Network Flows made simple

A mild but rigorous introduction to Network Flows[⋆]

Periklis A. Papakonstantinou

York University

Network Flows is the last algorithmic paradigm we consider. Actually, we only address the simplest of the Network Flows Problems, namely the Maximum s-t Flow problem. In class we have introduced three more algorithmic paradigms, namely Greedy Algorithm, Divide & Conquer and Dynamic Programming. Note that for some problems it may be possible to use e.g. a natural Dynamic Programming approach to solve problems that admit correct Greedy or Divide & Conquer algorithms. This does not mean that Dynamic Programming is a better approach. Each of the other paradigms has very important properties that cannot be counterbalanced. Despite this, when it comes to Network Flows the same thing does not hold. For example[1], there are problems solvable only using Dynamic Programming and there are problems solvable only using Network Flows.

Among all four algorithmic paradigms, at a first glance Network Flows seem to be the most restrictive one. We start by just introducing one very specific Problem (*Maximum Flow*) and we present an algorithm that solves it. This is quite different than giving general directions of how an algorithm paradigm can be applied to solve many problems. So, initially one may have some difficulty in seeing where is the "paradigm" in Network Flows. In fact Network Flows has a quite diverse and sometimes surprising applicability. There are two main reasons why this approach is a paradigm. The first is that we can use a (widely applied in Theoretical Computer Science) technique which is called "reduction". Informally, this means that given an input for a Problem we are going to see how to formulate it and use a "Network Flow solver" in order to solve it. For some problems "reduction" is not the best way to go. The second reason is that ideas used to solve the *Maximum Flow* problem may appear as ideas applied to solve other problems. Note that in this document we are going to give full proofs. We do so because parts of these proofs (and not the theorems themselves) recur in the proofs of related Problems.

This document is self-contained but far from being complete and it lacks examples. You are expected to read this document together with the your textbook and the notes you took during the lectures. For reasons related to the mathematical maturity level of the audience of this course it's strongly non-dense.

[⋆] Please, send any typos, suggestions and remarks to papakons@cs.yorku.ca. This document heavily relies on a preprint of the textbook "Algorithm Design", by Jon Kleinberg and Eva Tardos
I'd like to thank Mike Liscombe for his useful remarks on a previous draft of this set of notes. Mike took this course as an undergraduate student.

[1] Although (at least in this course) we lack formal models of computations to justify this.

1 Network Flows in contrast to other paradigms

When students are first asked to devise algorithms they usually think in two "very wrong" directions. One is brute-force and the other is to give an algorithm that first finds something close to the solution and then tries to "patch" it. Both of these "wrong" approaches seem to be connected to the way many people develop software. The first task one has to do when thinking about algorithms is understand the properties of the Problem, mainly properties and structure of the optimal solution and understand that an algorithm must work optimally in **general**. We are not just interested in **a** solution that works. We are interested in the most time-efficient (or something close), provably correct solution. Students have to see that in 3101 we start working from different principles. Therefore, one understands what the optimal (correct) output is, asks questions related to its structure, perhaps she/he divides the given input into subproblems, finds optimal solutions for the subproblems and generalizes to optimal solutions for bigger subproblems.

When it comes to Network Flows we somehow violate this way of thinking. It seems that we do what we wanted to avoid in the beginning. Network Flows are using a radically different approach than the other algorithmic paradigms. In some sense the way the algorithms work has a connection with the approach that before we strove to convince students to drop! In Network Flows we will build "solutions" and afterwards, as the execution of the algorithm evolves over time we will change our minds for parts of them. But, we are doing this in a very systematic way. You can think of this as "climbing" trying to go on the top of a hill. You may make some corrections (when choosing paths) but the net result is that you always go higher and higher. Furthermore, keep in mind that this approach (Network Flows) works for a specific class of Problems. In all other cases only other paradigms result algorithms that are correct.

2 A motivating example

We introduce Network Flows in order to solve a seemingly quite different problem. A fundamental problem in Computer Science is the Maximum Matching problem.

Definition 1. *Given an undirected graph $G = (V, E)$ we call as* matching *a subset of the edges $M \subseteq E$ such that every vertex $u \in V$ appears at most once in M. If every vertex appears exactly once in M then we say that M is a* perfect matching*.*

Exercise 1. Give an example of a graph with a perfect matching and give an example of a graph that does not have a perfect matching.

Naturally, the associated optimization problem is the one in which we are trying to match as many vertices as possible.

Problem: *Maximum Matching*
Input: An undirected graph $G = (V, E)$.
Output: A subset of the edges of G, $M \subseteq E$ such that M is a matching in G of maximum

cardinality (the cardinality of a set is the number of its elements) - over all matchings in G.

Matching problems appear in a big number of computational problems either as the main problem; or algorithms solving the *Maximum Matching* can be used as subroutines to solve more "complex" problems. We restrict the input (therefore we define another Problem) to a case of special interest regarding applications.

Definition 2. *An undirected graph $G = ((V_1, V_2), E)$ is called* bipartite *iff there exist edges only between vertices in V_1 and V_2.*
(i.e. given $G = (V, E)$, G is bipartite iff we can partition the vertex set V into two sets V_1 and V_2 such that there does not exists an edge among vertices in V_1 and there does not exists an edge among vertices in V_2.)

Exercise 2. Prove that a graph G is bipartite iff there does not exist an odd cycle in G.

Problem: *Maximum Bipartite Matching*
Input: An undirected bipartite graph $G = ((V_1, V_2), E)$.
Output: A subset of the edges of G, $M \subseteq E$ such that M is a matching in G and $|M|$ is maximum (over all matchings in G).

It appears that the previous three algorithmic paradigms cannot attack this problem. Although they seem to be quite different all of the previous paradigms had one intuitive thing in common. In some sense they were computing suboptimal solutions and they were generalizing. This is not the case for the approach that solves matching problems. It turns out that we need a radically new idea for this.

For now, we are leaving matching problems and move to the *Maximum Flow* problem.

Exercise 3. Come up with "natural" greedy algorithms for the *Maximum Bipartite Matching* and show that they are not correct. Also, try to solve the problem using Dynamic Programming (and again show that the algorithms are not correct).

3 The Maximum Flow Problem

3.1 Definitions

Consider a "network" consisting of pipes and switches. In one extreme of the network we have a *source* (e.g. a tap) that *generates* water and in the other end we have a *sink* (e.g. this is a field we want to irrigate) that *drains* the generated water. In between we use pipes and switches that split the *flow* of the water. Consider that the "network" is in a **steady state**. Clearly, as much flow enters a switch the same flow has to be distributed to the outgoing connections (recall that the source is the only node where flow is generated). Note that in general, not all pipes are of the same diameter and therefore they are capable of carrying different flows.

Formally, a *flow network* is the following:

Definition 3. *We call as a flow network a* directed *graph $G = (V, E)$, together with a function that assigns capacities on its edges $c : E \to \mathbb{N}$ and two designated vertices the source $s \in V$ and the sink $t \in V$.*

To simplify things we make the following assumptions to hold for every flow network:

- *There are no incoming edges to s.*
- *There are no outgoing edges from t.*
- *For every vertex there exists at least one edge incident to it.*

Informally, our goal is to send as much flow as possible from the source to the sink. Doing so we have to respect the capacity constraints and the fact that as much flow enters a vertex the same flow gets out of the vertex. We formalize this concept in the following definition.

Definition 4. *Given a flow network (G, c, s, t) an s-t flow (or simply a flow) on this network is a* ***function*** *$f : E \to \mathbb{R}^{\geq 0}$ subject to the following restrictions (i.e. it's not an arbitrary function):*

1. *For every edge $e \in E$, $0 \leq f(e) \leq c_e$ (i.e. the flow respects the capacity of the edge)*
2. *For every vertex $u \in V \backslash \{s, t\}$, $\sum_{e \text{ is an incoming edge of } u} f(e) = \sum_{e \text{ is an outgoing edge of } u} f(e)$; i.e. the total flow that goes into u equals to the total flow that gets out of u. This property is often called* flow conservation property.

Note that in the above definition we abused the notation since we denote by $c_e \equiv c(e)$. Also, we are going to abuse notation in the following way: for an edge (u, v) we will write $f(u, v)$ to mean $f\big((u, v)\big)$.

We have formalized what a flow network is and what a flow is. It remains to define one of the most important things with respect to the Problem we wish to solve (which we define just after this).

Definition 5. *Given a flow f on a network we define the* value *of this flow*

$$|f| = \sum_{e \text{ is an outgoing edge of } s} f(e)$$

Remark 1. Note the following:

- The capacities of a flow network and **a** flow on this flow network are two different things. For some reason (that I don't get) students keep confusing these two notions.
- The capacities are all non-negative *integers* where the flow can be a non-negative *real number.*
- Instead of defining the flow in terms of the flow getting out from (i.e. being generated in) the source we could have defined it as the flow that enters the sink t.

Exercise 4. Prove that given a flow network (G, c, s, t) and a flow f on this network

$$|f| = \sum_{e \text{ is an outgoing edge of s}} f(e) = \sum_{e \text{ is an incoming edge of t}} f(e)$$

(The proof is very simple, but perhaps you want to postpone doing it until you see the ideas in the proof of Lemma 5.)

We also introduce some notation to avoid writing too many $\sum$s.

Definition 6. *For a flow network $(G = (V, E), c, s, t)$, a flow f on this network and a vertex $u \in V$:*

$$f^{in}(u) = \sum_{e \textit{ is an incoming edge of } u} f(e), \qquad f^{out}(u) = \sum_{e \textit{ is an outgoing edge } u} f(e)$$

Now we can rephrase the flow conservation property: for every $u \neq s, t$, $f^{in}(u) = f^{out}(u)$. Also, we can rephrase the definition of the value of a flow f: $|f| = f^{out}(s)$. Also we generalize the above definition to hold for sets of vertices instead only for vertices. The definition of an outgoing and an incoming edge for a set of vertices V' is clear. Therefore, $f^{in}(V')$ and $f^{out}(V')$ also gets the obvious meaning.

Here is the associated computational Problem:

Problem: *Maximum s-t Flow* (or *Maximum Flow*)
Input: A flow network $N = (G, c, s, t)$.
Output: f, where f is a flow on N such that $|f|$ is maximum (over all possible flows for this network).

3.2 An intuitive approach

Consider the flow network of figure 1. It seems natural to choose a (simple) path between s and t and try to send as much flow as possible. For every s-t path the edge of the smallest capacity is the one that determines the maximum flow that can be send through this path.

In terms of our physical analog the edges correspond to pipes of a certain capacity. Therefore, the pipe of the smallest capacity is the one that determines the maximum flow that we can send through this specific (path) connections of pipes. Remember that under our formulation, the source can generate as much flow as we want. The restriction on how much of it we can transfer are only posed by the capacities of the pipes.

We are going to "solve" the Maximum Flow problem by "pushing" as much flow as we can through an s-t path. Then we will choose another s-t path and so on, until we cannot "push" any more flow from the source.

Choose an s-t path. Say that this path is $\langle s, u, v, t \rangle$. Note that the smallest edge in this path is of capacity 2. Therefore, we cannot send more than flow 2 using this path. So,

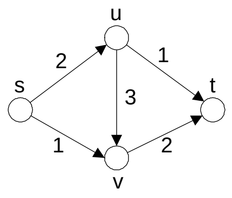

Fig. 1. Flow network

$f(s, u) = f(u, v) = f(v, t) = 2$. After doing so, we realize that we cannot send more flow (from the source s). If we try to send 1 unit of flow through (s, v) then we are stuck, since (v, t) has already been used to send 2 units (in the previously chosen s-t path). Formally, if we do so the Flow Conservation property doesn't hold: $3 = f^{in}(v) \neq f^{out}(v) = 2$. Therefore, no other s-t path can be selected. Is this the best we can do? That is, is 2 the maximum achievable (value of) flow? No! The maximum achievable flow is the following: $f'(s, u) = 2, f'(u, t) = 1, f'(u, v) = 1, f'(s, v) = 1, f'(v, t) = 2$. Therefore, $|f'| = 3$.

What we really need coincides with the first thing that comes to our mind. We wish to have a way of "*canceling*" some of our previous (seemingly, locally-optimal) choices. In order to do so we should have a way of storing the information needed to undo the effect of a previous choice of pushing flow through a chosen s-t path.

Intuitively this means the following: Initially the flow is zero on every edge. Then, suppose that we have already chosen the first s-t path and we have "pushed" flow 2 through its edges. Now, we wish to choose another s-t path, say $\langle s, v, t\rangle$ and push flow 1 through its edges. When we reach vertex v we realize that this is impossible. If we can undo part of the effect of the first chosen s-t path then we would be in a great shape. The only thing we need is to cancel only 1 unit from the flow going through (u, v). If we do so then we can continue pushing 1 unit from s then go to v and finally reach t. What happens to the 1 unit of flow we canceled? You can think of this as pushing it back from v to u. But then at u we would have a surplus of 1 unit of flow. Clearly, this is free (under the current flow in the network) to go through (u, t). What we really need here is a *systematic way* (i.e. algorithm) of doing something similar. In order to do so we need all of the information of the effect of the current flow on the network. And we need this information before choosing the next s-t path!

Formally, we introduce one more definition in which we do not merely look at the graph in the flow network but instead we will be looking in this graph together with the information provided by the current flow. Therefore, we are looking into a **new** network which is a **function** of both the initial network and the current flow.

Definition 7. *Given a flow network $N = (G, c, s, t)$ and a flow f on this network we call as a* residual graph *a weighted graph G_f that every edge in the initial graph may corresponds to two edges in the G_f. Specifically, G_f and its weights is defined as follows:*

– *We call as* forward edge *every edge belonging both to the initial graph G and to the residual graph G_f. Say that (u, v) is a forward edge. Then, the capacity of (u, v) in G_f*

is $c^f_{(u,v)} = c_{(u,v)} - f(u,v) > 0$. Therefore, the capacity of this edge equals to the remaining flow we may be able to "push" through it. Also, if $c_{(u,v)} = f(u,v)$ then $(u,v) \notin G_f$.
- *We call as* backward edge *an edge that gives as information on how much flow we can cancel. That is, if (u,v) is an edge in G then (v,u) is an edge added in G_f. Naturally, the capacity of (v,u) in G_f is $c^f_{(v,u)} = f(u,v) > 0$. Therefore, the capacity of this edge equals to the flow we can "cancel" (push back) of the flow assigned to (u,v) (i.e. the edge of the opposite direction in G). Also, if $f(u,v) = 0$ then $(v,u) \notin G_f$.*
- *Note, that we exclude from G_f edges of zero capacity.*

Let us introduce the following terminology just to save us from rewriting the same thing over and over.

Definition 8. *Given a flow network $(G = (V,E), c, s, t)$ and a flow on this flow network we say that an edge $e \in E$ is* saturated *(under flow f) iff $f(e) = c_e$.*

For example, say that a flow on the network depicted in figure 1 is: $f(s,u) = 2, f(u,v) = 2, f(v,t) = 2, f(s,v) = 0$ and $f(u,t) = 0$. Then, the residual network is depicted in figure 2. Now, looking on the residual network it turns out that what we were saying before

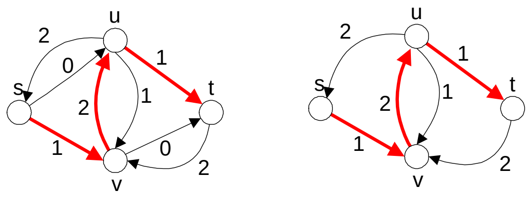

Fig. 2. Left: the residual network where we include the edges of zero capacity. Right: the residual network (when omitting edges of zero capacity - as in the definition). The thick (red) edges are edges of another s-t path.

concerning the choice of a second s-t path is intuitively equivalent to the choice of the s-t path depicted with the thick (red) edges.

Exercise 5. Do not read further unless you convince yourself that what we were intuitively discussing before regarding the choice of the second s-t path corresponds to the s-t path in the figure. Note that these two s-t paths are not the same but the net result is the same. Intuitively these two procedures (the intuitive and the formalized) should correspond to the same action.

After choosing the s-t path depicted on figure 2 and "pushing" flow 1 on its edges (this equals to the edge of minimum capacity) the residual graph is the one depicted on figure 3

The value of the flow is $|f| = 3$ and it is maximum. Note that on the graph of figure 3 there are no more s-t paths! Think about it and you will realize that intuitively this means that we cannot push any more flow. In order to make this precise we have to give an algorithm and prove it correct. This task is not straightforward and it is the topic of the next section.

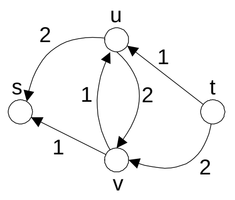

Fig. 3. Residual graph of the flow network of figure 1 when the flow is $f(s,u) = 1, f(s,v) = 1, f(u,v) = 1, f(v,t) = 2, f(u,t) = 1$.

Remark 2.

- Do not get confused on the following! A flow is a **function** that assigns "flow" (real numbers) to the edges. The value of a flow is a number that corresponds to the total flow getting out of s.
- In the previous example it happened that in the residual network there were no more s-t paths because there were no outgoing edges from s. Clearly, this is only a coincidence (i.e. it's not the general case). For example, figure 4 depicts a flow network and the residual graph under a maximum flow (actually for this network there exists a unique function f which of maximum value of flow). Note that the reason why there are no s-t paths in the residual graph is due to "internal edges".

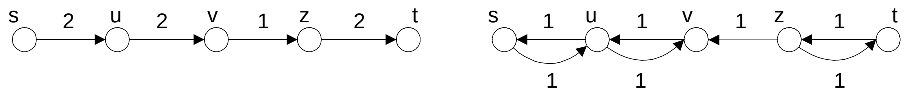

Fig. 4. Left: flow network. Right: residual graph under maximum flow. There does not exist an s-t path because there is no directed edge (v,z).

- There is an issue of how we construct a residual graph. If we already have a not-everywhere-zero flow then we can think of the residual graph (it is a weighted graph) as a new flow network. When we find an s-t path on the residual graph and we push some flow on it then we can either (i) construct the new residual graph by viewing the current residual graph as a network or (ii) superimpose the new flow values with the old ones and construct the new residual graph from scratch (i.e. from the initially given flow network and the flow function f). If you choose to do (i) you have to superimpose (i.e. accumulate) the assigned flow and keep it in some temporary memory because at the end this is what you have to output anyways.

  Exercise 6. Show that these two approaches are equivalent.

- In order to have a unified way to view things, you may consider the (initially given) flow network as a residual graph where the flow is everywhere zero.

Technical details: (i) Under our definition between two vertices u, v we allow the existence of the edges (u,v) and (v,u). But then when we discuss about the residual graph we have to distinguish between the forward edge (u, v) (due to (u, v) in G) and the backward edge (u, v) (due to (v, u) in G). Formally, in this case the edge set of G_f is a multiset and the capacity function c^f is a multi-function. (ii) Also, our definition does not exclude flow networks where the edge sets of the underlying graphs are multisets. That is, we may allow multiple edges of the same direction between two vertices (although we cannot think of any application where this is required).

4 The Ford-Fulkerson method

The previous discussion naturally gives rise to the following method (family of algorithms).

Definition 9. *Given a weighted directed graph $G = (V, E)$, $w : E \to \mathbb{R}^{\geq 0}$ and a directed path P in G we say that an edge $e \in P$ is a* critical edge *if it is of minimum weight w.r.t. the weights of the edges in P.*

```
Ford-Fulkerson[$G = (V, E), c, s, t$]
1 Initialize $f(u, v) = 0$ for every edge $(u, v) \in E$.
2 while in the residual graph $G_f$ there exists an $s$-$t$ path $P$
      do
3         Let $c^P_{min}$ be the minimum capacity of $P$ on $G_f$ (i.e. $c^P_{min}$ is the value of a critical edge)
4         for every edge $e \in P$
              do
5                 if $e$ is a forward edge
                     then
6                        $f(e) \leftarrow f(e) + c^P_{min}$
7                    else
8                        $f(e) \leftarrow f(e) - c^P_{min}$
9 Output $f$
```

Remark 3. Because of the importance of this s-t path P we will reserve the special name "augmenting path" for such s-t paths.

Why is this a family of algorithms and not an algorithm? In an algorithm every step must be completely specified. In the above description there is a degree of freedom regarding line 2; where we choose an augmenting path. When making precise what this step is then we have an algorithm. In this course almost every algorithm that we present is not a "real" algorithm in the sense that some steps are not specified.

For example, recall that regarding some scheduling algorithms we were stating something like "order the intervals in non-increasing finishing time". This statement by itself does not say what to do when the intervals are having equal finishing times. In this example, how do we break ties? Technically, unless we specify we don't have an algorithm. But in this example it simply didn't matter at all how do we break ties! No matter which among two or more intervals of equal finishing time we were going to put first in the ordering the algorithm: (i) is correct (i.e. the proof of correctness is independent of every specific choice of ordering intervals of the same finishing time) and (ii) has exactly the same running time. Therefore, we pretended that this was an algorithm and we had no reason to be more specific (actually the more unnecessary technical details you put in a description the less understandable an algorithm becomes).

Now, lets get back to the Maximum Flow problem and discuss what happens with respect to this step in line 2 of the Ford-Fulkerson method. Here things are very different than in the example regarding the ordering of intervals. As we will see the choice of the augmenting path (i) does not affect at all the correctness of the algorithm but (ii) it significantly affects its running time. Despite this we will give a proof of correctness pretending that the above is an algorithm. In this proof we will make no assertion regarding a choice of a specific augmenting path and thus we will prove that every algorithm that works as the Ford-Fulkerson method suggests is correct. As far as it concerns the running time we will see that for some choices of augmenting paths the algorithm has exponential worst-case running time where for others the running time is asymptotically bounded by a polynomial in $n \cdot m$, where $n = |V|$ and $m = |E|$.

We have to distinguish between the "flow" computed during the algorithm and the mathematical notion of flow. Unless we prove correctness these two things are not the same! We denote by f_{output} the "flow" outputted by the algorithm. Also, we denote by f_i the function ("flow") in the i-th iteration of the algorithm. If max is the number of iterations then $f_{max} = f_{output}$

Here comes the real fun. There are two things about Ford-Fulkerson that are not clear at all. The first is that why the preconditions of the Maximum Flow problem imply the post conditions (under Ford-Fulkerson). The other is even more basic. Why does Ford-Fulkerson terminate?

We begin by showing termination. As when proving termination of every algorithm we have to associate the iterations of Ford-Fulkerson with a strictly decreasing sequence in the natural numbers. As when proving that a simple for-loop terminates we have to find a quantity that is strictly *increasing* (i.e. in every iteration we make some progress w.r.t. this quantity) and subtract it from an upper bound that remains unchanged through the iterations. For example, in a simple for-loop **for** $i \leftarrow 1$ **to** n we were defining $t_i = n - i$ (here n is the upper bound and i is the value that strictly increases). In case of Ford-Fulkerson both things are not that straightforward. A natural choice is the value of the flow f itself! But then we even encounter more basic problems. For example, why this value is an integer? Recall that the capacities are integers but the flow is not necessarily an integer. For example for the flow network depicted on figure 5 a maximum flow is $f(s,u) = 1, f(s,v) = 0, f(u,z) = 1, f(v,z) = 0, f(z,t) = 0$. This flow has only integer values. Wait a minute! The following flow is also maximum for the same network: $f(s,u) = 0.5, f(s,v) = 0.5, f(u,z) = 0.5, f(v,z) = 0.5, f(z,t) = 1$, where the values of the flow are not integers. And as obvious we could also have irrational values, e.g. $f(s,u) = 1 - \frac{1}{\sqrt{2}}, f(s,v) = \frac{1}{\sqrt{2}}, f(u,z) = 1 - \frac{1}{\sqrt{2}}, f(v,z) = \frac{1}{\sqrt{2}}, f(z,t) = 1$

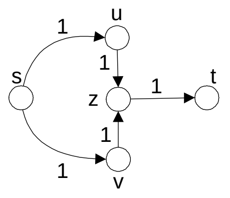

Fig. 5. Yet-another flow network example

Fortunately, a straightforward induction shows that for the Ford-Fulkerson every value of the flow is a natural number. Actually, a stronger result holds. In every iteration of the Ford-Fulkerson the value of the flow of every edge is an integer.

Lemma 1. *In every iteration i of the Ford-Fulkerson on $[G = (V, E), c, s, t]$ for every $e \in E$, $f_i(e)$ is an integer.*

Proof. We do induction on the iterations of Ford-Fulkerson. (Basis) When no iterations $f_0(e) = 0$. (Induction step) Suppose that it is true for k iterations. W.t.s. that it is true for $k + 1$. Fix an arbitrary edge $e \in E$. By the induction hypothesis $f_k(e)$ is an integer. Since c_e is an integer by the definition of G_f we have that the capacities of the residual network G_f will also be integers. Therefore, c^P_{min} will also be an integer. And thus, by the way the algorithm works $f_{k+1}(e)$ will also be an integer (because either $f_k(e) = f_{k+1}(e)$ or $f_k(e)$ either increases or decreases my c^P_{min}).

■

Above we used the term "flow" without knowing whether f_{output} (or f_i) is really a flow. We abused terminology and valid reasoning, but we really didn't need the fact that f is flow (we just needed that it is a function). Below we show that it is a flow (we need this to continue the analysis).

Lemma 2. *In every iteration f_i is a flow.*

Proof. We do induction on the iterations of the algorithm. (Basis) When no iterations f_0 has the properties of a flow. (Induction step) Suppose that f_i is a flow. W.t.s. that f_{i+1} is also a flow. We have to show that the two properties of a flow hold for f. Fix an arbitrary edge e in the **residual** graph G_{f_i}. If e is a forward edge then by the way the algorithm works $c^P_{min} \leq c_e - f_i(e)$ (c^P_{min} is the minimum residual capacity for the chosen augmenting path). Thus,

$$0 \leq f_{i+1}(e) = f_i(e) + c^P_{min} \leq f_i(e) + (c_e - f_i(e)) = c_e \Rightarrow 0 \leq f_{i+1}(e) \leq c_e$$

If e is an backward edge then

$$c_e \geq f_i(e) \geq \underbrace{f_i(e) - c^P_{min}}_{f_i(e) - c^P_{min} = f_{i+1}(e)} \geq f_i(e) - f_i(e) = 0 \Rightarrow 0 \leq f_{i+1}(e) \leq c_e$$

It remains to check that we have flow conservation. Fix an arbitrary vertex $u \in V$. We want to show that $f_{i+1}^{in}(u) = f_{i+1}^{out}(u)$. The augmenting path P is a simple path. If $u \notin P$ then there is nothing left to prove. Else, say that e is the incoming edge (in the **residual** graph) to u and e' is the outgoing edge (in the **residual** graph) where both e and e' belong in P. If e and e' are both forward edges then $\underline{f_{i+1}^{in}(u) = f_i^{in}(u) + c_{min}^P} = \underline{f_i^{out}(u) + c_{min}^P = f_{i+1}^{out}(u)}$. If e is a forward edge and e' is an backward edge then $\underline{f_{i+1}^{in}(u) = f_i^{in}(u) + c_{min}^P - c_{min}^P} = \underline{f_i^{out}(u) = f_{i+1}^{out}(u)}$. Similarly, we prove the last two cases.

■

We show that in every iteration the flow f_i strictly increases.

Lemma 3. *For every two iterations i and $i+1$ it holds that $|f_i| < |f_{i+1}|$.*

Proof. Fix an arbitrary $i \geq 0$. Say that the flow in the i-th iteration is $|f_i|$. Since the algorithm takes more iterations this implies that there exists an augmenting path P in G_f. Therefore, there exists a *forward* edge $e \in P$ going out of s (recall that in the definition of the flow network we have assumed that s has no incoming edges in G). Since e is a forward edge we have that $f_{i+1}(e) = f_i(e) + c_{min}^P$, where $c_{min}^P > 0$. Hence by the definition of $|f|$ we have $|f_i| < |f_{i+1}|$.

■

Since by Lemma 2 we have that f_i is a flow we know that it respects the capacities of the edges. Since s has only outgoing edges we have that $|f_{output}| = f^{out}(s) \leq \sum_{u \text{ adjacent to } s} c_{(s,u)} = F$. Therefore, F is an upper bound in the value of the flow. In Lemma 5 we will prove a much better upper bound and in Theorem 4 we will show that it is tight(in general F may be a very bad overestimation). For our current needs (we only want to show that $|f|$ does not grow arbitrarily) F suffices and it was the simplest thing to show.

Theorem 1. *Ford-Fulkerson terminates.*

Proof. Let i be the iteration of the algorithm and $d_i = F - |f_i|$. We have shown that $|f_i|$ is strictly increasing and that it is also an integer (the sum of integers is an integer). We also have that F is always greater or equal to $|f_i|$ and thus d_i is strictly decreasing sequence in $\mathbb{N}$.

■

It remains to show that the preconditions of the Maximum Flow imply the postconditions. This is not as simple as showing termination. As a bonus going through this analysis we are getting for free the proof of a theorem which is considered as one of the jewels of Discrete Mathematics, namely the Maximum Flow-Minimum Cut theorem, which also appears to have a reasonably easy proof w.r.t. an undergraduate course.

4.1 Cuts and flows

Here we investigate a few properties of networks and flows. Algorithms have nothing to do with this subsection.

Definition 10. *Say that $G = (V, E)$ is a directed graph. Partition its vertex set into two sets V_1 and V_2. We call (V_1, V_2) a* cut *of G. Say that $(G = (V, E), c, s, t)$ is a flow network and (V_1, V_2) is a cut of V such that $s \in V_1$ and $t \in V_2$. Then (V_1, V_2) is called an s-t* cut *of G. There are edges that go from the set V_1 to the set V_2. We define the capacity of an s-t cut,*

$$c(V_1, V_2) = \sum_{e \text{ goes from } V_1 \text{ to } V_2} c_e$$

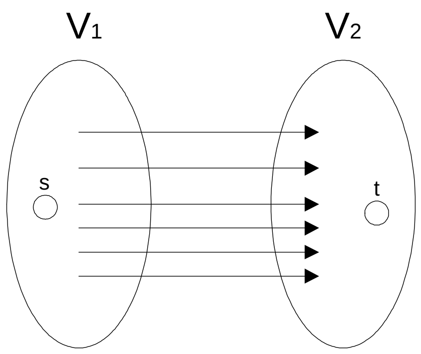

Fig. 6. An s-t cut of G. We depict only the edges that go from V_1 to V_2. (In general) there may be others going from V_2 to V_1 but we just chose not to draw them in this specific figure.

Intuitively, it should not be hard for somebody to see that the flow we are going to send from s to t cannot be greater than the capacity of **every** s-t cut. This is because the flow has necessarily to pass through these edges. Note that this is not an immediate implication of the definition (why?). Before reading further think of this for a while. The important thing is to see why this holds for *every* cut.

We prove this by first showing a stronger result.

Lemma 4. *For a given flow network $N = (G = (V, E), c, s, t)$ and every s-t cut (V_1, V_2) of G and every flow f on N it holds that $|f| = f^{out}(V_1) - f^{in}(V_1)$.*

Proof. Note that there is no typo in the above equation. We are just interested in V_1. This lemma should be intuitively true from the flow conservation property of the flows. To make it precise we do some small technical work with sums. By definition no edge enters s and thus we can define $f^{in}(s) = 0$ without worrying. Consider the vertices of V_1. Among these vertices only s is the one where $f^{out}(s) - f^{in}(s) \geq 0$. For every other vertex $u \in V_1$ (recall that t is in V_2) we have that $f^{out}(u) - f^{in}(u) = 0$. Moreover, by the definition of

$|f|$ we get $|f| = f^{out}(s) = f^{out}(s) - f^{in}(s)$. Let us rewrite this by adding zeros (recall that $f^{out}(u) - f^{in}(u) = 0$, $u \neq s$)

$$|f| = \sum_{u \in V_1} (f^{out}(u) - f^{in}(u))$$

We want to change the above summation so that it talks about edges going out of V_1 and getting into V_1. If every edge e has both endpoints (vertices) in V_1 then $f(e)$ appears in the above sum once with a positive and once with a negative sign. Therefore, the contribution of this edge to the sum is zero. If e has its outgoing vertex in V_1 and its ingoing in V_2 then $f(e)$ appears with a positive sign. If e has its incoming vertex in V_1 and its outgoing in V_2 then $f(e)$ appears with a positive sign. All told,

$$|f| = \sum_{u \in V_1} (f^{out}(u) - f^{in}(u)) = \sum_{e \text{ gets out of } V_1} f(e) - \sum_{e \text{ gets into } V_1} f(e) = f^{out}(V_1) - f^{in}(V_1)$$

■

As a corollary of the above lemma we get the following:

Lemma 5. *For a given flow network $N = (G = (V, E), c, s, t)$ and every s-t cut (V_1, V_2) of G and every flow f on N it holds that $|f| \leq c(V_1, V_2)$.*

Proof.

$$|f| = f^{out}(V_1) - f^{in}(V_1) \leq f^{out}(V_1) = \sum_{e \text{ out of } V_1} f(e) \leq \sum_{e \text{ out of } V_1} c_e = c(V_1, V_2)$$

■

Remark 4. Lemma 5 is a property that holds independently to the choice of the cut (no matter how small) and the flow (no matter how big). Clearly, it does not imply that the minimum cut (i.e. the cut of the minimum capacity) equals to the maximum flow. It could be the case that when in the proof we were getting rid of the term $f^{in}(V_1)$ it was the case that $f^{in}(V_1) > 0$ or that when bounding $f(e)$ with c_e, the edge e was not saturated. So we have two places where we could have overestimated (when bounding from above). Remarkably, it turns out that this is not true when it comes to a minimum cut and to a maximum flow! The next subsection is devoted to the proof of this fact, which is one of the most simple, unexpected and therefore beautiful results in combinatorics.

Also, note that in the previous section regarding the Ford-Fulkerson algorithm we needed an upper bound in the flow so as to show termination. Lemma 5 gives a better one (but for termination it suffices to have any upper bound).

4.2 "Max Flow-Min Cut" theorem and the correctness of Ford-Fulkerson

Recall that f_{output} is the flow outputted by the Ford-Fulkerson method. Lemma 2 tells us that f_{output} is a flow. Note that for every s-t cut (V_1, V_2) in the given network and for every *maximum* flow $\hat{f}$ by Lemma 5 holds that $|f_{output}| \leq |\hat{f}| \leq c(V_1, V_2)$. Recall that we did not know whether this inequality holds (for some flow and cut) as equality. Despite this, if we show that there exists an s-t cut (V_1^{min}, V_2^{min}) such that $|f_{output}| = c(V_1^{min}, V_2^{min})$ then we would have shown that $|f_{output}|$ is maximum over all flows for this network. And this is what we are going to do in order to prove correctness of the Ford-Fulkerson. Note that in the following proof we also implicitly describe an algorithm that can be used to modify Ford-Fulkerson so as to output a minimum cut.

Theorem 2. *Ford-Fulkerson is correct with respect to* Maximum Flow *problem.*

Proof. We have already shown termination. Since the algorithm terminates only if there are no s-t paths in the residual graph, to complete the proof of correctness it suffices to show the following theorem.

■

Theorem 3. *Let $N = (G, c, s, t)$ be a flow network and f be a flow on N. If the residual graph G_f has no s-t paths then there exists an s-t cut (V_1^{min}, V_2^{min}) in G such that $|f| = c(V_1^{min}, V_2^{min})$.*

Proof. Suppose that there is no s-t path. Partition the vertices of G into sets V_1^{min} and V_2^{min} according to the following property that holds in the **residual** graph. Put in V_1^{min} the vertex s and every vertex u reachable from s in G_f. Define V_2^{min} as the complement of V_1^{min} w.r.t. to V, i.e. $V_2^{min} = V \backslash V_1^{min}$. We have to show that (V_1^{min}, V_2^{min}) is an s-t cut (note that it's definitely a cut). Since there is no s-t path by construction $s \in V_1^{min}$ and $t \in V_2^{min}$.

Our next goal is to show that (i) $f^{in}(V_1^{min}) = 0$ and (ii) $f^{out}(V_1^{min}) = \sum_{e \text{ goes out of } V_1^{min}} c_e$. Recall that if these two things hold then

$$|f| = f^{out}(V_1^{min}) - f^{in}(V_1^{min}) = \sum_{e \text{ goes out of } V_1^{min}} c_e - 0 = c(V_1^{min}, V_2^{min})$$

Therefore it remains to show (i) and (ii).

(i) Suppose that $e = (u, v)$ is an edge that crosses the cut (V_1^{min}, V_2^{min}) by going out of V_1^{min}, i.e. $u \in V_1^{min}$ and $v \in V_2^{min}$. We wish to show that $f(e) = c_e$. Lets look in the residual graph G_f (recall that the construction of V_1^{min}, V_2^{min} was done regarding paths in the residual graph). Since by construction $v \notin V_1^{min}$ in the residual graph there is no s-v path. But (again by construction) there is an s-u path. This means that in the residual graph the forward edge (u, v) is of capacity zero and therefore removed. This happens only if $f(e) = c_e$.

(ii) Suppose that $e = (u, v)$ is an edge that crosses the cut (V_1^{min}, V_2^{min}) by going in V_1^{min}, i.e. $v \in V_1^{min}$ and $u \in V_2^{min}$. We wish to show that $f(e) = 0$. Again we look in the residual graph. Again by construction we observe that in the residual graph the backward edge e' that corresponds to e has $f(e') = c_e$ which means that $f(e) = c_e - f(e') = 0$.

■

Compare this theorem with what we said earlier in Remark 4.

The above theorem is a simple to show fact that reveals a fascinating connection between two problems. One is the *Maximum s-t Flow* problem (defined earlier). The other is the *Minimum s-t Cut.*

Problem: *Minimum s-t Cut*
Input: A flow network $(G, c, s, t)^{\cdot}$
Output: An *s-t* cut of minimum capacity

If we focus in the values of the outputs of the two problems then we see that the (optimal) solutions to both problems are equal. Lets write explicitly an immediate corollary of Theorem 3 and Lemma 5.

Theorem 4 (Max-Flow equals Min-Cut). *Let $N = (G, c, s, t)$ be a flow network. Then the value of a maximum flow in N equals to the capacity of a minimum cut in N.*

The proof of Theorem 3 implies an algorithm that computes the minimum cut of a given network. Here is the algorithm:

Algorithm [Compute Min-Cut]

– For a given network $(G, c, s, t,)$ compute the maximum flow f_{output} (using the Ford-Fulkerson method).
– From G and f construct the residual graph G_f.
– Find every vertex reachable from s in the **residual** graph G_f. This can be done e.g. using Breadth First Search or Depth First Search (in time $O(n + m)$ where n is the number of vertices and m the number of edges).
– The vertices reachable from s are the vertices in V_1^{min} and the rest in V^{min_2}.
– Output the minimum cut (V_1^{min}, V_2^{min}).

What is the running time of the above algorithm? It is $O(FF(n) + n + m)$, where $FF(n)$ is the running time of the (implementation) of the Ford-Fulkerson method.

Remark 5. Note that computing the Minimum *s-t* Cut or the Maximum Flow for a given network is important in applications. When we use solvers for these problems (in order to solve others) in some cases Minimum *s-t* Cut has an easier (more intuitive) application and in others Maximum Flow is more preferable.

4.3 Running time of the Ford-Fulkerson method

A pseudopolynomial time algorithm

Actually the Ford-Fulkerson method cannot have a running time because it is not an algorithm. In order to make it an algorithm we first have to specify how do we choose an augmenting path. It appears that if you use BFS or DFS (or any other "fast" algorithm that chooses an arbitrary path) then things can be bad, since we get a pseudopolynomial time algorithm. We will see that in this case the running time of the algorithm is asymptotically bounded by a polynomial in $F \cdot m$ (where F is an upper in the value of the maximum flow). Therefore, the algorithm polynomially depends on F which also depends on the *values* of the involved capacities in the input. That is, if the numbers in the input are represented in binary the algorithm has worst-case running time exponential in the number of bits in the input.

Suppose that we choose s-t paths on the residual graph using a BFS or DFS algorithm. This takes time $O(n + m)$, where $n = |V|$ and $m = |E|$. To update the flow it takes as much time as the number of edges in the s-t path which is at most m. Technical detail: we can maintain a copy of the residual network G_f and each time we update the flow we also update the edges the G_f. This also takes time $O(m)$. Alternatively we can in each iteration to reconstruct (from scratch) the residual graph from the flow network and the flow. This also takes time $O(m)$. Therefore, in every iteration we spend time $O(n+m+m+m) = O(m)$ (since by the assumptions every vertex in the flow network has at least one edge incident to it and thus $n = O(m)$). Therefore, the running time heavily depends on the number of iterations of the main loop. We have already seen that in every iteration the value of the flow strictly decreases by a natural number. Therefore, in the worst case it decreases by 1. Assuming that the sum of the capacities of the outgoing edges of s is F we have that the running time of the algorithm is bounded by $O(Fm)$.

In every case, but especially if you get such a big running time you should always ask whether this is an overestimation (i.e. is the worst case running time also $\Theta(Fm)$?). Unfortunately, the answer to this question is that this is not an overestimation. In order to lower bound the worst case running time we have to present an infinite family of input instances that make the algorithm to take as many steps as possible. Below (figure 7) we give a counter for a constant size of the input. We see that the algorithm can work by choosing by alternating between the augmenting paths $\langle s, u, v, t\rangle$ and $\langle s, v, u, t\rangle$. In this case it needs $10^6 + 10^6 = 2 \cdot 10^6$ iterations. Also, in this case $F = 2000000$. Note that for "most" heuristics you may think of (e.g. choose the path from the DFS by selecting the vertices in the order they appear in the input or choose the path from the DFS by selecting vertices starting from the end in the order they appear in the input etc.) we can engineer the input such that something horrible (similar to the example given below) will happen. We are not done yet! In order to establish the lower bound we need an *infinite* family of input networks graphs. We can trivially extend the below given (constant size) example. One way of doing so is to replace t with a directed chain of vertices where t is at the end of it and the capacity of the edges of this chain is huge.

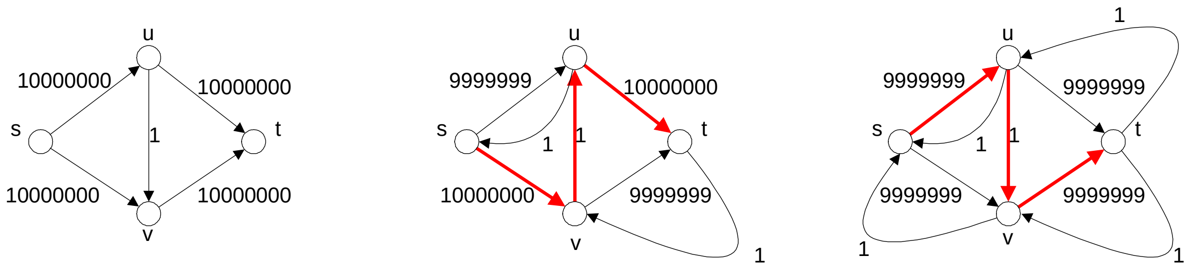

Fig. 7. Left: flow network (you can also consider it as a residual graph when the flow is zero). Middle: Residual graph when the chosen augmenting path (in the given flow network) is $\langle s, u, v, t\rangle$. We denote using thick(red) edges the next chosen augmenting path $\langle s, v, u, t\rangle$. Right: Residual graph when the chosen augmenting paths were $\langle s, v, u, t\rangle$ and $\langle s, u, v, t\rangle$. The red edges correspond to the choice of the next augmenting path.

The Edmonds-Karp algorithm

The question is whether there exists a choice of augmenting paths such that the algorithm always takes at most a polynomial number of steps. Surprisingly, the answer is yes! Even more surprisingly the answer is counter-intuitive. The Edmonds-Karp algorithm is a realization of the Ford-Fulkerson method where the augmenting paths are chosen according to the number of edges in the path. The rule is to choose an augmenting path with the minimum number of edges. This is an amazing and strongly counter-intuitive result. This is because, when applying this rule, the choice of the augmenting path has nothing to do with the residual capacities in the residual graph! In the above example at a first glance it seems that we were able to get the lower bound by engineering the graph but "mainly" its capacities. And it appears that in Edmonds-Karp the capacities themselves are not important.

The Edmonds-Karp algorithm has running time $O(n{\cdot}m)$ and therefore the Maximum Flow and the Minimum s-t Cut problems are solvable within polynomial time.

Theorem 5. *The running time of Edmonds-Karp is $O(nm)$.*

We omit the proof of this result. The reason is not because the proof is complicated (this means: more technical than other proofs given in this document). We do not give the proof since we believe that in the current level of study of Network Flows this proof does not add much in the understanding. But, the students are expected to know the theorem and the rule used to choose the augmenting paths! This is the/one reason why Max-Flow is solvable in polynomial time!

4.4 A remark on the values of the maximum flow

We have seen why the values of a maximum flow can, in general, be non-integers. On the other hand we have proved that the Ford-Fulkerson method outputs only an integer flow. We have also proved that the Ford-Fulkerson method is correct for the Maximum Flow problem. Putting these two together we take as a corollary the following theorem:

Theorem 6. *For every flow network there exists an integer-valued maximum flow.*

Obviously, for a fixed network we do not assert that every maximum flow is integer-valued.

Observe that this theorem has nothing to do with algorithms. Interestingly, we can show the theorem by using an algorithm in the proof. This is a general methodology of proving existence of combinatorial objects. Sometimes, it seems easier to prove existence by presenting a (correct) algorithm that outputs an object with the desired properties.

It also seems plausible to ask what would have happened if we had a different definition of a flow network where the capacities are non-negative real numbers (instead of integers). It turns out that in this case there are examples where the Ford-Fulkerson *does not terminate*. Note that in order to prove termination we heavily relied on the fact that the flows computed by Ford-Fulkerson are integer-valued. This property allows us to show that the sequence $\{d_i\}_i$ is strictly decreasing in the *natural numbers*. What would have happened if we had a strictly decreasing sequence in say the rational numbers? What if for some (arbitrary) reason $d_i = 1/i$? Then d_i is also strictly decreasing and converges to zero, but it needs an *infinite* number of elements to converge!

On the other hand the "Max-Flow equals Min-Cut" theorem holds even in case of real valued capacities. Note that in the proof of Theorem 3 we did not use any algorithmic fact. We just relied on the fact that there are no s-t paths in the residual graph.

5 Reductions

5.1 Basic concepts

The term "reduction" pops-up all over Theoretical Computer Science. *For two given Problems Π_1 and Π_2 the term "reduce Π_1 **to** Π_2" generally refers to an algorithm that solves every instance of Π_1 using a solver for Π_2.* There are literally tenths of types of reductions and in this course we are interested in some small fraction of a class of them known under the term "strong reductions". When reading further keep in mind the following:

- A reduction has to do with problems (not algorithms).
- A reduction is an algorithm (actually it is a computable relation - but lets skip this).
- The "direction" of the reduction is extremely important. Reducing Π_1 to Π_2 is completely different than reducing Π_2 to Π_1.
- As a rule of thumb the algorithm that reduces one problem to another is an algorithm that *does **not** do "real work"*. Our goal is to keep the running time of the algorithm that does the reduction very small (smaller than the time needed to solve the problem directly). That is, within the reduction we do not solve the Problem we just **transform** it so that we can use the help of the solver (the solver will do the actual work).

Under general definitions of reductions when we have an algorithm that uses another algorithm as a "subroutine" could be "classified" as a reduction. Forget about this option which is only misleading! It is better to think that the problem to which we reduce to, the solver for this problem (the algorithm that solves it) takes (much) more time than the algorithm that does the reduction.

Have we already come up with any reductions? Just before we saw an algorithm that solves the Maximum Flow. We also wanted to solve the Minimum *s*-*t* Cut Problem. What was our algorithm for the Minimum *s*-*t* Cut Problem? For the given input instance it just used a solver for the Maximum Flow Problem; and then it computed the vertices reachable from s in the residual graph. Therefore, we may say that we "reduced the Minimum *s*-*t* Cut to the Maximum Flow Problem". Conceptually, this is a reduction, since the "hard-work" is assigned to the algorithm (solver) that computes the Maximum Flow.

Note that most reductions require some more sophisticated *transformation of the input instance*. We will shortly see examples.

In the beginning we said that the "Network Flows" is an algorithmic paradigm. Reducing other Problems **to** Maximum Flow (or Min-Cut) is one way of seeing why it is a paradigm.

5.2 Network Flows with multiple sources and sinks

Here is a generalization of the Maximum Flow. The modifications in the definitions of a flow network and a flow are done in the obvious way. Perhaps in a not so obvious way for the value of the flow: $|f| = \sum_i f^{out}(s_i)$.

Problem: *Maximum Flow with multiple sources and sinks*
Input: A (multiple source-sink) "flow network" which consists of a directed graph G, a capacity function c, k sources $s_1, s_2, \dots, s_k$ and l sinks $t_1, t_2, \dots, t_l$.
Output: f, where f is a (multiple source-sink) flow on the (multiple source-sink) network such that $|f|$ is maximum (over all possible multiple source-sink flows for this network).

It's almost straightforward. Think before reading further!

We will reduce *Maximum Flow with multiple sources and sinks* to *Maximum Flow*. Suppose that we are given a (multiple source-sink) network with k sources $s_1, s_2, \dots, s_k$ and l sinks $t_1, t_2, \dots, t_l$. How will we modify this input so as to use a Maximum Flow solver? The idea is to add two more vertices. A super-source s and a super-sink t. We connect s with the given sources and t with the given sinks as depicted on figure 8.

We allow infinite capacities on the edges connecting s with the initially given sources and t with the initially given sinks. Technically, we cannot do that since our definitions do not allow infinite capacities (high-school knowledge ☺: $+\infty \notin \mathbb{R}$). Therefore, wherever you see ∞ you can read it as having the sum of every capacity in the initial multiple source-sink flow network.

Then we run a solver for the Maximum-Flow (say the Edmonds-Karp algorithm) and we output whatever the output of the solver is. Therefore the algorithm is the following:

- On input a multiple source-sink network apply the above transformation.
- Run Edmonds-Karp on the transformed input instance.
- From what you got from the output remove the values of the flow referring to the edges incident to s and t and output the rest f'.

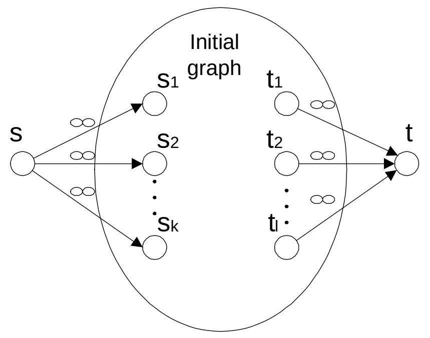

Fig. 8. Transformation of the initial multiple source-sink flow network into a flow network.

Now we have to show correctness of our algorithm. Note that our algorithm just implements the reduction. Proving correctness for such algorithms involves arguments that have quite a different flavor than what you have seen before in this course.

Theorem 7. *The above algorithm (which implements the reduction) is correct for* Maximum Flow with multiple sources and sinks.

Proof. By construction the sum of the flow in the transformed flow network equals to the sum of the flow assigned to the outgoing edges of $s_1, \dots, s_k$. Therefore, by definition of the value of the (multiple source-sink) flow we have that $|f| = |f'|$.

■

6 Maximum Bipartite matching

Now we are going to solve the motivating Problem. We reduce the Maximum Bipartite Matching to the Maximum Flow Problem. Observe that the two problems are of quite different flavor. That is, an input instance in the Maximum Bipartite Matching Problem consists only of an *undirected*, bipartite graph, where the input instance in the Maximum Flow is a *directed* graph, two designated vertices and edge capacity. Here is a correct algorithm for the Maximum Bipartite Matching (this algorithm does the reduction).

- Given $G = ((V_1, V_2), E)$ construct $G' = (V', E')$ as follows: (i) make all edges to have the direction from V_1 to V_2, (ii) add two vertices s, t to G' and (iii) connect with directed edges s with the vertices in V_1 (i.e. add the edges $(s, u), u \in V_1$) and connect t with directed edges with the vertices in V_2 (i.e. add the edges $(u, t), u \in V_2$).
  The capacities for every edge in E' is 1.
- Solve the Maximum Flow on (G', c, s, t) and take the output f.
- Output as a matching in G the corresponding edges that in G' f equal to 1.

Theorem 8. *The above algorithm is correct for the Maximum Bipartite Matching problem.*

Proof. First, lets restrict the maximum flows we are interested in. By the correctness of Ford-Fulkerson for the Maximum Flow problem we have that the outputted flow is integer-valued. Therefore, by construction this flow takes values 0 or 1. That is, there always exists a $\{0,1\}$ valued maximum flow and furthermore the Ford-Fulkerson (it is used as a "subroutine" in our algorithm) outputs it.

To prove correctness (of our reduction) we have to show *two* things.

One is that for every maximum matching on G there exists a $\{0,1\}$ valued maximum flow in the flow network (that the algorithm constructs) where for every edge between vertices in V_1 and V_2 the flow is 1 if this edge is in the maximum matching and otherwise the flow is 0.

The other is that for every maximum flow with values $\{0,1\}$ choosing the vertices where the edge between V_1 and V_2 has flow 1 this corresponds to a maximum matching.

We can shorten the proof by instead proving the following two stronger statements:

(i) for every matching on G there exists a $\{0,1\}$ valued flow in the flow network (that the algorithm constructs) where for every edge between vertices in V_1 and V_2 the flow is 1 if this edge is in the matching and otherwise the flow is 0.

Say that $M = \{(v_1^1, v_1^2), (v_2^1, v_2^2), \ldots, (v_k^1, v_k^2)\}$ is a matching in G. Then, the following can be verified that it is a flow in (G', c, s, t): $f(v_i^1, v_i^2) = 1$, $1 \leq i \leq k$, $f(s, v_i^1) = 1$, $1 \leq i \leq k$, $f(v_i^2, t) = 1$, $1 \leq i \leq k$ and the flow is zero everywhere else. Obviously f respects the capacity constraints and to complete the proof one has to verify the second property, i.e. the flow conservation property (this also trivially holds).

(ii) for every flow with values $\{0,1\}$ choosing the vertices where the edge between V_1 and V_2 has flow 1 this corresponds to a matching in G.

Say that f is a flow on (G', c, s, t). W.l.o.g. we have to show that there are no two edges (v, u) and (z, u) where $v, z \in V_1$ and $u \in V_2$ such that $f(v, u) = 1$ and $f(z, u) = 1$. This also trivially holds because if this where the case then $f^{in}(u) = 2$ where $f^{out}(u) \leq 1$ (it's less or equal than 1 since by definition the flow respects the capacities of the edges).

Combining (i) and (ii) we have: there exists a matching in G of k edges **iff** there exists a $\{0,1\}$ valued flow in (G', c, s, t) of flow value k. This means that for the maximum matching there exists a flow which is also maximum (and vice-versa). Since the algorithm uses a Max-Flow solver we are done.

■

Exercise 7. Spend as much time as you think it is necessary to understand why both directions ("iff") are **absolutely necessary** to prove correctness of the algorithm that computes the reduction. Show what is going wrong if only one direction of "iff" holds.

Exercise 8. Prove that the Min-Cut algorithm (section 4.2) is correct for the *Minimum s-t Cut* problem.