COMP 360: Algorithm Design

Julian Lore

Last updated: January 24, 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.

$$\sum_{i=1}^{n} \sum_{j\in\{1,\dots,n\}\setminus\{i\}} \sum_{r=1}^{n} M_{ir} = ?$$

$$= \sum_{i=1}^{n}\sum_{r=1}^{n}(n-1)M_{ir} = (n-1)100$$

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?

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

$$\lceil \log_2 n \rceil \text{ is the answer.}$$

Example 4

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

In binary, this is $\underbrace{1111\dots1}_{\text{binary}}$. Note that this is a geometric sum and that you should be able to calculate these.

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

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

Example:

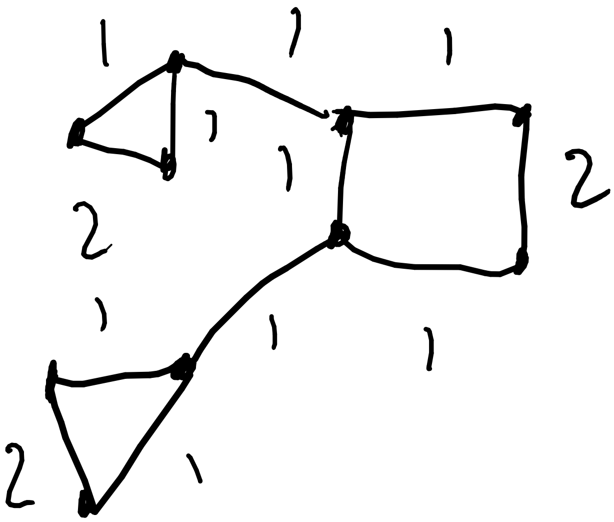
$S = \{1, \underline{3}, 2, \underline{5}, 4\}$
$A = ? = \sum_{i\in\{2,4\}} a_i = a_2 + a_4 = 3 + 5 = 8$

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.)

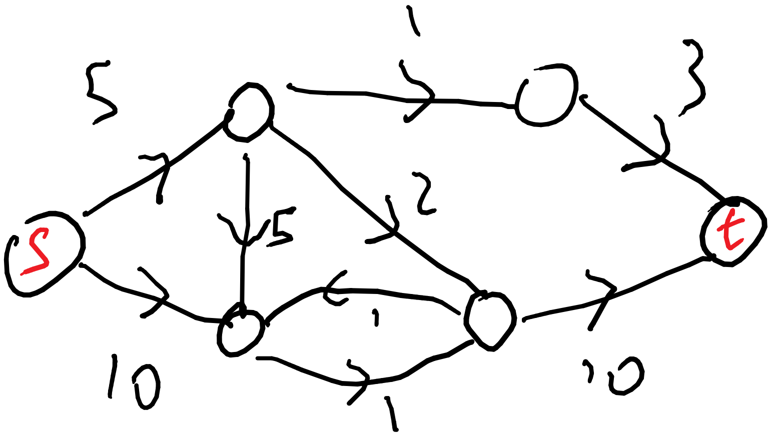
- Linear Programming (Bunch of constraints and want to optimize a linear function). This will be one of the most important concepts we learn in this course.
- Midterm
- Linear Programming again
- NP-Completeness (no good algorithms for problems that seem very basic, useful skill to have even if you aren't 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.

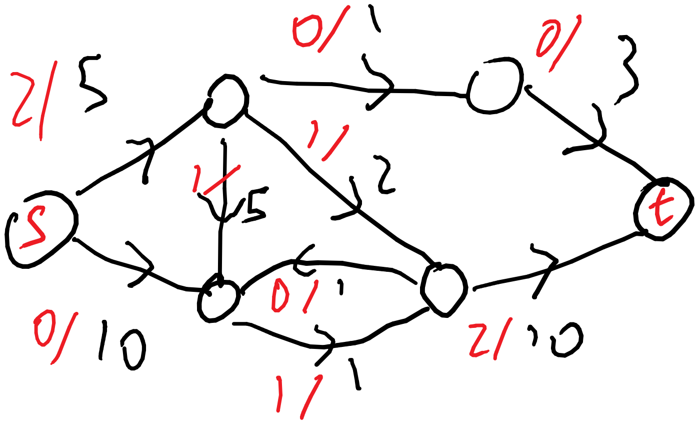
Def: [flow] A flow is a function $f : E \to \mathbb{R}^+$ such that: (Note that :$\mathbb{R}^+ = \{X \in \mathbb{R} | x \geq 0\}$)

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

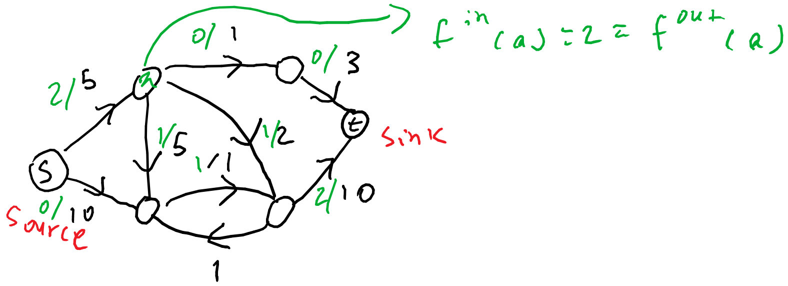
Max Flow Problem: Given a flow network find a flow with largest possible value.

2 01/10/18

2.1 Max Flow Problem (Continued)

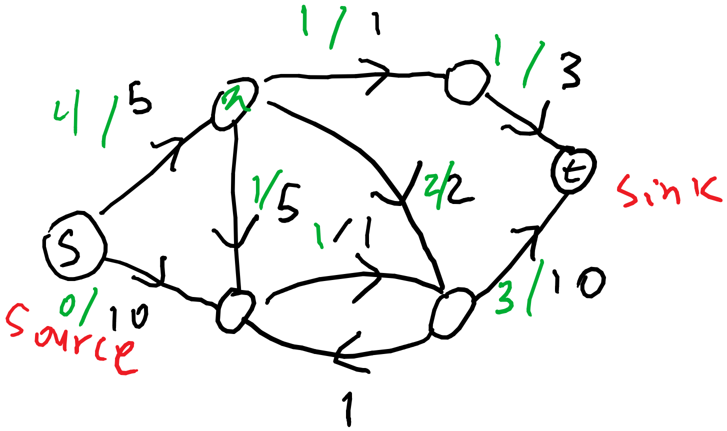
Recall: we want to process a flow network, essentially a directed graph with a source and a sink.

A flow network

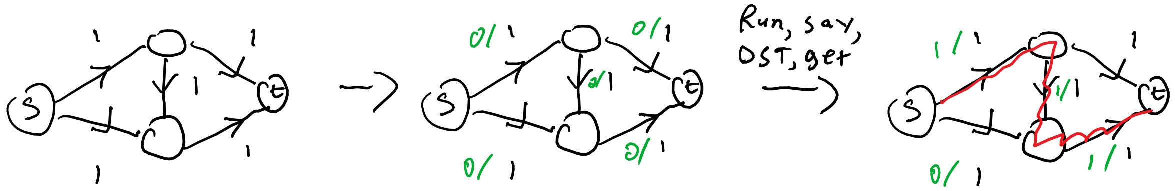


$val(f) = f^{out}(s)$

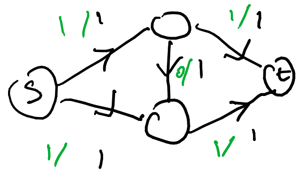
Max Flow Problem: Given a flow network find the maximum value of the flow. 2 is not the optimal value of the example. We could change it to:



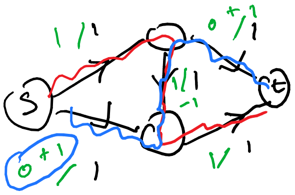
Ford-Fulkerson Algorithm Try to find $s-t$ paths that have not used their capacity and push more flow through them. There is a subtlety here though, we may run into trouble, like in the following example:



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

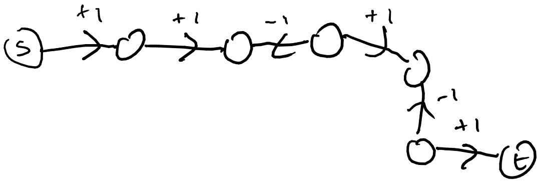


So we must change or else this algorithm won't work. We don't want to go back and change the first step, even though we are stuck. There is a way that we can change things. Say we try to add on more unit of flow:



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.

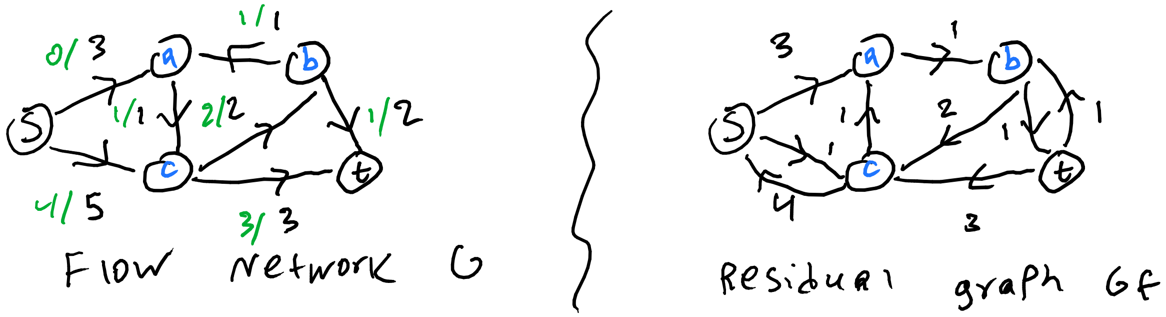


How do we implement this?

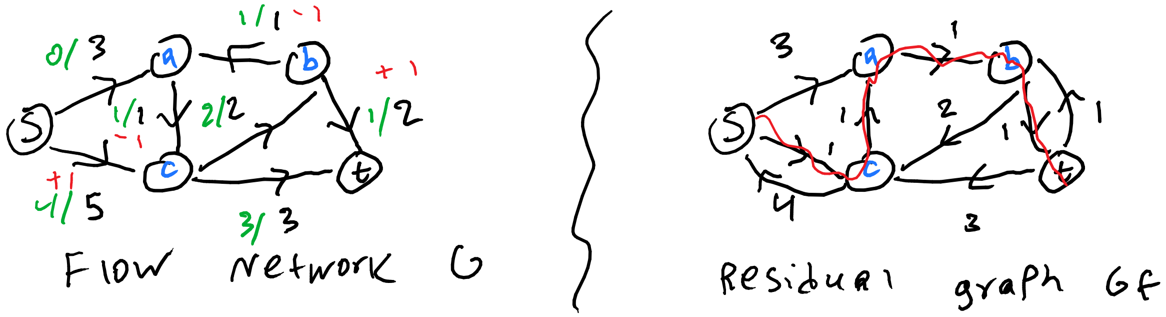
Def [Residual graph] Given a flow network $(G, s, t, \{c_e\})$ and an flow f on G, the residual graph G_f is as follows (we are already in the middle of the algorithm and this graph will tell us which edges are usable):

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



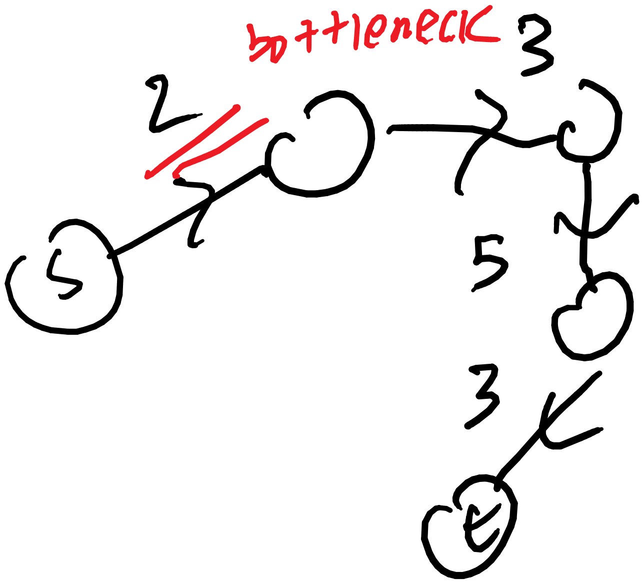
How do we use the residual graph? Just run a DFS on G_f to find an $s-t$ path and use it to modify the original flow, like so:



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.

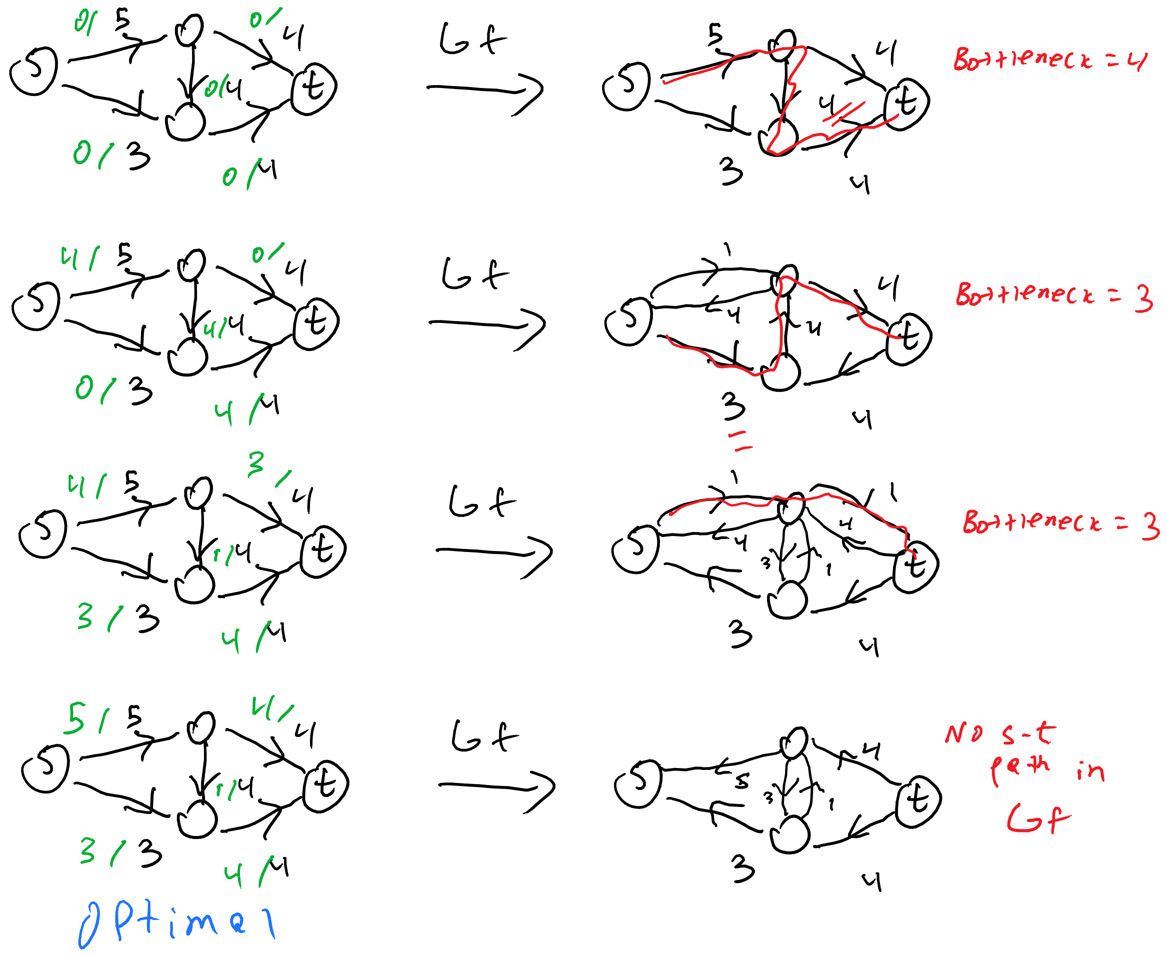
Augment(f, P)

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.

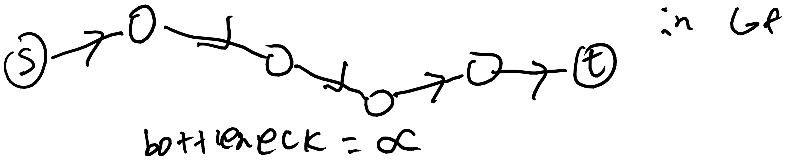
Example



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

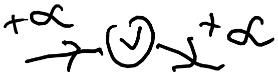
Proof Residual capacities are chosen so that updating with Augment(f, P) will never assign a number to an edge that is larger than its capacity or smaller than 0. $\implies$ capacity condition is satisfied throughout the algorithm.

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



In G:

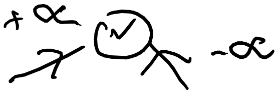
- Case 1:



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

Still the same.

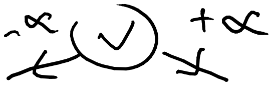
- Case 2:



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

Nothing changed.

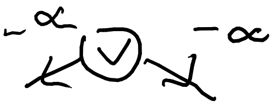
- Case 3:



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

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.

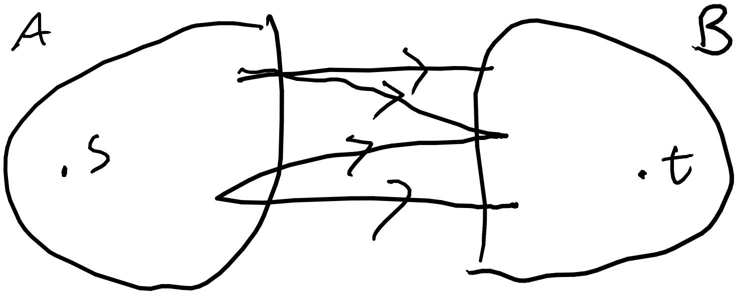
Running Time Let K be the largest capacity, n the number of vertices, m the number of edges. There are at most Km iterations. Finding an $s-t$-path: $O(m+n)$ (each iteration requires a DFS in the residual graph and an update). Augmenting: (n).
Since we assumed every vertex is adjacent to at least one 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.

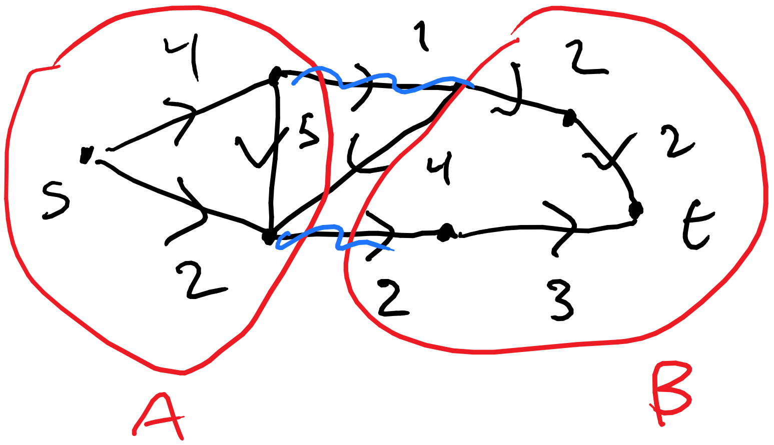
Def A cut ($s-t$-cut) in a flow network is a partition (A, B) of the vertices such that $s \in A, t \in B$.

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

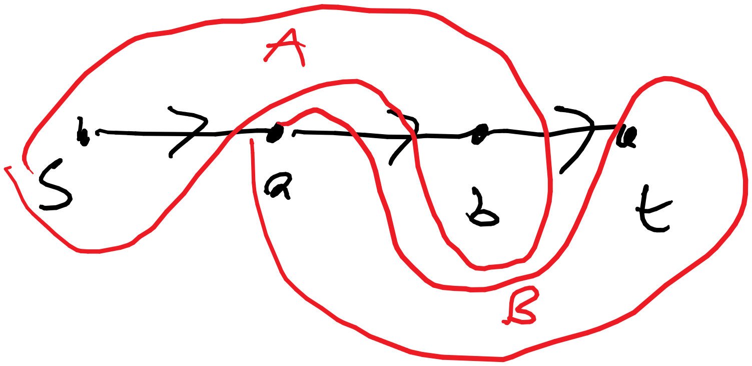


$$cap(A,B) = \sum_{\substack{uv \in E \\ u \in A \\ v \in B}} c_{uv}$$

Example



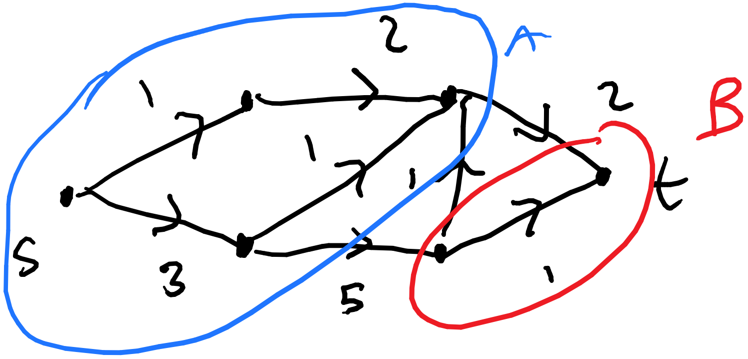
The capacity here is 3. We see that we can't pass more weight from A to B, i.e. cuts intuitively tell us something about the max flow.
How many cuts are in this network?



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: $2 \times 2 \times \ldots \times 2 = 2^{n-2}$)

3 01/15/18

Recall **Cut:** Partition of the vertices into two parts A, B such that $s \in A, t \in B$.



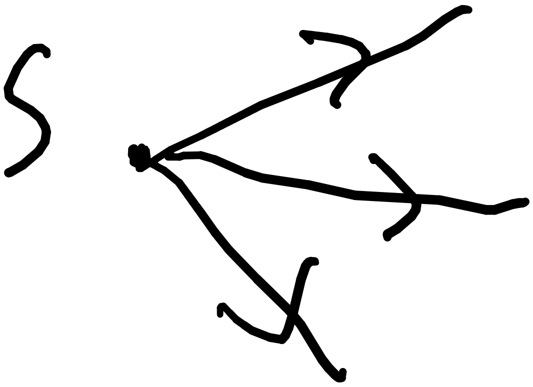
In this example, the capacity is $5+2$

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

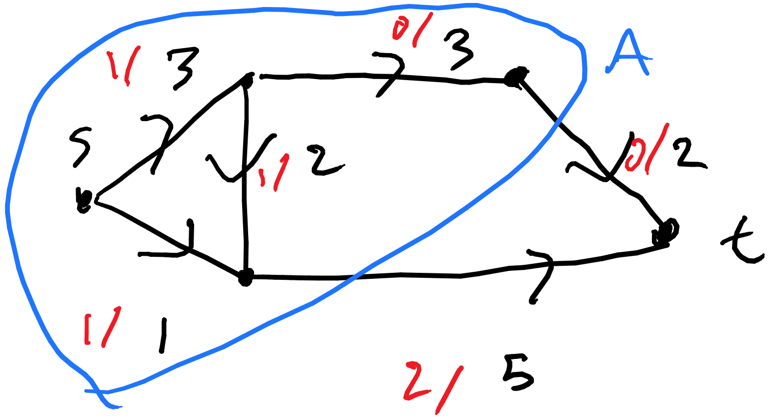
These capacities give us an upper bound on the maximum flow, but we have to prove this, intuition isn't enough. So how do we prove this?

Recall For a flow $f : E \to \mathbb{R}^+$,

$$val(f) = \sum_{su \in E} f(su)$$



Why do we define it this way? Why not talk about the flow going into the sink?

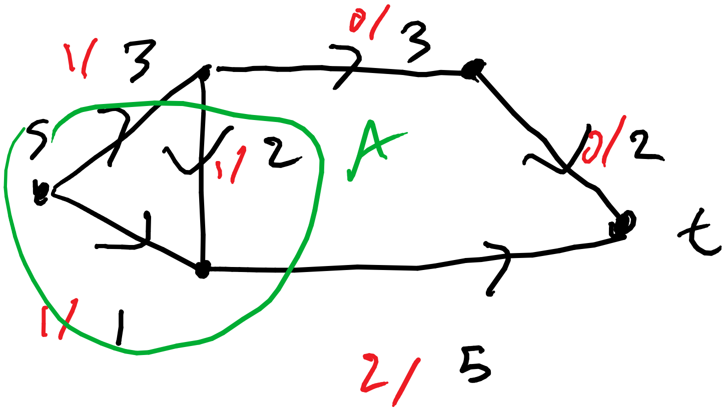


In this example we see that $val = 1 + 1 = 2$. We can also define a flow going into t. In this case it would also be 2. Are they equal? Our intuition says yes, because none of the intermediate nodes are adding or absorbing flow.

Claim For any $s-t$-cut (A,B),

$$val(f) = f^{out}(A) - f^{in}A = \sum_{\substack{uv\in E\\ u\in A\\ v\in B}} - \sum_{\substack{uv\in E\\ u\in B\\ v\in A}} f(uv)$$

In the example above, $f^{out}(A) = 1+1+0, f^{in}(A) = 0$.

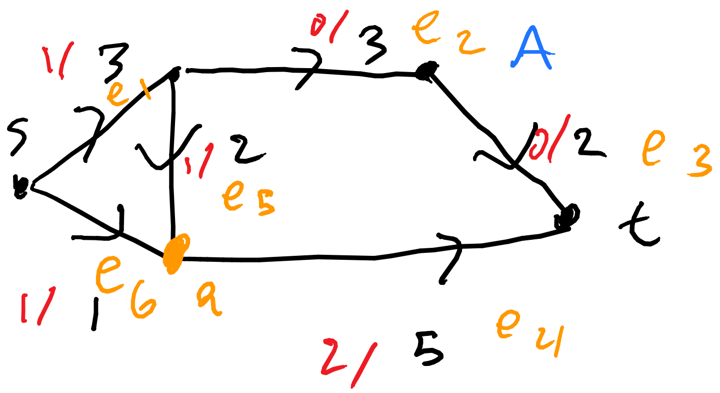


$f^{out}(A) = 1+2, f^{in}(A) = 1$

$$val(f) = \sum_{su\in E} f(su) = f^{out}(s)$$
$$val(f) = \sum_{u\in A} f^{out}(u) - f^{in}(u)$$

$f^{out} - f^{in}$ is always 0, unless u is s or t, but $t \notin A$.

$$\sum_{u\in A}\left(\left(\sum_{uv\in E} f(uw)\right) - \left(\sum_{vu\in E} f(vu)\right)\right)$$

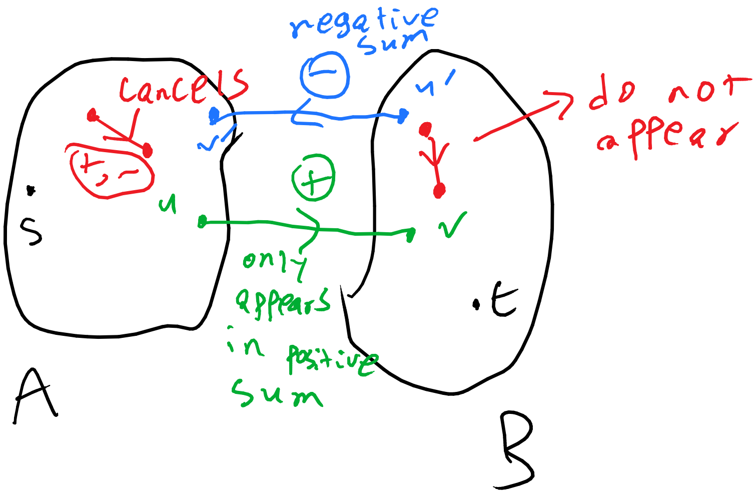


$$
\begin{aligned}
f^{out}(s) &= f(e_1) + f(e_6) \\
f^{in}(s) &= 0 \\
f^{out}(a) &= f(e_4) \\
f^{in}(a) &= f(e_6) + f(e_5)
\end{aligned}
$$

$$(f(e_1) + f(e_6) - 0) + (f(e_4) - f(e_6) - f(e_5)) = \underbrace{f(e_1) + f(e_4)}_{f^{out}(a)} - \underbrace{f(e_5)}_{f^{in}(a)}$$

Why did this come out to $f^{out} - f^{in}$?
Looking back at the double sum above: If e is an edge with both endpoints in $B \implies f(e)$ is not in the sum (since each term has at least one vertex in A). What if e has both endpoints in A? It will appear in the positive and negative sums, so they will cancel out, just like e_6 in our example.

Observations

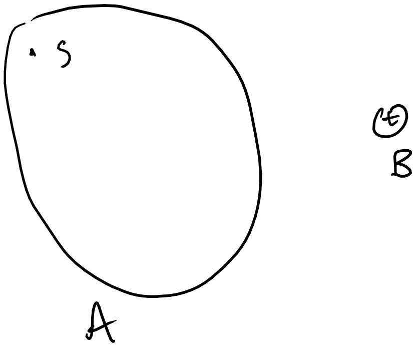


Because of some cancellations here, we can simplify the sum:

$$\sum_{u\in A}\left(\sum_{uw\in E} f(uw) - \sum_{vu\in E} f(vu)\right)$$
$$= \sum_{\substack{uv\in E\\ u\in A\\ v\in B}} f(uv) - \sum_{\substack{uv\in E\\ u\in B\\ v\in A}} f(uv) = f^{out}(A) - f^{in}(A)$$

This conclude the proof of the claim. □

Now why does $val(f) = f^{in}(t)$? Take the cut with $B = \{t\}$.



Then by the claim:

$$val(f) = \underbrace{f^{out}(A)}_{f^{in}(t)} - \underbrace{f^{in}(A)}_{0}$$

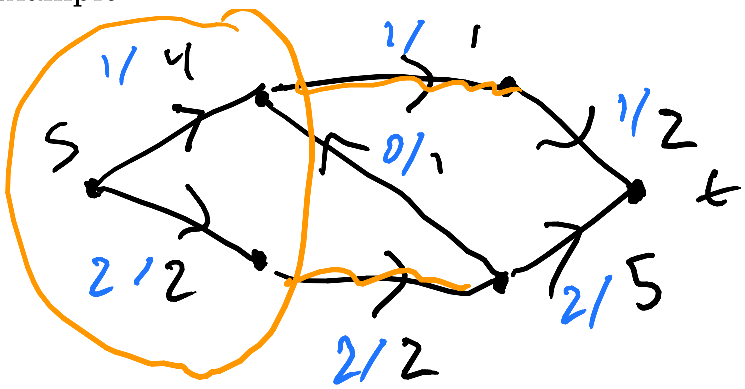
Corollary to this claim Let (A, B) be a cut, f be a flow. Then $val(f) \leq cap(A, B)$. i.e. the flow cannot exceed the capacity of the cut, any arbitrary cut puts an upper bound on the flow. How can we prove this using the previous claim?

Proof

$$val(f) = f^{out}(A) - f^{in}(A) \leq f^{out}(A) = \sum_{\substack{u\in A\\ v\in B\\ uv\in E}} f(uv) \leq \sum_{\substack{u\in A\\ v\in B\\ uv\in E}} C_{uv} = cap(A, B)$$

In other words, the flow of each edge is bounded by the capacity of each edge, but then this is just the definition of the capacity of a cut. Now why is this corollary useful? Let's look at an example.

Example

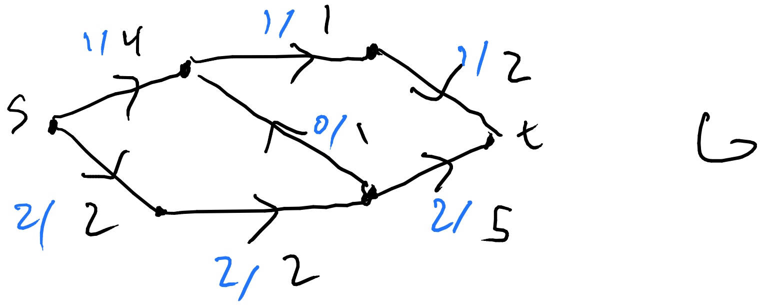


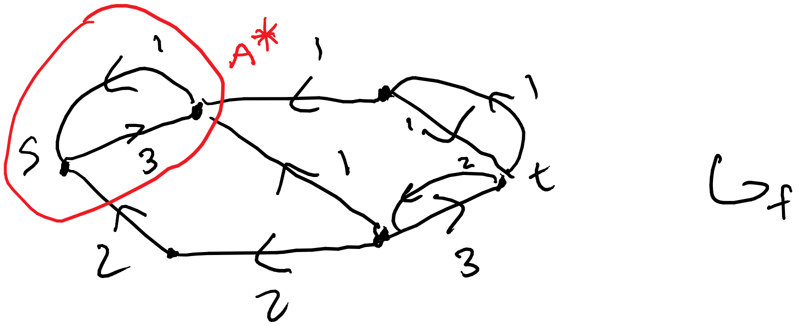
$cap(A, B) = 1 + 2 = 3$ and $val = 3 \implies$ max flow $= 3$. So if we get a flow and are asked if this is the max flow or not, either we find a flow with a better value to disprove it, or find a cut such that the capacity is the same as the flow, to prove that we can't do any better than that.

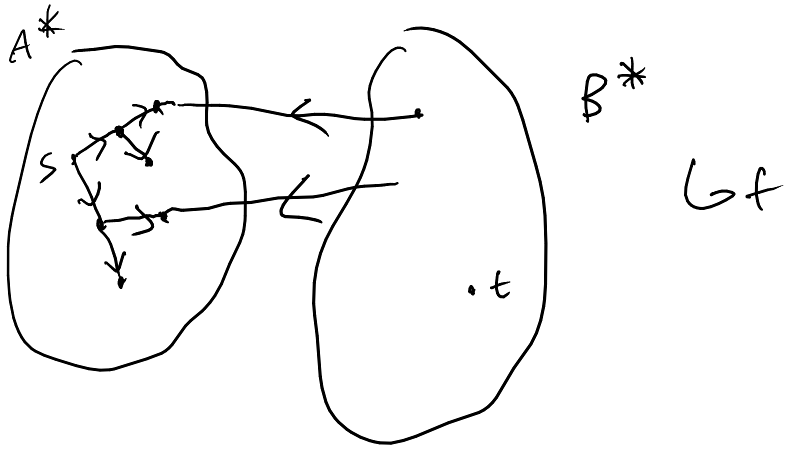
Proof of the fact that Ford-Fulkerson finds the max flow Recall:

```
FF: start with $f = 0$
while $s - t$ path $p$ in $G_f$ do
    Augment$(f, p)$
    update $G_f$
end while
```

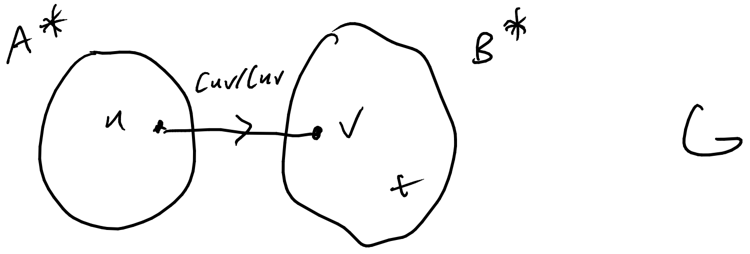
Consider the point where Ford-Fulkerson terminates. Let A^* be the set of the vertices that can be reached from S in the residual graph. Why is this a valid cut? Because at termination, there are no more $s - t$ paths, so $t \notin A^*$.



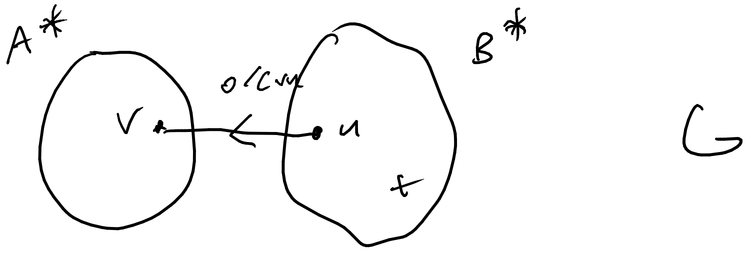




There are no edges in G_f from A^* to B^*, or else the endpoint vertex in B^* would be in A^*, because A^* consists of all the vertices we can reach from s. Thus: if uv is an edge in the original network with $u \in A^*, v \in B^*$



$f(uv) = C_{uv}$, or else uv edge would be in G_f.



$f(uv) = 0$, otherwise vu would be in G_f. Thus:

$$f^{in}(A^*) = 0$$
$$f^{out}(A^*) = \sum_{\substack{u \in A^* \\ v \in B^* \\ uv \in E}} C_{uv} = cap(A^*, B^*)$$

Therefore,

$$val(f) = f^{out}(A^*) - f^{in}(A^*) = cap(A^*, B^*)$$

So we showed that Ford-Fulkerson finds the cut that maximizes the flow, i.e. Ford-Fulkerson gives us the optimal solution. We have:

$$max\text{-}flow \leq cap(A^*, B^*) = val(f) \implies val(f) = max\text{-}flow$$

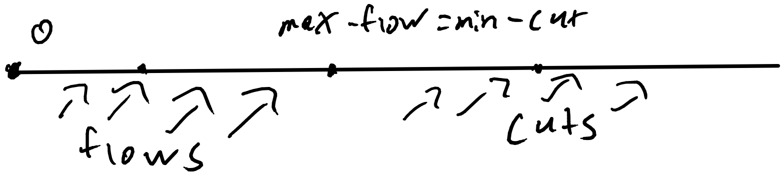
We showed that Ford-Fulkerson finds max flow. That is, when it terminates, $val(f) = max\text{-}flow$

Problem Given a flow network how can we find a min-cut? Run Ford-Fulkerson and output (A^*, B^*).

$$\underbrace{val(f)}_{\text{any flow } f} \leq max\text{-}flow \leq min\text{-}cut \leq cap(A^*, B^*)$$

When we run Ford-Fulkerson we find f with $val(f) = cap(A^*, B^*)$

$$\implies val(f) = \textbf{max-flow} = \textbf{min-cut} = cap(A^*, B*)$$



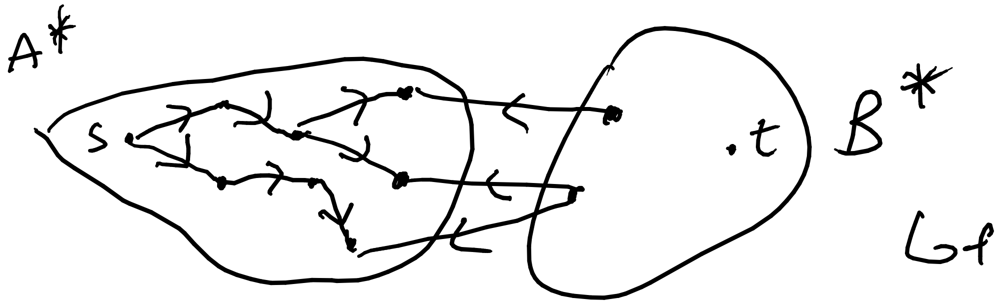
Thm For any flow network:

$$max\text{-}flow = min\text{-}cut$$

4 01/17/18

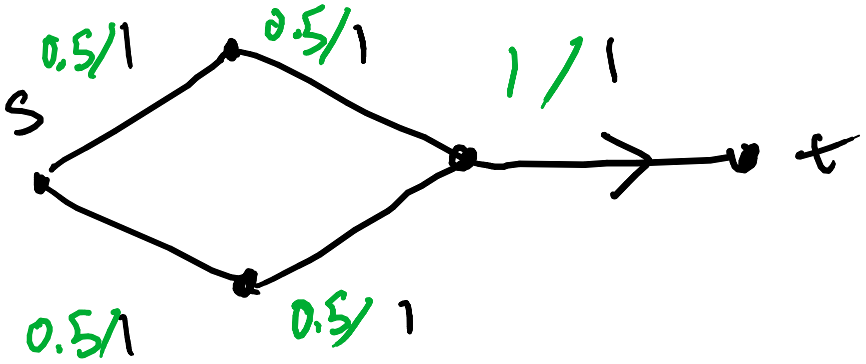
Recall

- Ford-Fulkerson finds the max flow.
- $max\text{-}flow = min\text{-}cut$. Kind of unexpected/unintuitive that they'd be equal. It's pretty intuitive that *min-cut* is an upper bound, but it's surprising that they are equal.
- Ford-Fulkerson runs in $O(m^2K)$, where m is the number of edges, K is the largest capacity of an edge. Can be quite slow if the largest capacity is big.
- $val(f) = f^{out}(s) = f^{in}(t) = f^{out}(A) - f^{in}(A)$ for all cuts (A, B)
- Ford-Fulkerson can be used to find *min-cut*.



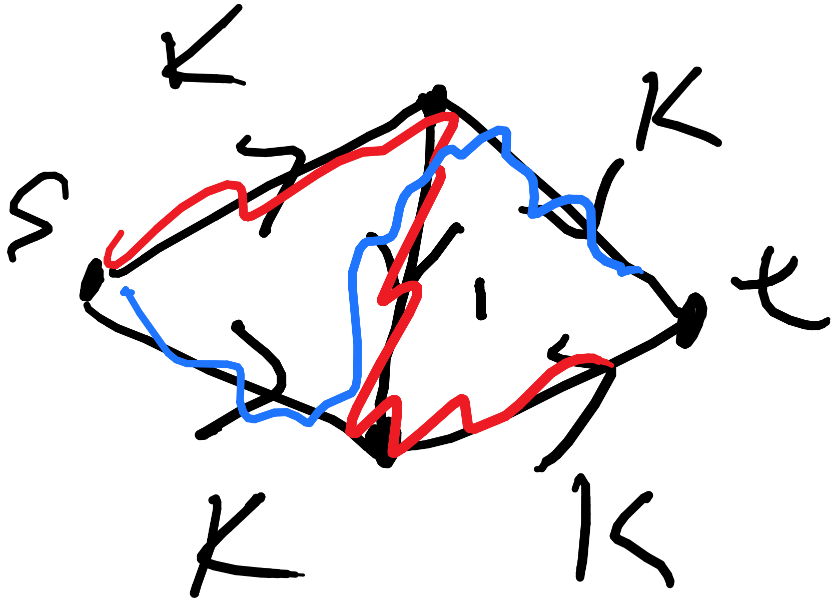
Can't reach t from s at the end of the algorithm in the residual graph.

Question (Recall all capacities are integers) Is it possible to have a *max-flow* that assigns non-integer values to some of the edges? (Remember that the flow function is defined as $f : E \to \mathbb{R}^+$) Yes, it is possible:



Question Is there always an all integer *max-flow*? Yes because Ford-Fulkerson always outputs integer valued flows and we know that it finds *max-flow*. i.e. there is at least one all integer *max-flow*, the one that can be found by Ford-Fulkerson and we already proved that it gives *max-flow*. If you try to prove this directly, it seems very hard unless you come up with something like Ford-Fulkerson. So we have obtained many important consequences and applications from analyzing Ford-Fulkerson.

Remark The running time $O(m^2K)$ is not efficient when K is a large number. Input size: $\Theta(m \log k)$, since we have m edges each that require as much as $\log k$ bits to write each number between $1 - K$. (This is an exponential time algorithm)



Running Ford-Fulkerson on this graph would require 2^K path augmentations, alternating between the red and blue path. So we want to get rid of this and improve it.

4.1 A Faster Ford-Fulkerson

Possible Approaches

1. Always pick the shortest path from s to t. This will work and leads to an efficient (polytime) algorithm. We will not discuss it here. Pretty easy to implement too, just run a BFS instead of a DFS.

2. Try to go with the paths that increase the flow by larger numbers. In the above example, we see that the red path only increases flow by 1, instead of the top path that can increase it by K. This is called the Fattest Path approach, where we find an augmenting path with the largest bottleneck. However, there is a bit of a problem here, finding this path is a bit complicated and not fast. (There is a way to implement it by modifying Dijkstra's, but not so fast)
   The problem with the first proposed solution is that it can't be analyzed easily (although it can be implemented easily), whereas the second solution can be analyzed

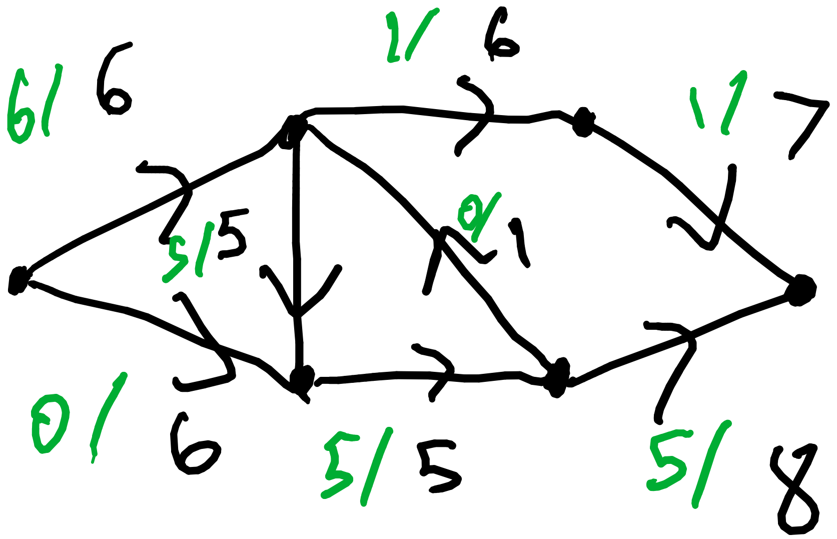
easily but not easily implemented.

We will do something similar:

High level description

```
Initially set $\Delta = 2^{\lceil \log_2 k \rceil}$, that is $\Delta$ is the smallest power of 2 that is at least $K$. (e.g. $K = 13 \implies \Delta = 16, K = 17 \implies \Delta = 32$)
while $\underline{\text{there are augmenting paths with bottleneck} \geq \Delta}$ do use them to augment the flow
    When we run out of these we set $\Delta \leftarrow \frac{\Delta}{2}$
    If $\Delta = 1$ here (when we want to decrease it) then stop.
end while
```

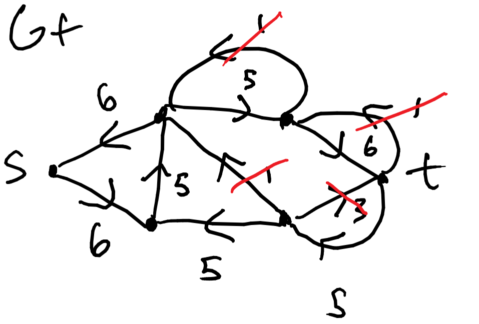
How can we check the underlined condition, that there are augmenting paths with bottleneck greater than Δ?

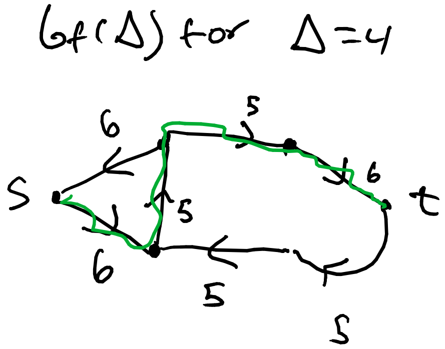


with $\Delta =$

4.

In this case, when we build the residual graph we will exclude edges that have weight less than 4. Let $G_f(\Delta)$ be the subgraph of G_f consisting only of the edges with residual cap $\geq \Delta$. We just need to find an $s - t$ path in $G_f(\Delta)$.





Here bottleneck$\geq \Delta$, we can increase the flow by 5 here.
We call this scaling.

Scaling Ford-Fulkerson

```
set $\Delta = 2^{\lceil \log_2 k \rceil}$, where $K$ is the largest capacity.
set $f = 0$, construct $G_f$
while $\Delta \geq 1$ do
    while $\exists$ an $s - t$ path $P$ in $G_F(\Delta)$ do
        Augment$(f, p)$
        update $G_f$
    end while
    $\Delta \leftarrow \frac{\Delta}{2}$
end while
```

Running Time

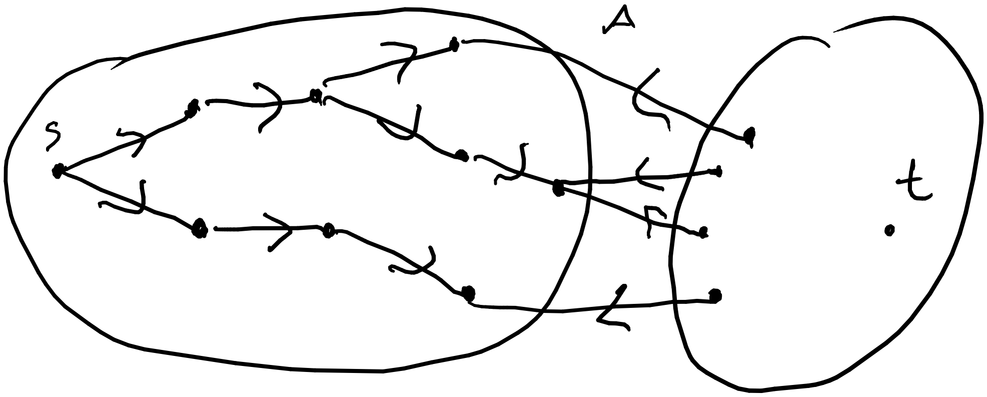
- Checking if there exists an $s - t$ path: $O(m)$
- Augmenting, $O(m)$
- Updating G_f, $O(m)$

So we need to understand the number of iterations. The other loop has $\lceil \log_@ K \rceil$ iterations. The inner loop? (actually will be a bit of work to analyze this.) How many times in the $\Delta - phase$?

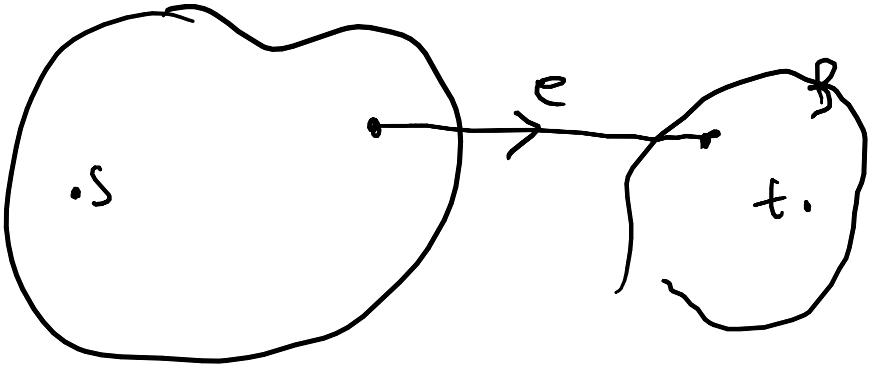
Claim Let f be the flow at the end of the Δ-phase (when no $s - t$ paths are in $G_f(\Delta)$). There is a cut (A, B) such that

$$max\text{-}flow \leq Cap(A, B) \leq val(f) + m\Delta$$

Proof Let A be the set of all nodes that can be reached from S in $G_f(\Delta)$ (very similar to *min-cuts* before)

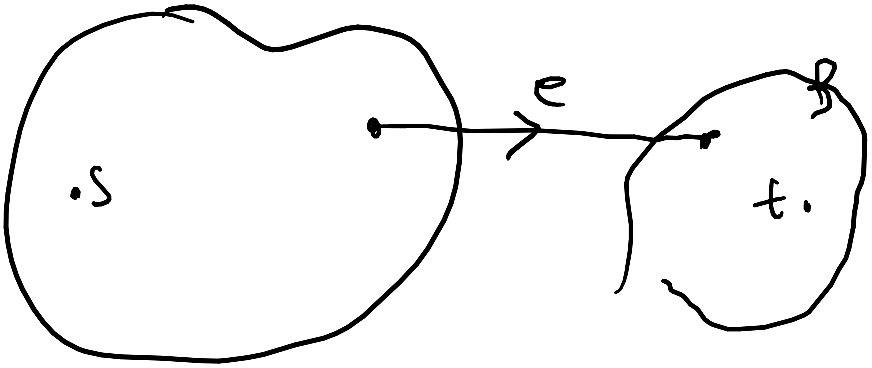


(No edge from A to B in $G_f(\Delta)$, otherwise A would have been extended further)
If e is an edge from A to B in the original network:



$$f(e) \geq c_e - \Delta$$
$$c_e - f(e) < \Delta$$

If e goes from B to A:



$$f(e) < \Delta$$

Or else we could expand A.

$$\begin{aligned}
val(f) = f^{out}(A) - f^{in}(A) &= \sum_{\substack{e\ from\\ A\ to\ B}} f(e) - \sum_{\substack{e\ from\\ B\ to\ A}} f(e)\\
&\geq \sum_{\substack{e\ from\\ A\ to\ B}} (c_e - \Delta) - \sum_{\substack{e\ from\\ B\ to\ A}} \Delta = \sum_{\substack{e\ from\\ A\ to\ B}} c_e - \sum_{\substack{e\ from\\ A\ to\ B\\ or\ B\ to\ A}} \Delta\\
&= Cap(A,B) - m\Delta \implies val(f) \geq Cap(A,B) - m\delta
\end{aligned}$$

□ So we showed

$$val(f) \geq Cap(A,B) - m\Delta \geq max\text{-}flow - m\Delta$$

Let's look at the flow at the end of the previous phase.

$$Val(f_{prev}) \geq max\text{-}flow - 2\Delta m$$

(since we halved Δ)
How many augmentations can we have in the Δ-phase? We can have at most $2m$ augmentations in this phase because each one increases the value by at least Δ and starting from $max\text{-}flow - 2m\Delta$ we cannot go above $max\text{-}flow$. So the number of iterations of this is good

as it only depends on m.
Back to the analysis, we figured out that the inner loop has $\leq 2m$ iterations. So the total running time is:

$$O(\log_2 K \times m \times m) = O(m^2 \log K)$$

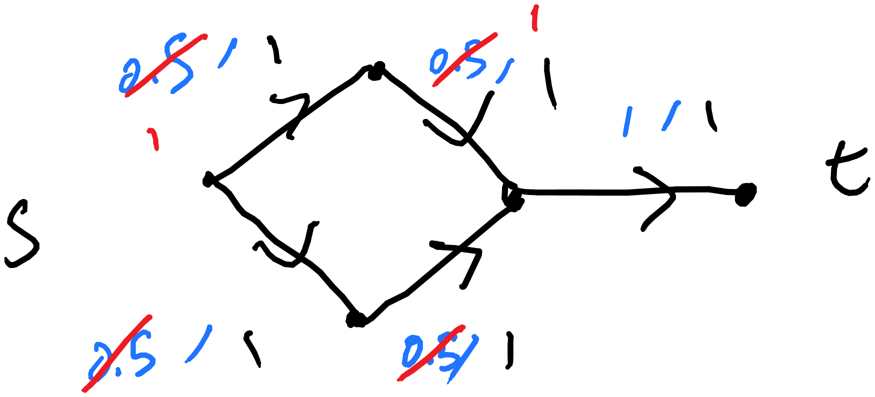
Instead of $O(m^2K)$ of the naive Ford-Fulkerson. This is a big improvement when K is a huge number.
One thing is left: Why does this algorithm find the *max-flow*? Because when it terminates, $\Delta = 1$ and it means there are no more augmenting $s - t$ paths in the residual graph.

Remark This is a special instance of Ford-Fulkerson $\implies$ it finds *max-flow*.

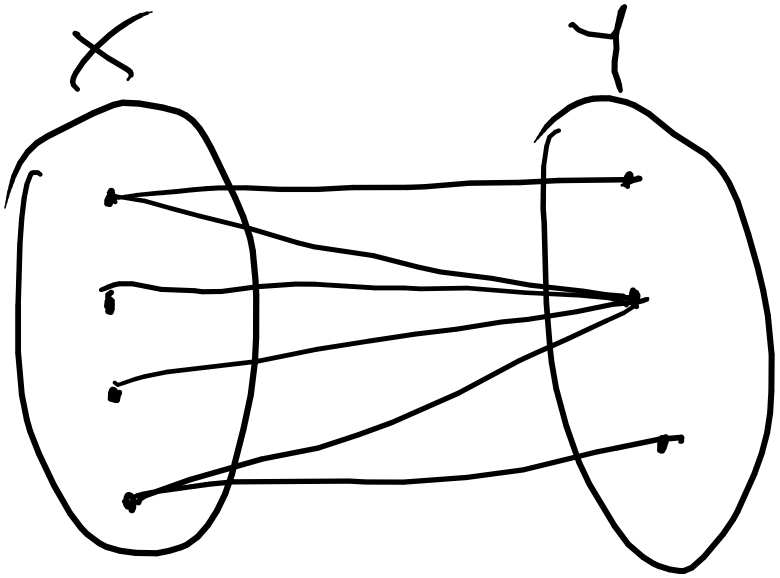
5 01/22/18

Recall

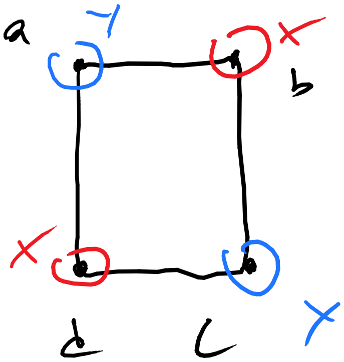
- Ford Fulkerson finds max-flow $O(m^2K)$.
- **Scaling Ford Fulkerson** finds max-flow $O(m^2 \log k)$, k =largest capacity.
- *Max-flow* = *Min-cut*
- There is a *max-flow* that assigns integer flows to all edges.

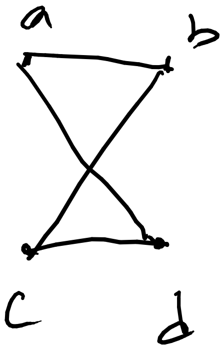


Bipartite Graph is an undirected graph such that the vertices can be partitioned into two parts X and Y such that all the edges are between X and Y.

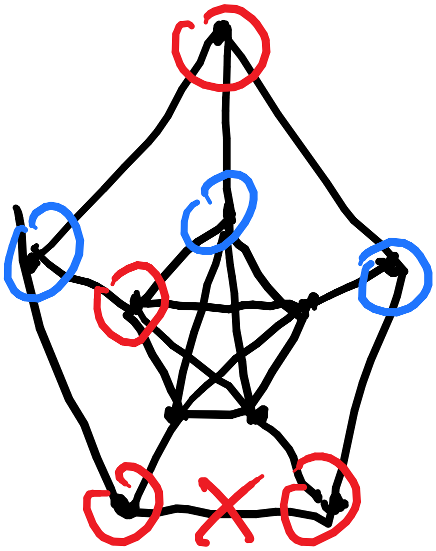


Examples Is this bipartite? Yes. We can check by just 2 coloring.



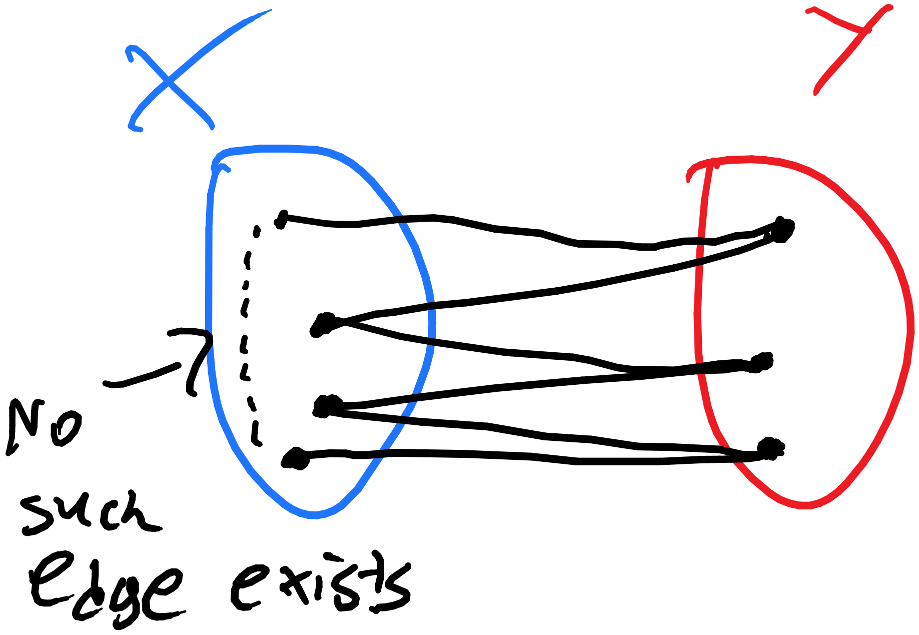


What about the Peterson graph? No.



We can just color it until we reach a contradiction.

A graph is bipartite $\iff$ it does not have any odd cycles. Trivially a bipartite graph does not have an odd cycle:

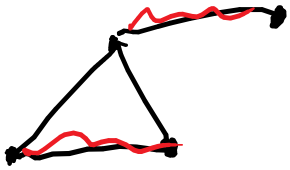


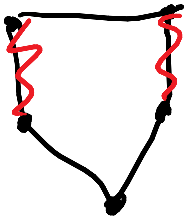
5.1 Largest Matching Problem

Def A matching is a set of edges, no two of them share an endpoint.

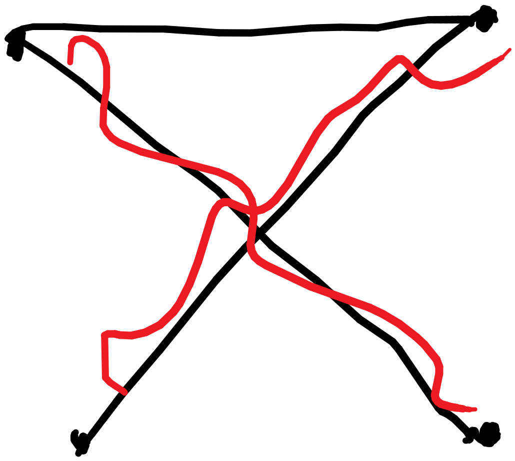
Def A perfect matching is a matching that includes all vertices.

Ex The maximum matching of both of these graphs is 2:



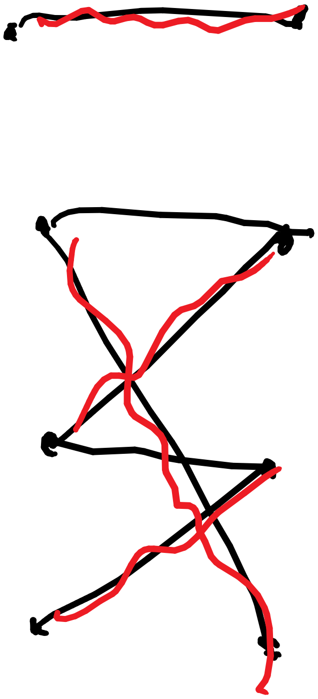


Note that there is nothing geometric about a matching, it does not matter how you draw the graph, the fact that the edges "cross" over each other does not matter, like here:



Given a bipartite graph with parts X and Y, how can we find the largest matching in G?

Ex Maximum matching here is 4:



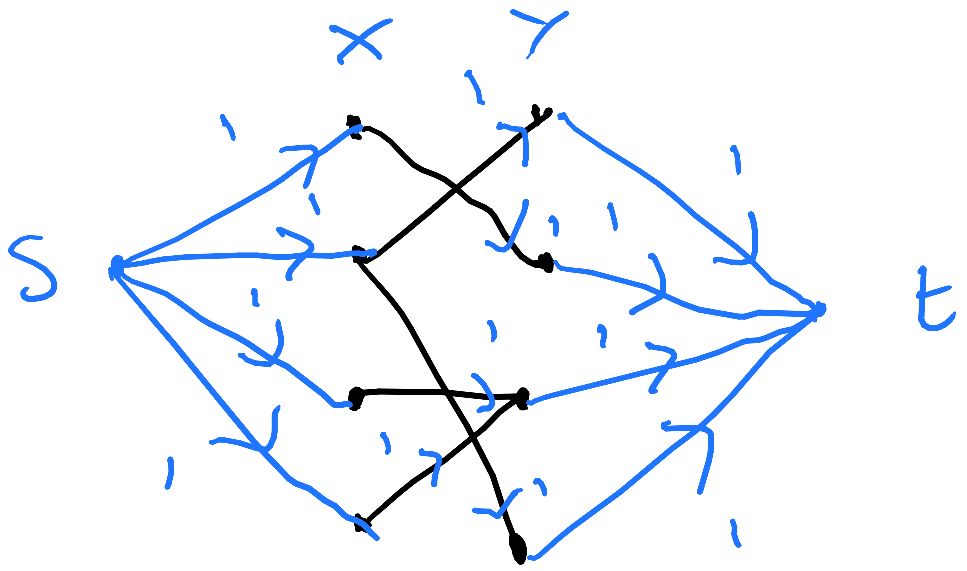
Greedy algorithms won't work here, so how should we solve this?

Why do we care about this problem? It has a very practical application, you can imagine it as a pairing of objects or something like pairing people with jobs, where every person has specific traits for certain jobs and we want to give as many people as possible a job, although each person can only have one.

We want to use the max flow problem here. How will we do so?

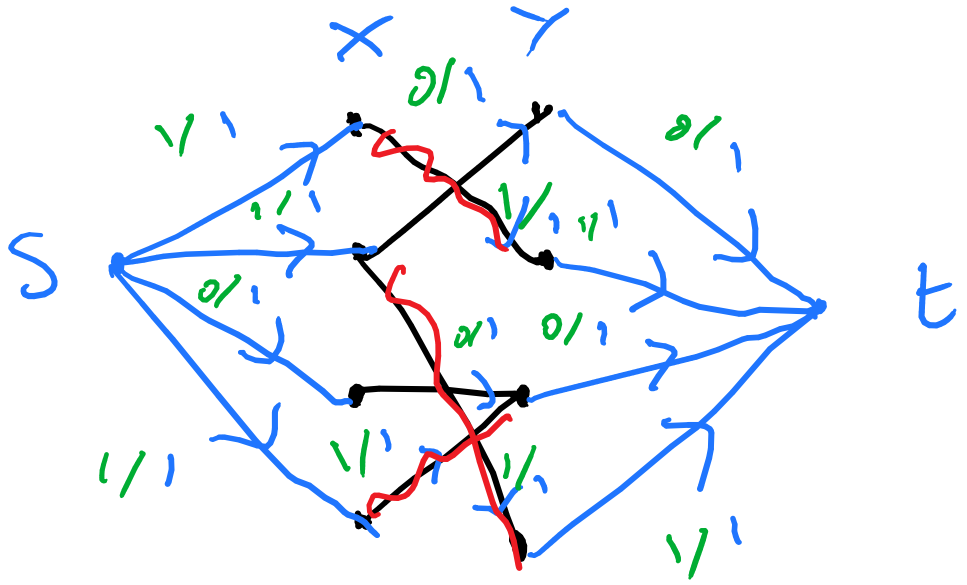
Solution We construct a flow network in the following manner:

1. We direct all the edges from X to Y.
2. We add two new vertices called s and t.
3. We put edges from s to all vertices in X and edges from all vertices in Y to t.
4. Assign capacity 1 to all the edges.

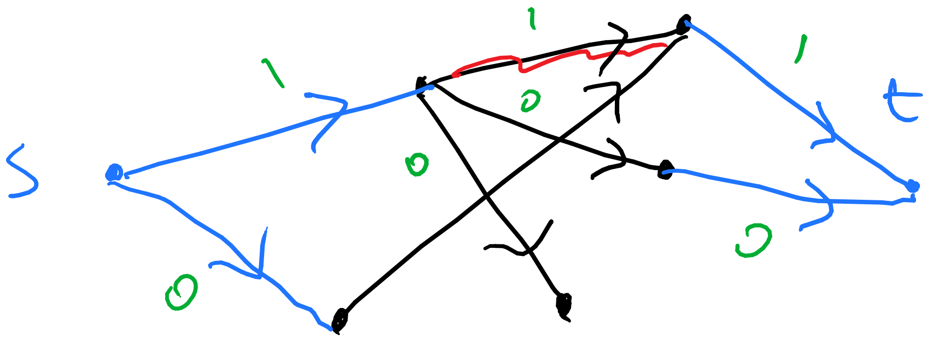


Claim Max flow in this network = max matching in G.

Proof First we show max matching (M) is at least max-flow.



We assign a flow of 1 to all edges in M and 0 to all other edges between X and Y. For the edges starting from s or ending at t, the ones that go to vertices in M get a value of 1 and the rest 0.



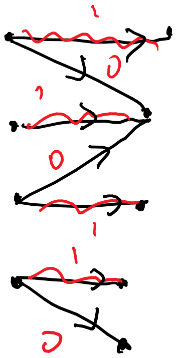
Why is this a valid flow? We aren't overloading capacities (only assigning 0 or 1) and we conserve flow going out of s to flow going into t, since every edge in M has 1 vertex with an edge coming from s and one with an edge going to t. The value of this flow is $|M|$. This is because there are $|M|$ vertices in X involved in M and we assign 1 to the edges from s to those vertices.

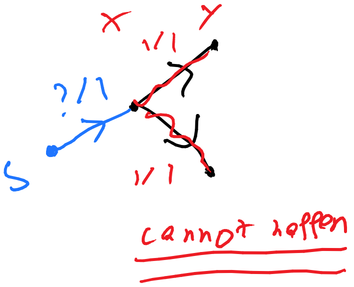
Now we want to do the opposite, convert the max-flow to a matching.

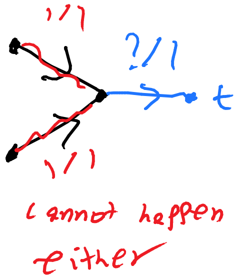
Next we need to show that there is a matching of size max-flow (assuming all edges are assigned either 1 or 0 flow, making the existence of an integer value flow proved last class important).

There is a max-flow with integer values. Thus all edges will have a flow of 0 or 1 (the capacities are all 1).

The edges between X and Y with 1 unit of flow on them form a matching.







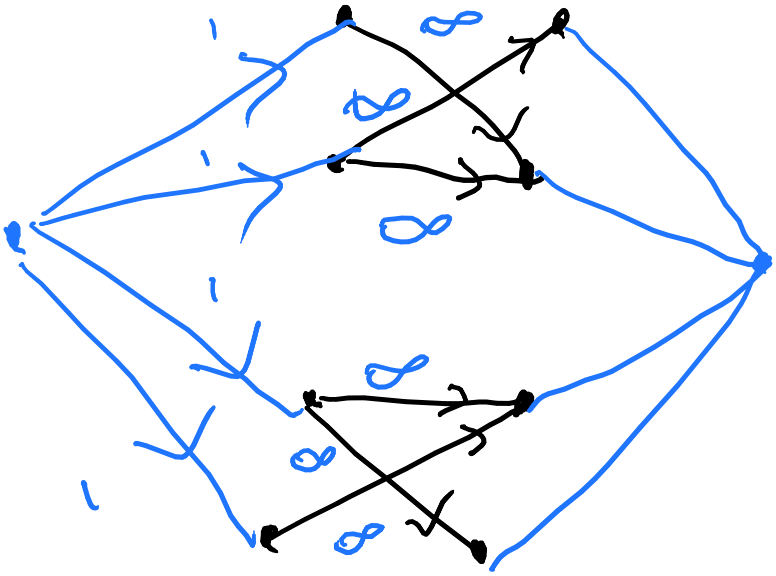
To summarize, we showed

$$|M| \leq max\text{-}flow$$

and

$$\textit{max-flow} = \text{some matching} \leq \text{max matching} = |M| \implies |M| = \textit{max-flow}$$

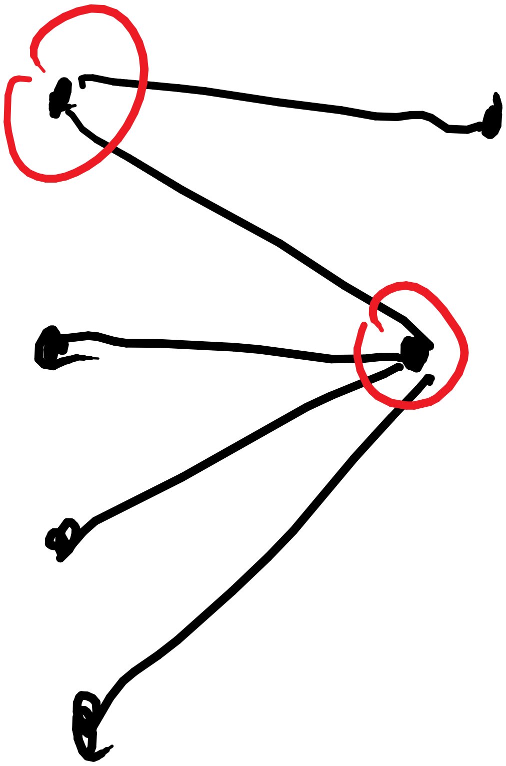
Remark Note that in the above proof we could assign ∞ capacities (instead of 1 to all) to edges between X and Y.



The incoming flow of all vertices in X will be at most 1 so the edges between X and Y will never have any flow > 1. This will be useful when considering *min-cut*, as it eliminates many cuts.

We know *max-flow* = *min-cut*. What does this mean in this context? (matching)

Def A vertex cover is a set of vertices such that removing them will remove all the edges.

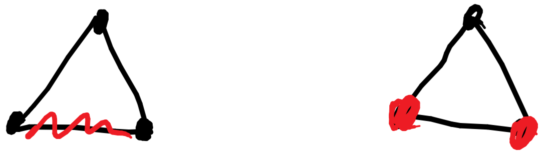


Here, the red vertices form a vertex cover. How can we find the smallest vertex cover? If we want to think of this practically, we can think of the vertices as monitors such that we can monitor all the connections/roads.

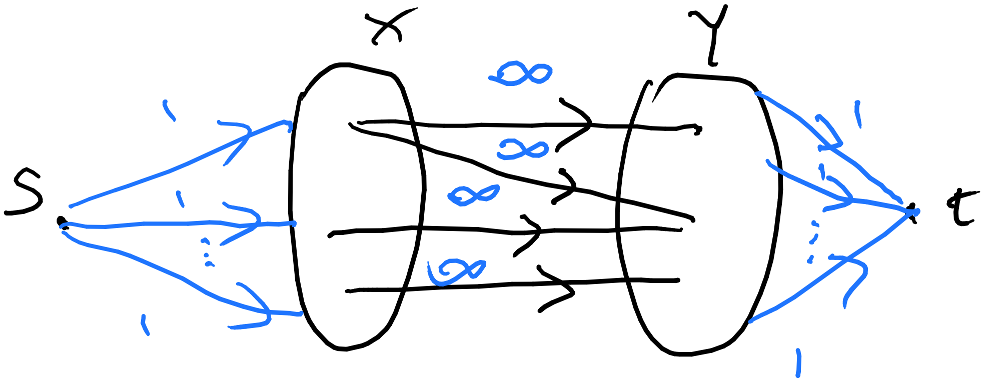
Thm In every bipartite graph max matching = min vertex cover.

Remark Note that if a graph has a matching of size k, then ever vertex cover needs to pick at least one vertex from each of these k edges and thus is of size $\geq k$.

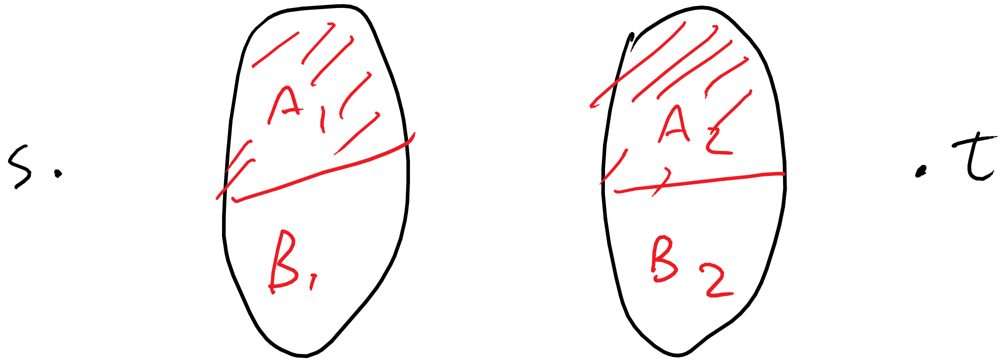
Remark The equality is not true in general graphs.



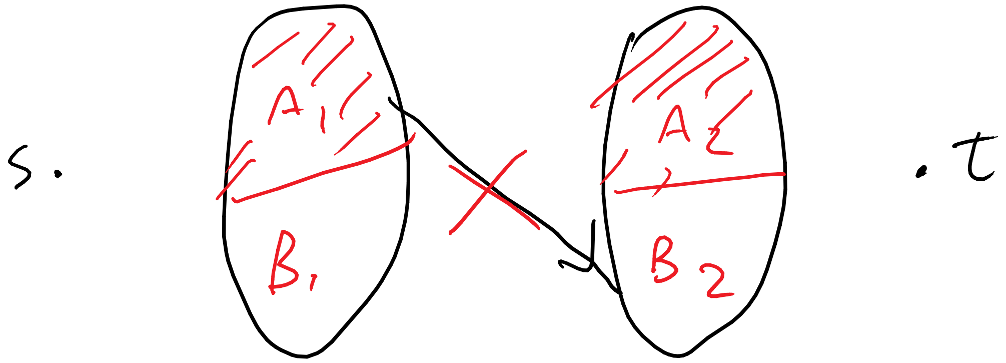
Here the max matching is 1 but the minimum vertex cover is 2.



Lets look at the min cut (A, B) (it is not ∞, can easily show one that isn't). Some vertices of A are in X (A_1), others are in Y (A_2), etc.

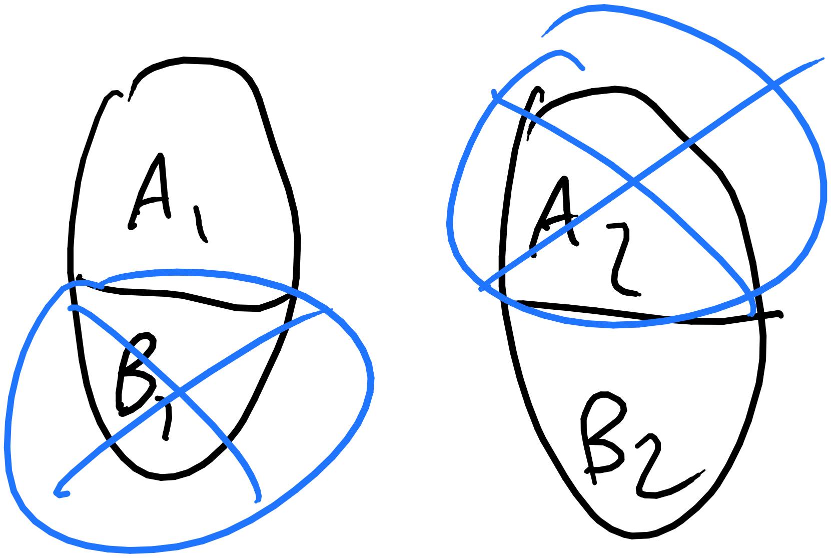


$A = \{s\} \cup A_1 \cup A_2, B = B_1 \cup B_2 \cup \{t\}$



No edges from A to B with ∞ cap $\implies$ no edges from A_1 to B_2. How can we use this to make a minimum vertex cover?

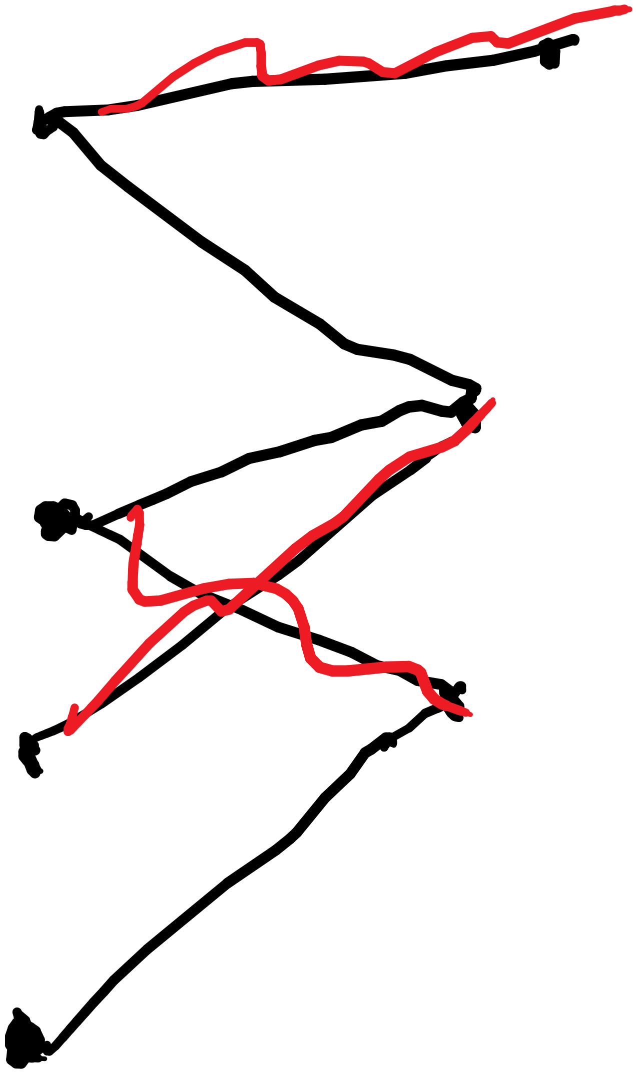
Thus $B_1 \cup A_2$ is a vertex cover in G.



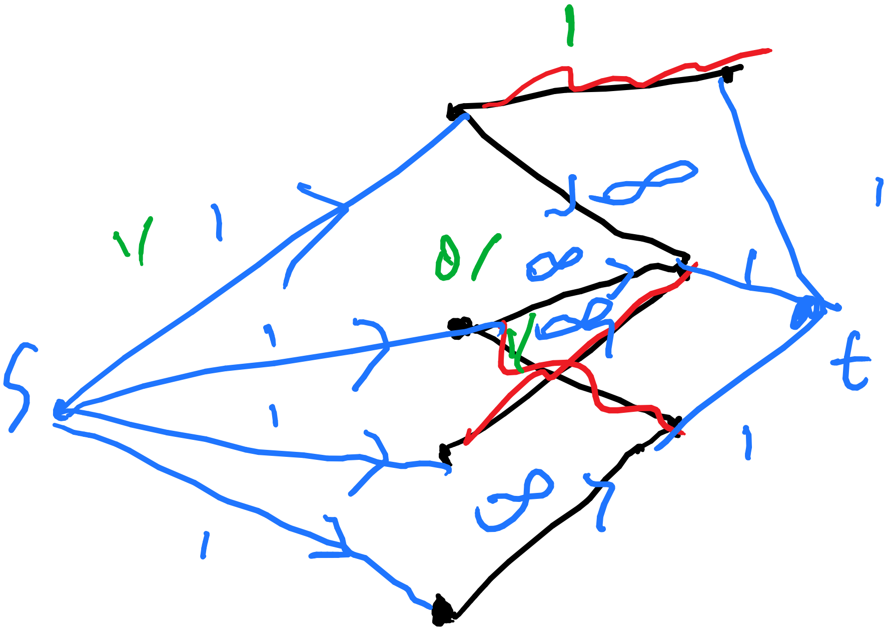
The only edges that could remain are from A_1 to B_2, but we just showed that those couldn't exist (to continue next class).

6 01/24/18

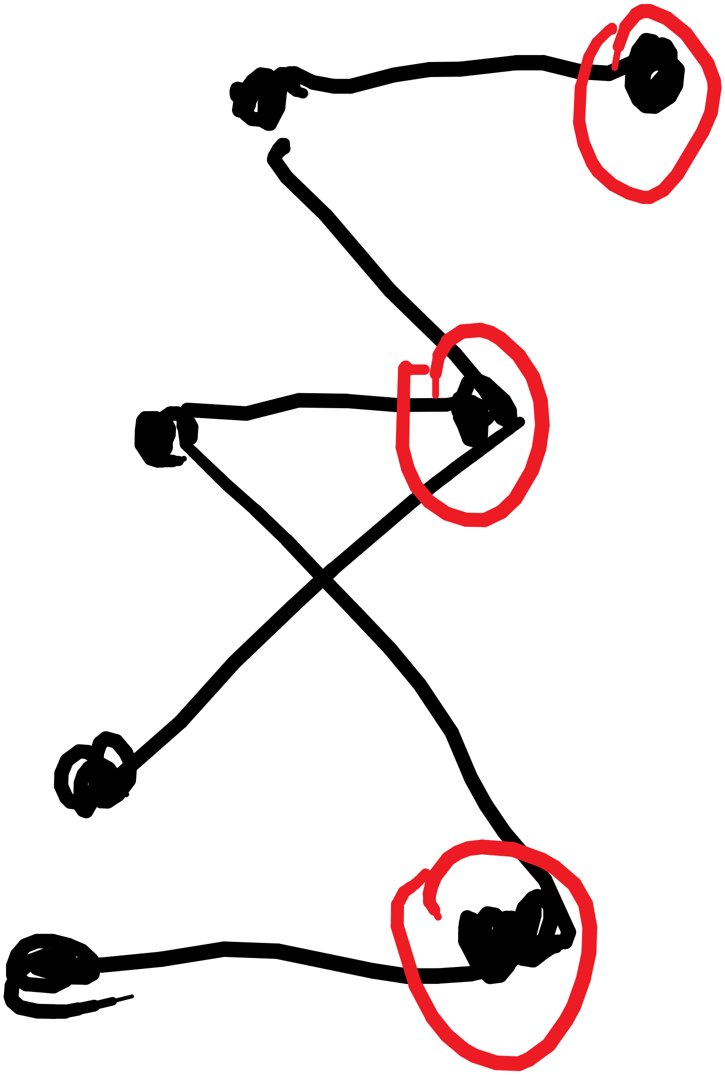
Recall Matching in Bipartite graphs.



Ford Fulkerson can be used to find the largest matching in a Bipartite graph.


s
t
1
1
1
1
1
1
1
1
1
∞
∞
∞
∞
0/1
1/1
1/1

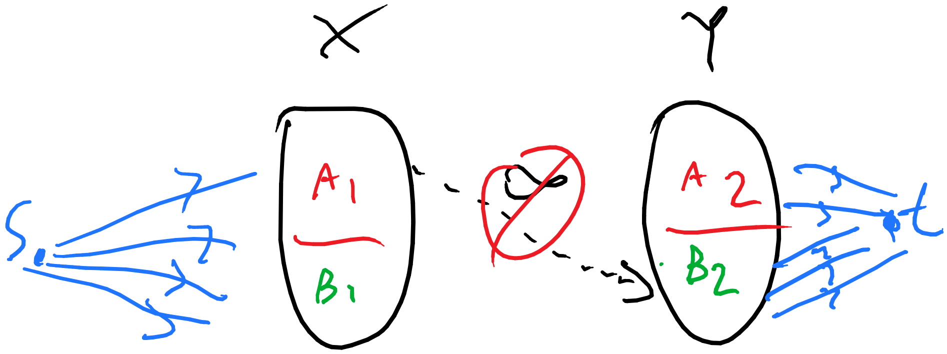
Vertex Cover A set of vertices such that deleting them will remove all the edges.



Thm For every bipartite graph G we have

$$\min \mathrm{VC} = \textit{max-matching}$$

Pf Consider a *min-cut* (A, B) in the constructed flow network.



$A = \{s\} \cup A_1 \cup A_2, B = \{t\} \cup B_1 \cup B_2$

No edges from A_1 to B_2 as otherwise the capacity at the cut would be ∞. Thus $B_1 \cup A_2$ is a vertex cover in the original graph. Its size is $|B_1| + |A_2|$. On the other hand,

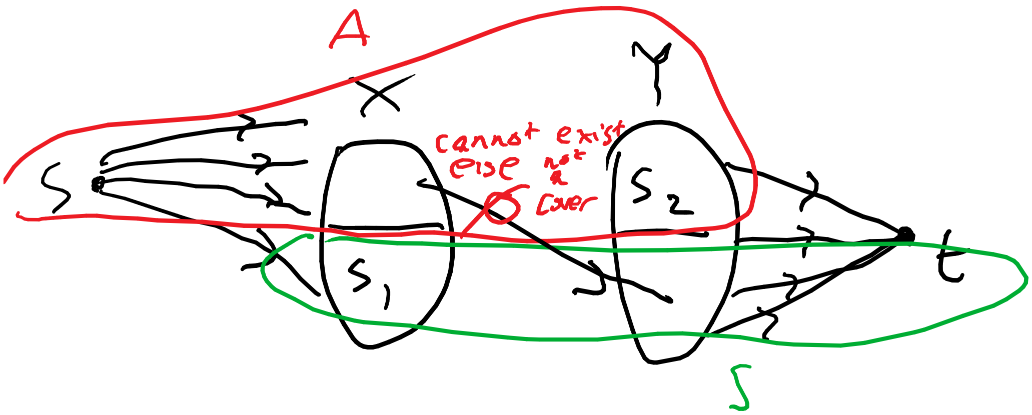
$$Cap(A, B) = \sum_{\substack{\text{edges from} \\ s \text{ to } B_1}} c_3 + \sum_{\substack{\text{from} \\ A_2 \text{ to } t}} = |B_1| + |A_2|$$

We showed that there is a vertex cover $(B_1 \cup A_2)$ whose size is equal to *min-cut* (A, B) $(\textit{min-}VC \leq \textit{min-cut})$.

Next we will show that

$$\textit{min-cut} \leq \textit{min-vc} = |S| = |S_1| + |S_2|$$

Let S be the smallest vertex cover.



$S = S_1 \cup S_2, B = A^c$

Let $A = (X - S_1) \cup S_2 \cup \{s\}$. $Cap(A, B) = \underbrace{|S_1|}_{\text{edges from } S \text{ to } S_1} + \underbrace{|S_2|}_{\text{edges from } S_2 \text{ to } t}$

We conclude

$$Max\text{-}flow = Min\text{-}Cut = Min\text{-}Vc = Max\text{-}Matching$$

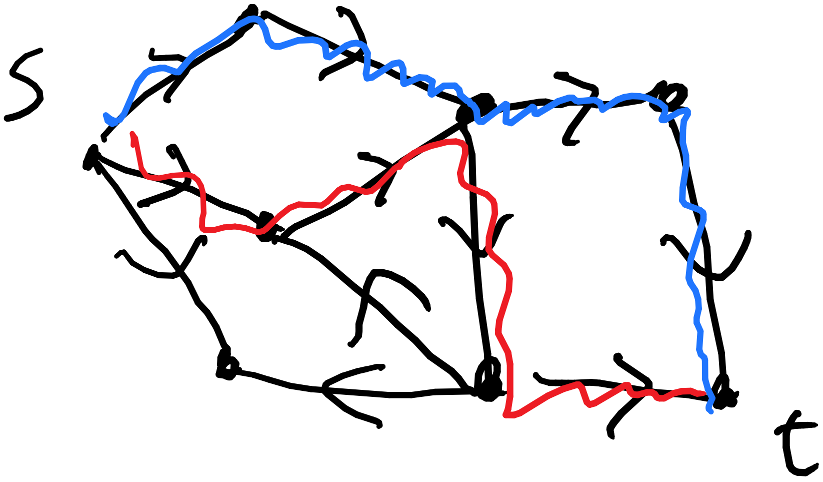
Thm (König) In a bipartite graph $Max\text{-}Matching = Min\text{-}Cut$

6.1 Disjoint Paths in directed graphs

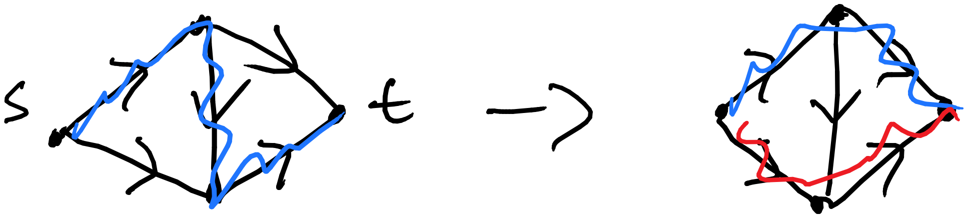
Input: A directed graph and two distinct nodes are marked as $\underline{s}$ and $\underline{t}$.

Goal: Find the maximum number of edge-disjoint paths from s to t.

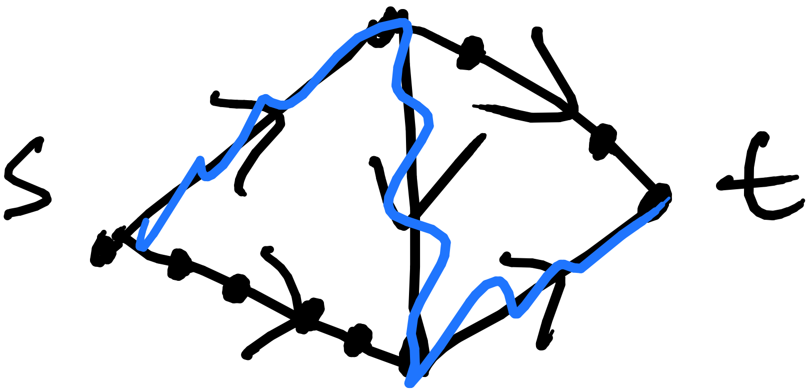
Ex



Could we just use BFS or DFS to find an s-t path? No, it might chose the wrong edges like in this example:

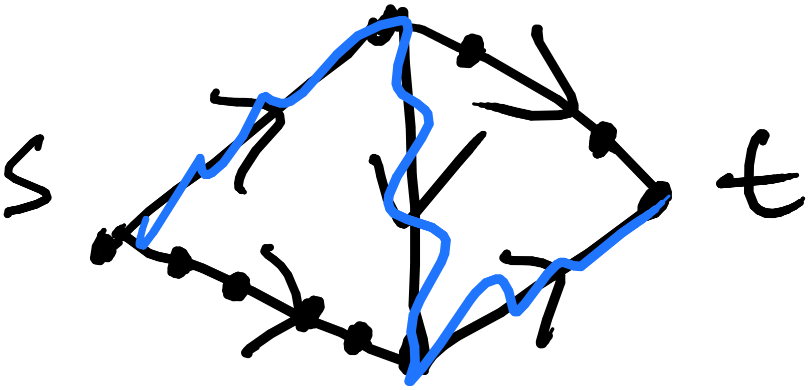


What if we chose the shortest path? That would also be problematic:



How do we solve this then? We assign capacity 1 to all the edges and run the Ford Fulkerson algorithm (note that we explicitly specify using Ford Fulkerson so we get integer values, not just max-flow).

Ex

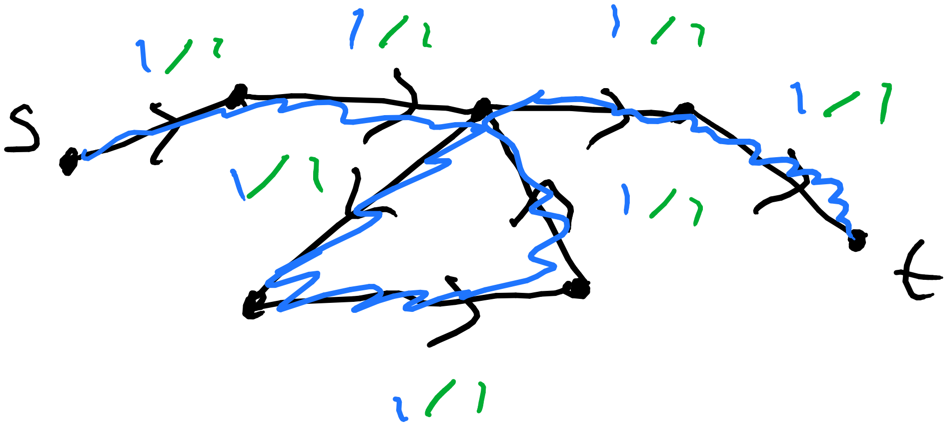


So the max-flow here is 3. How do we show there are 3 edge-disjoint paths?

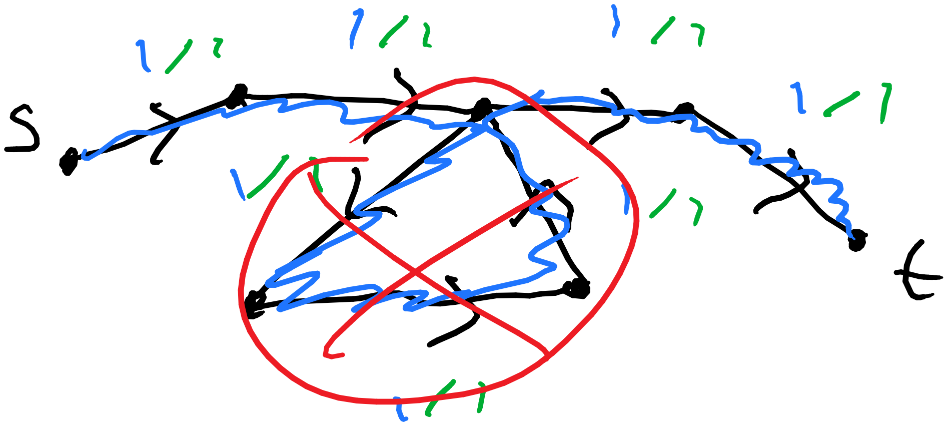
We solved the *max-flow* using Ford Fulkerson and let k be the value of *max-flow*. We want to show that there are k edge-disjoint paths from s to t.

Let's start with $k = 1$. In this case we have a flow of 1. We want to find one s-t path.

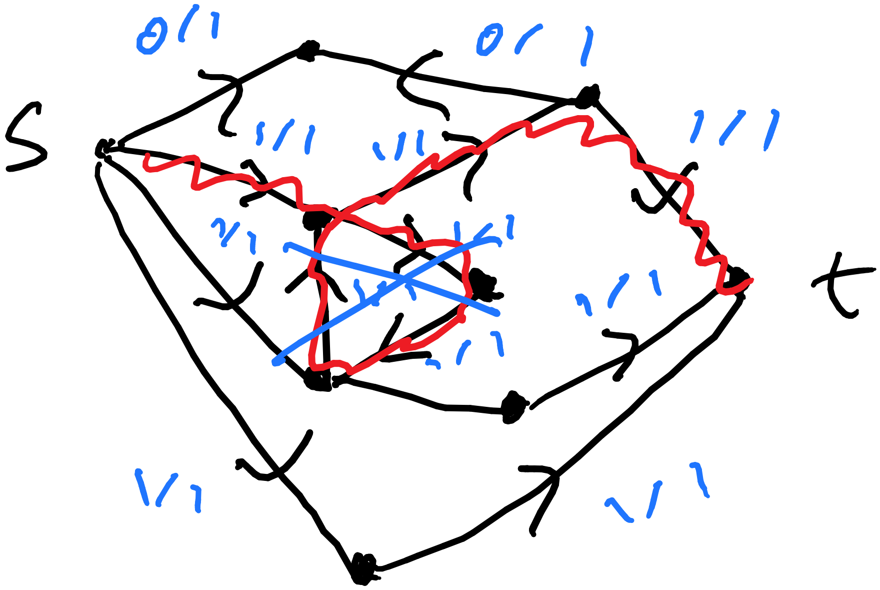
We start from s and trace this one unit of flow. Every time we enter an internal node (not source or sink) we can leave it as $f^{in} = f^{out}$ for such nodes.



We continue in this manner using only new edges, and eventually we end up at t. The above is not a path though, it is a walk as it visits the same vertex multiple times. By removing the loops we obtain a path from s to t.



What about $k > 1$? We can start from s and trace a path to t as above.



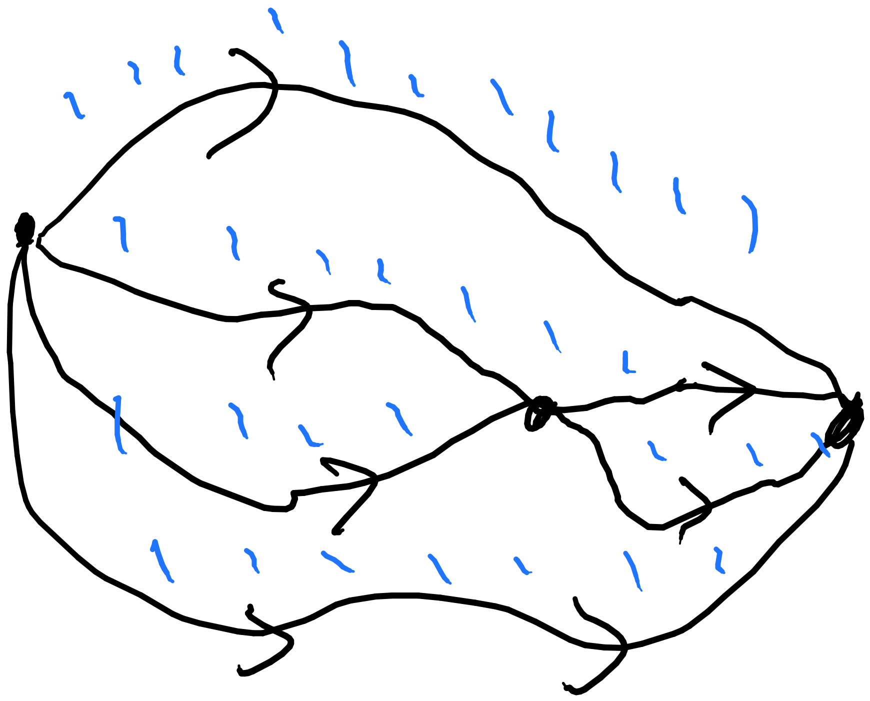
We remove this path and end up with a flow of $k-1$. We continue this process. Every time we find a path and remove it. (For k steps. You could also just say we apply induction here)

These paths are going to be edge-disjoint.

We proved that

$$\underbrace{Max\text{-}disjoint}_{r}\ paths \geq \underbrace{Max\text{-}flow}_{k}$$

To prove equality, note that if we have r edge-disjoint paths then



Here we have r paths, we can just assign a flow of 1 to every edge in these paths to get a max-flow of value r.

$$\implies \textit{Max-flow} \geq \textit{Max-disjoint paths}$$

With the two inequalities, we have equality:

$$\textit{Max-flow} = \textit{Max-disjoint paths}$$