COMP 360: Algorithm Design

Julian Lore

Last updated: April 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.

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

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?

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

$\lceil \log_2 n \rceil$ 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:

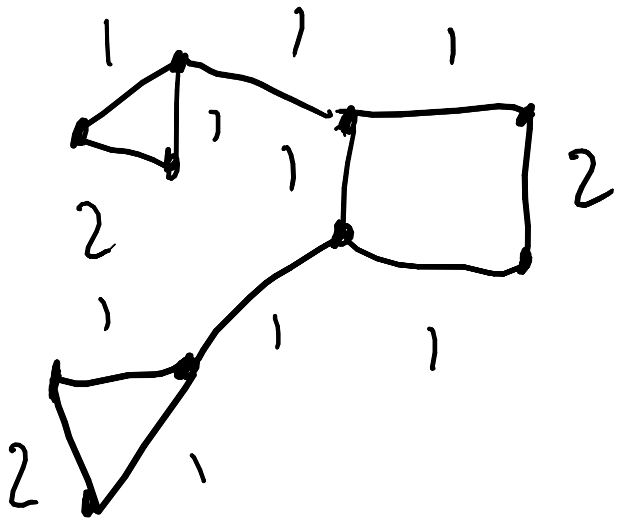
$S = \{1, \underline{3}, 2, \underline{5}, 4\}$
$A = ? = \sum_{i \in \{2,4\}} a_i = a_2 + a_4 = 3 + 5 = 8$

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)

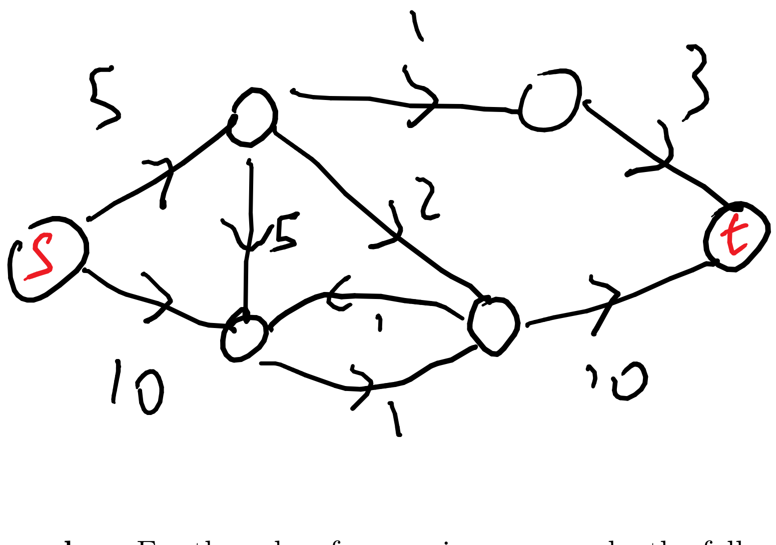
1.4 Network Flows

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.

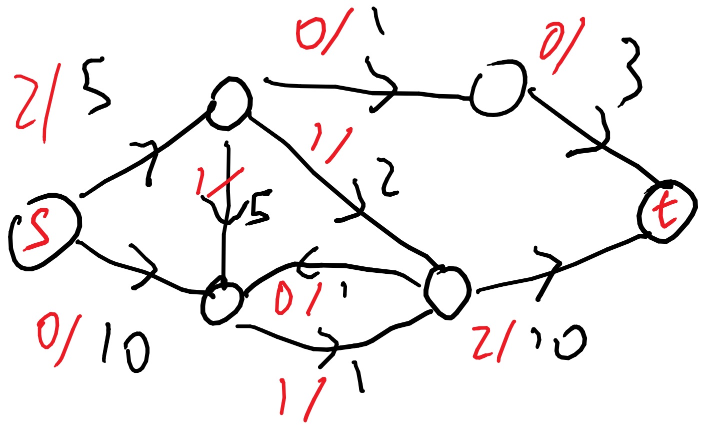
Def: [flow] A flow is a function $f : E \to \mathbb{R}^+$ such that: (Note that :$\mathbb{R}^+ = \{X \in \mathbb{R} | x \geq 0\}$)

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

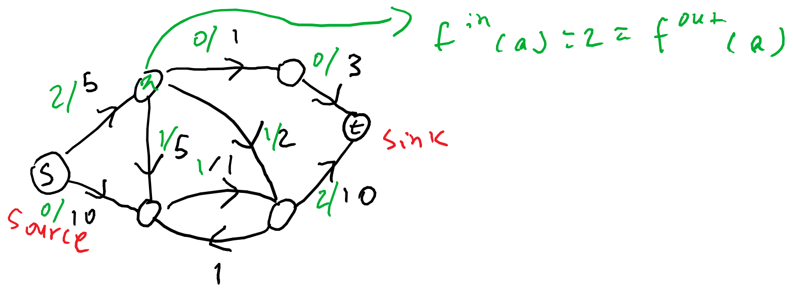
Max Flow Problem: Given a flow network find a flow with largest possible value.

2 01/10/18

2.1 Max Flow Problem (Continued)

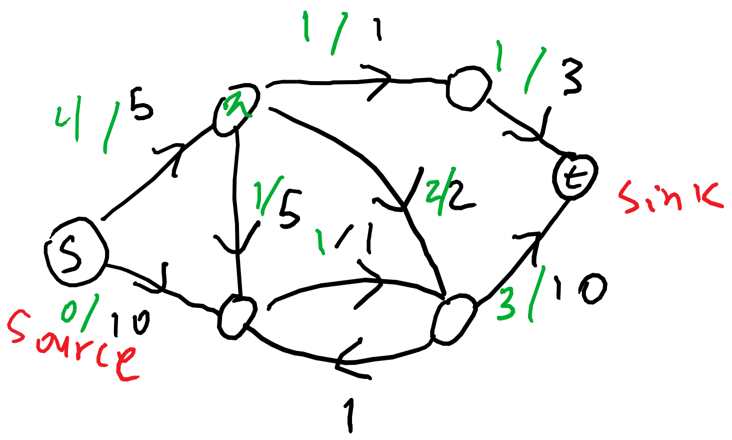
Recall: we want to process a flow network, essentially a directed graph with a source and a sink.

A flow network

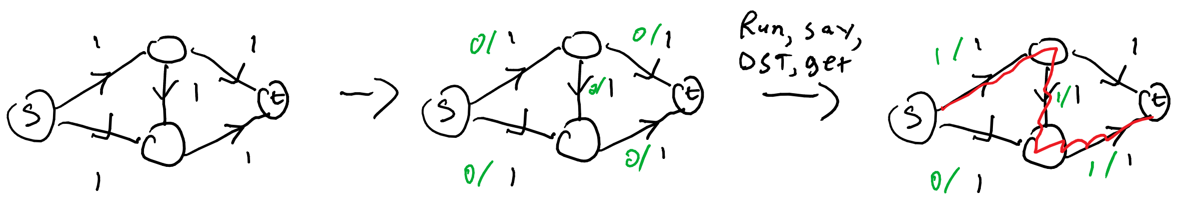


$val(f) = f^{out}(s)$

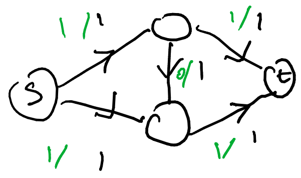
Max Flow Problem: Given a flow network find the maximum value of the flow. 2 is not the optimal value of the example. We could change it to:



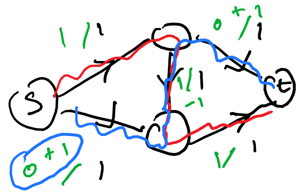
Ford-Fulkerson Algorithm Try to find $s-t$ paths that have not used their capacity and push more flow through them. There is a subtlety here though, we may run into trouble, like in the following example:



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

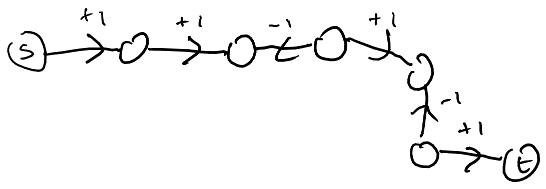


So we must change or else this algorithm won't work. We don't want to go back and change the first step, even though we are stuck. There is a way that we can change things. Say we try to add one more unit of flow:



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.

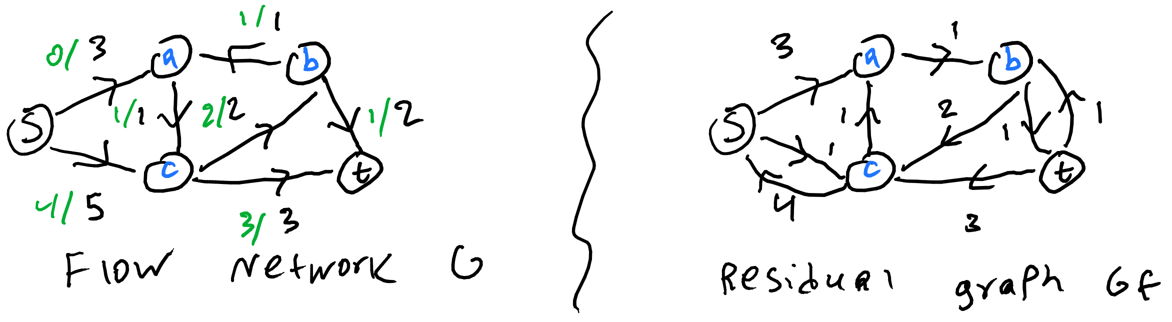


How do we implement this?

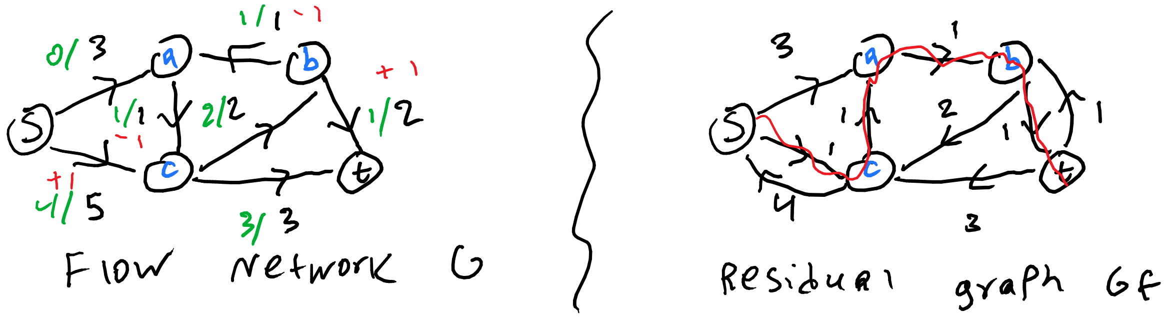
Def [Residual graph] Given a flow network $(G, s, t, \{c_e\})$ and an flow f on G, the residual graph G_f is as follows (we are already in the middle of the algorithm and this graph will tell us which edges are usable):

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:

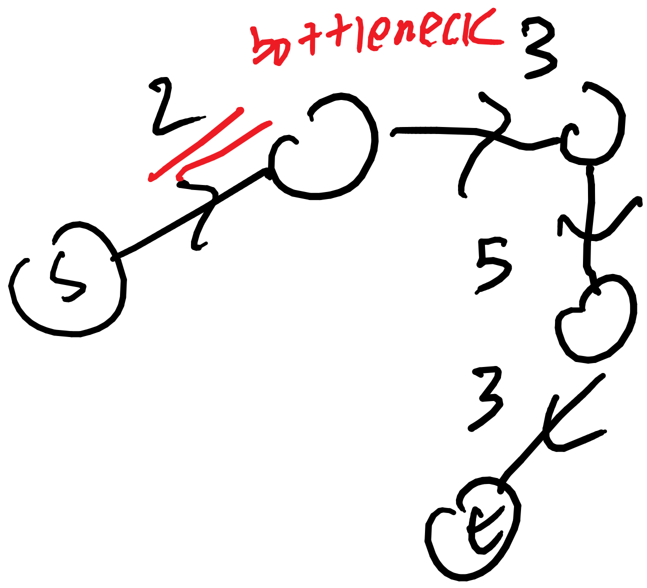


Pseudocode for Ford-Fulkerson

```
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.

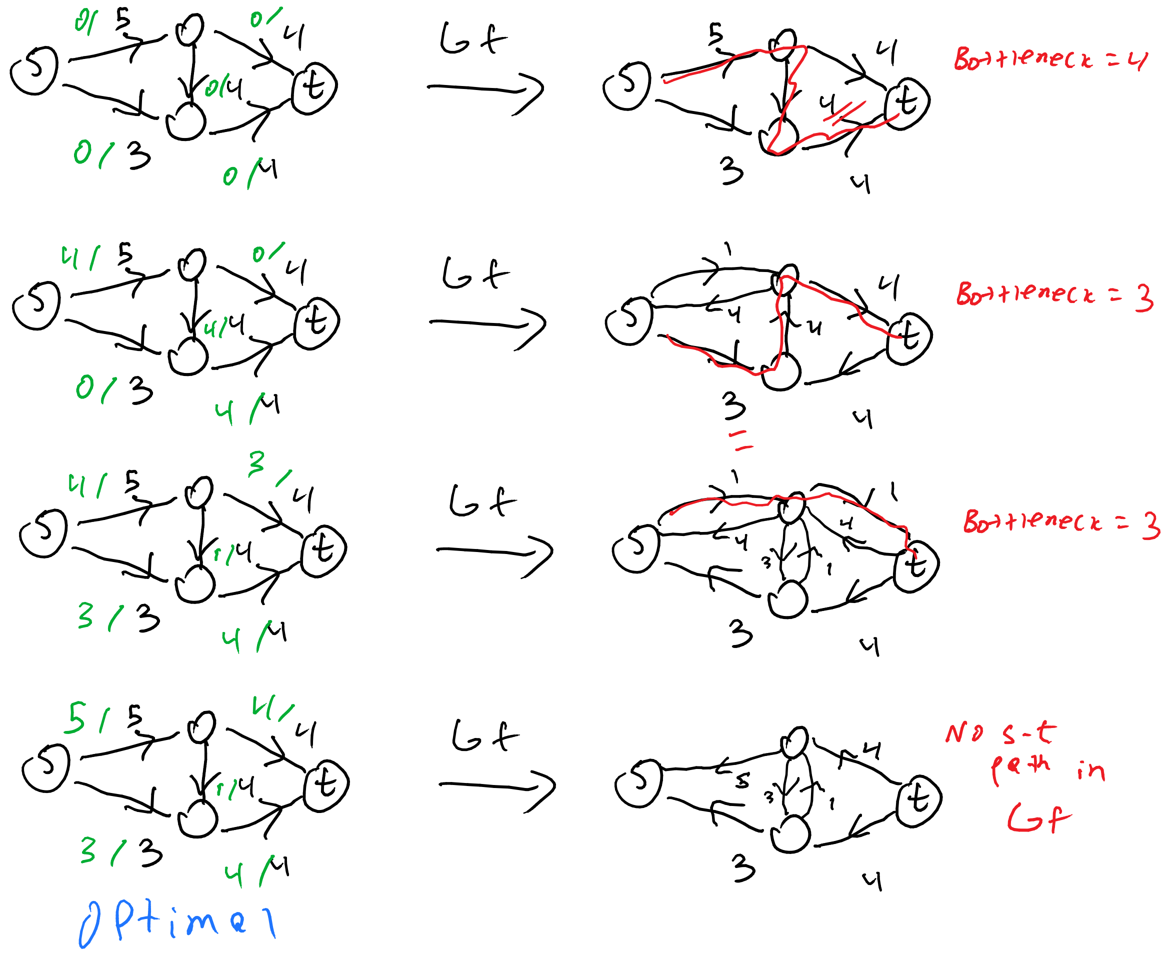
Augment(f,P)

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.

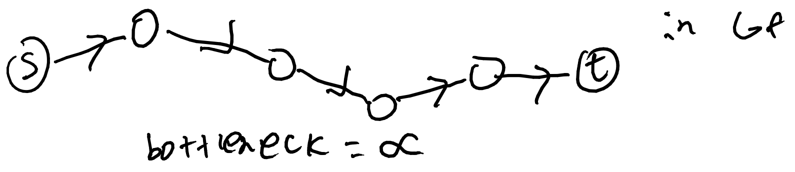
Example



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

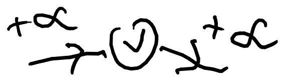
Proof Residual capacities are chosen so that updating with Augment(f, P) will never assign a number to an edge that is larger than its capacity or smaller than 0. $\implies$ capacity condition is satisfied throughout the algorithm.

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



In G:

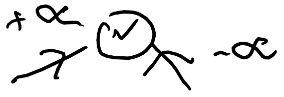
- Case 1:



$$
\begin{aligned}
f^{in} &\leftarrow f^{in} + \alpha \\
f^{out} &\leftarrow f^{out} + \alpha
\end{aligned}
$$

Still the same.

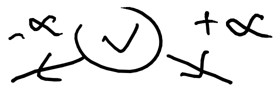
- Case 2:



$$
\begin{aligned}
f^{in} &\leftarrow f^{in} + \alpha - \alpha \\
f^{out} &\leftarrow f^{out}
\end{aligned}
$$

Nothing changed.

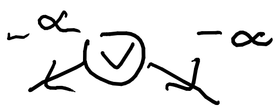
- Case 3:



$$
\begin{aligned}
f^{in} &\leftarrow f^{in} \\
f^{out} &\leftarrow f^{out} - \alpha + \alpha
\end{aligned}
$$

Still equal.

- Case 4:



$$\begin{aligned} f^{in} &\leftarrow f^{in} - \alpha \\ f^{out} &\leftarrow f^{out} - \alpha \end{aligned}$$

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:

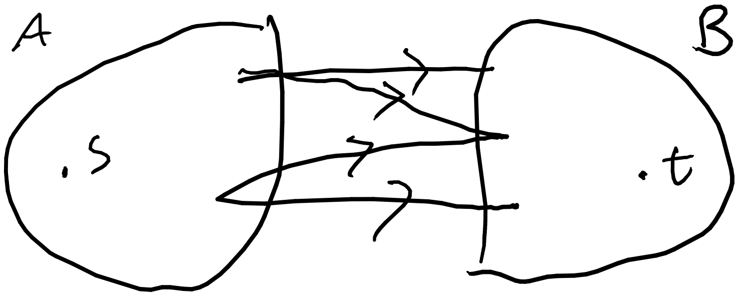
$$O(K \times m \times m) = O(Km^2)$$

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

2.2 Cuts

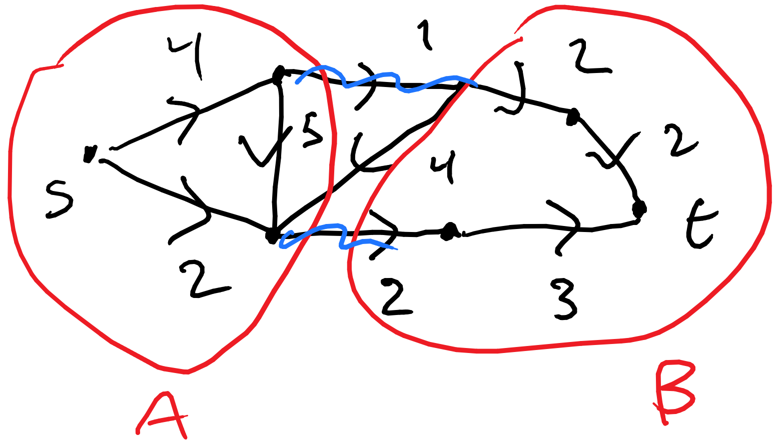
Def A cut ($s-t$-cut) in a flow network is a partition (A, B) of the vertices such that $s \in A, t \in B$.

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

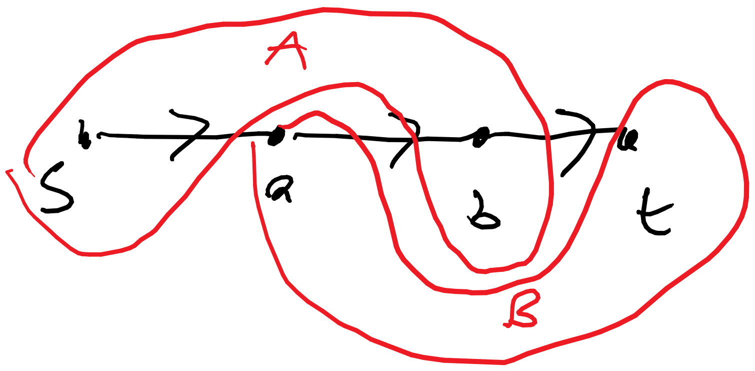


$$cap(A,B) = \sum_{\substack{uv \in E \\ u \in A \\ v \in B}} c_{uv}$$

Example



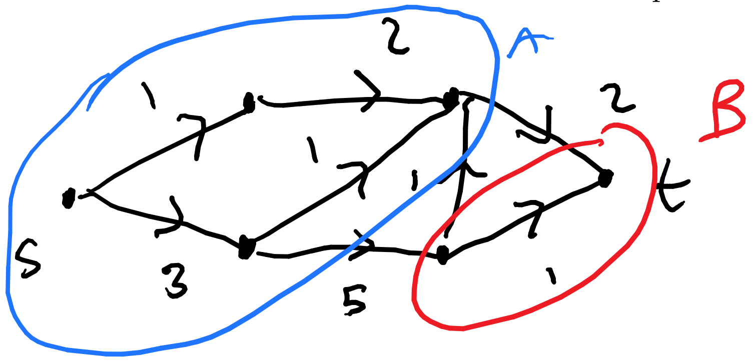
The capacity here is 3. We see that we can't pass more weight from A to B, i.e. cuts intuitively tell us something about the max flow.
How many cuts are in this network?



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 $s \in A, t \in B$.



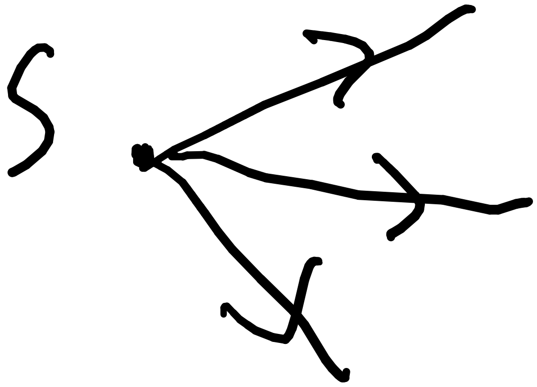
In this example, the capacity is $5+2$

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

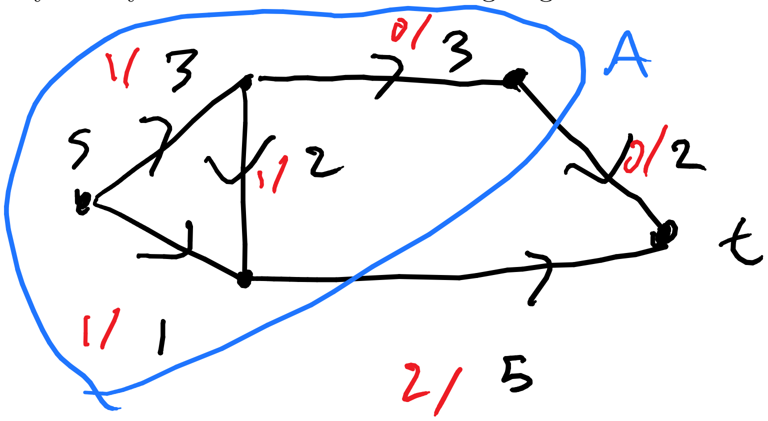
These capacities give us an upper bound on the maximum flow, but we have to prove this, intuition isn't enough. So how do we prove this?

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?

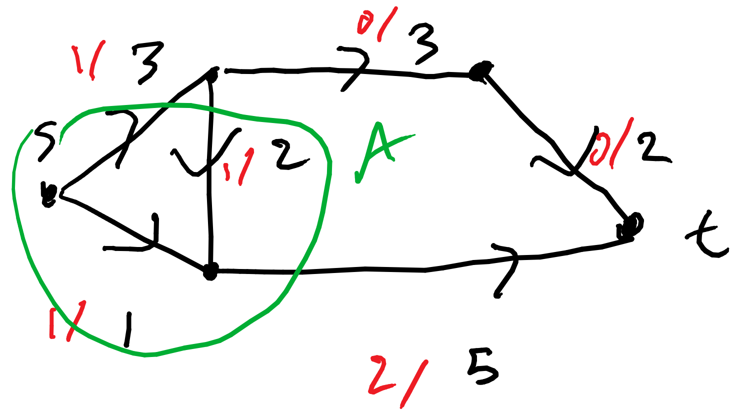


In this example we see that $val = 1 + 1 = 2$. We can also define a flow going into t. In this case it would also be 2. Are they equal? Our intuition says yes, because none of the intermediate nodes are adding or absorbing flow.

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

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

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



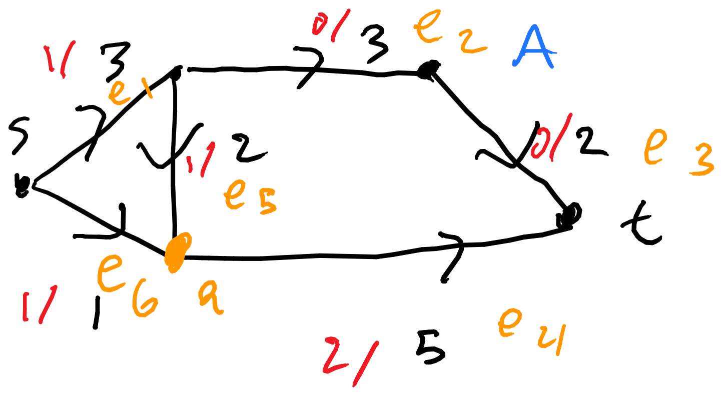
$f^{out}(A) = 1 + 2, f^{in}(A) = 1$

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

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

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

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

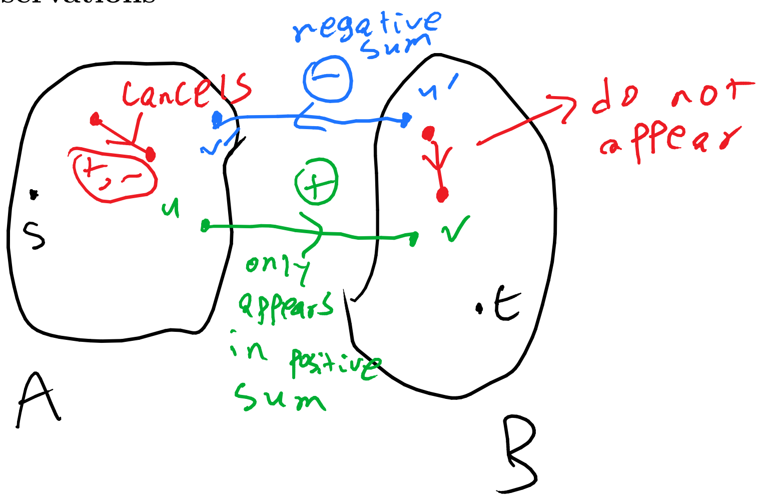


$$\begin{aligned}
f^{out}(s) &= f(e_1) + f(e_6) \\
f^{in}(s) &= 0 \\
f^{out}(a) &= f(e_4) \\
f^{in}(a) &= f(e_6) + f(e_5)
\end{aligned}$$

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

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

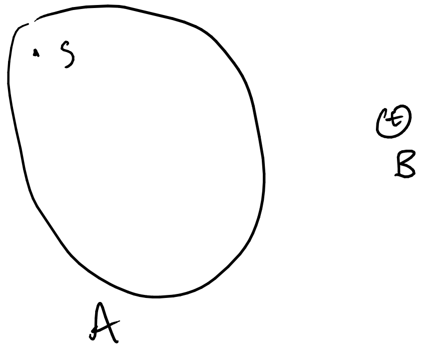


Because of some cancellations here, we can simplify the sum:

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

This concludes the proof of the claim. □

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



Then by the claim:

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

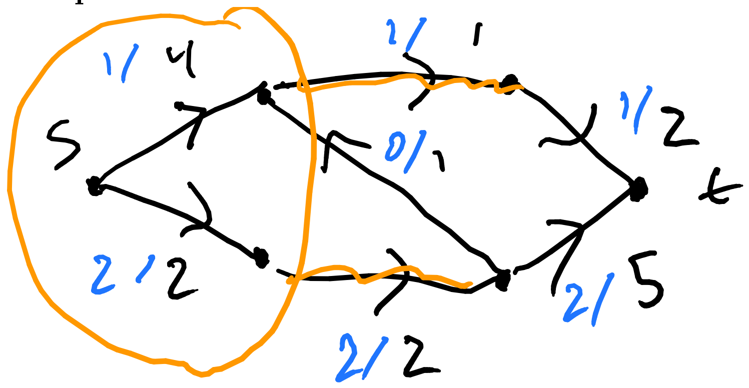
Corollary to this claim Let (A, B) be a cut, f be a flow. Then $val(f) \leq cap(A, B)$. i.e. the flow cannot exceed the capacity of the cut, any arbitrary cut puts an upper bound on the flow. How can we prove this using the previous claim?

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

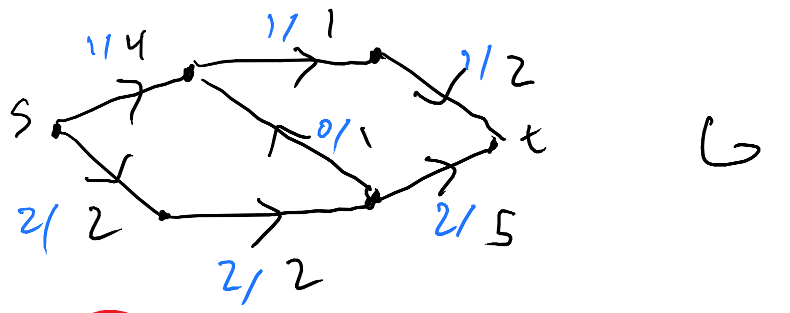


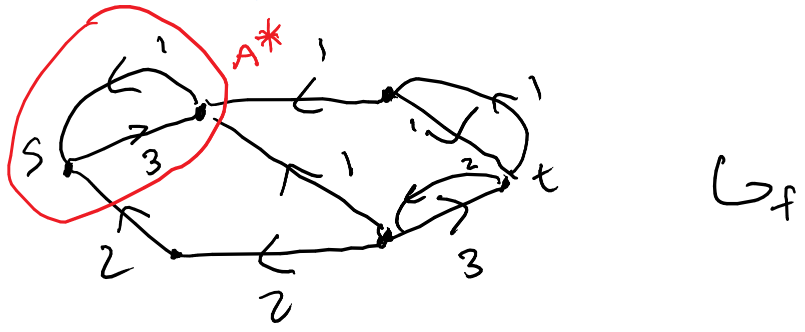
$cap(A, B) = 1 + 2 = 3$ and $val = 3 \implies$ max flow $= 3$. So if we get a flow and are asked if this is the max flow or not, either we find a flow with a better value to disprove it, or find a cut such that the capacity is the same as the flow, to prove that we can't do any better than that.

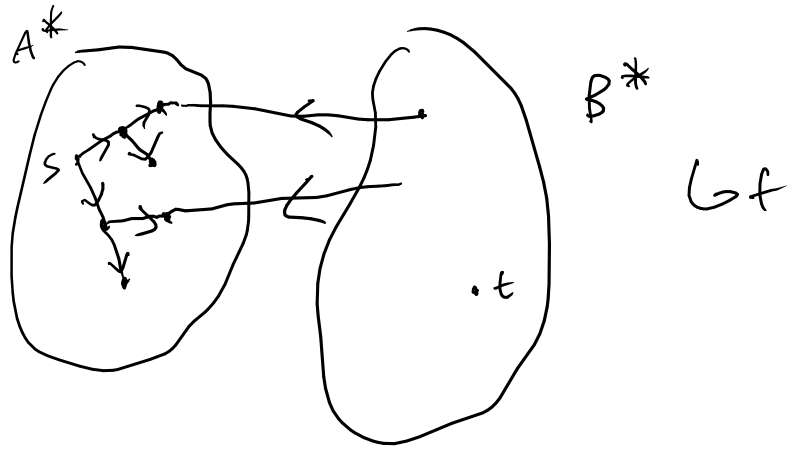
Proof of the fact that Ford-Fulkerson finds the max flow Recall:

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

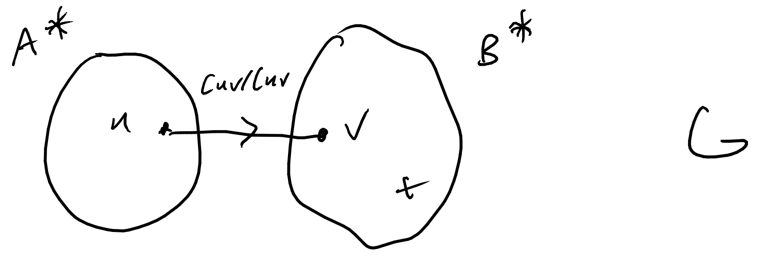
Consider the point where Ford-Fulkerson terminates. Let A^* be the set of the vertices that can be reached from S in the residual graph. Why is this a valid cut? Because at termination, there are no more $s-t$ paths, so $t \notin A^*$.



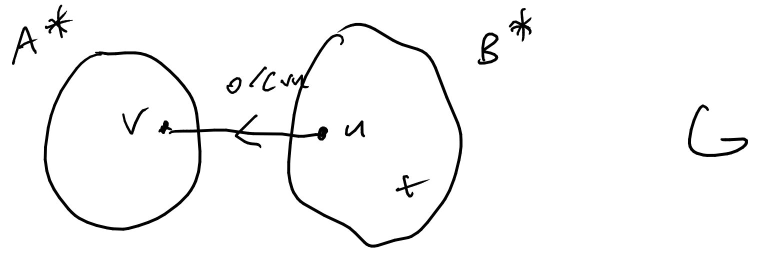




There are no edges in G_f from A^* to B^*, or else the endpoint vertex in B^* would be in A^*, because A^* consists of all the vertices we can reach from s. Thus: if uv is an edge in the original network with $u \in A^*, v \in B^*$



$f(uv) = C_{uv}$, or else the edge uv 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:

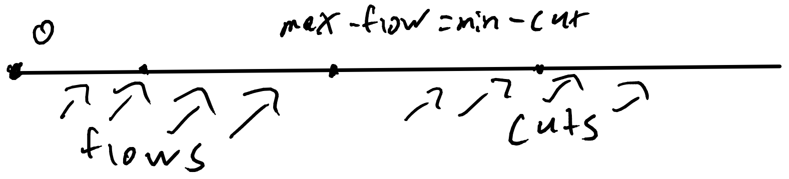
$$max\text{-}flow \leq cap(A^*, B^*) = val(f) \implies val(f) = max\text{-}flow$$

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 $val(f) = cap(A^*, B^*)$

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



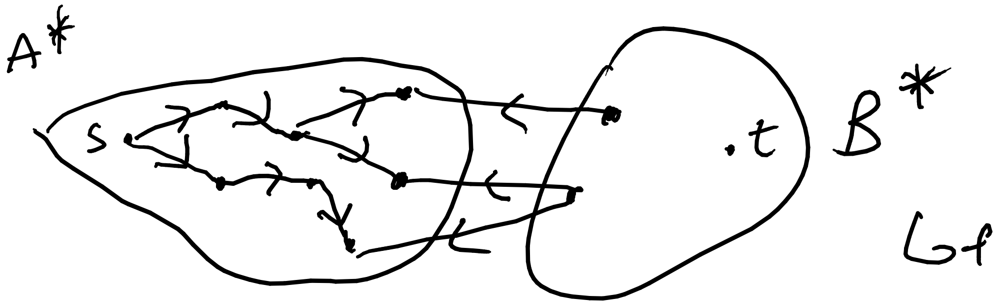
Thm For any flow network:

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

4 01/17/18

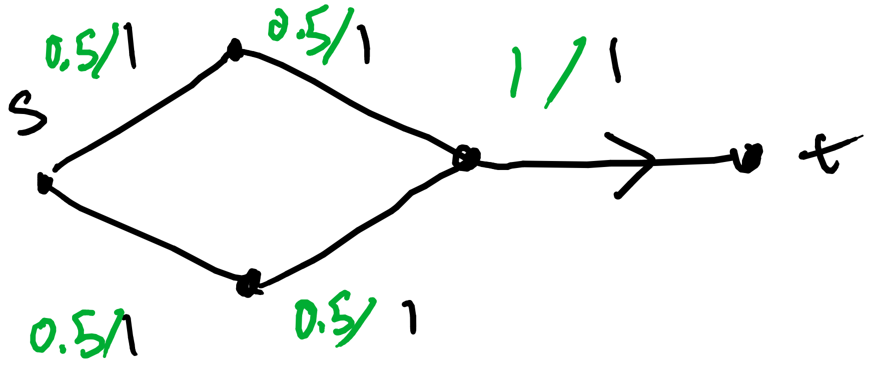
Recall

- Ford-Fulkerson finds the max flow.
- *max-flow* = *min-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*.



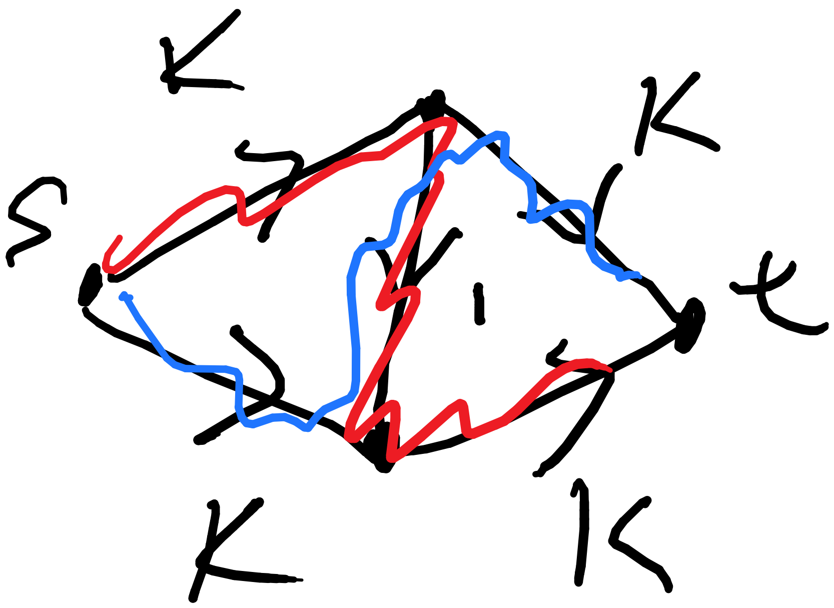
Can't reach t from s at the end of the algorithm in the residual graph.

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.

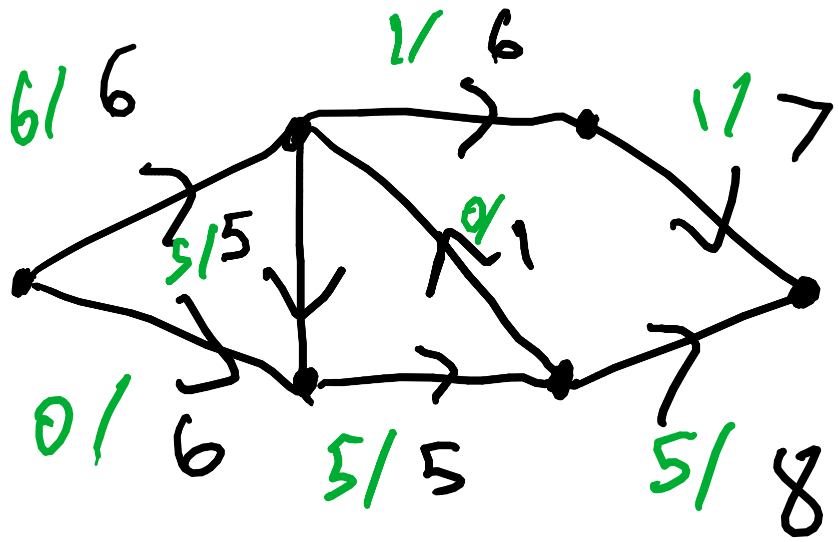
2. Try to go with the paths that increase the flow by larger numbers. In the above example, we see that the red path only increases flow by 1, instead of the top path that can increase it by K. This is called the Fattest Path approach, where we find an augmenting path with the largest bottleneck. However, 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 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
```

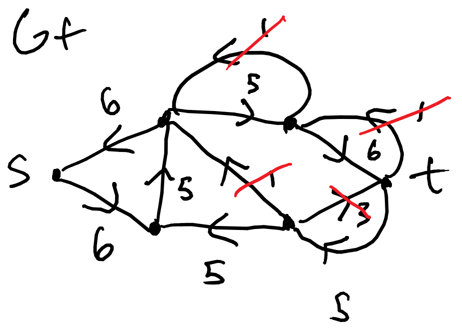
How can we check the underlined condition, that there are augmenting paths with bottleneck greater than Δ?

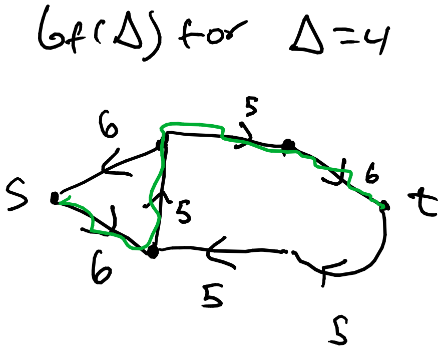


with $\Delta =$

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

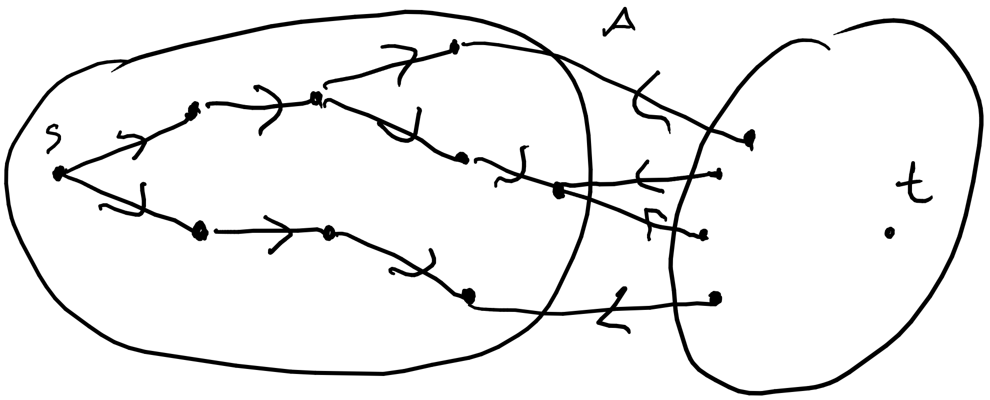
- Checking if there exists an $s-t$ path: $O(m)$
- Augmenting, $O(m)$
- Updating G_f, $O(m)$

So we need to understand the number of iterations. The outer loop has $\lceil \log_2 K \rceil$ iterations. The inner loop? (actually will be a bit of work to analyze this.) How many times in the $\Delta - phase$?

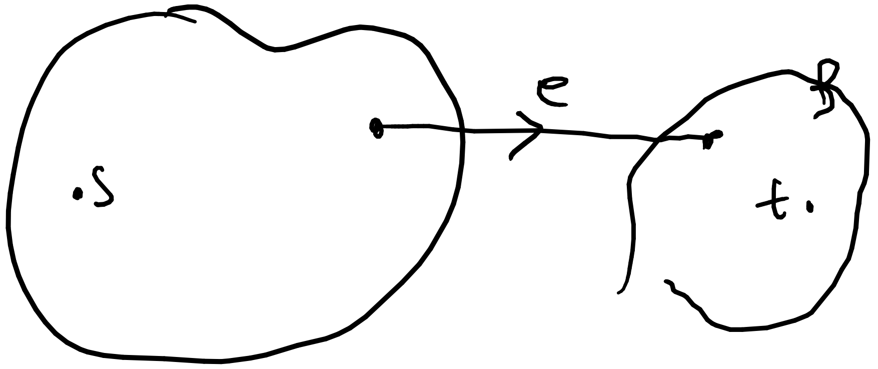
Claim Let f be the flow at the end of the Δ-phase (when no $s-t$ paths are in $G_f(\Delta)$). There is a cut (A, B) such that

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

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)

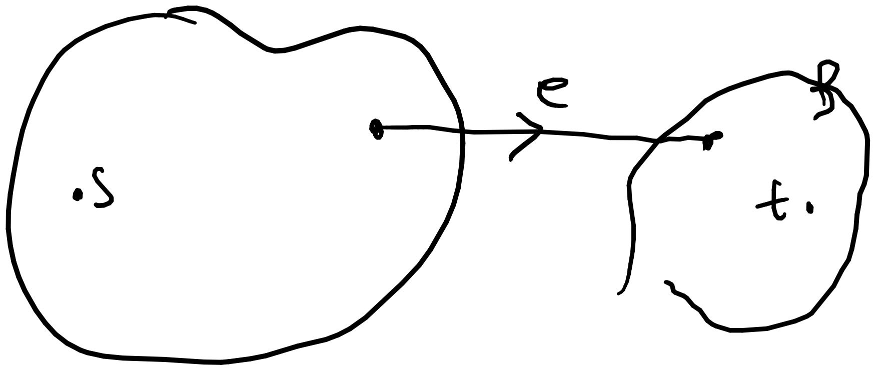


(No edge from A to B in $G_f(\Delta)$, otherwise A would have been extended further)
If e is an edge from A to B in the original network:



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

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:

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

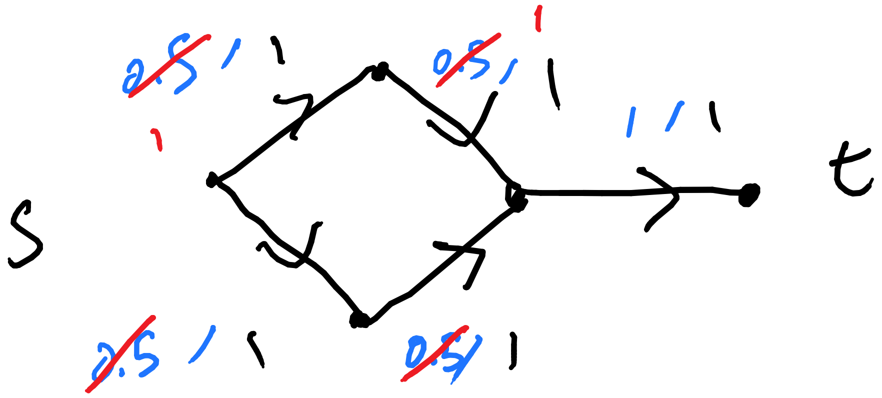
Instead of $O(m^2 K)$ of the naive Ford-Fulkerson. This is a big improvement when K is a huge number.
One thing is left: Why does this algorithm find the *max-flow*? Because when it terminates, $\Delta = 1$ and it means there are no more augmenting $s-t$ paths in the residual graph.

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

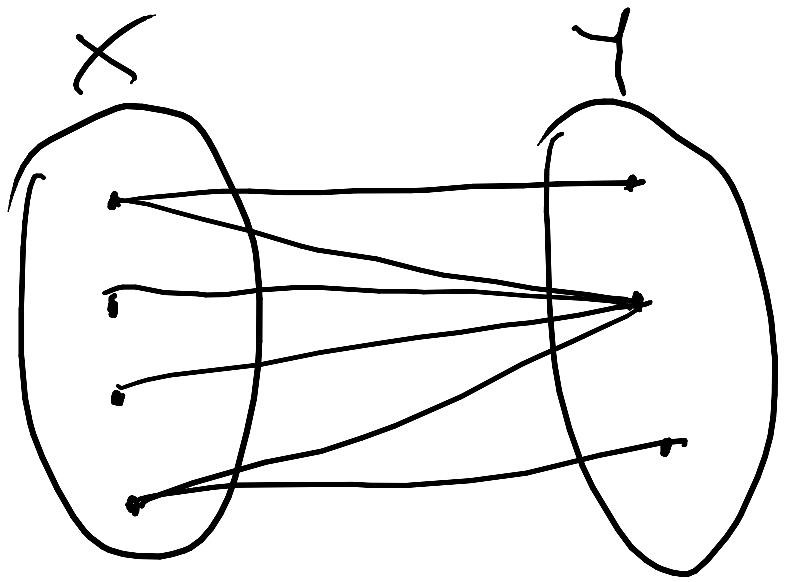
5 01/22/18

Recall

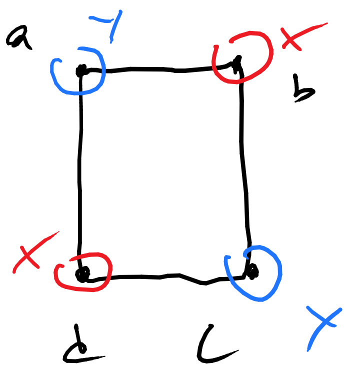
- Ford Fulkerson finds max-flow $O(m^2K)$.
- **Scaling Ford Fulkerson** finds max-flow $O(m^2 \log K)$, K =largest capacity.
- *Max-flow* = *Min-cut*
- There is a *max-flow* that assigns integer flows to all edges.

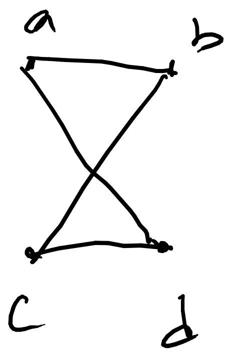


Bipartite Graph is an undirected graph such that the vertices can be partitioned into two parts X and Y such that all the edges are between X and Y.

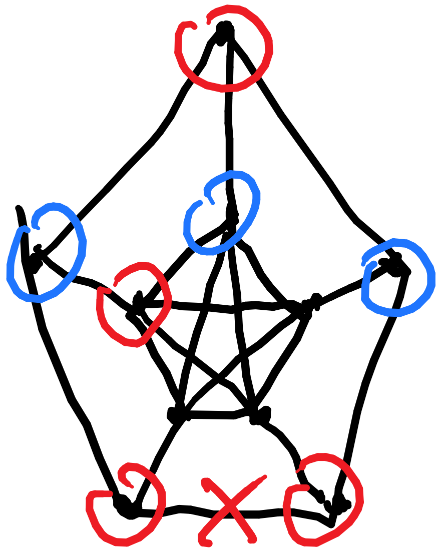


Examples Is this bipartite? Yes. We can check by just 2 coloring.



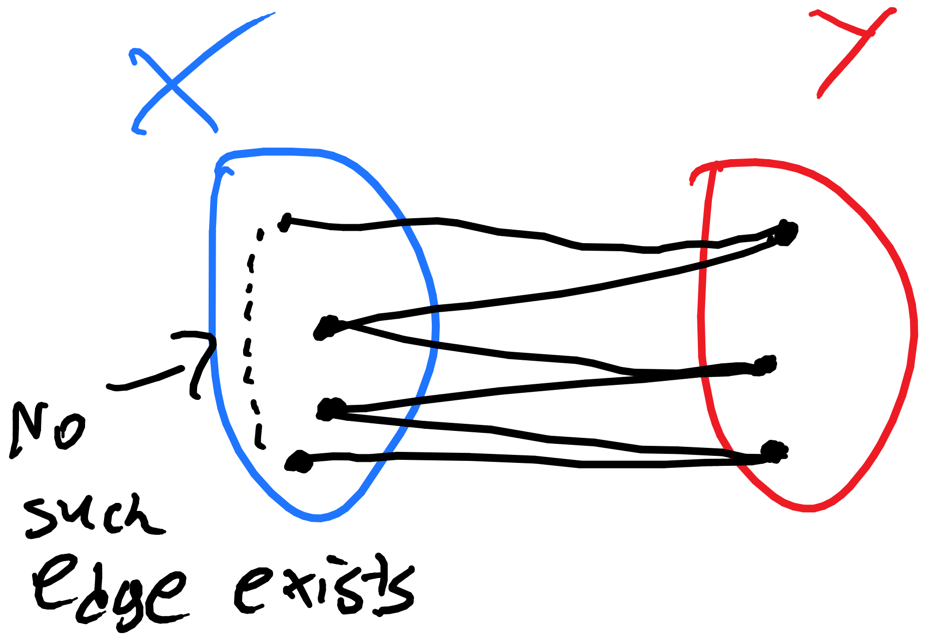


What about the Peterson graph? No.



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:

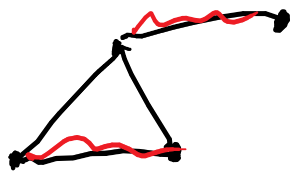
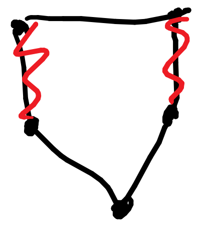


5.1 Largest Matching Problem

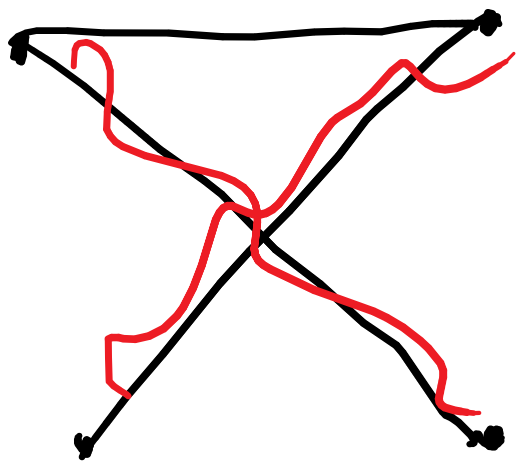
Def A matching is a set of edges, no two of them share an endpoint.

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

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

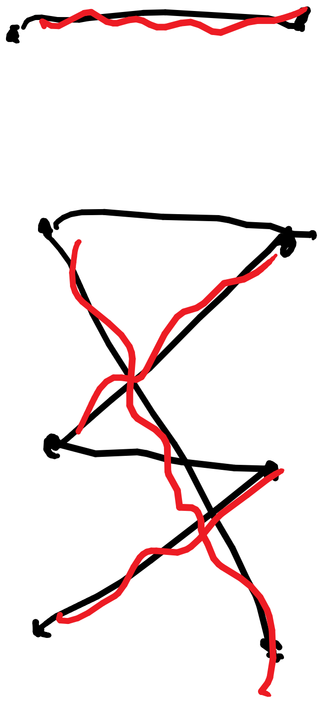



Note that there is nothing geometric about a matching, it does not matter how you draw the graph, the fact that the edges "cross" over each other does not matter, like here:



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

Ex Maximum matching here is 4:



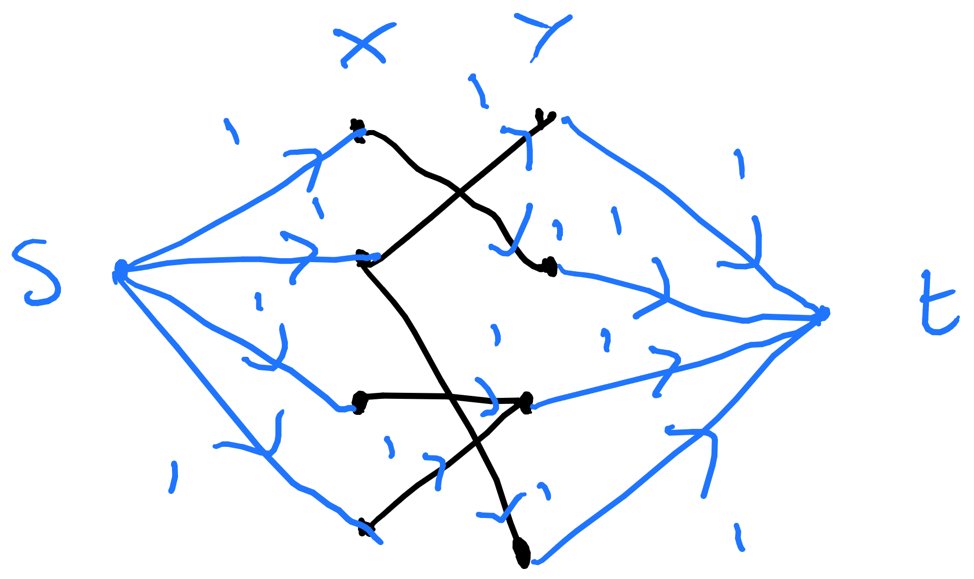
Greedy algorithms won't work here, so how should we solve this?

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?

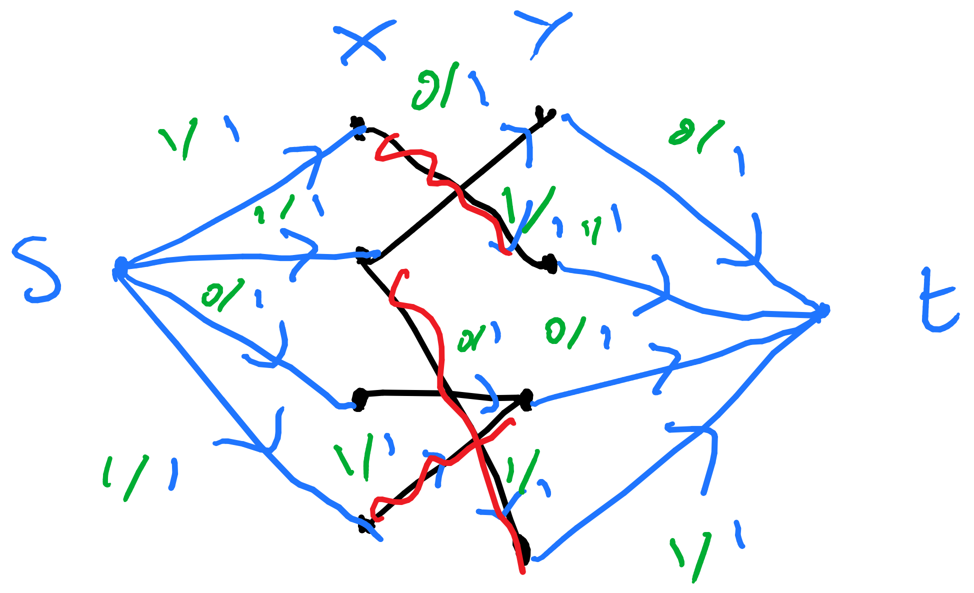
Solution We construct a flow network in the following manner:

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.

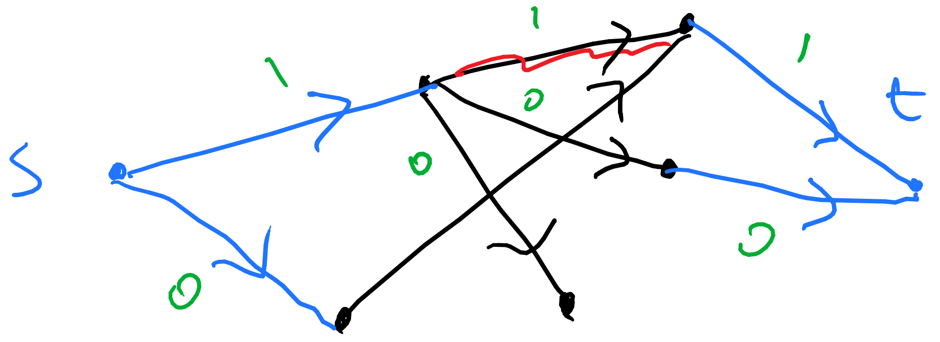


Claim Max flow in this network = max matching in G.

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.



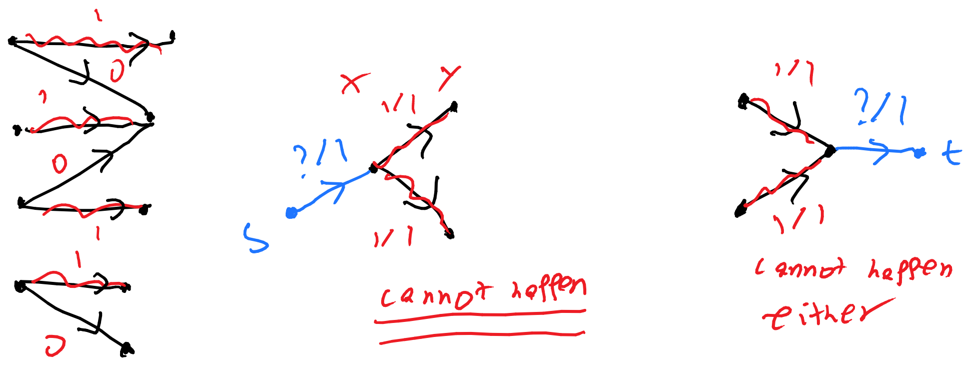
Why is this a valid flow? We aren't overloading capacities (only assigning 0 or 1) and we conserve flow going out of s to flow going into t, since every edge in M has 1 vertex with an edge coming from s and one with an edge going to t. The value of this flow is $|M|$. This is because there are $|M|$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



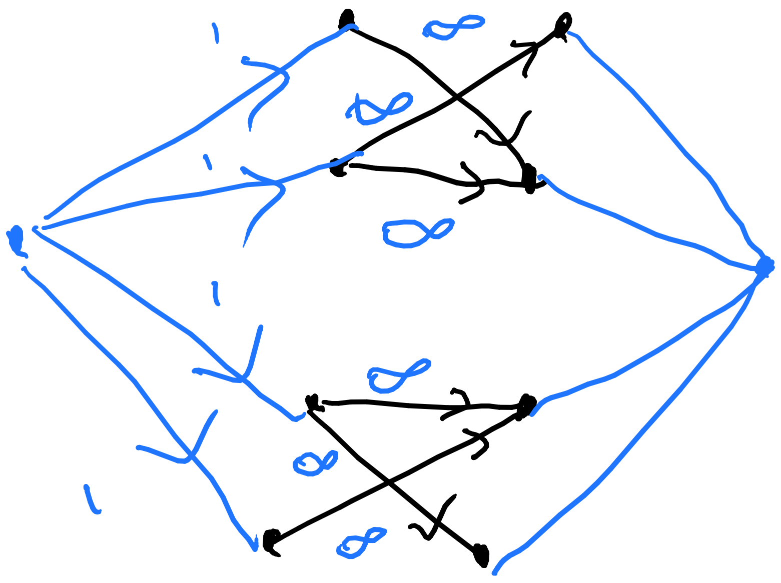
To summarize, we showed

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

and

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

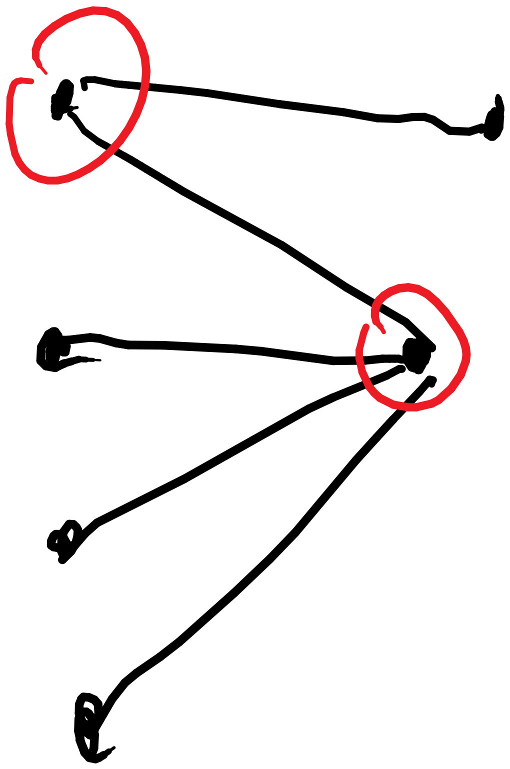
Remark Note that in the above proof we could assign ∞ capacities (instead of 1 to all) to edges between X and Y.



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.

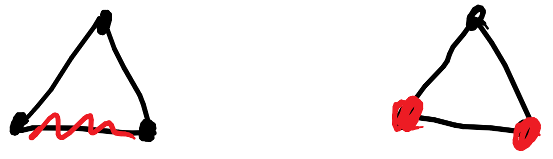


Here, the red vertices form a vertex cover. How can we find the smallest vertex cover? If we want to think of this practically, we can think of the vertices as monitors such that we can monitor all the connections/roads.

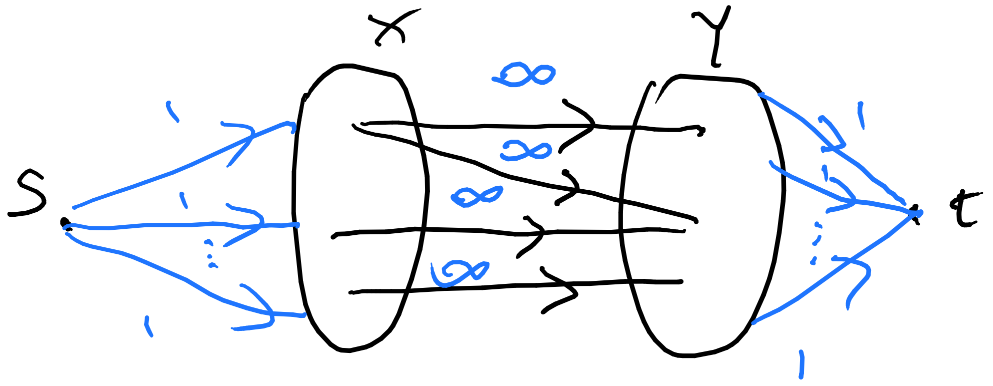
Thm In every bipartite graph max matching = min vertex cover.

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

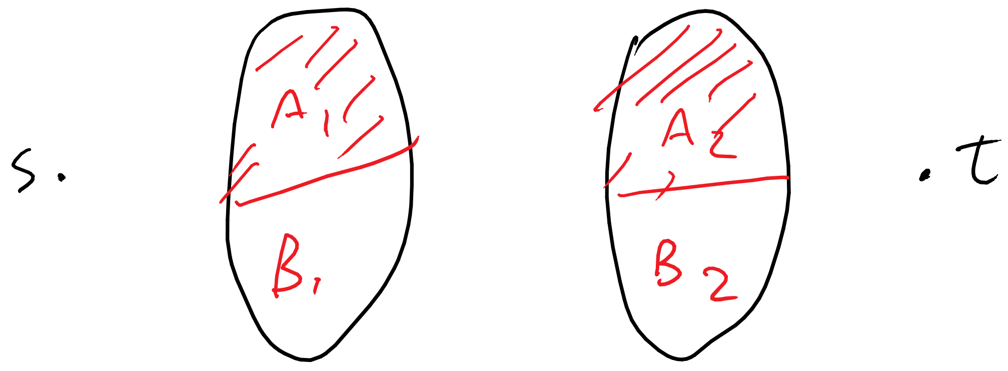
Remark The equality is not true in general graphs.



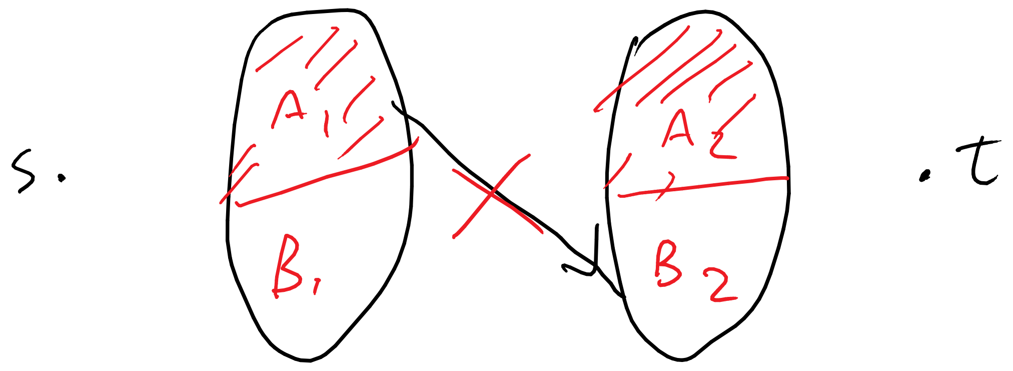
Here the max matching is 1 but the minimum vertex cover is 2.



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.

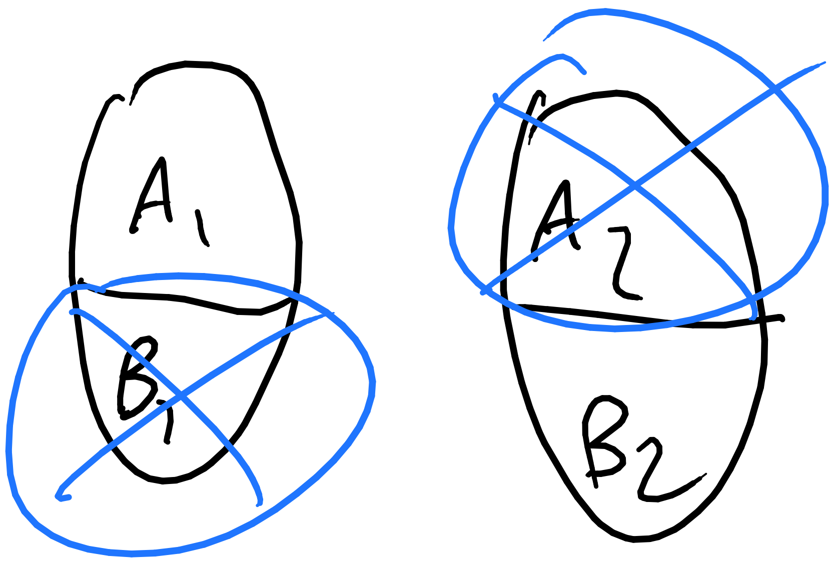


$A = \{s\} \cup A_1 \cup A_2, B = B_1 \cup B_2 \cup \{t\}$



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?

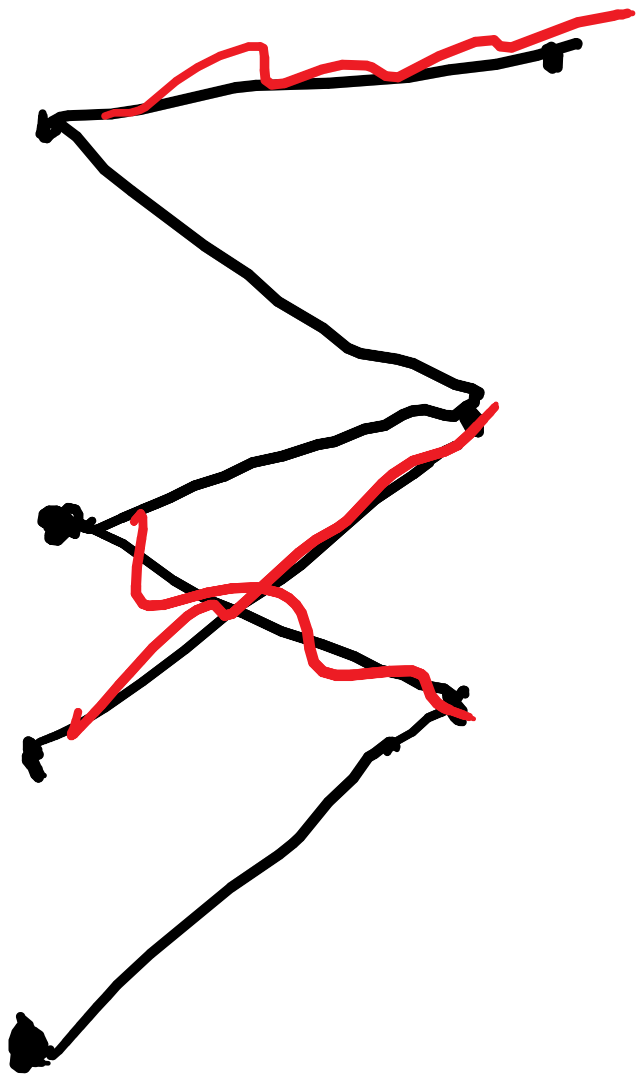
$B_1 \cup A_2$ is a vertex cover in G.



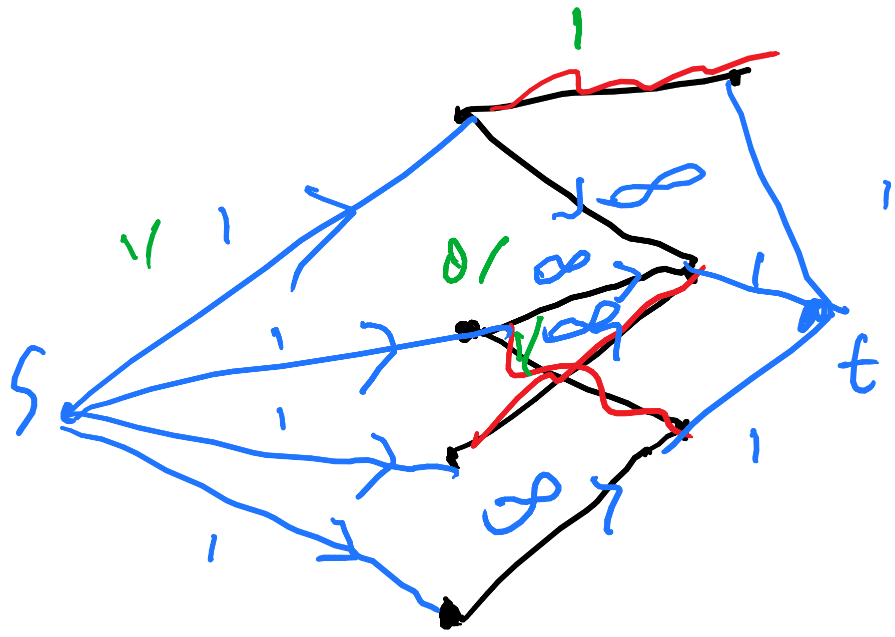
The only edges that could remain are from A_1 to B_2, but we just showed that those couldn't exist (to continue next class).

6 01/24/18

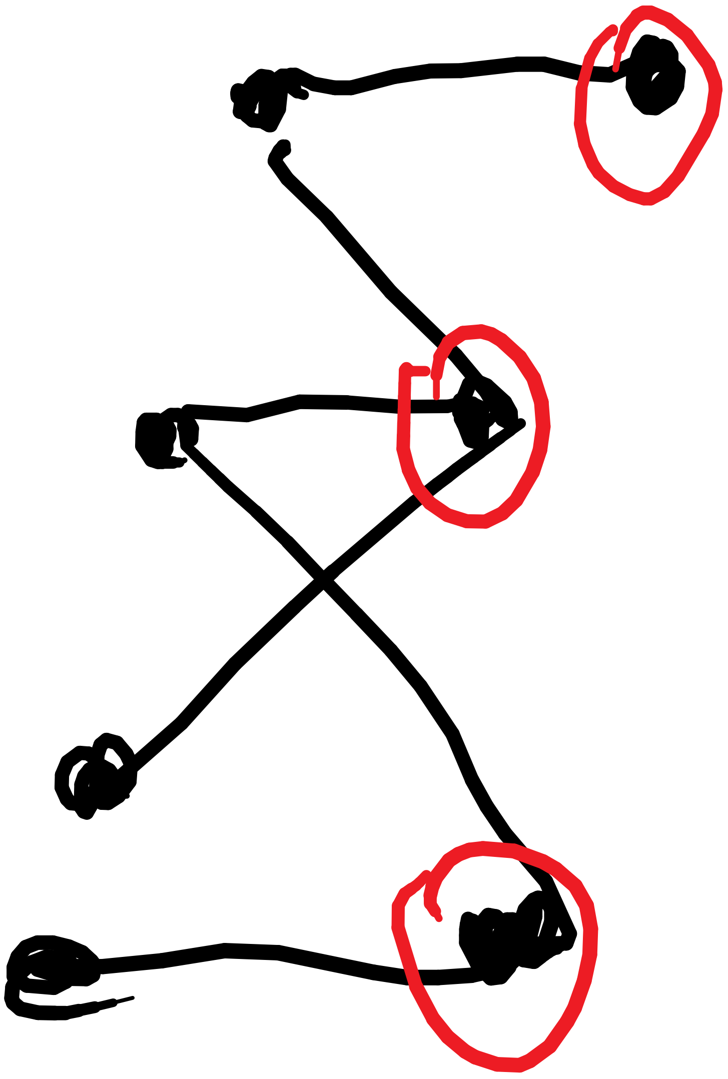
Recall Matching in Bipartite graphs.



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


s
t
1
∞
0/1
1/1

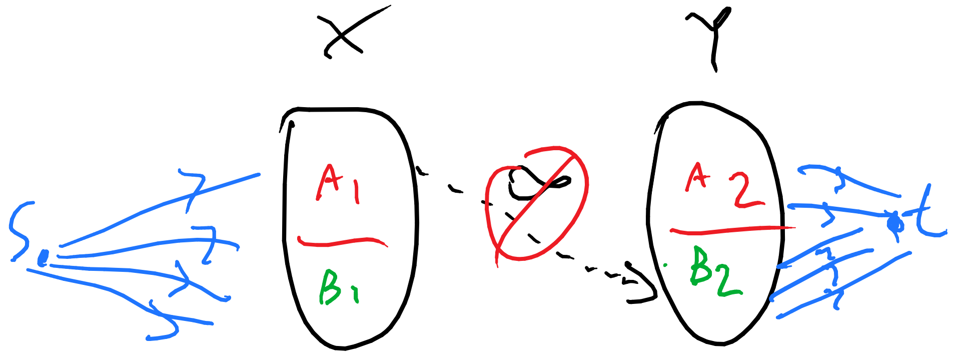
Vertex Cover A set of vertices such that deleting them will remove all the edges.



Thm For every bipartite graph G we have

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

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,

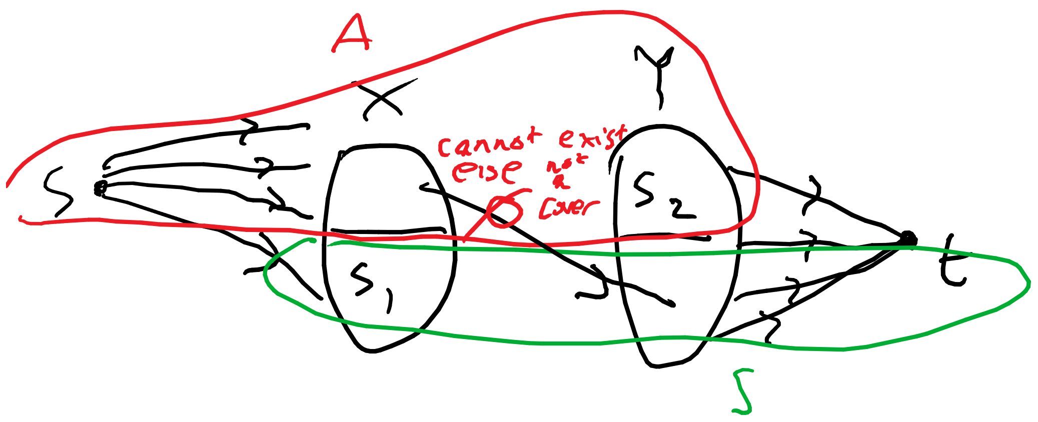
$$Cap(A, B) = \sum_{\substack{\text{edges from} \\ s \text{ to } B_1}} c_{sv} + \sum_{\substack{\text{edges from} \\ A_2 \text{ to } t}} c_{vt} = |B_1| + |A_2|$$

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

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

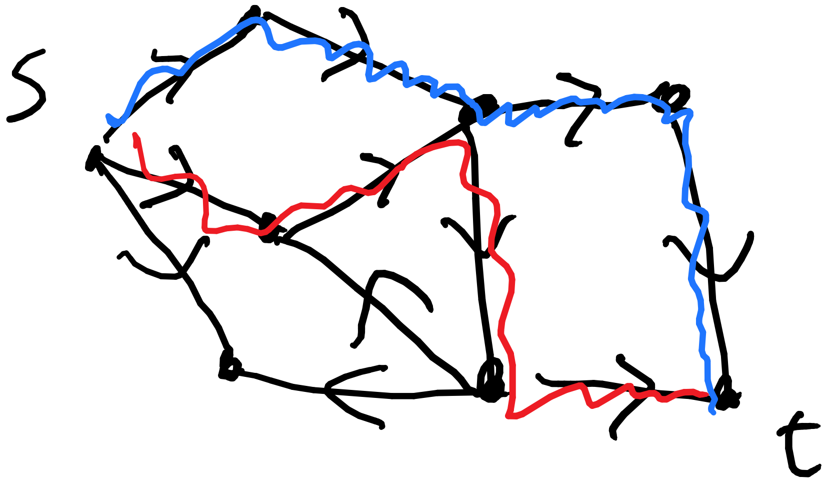
Thm (König) In a bipartite graph $Max\text{-}Matching = Min\text{-}Cut$

6.1 Disjoint Paths in directed graphs

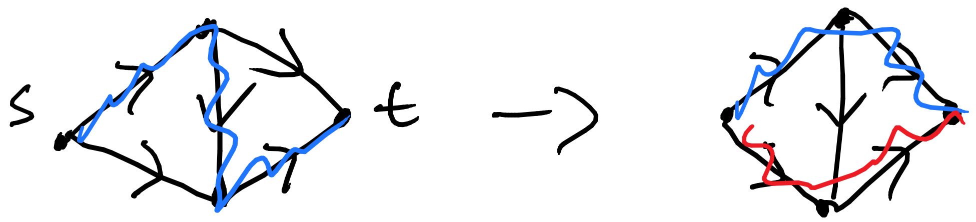
Input: A directed graph and two distinct nodes are marked as $\underline{s}$ and $\underline{t}$.

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

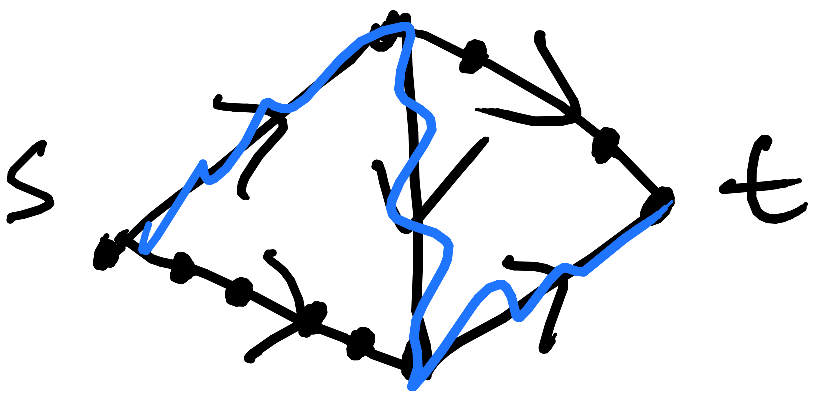
Ex



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

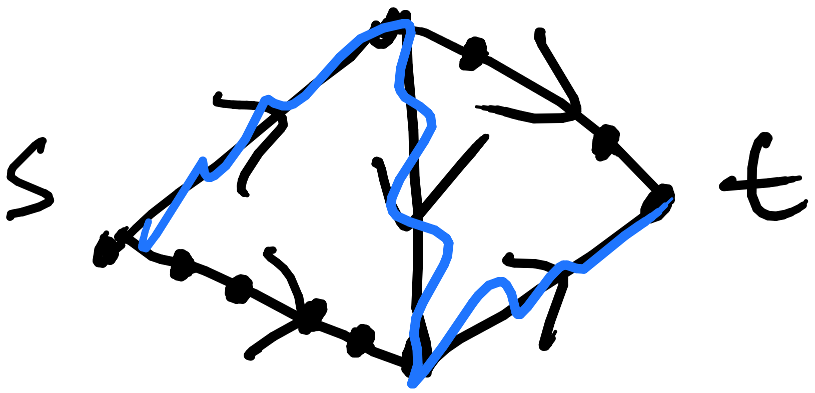


What if we chose the shortest path? That would also be problematic:



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

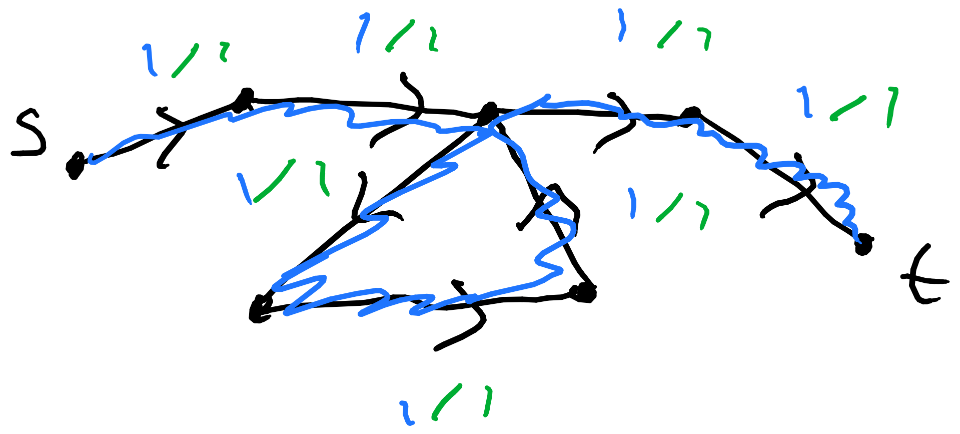


So the max-flow here is 3. How do we show there are 3 edge-disjoint paths?

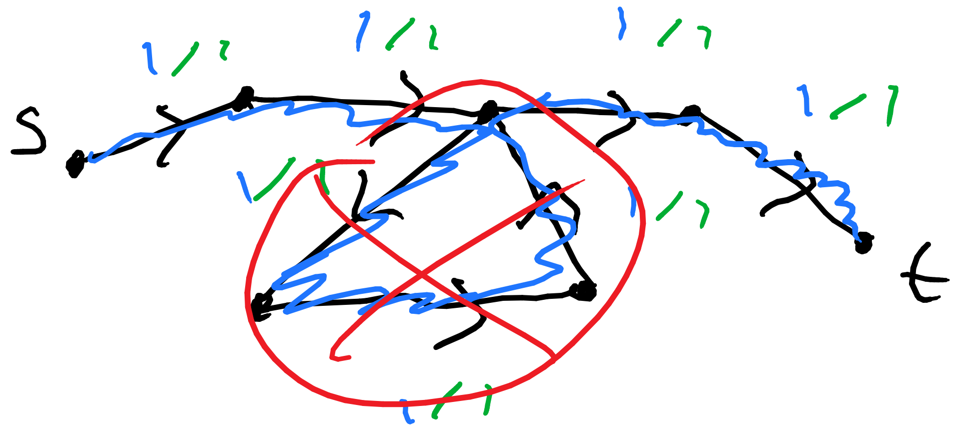
We solved the *max-flow* using Ford Fulkerson and let k be the value of *max-flow*. We want to show that there are k edge-disjoint paths from s to t.

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

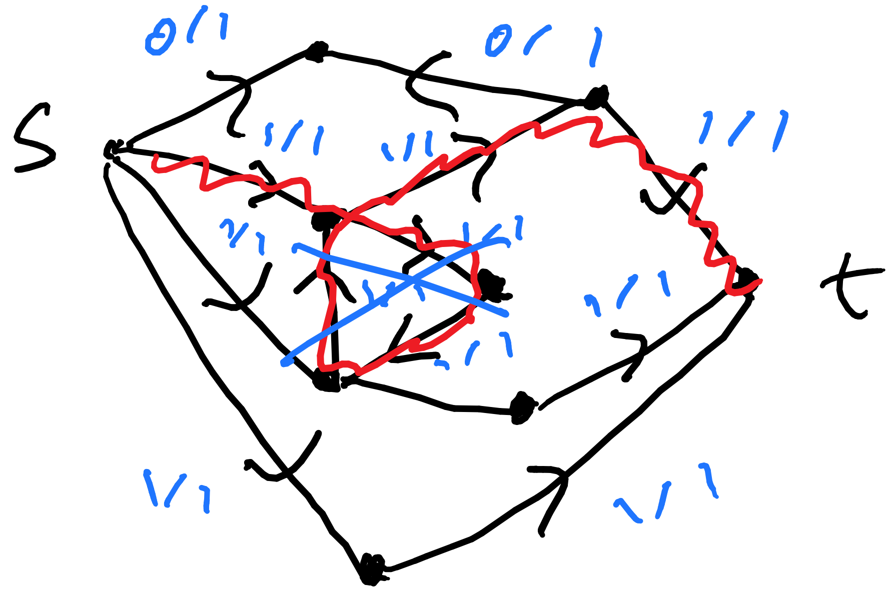
We start from s and trace this one unit of flow. Every time we enter an internal node (not source or sink) we can leave it as $f^{in} = f^{out}$ for such nodes.



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.



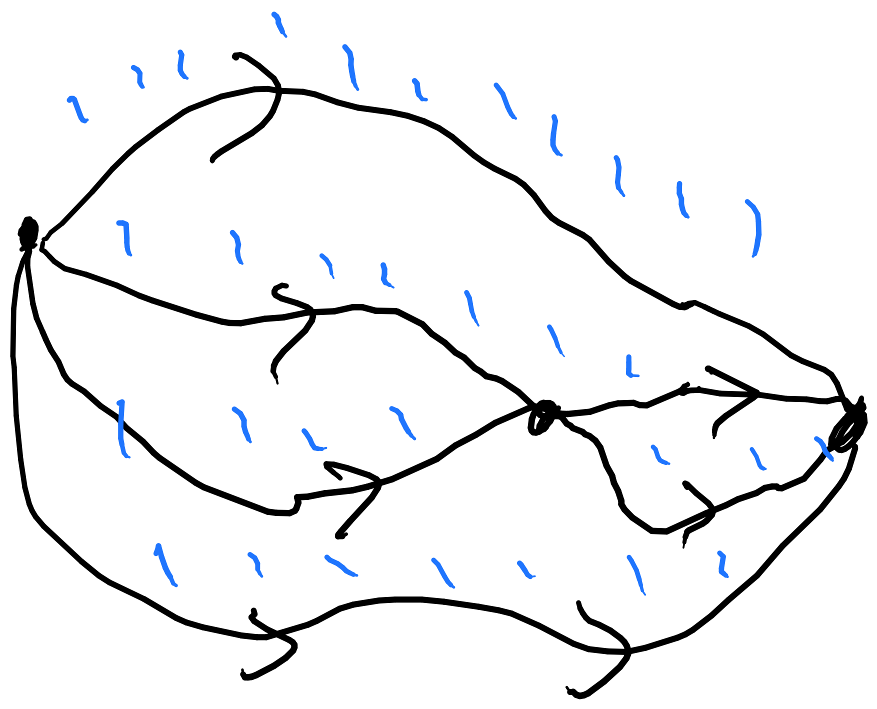
We remove this path and end up with a flow of $k-1$. We continue this process. Every time we find a path and remove it. (For k steps. You could also just say we apply induction here)

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.

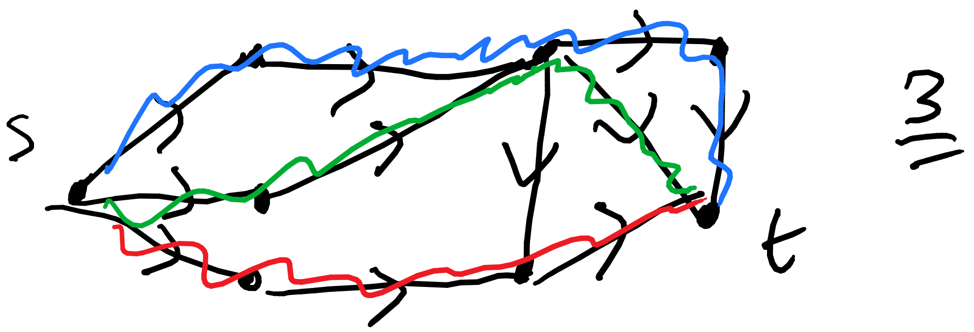
$$\implies \textit{Max-flow} \geq \textit{Max-disjoint paths}$$

With the two inequalities, we have equality:

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

7 01/29/18

Edge disjoint paths in directed graphs



We proved last class that we can convert this problem into a flow network with capacities 1. There were two directions:

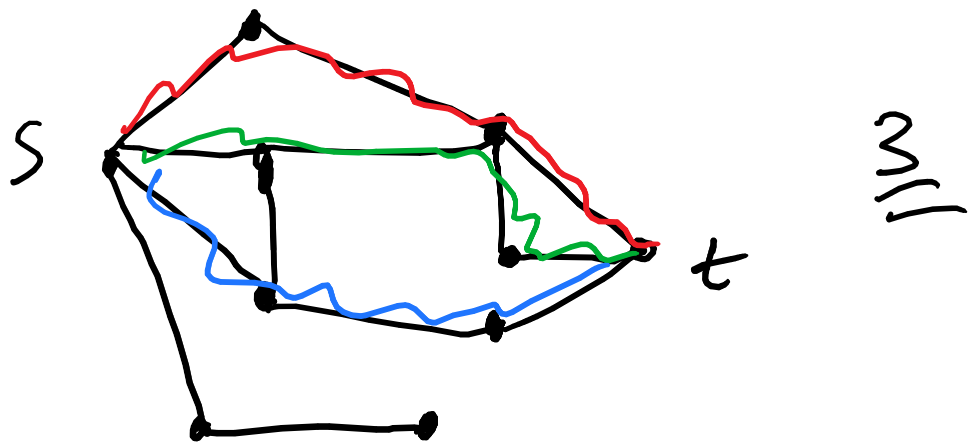
$$\textit{max-flow} \geq \#\text{ paths}$$

(easy, use the paths to direct the flow)

$$\textit{max-flow} \leq \#\text{ paths}$$

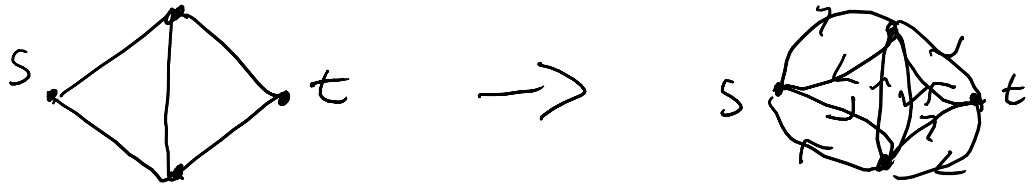
(we start from the flow and trace one path and then remove all the edges of this path. Repeat)

Can we solve the same problem in undirected graphs?

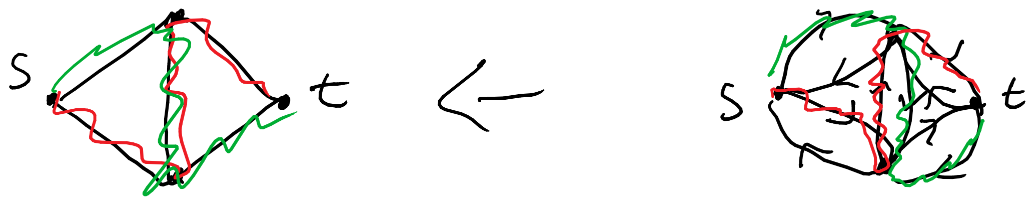


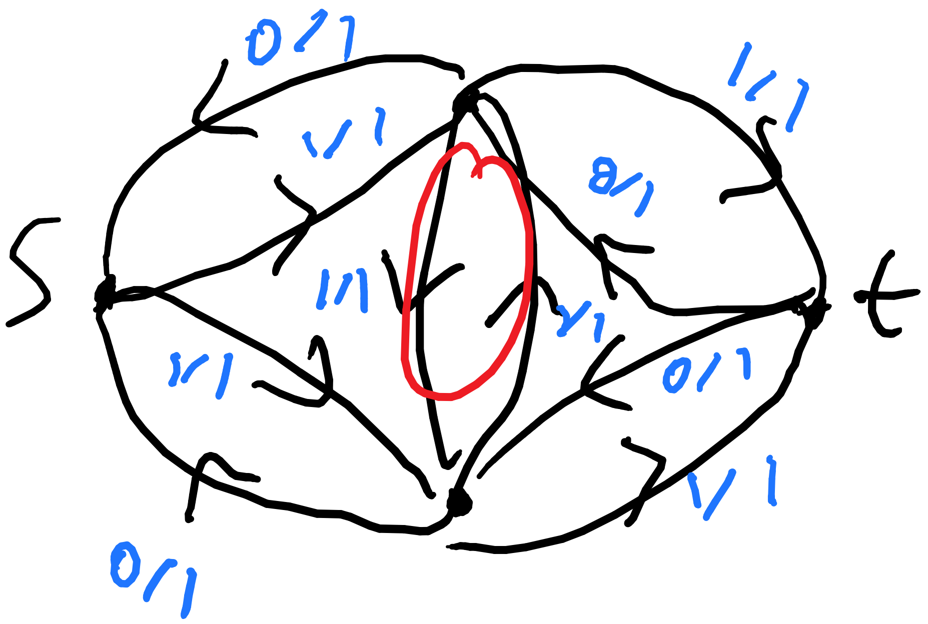
Goal: Find the max # of edge disjoint s-t paths.

We can replace every edge with two directed edges going in opposite directions.

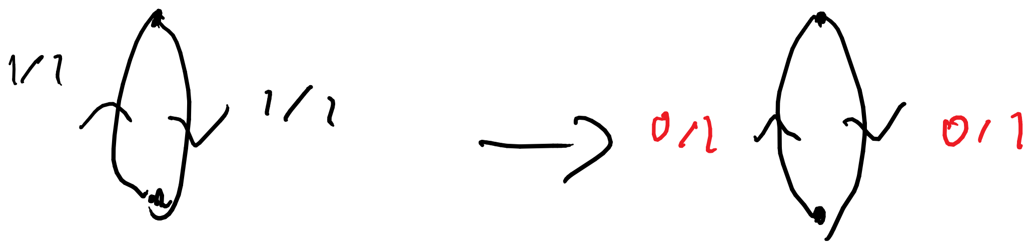


Now we can try to find max # of edge-disjoint paths in this directed graph using Ford Fulkerson. What kind of problems may arise here? This will give us the same number of edge-disjoint paths, but they may reuse edges in the original graph, i.e.





After running Ford Fulkerson if we have:



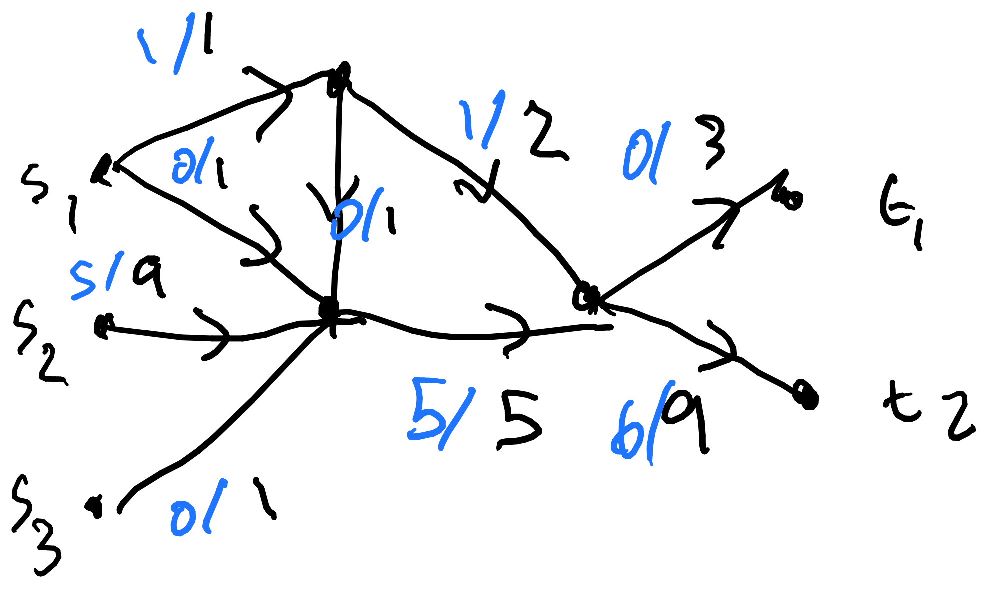
This does not change the value of flow. So we still have a *max-flow*. Using this flow will avoid using shared edges in the undirected graph.

We will be doing a lot more reduction like this, going from a general problem we learned and applying it to specific cases.

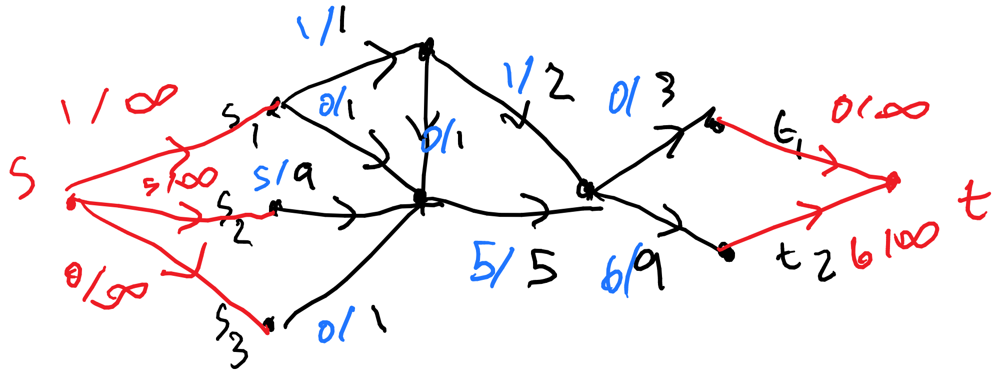
7.1 Multi Source - Multi - Sink Flow

Similar to the original *max-flow* problem except we now might have several sources and several sinks.

Ex



We want to generate the max # of units of flow at the sources.



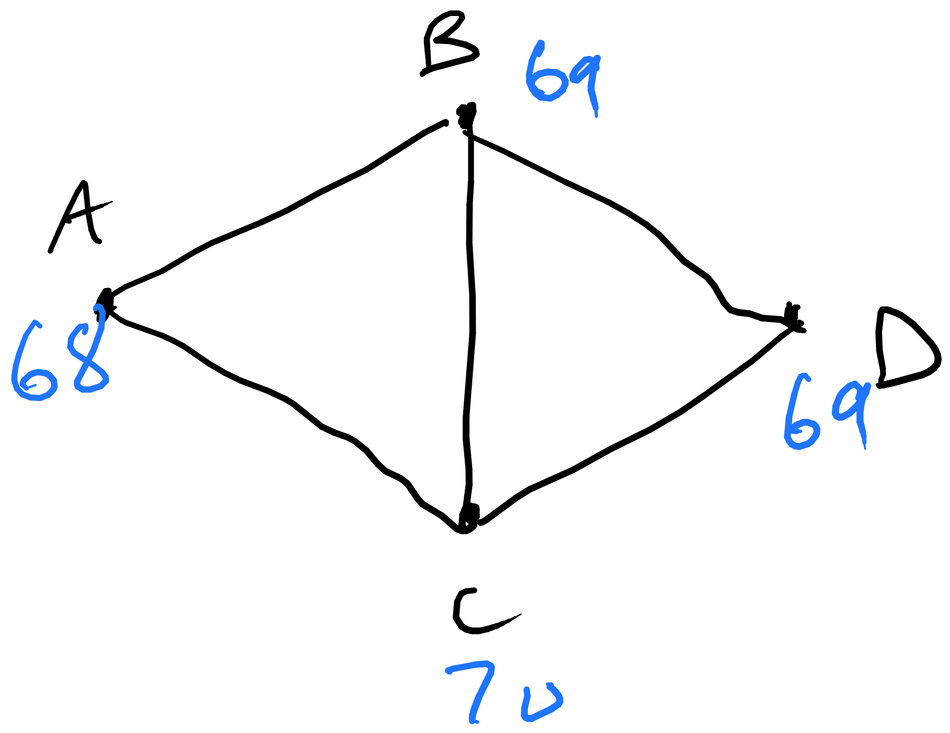
Solution: Add one source s and one sink t. Connect s to all the original sources with edges with ∞-cap and connect the original sinks to t with ∞-cap edges.

The *max-flow* on this new network will give us the desired solution.

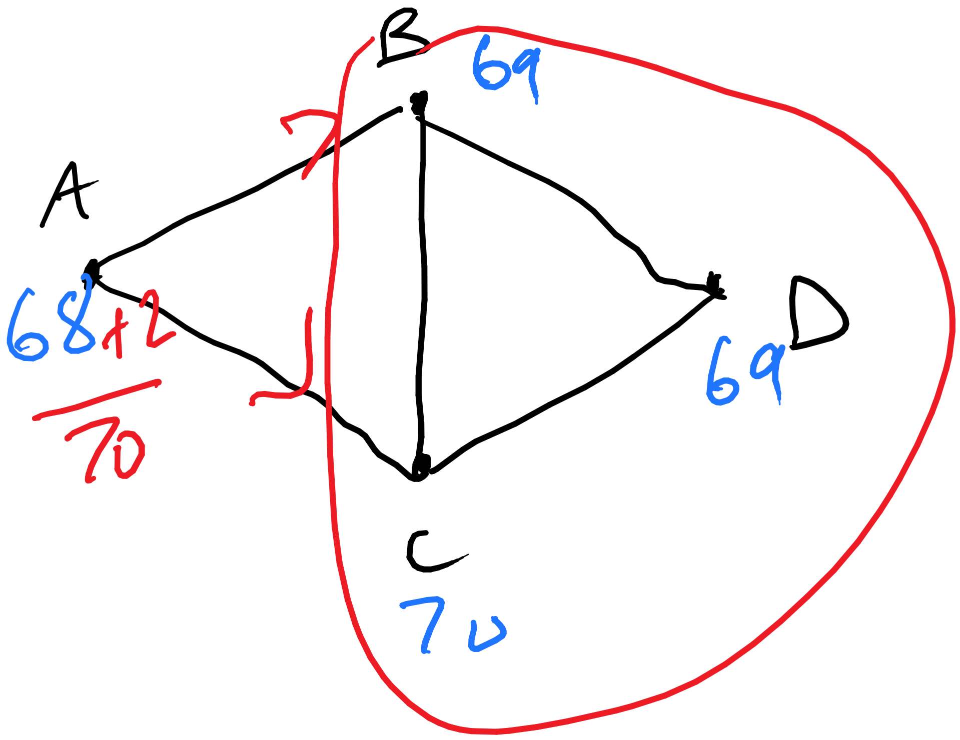
7.2 Baseball Elimination Problem

- We have a tournament.
- Currently we are in the middle and each team has some points and some remaining matches.

- We are interested in a specific team A.
- Does A have any chance of ending with the highest score (possibly in a tie)?



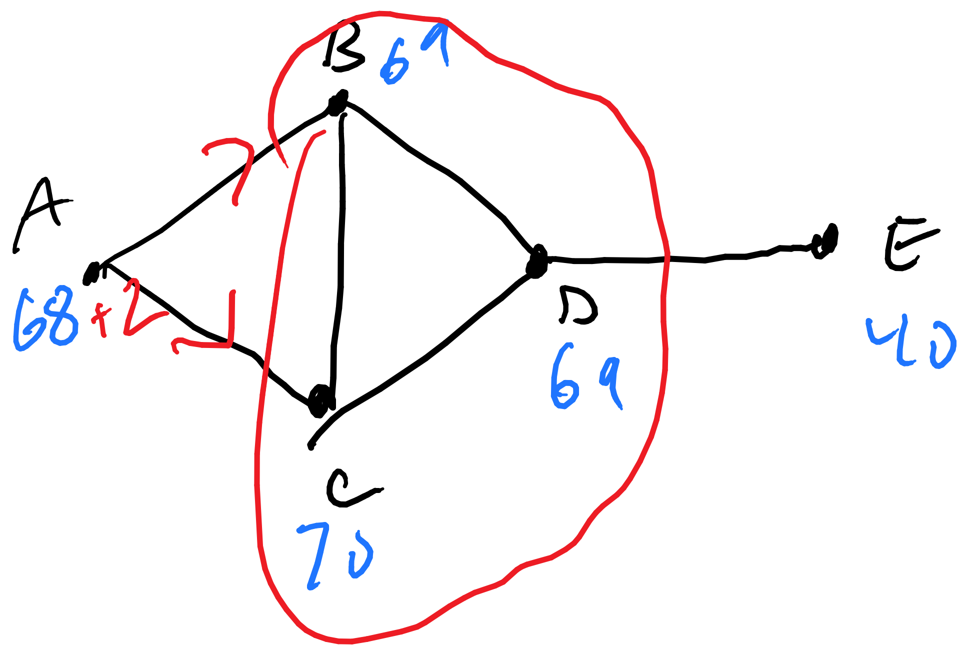
The edges show the remaining matches. Here we show that it is impossible for A to come out on top:



We see that even if we make A win both of its games, at least one of the teams encircled will end up with > 70 points, because if C doesn't get past 70 then it must lose both its matches and the winner among B and D will have 71. $\implies$ A is eliminated.

$$70 + 69 + 69 + 3 \leq \text{final points between } B, C, D$$
$$\frac{70 + 69 + 69 + 3}{3} > 70 \implies \text{ at least one will have } > 70 \text{ pts}$$

Will this always work though?



$$\frac{69+70+69+40+5}{4} < 70$$

Nevertheless A is still eliminated! However focusing only on B, C, D the argument still works:

$$\frac{70+69+69+3}{3} > 70$$

Is this always the case? Can I always find teams such that the average of their scores after factoring winning is greater than my team?

Let M be the total points A will have if it wins all its remaining matches. If A is eliminated then is it true that we can find a set T of teams such that

$$\frac{\left(\sum_{x\in T} P_x\right)+k}{|T|} > M?$$

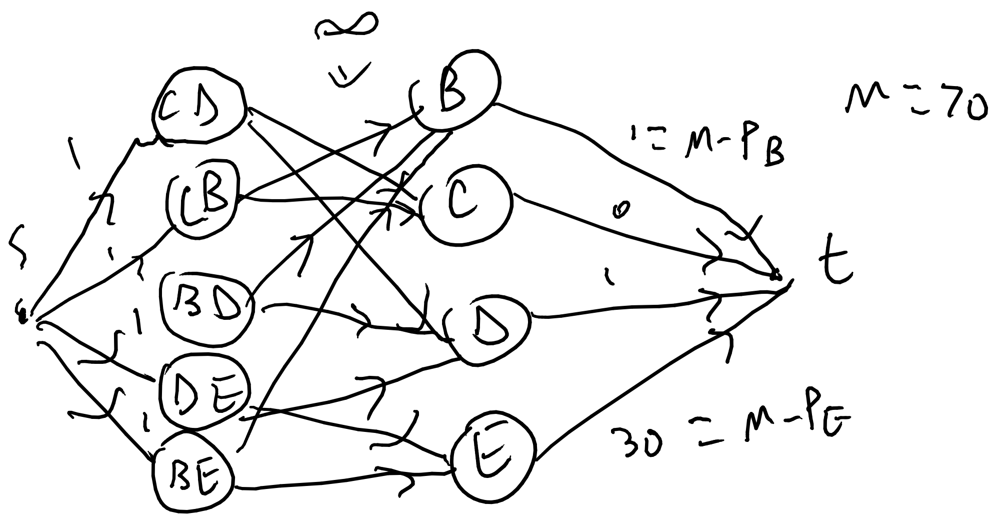
Where k is the number of remaining matches between teams in T and P_x is the points of x.

We want to show that this is true.

How can we decide whether A is eliminated?

1. Let M be the max number of points A can collect if it wins all the remaining matches. $M = P_A + deg(A)$. So now we are done dealing with A and can look at the rest of the graph.

2. Construct the following flow network: For every edge uv put a vertex uv in. If $M - P_x$ is negative then this is impossible, A is already eliminated.

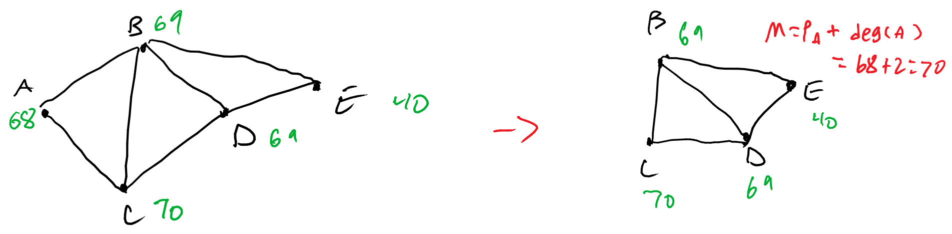


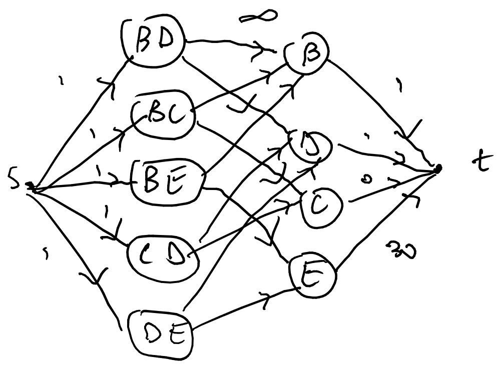
3. Add a source. Connect it to all uv with capacity 1 edges. Add edges uv-u and uv-v with ∞-cap. Add edges u-t.

4. We solve the *max-flow* if its value equals to the outgoing capacity of $s \implies A$ is not eliminated. Otherwise it is.

8 01/31/18

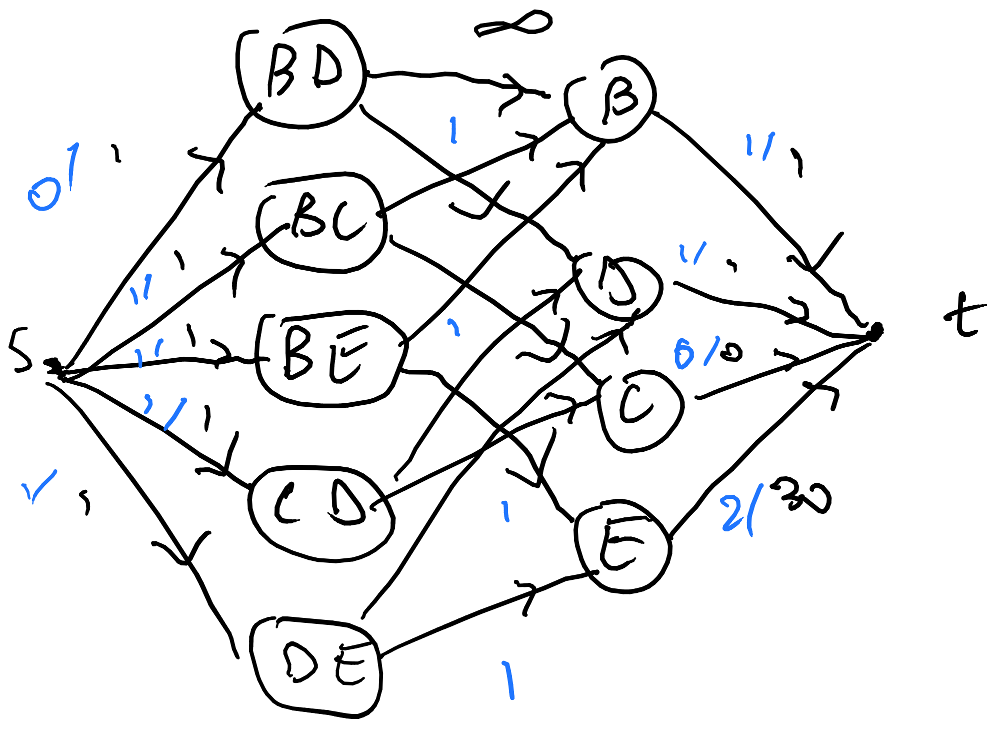
8.1 Baseball Elimination





The problem we want to solve is whether every team can have at most M points (not over) and we found a clever way to model this problem with a flow network. We give edges toward t the capacity $M - P_i$, for example $1 = M - P_B = 70 - 69$ in the above.

Solve *max-flow*. If it is equal to # remaining matches $\implies$ not eliminated.

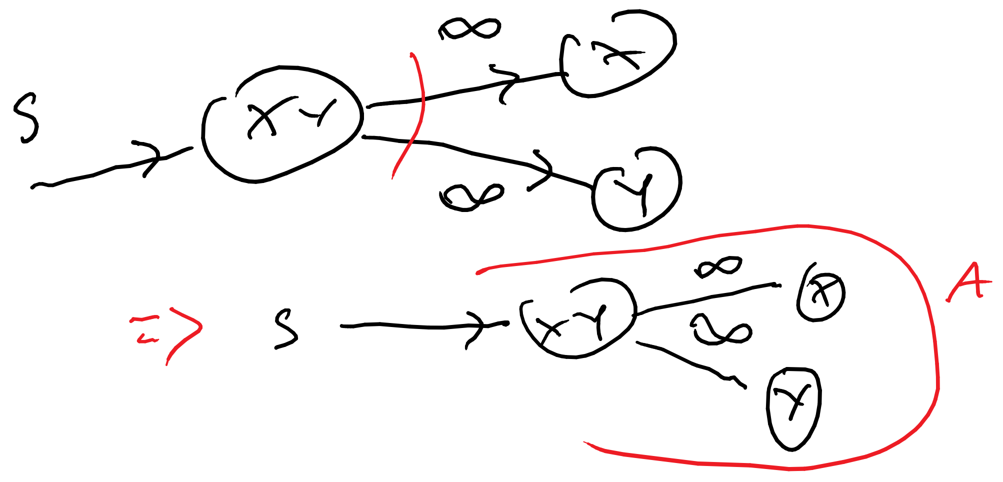


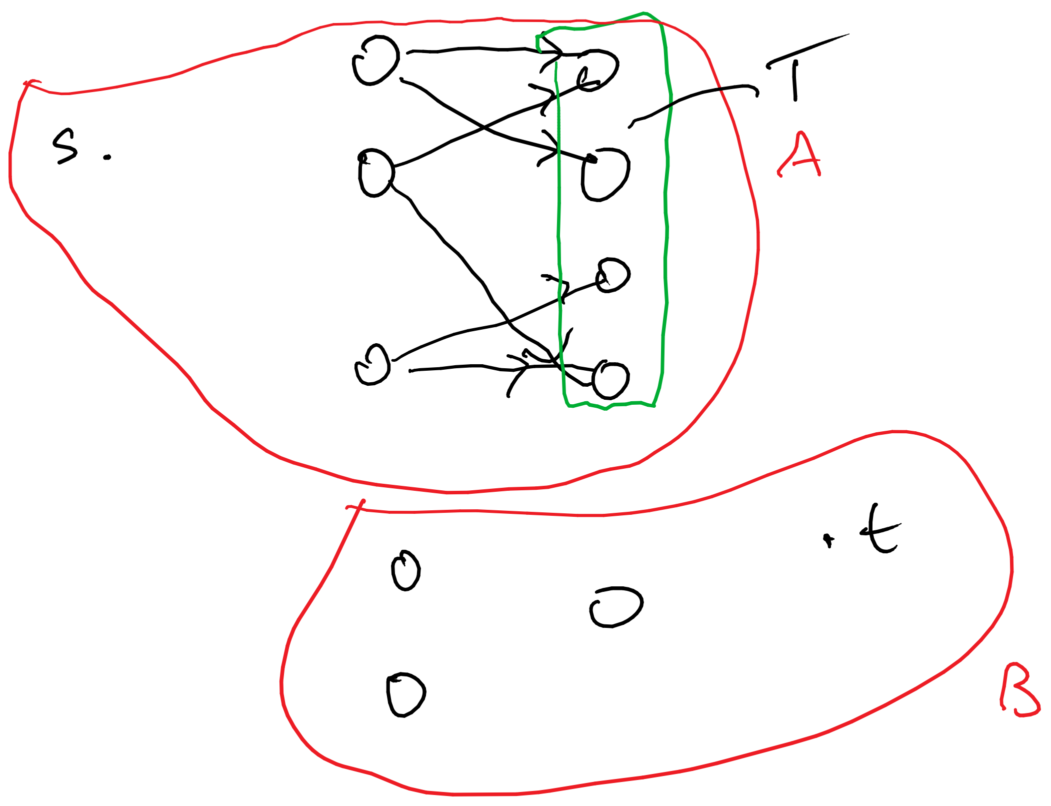
So we see in the example that it is not possible that A is not eliminated, since the *max-flow* is 4.

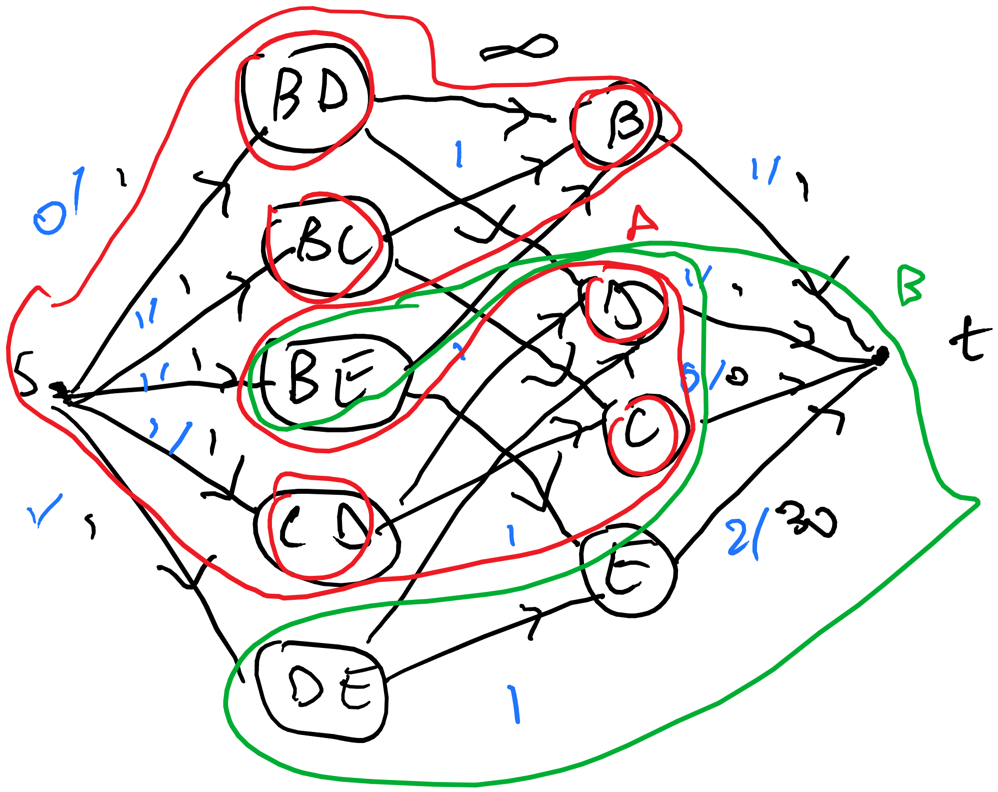
We know *max-flow* = *min-cut*. What does *min-cut* tell us?

What can we say about *min-cut*?

- *Min-cut* $\neq \infty$ (we already have $A = \{s\}, B = \{s\}^c$)
- Consider *min-cut*(A, B). If $xy \in A \implies x \in A, y \in A$ (or else the capacity would be ∞)


S
XY
∞
X
Y
⇒
A


S
T
A
t
B



Let T be the set of teams in A from cut(A, B)

$$\begin{aligned}Cap(A,B) &= \sum_{x \in T}(M - P_x) + \text{number of matches } xy \text{ with at least one of } x \text{ or } y \text{ not in } T\\ &= \mathit{max\text{-}flow}\end{aligned}$$

Call this number of matches K

What do we know about *max-flow*?

- If *max-flow* $<$ # of edges $\implies$ our team is eliminated.

$$\begin{aligned}\mathit{Max\text{-}flow} = Cap(A,B) &= M \times |T| - \sum_{x\in T} P_x + K\\ \iff M \times |T| - \sum_{x \in T} P_x + K &< \text{\# of edges}\end{aligned}$$

$$\iff M \times |T| < \# \text{ edges in } T + \sum_{x \in T} p_x$$
$$\iff M < \frac{\# \text{ edges in } T + \sum_{x \in T} p_x}{|T|}$$

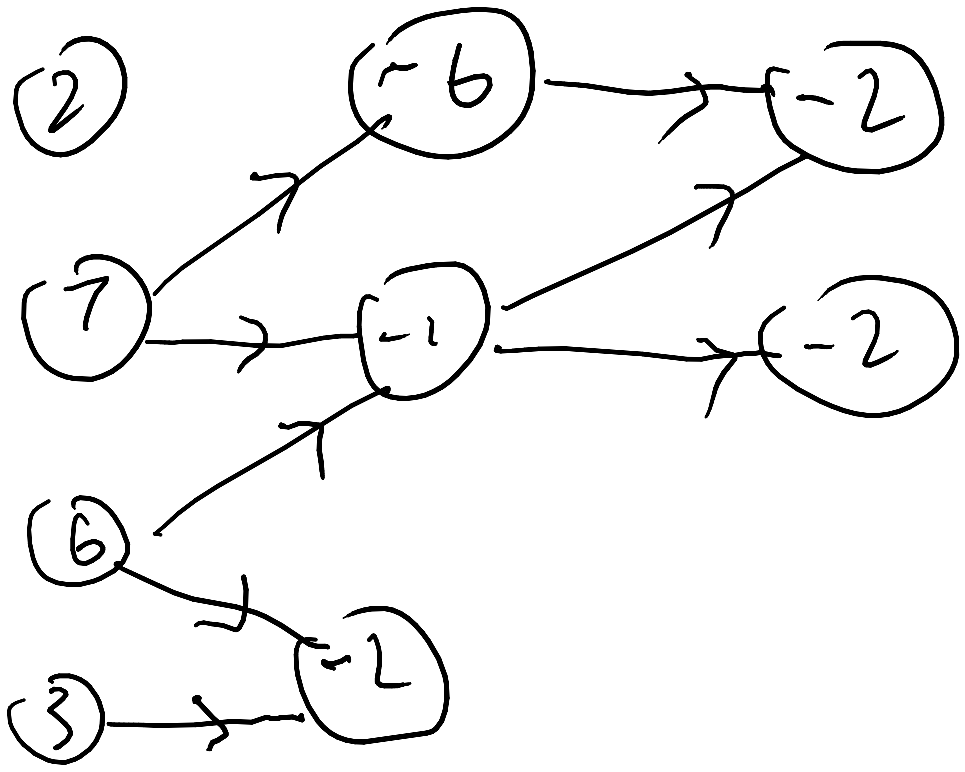
This proves the theorem: If our team is eliminated $\implies$ there exists a set of teams T that provides a proof:

$$M < \frac{\# \text{ edges in } T + \sum_{x \in T} p_x}{|T|}$$

8.2 Project Selection

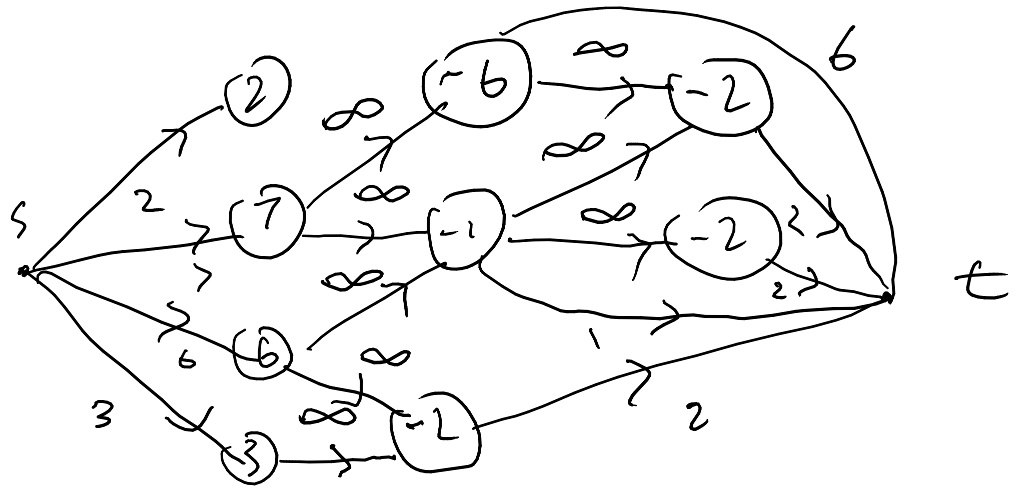
- We are given a set of projects.
- Each project x has a revenue $P_x = \begin{cases} \text{provides a profit} & \text{if } P_x > 0 \\ \text{Provides a loss} & \text{if } P_x < 0 \end{cases}$
- Some projects are prerequisites for other projects.
- An edge $x \to y$ means that y is a prerequisite for x (if we choose x we also have to choose y). **Goal:** Select a subset of projects that respects all the prerequisites and maximizes the total revenue.

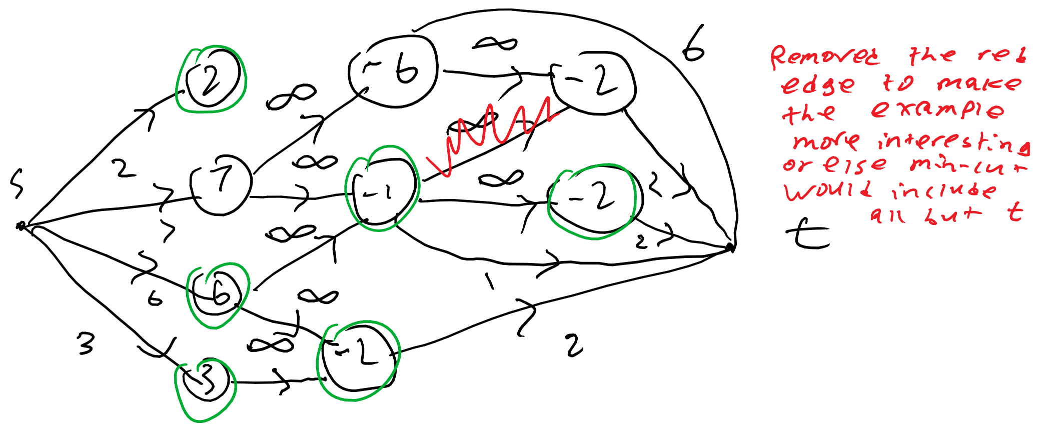
Ex



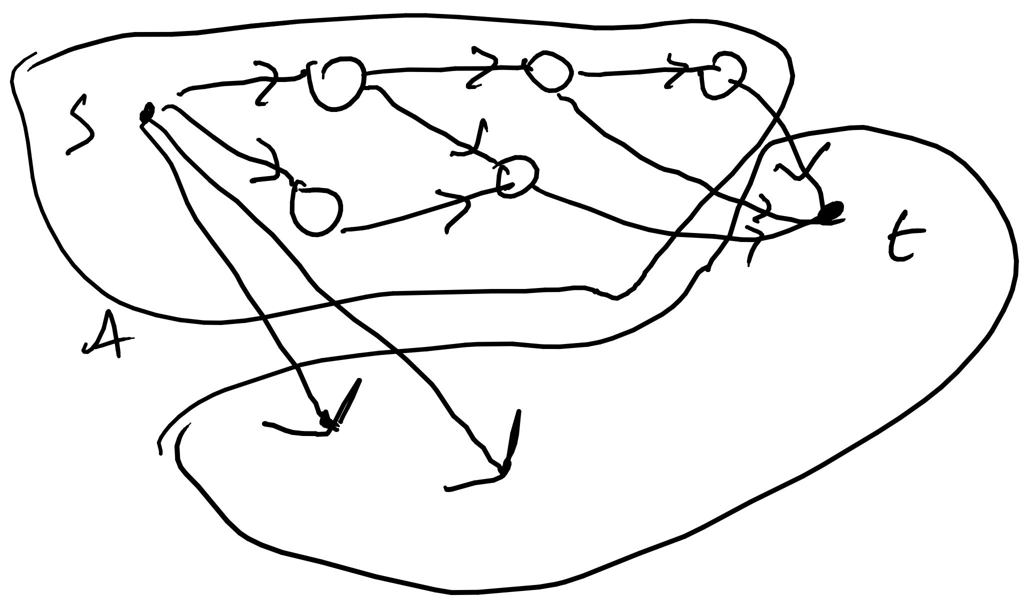
So we want to maximize profit.

- We will use *min-cut*.
- We assign ∞ capacity to all the edges. This way if a project x is in part A of a *min-cut* and we have the prereq $x \to y$ then y also has to be in A (i.e. you cannot cut any of the infinite edges linking prerequisites).
- We add a source s, a sink t, edges from s to x for projects with $P_x > 0$ (capacity P_x), edges from x with $P_x < 0$ to t with cap $|P_x|$




s
t
Removed the red edge to make the example more interesting or else min-cut would include all but t

Let (A, B) be a *min-cut* and let $M = \sum_{x:P_x>0} P_x$



$$Cap(A,B) = ? = \sum_{\substack{x\in B\\P_x>0}} P_x + \sum_{\substack{x\in A\\P_x<0}} |P_x| = \sum_{\substack{x\in A\\P_x<0}} -P_x + \sum_{\substack{x\notin A\\P_x>0}} P_x = \sum_{\substack{x\in A\\P_x<0}} -P_x + \left(M - \sum_{\substack{x\in A\\P_x>0}} P_x\right)$$
$$= M - \sum_{x\in A} P_x$$

- We also know that the projects in A respect the prereq condition.

min-cut is minimizing the term above, which maximizes the negative sum. So in order to max our profit, we choose all the jobs in A.

The total profit we can make $= \sum_{x\in A} P_x$

9 02/05/18

9.1 Linear Programming

What is linear programming? So far we've done many optimization problems, where we have many constraints that we have to satisfy and what we wanted to do is optimize the function (max flow, maximum matching, min cut, etc.).

A linear program is a special class of optimization problems: optimizing a linear function in a certain number of variables over a set of linear constraints.

Why do we care about this?

- Many optimization problems can be modeled as Linear Programs
- 40's: A very practical algorithm (called simplex) discovered for solving LP's. *Theoretically it is not an efficient algorithm (exponential time), but in practice it's almost always very fast.*
- 79 (Leonid Khachiyan) Proved that LP's can be solved in polynomial time (Ellipsoid algorithm, worst case is better than Simplex, but overall slower)

A linear program has

1. A set of variables: $x_1, \ldots, x_n$ that can take **real** values.
2. A set of linear constraints each of the form

$$\alpha_1 x_1 + \alpha_2 x_2 + \ldots + \alpha_n x_n = \beta$$
$$\alpha_1 x_1 + \alpha_2 x_2 + \ldots + \alpha_n x_n \leq \beta$$
$$\alpha_1 x_1 + \alpha_2 x_2 + \ldots + \alpha_n x_n \geq \beta$$

   where $\alpha_1, \ldots, \alpha_n, \beta$ are (fixed) real numbers. Note that we cannot have strict inequalities, because then we will never be able to optimize the problem, it'll be an open problem which we can keep making better and better.

3. A linear objective function that we want to **minimize** or **maximize**.

$$c_1 x_1 + c_2 x_2 + \ldots + c_n x_n$$

   where $c_1, \ldots, c_n$ are real numbers.

It would be good to follow these steps whenever you are trying to formulate something as a linear programming problem and/or if you are given a linear programming problem, especially if written in an abstract way.

Example

- Variables x_1, x_2, x_3

- max $2x_1 + 5x_2 - x_3$ (objective function)
- Subject to (s.t.)

$$\begin{aligned} x_1 + x_2 + x_3 &\leq 5 \\ 2x_1 + 6x_2 - x_3 &\leq 1 \\ -x_1 - 2x_2 - x_3 &= 2 \end{aligned}$$

(constraints)

Example

- variables x_1, x_2
- $\min x_1 + x_2$
- s.t.

$$\begin{aligned} x_1 + 2x_2 &\geq 1 \\ x_1 - x_2 &= 5 \\ x_2 &\geq 0 \end{aligned}$$

(note this is still a linear program, the last equation has $0x_1$ omitted, the α's don't need to be nonzero)

Example

- variables x_1, x_2, x_3
- $\max x_1 + x_2 + x_3$
- s.t.

$$\begin{aligned} x_1 + 2x_2 &\leq 1 \\ 2x_1 + x_2 &\leq 1 \\ x_3 &= 1 \\ x_1 &\geq 0 \\ x_2 &\geq 0 \end{aligned}$$

$x_1 = \frac{1}{3}, x_2 = \frac{1}{3}, x_3 = 1$ gives $\frac{5}{3}$. It's easy to convince someone that the maximization is at least some number, just give them an example like this. But how do we prove to someone that this is the best? What happens if we add the first two constraints? We get $3x_1+3x_2 \leq 2$. More rigorously we can do the following:

$$\begin{aligned} 1 \times (x_1 + 2x_2 \leq 1) \\ 1 \times (2x_1 + x_2 \leq 1) \\ 3 \times (x_3 = 1) \end{aligned}$$

We get:

$$\begin{aligned} 3x_1 + 3x_2 + 3x_3 &\leq 5 \\ \iff x_1 + x_2 + x_3 &\leq \frac{5}{3} \end{aligned}$$

We will see a big theorem that tells us we can always do this. Note, we are showing that the maximum is at least $\frac{5}{3}$ and then we are showing that the maximum cannot be larger than $\frac{5}{3}$, in other words, this is similar to showing *max-flow* and *min-cut*.

Example We have a small firm producing bookcases and tables.

	Cutting time	Assembly	Finishing time
Bookcase	$\frac{6}{5}$ hr	1 hr	$\frac{3}{2}$ hr
Table	1 hr	$\frac{1}{2}$ hr	2 hr

We have people working for us following:

- 72 hr cutting time
- 50 hr assembly
- 120 hr finishing time

We can sell:

- A bookcase \$80
- A table \$55

Goal: maximize profit. How many tables and bookcases should we build?

- Variables: x_1 number of tables and x_2 number of bookcases (these are our unknowns, what we are trying to solve for)

- Objective: $\max 55x_1 + 80x_2$
- Constraints:

$$\begin{aligned} x_1 + \frac{6}{5}x_2 &\leq 72 \text{ (cutting)} \\ \frac{1}{2}x_1 + x_2 &\leq 50 \text{ (assembly)} \\ 2x_1 + \frac{3}{2}x_2 &\leq 120 \text{ (finishing)} \\ x_1 &\geq 0 \quad \text{(Can't make negative number of tables} \\ x_2 &\geq 0 \quad \text{ or bookcases)} \end{aligned}$$

Remark: We wish to add that x_1, x_2 are integers but adding that is not allowed in LP's (we cannot solve such optimization problems efficiently).

If we solve the linear program

$$\begin{aligned} \max 55x_1 + 80x_2 \\ \frac{1}{2}x_1 + x_2 &\leq 50 \\ 2x_1 + \frac{3}{2}x_2 &\leq 120 \\ x_1 &\geq 0 \\ x_2 &\geq 0 \end{aligned}$$

We get $x_1 = 30, x_2 = 35$. Fortunately in this case the optimal solution is integer (in practice we'd round them to integers if required).

Example

- Two factories P_1, P_2
- Four products A, B, C, D

	$P_!$ (prod/day)	P_2 (prod/day)	total demand
A	200	100	1000
B	60	200	800
C	90	150	900
D	130	80	1500

Cost of running P_1: $\$800/day$, P_2 : $\$1100/day$

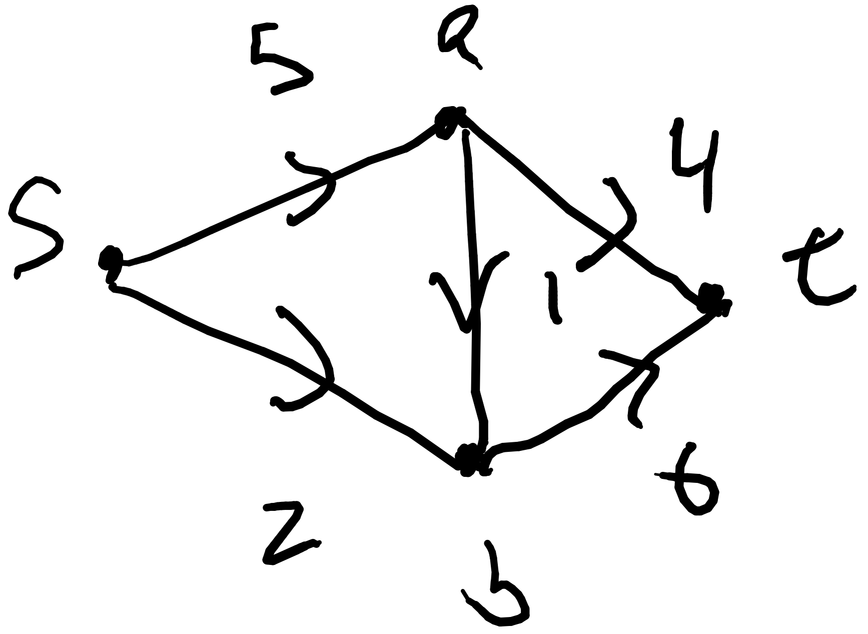
- Goal: meet the demands minimize the cost.
- How many days of P_1? x_1
  How many days of P_2? x_2
- Constraints:

$$\begin{aligned} A : 200x_1 + 100x_2 &\geq 1000 \\ B : 60x_1 + 200x_2 &\geq 800 \\ C : 90x_2 + 150x_2 &\geq 900 \\ D : 130x_1 + 80x_2 &\geq 1500 \\ x_1 &\geq 0 \\ x_2 &\geq 0 \end{aligned}$$

Solution: $x_1 = 11.132, x_2 = 0.66$. $obj = 96320.75$

Model the following problem as a linear program:

"Find the max flow in this network"



- Variables: $f_{sa}, f_{sb}, f_{ab}, f_{at}, f_{bt}$
- Objective: $\max f_{sa} + f_{ab}$
- Constraints:

$$f_{sa} \geq 0$$
$$f_{sb} \geq 0$$
$$f_{ab} \geq 0$$
$$f_{at} \geq 0$$
$$f_{bt} \geq 0$$

(positive flow)

$$f_{sa} \leq 5$$
$$f_{sb} \leq 2$$

$$f_{ab} \leq 1$$
$$f_{at} \leq 4$$
$$f_{bt} \leq 6$$

(capacity condition)

$$f_{sa} - f_{at} - f_{ab} = 0$$
$$f_{sb} + f_{ab} - f_{bt} = 0$$

(conservation conditions)

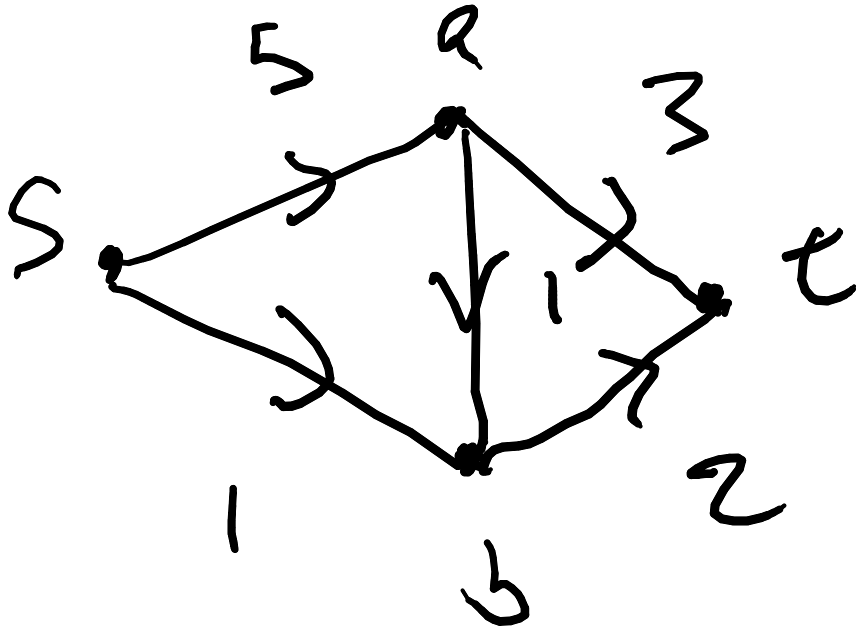
So the fact that we can solve max flow in polynomial time follows from the fact that we can solve linear programs in polynomial time.

Max flow problem is a special case of linear programs.

10 02/07/18

10.1 Modeling Problems as Linear Programs

Max flow



Variables:

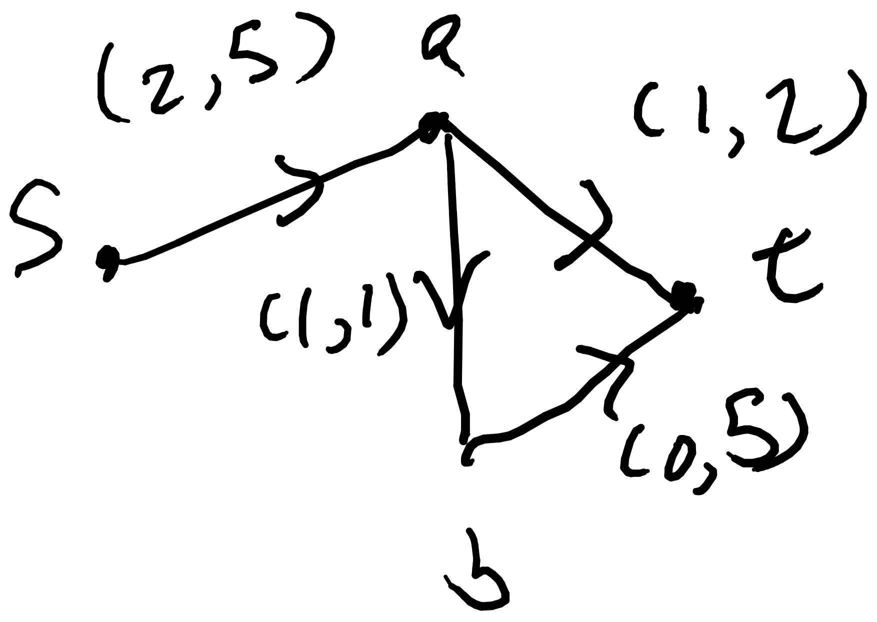
- f_{sa}
- f_{sb}
- f_{ab}
- f_{ab}
- f_{bt}

Objective: $\max f_{sa} + f_{sb}$
Constraints:

- $f_{sa} \leq 5$, $f_{sa} \geq 0$

- $f_{sb} \leq 1,\ f_{sb} \geq 0$
- $f_{ab} \leq 1,\ f_{ab} \geq 0$
- $f_{at} \leq 3,\ f_{at} \geq 0$
- $f_{bt} \leq 2,\ f_{bt} \geq 0$
- $f_{sa} - f_{ab} - f_{at} = 0$
- $f_{sb} + f_{ab} - f_{bt} = 0$

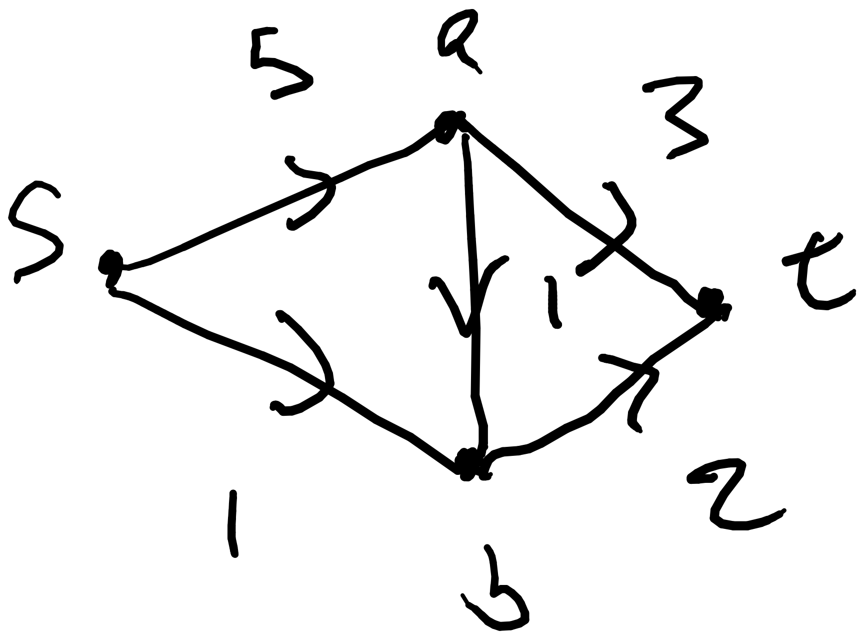
Example Suppose some edges also have lower bounds. That is an edge e with lower bound l_e requires that the flow on e has to be at least l_e. This can be formulated as an LP. E.g.



- $\max f_{sa}$
- $f_{sa} \leq 5,\ f_{sa} \geq 2$
- $f_{at} \leq 2,\ f_{at} \geq 1$

- $f_{ab} \leq 1$, $f_{ab} \geq 1$
- $f_{bt} \leq 5$, $f_{bt} \geq 0$
- $f_{sa} - f_{at} - f_{bt} = 0$
- $f_{ab} - f_{bt} = 0$

We can add costs to edges. Every edge e has cost d_e. That is passing f_e unit of flow through the edge costs $d_e \times f_e$. What is max flow with cost at most d, when d is given to us. (think of d like a budget)



$d = 4$

Costs:

d_{sa}	10
d_{sb}	1
d_{ab}	1
d_{at}	2
d_{bt}	1

Vars: $f_{sa}, f_{ab}, f_{at}, f_{bt}, f_{sb}$

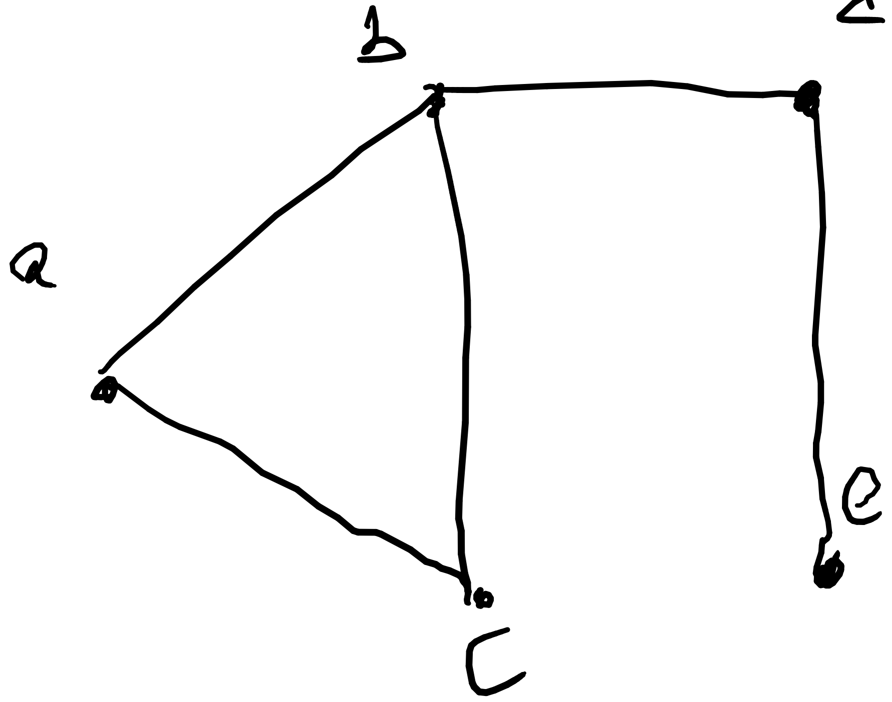
$\max f_{sa} + f_{sb}$

Subject to

- $f_{sa} \leq 5$
- $\dots$
- $f_{sb} \leq 1$
- $f_{sa} \geq 0$
- $\dots$
- $f_{bt} \geq 0$
- $f_{sa} - f_{ab} - f_{at} = 0$
- $f_{sb} + f_{ab} - f_{bt} = 0$
- $\sum_{\text{edge } e} f_e \cdot d_e \leq 4$

Ex We have a network and highways between cities, given to us as an undirected graph. We are given a positive number $\alpha \geq 0$. We want to store some amount of supply in each city so that the sum of supply in each city and its neighboring cities is at least α. What is the minimum total supply that we need to meet this condition?

Ex $\alpha = 6$



vars: x_a, x_b, x_ckx_d, x_e corresponding to the supply in cities a, b, c, d, e.
$\min x_a + x_b + x_c + x_d + x_e$
Subject to

$$\begin{aligned} x_a + x_b + x_c \geq 6, x_a \geq 0 \\ x_b + x_a + x_c + x_d \geq 6, x_b \geq 0 \\ x_c + x_a + x_b \geq 6, x_c \geq 0 \\ x_d + x_b + x_e \geq 6, x_d \geq 0 \\ x_e + x_d \geq 6, x_e \geq 0 \end{aligned}$$

For a general graph $G = (V, E)$ we can write this as:
Variables: $\forall v \in V$ we have a variables x_v.
$\min \sum_{v \in V} x_v$

s.t.

$$x_v + \sum_{u:uv\in E} x_u \geq \alpha, \forall v \in V$$
$$x_v \geq 0, \forall v \in V$$

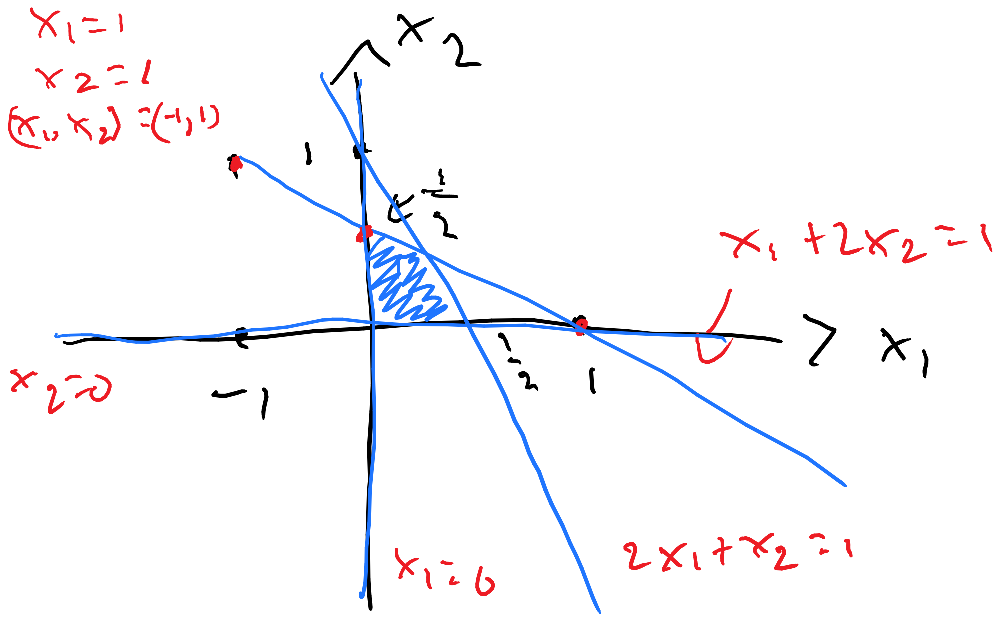
This gives us $2|V|$ constraints.

10.2 Geometric Interpretation of LP's

Consider the following LP.

$$\max\ x_1 + x_2$$
$$x_1 + 2x_2 \leq 1$$
$$2x_1 + x_2 \leq 1$$
$$x_1, x_2 \geq 0$$

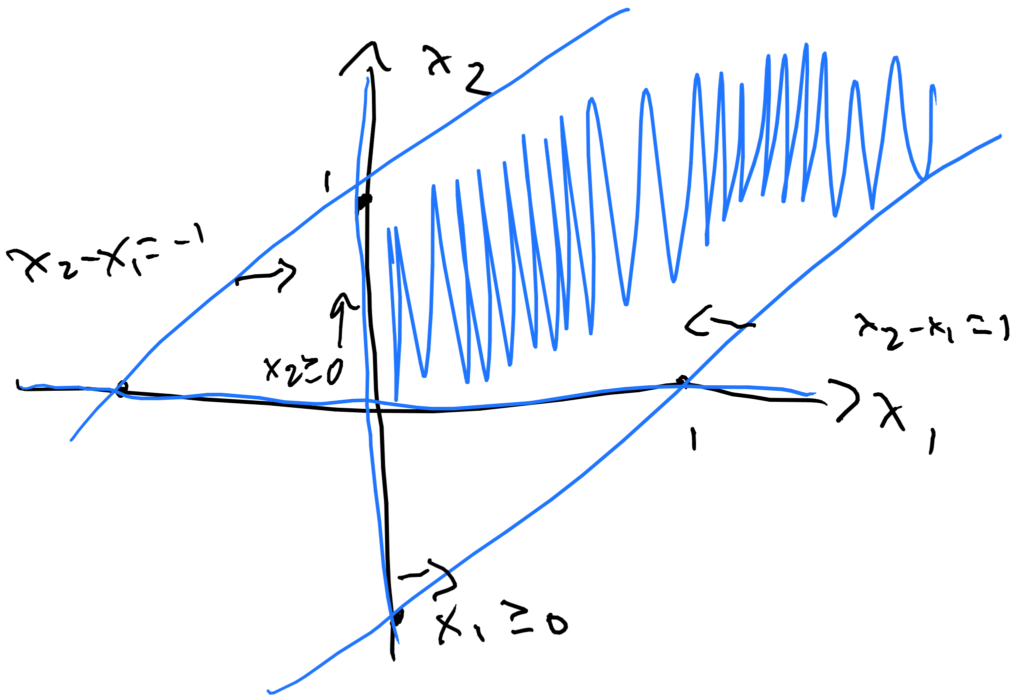
Every potential solution x_1, x_2 gives us a point (x_1, x_2) on the plane. What does the constraint $x_1 + 2x_2 \leq 1$ tell us? What about the other constraints?



The above is the set of points that satisfy all the constraints. This is called the feasible region. It is the set of all points that satisfy all the constraints. (feasible solutions).

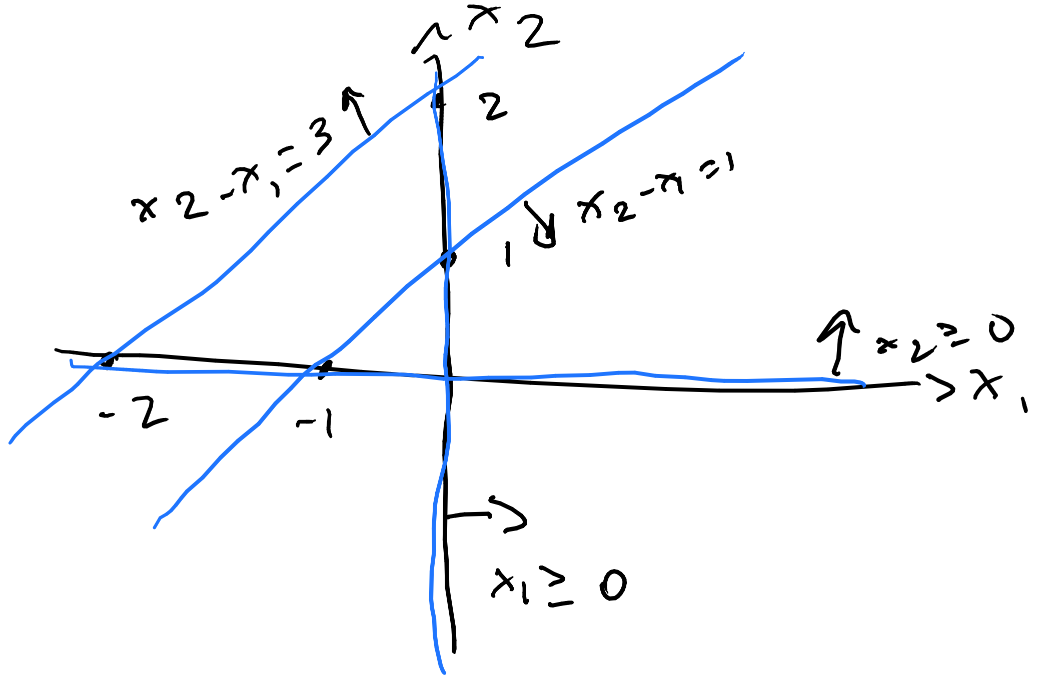
Remark Here every constrain with an inequality gives us a half space, which is the set of points that satisfy that constraint. The feasible region is the intersection of them.

Ex Draw the feasible region for the following constraints:
$x_1 \geq 0 \quad x_2 - x_1 \geq -1$
$x_2 \geq 0 \quad x_2 - x_1 \leq 1$



Here the feasible region is unbounded.

Ex $\begin{array}{ll} x_1 \geq 0 & x_2 - x_1 \geq 2 \\ x_2 \geq 0 & x_2 - x_1 \leq 1 \end{array}$



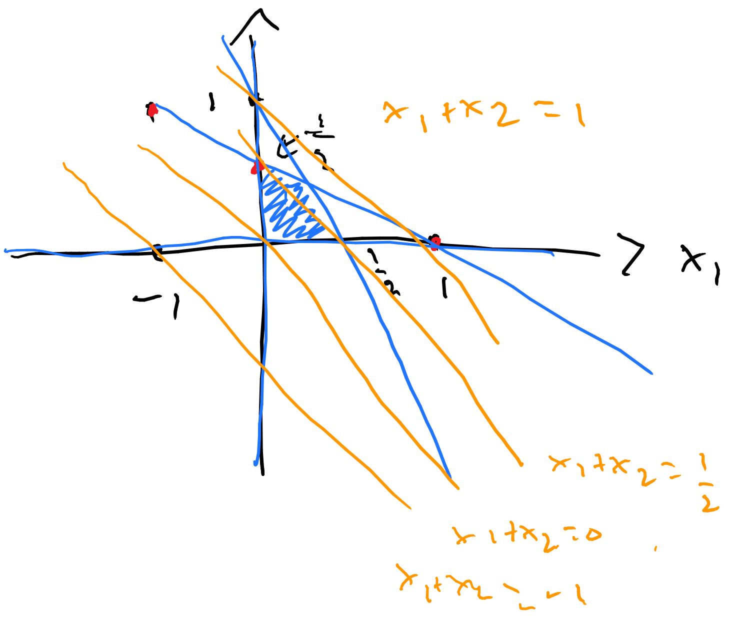
Three possible cases for feasible regions:

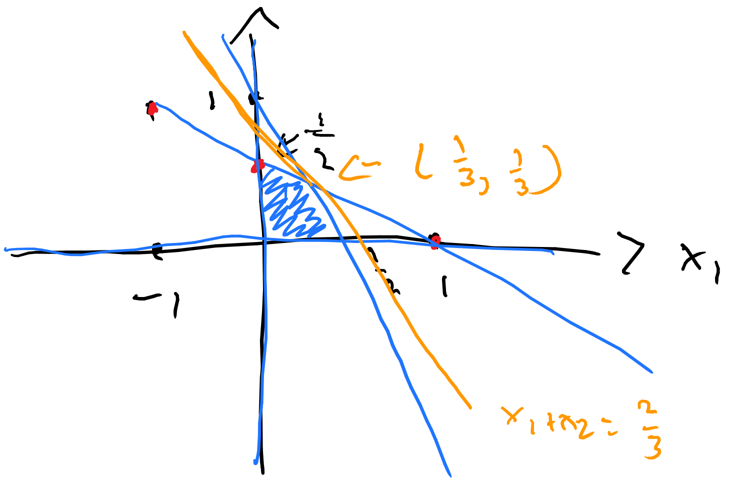
- Bounded
- Unbounded
- Empty

First example $\max x_1 + x_2$?

$$\begin{aligned} \max x_1 + x_2 \\ x_1 + 2x_2 \leq 1 \\ 2x_1 + x_2 \leq 1 \\ x_1, x_2 \geq 0 \end{aligned}$$

Try $x_1 + x_2 = \alpha$.





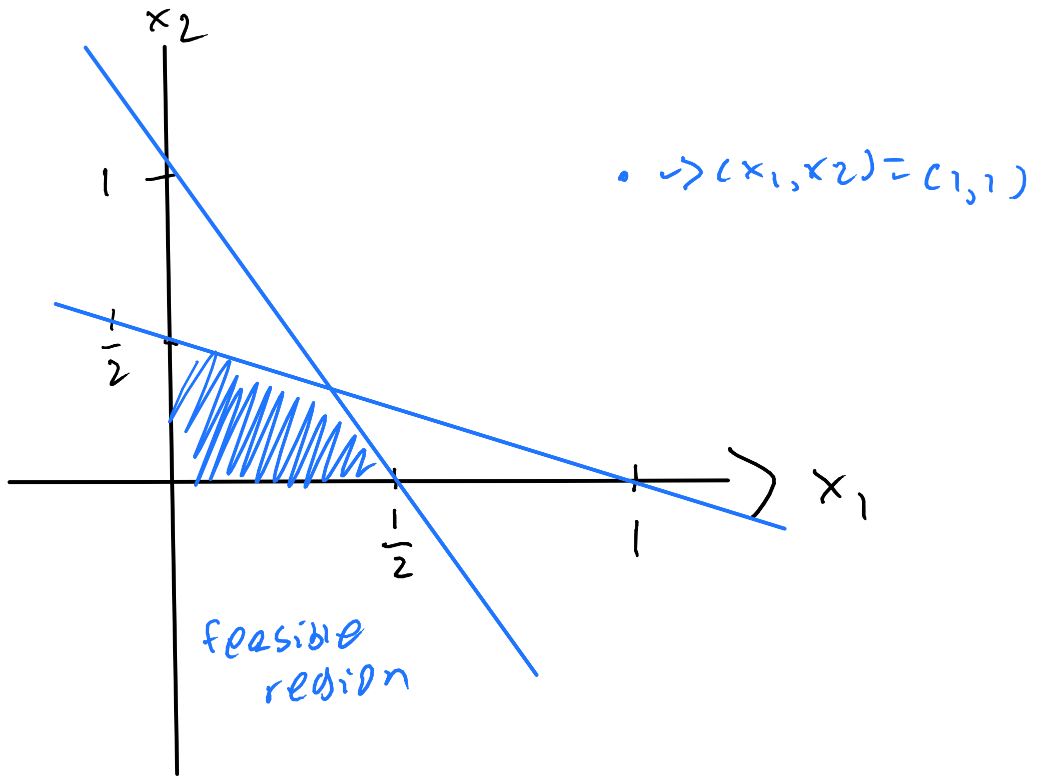
We got a line that intersects with the boundary of the feasible region.

Midterm on Wednesday, covers everything up to today, mainly max flow, little bit of linear programming formulation and maybe some geometric interpretation.

11 02/12/18

Recall Geometric Interpretation of LP's.

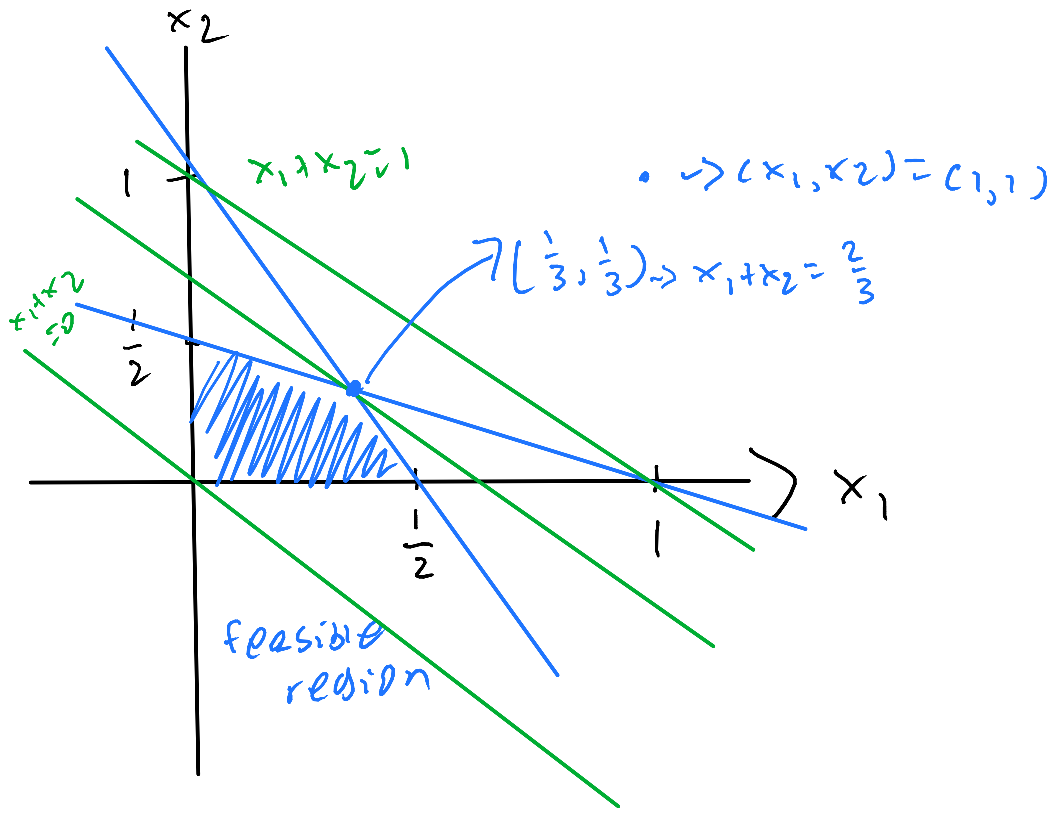
$$\begin{aligned} \max\ & x_1 + x_2 \\ & x_1 + 2x_2 \leq 1 \\ & 2x_1 + x_2 \leq 1 \\ & x_1, x_2 \geq 0 \end{aligned}$$



Each constraint gives us a half plane, a line such that the points that satisfy the inequality must be above or below the line. We then get a feasible region, i.e. the area where all points satisfy the constraints. We then want to maximize or minimize the variables.

$$x_1 + x_2 = \alpha?$$

So we are interested in finding a point that is in the feasible region that gives us the largest value for $x_1 + x_2 = \alpha$

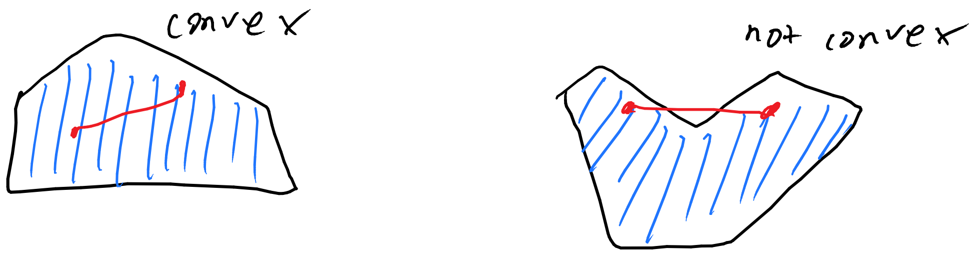


Feasible region: The set of all solutions that satisfy all the constraints.

It can be:

1. Empty.
2. Unbounded.
3. It is a bounded region inside a convex polygon.

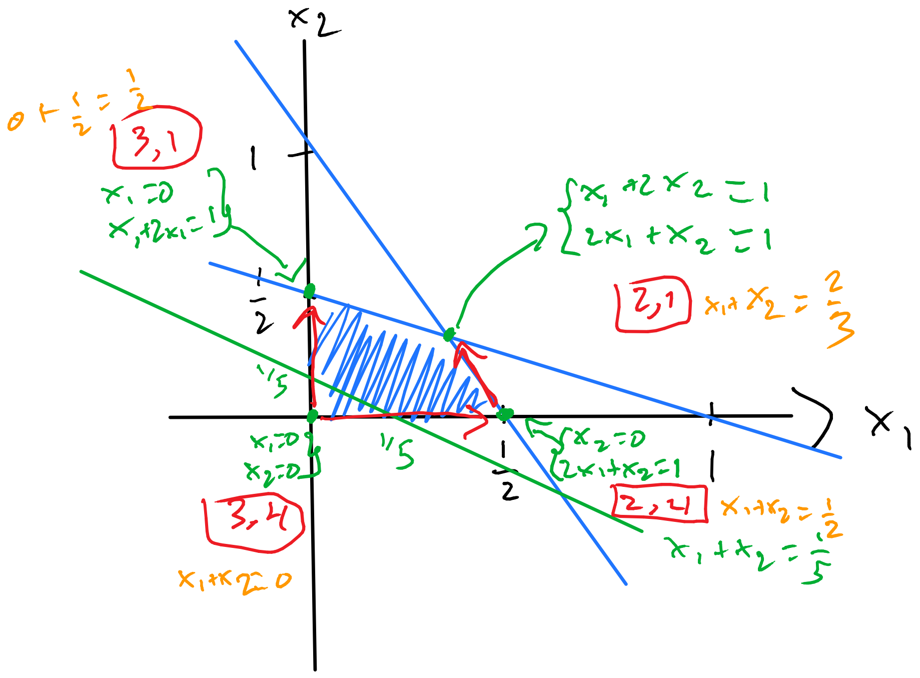
Convex: The line segment between any two points in the region falls into the region.



Let's try and solve the same LP in a more systematic way.

$$\max x_1 + x_2$$

$$x_1 + 2x_2 \leq 1 \tag{1}$$
$$2x_1 + x_2 \leq 1 \tag{2}$$
$$x_1 \geq 0 \tag{3}$$
$$x_2 \geq 0 \tag{4}$$



The line $x_1 + x_2 = \frac{1}{5}$ intersects the feasible region, but we can still move the line up. Every two inequalities gives us a vertex of the polygon when we equate them.

So we start with two inequalities and then replace one of them to get to another point to see if we can get better. With a convex polygon, we can just keep getting closer to the optimal answer, not like with non convex polygons, where we might go up and then down and have a local optimum. From 2, 1, in either direction we go we will decrease, so we know that we are at the maximum because the polygon is convex. This is essentially the idea behind the simplex algorithm.

Def A linear program is in standard form if it is in one of the following forms

$$\begin{aligned} \max\ & c_1x_1 + \ldots + c_nx_n \\ \text{s.t. } & a_{11}x_1 + \ldots + a_{1n}x_n \leq b_1 \\ & a_{21}x_1 + \ldots + a_{2n}x_n \leq b_2 \end{aligned}$$

$$\begin{aligned}
&\dots \\
&a_{m1}x_1 + \ldots + a_{mn}x_n \leq b_m \\
&x_1, \ldots, x_n \geq 0
\end{aligned}$$

$$\begin{aligned}
\min\ & c_1x_1 + \ldots + c_nx_n \\
\text{s.t. } & a_{11}x_1 + \ldots + a_{1n}x_n \geq b_1 \\
& a_{21}x_1 + \ldots + a_{2n}x_n \geq b_2 \\
& \dots \\
& a_{m1}x_1 + \ldots + a_{mn}x_n \geq b_m \\
& x_1, \ldots, x_n \geq 0
\end{aligned}$$

Ex

$$\begin{aligned}
\max\ & x_1 + x_2 \\
& x_1 + 2x_2 \leq 1 \\
& 2x_1 + x_2 \leq 1 \\
& x_1, x_2 \geq 0
\end{aligned}$$

is in standard form.

Can we convert every linear program to standard form?

$$\begin{aligned}
\max\ & x_1 + x_2 + 2x_3 & \\
& x_1 + 6x_2 + x_3 \leq 10 & \checkmark \\
& x_1 - x_2 + x_3 \geq 1 & \times \\
& x_1 + 2x_2 - 3_x = -2 & \times \\
& x_1 \geq 0 & \checkmark \\
& x_3 \leq 0 & \times
\end{aligned}$$

We are also missing $x_2 \geq 0$.

How to fix?

$$\max\ x_1 + x_2 + 2x_3$$

$$x_1 + 6x_2 + x_3 \leq 10$$

$$-x_1 + x_2 - x_3 \leq -1 \iff x_1 - x_2 + x_3 \geq 1 \times (-1)$$

$$\begin{cases} x_1 + 2x_2 - x_3 \leq -2 \\ -x_1 - 2x_2 + x_3 \leq 2 \end{cases} \iff x_1 + 2x_2 - 3_x = -2$$

$$x_1 \geq 0$$

$$x_3 \leq 0$$

Use a new variable, $x_3' = -x_3$

$$\begin{aligned} \max\ & x_1 + x_2 - 2x_3' \\ & x_1 + 6x_2 - x_3' \leq 10 \\ & -x_1 + x_2 + x_3' \leq -1 \\ & x_1 + 2x_2 + x_3' \leq -2 \\ & -x_1 - 2x_2 - x_3' \leq 2 \\ & x_1 \geq 0 \\ & x_3' \geq 0 \end{aligned}$$

Finally we introduce two new variables x_2', x_2'' and add the constraints $x_2' \geq 0, x_2'' \geq 0$ and replace all occurrences of x_2 with $(x_2' - x_2'')$.

$$\begin{aligned} \max\ & x_1 + x_2' - x_2'' - 2x_3' \\ & x_1 + 6x_2' - 6x_2'' - x_3' \leq 10 \\ & -x_1 + 6x_2' - 6x_2'' + x_3' \leq -1 \\ & x_1 + 2x_2' - 2x_2'' + x_3' \leq -2 \\ & -x_1 - 2x_2' + 2x_2'' - x_3' \leq 2 \\ & x_1, x_3, x_2', x_2'' \geq 0 \end{aligned}$$

We like the standard form because we can write them in a very efficient, linear algebra way.

$$c = \begin{bmatrix} c_1 \\ \dots \\ c_n \end{bmatrix}$$

$$A = \begin{bmatrix} a_{11} & \dots & a_{1n} \\ \dots & & \\ a_{m1} & \dots & a_{mn} \end{bmatrix}$$
$$b = \begin{bmatrix} b_1 \\ \dots \\ b_m \end{bmatrix}$$
$$x = \begin{bmatrix} x_1 \\ \dots \\ x_m \end{bmatrix}$$

So we want:

$$\begin{aligned} \max\ & c^T x (\text{or } \langle c, x \rangle) \\ & A\vec{x} \leq \vec{b} \\ & \vec{x} \geq 0 \end{aligned}$$

$$\begin{aligned} \min\ & c^T x \\ & A\vec{x} \geq \vec{b} \\ & \vec{x} \geq 0 \end{aligned}$$

11.1 Duality

(Very important concept)
Consider the following LP (in standard form).

$$\begin{array}{lccccl} \max & x_1 + & 2x_2 + & x_3 + & x_4 & \\ & x_1 + & 2x_2 + & x_3 & & \leq 2 \\ & & x_2 + & & x_4 & \leq 1 \\ & x_1 & + & 2x_3 & & \leq 1 \\ & x_1, & x_2, & x_3, & x_4 & \geq 0 \end{array}$$

Suppose the LP solver finds a solution $x_1 = 1, x_2 = \frac{1}{2}, x_3 = 0, x_4 = \frac{1}{2} \implies x_1 + 2x_2 + x_3 + x_4 = \frac{5}{2}$. How can we convince ourselves that this is optimal (without solving the LP by ourselves).

Can we use these constraints to arrive at $x_1 + 2x_2 + x_3 + x_4 \leq \frac{5}{2}$? We can multiply the constraints by positive numbers and add them up.

We can get

$$(y_1 + y_3)x_1 + (2y_1 + y_2)x_2 + (y_1 + 2y_3)x_3 + y_2x_4 \leq 2y_1 + y_2 + y_3$$

We want

$$x_1 + 2x_2 + x_3 + x_4 \underline{\leq} (y_1 + y_3)x_1 + (2y_1 + y_2)x_2 + (y_1 + 2y_3)x_3 + y_2x_4 \leq 2y_1 + y_2 + y_3$$

What do we need to know about y_1, y_2, y_3 to guarantee the first inequality?

We have already assumed that $y_1, y_2, y_3 \geq 0$ (or else they might have flipped the signs of the inequalities that we multiplied by).

We also need

$$\begin{aligned} y_1 + y_3 &\geq 1 \\ 2y_1 + y_2 &\geq 2 \\ y_1 + 2y_3 &\geq 1 \\ y_2 &\geq 1 \end{aligned}$$

If we satisfy all these then we will have the upper bound $2y_1 + y_2 + y_3$. So to get the best upper bound we need to solve

$$\begin{aligned} \min\ 2y_1 + y_2 + y_3 & \\ y_1 + y_3 &\geq 1 \\ 2y_1 + y_2 &\geq 2 \\ y_1 + 2y_3 &\geq 1 \\ y_2 &\geq 1 \\ y_1, y_2, y_3 &\geq 0 \end{aligned}$$

The solution is:

$$\begin{aligned} y_1 &= \frac{1}{2} \\ y_2 &= 1 \\ y_3 &= \frac{1}{2} \end{aligned}$$

$\implies 2y_1 + y_2 + y_3 = \frac{5}{2}$

What is this? A linear program in standard form. So we tried to prove that a linear program in standard form could not be larger than something and we ended up with another linear program in standard form, so these two things are the **dual** of each other.

12 02/19/18

12.1 Duality

$$\begin{array}{lccccccccl} \max & x_1 & + & 2x_2 & + & x_3 & + & x_4 & & \\ & x_1 & + & 2x_2 & + & x_3 & & & \leq 2 \\ & & & x_2 & + & & & x_4 & \leq 1 \\ & x_1 & + & & & 2x_3 & & & \leq 1 \end{array}$$

$$x_4, x_1, x_2, x_3 \geq 0$$

Is $x_1 = 1, x_2 = \frac{1}{2}, x_3 = 0, x_4 = \frac{1}{2}$ optimal? (objective $= \frac{5}{2}$)

We know the solution $\geq \frac{5}{2}$.

We want to show $x_1 + 2x_2 + x_3 + x_4 \leq \frac{5}{2}$if the constraints are satisfied.

We can deduce new constraints, e.g.

$$\left.\begin{array}{cccccl} x_1 & + & 2x_2 & + & x_3 & \leq 2 \\ & & x_2 & + & x_4 & \leq 1 \end{array}\right\} \implies x_1 + 3x_2 + x_3 + x_4 \leq 3$$

e.g.

$$\begin{aligned} x_2 + x_4 &\leq 1 \\ (x_1 + 2x_3 &\leq 1) \times 3 \\ = 3x_1 + x_2 + 6x_3 + x_4 &\leq 4 \end{aligned}$$

$$\begin{aligned} y_1 \times (x_1 + 2x_2 + x_3 \leq 2) \\ y_2 \times (x_2 + x_4 \leq 1) \\ y_3 \times (x_1 + 2x_3 \leq 1) \end{aligned}$$

$$(y_1 + y_3)x_1 + (2y_1 + y_2)x_2 + (y_1 + 2y_3)x_3 + y_2x_4 \leq 2y_1 + y_2 + y_3$$

Provided that $y_1, y_2, y_3 \geq 0$

We want to show

$$x_1 + 2x_2 + x_3 + x_4 \leq \frac{5}{2}$$

If we find $y_1, y_2, y_3 \geq 0$ so that

$$\begin{aligned} y_1 + y_3 &\geq 1 \\ 2y_1 + y_2 &\geq 2 \\ y_1 + 2y_3 &\geq 1 \\ y_2 &\geq 1 \\ \text{and } 2y_1 + y_2 + y_3 &= \frac{5}{2} \implies \text{done!} \end{aligned}$$

In that case

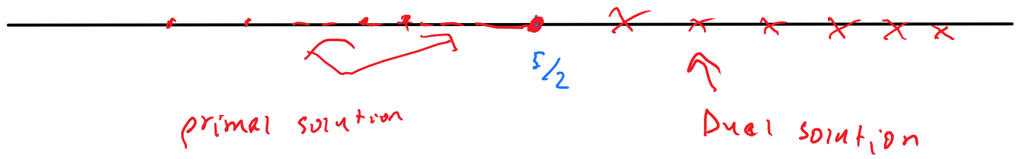
$$\begin{aligned} x_1 + 2x_2 + x_3 + x_4 &\leq (y_1 + y_3)x_1 + (2y_1 + y_2)x_2 + (y_1 + 2y_3)x_3 + y_2 x_4 \\ &\leq 2y_1 + y_2 + y_3 \end{aligned}$$

The best upper-bound we can get here is

$$\begin{aligned} \min\ 2y_1 + y_2 + y_3 \text{ vs} \\ s.t.\ y_1 + y_3 \geq 1 \\ 2y_1 + y_2 \geq 2 \\ y_1 + 2y_3 \geq 1 \\ y_2 \geq 1 \\ y_1, y_2, y_3 \geq 0 \end{aligned} \qquad \begin{aligned} \max\ x_1 + 2x_2 + x_3 + x_4 \\ s.t.\ x_1 + 2x_2 + x_3 \leq 2 \\ x_2 + x_4 \leq 1 \\ x_1 + 2x_3 \leq 1 \\ x_1, x_2, x_3, x_4 \end{aligned}$$

We showed $opt(Primal\ LP) \leq opt(Dual\ LP)$.

$y_1 = \frac{1}{2}, y_2 = 1, y_3 = \frac{1}{2} \implies Opt(Dual\ LP) \leq \frac{5}{2}$



Rem If we find solutions with exact same value for primal and dual then the value is optimal for both of them.

Dual for standard LP's

$$\begin{aligned} &\max\ c_1x_1 + \ldots + c_nx_n \\ s.t.\ y_1 \times (\ &a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n \leq b_1) \\ y_2 \times (&a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n \leq b_2) \\ &\ldots \\ y_m \times (&a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n \leq b_m) \\ &x_1, x_2, \ldots, x_n \geq 0 \end{aligned}$$

$$\begin{aligned} &\min\ b_1y_1 + b_2y_2 + \ldots + b_my_m \\ x_1 \to &\underbrace{a_{11}y_1 + a_{21}y_2 + \ldots + a_{m1}y_m}_{\text{coeff of } x_1 \text{ in LHS}} \leq b_1 \\ x_2 \to &a_{12}y_1 + a_{22}y_2 + \ldots + a_{m2}y_m \leq b_2 \\ &\ldots \\ x_n \to &a_{1n}y_1 + a_{2n}y_2 + \ldots + a_{mn}y_m \leq b_n \end{aligned}$$

$$\begin{array}{rr} \max\ c_1x_1 + \ldots + c_nx_n & \min\ b_1y_1 + \ldots + b_my_m \\ a_{11}x_1 + \ldots + a_{n1}x_n \leq b_1 & a_{11}y_1 + \ldots + a_{m1}y_m \geq c_1 \\ \ldots & \ldots \\ a_{1m}x_1 + \ldots + a_{nm}x_n \leq b_1 & a_{1n}y_1 + \ldots + a_{mn}y_m \geq c_1 \\ x_1, \ldots, x_n \geq 0 & y_1, \ldots, y_m \geq 0 \end{array}$$

Remark: value of every feasible solution to primal $\leq$ value of every feasible solution to dual.

Q: If primal is unbounded $\implies$ Dual is infeasible and vice versa.

Thm (Weak Duality)

1. If primal is unbounded $\implies$ Dual is infeasible.
2. If Dual is unbounded $\implies$ primal is infeasible.

3. If both primal and dual are feasible and bounded $\implies$

$$Opt(\underbrace{Primal}_{\text{max}}) \leq Opt(\underbrace{Dual}_{\text{min}})$$

Similar to

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

$$\begin{aligned}
\max\ & x_1 + 5x_2 - x_3 \\
& 3x_1 + x_2 \leq 1 \leftarrow y_1 \\
& 4x_2 - x_3 \leq 5 \leftarrow y_2 \\
& x_1, x_2, x_3 \geq 0
\end{aligned}$$

Dual:

$$\begin{aligned}
\min\ & y_1 + 5y_2 \\
& 3y_1 \geq 1 \\
& y_1 + 4y_2 \geq 5 \\
& -y_2 \geq -1 \\
& y_1, y_2 \geq 0
\end{aligned}$$

Writing the dual without converting to standard form.

$$\begin{array}{rcl}
\max\ x_1 + 2x_2 + 3x_3 & & \min\ y_1 + 4y_2 + 3y_3 \\
s.t.\ y_1(x_1 - x_2 - x_3 \leq 1) & \Leftrightarrow & y_1 \geq 0 \\
y_2(5x_1 + x_2 + 2x_3 \geq 0) & \Leftrightarrow & y_2 \leq 0 \\
y_3(3x_1 + 2x_2 - x_3 = 3) & \Leftrightarrow & y_3 \text{ free} \\
x_1 \geq 0 & \Leftrightarrow & y_1 + 5y_2 + 3y_3 \geq 1 \\
x_2 \leq 0 & \Leftrightarrow & -y_1 + y_2 + 2y_3 \leq 2 \\
x_3 \text{ free} & \Leftrightarrow & -y_1 + 2y_2 - y_3 = 3
\end{array}$$

So we have:
standard $\Leftrightarrow$ positive
nonstandard $\Leftrightarrow$ negative
equality $\Leftrightarrow$ free

Thm (Strong Duality) : If Primal and dual are both feasible

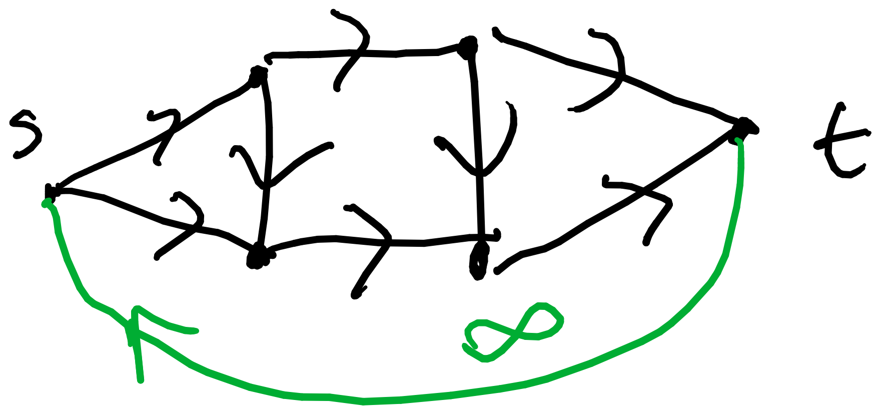
$$\implies Opt(Primal) = Opt(Dual)$$

13 02/21/18

13.1 Max-flow and duality

$$\max \sum_{su \in E} f_{su} \leftarrow (f^{out}(s))$$
$$\text{st } f_{uv} \leq c_{uv}, \forall uv \in E$$
$$\sum_{vu \in E} f_{vu} - \sum_{uw \in E} f_{uw} = 0, \forall u \in V - \{s,t\}$$
$$f_{uv} \geq 0, \forall uv \in E$$

Note that this linear program is written in an ugly way, we want to write it in a clearer way such that the dual will be easier to understand. So we add an edge with infinite capacity from t to s such that we can treat all vertices the same way:



$$\max f_{ts}$$
$$f_{uv} \leq c_{uv}, \forall uv \in E \ (x_{uv}, \text{capacity})$$
$$\sum_{vu \in E'} f_{vu} - \sum_{uw \in E'} f_{uw} = 0, \forall u \in V \ (y_u, \text{conservation})$$

$$f_{uv} \geq 0, \forall uv \in E'$$

, where E' consists of the edges E and the newly added ts edge.
Dual: Vars: $x_{uv} \forall uv \in E, y_u, \forall u \in V$

$$\begin{aligned}
&\min \sum_{uv \in E} c_{uv} x_{uv} \\
&y_s - y_t \geq 1, (\text{constraints for } f_{ts}) \\
&x_{uv} + y_v - y_u \geq 0, \forall uv \in E (\text{constraints for} f_{uv}) \\
&x_{uv} \geq 0, \forall uv \in E \\
&y_u \text{ free } \forall u
\end{aligned}$$

Now what does this tell us?

Let (A, B) be an s-t-cut. Consider the solution:

$$x_{uv} = \begin{cases} 1 & u \in A, v \in B \\ 0 & \text{otherwise} \end{cases}$$

$$y_s = 1, y_t = 0, y_u = \begin{cases} 1 & u \in A \\ 0 & u \in B \end{cases}$$

$$x_{uv} + y_v - y_u \geq 0, \forall uv \in E?$$

This shows $Opt(Dual) \leq Min\text{-}cut$

By strong duality

$$Max\text{-}flow = Opt(Dual) \leq Min\text{-}cut$$

13.2 Complementary Slackness

$$\begin{aligned}
\max c_1 x_1 + \ldots + c_n x_n \\
\text{s.t. } a_{11} x_1 + \ldots + a_{1n} x_n \leq b_1 \ (y_1) \\
\ldots\ldots \\
a_{m1} x_1 + \ldots + a_{mn} x_n \leq b_m \ (y_m) \\
x_1, \ldots, x_n \geq 0
\end{aligned}$$

$$\begin{aligned} \min\ & b_1y_1 + \ldots + b_ny_n \\ \text{s.t.}\ & a_{11}y_1 + \ldots + a_{1m}y_m \leq b_1\ (x_1) \\ & \ldots\ldots \\ & a_{m1}y_1 + \ldots + a_{mn}y_m \leq b_m\ (x_n) \\ & y_1, \ldots, y_m \geq 0 \end{aligned}$$

Let $(x_1^*, \ldots, x_n^*), (y_1^*, \ldots, y_m^*)$ be optimal solutions to primal and dual respectively.

$$Opt(Primal) = Cost(x_1^*, \ldots, x_n^*) = Cost(y_1^*, \ldots, y_m^*) = Opt(Dual)$$

Suppose $a_{11}x_1^* + \ldots + a_{1n}x_n^* < b_1$. What do we know about y_i^*? It is equal to 0 (that way we managed to turn inequalities into equalities), because:

Say we have:

$$\begin{aligned} (x_1^* + x_2^* < 5)y_1^* \\ (5x_1^* - x_2^* = 4)y_2^* \end{aligned}$$

Add them up:

$$\text{objective function} \overset{?}{=} 5y_1^* + 4y_2^*$$

This is only possible if y_1^* is 0. This is the complementary slackness theorem.

Complementary Slackness Theorem
If $y_i^* > 0 \implies a_{i1}x_1^* + \ldots + a_{in}x_n^* = b_i$
If $x_j^* > 0 \implies a_{11}y_1^* + \ldots + a_{mj}y_m^* = c_i$

$$\begin{aligned} \max 2x_1 + 4x_2 + 3x_3 + x_4 & \\ 3x_1 + x_2 + x_3 + 4x_4 &\leq 12 \\ x_1 - 3x_2 + 2x_3 + 3x_4 &\leq 7 \\ 2x_1 + x_2 + 3x_3 - x_4 &\leq 10 \\ x_1, x_2, x_3, x_4 &\geq 0 \end{aligned}$$

$$\begin{aligned}
\min 12y_1 + 7y_2 + 10y_3 \\
3y_1 + y_2 + 2y_3 \geq 2 \\
y_1 - 3y_2 + y_3 \geq 4 \\
y_1 + 2y_2 + 3y_3 \geq 3 \\
4y_1 + 3y_2 - y_3 \geq 1 \\
y_1, y_2, y_3 \geq 0
\end{aligned}$$

Show that $x_1^* = 0, x_2^* = 10.4, x_3^* = 0, x_4^* = 0.4$ is an optimal solution to primal.

$$Cost = 4 \times 10.4 + 0.4 = 42$$

We have slack in 2nd constraint

$$\begin{aligned}
x_1^* - 3x_2^* + 2x_3^* + 3x_4^* = -30 < 7 \\
\implies y_2^* = 0
\end{aligned}$$

$$\left.\begin{aligned} y_1^* - 3y_2^* + y_3^* = 4 \\ 4y_1^* + 3y_2^* - y_3^* = 1 \end{aligned}\right\} \implies y_1^* = 1, y_3^* = 3$$

$$12y_1^* + 7y_2^* + 10y_3^* = 42$$

14 03/12/18

14.1 NP-Completeness and Computational Interactability

All the courses leading up to and including this have been focusing on making efficient algorithms. Now we will focus on what we cannot do.

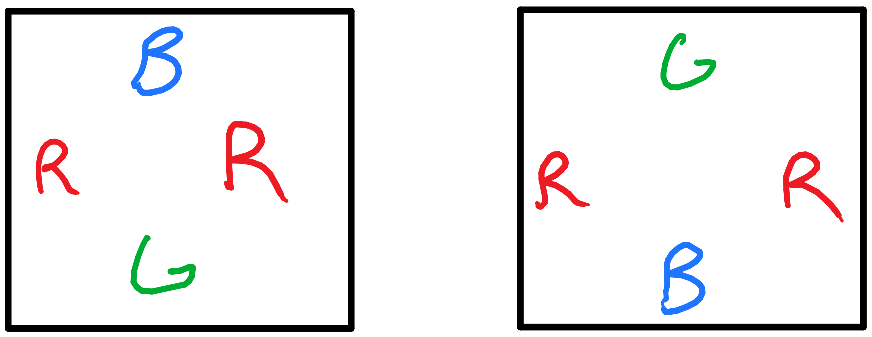
- So far we designed efficient algorithms for many problems.
- There are problems that cannot be solved by any algorithm (even if we don't care about running time, undecidable problems). Hilbert (asked this question), Gödel, Church, Turing 1930's, see COMP 330 for more information. This was studied before computers even existed.

Ex: Halting Problem: Given a code with an input, we want to decide whether it eventually terminates (impossible). If something terminates, you can run it and just wait for it to finish.

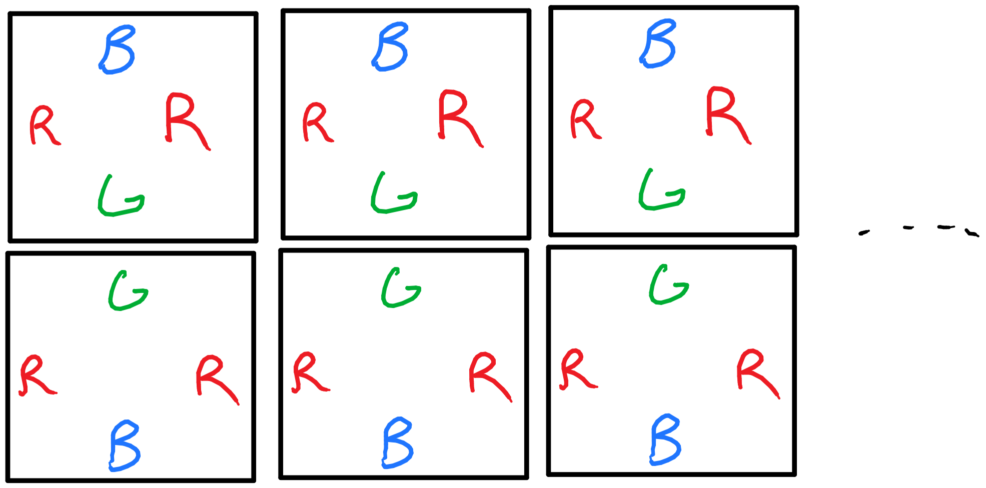
But if something doesn't terminate, there's no definite time you can run it for to know that it doesn't terminate.

Ex: We are given 10 types of tiles, each with four colors on its edges, and $1m \times 1m$. Can we tile the whole plane such that the color of the neighboring tiles match on the bounding edge?

Ex:



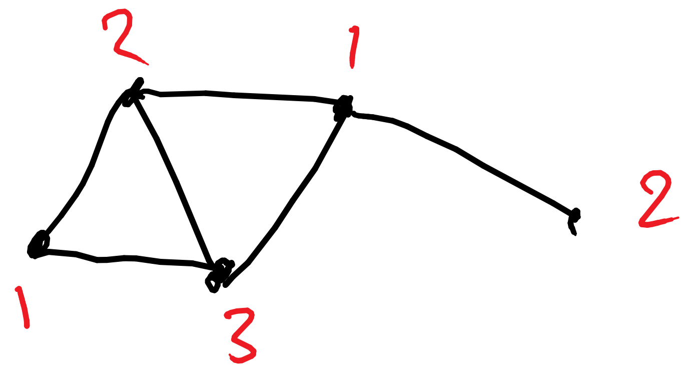
This can be solved by placing them like:



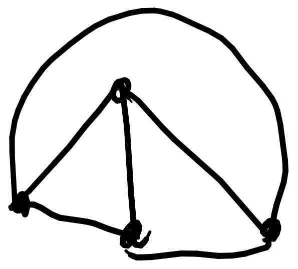
So we can keep trying and find a nice pattern, if one exists. But when do we stop and say there isn't one? This is not possible.

- There are also problems that can be solved using computers. Computational complexity studies the running time required for solving these problems.
- Are there problems for which we cannot do significantly better than brute-force search?

Ex: 3 colorability of a graph:
Input: An undirected graph $G = (V, E)$
Goal: Can we color the vertices of G with 3 colors such that neighboring vertices receive different colors?



Yes



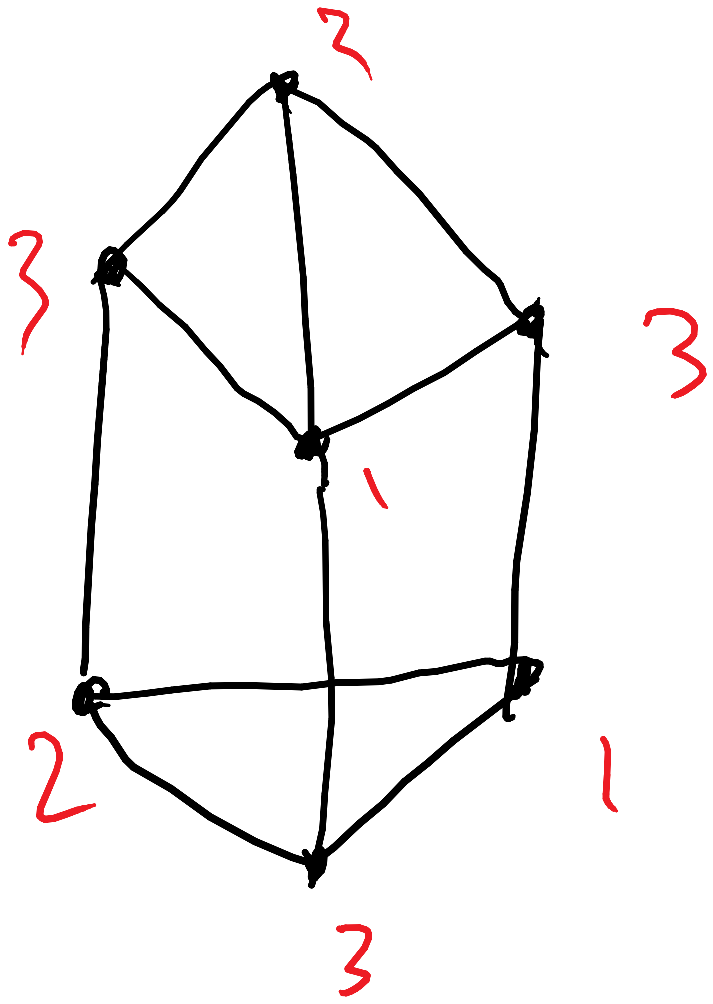
No

Brute-force algorithm: Check all the 3^n different colorings where $n = |V|$. Running time $O(3^n m)$ where $m = |E|$.

If we were checking for 2 colorability, we could do better than brute force and just color as we go.

It seems that we cannot do anything significantly better. Are there any efficient algorithms for this? It seems not.

On the other hand if someone provides us with a potential proper 3-coloring, we can check its validity in polynomial time.



Are there problems for which verifying the correctness of a solution (NP, will be formally defined later) is significantly easier than solving them? Of course, some things like grading an assignment seem easier than completing an assignment and checking the validity of a sudoku solution is easier than solving it. So it seems to be true.

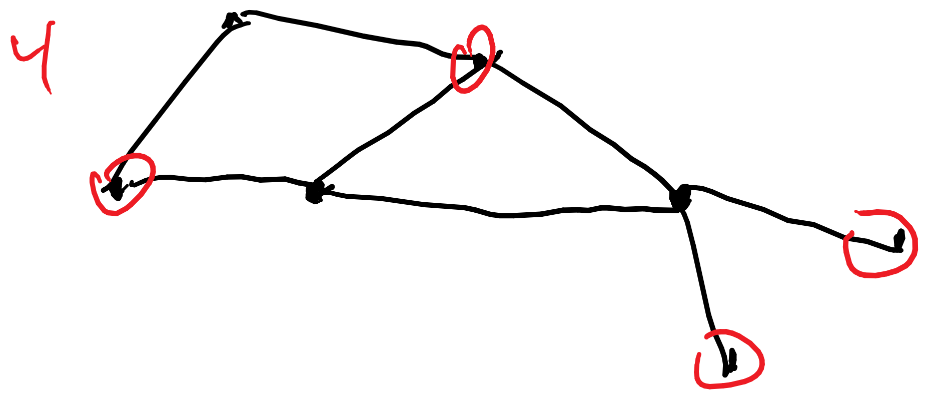
3-colorability seems to be on of these problems.

This is essentially the P vs NP problem (are these two types of problems the same? Most important problem in computer science).

We will focus on yes/no problems (decision problems, like 3 colorability, not things like *max-flow*). Problems with yes/no answer.

We can convert other problems to decision problems without making them easier.

Ex: Max Independence Problem: Given an undirected graph $G = (V, E)$ what is the size of the largest set of vertices no two which are adjacent (independent set).



Decision version: Input: $G, k \in \mathbb{N}$.

Q: Does G have an independent set of size $\geq k$?

If we can solve this efficiently then we can use a "for loop" on k to solve the original problem efficiently.

We are going to focus on decision problems.

We call an algorithm **efficient** if its running time is $O(n^c)$ for some constant c where $\underline{n}$ is the number of bits required to represent the input.

Ex: Primality test

Input: m

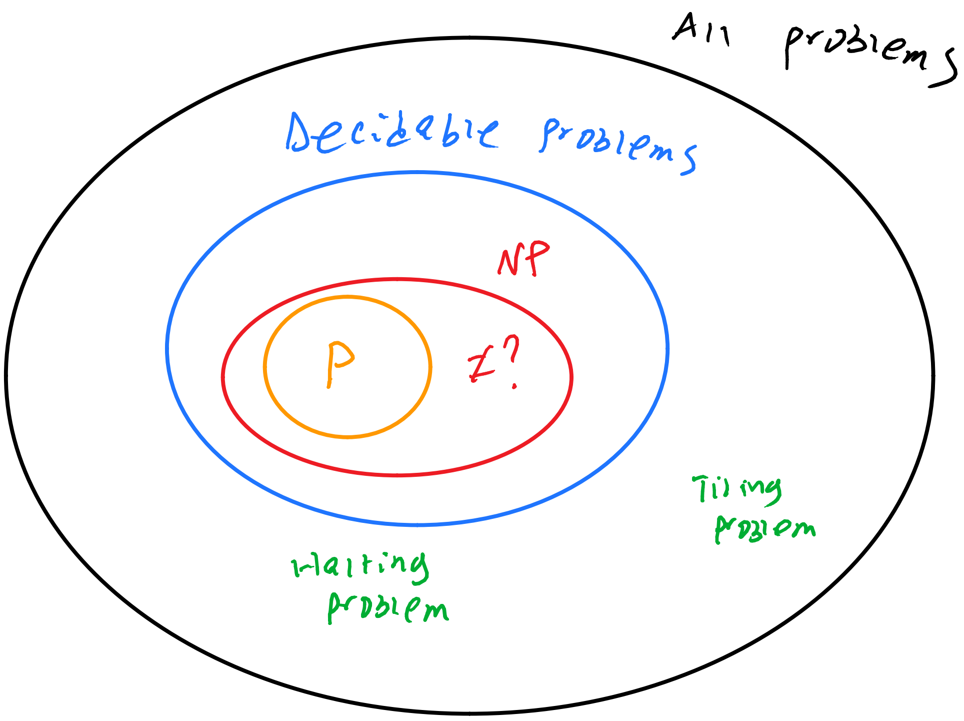
```
Alg.
for $i = 2, \ldots, n-1$ do
    if $i|m$ then output "No"
        terminate
    end if
end for
output "Yes"
```

Running time: $O(m)$ essentially. But is this efficient? No. Input size: $n = \lceil \log_2 m \rceil$. Running time: $\Theta(2^n)$. This same issue came up when we were looking at Ford Fulkerson and the efficient version of it. **Not efficient**.

Def: P is the class of all decision problems that can be solved efficiently (Polynomial time)

Ex:
Input: $G = (V, E)$ undirected
Q: Is G connected?
Belongs to P.

Ex:
Input: Flow network G and a number k.
Q: Is $max\text{-}flow \geq k$?
Belongs to P. (Can be solved by scaling Ford Fulkerson)



Def: An efficient certifier for a problem X is an algorithm that takes as input $\langle W, t\rangle$ where w is an input for X and t is a "potential solution" (certificate, can be regarded as a hint to the problem).

1. It runs in polynomial time $O(n^c)$ where $n = |w|$.
2. If w is a NO input then for all t the algorithm outputs NO.
3. If w is a YES input then $\exists\ t$ for which the algorithm outputs YES.

Ex: For 3-colorability the efficient certifier takes $\langle G, c\rangle$ where G is the original graph and c a potential coloring. For every edge in G it checks whether c assigns different colors to endpoints, if Yes outputs Yes else outputs NO.

Running time $O(m)$ where m is the number of edges. Note if G is not 3-colorable $\implies$ for all c we output NO. On the other hand for every Yes input (3-colorable graph) there is a coloring c for which $\langle G, c\rangle$ will be accepted (The certifier outputs Yes).

NP (non-deterministic polynomial) is the set of problems with efficient certifiers.

Ex: 3-colorability is in NP as we just saw.

Q: Is $P \neq NP$?

Thm: $P \subseteq NP$.

<u>Proof</u>: Consider a problem X in P and let A be an efficient algorithm that solves A. Consider the following efficient certifier:

- On input $\langle w, t\rangle$: Ignore t and run A on w and if it outputs Yes $\implies$ output Yes, No $\implies$ output NO.

  Running time: Same as $A \implies$ efficient.

15 03/14/18

Recall:

- Focus on decision (yes/no) problems. We saw that we can convert optimization problems to decision problems, just add an extra input to see if optimal value is greater than or equal to some parameter k and we can just search over all those values.
- Solving a problem vs verifying a solution (the key of P vs NP).

Def: An efficient certifier for a problem X is a polynomial time algorithm that takes as input $\langle w, t\rangle$ (w is an instance of the original problem we want to solve, t is the solution, a certificate or a hint). We are slightly modifying the definition from last lecture.

1. $|t| \leq O(|w|^c)$ where c is a fixed constant. i.e. the solution we want to verify has to be polynomial (we did not mention this last lecture, but we said the algorithm has to be polynomial, so essentially the same).
2. w is a Yes input $\iff \exists t$ for which our certifier accepts $\langle w, t\rangle$ (i.e. if w is a No input we just negate this, i.e. $\forall t$ the certifier does not accept)

Def: NP is the set of all problems that have efficient certifiers.

Def: P is the set of problems that can be solved in polynomial time.

Thm: $P \subseteq NP$ (proved last lecture, just ignore t). Is $P \neq NP$? (most important question in computer science)

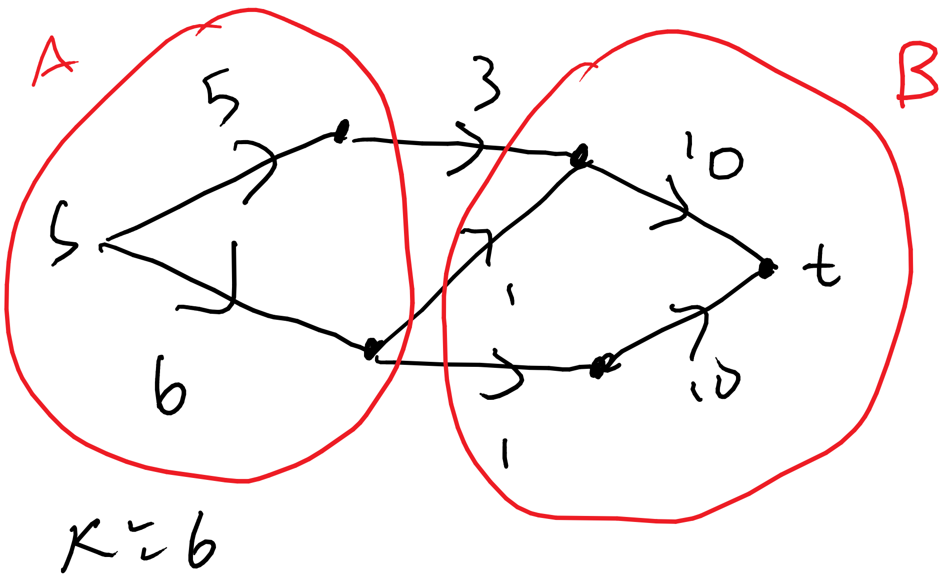
Max-flow: (G, k) input, where G is a flow network. Is $Max\text{-}flow$ of $G \geq k$?

$\overline{Max\text{-}flow}$: is $Max\text{-}flow < k$? (negation)

Problem: Without using the fact that $Max\text{-}flow \in P$ prove that both these problems are in NP (we want to design an efficient certifier for the problems).

An efficient certifier for $max\text{-}flow$: $\langle (G, k), f\rangle$ where f is a flow. It verifies that f is a valid flow and its value is $\geq k$. It accepts if these conditions are satisfied.

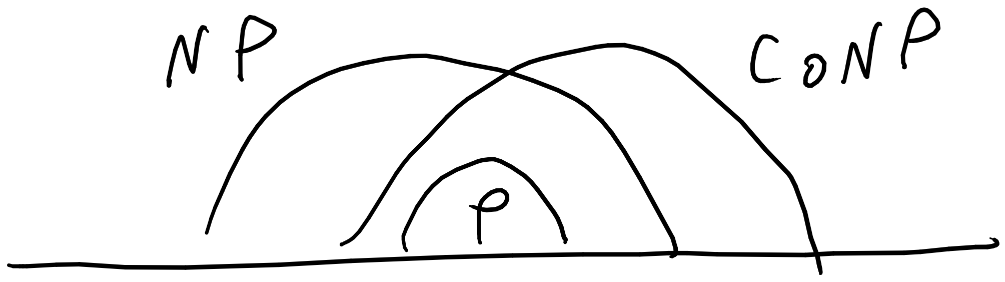
An efficient certifier for $\overline{Max\text{-}flow}$ (complement). $\langle (G, k), (A, B)\rangle$.
It accepts $\iff cap(A, B) < k$.



Could ask the same 2 questions for linear programming, both of which are also in NP. If a problem is in NP, it does not mean its complement is in NP.

CoNP: Is the set of problems whose complements are in NP.

Thm: $P \subseteq CoNP$.
Conjectured picture (this is how people believe things are):



Def: EXP is the set of problems that can be solved in exponential time $O(2^{n^c})$ for some constant c.

Ex: 3-col is in EXP.

Algorithm: Generate all possible 3-coloring and see if any of them is proper.

$$O(3^n n^2) \leq O(2^{2n})$$

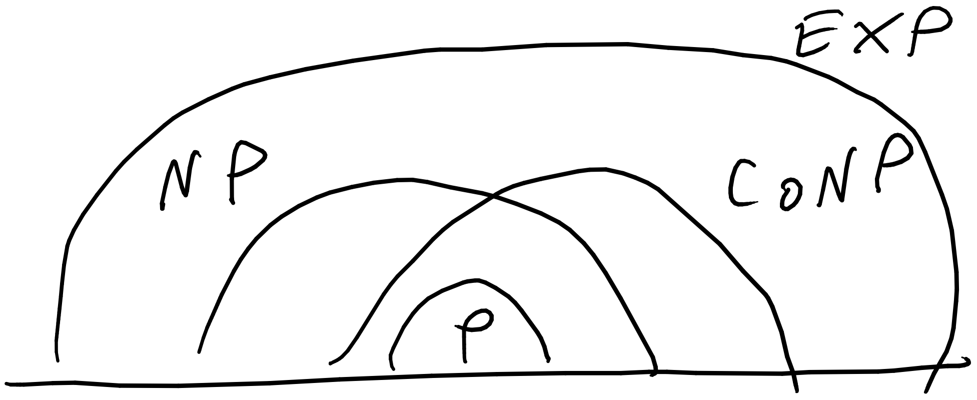
Obviously $P \subseteq EXP$.

Thm: $NP \subseteq EXP$.

Proof: Let X be a problem in NP. Then there is an efficient certifier $\mathcal{A}$ that takes $\langle w, t\rangle$ and w is a Yes input $\iff \exists t$ such that $\mathcal{A}$ accepts $\langle w, t\rangle$. How do we change this into an exponential time algorithm without a certifier? Just check every t (brute force).

Let $\mathcal{B}$ be the following algorithm: generate all t up to size $O(|w|^c)$ (we put a bound on the size of t earlier in the lecture). We run $\mathcal{A}$ on $\langle w, t\rangle$ and if any of them is accepted $\implies$ output "Yes", else "No".

Another interpretation of the P vs NP question: Does having a brute force algorithm imply there is an efficient algorithm?

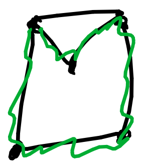
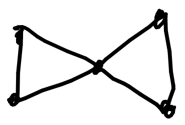


Polynomial Time Reductions: Can instances of a problem X be solved using a blackbox that solves problem Y?

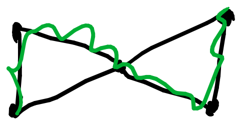
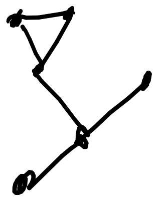
We say that X is polynomial-time reducible to Y if there is an efficient "oracle" algorithm that solves X in polynomial time using a blackbox ("oracle") that solves whether y is a Yes input for Y or not.

We write $X \leq_p Y$, i.e. X is easier to solve than Y. We've seen this earlier in the course, for example, solving the baseball elimination problem using *Max-flow* or using linear programming to solve *Max-flow*.

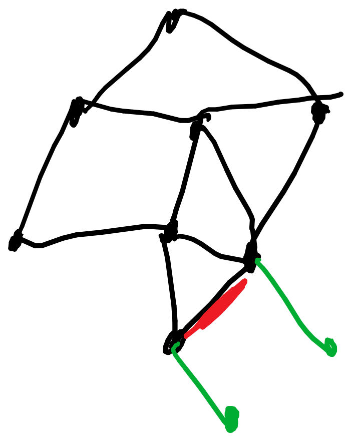
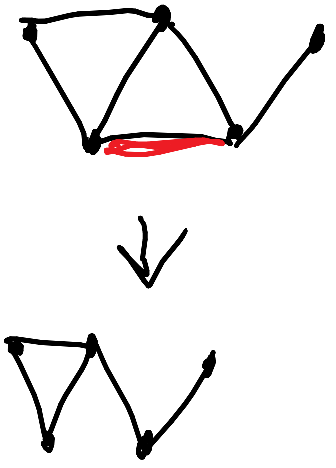
Ex: Hamiltonian Cycle
Input: Undirected graph
Q: Does G have a cycle that visits all vertices?

Yes   No

Hamiltonian Problem
Input: G undirected
Q: Is there a path that visits all the vertices?

Yes   No

Show Hamiltonian Cycle $\leq_P$Hamiltonian Path (note that this means we are reducing Hamiltonian path to Hamiltonian cycle, not the other way around, common mistake on exams and assignments).

If we remove the edges in green, just because there is a Hamiltonian path does not imply there is a Hamiltonian cycle with the removed edge, because it may not go through there. To solve this, we add dangling edges in green. With these dangling edges the only possibility of a Hamiltonian path is a path that starts and ends in the new vertices.

```
On an input $G$ for Hamiltonian cycle:
for every edge $xy$ in $G$ do
    remove $xy$, add dangling edges to $x$ and $y$ and call this $H_{xy}$
    if $H_{xy}$ $\underbrace{\text{has a Hamiltonian path}}_{(oracle)}$ then output Yes; terminate
    end if
end for
output "No"
```

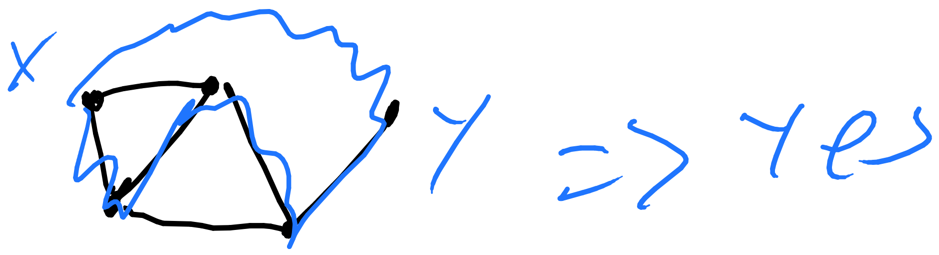
So if Hamiltonian Path $\in P \implies$ Hamiltonian Cycle $\in P$.

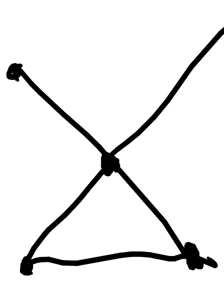
Thm: If $X \leq_P Y$ and $Y \in p \implies X \in P$.

Pf: Take the "oracle" alg that solves X using Y and replace the oracle with an efficient algorithm for Y. This will give us an efficient algorithm for X.

Ex: Hamiltonian Path $\leq_P$ Hamiltonian Cycle.

```
for every pair of vertices $x, y$ do
    add the edge $xy$ (if it doesn't exist)
    if this graph has a Hamiltonian cycle then $\implies$ Yes; terminate
    end if
end for
No
```

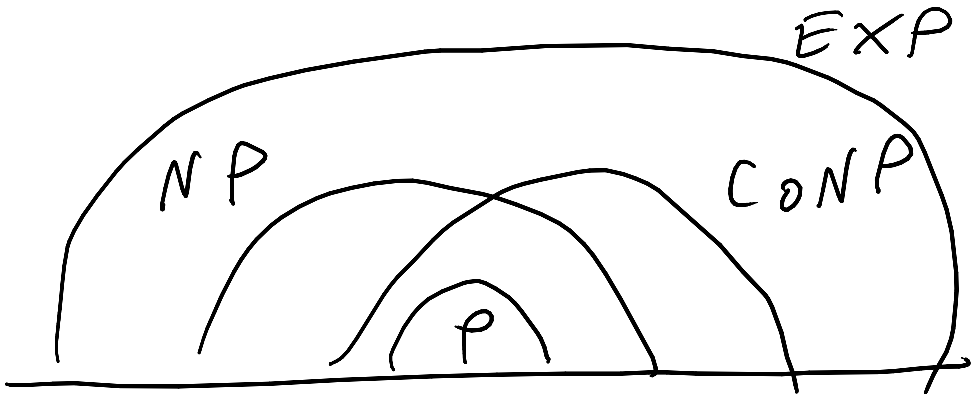




16 03/19/18

Recall:

- P polynomial time solvable (easy problems for us, we have an efficient algorithm for them)
- NP efficient certifiers (if it's a yes input, there's an easy way to convince someone that it is a yes input, easy to verify given a certificate)
- $CoNP$ (No inputs are easy to verify, complement of things in NP)
- EXP exponential time solvable (still slow, but at least there's a bound)



This is interesting, because we see that $NP \to$ Brute force (exponentials) $\implies NP \subseteq EXP$, same for $CoNP$

Polynomial reductions: $X \leq_P Y$ if X can be solved efficiently (polynomial time) using an oracle for Y.

Example: X is the following problem:
Input: Undirected G
Q: Does G have a Hamiltonian Cycle?
Problem Y:
Input: Undirected G, $k \in \mathbb{N}$
Q: Does G have a cycle of length k?

Which of these problems is harder? Y, since it takes in an extra parameter k and can be used to solve X by feeding the number of vertices of G into Y.

$X \leq_P Y$: Given an input G for problem X, then we can set $k = |V(G)|$ and then use the oracle for Y to see if G has a Hamiltonian Cycle.

How about $Y \leq_P X$? True, but more complicated, won't go through this.

Example: Let Z:
Input: G
Q: Does G have a cycle of prime length?
$Z \leq_P Y$: Given an input G to Z consider the following efficient oracle algorithm:

```
for $k = 1, \ldots, |V(G)|$ do
    if $k$ is a prime number then
        if $(G, k)$ is a Yes input for $Y$ then Output Yes and stop
        end if
    end if
end for
Output "NO".
```

16.1 SAT Problem

Def: Suppose $x_1, \ldots, x_n$ are Boolean variables (True/False vars).

- A term (aka literal) is a variable or its negation (x_i or $\underbrace{\overline{x_i}}_{\text{or } \neg x_i}$).
- A clause (or clause) is an OR of a few terms

$$C = (t_1 \lor t_2 \lor t_3 \lor \ldots \lor t_\ell)$$

  where $t_1, \ldots, t_\ell$ are terms.

- A conjunctive normal form (CNF) is an AND of clauses $C_1 \land C_2 \land \ldots \land C_m$

Example of a CNF:

$$\underbrace{(x_1 \lor x_2 \lor \overline{x_3})}_{C_1} \land \underbrace{(\overline{x_2} \lor x_4)}_{C_2} \land \underbrace{(\overline{x_4})}_{C_3}$$

Notation: $\lor \iff$ OR, $\land \iff$ AND

SAT Problem:
Input: A CNF ϕ
Q: Is it possible to assign T/F values to variables such that ϕ becomes True? In other words is ϕ satisfiable?

In the above example,$x_1 = x_3 = T, x_2 = x_4 = F$ to satisfy ϕ, this is a Yes input.

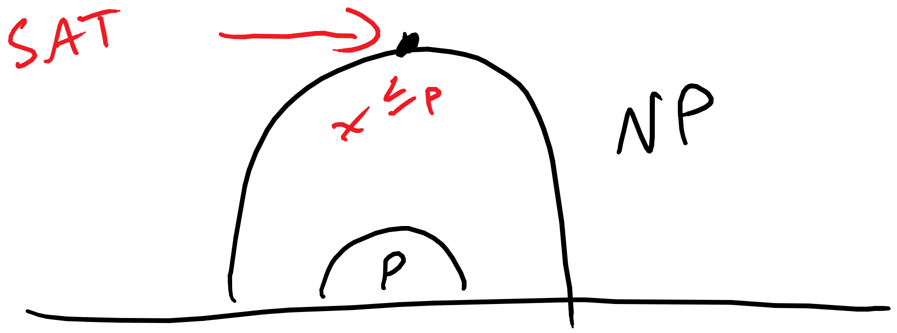
Thm: $SAT \in NP$

Proof: The efficient certifier takes a truth assignment to the variables and verifies whether it satisfies all the clauses (can be done in polytime).

Thm: (Cook-Levin 71) Every problem X in NP can be polynomialy reduced to SAT (if you can solve SAT then you can solve anything in NP).

$$X \leq_P SAT$$

Corollary: If $SAT \in P \implies$ every problem $X \in NP$ belongs to $P \implies P = NP$.
If $SAT \notin P \implies P \neq NP$ (example of a problem that is in NP but not in P)
So, thanks to the Cook-Levin Theorem, the P vs NP problem is equivalent to "Is $SAT \in P$?"



i.e. SAT is the hardest problem in NP.
Is SAT the only such problem?

Def: (NP-Complete) A problem Y is called NP-Complete if

(i) $Y \in NP$

(ii) $X \leq_P Y$ for all $X \in NP$ (Completeness)

SAT is NP-Complete (do not confuse this with being NP, they are two different things).

How can we show a problem Z is NP-Complete? First we show $Z \in NP$ by giving an efficient certifier (if this is asked on an exam this is usually about 20% of the question, **don't forget it**). Then we reduce SAT (or any other NP-Complete problem) to Z.

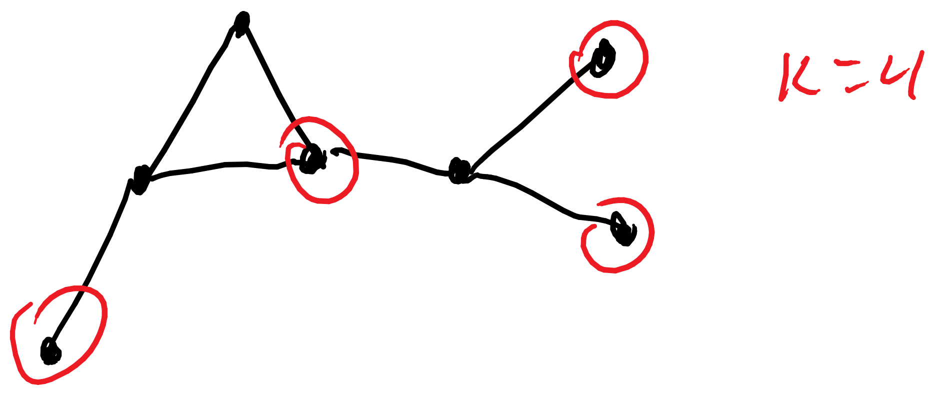
$$X \leq_P SAT \leq_P Z$$

since being polynomial reducible is a transitive relationship.

This approach will always work, because if something is NP-Complete, then it must be polynomial reducible to SAT, although this approach might not be the best approach at times, might be overly complicated.

Independent Set Problem
Input: G undirected, $k \in \mathbb{N}$
Q: Does G have an independent set (no edges between these vertices) of size k?

Ex:



Thm: IND is NP-Complete.

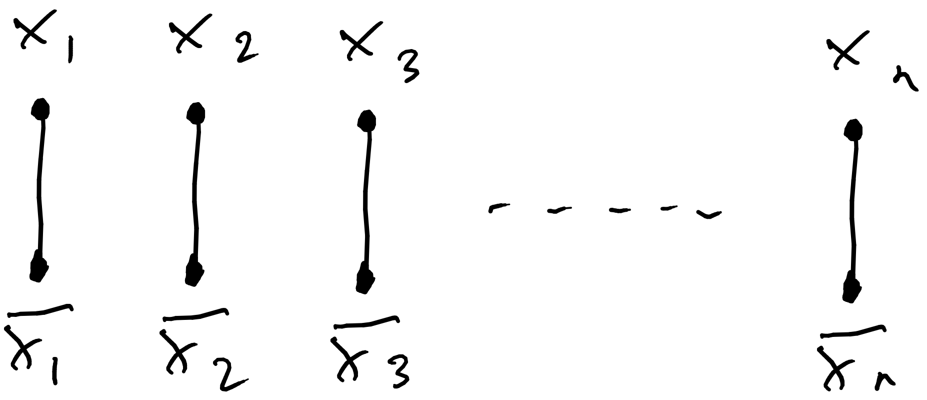
Pf: It is in NP. For a Yes instance we can give an independent set of size k and it can be easily verified.

To prove completeness $SAT \leq_P IND$ (don't do the reduction in the opposite direction, common mistake, it'll tell us nothing new).

Let ϕ be an input to SAT with variables $x_1, \dots, x_n$.

Construct a graph G_ϕ in the following manner:

1. Start with n isolated edges each between a variable x_i and its negation $\overline{x_i}$.

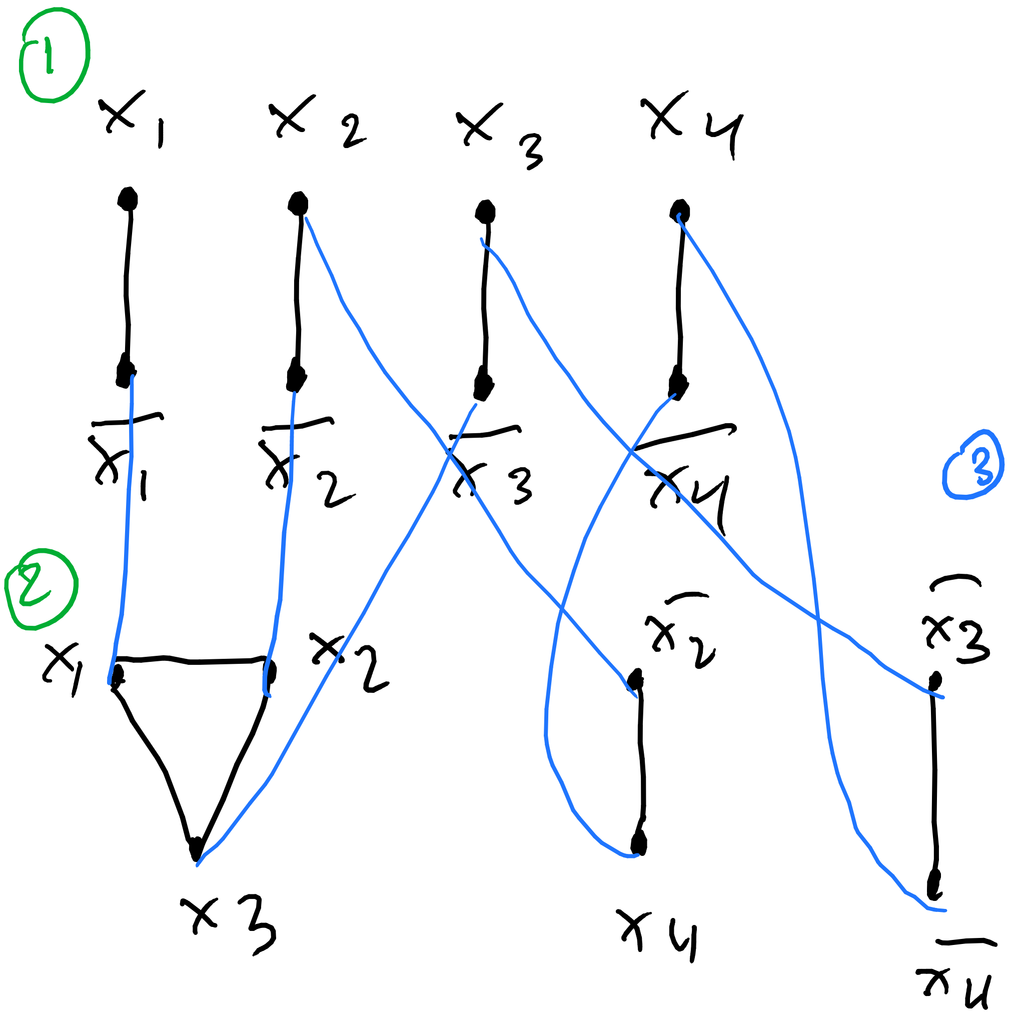


The largest independent set here is of size n. Notice that each vertex we decide to include gives us a truth assignment (which one to assign true).
(so a max independent set selects exactly one of x_i or $\overline{x_i}$ for each i and can be interpreted as a truth assignment)

```
2.  for each clause $C_i = (t_{i1} \lor t_{i2} \lor \ldots \lor t_{ir})$ do
        put $r$ new vertices in the graph and join all these $r$ vertices together.
    end for
3.  for every term in each clause do
        add an edge between that and the opposite term in the set of vertices introduced in part 1.
    end for
```

This assures that the truth assignment from the first part complies with the second part.

Ex: $\phi = (x_1 \lor x_2 \lor x_3) \land (\overline{x_2} \lor x_4) \land (\overline{x_3} \lor \overline{x_4})$



Claim: G_ϕ has an independent set of size $n + m \iff \phi$ is satisfiable (m is the number of clauses).

17 03/21/18

17.1 Np-Completeness

There are problems in NP that every other problem is reducible to them

$$X \leq_P Y, \forall X \in NP$$

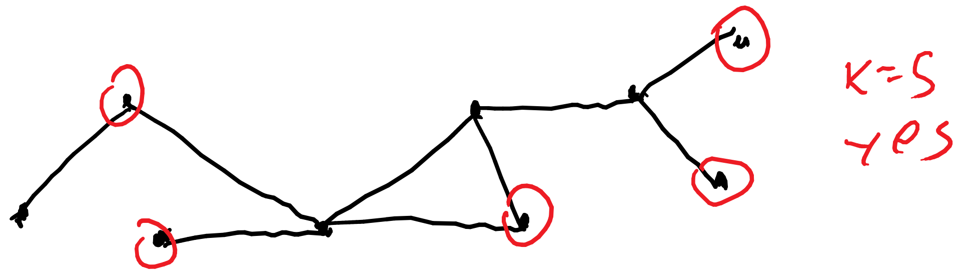
1. $Y \in NP$
2. $\forall X \in NP, X \leq_P Y$

If these two conditions are satisfied, then Y is called NP-Complete. If you find a polynomial algorithm for a problem that is NP-Complete, then you'll find a polynomial algorithm for all problems in NP, i.e. you'll show that $P = NP$.

Thm (Cook-Levin 71) SAT is NP-Complete.

Ex. $\phi = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x}_2 \vee x_4) \wedge (\overline{x}_3 \vee \overline{x}_4)$ (This is called a CNF, $\wedge$ of $\vee$'s)
$\vee$ OR, $\wedge$ AND
SAT: Can the formula be satisfied? For the above example, we can set $x_2 = x_4 = F, x_1 = x_3 = T$.
We can say $P = NP \iff SAT \in P$
Are there other NP-Complete problems? Note if $Z \in NP$ and $(X \leq_P)SAT \leq_P Z \implies Z$ is NP-Complete.

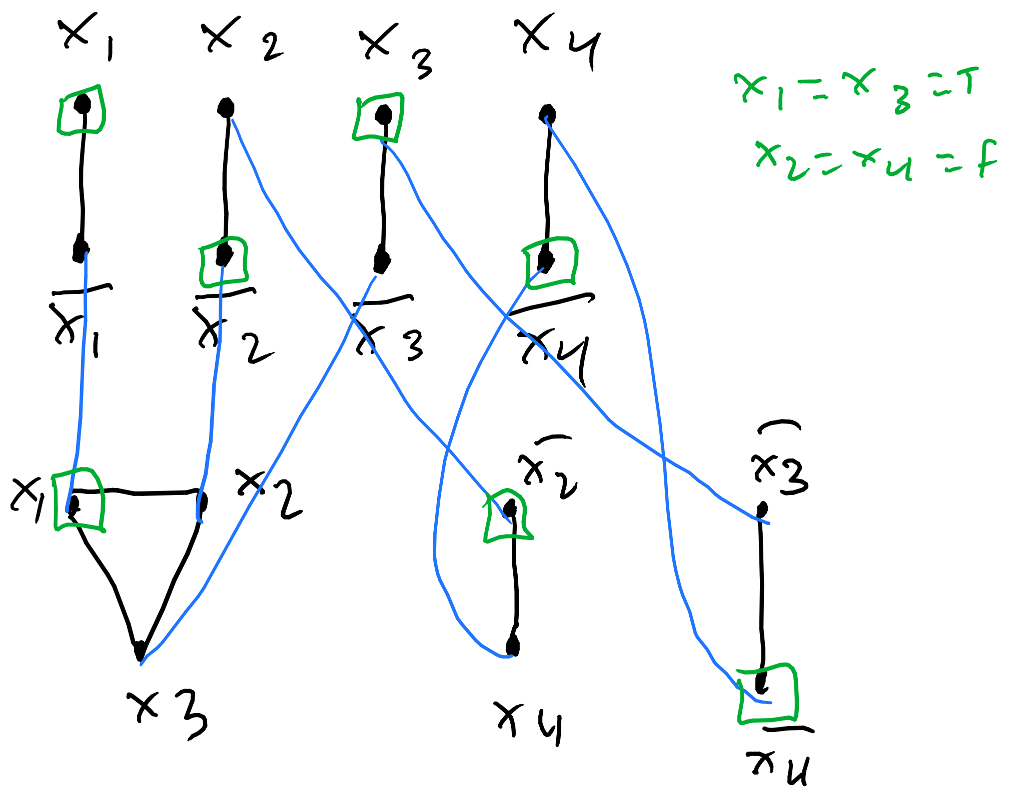
IND:
Input: Undirected G, $k \in \mathbb{N}$
Q: Does G have an independent set of size $\geq k$?



How to show that SAT is polynomially reducible to IND? (Look at last lecture for construction of a graph G_ϕ representing SAT)

Claim: ϕ is satisfiable $\implies$ G_ϕ is an independent set of size $k = m + n$ where n is the number of variables, m is the number of clauses.

In our example:



Pf: Consider a truth assignment that satisfies ϕ. We pick the corresponding vertices from the matching that was added in step 1 and one true term from each clause. It is not hard to see that this is an independent set.

Claim: If G_ϕ has an independent set of size $k = m + n \implies \phi$ is satisfiable.

Pf: An independent set of size $k = m + n$ has to pick exactly one node from each edge in step 1 and one node from each clause. Then the corresponding truth assignment will have at least one true term in each clause $\implies \phi$ is satisfiable.

$$\phi \text{ is satisfiable} \iff G_\phi \text{ has an independent set of size } k$$

$SAT \leq_P IND$:

Given an input ϕ for SAT, construct G_ϕ and compute $k = m + n$.

```
**if** $G_\phi$ has an independent set of size $\geq k$ **then** Output "$\phi$ is satisfiable"
**else** Output "$\phi$ is not satisfiable"
**end if**
```

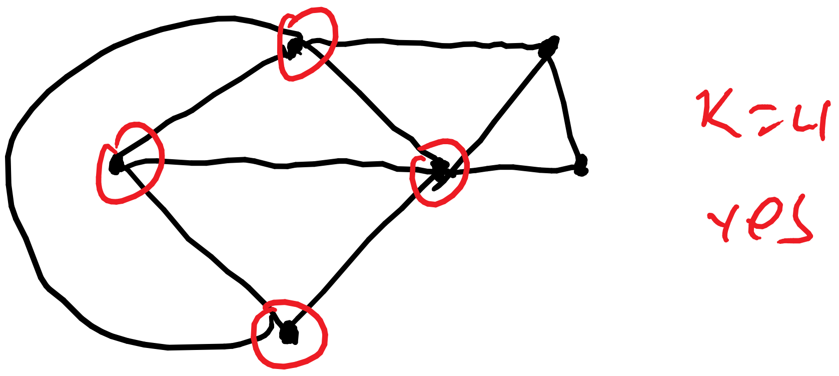
Thm: IND is NP-Complete.

$$P = NP \iff IND \in P$$

17.2 CLIQUE:

Input: Undirected $G, k \in \mathbb{N}$.
Q: Does G have a clique (set of vertices that are mutually adjacent) of size $\geq k$?

Ex:



Thm: $CLIQUE$ is NP-Complete.

Pf: It is in NP. A clique of size k can be used as a certificate and verified efficiently by checking the adjacency of the vertices.

To prove completeness we show $IND \leq_P CLIQUE$.

Given an input (G, k) for IND
We construct $\overline{G}$ by replacing edges with non-edges and vice versa.

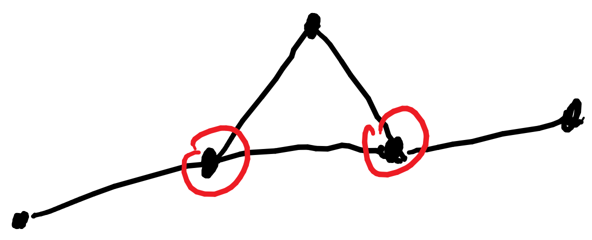
if $\overbrace{\overline{G} \text{ has a clique of size } k}^{\text{oracle}}$ **then** Output "Yes : G has an independent set of size k"

```
else Output "NO: $G$ does not have an independent set of size $k$"
end if
```

17.3 Vertex Cover:

Input: Undirected G, $k \in \mathbb{N}$.
Q: Does G have a vertex cover of size $\leq k$?

Recall: Vertex cover: a set of vertices such that removing them will remove all the edges.



$K = 2$
yes

Recall: König: For bipartite graphs, the size of the minimum vertex cover = size of the largest matching and this can be solved using *Max-Flow* in polytime. $\implies$ For Bipartite graphs, VC can be solved in polytime.

Thm: VC is NP-Complete.

Pf: $VC \in NP$, certificate is a vertex cover of size k and can be verified in polytime.
To prove completeness we show $IND \leq_P VC$.

```
Given an input $(G, k)$ for $IND$.
if $G$ has a vertex cover of size $|V(G)| - k$ then Output "Yes (independent set of size $k$)"
else Output "NO (independent set of size $k$)"
end if
```

Note that if there is a vertex cover S of size $n - k \implies \overline{S}$ is an independent set of size k, i.e. after removing the vertices of a vertex cover, the vertices we are left with form an independent set.

On the other hand, if G has an independent set T of size $k \implies \overline{T}$ is a vertex cover of size $\leq n - k$.

G has an independent set of size $k \iff G$ has a vertex cover of size $n-k$

$SAT, IND, CLIQUE, VC$

17.4 SET COVER

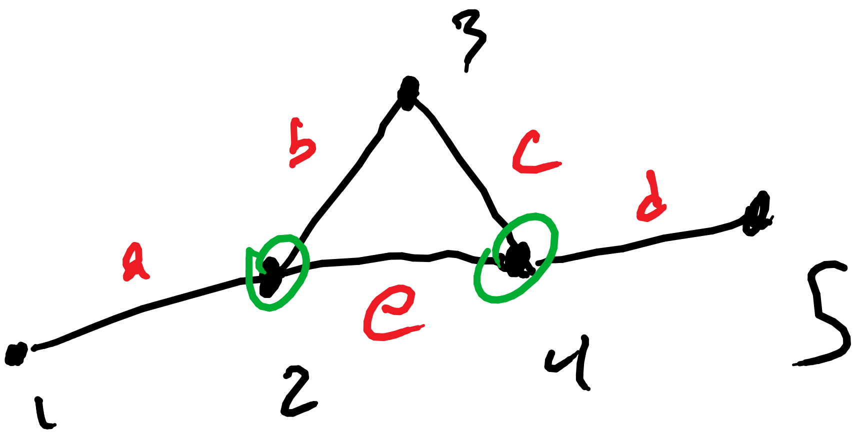
Input: Sets $S_1, S_2, \ldots, S_m \subseteq U$ where U is finite. $k \in \mathbb{N}$
Q: Can we pick k of these sets so that their union is all of U?

Ex: $S_1 = \{1,2,3\}, S_2 = \{2,3\}, S_3 = \{1,4\}, U = \{1,2,3,4\}, k = 2$, yes, pick S_1 and S_3.

Thm: $SET\ COVER$ is NP-Complete.

Pf: It is in NP (the certificate is a selection of the sets, check if they make U). To prove completeness $VC \leq_P SET\ COVER$

```
Given an input $(G, k)$ for $VC$
Let $U = E(G)$
if $v_1, \ldots, v_n$ are the vertices of $G$ then let $S_i = \{e \in E \mid e \text{ is adjacent to } v_i\}$
    if $S_1, \ldots, S_n \subseteq U$ has a set cover of size $k$ then Output "Yes $(G,k) \in VC$"
    else Output "NO $(G,k) \notin VC$"
    end if
end if
```



$S_1 = \{a\}, S_2 = \{a,b,e\}, S_3 = \{b,c\}, S_4 = \{e,c,d\}, S_5 = \{d\}, U = \{a,b,c,d,e\}$.

18 03/26/18

Recall So far we showed the following are NP-Complete:

$$SAT, IND, CLIQUE, Vertex\ Cover, Set\ Cover$$

The strategy to show that something is NP-Complete:

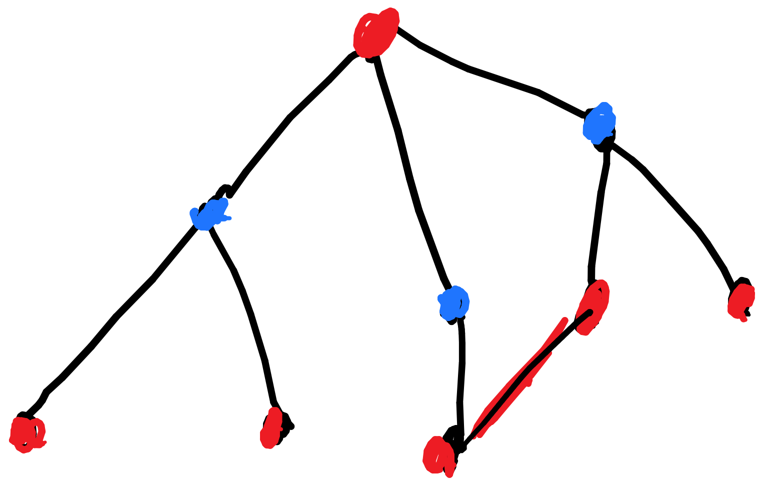
1. $X \in NP$
2. $Y \leq_P X$ for some NP-Complete Y

$\implies X$ is NP-Complete.

18.1 3-COL

Input: Undirected G
Q: Does G have a proper 3-coloring of the vertices?

Remark: 1-COL: Easy ($\in P$), G is 1COL $\iff E = \emptyset$
2COL $\in P$: We can start by coloring a vertex and then that forces the colors of its neighbors, and the neighbors of neighbors, etc.



Thm 3-COL is NP-Complete.

Pf: 3-$COL \in NP$ as we showed earlier (coloring is the certificate, easy to verify if the coloring is valid).

To prove completeness, reduce 3SAT to this problem.

3-SAT
Input: A CNF ϕ such that every clause has exactly 3 terms.
Q: Is ϕ satisfiable?

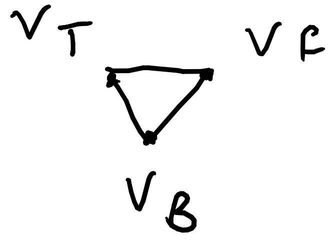
Thm $SAT \leq_P$ 3-$SAT \implies$ 3-SAT is NP-Complete. (Proof omitted, it is in the textbook and we are running behind)

$$\phi = (x_1 \lor x_2 \lor x_3) \land (\overline{x_1} \lor x_2 \lor x_4) \land (\overline{x}_2 \lor \overline{x}_3 \lor x_4) \land \dots$$

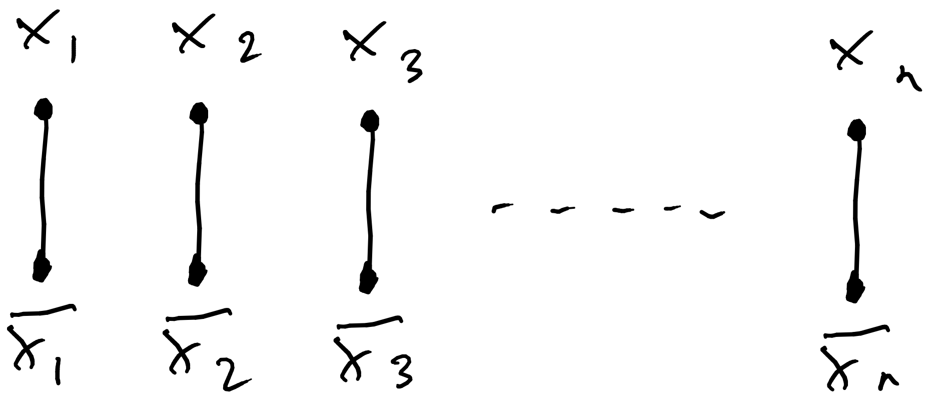
We want to show 3-$SAT \leq_P$ 3-COL. So we are given a $3CNF\phi$ and we want to decide whether ϕ is satisfiable using an oracle for 3-COL. We will construct a graph G_ϕ such that

$$\phi \text{ is satisfiable} \iff G_\phi \text{ is 3-}COL$$

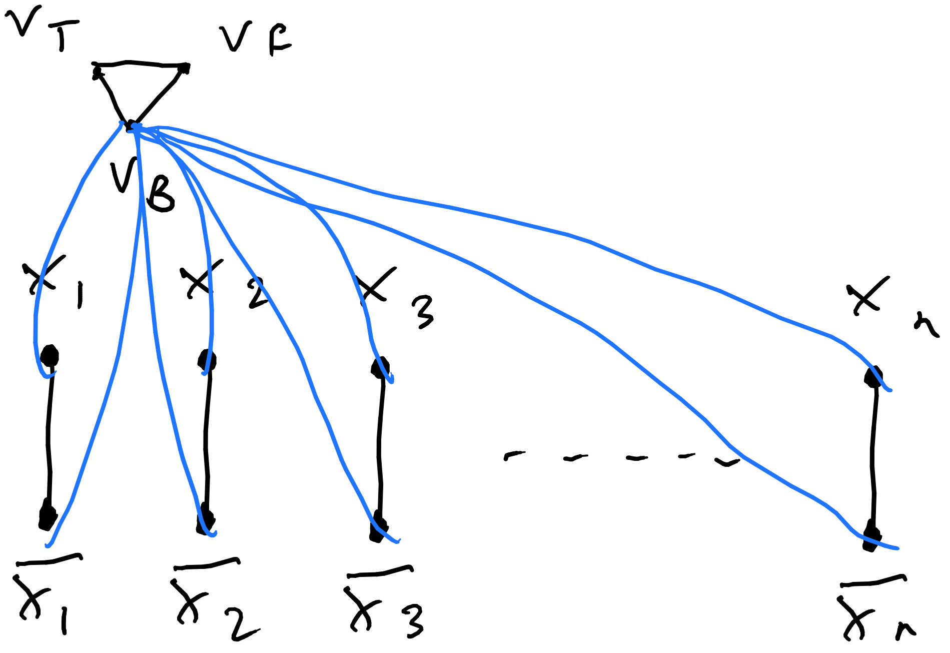
Idea: We will think of colors as $\{T, F, B\}$, where B is some other color (ex. Black). Start with a triangle with nodes v_T, v_F, v_B (without loss of generality we can assume that we are looking for a 3-COL that colors v_T with T, v_F with F and v_B with B).



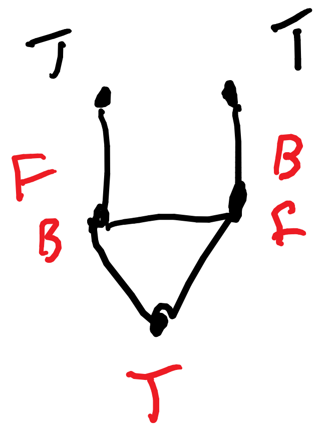
G_ϕ :
Next we add a matching of size n by adding an edge between two vertices with labels x_i and $\overline{x}_i$ for $i = 1, \dots, n$.

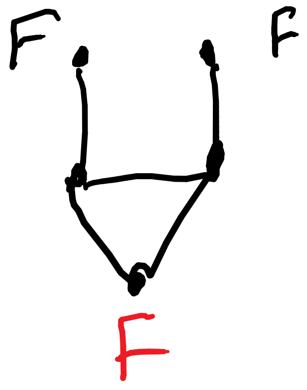


We connect v_B to all these vertices (so they don't receive the color B). Now any 3 coloring of this graph gives us a truth assignment to $x_1, \dots, x_n$ and vice versa.

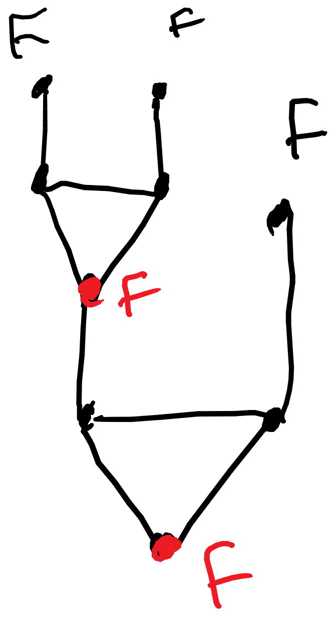


Next we want to deal with the restriction that every clause has to have at least one true term. Take note of the following interesting graph:

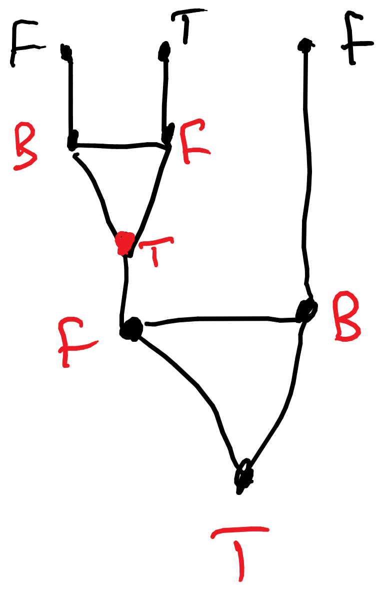




If the top vertices have the same color then the bottom vertex has to have the same color. Now look at the following graph:



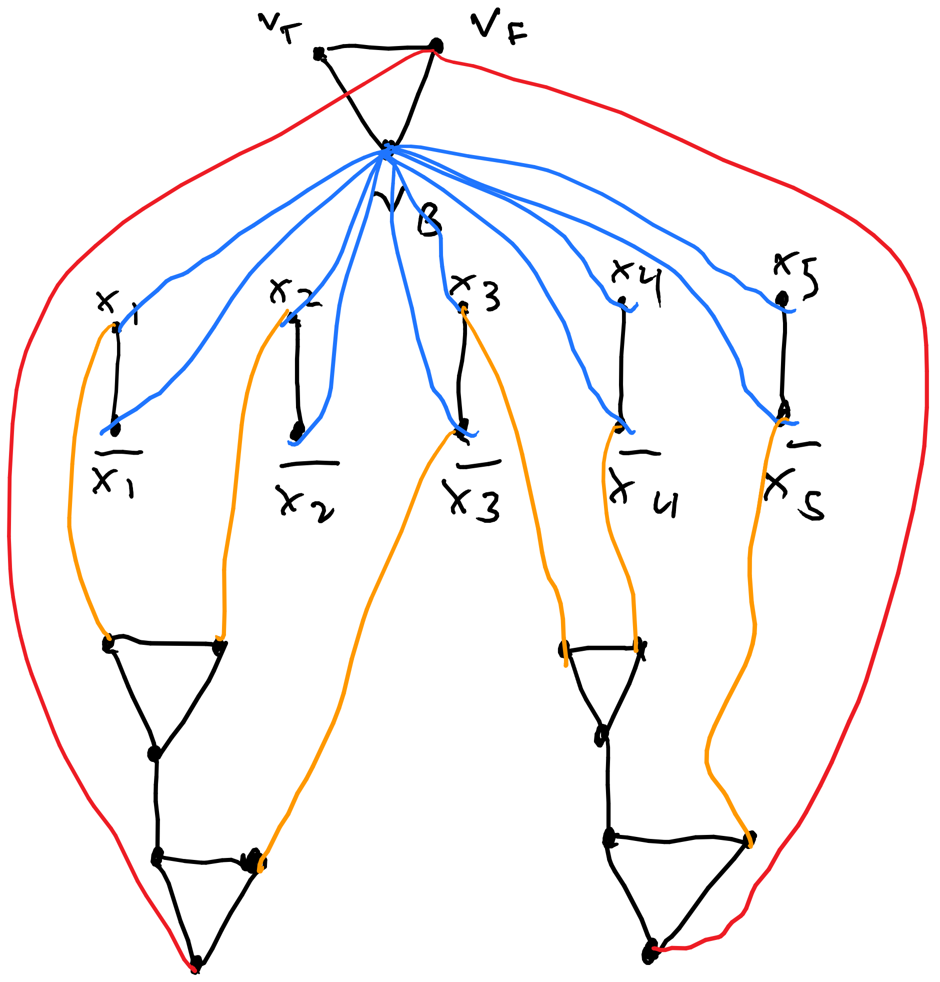
If we have this, then we once again have no choice but to color the bottom vertex F. On the other hand if we have anything other than FFF at the top, we can color this so that the bottom vertex is T.



Ex:
And this is true for all $TTT, TTF, TFT, FTT, FTF, FFT, TFF$

Now for every clause in ϕ we glue a copy of this gadget on the corresponding terms (they become the top vertices of the gadget).

Ex: $\phi = (x_1 \vee x_2 \vee \overline{x}_3) \wedge (x_3 \vee \overline{x}_4 \vee \overline{x}_5)$



We connect the bottom vertices to v_F.

$$G_\phi \text{ is 3-}COL \iff \phi \text{ is satisfiable}$$

If ϕ is satisfiable $\implies$ color the terms according to the satisfying the truth assignment. Every clause gets at least one true term $\implies$ all bottom vertices can be colored with T.

On the other hand, any proper 3 coloring give us a truth assignment that assigns at least one true term to each clause (because $FFF \to F$ property of the gadget) $\square$

18.2 COL

Input: $G, k \in \mathbb{N}$
Q: Is G k-colorable?
NP-Complete. 3-$COL \leq_P COL$. Given G for 3-COL, we query the oracle with (G, k) where $k = 3$ and that will tell us whether G is 3-colorable or not.

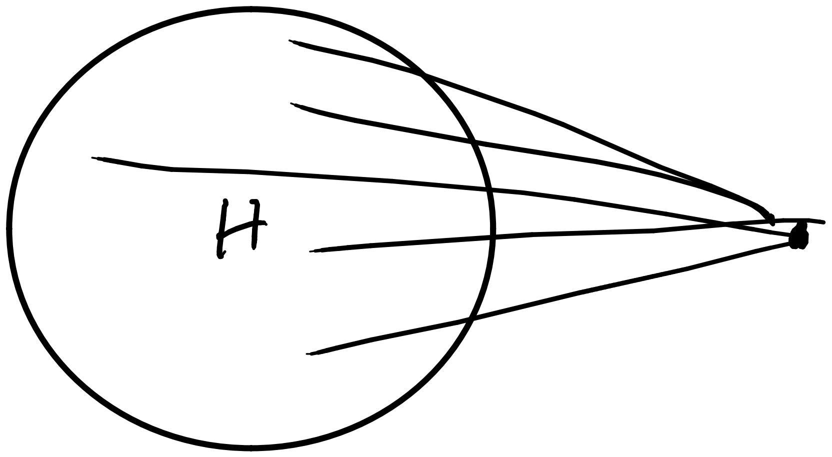
18.3 4-COL

Input: G
Q: Is G 4-colorable?
4-COL is NP-Complete.

Pf: It is in NP (easy)

$$3\text{-}COL \leq_P 4\text{-}COL$$

Given an input H for 3-colorability, add a new node and connect it to every node in H, call it G.



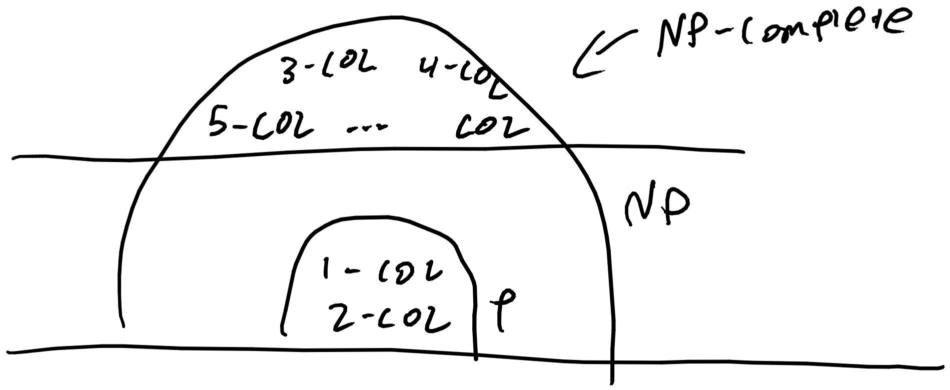
Note H is 3-$COL \iff G$ is 4-COL

Oracle alg:

```
Given $H$ (for 3-$COL$)
Construct $G$
if $G$ is 4-$COL$ then $H$ is 3-$COL$
else $H$ is not 3-$COL$
end if
```

Similarly, we can do the same thing for 5-colorability and 6-colorability, ...

Thm: For every $k \geq 3$, the k-colorability is NP-Complete. Note this is different from the COL problem mentioned above, the problem is only defined after you pick k, i.e. here we have infinitely many problems, whereas COL is only one problem that is NP-Complete.



Thm: 2-SAT is in P.

Pf: In 2-SAT, every clause has exactly two terms.
Example: $\phi = (x_1 \vee x_2) \wedge (x_1 \vee \overline{x}_3) \wedge (\overline{x}_1 \vee \overline{x}_2) \wedge (x_2 \vee x_3) \wedge (\overline{x}_1 \vee x_3) \wedge (\overline{x}_3 \vee x_2)$
What happens (similar to 2-COL):

$$x_1 = T \implies \begin{cases} \overline{x}_2 = T \implies \begin{cases} x_1 = T \\ x_3 = T \\ x_3 = F \times \end{cases} \\ x_3 = T \implies x_1 = T \end{cases}$$

Now we know $x_1 = F \implies \begin{cases} x_2 = T \\ x_3 = F \end{cases}$

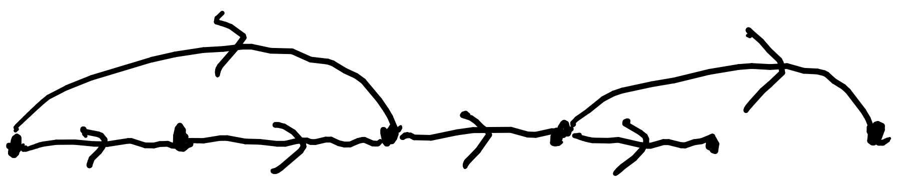
19 03/28/18

So far, we proved the following are NP-Complete:

$$SAT, IND, CLIQUE, 3SAT, \mathit{Vertex\ Cover}, \mathit{Set\ Cover}, k\text{-}col(k \geq 3)$$

On the other hand, $2SAT, 2COL \in P$.

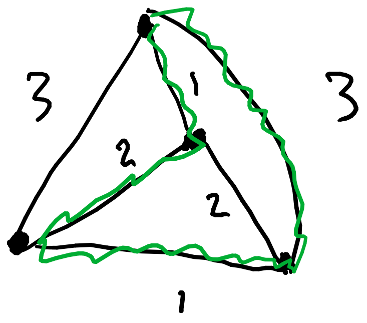
Hamiltonian Path, Hamiltonian Cycle in both directed and undirected graphs are NP-Complete.
Remark: Hamiltonian path in directed acyclic graphs is in P. There's a topological order, since there's one node that has no incoming edges, remove that and then remove the next one, etc.



19.1 Traveling Salesman Problem

There are n cities and we are given the pairwise distances between them. A traveling salesman wants to start from a city, visit every other city and come back to the starting point, minimizing the traveled distance.
Input: d_{ij} for $1 \leq i < j \leq n$ (distance between i and j)
$K \in \mathbb{N}$
Q: Can it be done with $\leq K$ traveled distance?

Ex:



Here, $K = 7$ is true, with a possible path shown in green.

Thm TSP is NP-Complete.

Pf: It is in NP...

To prove completeness we show Hamiltonian cycle $\leq_P TSP$

```
Given an input $G$ for Hamiltonian cycle
if $ij \in E$ then Set $d_{ij} = 1$
else if $ij \notin E$ then Set $d_{ij} = 2$
end if
$K = n$
if there is a $TSP$ of length $\leq K$ then $G$ has a Hamiltonian cycle
else "No"
end if
```

Why? If G has a Hamiltonian cycle $\implies$ same cycle has total distance $\underline{n}$.

On the other hand if there is a TSP cycle of length $\leq n$ since there are n traveled pairs each has to be of distance ≤ 1.

19.2 Subset Sum

(Very similar to the Knapsack problem)

Input: Numbers $w_1, \ldots, w_n \in \mathbb{N}$

Number $W \in \mathbb{N}$

Q: Is there a subset of $w_1, \ldots, w_n$ where sum is exactly W?

Ex: $5, 4, 3, 4, 8, 5$
$W = 13 \implies$ Yes, $5 + 5 + 3 = 13$
$6, 5, 8, 4$
$W = 7 \implies$ NO
NP-Complete: Easy to see in NP. It is possible to show $3SAT \leq_P$ *Subset Sum* (write the numbers as digits, 1 if the clause in that term is true, 2 if it's false, etc.)

19.3 Knapsack

Input: $w_1, \ldots, w_n \in \mathbb{N}$
$W \in \mathbb{N}$ capacity
$K \in \mathbb{N}$
Q: Can we pick a subset of $w_1, \ldots, w_n$ whose sum does not exceed W and is at least K?

Knapsack is NP-Complete.

Pf: Easy to see it is in NP ...
Subset Sum $\leq_P$ *Knapsack*

```
Take an input of Subset Sum
Set $K = W$
Feed it to the oracle for Knapsack
**if** Yes **then** Output Yes
**else if** No **then** Output No
**end if**
```

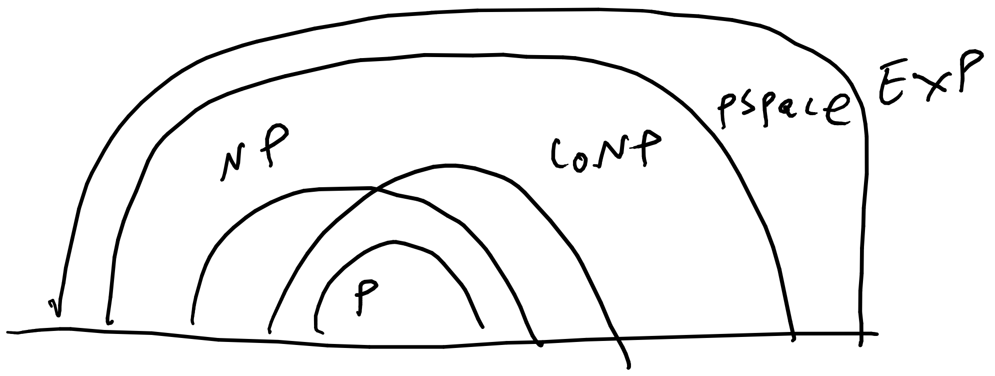
19.4 PSPACE:

The class of problems that can be solved using polynomial space (number of memory bits).

$P \subseteq PSPACE$
An algorithm that runs in polynomial time cannot use more than polynomially many bits (as writing those bits would take more time than the algorithm can afford).
$NP \subseteq PSPACE$
We can use the same proof that showed $NP \subseteq EXP$, except we make sure we reuse our memory.
Consider $X \in NP$ and an efficient certifier for X that takes (w, t) where w is the input and t a potential certificate where $|t| \leq p(|w|)$ for some polynomial $p()$.

```
Normal Alg:
Input $W$
Generate all possible $t$ of size at most $p(|w|)$ one by one reusing the space.
Run the certifier on $(w, t)$
if outputs Yes then Output Yes and terminate
else if they all fail then Output No
end if
```

$PSPACE \subseteq EXP$



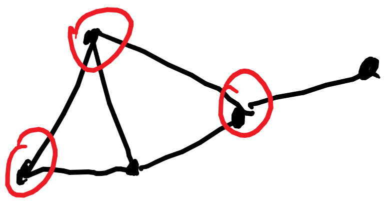
19.5 QSAT

(We think this problem is in $PSPACE$ but not in EXP)
Input: CNF ϕ and a list of quantifiers over the variables.
Q: Is the corresponding formula true?

Ex: $\forall x_1 \exists x_2 \exists x_3 (x_1 \vee x_2) \wedge (x_1 \vee \overline{x}_2) \wedge (\overline{x}_1 \vee x_2 \vee x_3)$, this is false, if $x_1 = F$ then we cannot satisfy it.
$QSAT$ is $PSPACE$-Complete.

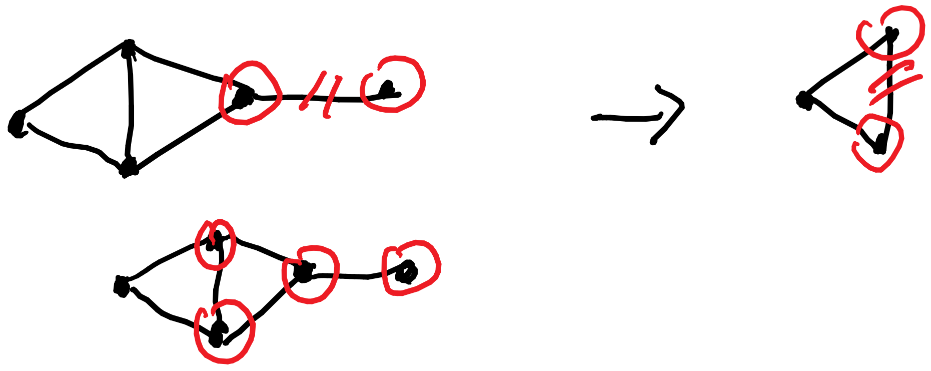
Many natural problems that we would like to solve are NP-Complete.
It is believed that there are no sub-exponential algorithms for NP-Complete problems!

Example: Vertex Cover
Input: Graph G
Q: What is the size of the smallest Vertex Cover in G?

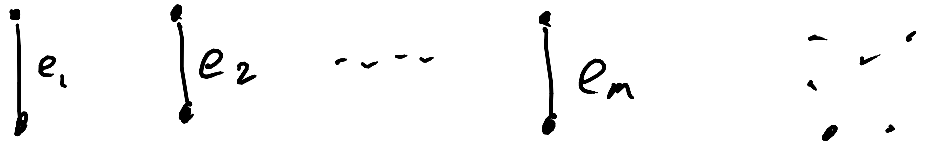


A naive algorithm:

```
while there is an edge in $G$ do
    Pick both of its endpoints and remove them.
end while
```



The algorithm picks $2m$ nodes if it comes across $e_1, e_2, \ldots, e_m$ throughout its execution.



No vertex can cover more than one e_i, so the minimum vertex cover is of size at least m.

Thm: The output of our algorithm $\leq 2 \times \min\ VC$. Called a **2-factor approximation** algorithm.

20 04/04/18

20.1 Approximation Algorithms

For a maximization problem, an algorithm is an α-factor approx algorithm if (not too small, $0 < \alpha \leq 1$)

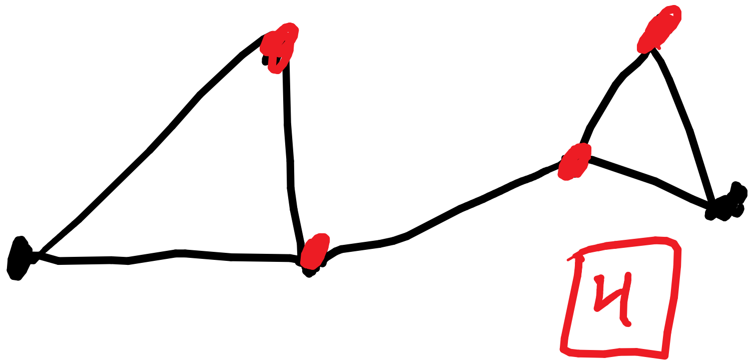
$$output \geq \alpha \times optimal$$

For a minimization if (not too big, $1 \leq \alpha$)

$$output \leq \alpha \times optimal$$

Ex: Seen last class, for vertex cover: Pick both endpoints of an edge, delete them and remove until all edges are gone. 2-factor.

Another algorithm for vertex cover



Cover all edges.

20.2 Integer Linear Program for Vertex Cover

Note that we can formulate any NP-complete program as an integer linear program. ILP-VC
Vars: $x_v, v \in V$

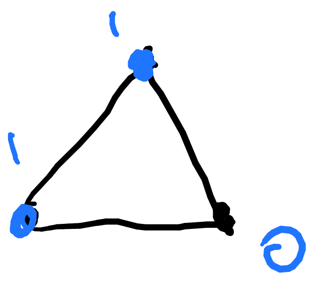
$$\min \sum_{v \in V} x_v$$

$$x_u + x_V \geq 1, \forall uv \in E$$
$$x_v \in \{0,1\}, \forall v \in V$$

LP relaxation: *LP-VC*

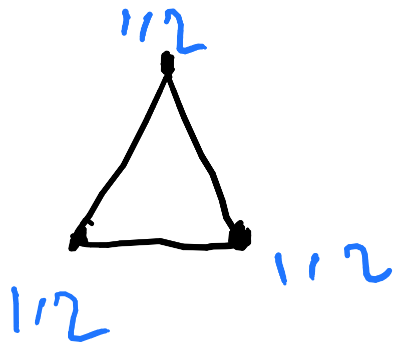
$$\min \sum_{v\in V} x_v$$
$$x_u + x_v \geq 1, \forall uv \in E$$
$$x_v \geq 0, \forall v \in V$$
$$x_v \leq 1, \forall v \in V \text{ (Remark: redundant and can be removed)}$$

Ex:



ILP

$opt(ILP\text{-}VC) = 2$



LP:

$opt(LP\text{-}VC) = \frac{3}{2}$

So $opt(LP\text{-}VC) \neq opt(ILP\text{-}VC)$

Obs: $opt(LP\text{-}VC) \leq opt(ILP\text{-}VC)$. Solutions for *ILP-VC* are also solutions for *LP-VC*, but *LP-VC* has less constraints and can sometimes be further minimized, like in the example. So solving *LP-VC* will tell us the optimal vertex cover is at least the size of the optimal solution for *LP-VC*.

We solve the LP in polynomial time and obtain an optimal solution (x_v^* for $v \in V$) for LP.

We want to round these to an integer solution to the ILP.

Let $\overline{x}_v = \begin{cases} 0 & \text{if } x_v^* < \frac{1}{2} \\ 1 & \text{if } x_v^* \geq \frac{1}{2} \end{cases}$

Note for every edge $uv, x_u^* + x_v^* \geq 1$.

$\implies$ at least one of x_u^* or $x_v^* \geq \frac{1}{2} \implies$ at least one of $\overline{x}_u$ or $\overline{x}_v$ is 1 $\implies \overline{x}_u + \overline{x}_v \geq 1$.

So $\overline{x}_v$ for $v \in V$ is a feasible solution to

$$\begin{aligned} \min \sum_{v \in V} & x_v \\ & x_u + x_v \geq 1, \forall uv \in E \\ & x_v \in \{0,1\}, \forall v \in V \end{aligned}$$

What can we say about $\sum_{v \in V} \overline{x}_v$?

Note $\overline{x}_v \leq 2x_v^*$

$$\implies \mathit{Output} = \sum_{v \in V} \overline{x}_v \leq 2 \sum_{v \in V} x_v^* = 2\mathit{opt}(LP\text{-}VC)$$

Alg:

1. Solve the Linear Program relaxation. Let x_v^*'s be an optimal solution.
2. Round x_v^*'s to $\overline{x}_v$'s using the above formula and output the vertices with $\overline{x}_v = 1$.

$$\mathit{Output} \leq 2\mathit{opt}(LP\text{-}VC) \leq 2\mathit{opt}(ILP\text{-}VC)$$

This is a 2-factor approximation algorithm.

20.3 Knapsack

Input: Integers $w_1, \ldots, w_n \geq 0$ and capacity $W \in \mathbb{N}$.

Goal: Pick a subset $S \subseteq \{1, \ldots, n\}$ so that $\sum_{i \in S} w_i \leq W$ and this sum is maximized.

Ex: $W = 10, w_1 = 6, w_2 = 5, w_3 = 3$

$9 = 6 + 3 \leq 10, \mathit{opt} = 9$

NP-Complete (as seen in a previous lecture).

A greedy alg:

Sort the items $w_1 \geq w_2 \geq \ldots \geq w_n$

```
for $i = 1, \dots, n$ do
    if we can add $w_i$ to the knapsack add it. then
    end if
end for
```

Ex: $W = 10$

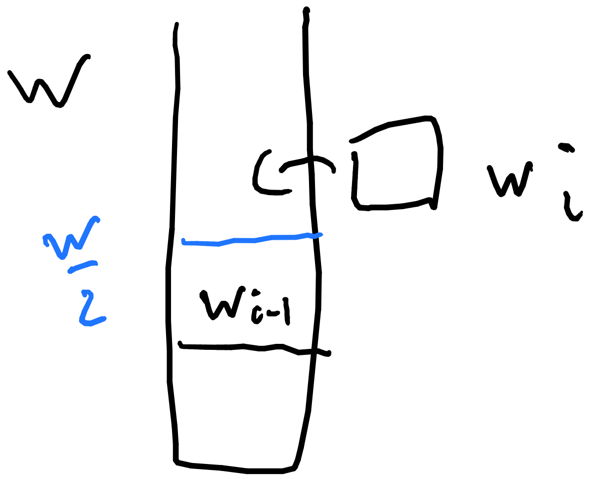
$6, 5, 3, 2 \implies 6 + 3$ (output of greedy)

Optimal is $5 + 3 + 2 = 10$

We can disregard any item whose weight is larger than W. So we can assume $W \geq w_1 \geq \dots \geq w_n$. If the algorithm doesn't pick some item $\implies$ the knapsack is at least half full.

This is because if w_i is the first item that is not picked by the algorithm $\implies w_{i-1} \geq w_i$ is in the knapsack and the knapsack has at least $W - w_i$ weight in it.

If $\sum_{j=1}^{i-1} w_j < \frac{W}{2} \implies w_i \leq w_{i-1} < \frac{W}{2} \implies$ we could add w_i to the knapsack.



The algorithm either picks all the items or (inclusive) it fills up at least $\frac{W}{2} \geq \frac{opt}{2}$ in the knapsack.

In both cases:

$$Output \geq \frac{1}{2} Optimal$$

$\implies \frac{1}{2}$-factor approximation.

20.4 Load balancing problem

- Jobs with processing times $t_1, t_2, \ldots, t_n \in \mathbb{N}$
- m number of machines.

Goal: Distribute the jobs over these machines so that we minimize the last finishing time.

Ex: $m = 3$
$2, 3, 4, 6, 2, 2$

$$7 = opt, \begin{array}{|c|c|c|} \hline & & 2 \\ 4 & & 2 \\ 3 & 6 & 2 \\ \hline \end{array}$$

Alg I:

```
while there are jobs do
    Assign the current job to the machine with smallest current load
end while
```

In our example: $8 = \begin{array}{|c|c|c|} \hline 6 & 2 & 2 \\ 2 & 3 & 4 \\ \hline \end{array}$. This is an online algorithm, can use it while receiving jobs live. Clearly not optimal, we will show next lecture that it is a 2-factor approximation. The better approach would be to sort them first.

Ex: $\underbrace{1, 1, 1, \ldots, 1}_{(m-1)\times m}, m$

$$m + m - 1 = 2m - 1 = \begin{array}{|c|c|c|c|} \hline m & & & \\ 1 & 1 & \ldots & 1 \\ \ldots & \ldots & \ldots & \ldots \\ 1 & 1 & \ldots & 1 \\ \hline \end{array}$$

Optimal: $m = \begin{array}{|c|c|c|c|} \hline & 1 & \ldots & 1 \\ & \ldots & \ldots & \ldots \\ m & 1 & \ldots & 1 \\ \hline \end{array}$ So we see that this algorithm can give us almost 2 times the optimal solution.

21 04/09/18

21.1 Load Balancing

Input: Processing times $t_1, t_2, \ldots, t_n$
Number of machines m.

Goal: Distribute the jobs among the m machines so that the last finishing time is minimized.

Ex: $2, 3, 4, 6, 2, 2$, $m = 3$

$$Opt = 7, \begin{array}{|c|c|c|} \hline & & 2 \\ 3 & & 2 \\ 4 & 6 & 2 \\ \hline \end{array}$$

Alg I: Assign the current job to the machine with the current lowest load.

$$\text{Output} = 8, \begin{array}{|c|c|c|} \hline 6 & 2 & 2 \\ 2 & 3 & 4 \\ \hline \end{array}$$

Thm: Alg I is a 2-factor approximation algorithm.

Proof: Let T^* be the optimal solution and T be the output of the alg. We need to show

$$T \leq 2T^*$$

Note:

$$\max_i t_i \leq T^* \tag{5}$$

as every job has to be assigned to a machine. Also:

$$\frac{\sum_{i=1}^n t_i}{m} \leq T^* \tag{6}$$

as the total processing time $\sum_{i=1}^n t_i$ has to be distributed among m machines.

Let i be the machine with the highest load (T) in our algorithm and let t_j be the last job assigned to this machine.

For $\begin{array}{|c|c|c|} \hline 6 & 2 & 2 \\ 2 & 3 & 4 \\ \hline \end{array}$, $T = 8$, $t_j = 6$.

Let's look at the time that t_j is being assigned to this machine.

$$6 \rightarrow \begin{array}{|c|c|c|} & & \\ 2 & 3 & 4 \\ \hline \end{array}$$

The load was $T - t_j$ and that was the smallest load. $\implies$ The final loads of all machines are at least $T - t_j$. $\implies \sum_{i=1}^n t_i \geq m(T - t_j)$

$$\implies T^* \geq T - t_j \text{ by (6)}$$

Also

$$T^* \geq t_j \text{ by } (5)$$

Now, adding both inequalities, we get:

$$2T^* \geq T$$

Note that a lot of 2-factor proofs are like this, we show easier bounds and use them together.

Alg II: Sort the jobs $t_1 \geq t_2 \geq \ldots \geq t_n$ and then run Alg I.

Thm: Alg II is a $\frac{3}{2}$-factor approximation algorithm.

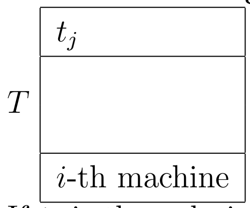
Pf: We need to show $T \leq \frac{3}{2}T^*$.

Let i be the machine with highest load and t_j be the last job on this machine. If $n \leq m \implies$ algorithm is optimal (each job goes to a separate machine).
Claim: If $n \geq m+1 \implies T^* \geq 2t_{m+1}$

Ex: $m = 3$
$6 \geq 4 \geq 3 \geq \underline{2} \geq 2 \geq 2$, $T^* \geq 2 \times 2$
$t_1 \geq t_2 \geq \ldots \geq t_{m+1}$
$t_a + t_b \geq t_{m+1} + t_{m+1} = 2t_{m+1}$

Pf of claim: One of the machines will have two of $t_1 \geq t_2 \geq \ldots \geq t_{m+1}$ and its load is going to be at least $2t_{m+1}$.

Back to the original proof: We focus on the case where $n \geq m+1$.



If t_j is the only job assigned to this machine $\implies T = t_j \leq T^* \implies T = T^*$.

So we can assume there are at least two jobs assigned to the i-th machine. Since $n \geq m+1 \implies j \geq m+1 \implies t_{m+1} \geq t_j$. This together with the claim shows $T^* \geq 2t_{m+1} \geq 2t_j$

Looking at the previous proof, we now have:

$$T^* \geq T - t_j$$

$$\frac{1}{2}T^* \geq t_j$$

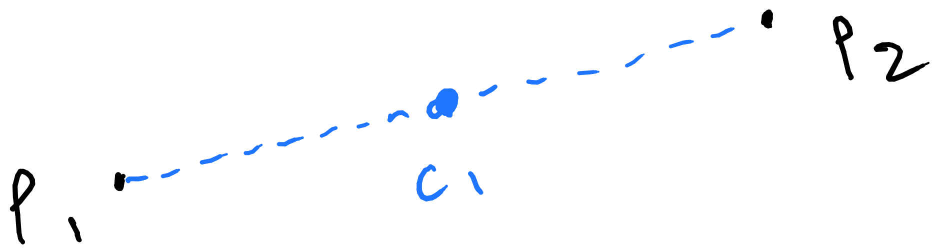
Summing the two:

$$\frac{3}{2}T^* \geq T$$

21.2 Center Selection

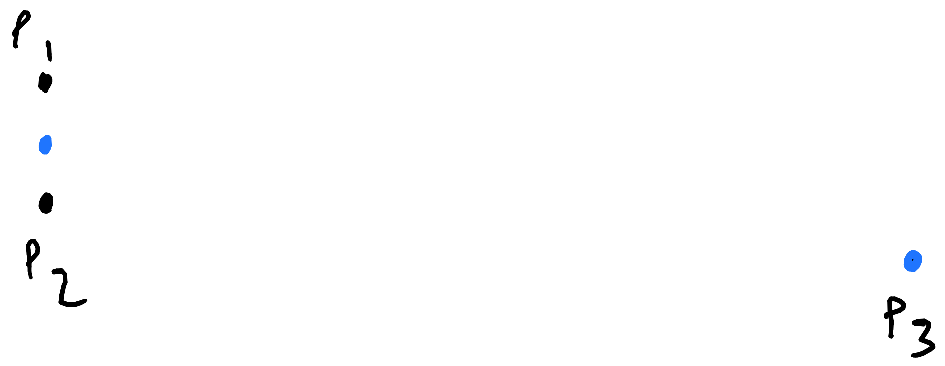
Input: A set S of n points $p_1, \ldots, p_n$ on the plane. A number $k \in \mathbb{N}$.
Goal: To select k "centers" on the plane such that the maximum distance of any point in S to a center is minimized.
$dist(p, C) = \min_{q \in C} dis(p, q)$, where C is the set of centers.

Ex: $k = 1$ and $n = 2$, put it in the middle.



$$dist(p_1, C) = dist(p2, C) = \frac{dist(p_1, p_2)}{2}$$

Ex: $k = 2$ and $n = 3$



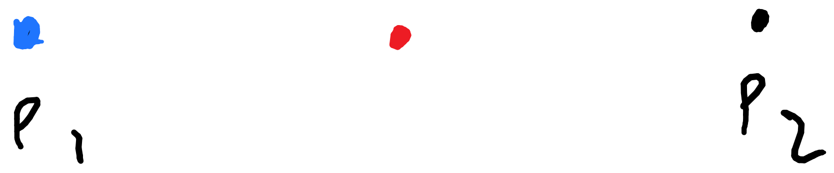
$$opt = \frac{dist(p_1, p_2)}{2}$$

Alg:

```
Put one of the points $p_1, \ldots, p_n$ in $C$.
for $i = 2, \ldots, k$ do
    Pick the point with largest distance to $C$ and add it to $C$
end for
```

Ex: $k = 3$

Ex: $k = 1$, $n = 2$. Output $= d_{12}$, Opt $= \frac{d_{12}}{2}$



$\implies$ The alg is not better than 2-factor.

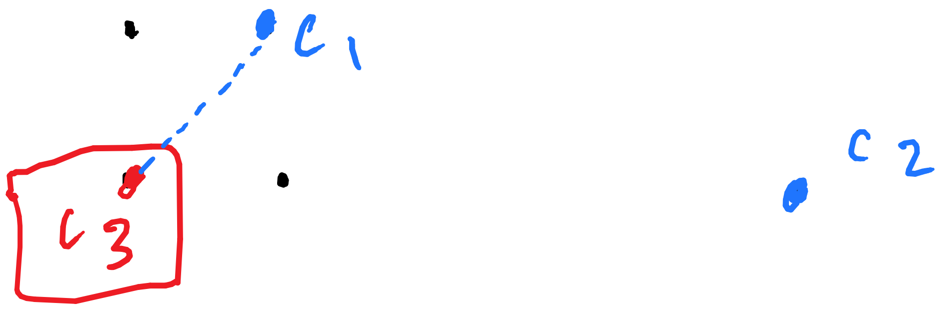
Thm: This is a 2-factor approximation algorithm.

Pf: If $n \geq k \implies$ alg is optimal as it puts a center on each point.

If $n > k \implies$ let $c_1, \ldots, c_k$ be the centers chosen by the algorithm and let c_{k+1} be the point with the largest distance from $\{c_1, \ldots, c_k\}$

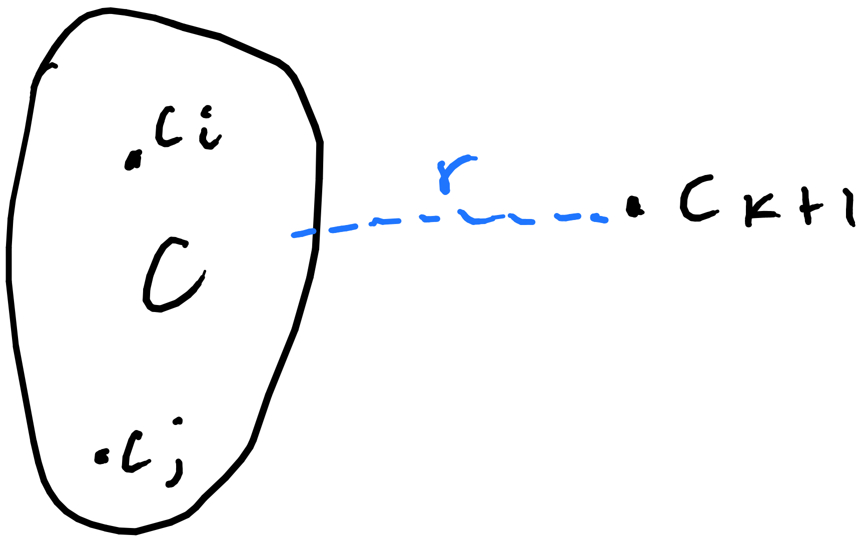
Output $= dist(c_{k+1}, C) = r$

Ex:



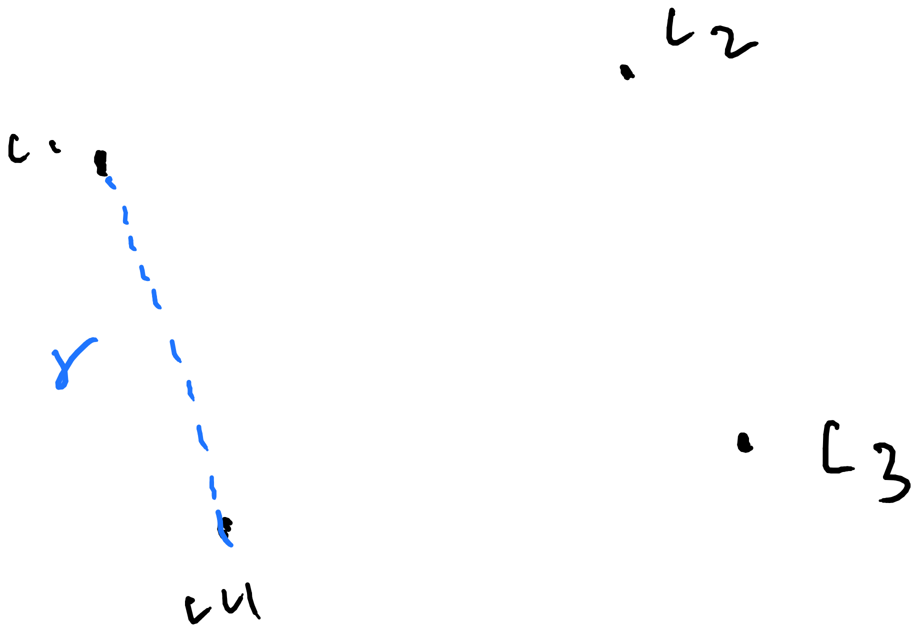
Consider $c_1, \ldots, c_k, c_{k+1}$

Note $dist(c_i, c_j) \geq r$ for all $i \neq j \in \{1, \ldots, k+1\}$



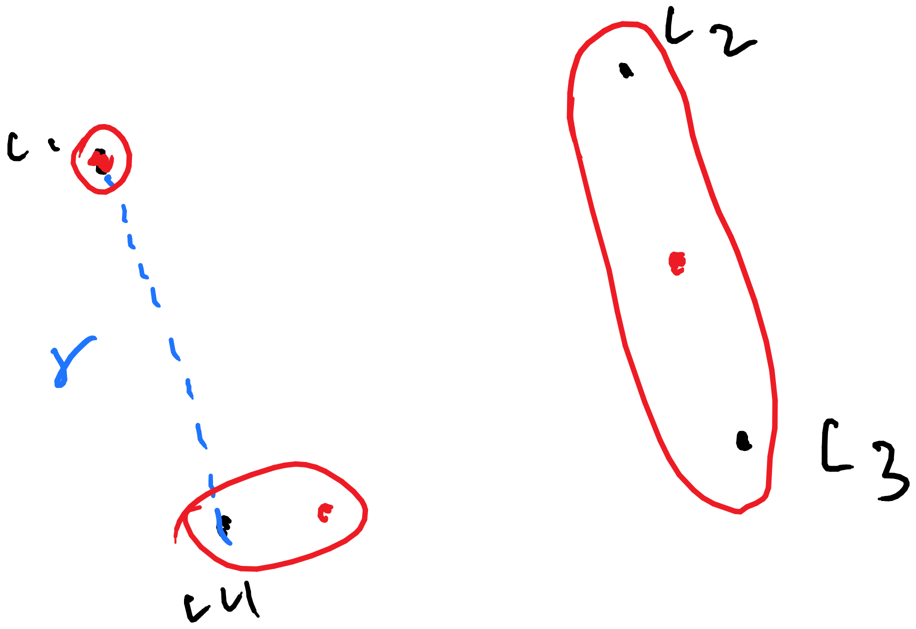
Indeed for $i < j \leq k$ then when j is chosen, its distance to c_i must have been larger than the distance between c_{k+1} and C, otherwise c_{k+1} would have been chosen.

Recap: We showed that there are $k + 1$ points whose pairwise distances are $\geq r$.

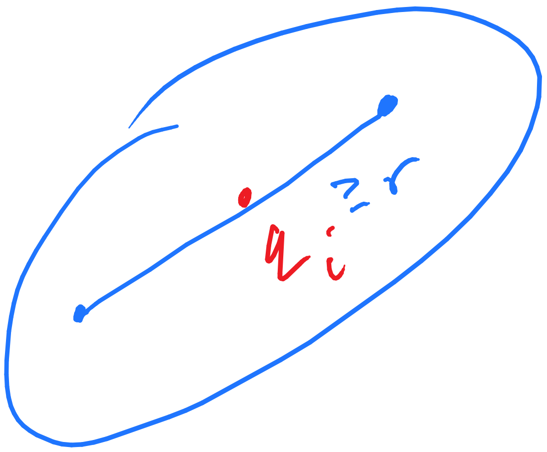


Q: Can opt be $< \frac{r}{2}$

Let $q_1, \ldots, q_k$ be the optimal centers. For each of these, let's look at the points that this center is the closest to.



One of $q_1, \ldots, q_k$ gets at least two points (whose distance is $\geq$ r). So this center is in distance at least $\frac{r}{2}$ to one of those.



$\implies output = r \leq 2 \times opt$