithm

We can now define an algorithm, but first, some useful observation are helpful in understanding the algorithm:

- The algorithm takes as input the network and the function that assigns a capacity to each edge.
- A function is created that associates a flow with each edge.
- For brevity and readability, the residual graph is denoted as G_f, which represents the residual graph where the flow f is routed, for the current flow f.
- The BFS takes the residual graph as input and thus does not consider saturated edges.

Algoritmh 1 *Dinics-Algorithm(G, c)*

$f_{ij} = 0 \quad \forall (i,j) \in E(G)$
$d = BFS(G_f)$
$A = A(d, f)$
while $d(s) \neq \infty$ **do**
    $\delta = \min_{\forall (i,j) \in P} c(i,j)$
    $f_{ij} = f_{ij} + \delta \ \forall (i,j) \in P$
    $P = findPath(A, s, t)$
    if $P = \varnothing$ **then**
        $d = BFS(G_f)$
        $A = A(d, f)$
    end if
end while
return f



Here we have an example of an admissible graph extracted from a network. The nodes are divided by distance labels from t. Note that all edges have a reverse; a

dashed edge means that it is not admissible. It is also important to remember that an edge can have zero capacity even if it is present in the representation (obviously, since it is not admissible, it is represented as a dashed edge).

2.4 Time complexity

So, since you need $O(m + n)$ time to perform a BFS and another $O(nm)$ time to find all the paths from s to t, the time required to find a blocking flow in each phase is $O(nm)$. Since every time we find a blocking flow, the distance from s increases by at least one and can be at most n, the maximum number of blocking flows found in the algorithm is $O(n)$. Based on these two observations, we can conclude that Dinic's algorithm reduces the time complexity of the max flow problem from the $O(nm^2)$ required by the Edmonds-Karp algorithm to $O(n^2m)$.

Chapter 3

Goldberg-Rao Algorithm

After understanding how Dinic's algorithm works, we can move to the next step and focus on the algorithm published by Andrew V. Goldberg and Satish Rao in 1998 [GR98].

By optimizing the Dinic's Algorithm, the Goldberg-Rao algorithm achieves a **computational cost** of

$$\tilde{O}(\min\{n^{2/3}, m^{1/2}\} \cdot m)$$

on a network with integer capacities, which, when considering logarithmic factors, becomes $O(\min\{n^{2/3}, m^{1/2}\} m \log n \log nU)$.

Note: From here on, we will abbreviate the expression $\min\{n^{2/3}, m^{1/2}\}$ using Λ.

3.1 Overview

At the core of the optimization is the idea of **contracting** the network according to certain specific parameters. The speed-up lies in computing the flow in a contracted graph, which is more efficient than computing it in the original one. The algorithm is based on Valid distance labeling and introduces a new **binary length function** : $\bar{l}((v, w)) : E \to \{0, 1\}$.

The new length function assigns a value of zero to all edges that meet certain capacity requirements (which we will describe in more detail later), such edges are called "zero-length".

By setting the length of the edges connecting two or more nodes to zero, we can consider them as a single node. Thus, by contracting the components connected by edges of length 0, it is possible to significantly reduce the number of flow increments and therefore the computational cost of the algorithm.

3.2 The Δ parameter

The issue with contracting the graph is that when we send flow from the source to the sink, we must ensure that this flow respects the capacity constraints of all the edges, including those contracted. To ensure that the flow calculated on the contracted graph is valid for the original graph as well, a parameter Δ is used, which serves two purposes. The first function is as a **lower bound** on the capacity of zero-length edges. Edges with residual capacity greater than Δ are first selected, and the length of these edges is set to 0. Subsequently, all components connected by zero-length edges are contracted. Finally, a blocking flow (exactly as in Dinic's algorithm) is calculated in the contracted graph. At this stage, the parameter Δ serves its second function, which is as an **upper bound for the blocking flow**. The computation of the blocking flow stops either when such a flow is found, or just before the flow value exceeds Δ. This second condition ensures that the flow remains feasible even for the original network.

By increasing the flow by at most Δ, we ensure that the capacity constraints are respected, but we can no longer guarantee that the flow is blocking. Therefore, we must choose a value for Δ that is both small enough to contract the graph as much as possible, but also adequately selected to keep the number of flow increments as low as possible, thus ensuring the desired computational cost.

3.3 The stop condition

To terminate its execution, the algorithm estimates the difference between the maximum achievable flow (which will be called F) and the flow it has computed. When this difference becomes less than 1, the algorithm terminates. Since the capacities of the network are all integers, this ensures that the maximum possible flow has been reached. An initial useful value for F is $F = n \cdot U_{max}$, and later, the residual capacity of the canonical cut will be used (further details will be provided later).

3.4 Estimating the residual flow

We already know that the residual capacity of each cut $r(S, T)$ represents an upper bound on the max flow (Max residual flow min residual cut). To estimate the residual flow quickly and efficiently, we can analyze the Canonical cut.

Lemma 3.1. *min $r(S_k, T_k)$ in $O(m)$ time The canonical cut with the minimum capacity can be found in $O(m)$ time.*

Proof. Exploiting the fact that each edge has a length of at most 1, and therefore can cross at most one canonical cut, we can define the following subroutine.

Algoritmh 2 *canCutCapacity(G_f, d, l)*

1: **for** $k \leftarrow 0$ *to* $d(s)$ **do** $r(S_k, T_k) = 0$
2: **end for**
3: **for** $(u, v) \in E(G_f)$ **do**
4: **if** $d(v) > d(w)$ **then** $r(S_{d(v)}, T_{d(v)}) += r(v, w)$
5: **end if**
6: **end for**
7: **return** argmin $r(S_k, T_k)$

The correctness and computational cost of this routine are fairly straightforward. □

To manage the computational cost, we need to ensure that the value of F decreases quickly enough without overburdening the algorithm. First, we can group all the iterations of the algorithm into **phases** and update the value of F at the minimum canonical cut only between the end of one phase and the start of a new one. If we update the value of F only when $\min r(S_k, T_k) \leq F/2$, the algorithm will terminate after at most $\log nU_{max}$ phases.

3.5 Binary length function

Two other issues arise from contracting the graph and modifying the length function:

1. Choosing a Δ that is too small would make the flow increase too slowly while choosing it too large would not contract the graph enough to justify the management costs.

2. In Dinic's algorithm, the blocking flow always guaranteed an increase in the distance from s to t, but with zero-length edges, this is no longer guaranteed.

In this section, we show the choices that were made to address these two issues. While the effectiveness of the solution to the second problem is promptly demonstrated, the effectiveness of the choice of the Δ parameter will only become clear in the section where the various computational costs are proven.

3.5.1 How to zero lengths

As previously mentioned, we need an upper bound Δ to respect the capacity constraints. At the same time, to meet the declared computational cost, we need the blocking flow increments to be at most Λ. Thus, we can initialize:

$$\Delta = \lceil F/\Lambda \rceil$$

The criterion for assigning zero-length to an edge is as follows:

Definition 3.1 (Length function)**.** Let r be the residual function of any residual graph G_f. We define the length function $l((u,v))$ as a function that associate to the edge (u,v) the value 1 or 0 as follow:

$$l(u.w) = \begin{cases} 0 & r_{vw} \geq 3\Delta \\ 1 & altrimenti \end{cases}$$

However, to achieve the desired computational cost, it is necessary to add a specification to this function.

Definition 3.2 (Special Arc)**.** Any edge (v,w) is said **special** If it meets all the following requirements:

- $2\Delta \leq r_{v,w} < 3\Delta$
- $d(v) = d(w)$
- $r_{wv} \geq 3\Delta$

By applying this definition to the length function, we can define a more complex function, which we distinguish from the first by calling it $\bar{l}$. The modified function also takes into account special edges:

$$\bar{l}(u.w) = \begin{cases} 0 & r_{vw} \geq 3\Delta \vee specialArc((v,w)) \\ 1 & altrimenti \end{cases}$$

Observation 3.1. *Introducing special edges does not change the distance labeling:* $d_l = d_{\bar{l}}$

Lemma 3.2 (From contract to original)**.** *Let's suppose we have contracted the original network as described so far, and routed a flow f through the contracted graph.*

The computational cost of adapting this flow through the original graph is $O(m)$.

Proof. Through the following steps, it is intuitive how the flow can be adapted:

1. Choose any vertex in each contracted component.
2. Form an in-tree and an out-tree rooted at the chosen vertices.
3. Route the positive flow from the in-tree to the root.
4. From the out-tree, redirect the incoming flow from the root to all other connected nodes.

Since the maximum flow we route is Δ and all nodes in the contracted components have a cost of at least 2Δ, we are assured that the flow respects the capacities of the network. This method has a cost directly proportional to the number of edges in the connected components. □

3.5.2 How to increase distance

In Dinic's Algorithm, the proof that the blocking flow strictly increases the distance between s and t is quite obvious. The same cannot be said for the Goldberg-Rao case due to the presence of zero-length edges. Therefore, it is essential to prove the following theorem to ensure that the algorithm terminates.

Theorem 3.1. *Blocking flow with binary length Let $\bar{f}$ be a flow in $A(f, \bar{l}, d_l)$, let $f' = f + \bar{f}$ be the increased flow, and let l' is the length function corresponding to f'. Then:*

1. *d_l is a distance labeling with respect to l'*
2. *$d_{l'}(s) \geq d_l(s)$*
3. *if $\bar{f}$ is blocking $\implies$ $d_{l'}(s) > d_l(s)$*

Proof. Let's proceed point by point

1. d_l is a distance labeling with respect to l' By the definition of distance labeling, $d_l(v) \leq d_l(w) + \bar{l}(v, w)$ (remembering that $d_l = d_{\bar{l}}$), we therefore need to prove that $d_l(v) \leq d_l(w) + l'(v, w)$.
This is trivially true if $d_l(v) \leq d_l(w)$.
If $d_l(v) > d_l(w)$ i.e. $d_{\bar{l}}(v) > d_{\bar{l}}(w)$ then (w, v) is not admissible with respect to $\bar{l}$.
Since $l'(v, w) \geq \bar{l}(v, w)$, the statement follows.

2. $d_{l'}(s) \geq d_l(s)$ Let $L := \{l_0, l_1, ..., l_n\}$ be the ordered set of all length functions calculated between the iterations of the algorithm. Then, for any $0 \leq i \leq j \leq n$, we have $d_{l_i}(s) \leq d_{l_j}(s)$.

We distinguish between two iterations as follows:

1. In iteration i: Let $l(u, v) = l_i(u, v)$ be the length function and $d(x) = d_{l_i}(x)$ be the distance. Together with the flow, they define $A(f, l_i, d_{l_i})$. Let Γ be the shortest $s \to t$ path in A, where $\Gamma \subseteq A$.
2. In iteration j: Let $l'(u, v) = l_j(u, v)$ be the length function and $d'(x) = d_{l_j}(x)$ be the distance. Together with the flow, they define $A'(f, l_j, d_{l_j})$. Let Γ' be the shortest $s \to t$ path in A', where $\Gamma' \subseteq A'$.

Suppose by contradiction that there exist two iterations $0 \leq i \leq j$ such that $d(s) > d'(s)$:

$$\implies \exists \Gamma\, s \to t,\ \Gamma'\, s \to t : \sum_{(v,w)\in\Gamma} l(v, w) \geq \sum_{(v,w)\in\Gamma'} l'(v, w)$$

In other words, as the iterations progress, s and t have gotten closer.

We immediately exclude the case where $\Gamma = \Gamma'$ since

$$\forall (v,w) \in A \cap A',\ l(v,w) \leq l'(v,w) \implies l(\Gamma) \leq l'(\Gamma')$$

Note: $l(\Gamma) = \sum_{(v,w)\in\Gamma} l(v,w)$.

Now consider Γ and Γ'. Let w be the last node in Γ for which $d(w) > d'(w)$, and let x be the next node:

$$w \in \Gamma : d(w) > d'(w) \wedge \exists x = succ_\Gamma(w) : d(x) \leq d'(x)$$

w and x are always well defined because we assume $d(s) > d'(s)$ and $d(t) = d'(t) = 0$ by definition.

Thus, there exists an arc (w,y) in Γ' with $y \neq x$ such that $d'(y) < d'(x)$. $x \neq y$, because if they were the same node, then:

$$d'(w) = d'(x) + l'(w,x) \geq d(w)$$

which contradicts the hypothesis.

To summarize, we know that:

1. $d(w) > d'(w) \iff d(x) + l(w,x) > d'(y) + l'(w,y)$.
   While we don't know the exact distance $d(y)$, we know that:

   $$d'(y) = \sum_{(a,b)\in y-t\subseteq\Gamma'} l'(a,b) \geq \sum_{(a,b)\in y-t\subseteq\Gamma'} l(a,b)$$

   Therefore, the path in iteration j is greater than or equal to the path in iteration i.

2. $d(y) + l(w,y) \leq d'(y) + l'(w,y) < d(x) + l(w,x)$.

However, we know that $d(w) = d(x) + l(w,x)$, which is **absurd** because it is not the minimal distance from $w \to t$, as it is greater than $d(y) + l(w,y)$.

We know for certain that the path $w - y \to t$ exists in A because (unless there is a shorter one) it represents the minimal distance from $w \to t$.

From this contradiction, the only conclusions are that either the path through y was not reachable in iteration i, making it impossible to reach it later, or if a shorter $s \to t$ path exists in iteration j than in iteration i, we made an error in considering the path in iteration i.

3. Se $\bar{f}$ è bloccante allora $d_l(s) < d_{l'}(s)$ To show that the blocking flow increases the distance of node s, we define the following notation:

$$c(v,w) := d_l(w) - d_l(v) + l'(v,w)$$

which represents the change in the length of an edge connecting two adjacent nodes.

We can assert that:

$$\forall (v,w) \in E,\ c(v,w) \geq 0$$

since $l'(v,w) \geq l(v,w)$, which implies:

$$d_l(w) - d_l(v) < 0 \iff l(v,w) = 1 \implies l'(v,w) = 1$$

Now, consider any path Γ in $G_{f'}$, the length of the path is equal to:

$$l'(\Gamma) = d_l(s) + c(\Gamma)$$

Therefore, to show that the path is longer, we need to show that:

$$\forall \text{ shortest } s-t \text{ path } \Gamma \in G_{f'} \implies \exists (v,w) \in \Gamma \text{ where } c(v,w) > 0$$

We now have a tool to demonstrate that the blocking flow increases the distance of s.

Since $\bar{f}$ is blocking in $A(f, \bar{l}, d_l)$, Γ must contain an edge (v,w) that is not present in $A(f, \bar{l}, d_l)$.

Furthermore, we can state that $d_l(v) \leq d_l(w)$, either because $(v,w) \in G_f$, but then if $d_l(v) > d_l(w)$, we would have $(v,w) \in A(f, \bar{l}, d_l)$, or because $(v,w) \notin G_f$, but it appears in $G_{f'}$, which is only possible if the flow is incremented in the opposite direction, causing the residual edge to appear. Therefore, $(w,v) \in A(f, \bar{l}, d_l)$, which implies that $d_l(v) \leq d_l(w)$.

Now, suppose for contradiction that $c(v,w) = 0$, so $d_l(v) = d_l(w)$ and $l'(v,w) = 0$. The fact that (v,w) is not in $A(f, \bar{l}, d_l)$ implies that either (v,w) is not in G_f, but then we have already shown that the opposite edge $(w,v) \in A(f, \bar{l}, d_l)$, or that $(v,w) \in G_f$ but does not meet the distance labeling requirements to belong to the Admissible graph $A(f, \bar{l}, d_l)$. Since $d_l(v) = d_l(w)$, then $l(v,w) = 1$. We note that $1 = l(v,w) > l'(v,w) = 0$, which implies that we have incremented the flow on the opposite edge (w,v). Thus, in any case, the edge $(w,v) \in A(f, \bar{l}, d_l)$.

As shown earlier, since $d_l(v) = d_l(w)$,

$$(w,v) \in A(f, \bar{l}, d_l) \iff l(w,v) = 0$$

We conclude that:

- During the flow increments, we routed a flow (of at most Δ) through the edge (w,v)
- $u_f(w,v) \geq 3\Delta$ because $l(w,v) = 0$
- After this increment, we have $u_{f'}(v,w) \geq 3\Delta$ because $l'(v,w) = 0$
- Thus $u_f(v,w) \geq 2\Delta$
- But then the edge (v,w) was a *special edge* even before the increment, since $d_l(v) = d_l(w) \wedge u_f(w,v) \geq 3\Delta \wedge u_f(v,w) \geq 2\Delta$

We therefore conclude that:

$$d_l(v) = d_l(w) \implies d_{\bar{l}}(v) = d_{\bar{l}}(w) \wedge \bar{l}(v,w) = 0 \implies (v,w) \in A(f, \bar{l}, d_l)$$

which is a contradiction.

□

3.6 Complexity

We have already shown how to find the maximum flow in the graph. Before diving into the cost of a phase, let's review the structure of the algorithm described so far.

Algoritmh 3 *Goldberg-RaoAlgorithm(G, c)*

1: $n = |N(G)|$
2: $F = U \cdot n$
3: $\Delta = F/\Lambda$
4: **for** $(i, j) \in E(G)$ **do** $f_{ij} = 0$
5: **end for**
6: **while** $F \geq 1$ **do**
7: $l = update\_length(n, \Delta)$ ▷ return a length function w.r.t. Δ
8: $d_l = BFS(G_f, l)$ ▷ return a distance labeling w.r.t. l
9: $G^c = contract(G_f, l)$
10: $A = A(G^c, d_l, l)$ ▷ return the admissible graph
11: $f' = find\_blocking\_or\_Delta\_flow(A)$ ▷ Dinic's style
12: $f_{ad} = fit(f')$ ▷ the procedure to adapt the flow to the original graph
13: $f = f + f_{ad}$
14: $c = r(canCutCapacity(G_f, d, l))$ ▷ residual capacity of min canon. cut
15: **if** $c \leq F/2$: **then**
16: $F = c$
17: $\Delta = F/\Lambda$
18: **end if**
19: **end while**
20: **return** f

Remark:

The cost stated at the beginning is in:

$$O(\min\{n^{2/3}, m^{1/2}\} \cdot m \log n \log mU_{max})$$

Using more advanced data structures, you can achieve the cost of:

$$O(\min\{n^{2/3}, m^{1/2}\} \cdot m \log \frac{n^2}{m} \log U_{max})$$

We divided the number of **phases** so that the estimated maximum flow (F) is halved in each phase. This gives us several phases on the order of $\log(F)$, which is $\log(mU_{max})$. We have shown how both the calculation of the minimum canonical cut and the adjustment of the flow to the original network can be computed in $O(m)$ time. However, we still need to analyze the cost of each phase, that is, how quickly the minimum canonical cut is halved.

From the corollary of the following lemma, we demonstrate what was previously stated when we fixed the value of the parameter Δ. With this lemma, we estimate the maximum capacity for the canonical cut, which is then used to estimate F, while in the subsequent corollary, we show how the parameters for which we contract the graph lead this capacity to halve within $O(\Lambda)$ blocking flows.

Lemma 3.3. *The minimum capacity of a canonical cut* $(\bar{S}, \bar{T})$ *satisfies*

$$u_f(\bar{S}, \bar{T}) \leq \frac{mM}{d_l(s)}$$

where M *represents the length-one edge with the highest capacity.*

Proof. It is clear that the best way to maximize the capacity of the minimum canonical cut is by assuming that all edges have the capacity of the edge with the highest capacity, and then evenly dividing the edges among the various cuts.

□

From this initial estimate follows the corollary.

Corollary 3.1. *During each phase, there are at most* $O(\Lambda)$ *blocking flow increments.*

Proof. Let us assume that $\Lambda = m^{1/2}$. Since we have shown that each blocking flow increases $d(s)$ by at least one, we can be sure that after $6\lceil\Lambda\rceil$ increments, $d_l(s) \geq 6m^{1/2}$. Thus, we can take the estimate from the lemma and state that:

$$u_f(\bar{S}, \bar{T}) \leq \frac{mM}{d_l(s)} \leq \frac{3m}{d_l(s)}\Delta \leq \frac{3m}{6m^{1/2}}\frac{F}{m^{1/2}} = \frac{F}{2}$$

Thus, after $\lceil\Lambda\rceil$, the phase ends.
For $\Lambda = n^{2/3}$, the proof is analogous and leads to the same conclusion. In conclusion, the cost of each phase is on the order of $O(\Lambda)$. □

At this point, the last bottleneck is represented by the cost of finding a blocking flow or a flow of value Δ (which are computationally equivalent): This would require a cost of:

- $O(mn)$ in a naive approach;
- $O(m \log n)$ using dynamic trees[SE83];
- $O(m \log(n^2/m))$ using size-bounded dynamic trees;

Combining the cost of:

$\times$ finding a blocking flow

$\times$ iterations in each phase

$\times$ the number of phases

$\times$ additional costs in $O(m)$

Conclusion

Goldberg and Rao published their algorithm in a 1998 paper. About four years earlier, V. King, S. Rao, and R. Tarjan published a paper in which they presented an algorithm capable of finding the maximum flow in $O(nm)$ time, provided the algorithm had enough edges relative to the number of nodes. The stated cost is $O(nm(\log_{m/n \log n} n))$, but if $m/n = \Omega(n^{\varepsilon})$ for some $\varepsilon > 0$, the cost becomes $O(nm)$. However, the problem of finding the maximum flow in polynomial time remains unresolved, as it is still unclear how to calculate it for sparser graphs than those covered by the King-Rao-Tarjan[KRT92] algorithm. The next chapter analyzes the solution proposed in 2013 by James B. Orlin[Orl13], which leverages a specific condition of the Goldberg-Rao algorithm and develops a strategy for compacting and contracting the graph, along with approximations in a series of optimal flows, to make the algorithm strictly polynomial where King-Rao-Tarjan fails.

Chapter 4

Orlin's Algorithm

Before delving into Orlin's algorithm, let's review the current state of the problem. The Goldberg-Rao algorithm achieves what is called a *weakly polynomial* time complexity, solving the problem in $\log mU$ phases, each with a cost of $O(\Lambda m \log(n^2/m))$, where $\Lambda = \min\{n^{2/3}, m^{1/2}\}$. If we aim to solve the maximum flow problem with a *strongly polynomial* time complexity, there is the King-Rao-Tarjan algorithm. However, this algorithm achieves a time cost of $O(nm)$ only under the condition that $m = \Omega(n^{1+\varepsilon})$ for some $\varepsilon > 0$. If the number of edges is insufficient, its cost is $O(nm \log_{m/(n \log n)} n)$.

In the following algorithm, James B. Orlin proposes a solution that, by leveraging the Goldberg-Rao algorithm, manages to solve the maximum flow problem in $O(nm)$ when $m = O(n^{1+\varepsilon})$. This makes it possible to solve the problem in strictly polynomial time for any values of n and m, without being limited by edge capacities.

4.1 Overview

The idea arises from several observations:
The Goldberg-Rao algorithm operates through **increment phases** that take a Δ-optimal flow and make it $\Delta/2$-optimal. Furthermore, it is noted that $\log_{8m} mU \leq 1 + \log U$; in fact, from now on, $\log U$ increment phases will be considered. If we set $\Lambda = O(m^{1/2})$, we can observe that

$$\log U < m^{7/16} \implies \tilde{O}(m^{3/2} \cdot m^{7/16}) = \tilde{O}(m^{31/16})$$

(the notation $\tilde{O}$ ignores logarithmic factors).
Delving deeper into the calculations, we observe

$$\tilde{O}(m^{31/16}) = O(m \cdot m^{15/16} \cdot \log(n^2/m)) = O(m \cdot n^{(16/15)^{15/16}} \cdot \log(n^2/m))$$

since we are looking for an algorithm when $m = O(n^{1+\varepsilon})$. If $1 + \varepsilon < 16/15$ and $\log U < m^{7/16}$, we can achieve an optimal solution with a polynomial cost of $O(nm)$ by using only the Goldberg-Rao algorithm.

However, this is only true if the number of edges is sufficiently large compared to the edge with the maximum capacity.

Hence, the idea of contracting and compacting the network arises to ensure the calculation of the maximum flow under optimal conditions.

The algorithm presents two bottlenecks:

1. Creation of the compacted representation (more precisely, maintaining the transitive closure)
2. Transitioning from the compacted flow to the extended flow.

4.2 The improvement phase

If the Goldberg-Rao algorithm, in a sense, 'wraps' Dinic's Algorithm into a higher level of abstraction where the original graph is modified, Orlin does the same with Goldberg-Rao.

A higher level of increment phase is introduced, within which a slightly modified version of the Goldberg-Rao algorithm is executed. So let's examine the input and output of each phase:

- **Input**
  1. a *Flow* f
  2. a *Residual Graph* G_f,
  3. an s-t cut (S, T)

  Since with the flow and the residual graph, we can compose the residual function, when we consider the flow and the graph we can say to have also the residual function "r".
  We can represent the input as the tuple (r, S, T).
- **Output**
  1. a *Flow* f'
  2. a *Residual Graph* G'_f
  3. an s-t cut (S', T') such that $r'(S', T') \leq \frac{r(S,T)}{8m}$

This phase is called the Δ-improvement phase, where $\Delta = r(S, T)$. Alongside the parameter Δ, a parameter Γ is introduced, where $\Gamma \leq \Delta$, which will be used to create the Γ-compact network.

Depending on the conditions, the Δ-improvement will be executed either on the original network G or on the Γ-compact network G^c, which we will introduce later.

4.3 Δ-abundant the abundance graph

In questa sezione viene presentato il concetto di **Abbondanza**.

Definition 4.1 (Δ-abundant arc)**.** Let $\Delta = r(S,T)$ then any edge (i,j) is said Δ-abundant if $r_{ij} \geq 2\Delta$

Lemma 4.1. *Let (r,S,T) be the input for a Δ-improvement phase. If the edge (i,j) is abundant before the phase then it will be abundant for for all subsequent phases.*

Proof. Since $\Delta' \leq \frac{\Delta}{8m}$ and recalling that $r_{ij} \geq 2\Delta$ it follows that allora

$$r'_{ij} \geq r_{ij} - \Delta \geq \Delta \geq 2\Delta'$$

□

Definition 4.2 (Abundance Graph)**.** Given a network G, its **Abundance Graph** G^{ab} is defined as:

$$G^{ab} := (N(G), \{(i,j) | (i,j) \in E(G) \wedge r_{ij} \geq 2\Delta\})$$

Observation 4.1. *By lemma 4.1, as iterations proceed, the abundant graph can only acquire new edges, and never lose them.*

The abundant graph has two purposes:

1. All cycles formed by abundant edges are *contracted* into a single node.
2. All nodes adjacent only to abundant edges (or edges with capacity that is too small) are *compacted*.

Additionally, the algorithm maintains the transitive closure of all nodes connected by an **abundant path**, meaning a path composed only of abundant edges.
If there exists an abundant path between nodes i and j, this is indicated as $i \implies j$, and the information is stored in an $\mathbf{M}_{n\times n}$ matrix, where at position $\mathbf{M}_{i,j}$ is the node that precedes j in the path starting from i. If multiple paths are created during the iterations, the first one found is kept.

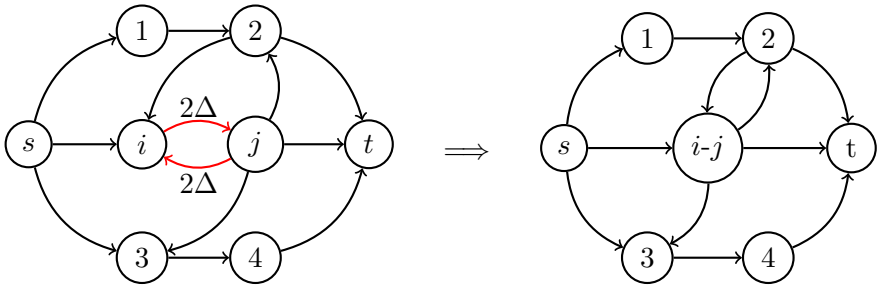
The transitive closure can be maintained in time $O(nm)$ using Italiano's algorithm[Ita86]. In this way, it is always possible to reconstruct an abundant path P in $O(|P|)$ (we will see later that contracting the graph does not prevent this nor does it alter its cost).

4.4 Contractions in the abundance graph

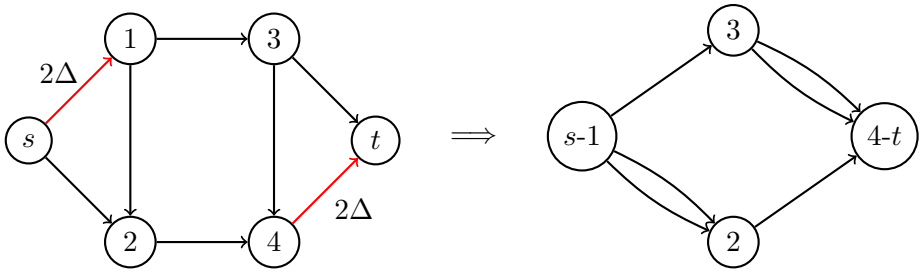
Let's now see how to exploit the abundant graph to contract the graph on which we calculate the max-flow and make the algorithm more efficient.

We analyze three different examples of contractions:

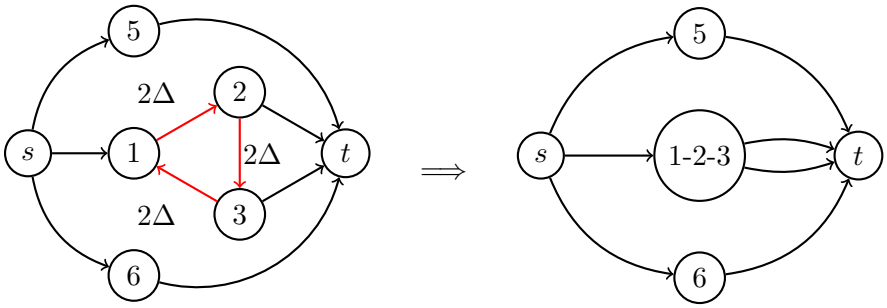
Suppose there are two nodes i and j such that $r_{ij} \geq 2\Delta$ and $r_{ji} \geq 2\Delta$. We can contract the two nodes into a single one, preserving the edges of both.



Since there are no reverse edges for the external edges, it is possible to contract external edges under the sole condition that they are abundant.



And so all the abundant cycles



Observation 4.2. *It is possible that when the contracted graph is expanded again, the flow conservation law may be violated.*

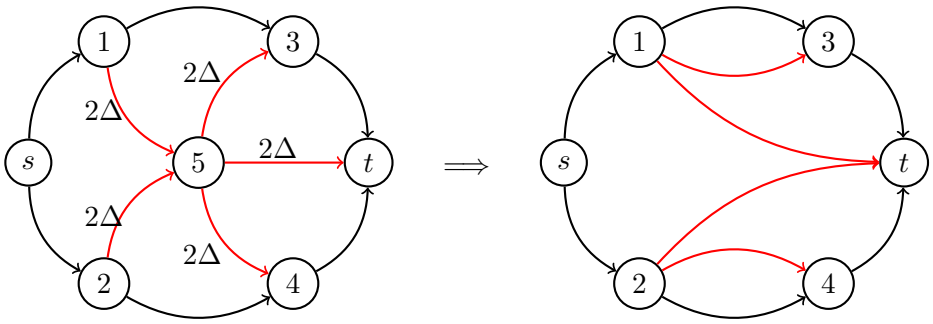
However, this violation is minor, at most 2Δ units; thus, as shown by Goldberg and Rao, the contraction, expansion, and adjustment for flow conservation can be performed in $O(m)$ time.

4.5 How to compactify a network

In addition to the contraction of the graph, another transformation is necessary: the *compaction*. To obtain a compact graph, we first demonstrate how to achieve an intermediate version, namely the **strongly compact network**.

It is important to understand the difference between contracting and compacting:

In contraction, a single node is created that represents the abundant cycle, and the original edges not belonging to the cycle are preserved. However, when compacting a graph, a node that has all adjacent edges abundant is eliminated, and the outgoing edges are consequently replaced by pseudo-edges.



The following algorithm has a time complexity of $O(m + |E^{sc}|)$ since pseudo-edges can be constructed in $O(1)$ time, given that the transitive closure is dynamically preserved.

Definition 4.3 (Strongly compact network)**.** We define the **Strongly compact** as $G^{sc} = (N^{sc}, E^{sc})$ originated from the network G:

1. Contract the graph of all abundant cycles and the external abundant edges. Let (r, S, T) be the input after contraction.
2. Let $N^{sc} \subseteq N(G)$ be the set of nodes that are adjacent to at least one non-abundant edge.
   We will refer to $N(G) \setminus N^{sc}$ as the set of *strongly compactible* nodes.
3. We define the edges as $E^{sc} = E^1 \cup E^2$ where:
   $E^1 = \{(i,j) : i \in N^{sc} \wedge j \in N^{sc} \wedge (i,j) \in E(G)\}$
   $E^2 = \{(i,j) : i \in N^{sc} \wedge j \in N^{sc} \wedge i \implies j\}$
   Thus, we have original edges in E^1 and pseudo-edges that derive from the abundant paths.

When we contract the graph, we are sure that if the flow routed in the contracted graph is less than a certain parameter Δ with which we contracted the graph, then that flow is also adaptable to the original one. The following theorem shows us that the same is true for strongly compact graphs.

Theorem 4.1 ($f_{max} = f^{sc}_{max}$)**.** *Let f_{max} be the maximum flow in the network G and let f^{sc}_{max} be the maximum flow in G^{sc}. Then*

$$f_{max} = f^{sc}_{max}$$

Proof. We have already shown that any flow in G^{sc} can be rerouted in G. If we take a flow in G, it can be routed in G^{sc} using the **flow decomposition** to obtain from f a set of paths that differ by at least one edge,

$$f := \{P^0, P^1, ..., P^k\}$$

We can further subdivide each $P^a \in f$ into subpaths

$$P^a_{i\to j} | i \in N^{sc} \land j \in N^{sc} \land \forall q \in P^a_{i\to j}, q \neq i \land q \neq j \implies q \in N \setminus N^{sc}$$

At this point, we replace each $P^a_{i\to j}$ in G that is not entirely contained in G^{sc} with the corresponding pseudo-edge (i, j).

□

4.6 From sc-compact to Γ-compact

An sc-compact graph is not sufficiently compacted for our purpose, so it will be further compacted. To avoid confusion, we will use another parameter for this additional compaction.

The second parameter is Γ. The Γ-compact graph (composed solely of Γ-critical nodes) is formed starting from the sc-compact network and transferring capacities from paths where the only Γ-critical nodes are the endpoints, to pseudo-edges that connect these endpoints.

The parameter Γ is also used to ensure that in each improvement phase, there are always "enough" nodes. Thus, Γ initially takes the same value as Δ, but if, in a given improvement phase, the Γ-critical nodes are too few, a smaller Γ will be chosen to increase their number.

The definition of "enough" nodes and how the value of Γ is selected will be shown later.

Before proceeding, it's important to distinguish between different types of edges.

Definition 4.4 (Capacity Classifications)**.** An edge (i, j) with respect to Γ has:

1. **small capacity** if $u_{ij} + u_{ji} < \Gamma/(64m^3)$
2. **medium capacity** if $\Gamma/(64m^3) \leq u_{ij} + u_{ji} \ \land r_{ij} < 2\Delta \land r_{ji} < 2\Delta$
3. **abundant capacity** if $r_{ij} \geq 2\Delta$
4. **anti-abundant capacity** if $(j, i) \in E^{ab} \lor \ (i, j)$ is a non-abundant external edge.

Where E^{ab} and E^{-ab} represent, respectively, the set of abundant and anti-abundant edges at the beginning of the improvement phase.

Observation 4.3. *Since we have contracted abundant cycles,*

$$(i,j) \in E^{ab} \implies (j,i) \notin E^{ab}$$

Another necessary tool to decide which nodes to compact is the **potential function**.

Definition 4.5 (Potential function)**.** Given a node $j \in N$, a residual capacity function r, and a subset of edges adjacent to j ($\tilde{E}(j)$) we can define the potential function as:

$$\Phi(j, r, \tilde{E}(j)) = \sum_{(i,j)\in\tilde{E}(j)} r_{ij} - \sum_{(j,i)\in\tilde{E}(j)} r_{ji}$$

Based on these parameters, in each improvement phase, nodes that can be compacted are distinguished from those that cannot. These two distinctions are referred to as Γ-compactible nodes and Γ-critical nodes respectively.

Definition 4.6 (Γ-critical e Γ-compactible)**.** A node j is said to be Γ-critical if it is adjacent to at least one Γ-medium edge or if $|\Phi(j, r, E^{-ab})| > \Gamma/(16m^2)$.

If a node is not Γ-critical, then it is said to be **Γ-compactible**.

Given a network G, we define the **Γ-compact network** of G as:

$$G^c := (N^c, E^c)$$

where N^c consists of all and only Γ-critical nodes, while E^c is the set of edges that will be defined later.

To build the Γ-compact network, units of residual capacity are iteratively transferred from various paths to pseudo-edges. The idea is to transfer capacity from paths connecting two nodes $i, j \in N$ to the edge (or pseudo-edge) (i, j), allowing further compaction of the graph. However, these pseudo-edges are only part of the edges comprising E^c, which can be defined as:

$$E^c = E^1 \cup E^2 \cup E^3$$

$E^1 = \{(i,j) | i, j \in N^c \wedge (i,j) \in E(G)\}$ meaning the original edges connecting two Γ-critical nodes.
$E^2 = \{(i,j) | i, j \in N^c \wedge i \implies j\}$, i.e., the abundant edges.

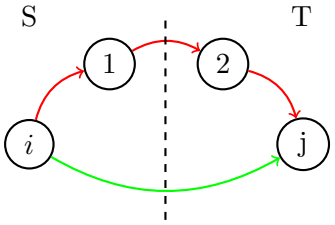
The edges in E^3 are the pseudo-edges created by transferring flow from paths consisting of anti-abundant edges. The following lemma shows that, if selected based on an appropriate criterion, the flow transfer does not reduce the capacity of any (S, T) cut, thus preserving the maximum flow that can be computed.

Lemma 4.2 (Flow transfer safety)**.** *Let (S, T) be an s-t cut in G with $r(S, T) \leq \Delta$, and let $A' = E^{-ab}$. Suppose there exists $P \subseteq A'$, a path from $i \to j$, and that $(i, j) \in A'$. If r' is the new residual capacity function obtained by shifting δ units of residual capacity from P to (i, j), then we can state that:*

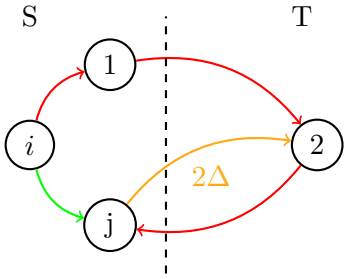
1. $\forall k \in N(G),\ \Phi(k, r', A') = \Phi(k, r, A')$
2. $r'(S,T) = r(S,T)$

Proof. Let's divide the proof by points:

1. The first point is intuitive because, for each node in P that is different from i and j, I am subtracting the same residual capacity for both incoming and outgoing edges, while for nodes i and j, the summations of Φ remain unchanged.
2. The second point is trivial if $|P| = 1$, so let's consider the case where $|P| \geq 2$. Define $P = p_1, ..., p_k$,where $p_1 \in S$ and at least one $p_q \in T$.
   Since we have established that $r(S,T) \leq \Delta$ and $P \subseteq A'$ we are definitely in a situation like this:



   Because if an edge of P passed from T to S, it would violate $r(S,T) \leq \Delta$ since $\forall (a,b) \in A'(\Delta),\ r_{ab} < 2\Delta \wedge r_{ba} \geq 2\Delta$, we would have:



   So the (S,T) would have certainly a residual capacity $\geq \Delta$. Once this has been established, it becomes clear that the transfer of residual capacity does not affect the residual capacity of the cut.

□

We can then observe the following:

Definition 4.7 (Transferrable residual capacity)**.** To transfer δ capacity from a path P from $i \to j$ to the arc (i,j), the following conditions must be satisfied:

1. $|P| \geq 2$
2. $r(P) > 0$
3. $P \subseteq A'$

Moreover, when creating the Γ-compact network, the following additional conditions are essential:

1. $i, j \in N^c$
2. $P \setminus \{i, j\} \subseteq N(G) \setminus N^c$

The capacity transferred from P is given by:

$$\delta = r(P) = \min_{(a,b)\in P} r(a, b)$$

Thus, each time, at least one anti-abundant arc is saturated. If there exists a path $P \subseteq A'$ from $i \to j$ but $(i, j) \notin E(G)$, a pseudo-arc is created. These are the anti-abundant pseudo-arcs that will form E^3.

We now analyze the procedure to transfer all residual capacities needed to form E^3 and return the set of arcs with their respective residual capacities r^c_{ij} for every $(i, j) \in E^3$.

The algorithm takes as input (in addition to the residual function r) the set of Γ-critical nodes and the set of anti-abundant arcs that do not connect two Γ-critical nodes:

- $G^c := \{n \mid n \in N(G) \wedge \Gamma\text{-critical}(n)\}$
- $H := \{(i, j) \mid (i, j) \in E^{-ab} \wedge (i \notin G^c \vee j \notin G^c)\}$

Algoritmh 4 *capacity-transfer(r, H, G^c)*

1: **for** $(i, j) \in H$ **do**
2: $q_{ij} = r_{ij}$
3: **end for**
4: **while** H $\neq \varnothing$ **do**
5: pick $i \mid \exists (i, k) \in H \wedge \nexists (j, i) \in H$
6: pick $l \mid l \in N^c \vee \nexists (l, k) \in H$
7: P $= DFS(i, l)$ ▷ use DFS to find a path from i to l
8: $\delta = \min_{(a,b)\in P} q_{ab}$
9: **if** $i, l \in N^c$ **then**
10: $A^3 = A^3 \cup (i, l)$
11: r^c_{il}**+=**δ
12: **end if**
13: **for** $(a, b) \in P$ **do**
14: q_{ab}**-=**δ
15: **end for**
16: $H = H \setminus \{(a, b) | q_{ab} = 0\}$
17: **end whilereturn** H, r

In step 6, a path from the selected node i to a node l is created using a **depth-first search**, satisfying specific requirements. It's important to note that it **is not**

guaranteed that i and l are Γ-critical, meaning that this path (which always exists) might be discarded.

When a path is discarded, we say that δ capacity has been **lost**. Thus, the maximum flow in the Γ-compact graph is lower than the optimal one. However, the following lemma shows that there is an upper bound to this lost residual capacity.

Lemma 4.3 (Bound on lost capacity in Γ-compact)**.** *Let f_{max} be the maximum flow computed in G, the original network, and let f^* be the flow computed in G^c, the compacted network created by the capacity-transfer(r, H, G^c). Then:*

$$f^* \leq f_{max} \leq f^* + \frac{\Gamma}{16m}$$

meaning that the maximum flow computed in G^c is underestimated by at most $\frac{\Gamma}{16m}$.

Proof. For a path to be discarded, it must start or end at a Γ-compactible node, that is, a node j not adjacent to a medium arc such that:

$$|\Phi(j, r, E^{-ab})| = \left| \sum_{(i,j)\in E^{-ab}} r_{ij} - \sum_{(j,i)\in E^{-ab}} r_{ji} \right| \leq \frac{\Gamma}{16m^2}$$

However, for a non-critical node to be chosen, it must have only incoming or only outgoing arcs, depending on which end of the path we are discussing.

We can estimate that the maximum capacity of a discarded path P_s is:

$$r(P_s) \leq \frac{\Gamma}{16m^2}$$

which represents the maximum residual value achievable by an arc at one of the path's ends.

Given that there can be at most n such paths, we have:

$$n \cdot \frac{\Gamma}{16m^2} \leq m \cdot \frac{\Gamma}{16m^2} = \frac{\Gamma}{16m}$$

Therefore, the maximum capacity lost in creating G^c is precisely $\Gamma/16m$. □

We now need to ensure that a flow computed in G^c, which we will call α-optimal, can be transferred back to the original network G.

Let f' be the flow computed in G^c, and let f represent the transposition of f' in G:

- If $f'_{i,j} > 0$ and $(i, j) \in E(G)$, then $f_{i,j} = f'_{i,j}$, meaning it can be directly applied without modification.
- If $f'_{i,j} > 0$ and $i \Rightarrow j$, it represents the compaction of an abundant path, and to restore the flow, we can use the transitivity matrix.

The most interesting case occurs when we need to transfer the flow from a pseudo-arc of abundant edges back to the paths that generated it. It is important to recall that the capacity of the pseudo-arc is the sum of the residual capacities of the paths that were previously transferred.

Tracking all transferred paths would be too inefficient; however, by using dynamic trees, we can enhance the algorithm previously used to keep a record of all operations performed on the tree. This way, by consulting the record backward, we can reconstruct in time $k \log n$ (where k is the number of operations on the link-cut tree) the capacities transferred during the procedure, thereby adjusting the correct portion of flow for each edge.

Let's now study the adaptation from the perspective of the (S, T)-cut:

Let (S', T') be a cut in $G[r]$, and suppose there are no abundant edges from S to T. A cut (S^c, T^c) in G^c is said to be **induced by** (S', T') if:

$$(S^c, T^c) := (S' \cap N(G^c),\ T' \cap N(G^c))$$

In other words, S^c is the intersection of S' with the set of nodes in G^c, and similarly for T^c.

Conversely, a cut in G^c is said to be induced by one in $G[r]$ if composed as follows:

Observation 4.4. *We observe that if a cut (S', T') is induced by (S^c, T^c), then (S^c, T^c) is also induced by (S', T').*

$$(S', T') \leftarrow (S^c, T^c) \implies (S^c, T^c) \leftarrow (S', T')$$

The reverse is not true, as different cuts on $G[r]$ can induce the same (S^c, T^c).

Lemma 4.4. *Suppose (S', T') is a cut in $G[r]$ and there are no abundant edges from S to T. If (S^c, T^c) is induced by (S', T'), then*

$$r(S^c, T^c) \leq r(S', T') \leq r(S^c, T^c) + \Gamma/16m$$

Proof. We know that the original edges in E^1 contribute equally in both (S^c, T^c) and (S', T'). The abundant edges are absent by hypothesis, leaving only those in E^3.

We divide the paths computed by the *capacity-transfer(r, H, G^c)* into $P \cup Q$, where Q are those that are eventually discarded. From lemma 4.2, we know that transferring capacity does not affect the cut capacity. Therefore, the only edges that can influence the residual capacity are those in Q.

But from lemma 4.3, we know that:

$$\sum_{p \in Q} r(p) \leq \Gamma/16m$$

□

From the previous lemmas, we can assert the following theorem:

Theorem 4.2. *Let y be an α-optimal flow in the Γ-compact network G^c. Let (S^c, T^c) be a cut in G^c such that*

$$r(S^c, T^c) \leq val(y) + \alpha.$$

If (S', T') is the cut induced by (S^c, T^c) in $G[r]$ and y' is the respective flow, then

$$val(y') = val(y).$$

Moreover, y' is said to be α'-optimal, where $\alpha' = \alpha + \Gamma/16m$.
Therefore, $r(S', T') \leq v + \alpha'$.

4.7 Max flow in $O(nm)$ time

In this section, we will demonstrate how it is possible to compute the max flow in time $O(nm)$ when $m = O(n^{1.06})$ (Note: $16/15 = 1.0\bar{6}$). We will also show that the bottleneck of this procedure is due to maintaining the transitive closure of G^{ab}.

Algoritmh 5 *Improve-approx-2(r,S,T)*

```
1: Δ := r(S,T)
2: c = |N^c|
3: if c ≥ m^{9/16} then
4:     Γ = Δ
5:     find a Γ/(8m)-optimal flow in G[r]
6: else if m^{1/3} ≤ c ≤ m^{9/16} then
7:     Γ = Δ
8:     G^c := Γ-compact network
9:     y = Γ/(8m)-optimal flow in G^c
10:    y' = induced(y, G[r])
11:    update(r)
12: else if c < m^{1/3} then
13:    Γ = choseGamma(c, Δ)
14:    G^c := Γ-compact network
15:    y = optimal flow in G^c
16:    y' = induced(y, G[r])
17:    update(r)
18: end if
```

One of the first things we can understand by observing this algorithm is the complexity required for the creation of the Γ-compact network, which we state in the following theorem.

Theorem 4.3 (Constructing a compact network)**.** *Suppose the algorithm dynamically maintains the transitive closure of the abundance graph and that the Γ parameter is provided initially. Then, the algorithm takes $O(m^{9/8})$ time to create the compact graph G^c.*

Proof. To contract the connected abundant components, as well as cycles, takes $O(m)$ time. Regarding the compacted graph:

- The edges in E^1 can be calculated in $O(m)$;
- The edges in E^3 can be calculated in $O(m \log m)$ using dynamic trees;
- The most complex to calculate are the abundant edges in E^2, which are determined based on the transitive closure that requires a cost of $|N^c|^2$ to maintain.

However, the algorithm constructs G^c only if the number of Γ-critical nodes is less than $m^{9/16}$, thus the cost becomes

$$O((m^{9/16})^2) = O(m^{9/8}).$$

□

To demonstrate that the overall complexity of the algorithm is indeed as stated initially, we need to establish bounds both on the actions performed and the objects analyzed. The first thing to declare is that the total number of all Γ-critical nodes analyzed during the various phases is $O(m)$.

Theorem 4.4 (Max critical node in $O(m)$)**.** *Suppose that each improvement phase meets the required conditions; then the* Γ*-critical nodes calculated during the iterations total* $O(m)$*.*

Proof. For a node j to be Γ-critical, it must be adjacent to a Γmedium edge, or it must not have adjacent Γmedium edges but have $|\Phi(j, r, E^{-ab})| > \Gamma/(16m^2)$, i.e., it must be Γ-special.

First, consider the nodes adjacent to a Γ-medium edge:
Claim: An edge can have medium Γ-capacity for at most 3 consecutive phases.

Proof: Let (i, j) be an edge with Γ-medium capacity, then $u_{ij} + u_{ji} \geq \Gamma/64m^3$. Given that in each phase $\Delta' = \frac{\Delta}{8m}$, in the immediate next phase, we have

$$u_{ij} + u_{ji} \geq \Gamma/64m^3 \geq \Delta'/8m^2 = \Delta''/m = 8\Delta'''$$

Thus, after 3 phases, $u_{ij} + u_{ji} \geq 8\Delta$, implying that either (i, j) or (j, i) has become abundant, and the edge is no longer Γ-medium. □

For the other Γ-special edges:
Claim: Let Γ be the compactness parameter of a given Δ-improvement phase, and let j be a Γ-special node. If Δ^* is the bound 4 phases after Δ, then there exists a node k such that

$$r_{jk} \geq 2\Delta^* \quad \text{and} \quad r_{kj} \geq 2\Delta^*$$

implying that (j, k) (and also (k, j)) is *doubly-abundant*, and thus will be contracted.

Proof:
First, define v^* as the flow in phase Δ^* such that $r^* = r_{ij} - v_{ij} + v_{ji}$. From lemma 4.1, we know that every Δ-abundant edge will also be Δ^*-abundant. Furthermore,

$$r^*_{ij} > \Gamma/64m^3 \implies r^*_{ij} > 8\Delta^*$$

Suppose there exists an abundant edge (j,k) with $v^*_{jk} > \Gamma/64m^3$; then for the opposite edge (k,j), we have:

$$r^*_{k,j} = r_{kj} - v^*_{kj} + r^*_{jk} > 8\Delta^*$$

Thus, the opposite is also Δ^*-abundant, and the nodes j and k will be contracted.

It remains to check the case where a node j is Γ-special without having abundant edges with flow greater than $\Gamma/64m^3$.
We know that:

$$|\Phi(j, r, E^{-ab})| = |\hat{r}_{out}(j) - \hat{r_{in}}(j)| > \Gamma/(16m^2)$$

Consider the case where $\hat{r}_{out}(j) - \hat{r_{in}}(j) > \Gamma/(16m^2)$ (the other case is similar). We have:

$$\sum_{j:(j,k)\in E^{-ab}} y^*_{jk} \leq \sum_{j:(j,k)\in E} y^*_{jk} = \sum_{j:(i,j)\in E} y^*_{ij}$$

due to flow conservation. Furthermore,

$$\sum_{j:(i,j)\in E} y^*_{ij} < \sum_{j:(i,j)\in E^{-ab}} y^*_{ij} + \sum_{j:(i,j)\in E^{ab}} y^*_{ij} + m\Gamma/64m^3$$

But since we have assumed that no abundant edge has flow greater than $\Gamma/64m^3$,

$$\begin{aligned} &< \hat{r}_{in}(j) + 2m\Gamma/64m^3 \\ &< (\hat{r}_{out}(j) - \Gamma/16m^2) + \Gamma/32m^2 \\ &< \hat{r}_{out}(j) - \Gamma/32m^2 \\ &= \textstyle\sum_{j:(j,k)\in E^{-ab}} r_{jk} - \Gamma/32m^2 \end{aligned}$$

Thus, there must exist some edge for which

$$y^*_{jk} < r_{jk} - \Gamma/32m^3 \implies r^*_{jk} \geq r_{jk} - y^*_{jk} > \Gamma/32m^3 > 16\Delta^*$$

Therefore, there exists some (j,k) non-abundant edge in the Δphase that becomes abundant in the phase Δ^*, and since (k,j) is also abundant, the cycle will be contracted. □

We have shown through the two claims that all Γ-critical nodes in a phase cannot remain Γ-critical for more than a constant number of consecutive phases. The fact that they have a "limited duration" and are at most $O(m)$ proves the statement. □

Knowing the number of nodes to analyze, the next step is to estimate the number of **improvement phases** required to calculate the maximum flow. However, it is necessary first to understand how the Γ parameter is chosen.

Lemma 4.5 (Γ Parameter)**.** *The* Γ *parameter can be chosen in* $O(m + n \log n)$ *time.*

Proof. We give a procedure that does this:

1. For each node j, calculate the largest Γ' value for which j is Γ'-critical (*time required* $O(m)$).
2. Order the nodes j by their Γ' value (*time required* $O(n \log n)$).
3. Choose the Γ value such that there are at most $m^{1/3}$ nodes j with $\Gamma'(j) \geq \Gamma$ (*time required* $O(1)$).

□

We now have all the tools to calculate the number of improvement phases:

Lemma 4.6. *The number of improvement phases is* $O(m^{2/3})$.

Proof. From theorem 4.4, we know that the number of Γ-critical nodes analyzed is $O(m)$, and from lemma 4.5, we know that in each improvement phase at least $m^{1/3}$ nodes are analyzed.

When we demonstrated that the number of nodes was $O(m)$, the proof relied on the fact that the nodes had a "deadline" and would be contracted in at most 3 or 4 consecutive phases. Thus, all at least $m^{1/3}$ nodes analyzed in one phase will be "consumed" in $O(1)$ phases.

Consequently, the number of phases required to "consume" them all is $O(m^{2/3})$. □

From the following lemmas, we can derive the total time required to create all the G^c.

Lemma 4.7. *The total time to create all the compact networks is* $O(nm + m^{43/24})$.

Proof. We know that the Γ parameter requires time $O(m + n \log n)$.
We know that creating a compact network takes $O(m^{9/8})$ time.
Since the number of phases is $O(m^{2/3})$,
putting everything together, we obtain:

$$O(mn + m^{43/24})$$

□

Finding a flow that is α-optimal means finding a flow that is at most α less than the maximum capacity flow. We have seen that if we search for the maximum flow in the Γ-compact network G^c and then transfer it to the original network G, this is already $\Gamma/16m$-optimal.

However, during the algorithm, we aim to obtain this approximation directly in G. If we recall the functioning of the Goldberg-Rao algorithm, we can see that the

algorithm terminates when its estimate of the maximum flow is less than 1. By intervening on this maximum flow estimate, it is possible to terminate the algorithm before reaching the optimal flow, resulting in a gap of at most a value of α of our choosing.

Now let's reason about the cost T of a phase of the Goldberg-Rao algorithm, knowing that we are executing it on a graph with C nodes and $O(C^2)$ edges.

$$\Lambda = O(C^{2/3}), \quad T = \tilde{O}(C^{2/3} \cdot C^2) = \tilde{O}(C^{8/3})$$

We can now evaluate the cost of the *Improve-approx-2(r,S,T)* procedure.

Moreover, we can note that if we execute the Goldberg-Rao algorithm on a total of $O(m)$ nodes while performing a maximum of $\log U$ phases, the average number of nodes in each improvement phase is

$$C = O\left(\frac{m}{\log U}\right)$$

This is the reason why we construct the compact graph only if $C \leq m^{9/16}$. In fact, if the Goldberg algorithm is polynomial, then if $\log U \leq m^{7/16}$:

$$C \geq m^{9/16} \implies \frac{m}{\log U} \geq m^{9/16} \implies \log U \leq \frac{m}{m^{9/16}} = m^{7/16}$$

Lemma 4.8 (Time of improve-max)**.** *The time to compute the optimal flow using the improve-approx-2 procedure is*

$$O(m^{31/16} \log^2 m)$$

Proof. Given that a total of $O(m)$ Γ-critical nodes are computed, I will calculate the cost of the procedure for a single node rather than for the number of phases:

Let T be the time required to find an α-optimal flow.
We know that if $C \geq m^{9/16}$, we can find an optimal flow with $T = O(m^{3/2} \log^2 n)$ by executing $\log n$ phases of the Goldberg-Rao algorithm on the original graph. When scaled to the number of Γ-critical nodes, we have $T/C = m^{15/16} \log^2 n$.

If instead $m^{1/3} \leq C < m^{9/16}$, we work in the compact graph and seek the maximum flow, executing $\log n$ phases of the Goldberg-Rao algorithm, thus $T = O(C^{8/3} \log n)$. Therefore,

$$T/C = O(m^{5/3} \log n = m^{15/16})$$

.

Finally, if $C < m^{1/3}$, the small number of edges leads the cost to be $T = O(C^3)$, resulting in $T/C = O(C^2) = O(m^{2/3})$.

We can assert that in every case, the cost of the procedure for each node is $O(m^{15/16} \log^2 n)$. Multiplying this result by the number of Γ-critical nodes across all increments yields

$$O(m \cdot m^{15/16} \log^2 n) = O(m^{31/16} \log^2 n)$$

. □

Lemma 4.9. *The total time to transform all the flows computed on G^c into flows on the residual graph is*

$$O(nm + m^{5/3} \log n)$$

.

Proof. Let G^c be the compact graph derived from $G[r]$ (the residual graph), and let y^c be the flow in G^c while y is the flow induced on the residual graph.

The edges of G^c are divided into three categories: $E^c = E^1 \cup E^2 \cup E^3$, corresponding to original edges, abundant pseudo-edges, and anti-abundant pseudo-edges.

For any edge (i, j) such that $y^c_{ij} > 0$, we distinguish the three cases:

1. If $(i, j) \in E^1 \implies y_{ij} = y^c_{ij}$.
2. If $(i, j) \in E^2$, we know that to reconstruct the abundant path we need $O(|P|) = O(n)$. Furthermore, by utilizing dynamic trees, we can keep the number of edges with positive flow in E^2 below the value C at a cost of $O(m \log n)$, repeated for $O(m^{2/3})$ phases, resulting in $O(m^{5/3} \log n)$. In this way, all abundant paths are restored in $O(nm)$. We always remember that the cost of dynamically maintaining the transitive closure is $O(nm)$.
3. Regarding the edges in E^3, we again have to resort to dynamic trees to reconstruct the anti-abundant paths that we contracted. Specifically, it is not possible to keep track of all paths efficiently; however, we can maintain a record of all operations performed on the graph to retrace and restore the old paths. This procedure also has a total cost of $O(m \log n \cdot m^{2/3})$.

We conclude that the total time to induce the flow in G^c to $G[r]$ for all $m^{2/3}$ phases is

$$O(nm + m^{5/3} \log n)$$

□

From the previous lemmas, we can deduce the following theorem:

Theorem 4.5 (Max flow in $O(nm)$)**.** *If the flow in each improvement phase is computed using the improve-approx-2 procedure, then the time to find the maximum flow is*

$$O(nm + m^{31/16} \log^2 n)$$

If $m = O(n^{1.06})$, the running time is $O(nm)$.

Conclusions

In this thesis, we have outlined all the key intermediate steps that lead from the Edmonds-Karp algorithm to Orlin's algorithm. A crucial point that emerged is the strong interconnection between the various algorithms.

Although each algorithm significantly enhances the efficiency of the previous one, the core principles behind the solutions are not radically different. For instance, we observed that Dinitz's algorithm modifies only the sequence of the augmenting phase and the BFS computation found in the Edmonds-Karp algorithm. In turn, the Goldberg-Rao algorithm builds upon Dinitz's work, executing the augmentation phases on a more contracted graph.

Orlin's algorithm further capitalizes on the Goldberg-Rao method, using it in its original form to compute an α-optimal flow on a specially compacted graph, designed to reduce computational costs even further.

For graphs that are even sparser ($n = O(m)$), Orlin proposed an alternative approach. In this version, given the number of edges, the procedure for dynamic transitive closure (one of the main bottlenecks of the algorithm) can be replaced with a more efficient method, thereby reducing the computational cost by a logarithmic factor, bringing it down to $O(n^2/\log n)$.

Further advances in this field were made in 2018, when Orlin, together with Xiao-Yue Gong, introduced an algorithm[OG19] that strictly outperforms the King et al. algorithm, particularly in cases where $m = o(n \log n)$, surpassing it by a factor of $\log \log n$.

Acknowledgements

We've reached the end, not just of this thesis but of the entire journey. Now that I've arrived, I feel I have some thank-yous to give.

First and foremost, I must thank my supervisor, who, despite numerous commitments, guided me to the end, allowing me to complete this work. I challenged myself and pushed beyond what I thought I was capable of, and I owe this achievement to him.

There is no knowledge I've gained in these three years that I couldn't have learned on my own from some online article or textbook. For this reason, a huge thank you goes to my classmates. You are the true added value of this university. Genuine companions, always willing to help others out of pure altruism. There is no shame or exaggeration in saying that without you, I wouldn't have gotten to where I am now. The exchange and interaction we've had has been the unique and indispensable resource that enriched this journey. Together, we encouraged each other and forged our way along a path that was often uphill but, in the end, always rewarded the effort.

A special thank you goes to the friends who have stood by me during the writing of this thesis and throughout this past year. You have given me so much, and I hope I can give back as much as I have received.

Thanks to my lifelong friends, who have stayed close to me even when I've been difficult or had my head elsewhere. Your support, lightness, and spontaneity remind me that we are not made of just flesh and rationality.

I have two more thank yous to give: The first goes to my girlfriend, Sofia.
I will never find the right words to thank you for everything. Thank you for every moment we've shared; seeing things from your perspective continues to improve and surprise me. Lastly, I must thank my parents, who push me every day to give my best because they believe I can do far more and better than I think I can (and, in the end, maybe they are right).

Thanks again to everyone.

Bibliography

[Din06] Yefim Dinitz. "Dinitz' Algorithm: The Original Version and Even's Version". In: *Theoretical Computer Science: Essays in Memory of Shimon Even*. Ed. by Oded Goldreich, Arnold L. Rosenberg, and Alan L. Selman. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 218–240. ISBN: 978-3-540-32881-0. DOI: 10.1007/11685654_10. URL: https://doi.org/10.1007/11685654_10.

[EK72] Jack Edmonds and Richard M. Karp. "Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems". In: *J. ACM* 19.2 (Apr. 1972), pp. 248–264. ISSN: 0004-5411. DOI: 10.1145/321694.321699. URL: https://doi.org/10.1145/321694.321699.

[FF56] Lester Randolph Ford and Delbert Ray Fulkerson. "Maximal Flow Through a Network". In: *Canadian Journal of Mathematics* 8 (1956), pp. 399–404. URL: https://api.semanticscholar.org/CorpusID:16109790.

[GR98] Andrew V. Goldberg and Satish Rao. "Beyond the flow decomposition barrier". In: *J. ACM* 45.5 (Sept. 1998), pp. 783–797. ISSN: 0004-5411. DOI: 10.1145/290179.290181. URL: https://doi.org/10.1145/290179.290181.

[Ita86] G.F. Italiano. "Amortized efficiency of a path retrieval data structure". In: *Theoretical Computer Science* 48 (1986), pp. 273–281. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(86)90098-8. URL: https://www.sciencedirect.com/science/article/pii/0304397586900988.

[KRT92] Valerie King, Satish Rao, and Robert Tarjan. "A Faster Deterministic Maximum Flow Algorithm". In: Jan. 1992, pp. 157–164.

[OG19] James B. Orlin and Xiao-Yue Gong. *A Fast Max Flow Algorithm*. 2019. arXiv: 1910.04848 [cs.DS]. URL: https://arxiv.org/abs/1910.04848.

[Orl13] James B. Orlin. "Max flows in O(nm) time, or better". In: *Proceedings of the Forty-Fifth Annual ACM Symposium on Theory of Computing*. STOC '13. Palo Alto, California, USA: Association for Computing Machinery, 2013, pp. 765–774. ISBN: 9781450320290. DOI: 10.1145/2488608.2488705. URL: https://doi.org/10.1145/2488608.2488705.

[SE83] Daniel D. Sleator and Robert Endre Tarjan. “A data structure for dynamic trees”. English (US). In: *Journal of Computer and System Sciences* 26.3 (June 1983), pp. 362–391. ISSN: 0022-0000. DOI: 10.1016/0022-0000(83)90006-5.