COMP 360: Algorithm Design

Julian Lore

Last updated: January 10, 2018

Notes from Hatami Hamed's Winter 2018 lectures.

Contents

1 01/08/18

Course webpage. Look at it for more details on the grading scheme, assignments and more.

We are assumed to have some background in the course, so today Hatami will be looking over what we should know for this course.

1.1 Background Knowledge

- Tree
- Graph, $G = (V, E)$ (all questions in assignments and exams will be written formally, so you should know what the letters mean)

- DFS, BFS
- Basic algorithm techniques: Greedy algorithms, dynamic programming, divide and conquer, recursion
- Running time analysis (Big-O notation)
- It's important that you should be able to read math, like precise and formal notation.

1.2 Sample Problems

You should be able to read and understand these problems. The problems are available here on the course webpage.

Example 1 S is a set of positive integers.

$$A = \sum_{x \in S} x^2$$
$$B = \sum_{\substack{x \in S, \\ x^2 \in S}} x$$

Let $S = \{1, 2, 3, 4, 5\}$. What are A and B?

$A = 1^2 + 2^2 + 4^2 + 5^2 = 1 + 4 + 9 + 16 + 25 = 46$
$B = 1 + 2 = 3$

For B, the number must be in S and its square must also be in S.

Example 2 M is an $n \times n$ matrix. M_{ij} denotes ij-entry of M. The total sum of the entries of M is 100.

$$\sum_{i=1}^{n} \sum_{j \in \{1,\ldots,n\} \setminus \{i\}} \sum_{r=1}^{n} M_{ir} = ?$$
$$= \sum_{i=1}^{n} \sum_{r=1}^{n} (n-1) M_{ir} = (n-1)100$$

Since we are summing the inner entry $n - 1$ times (the second summation).
Binary expansion/representation.

Example 3 How many digits are in the binary expansion of n?

$$\text{Ex.} n = 5 \implies n = \underbrace{101}_{\text{binary}}$$

$\lceil \log_2 n \rceil$ is the answer.

Example 4

$$\sum_{n=0}^{k} 2^n =? = 2^{k+1} - 1$$

In binary, this is $\underbrace{1111\ldots1}_{\text{binary}}$. Note that this is a geometric sum and that you should be able to calculate these.

Example 5 $S = (a_1, a_2, \ldots, a_n)$ a sequence of integers. E is the set of even numbers in $\{1, \ldots, n\}$.

$$A = \sum_{i \in E} a_i$$

Example:

$S = \{1, \underline{3}, 2, \underline{5}, 4\}$

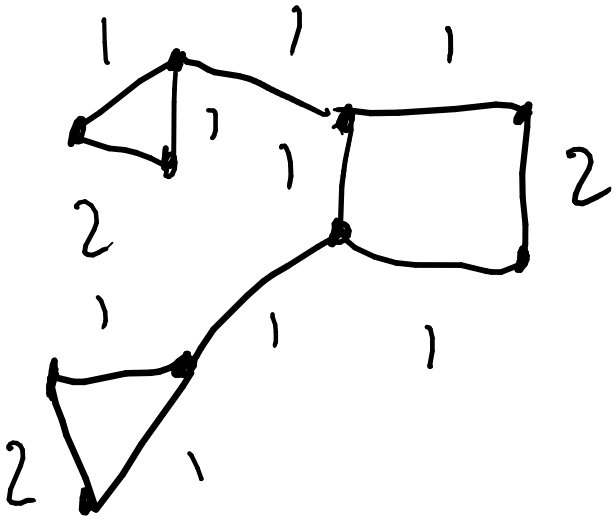
$$A =? = \sum_{i \in \{2,4\}} a_i = a_2 + a_4 = 3 + 5 = 8$$

Example 6 $G = (V, E)$ an undirected graph. Suppose to every edge uv a number C_{uv} is assigned. What does the following statement mean?

$$\exists c \forall u \in v \sum_{uv \in E} c_{uv} = c$$

There exists some number c, such that for every vertex we choose, the sum of all edges containing this vertex is the same for all vertices.

Example



In this case, $c = 3$.

Example 7 $G = (V, E)$ undirected graph degree of every vertex is 10. Suppose to every vertex $v \in V$ a positive integer a_v is assigned.
If $\sum_{v \in V} a_v = 5$ then what is $\sum_{u \in V} \sum_{\substack{w \in V: \\ uw \in E}} a_w =? = \sum_{w \in V} 10 a_w = 10 \times 5 = 50$. Each a_w appears in the sum 10 times since the degree of each vertex is 10.

1.3 Topics Covered

The following are the topics we will be covering in this course:

- Network flows (More of like a practice topic for what we'll be seeing in the course, will use the algorithm to solve this problem for seemingly unrelated problems. We'll be doing this a lot in the course, called reduction, where we reduce solving one problem to another problem.)
- Linear Programming (Bunch of constraints and want to optimize a linear function). This will be one of the most important concepts we learn in this course.
- Midterm
- Linear Programming again

- NP-Completeness (no good algorithms for problems that seem very basic, useful skill to have even if you aren't a theoretician)
- Approximation algorithms (settling for the next best thing for NP-Complete problems, might be able to find an algorithm that approximates things, not exactly optimal, but some sort of factor of how good the approximation is; lots of research happening in this area, better and better approximations). Will use a lot of linear programming here.
- Randomized algorithms (randomness can actually help us; probability theory/knowledge of random variables may help a little bit here, but this is the last stretch of the course and not very essential)

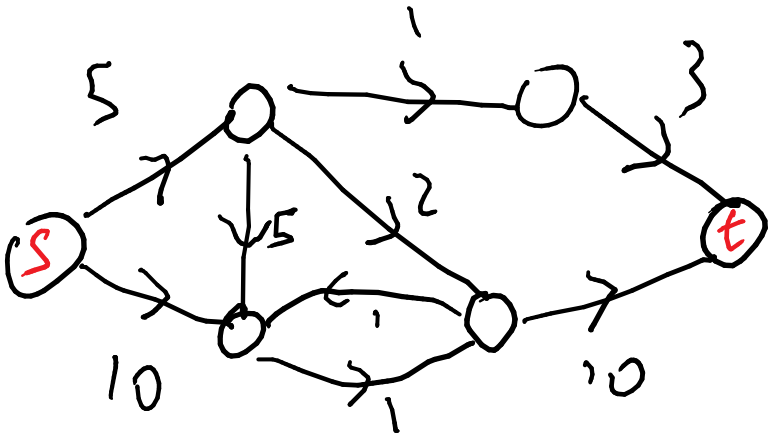
1.4 Network Flows

Max Flow Problem

Very important, used in things like game theory. Def: A flow is a directed graph $G = (V, E)$ such that:

1. Every edge e has a capacity $c_e \geq 0$.
2. There is a source $s \in V$.
3. There is a sink $t \in V$ such that $t \neq s$.

Example



Remark : For the sake of convenience we make the following assumptions.

1. No edge enters the source.

2. No edge leaves the sink.
3. All capacities are integers.
4. There is at least one edge incident to every vertex.

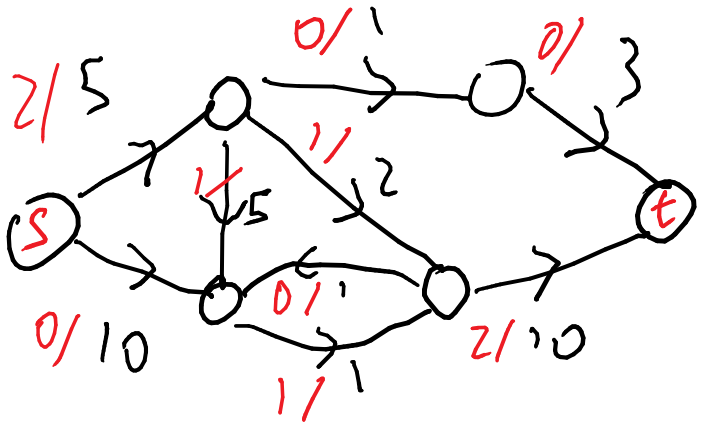
Def: [flow] A flow is a function $f : E \to \mathbb{R}^+$ such that: (Note that :$\mathbb{R}^+ = \{X \in \mathbb{R} | x \geq 0\}$)

(i) [capacity] $\forall e \in E, 0 \leq f(e) \leq c_e$ (flow cannot be negative nor can it exceed capacity)

(ii) [conservation] For every node u other than source and sink the amount of flow that goes into u = the amount of flow that leaves u. Formally:

$$\forall u \in V \setminus \{s,t\} \underbrace{\sum_{vu \in E} f(uv)}_{f^{\text{in}}(u)} = \underbrace{\sum_{uw \in E} f(uw)}_{f^{\text{out}}(u)}$$

Example



Def: $Val(f) = \sum_{su \in E} f(su) = f^{\text{out}}(s)$

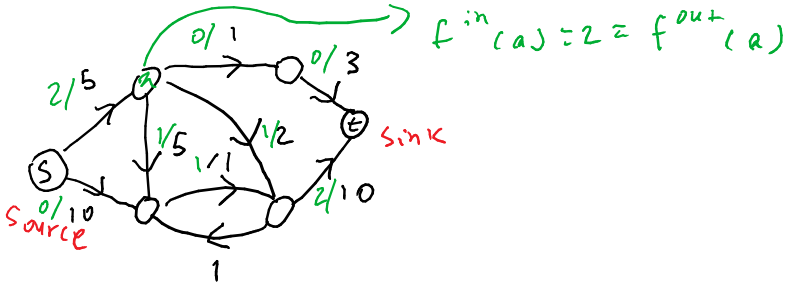
Max Flow Problem: Given a flow network find a flow with largest possible value.

2 01/10/18

2.1 Max Flow Problem (Continued)

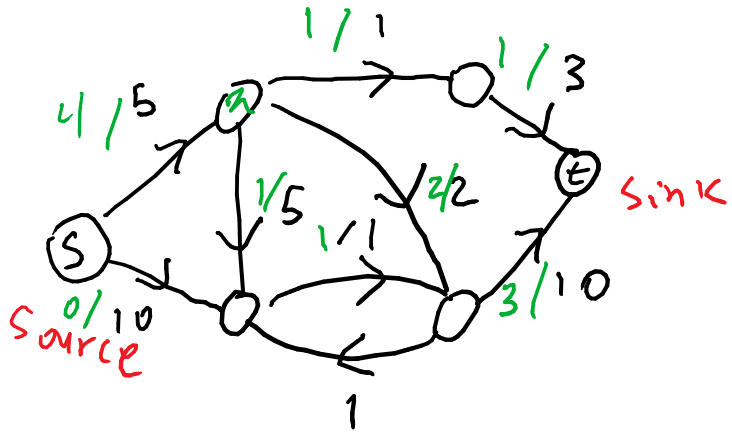
Recall: we want to process a flow network, essentially a directed graph with a source and a sink.

A flow network

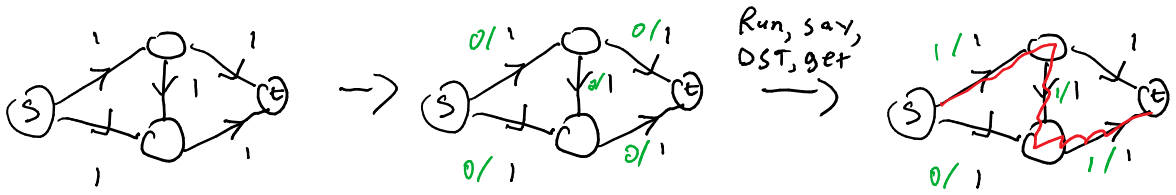


$val(f) = f^{out}(s)$

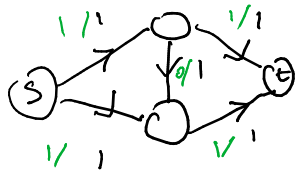
Max Flow Problem: Given a flow network find the maximum value of the flow. 2 is not the optimal value of the example. We could change it to:



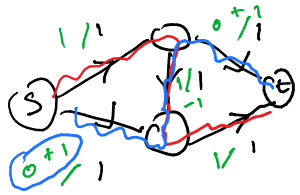
Ford-Fulkerson Algorithm Try to find $s-t$ paths that have not used their capacity and push more flow through them. There is a subtlety here though, we may run into trouble, like in the following example:



Now we are stuck. This is **not optimal**. The following is:

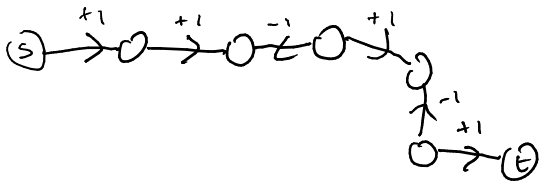


So we must change or else this algorithm won't work. We don't want to go back and change the first step, even though we are stuck. There is a way that we can change things. Say we try to add on more unit of flow:



Essentially, the flow we added "cancels" the edge in the middle and makes it go back. Formally:

1. Start from the all zero flow.
2. Find a "path" (not a real path since we can also reverse directions) from $s-t$ such that the edges that are in the forward direction have **unused capacity** (not saturated) and the backward edges have **strictly positive** flow on them. Add one unit to forward edges and subtract one unit from backwards edges. Repeat this step until we cannot find any more paths.

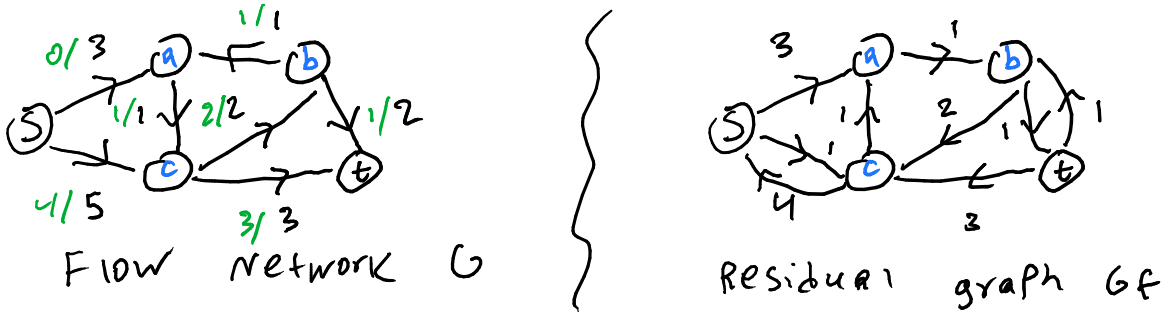


How do we implement this?

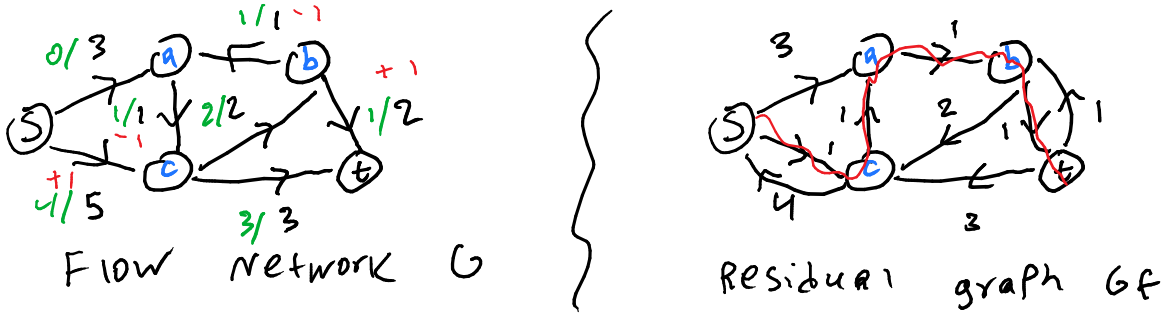
Def [Residual graph] Given a flow network $(G, s, t, \{c_e\})$ and an flow f on G, the residual graph G_f is as follows (we are already in the middle of the algorithm and this graph will tell us which edges are usable):

1. Nodes are the same as G.
2. For every edge $uv \in G$ with $f(uv) < c_{uv}$ (flow strictly smaller than capacity), add the edge uv with residual capacity $\mathbf{c_{uv} - f(uv)}$ to G_f.
3. For every edge $uv \in G$ with $f(uv) > 0$ add the opposite edge $\underline{vu}$ with residual capacity $f(uv)$.

Example



How do we use the residual graph? Just run a DFS on G_f to find an $s - t$ path and use it to modify the original flow, like so:



Pseudocode for Ford-Fulkerson

Initially set $f(e) = 0, \forall e \in E$

Construct G_f

while there is an $s-t$-path P in G_f **do**

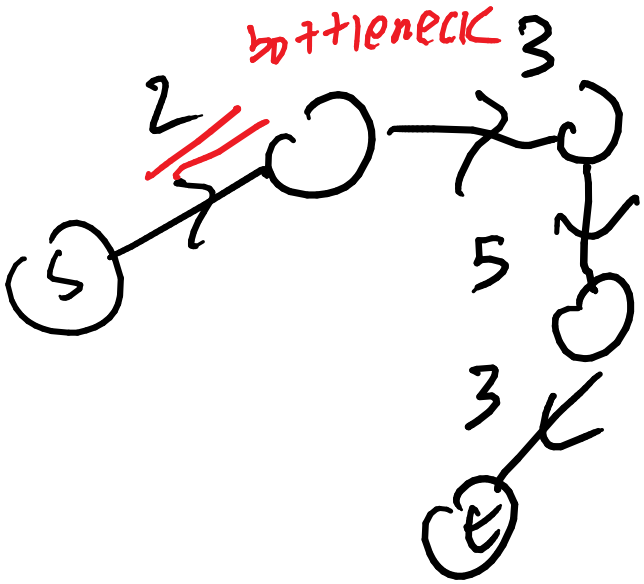
$f' \leftarrow$Augment(f, p), where Augment means increase the flow using path P

update $f \leftarrow f'$

update G_f

end while

How many units of flow can we push if we find the following path in G_f?



The smallest weight, the bottleneck.

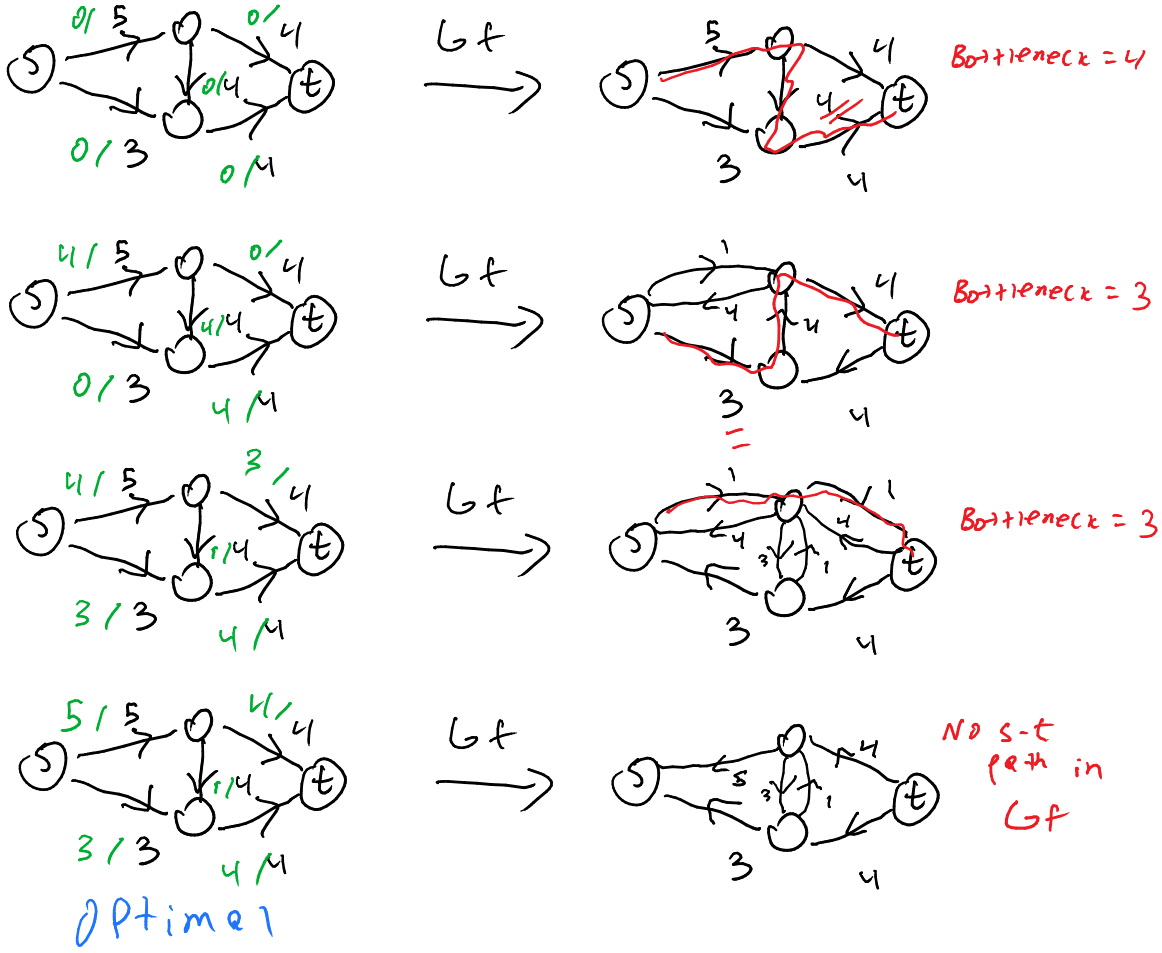
Augment(f, P)

Find the bottleneck of P, which is the smallest residual capacity on P.

For forward edges we add this number to their flow.

For backward edges we subtract.

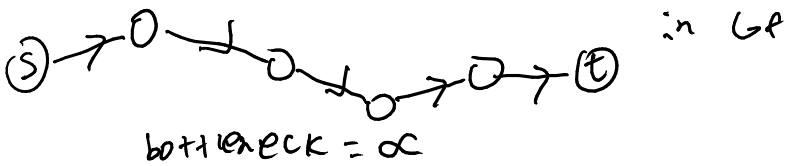
Example



Claim FF always returns a valid flow (proof of correctness).

Proof Residual capacities are chosen so that updating with Augment(f, P) will never assign a number to an edge that is larger than its capacity or smaller than 0. $\implies$ capacity condition is satisfied throughout the algorithm.

Conservation Condition $f^{in}(v) = f^{out}(v)$



In G:

- Case 1:

$$f^{in} \leftarrow f^{in} + \alpha$$
$$f^{out} \leftarrow f^{out} + \alpha$$

Still the same.

- Case 2:

$$f^{in} \leftarrow f^{in} + \alpha - \alpha$$
$$f^{out} \leftarrow f^{out}$$

Nothing changed.

- Case 3:

$$f^{in} \leftarrow f^{in}$$
$$f^{out} \leftarrow f^{out} - \alpha + \alpha$$

Still equal.

- Case 4:

$$f^{in} \leftarrow f^{in} - \alpha$$
$$f^{out} \leftarrow f^{out} - \alpha$$

Equal.

In all cases $f^{in}(v)$ remains equal to $f^{out}(v)$. So we have shown that the flow remains valid, but we still don't know if it gives us the optimal solution or not.

Claim The algorithm terminates.

Proof At every iteration, the flow increases by at least 1 unit. It can never exceed the total sum of all the capacities, so it has to terminate.

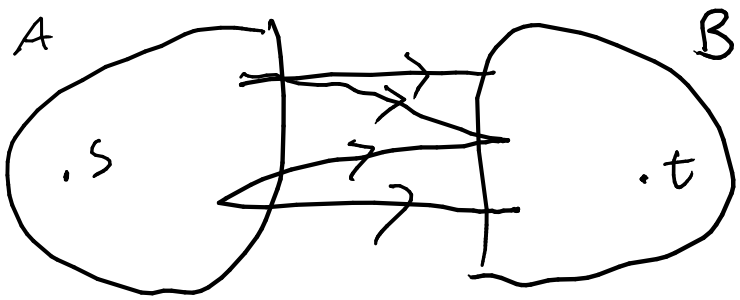
Running Time Let K be the largest capacity, n the number of vertices, m the number of edges. There are at most Km iterations. Finding an $s-t$-path: $O(m+n)$ (each iteration requires a DFS in the residual graph and an update). Augmenting: (n).
Since we assumed every vertex is adjacent to at least one edge $\frac{n}{2} \leq m$ (with this assumption we can just talk about m). This makes the DFS $O(m)$.
The total running time:

$$O(K \times m \times m) = O(Km^2)$$

Unfortunately not that great if K is a large number. We'll try to improve this a little bit later.

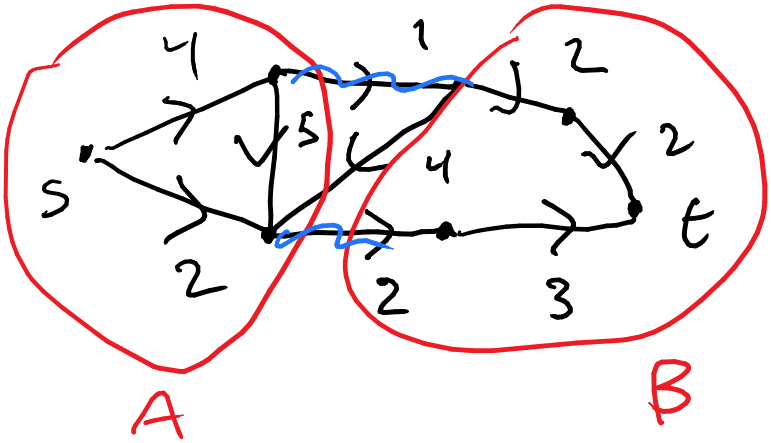
Def A cut ($s-t$-cut) in a flow network is a partition (A, B) of the vertices such that $s \in A, t \in B$.

Def Capacity of this cut is the sum of the capacities and edges going from A to B.

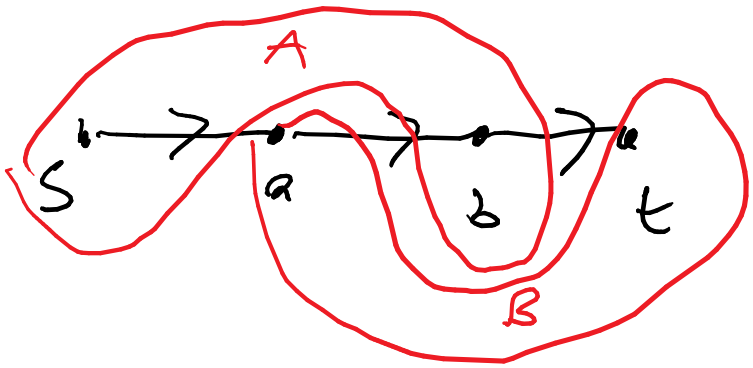


$$cap(A, B) = \sum_{\substack{uv \in E \\ u \in A \\ v \in B}} c_{uv}$$

Example



The capacity here is 3. We see that we can't pass more weight from A to B, i.e. cuts intuitively tell us something about the max flow.
How many cuts are in this network?



4. There's no geometry in cuts, the only restriction is that s is in A and t is in B, doesn't matter how network is drawn.
A network with n vertices has 2^{n-2} (s,t)-cuts. ($n-2$ vertices each with two choices: $2 \times 2 \times \ldots \times 2 = 2^{n-2}$)