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

TO DO

Contents

Introduction

Ciò che segue è una relazione che ha lo scopo di analizzare la soluzione proposta nell'algoritmo di Orlin per il calcolo del **max flow in un Network**. Dato che per comprendere a pieno il suo funzionamento è necessario conoscere anche alcuni algoritmi precedenti, anche questi vengono spiegati nella relazione. Gli algoritmi sono presentati in ordine cronologico a partire dall'algoritmo di Dinics, continuando con Goldberg-Rao e giungendo alla soluzione più recente e efficace di Orlin. Prima di iniziare con le soluzioni però, è presente un capitolo che spiega alcune nozioni preliminari sulla teoria dei grafi necessarie per comprendere il funzionamento delle soluzioni.

Chapter 1

Preliminary notions

1.1 Network e flow

Before starting, we need to establish some essential preliminary notions. In particular, we need to define what a network is and what it is composed of.

Definition 1.1 (Network)**.** A network is a structure composed of a graph G such that $G = (N, E)$ such that:

- N = the set of nodes
- E = the set of edges such that $(i, j) \in E \implies i, j \in N$
- $n = |N|$
- $m = |E|$

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}

In each network 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.

Definition 1.2 (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

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

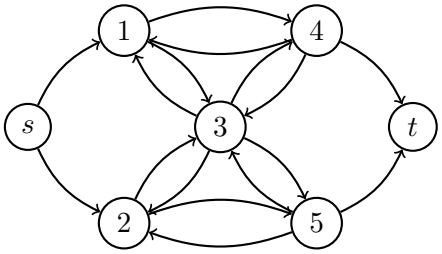


Figure 1.1. A classic example of a network

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

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

Observation 1.1. *Node s has no incoming arcs just as node t has no outgoing arcs.*

Definition 1.3 (Flow)**.** We define the flow as the function $f : E \to \mathbb{R}_+ \cup \{0\}$ which satisfies the **flow conservation role**:

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

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

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

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)

Definition 1.4 (Residual capacity)**.** The residual capacity of an edge (i,j) means the amount of flow we can route in 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 1.5 (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 1.6 (s-t Cut)**.** Given a network G we define an *s-t* cut on G as a partition into two subsets (T, S) 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.*

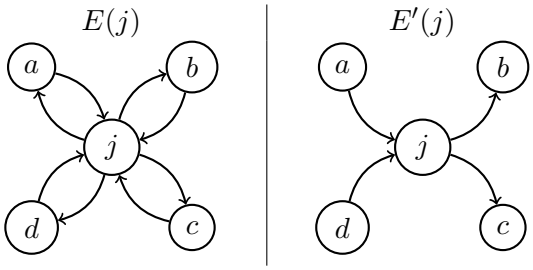
Proof. Omitted.

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 1.7. (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-symmetriy lemma)**.** *Given $E'(j)$ an anti-symmetric subset of $E(j)$ and a flow f on G with $r = r[f]$ then is true 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 Decomposition e transferring del flow

Definition 1.8 (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

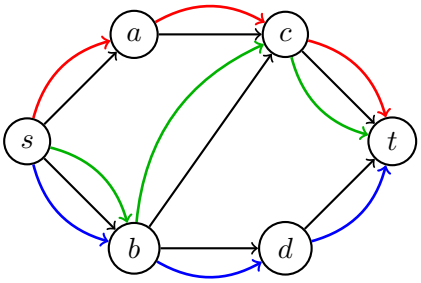
$$P_1, ..., P_k \quad where\ k < m$$

To each path P_i corresponds a value $\phi_i \in \mathbb{N}^+ | \phi_i > 0$ that 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. $val(f) = \sum_{i=1}^{k} \phi_i$

An intuitive observation is that the maximum number of decompositions of any flow is m.



Example of a decomposed flow

Once we establish what decomposing a flow means, we can talk about capacity transfer

Definition 1.9 (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

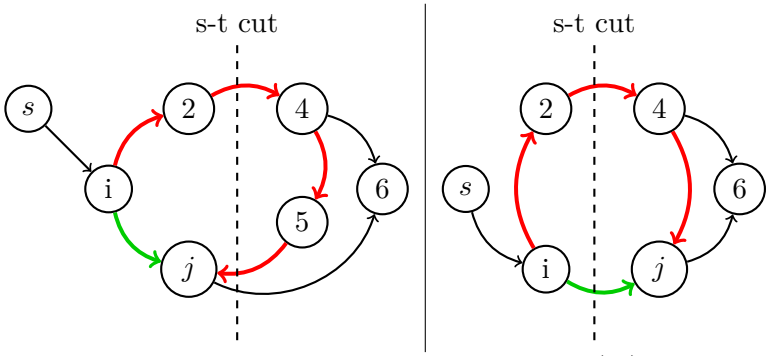
Lemma 1.3 (Capacity tranfer lemma)**.** *Let P be path in G from node i to node j and let (S, T) be an s-t cut. If we transfer delta capacity from the path P to de edge (i, j) and r and r' are respectively the residual capacity of the network before and after the transfer then is true that:*

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

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 $l \in S \wedge k \in T$ we estimate

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

□



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

At the end of this consideration, we can deduce that transferring capacity from a path to an edge doesn't increase the maximum routable flow in a network

1.3 Distance based

We usually think about graphs composed of nodes linked by edges and measure the distance between two nodes i and j as the sum of the edges on the shortest path that brings from i to j. That is true just because we don't specify the length of an edge then we assume that it is one. Instead, we can specify the length of each edge and still divide the nodes by labels, i.e. by the distance from a specific node. But in doing this we have to pay attention to some rules that allow us to achieve our goal. First of all we need to establish what a valid distance labeling is:

Definition 1.10 (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 said **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. *$d(i)$ represents the lower bound of the length of the shortest path from $i \to t$ in the residual graph*
2. *$d(s) \geq n \implies \nexists\ p\ path \in G[r] \mid p = s \to t$*

Another point of view of the second property that a valid distance label has to respect ($d(i) \leq d(j) + l((i, j))$) is that:

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

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

Definition 1.11 (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 called **Admissible arc** if it satisfies:

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

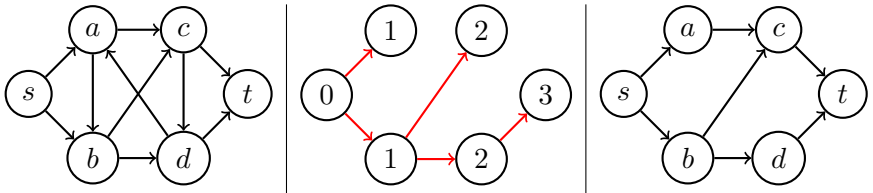
i.e.

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

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 l'admissible graph of G_f then*

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



grafo originale | arborescenza | Admissible graph

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

Definition 1.12 (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 Max flow

To find the maximum flow in the network, we can use the **Edmonds-Karp algorithm**.

This algorithm finds the 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 other $O(n)$ time tu augments 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

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.

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.

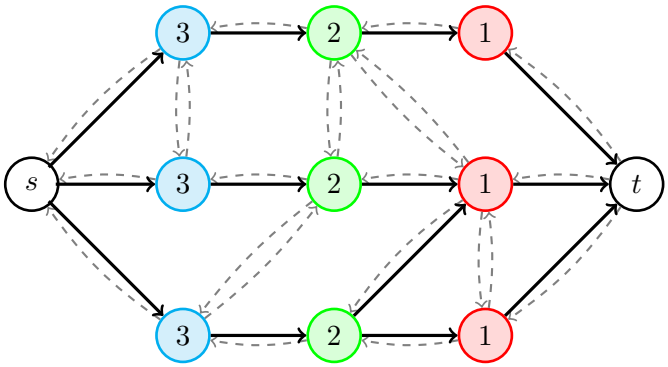
Observation 2.1. *Max flow $\implies$ blocking flow The max flow is a blocking flow, but the reverse implication is not true.*

We can now define an algorithm, but first, some useful notes are necessary for its understanding:

- 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, updated to the moment it is referenced.
- The BFS takes the residual graph as input and thus does not consider saturated edges.

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

$f_{ij} = 0 \forall (i,j) \in E(G)$
$d = BFS(G_f)$
$A = A(d, f)$
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 dashed).

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 $O(n^2m)$.

Altri link utili

1. Lecture from MIT
2. Wikipedia

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.

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 Idea

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 that were 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. In fact, 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**. In fact, 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 Stop condition

To terminate its execution, the algorithm estimates the difference between the maximum achievable flow (which from now on 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 Stimare il 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

There are two other issues that 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. It is evident that 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' be 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 lebeling 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 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:

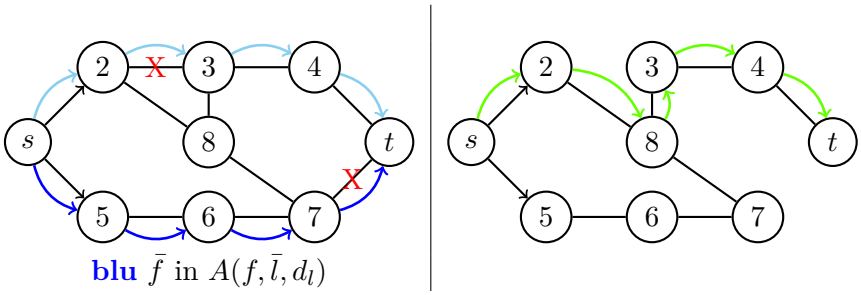
$$l'(\Gamma) = d_l(s) + c(\Gamma)$$

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.

Table 3.1. graphic example to better visualize what was declared



blu $\bar{f}$ in $A(f, \bar{l}, d_l)$

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 Costo computazionale

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 a number of 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;
- $O(m \log(n^2/m))$ 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 had 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 algorithm. The next chapter analyzes the solution proposed in 2013 by James B. Orlin, 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 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 Idea

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 larger 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 Fase di incremento

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 e grafo di abbondanza

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, 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.

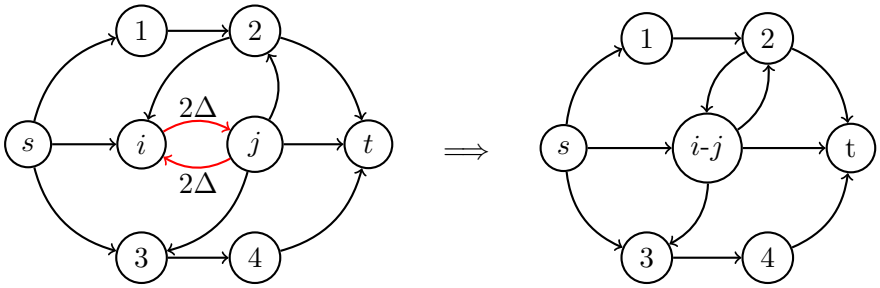
The transitive closure can be maintained in time $O(nm)$ using Italiano's algorithm. In this way, it is always possible to reconstruct an abundant path P in $O(|P|)$ (we will see later that contracting the graph does not prevent this nor does it alter its cost).

4.4 Contractions of abundant graph

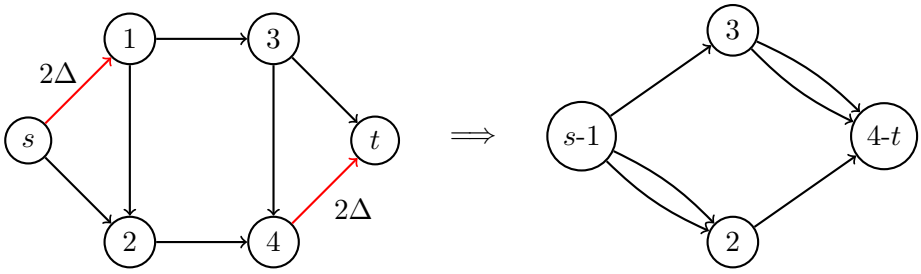
Let's now see how to exploit the abundant graph to contract the graph on which we calculate the max-flow and make the algorithm more efficient.

We analyze three different examples of contractions:

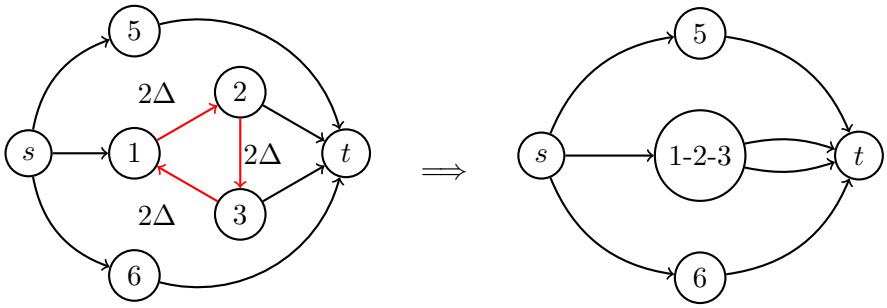
Suppose there are two nodes i and j such that $r_{ij} \geq 2\Delta$ and $r_{ji} \geq 2\Delta$. We can contract the two nodes into a single one, preserving the edges of both.



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.*

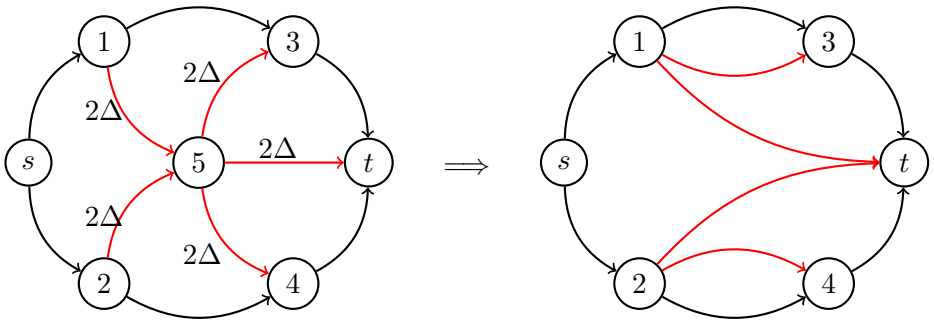
However, this violation is minor, at most 2Δ units; thus, as shown by Goldberg and Rao, the contraction, expansion, and adjustment for flow conservation can be performed in $O(m)$ time.

4.5 How to compact 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 on 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

Il grafo sc-compact non è abbastanza compattato per raggiungere il costo desiderato. Per compattarlo ulteriormente dovremmo utilizzare un parametro Γper scegliere quali nodi compattare e da quali archi trasferire capacità residua. La scelta del parametro Γ verrà mostrata in seguito. Prima di proseguire è importante distinguere diversi tipi di archi.

Definition 4.4 (Classificazioni di capacità)**.** Un arco (i, j) rispetto a Γ ha:

1. **small capacity** se $u_{ij} + u_{ji} < \Gamma/(64m^3)$
2. **medium capacity** se $\Gamma/(64m^3) \leq u_{ij} + u_{ji} \land$ $r_{ij} < 2\Delta \land r_{ji} < 2\Delta$
3. **abundant capacity** se $r_{ij} \geq 2\Delta$
4. **antiabundant capacity** se $(j, i) \in E^{ab} \lor$ (i, j) è un arco esterno non abbondante.

Dove E^{ab} e E^{-ab} rappresentano rispettivamente l'insieme degli archi abbondanti e anti abbondanti all'inizio dell'improvement phase.

Observation 4.3. *Dato che abbiamo contratto i cicli abbondanti se $(i, j) \in E^{ab} \implies (j, i) \notin E^{ab}$*

Un altro strumento necessario per decidere quali nodi compattare è la funzione *potenziale*

Definition 4.5 (Potential function)**.** Dato un nodo $j \in N$ una funzione di capacità residua r e un sottoinsieme di archi adiacenti a j $\tilde{E}$ possiamo definire la funzione potenziale come:

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

È secondo questi parametri che in ogni improvement phase vengono distinti i nodi che si possono compattare da quelli che non possono essere compattati. Queste due distinzioni prendono il nome rispettivamente di nodi Γ-compactible e nodi Γ-critical.

Definition 4.6 (Γ-critical e Γ-compactible)**.** Un nodo j si dice **Γ-critical** se è adiacente almeno ad un arco Γ-medio oppure se $|\Phi(j, r, E^{-ab})| > \Gamma/(16m^2)$.

Se un nodo non è Γ-critical allora si dice **Γ-compactible**.

Dato un network G definiamo il Γcompact network di G come

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

Dove N^c sono tutti e soli i nodi Γ-critical mentre E^c l'insieme di archi che definiremo in seguito.

Per costruire il Δ-compact network vengono iterativamente trasferite unità di capacità residue di vari path a pseudo archi. l'idea è quella di sottrarre capacità a dei percorsi che collegano due nodi $i, j \in N$ per passarla all'arco (o pseudo arco) (i, j) e poter ulteriormente compattare il grafo. Ovviamente però questi pseudo archi sono solo parte di quelli che compongono E^c che potremmo definire come

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

$E^1 = \{(i, j)|i, j \in N^c \wedge (i, j) \in E(G)\}$ dunque gli archi originali che collegano due nodi Γ-critici
$E^2 = \{(i, j)|i, j \in N^c \wedge i \implies j\}$ ovvero gli archi abbondanti

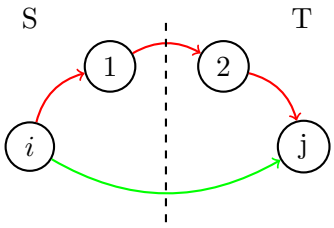
Il seguente lemma mostra come se scelti secondo un appropriato criterio, il trasferimento di flusso non riduce la capacità di nessun (S,T) cut e dunque preserva il max flow calcolabile.

Lemma 4.2 (label = ftsafe)**.** *Flow transfer safety Sia (S, T) un s-t cut in G con $r(S, T) \leq \Delta$, e sia $A' = E^{-ab}$.*
Supponiamo che esista $P \subseteq A'$ un path da $i \to j$ e che $(i, j) \in A'$.
Se r' è la nuova funzione di capacità residua ottenuta spostando δ unità di capacità residua da P a (i, j), allora possiamo affermare che:

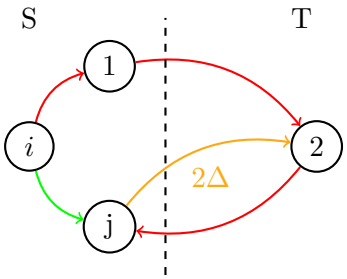
1. $\forall k \in N(G),\ \Phi(k, r', A') = \Phi(k, r, A')$
2. $r'(S, T) = r(S, T)$

Proof. 1. Il primo punto è intuitivo in quanto per ogni nodo in P diverso da i e j sto sottraendo la stessa capacità residua sia in entrata che in uscita, mentre nei nodi i e j le sommatorie di Φ rimangono identiche.

2. Il secondo punto è banale se $|P| = 1$ dunque consideriamo $|P| \geq 2$. Definiamo $P = p_1, ..., p_k$, $p_1 \in S$ e almeno un $p_q \in T$.
   Dato che abbiamo stabilito che $r(S,T) \leq \Delta$ e che $P \subseteq A'$ ci troviamo sicuramente in una situazione di questo tipo:



   In quanto se un arco di P passasse da T a S violerebbe $r(S,T) \leq \Delta$ dato che $\forall (a,b) \in A'(\Delta),\ r_{ab} > 2\Delta \wedge r_{ba} \geq 2\Delta$, avremmo che



   Una volta appurato ciò risulta evidente che il trasferimento di capacità residua non influenza la capacità residua del taglio. □

□

Notiamo dunque che:

Definition 4.7. Transferrable residual capacity Per poter trasferire δ capacità da un path P da $i \to j$ all'arco (i,j) è necessario che:

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

   In oltre quando creeremo il Γ compact network saranno essenziali anche i seguenti requisiti

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

La capacità che viene trasferita da è $\delta = r(P) = \min_{(a,b)\in P} r(a,b)$, dunque ogni volta almeno un arco anti-abbondante viene saturato. Se esiste un path $P \subseteq A'$ *da* $i \to j$ ma $(i,j) \notin E(G)$ allora viene creato come pseudo arco. Sono proprio questi Pseudo

Algoritmh 4 *Improve-approx-2(r,S,T)*

1: sia $G^c := \{n | n \in N(G) \wedge \Gamma\text{-critical}(n)\}$
2: sia $H := \{(i,j) | (i,j) \in E^{-ab} \wedge i \notin G^c \vee j \notin G^c\}$
3: $\forall (i,j) \in H,\ q_{ij} = r_{ij}$
4: **while** H $\neq \varnothing$ **do**
5: seleziona $i \in H | \nexists (j,i) \in H$:
6: P $= DFS(i,l)$ s.t $l \in N^c \vee \nexists (l,k) \in H$ ▷ usa la DFS per un path da i a l
7: sia $\delta = \min_{(a,b)\in P} q_{ab}$
8: **if** $i, l \in N^c$ **then**
9: $A^3 = A^3 \cup (i,l); r^c_{il} += \delta$
10: **end if**
11: $\forall (a,b) \in P,\ q_{ab} -= \delta$
12: $H = H - \{(a,b) | q_{ab} = 0\}$
13: **end while**

archi antiabbondanti che formeranno E^3. Analizziamo ora la procedura per trasferire tutte le capacità residue necessarie a formare E^3 e restituirne l'insieme di archi con le relative capacità residue $r^c_{ij} \forall (i,j) \in E^3$

Nel passaggio 6 viene creato, utilizzando una *deep first search* un percorso dal nodo scelto i fino ad uno l che soddisfi certi requisiti. Da notare con attenzione che **non è garantito** che i e l siano Γ-critical e dunque è possibile che tale percorso (che esiste sempre) venga scartato.

Quando un percorso viene scartato si dice che δ capacità è stata **persa**. Dunque il max flow nel grafo Γ-compact è inferiore a quello ottimale, tuttavia il seguente lemma mostra che esiste un bound a questa capacità residua *persa*.

Lemma 4.3 (Bound to Γ-compact lose capacity)**.** *Sia f_{max} il max-flow calcolato in G, il network originale, e f^* quello calcolato in G^c, il network compattato creato dalla* **??***. Allora si ha che:*

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

ovvero il flusso massimo calcolato in G^c è sottostimato di al più $\Gamma/16m$.

Proof. Per essere scartato, un percorso deve iniziare o terminare in un nodo Γ-compactible ovvero un nodo j non adiacente ad un arco medio tale che:

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

Tuttavia un nodo non critical per essere scelto deve avere solo archi entranti o solo archi uscenti a seconda di quale estremo del path stiamo parlando. Possiamo quindi stimare che la capacità massima di un certo path P_s scartato sia $r(P_s) \leq \Gamma/16m^2$ ovvero il valore residuo massimo raggiungibile da un arco estremo a P_s. Dato quindi che possono esistere al massimo n di questi percorsi allora abbiamo che:

$$n \cdot \Gamma/16m^2 \leq m \cdot \Gamma/16m^2 = \Gamma/16m$$

Dunque la massima capacità che viene persa nella creazione di G^c è proprio $\Gamma/16m$. □

Dobbiamo ora assicurarci che un flow calcolato in G^c che chiameremo α-ottimale, sia trasferibile nel Network originale G.

Sia f' il flow calcolato in G^c e rappresentiamo con f la trasposizione di f' in G:
Se $f'_{i,j} > 0 \wedge (i,j) \in E(G) \implies f_{i,j} = f'_{i,j}$ basta riportarlo così com'è. se $f'_{i,j} > 0 \wedge i \Rightarrow j \implies$ si tratta del compattamento di un path abbondante e per ripristinare il flusso basta usare la matrice di transitività. Il caso più interessante rimane quello che si verifica quando dobbiamo trasporre il flusso da uno pseudo-arco di archi abbondanti ai path che lo hanno generato. Infatti è importante ricordare che lo la capacità dello pseudo-arco è la somma delle capacità residue dei path che sono strati trasferiti in precedenza. Tenere traccia di tutti i path trasferiti risulterebbe troppo inefficiente, tuttavia utilizzando gli alberi dinamici possiamo potenziare l'algoritmo precedentemente utilizzato per fare in modo di mantenere un record con tutte le operazioni effettuate sull'albero. In questo modo è possibile, consultando il record a ritroso, ricostruire in tempo $k \log n$ (dove k è il numero di operazioni sul link-cut tree) le capacità trasferite dalla procedura in maniera sequenziale, potendo così adattare la giusta porzione di flusso in ogni arco.

Studiamo ora l'adattamento dal punto di vista dell'(S,T)-*cut*:
Sia (S',T') un cut in $G[r]$ e supponiamo che non esistano archi abbondanti da S a T, un taglio (S^c,T^c) in G^c di dice **indotto da** (S',T') se:

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

Viceversa un taglio in G^c si dice indotto da uno in $G[r]$ se composto come segue:

$$S' := \{n | n \in S^c \vee \exists m \in S^c,\ m \implies n\}$$
$$T' = N(G) \setminus S'$$

Observation 4.4. *Osserviamo che se un (S',T') è indotto da (S^c,T^c) allora (S^c,T^c) è indotto da (S',T').*

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

Non è vero il contrario in quanto diversi cut su $G[r]$ possono indurre lo stesso (S^c,T^c).

Lemma 4.4. *Supponiamo che (S',T') sia un cut in $G[r]$ e che non esistano archi abbondanti da S a T. Se (S^c,T^c) è indotto da (S',T') allora*

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

Proof. Sappiamo che gli archi originali in E^1 contribuiscono in egual misura sia in (S^c,T^c) che in (S',T'), quelli abbondanti non sono presenti per ipotesi e dunque rimangono solo quelli in E^3. dividiamo path calcolati dalla **??** come $P \cup Q$ dove Q sono quelli che alla fine vengono scartati. Dal lemma **??** sappiamo che trasferire

la capacità non influenza la capacità del taglio. Quindi gli unici archi che possono influenzare la capacità residua restano quelli in Q. Ma sappiamo dal lemma 4.3 che:

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

□ □

Dai precedenti lemmi possiamo quindi giungere all'asserzione del seguente teorema

Theorem 4.2. *Sia y un α-optimal flow nel Γ-compact network G^c. Sia (S^c, T^c) un taglio in G^c tale che*

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

Se (S', T') è il taglio indotto da (S^c, T^c) in $G[r]$ e y' il rispettivo flow allora

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

e y' si dice α'-ottimale dove $\alpha' = \alpha + \Gamma/16m$.
Dunque $r(S', T') \leq v + \alpha'$

4.7 Max flow in $O(nm)$ time

Mostreremo in questa sezione come è possibile calcolare il max flow in tempo $O(nm)$ quando $m = O(n^{1.06})$. Mostreremo anche che il bottleneck di questa procedura è dovuto al mantenimento della chiusura transitiva di 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 Γ = choseGamma(c, Δ)
13:     G^c := Γ-compact network
14:     y = optimal flow in G^c
15:     y' = induced(y, G[r])
16:     update(r)
17: end if
```

Una delle prime cose che possiamo capire osservando questo algoritmo è la complessità richiesta per la creazione del network Γ-compact che enunciamo nel seguente teorema.

Theorem 4.3 (label = tgcomp)**.** *Costrutire un compact network Supponiamo che l'algoritmo mantenga dinamicamente la chiusura transitiva del grafo di abbondanza e che il parametro* Γ *sia fornito in partenza, allora l'algoritmo impiega tempo* $O(m^{9/8})$ *a creare il grafo compatto* G^c*.*

proof:
Per contrarre i componenti abbondanti connessi, così come i cicli, è necessario tempo $O(m)$. Per quanto riguarda il grafo compattato:

- Gli archi in A^1 possono essere calcolati in $O(m)$;
- Gli archi in A^3 possono essere calcolati in $O(m \log m)$ utilizzando gli alberi dinamici;
- Quelli che sono più complessi da calcolare sono gli archi abbondanti di A^2 che vengono calcolati basandosi sulla chiusura transitiva che richiede costo $|N^c|^2$ per essere mantenuta.

Tuttavia l'algoritmo costruisce G^c solo se il numero di nodi Γ-critici è minore di $m^{9/16}$ dunque il costo diventa

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

Di seguito se vogliamo dimostrare che la complessità di tutto l'algoritmo è proprio quella dichiarata in partenza abbiamo bisogno di porre dei bound tanto alle azioni che vengono compiute quanto agli oggetti che vengono analizzati. La prima cosa da dichiarare è che il numero di tutti i nodi Γ-critici analizzati durante le varie fasi è in $O(m)$

Theorem 4.4 (label = maxM)**.** *max critical node in* $O(m)$ *Supponiamo che ogni improvement phase soddisfi i requisiti richiesti allora i nodi* Γ*-critici calcolati durante le iterazioni sono in tutto* $O(m)$

proof:
per essere Γ-critico un nodo j deve essere adiacente ad un arco Γmedio oppure non avere archi adiacenti Γmedi ma avere $|\Phi(j, r, E^{-ab})| > \Gamma/(16m^2)$ ovvero essere Γ-special.

Consideriamo prima i nodi adiacenti ad un arco Γ-medio:
Claim:
Un arco può avere capacità Γ-media per al massimo 3 fasi consecutive.

proof:
Sia (i,j) un arco di Γ-media capacità allora $u_{ij} + u_{ji} \geq \Gamma/64m^3$ dato che ad ogni fase $\Delta' = \frac{\Delta}{8m}$ allora nella fase subito successiva

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

Ottenendo dunque che dopo 3 fasi $u_ij + u_ji \geq 8\Delta$ dunque o (i, j) o (j, i) sono diventati abbondanti e l'arco non è più Γ-medio.

Per quanto riguarda gli altri archi Γ-special invece:
Claim: Sia Γil parametro di compattezza di una certa Δ-improvement phase e sia j un nodo Γ-special. Se Δ^* è il bound 4 fasi dopo Δallora esiste un nodo k tale che

$$r_{jk} \geq 2\Delta^* \wedge r_{kj} \geq 2\Delta^*$$

ovvero (j,k)(ma anche (k,j)) è *doubly-abundant*, e dunque verrà contratto. *proof:* Per prima cosa definiamo v^* il flusso nella fase Δ^* tale che $r^* = r_{ij} - v_{ij} + v_{ji}$. Dal lemma 4.1 sappiamo che ogni arco Δ-abbondante sarà anche Δ^*-abbondante e in oltre

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

Supponiamo che esista un arco abbondante (j,k) con valore di $v^*_{jk} > \Gamma/64m^3$ allora per l'arco opposto (k,j) vale

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

Dunque anche l'opposto è Δ^*-abbondante e i nodi j e k vengono contratti.

Rimane dunque da verificare il caso in cui un nodo j sia Γ-special senza avere archi Δ-abbondanti con flow maggiore di $\Gamma/64m^3$.
Sappiamo che:

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

consideriamo il caso in cui $\hat{r}_{out}(j) - \hat{r_{in}}(j) > \Gamma/(16m^2)$(l'altro è speculare) Abbiamo che:

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

per conservazione del flusso, in oltre

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

ma dato che abbiamo assunto che nessun arco abbondante ha flow maggiore di $\Gamma/64m^3$

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

Deve esistere dunque qualche arco per cui

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

Dunque esiste qualche (j,k) arco antiabbondante nella fase Δche diventa abbondante nella fase Δ^* e dunque, dato che anche (k,j) è abbondante il ciclo viene contratto.□

Conoscendo il numero di nodi da analizzare il passo successivo sarebbe stimare il numero di **improvement phase** per calcolare il max flow. Prima però è necessario comprendere il modo in cui viene scelto il parametro Γ.

Lemma 4.5 (label = gammchose)**.** *Il parametro* Γ *Il parametro* Γ *può essere scelto in tempo* $O(m + n \log n)$

proof:

1. Per ogni nodo j calcoliamo il più grande valore Γ' per cui j è Γ'-critical (*tempo richiesto* $O(m)$)
2. Ordiniamo i nodi j per il loro valore Γ' (*tempo richiesto* $O(n \log n)$)
3. scegliamo il valore Γ tale che esistano al massimo $m^{1/3}$ nodi j con $\Gamma'(j) \geq \Gamma$ (*tempo richiesto* $O(1)$)

□

Abbiamo ora tutti gli strumenti per calcolare il numero di improvement phase:

Lemma 4.6. *Il numero di improvement phase in* $O(m^{2/3})$

proof:
Sappiamo dal teorema **??** che il numero di nodi Γ-critici analizzati è $O(m)$ e sappiamo dal lemma **??** che in ogni improvement phase almeno $m^{1/3}$ vengono analizzati.

Quando abbiamo dimostrato che il numero di nodi era $O(m)$ la dimostrazione verteva sul fatto che i nodi avessero una "scadenza" e che in massimo 3 o 4 fasi consecutive sarebbero stati contratti. Quindi tutti gli almeno $m^{1/3}$ nodi analizzati in una fase verranno consumati in $O(1)$ fasi. Di conseguenza il numero di fasi necessarie per "consumarli" tutti è proprio $O(m^{2/3})$. □

Dai seguenti lemmi possiamo giungere dunque al tempo totale richiesto per creare tutti i G^c.

Lemma 4.7. *Il tempo totale per creare tutti i compact network è* $O(nm + m^{43/24})$

Proof. Sappiamo che il parametro Γ richiede tempo $O(m + n \log n)$
Sappiamo che per creare un compact network serve tempo $O(m^{9/8})$.
Dato che il numero di fasi sono $m^{2/3}$,
mettendo tutto insieme otteniamo:

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

□ □

Trovare un flusso che che sia α-ottimale significa trovare un flusso che sia inferiore a quello di capacità massima di al più α. abbiamo visto che se cerchiamo il flusso massimo sul Γ-compact network G^c e poi lo trasponiamo sul network originale G questo è già $\Gamma/16m$-ottimale.

Tuttavia durante l'algoritmo noi cerchiamo di ottenere questa approssimazione direttamente in G. Se ricordiamo però il funzionamento del Goldberg-Rao possiamo

vedere che l'algoritmo termina quando la stima che fa del flusso massimo è minore di 1. Intervenendo su questa stima del flusso massimo è possibile far terminare l'algoritmo prima che si arrivi al flusso ottimale ottenendo un distacco di massimo un valore α a nostra scelta.

Ragioniamo ora sul costo T di una fase del Goldberg-Rao sapendo di eseguirlo su un grafo con C nodi e $O(C^2)$ archi.

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

Possiamo ora valutare il costo della procedura Improve-approx-2

In oltre possiamo notare che se eseguiamo il Goldberg rao su un totale di O(m) nodi eseguendo un massimo di $\log U$ fasi il numero medio di nodi in ogni improvement phase è

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

da qui il motivo per cui nella procedura costruiamo il grafo compatto solo se $C \leq m^{9/16}$ infatti se il Goldberg è polinomiale se $\log U \leq m^{7/16}$ allora

$$C \geq m^9/16 \implies \frac{m}{\log U} \geq m^{9/16} \implies \log U \leq \frac{m}{m^{9/16}} = m^{7/3}$$

Lemma 4.8. *Time of improve-max Il tempo per calcolare il flusso ottimale utilizzando sempre la procedura improve-approx 2 è*

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

Proof. Dato che in tutto vengono calcolati $O(m)$ nodi Γ-critici, calcolo il costo della procedura sul singolo nodo invece che per il numero di fasi: Sia T il tempo necessario per trovare un α-optimal flow
Sappiamo che se $C \geq m^{9/16}$ possiamo trovare un optimal flow con $T = O(m^{3/2} \log^2 n)$ eseguendo $\log n$ fasi del goldberg rao sul grafo originale. Rapportato al numero di nodi Γ-critici otteniamo che $T/C = m^{15/16} \log^2 n$. se invece $m^{1/3} \leq C < m^{9/16}$ lavoriamo nel grafo compattato e cerchiamo il max flow eseguendo $\log n$ fasi del goldberg avendo $T = O(C^{8/3} \log n)$ dunque

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

In fine se $C < m^{1/3}$ otteniamo che l'esiguo numero di archi porta il costo ad essere $T = O(C^3)$ di conseguenza $T/C = O(C^2) = O(m^{2/3})$.

Possiamo affermare che in ogni caso il costo del della procedura per ogni nodo è $O(m^{15/16} \log^2 n)$. Moltiplicando il risultato per il numero di nodi Γ-critici su tutti gli incrementi otteniamo

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

□ □

Lemma 4.9. *Il tempo totale per trasformare tutti i flussi calcolati su G^c in flussi sul grafo residuo è*

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

Proof. Sia G^c il grafo compatto determinato da $G[r]$ il grafo residuo e sia y^c il flusso in G^c mentre y è quello indotto sul grafo residuo.

Gli archi di G^c sono divisi in tre categorie $E^c = E^1 \cup E^2 \cup E^3$ rispettivamente gli archi originali, gli pseudoarchi abbondanti e gli pseudoarchi antiabbondanti.

preso un qualsiasi arco (i, j) tale che $y^c_{ij} > 0$ distinguiamo i tre casi:

1. se $(i, j) \in E^1 \implies y_{ij} = y^c_{ij}$
2. se $(i, j) \in E^2$ sappiamo che per ricostruire l'abundant path abbiamo bisogno di $O(|P|) = O(n)$, in oltre sfruttando gli alberi dinamici possiamo tenere il numero di archi con flow positivo in E^2 sotto il valore C al costo $O(m \log n)$, che ripetuto per $O(m^{2/3})$ fasi fa $O(m^{5/3} \log n)$. In questo modo, tutti gli abundant path vengono ripristinati in $O(nm)$. Per ricotruire gli abundant path ricordiamo sempre che il costo di mantenere dinamicamente la chiusura transitiva è di $O(nm)$
3. Per quanto riguarda gli archi in E^3 ancora una volta dobbiamo ricorrere agli alberi dinamici per ricostruire i percorsi anti abundant che abbiamo contratto. In particolare non è possibile tenere traccia di tutti i percorsi in maniera efficiente, tuttavia possiamo tenere un record con tutte le operazioni fatte sul grafo per poterle ripercorrere a ritroso e restaurare i vecchi path. Anche questa procedura ha costo complessivo di $O(m \log n \cdot m^{2/3})$

Concludiamo che il tempo totale per indurre il flow su G^c in $G[r]$ per tutte le $m^{2/3}$ fasi è

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

□ □

Dai precedenti Lemmi possiamo dedurre il seguente teorema

Theorem 4.5. *max flow in $O(nm)$ Se il flow in ogni improvement phase è calcolato utilizzando la procedura improve-approx-2 allora il tempo per trovare il flow di valore massimo è*

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

se $m = 1^{1.06}$ il running time è

$$O(nm)$$

Conclusions

Acknowledgements