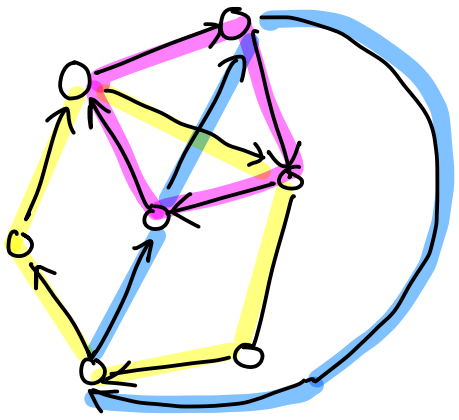
LECTURE 15 (October 23rd)

APPLICATION OF MINIMUM CUTS

We have been talking about various kinds of abstract problems that can be reformulated as different kinds of maximum flow problems usually through some generalization of the idea of finding disjoint paths / bipartite matching.

Cycle Cover Problem

Given a graph (not a DAG) cover the edges of G with cycles.
The cycles are allowed to share vertices but not edges!

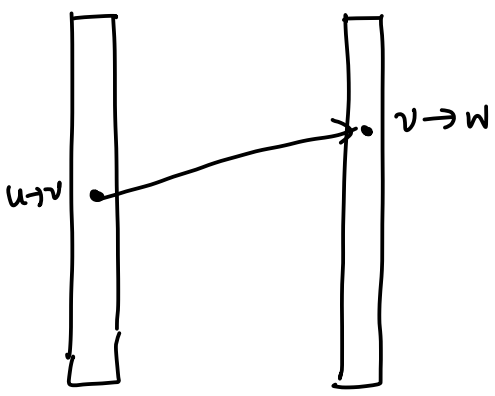


How can we turn this into a matching problem?

We want to find the successor of every edge on the same cycle

We can build a bipartite graph

$$G' = (L \cup R, E')$$



where $L = \{u \to v \mid (u \to v) \in E\} = R$
i.e. the vertices of G' are the edges of G. The only edges in E' are those where the tail of the left side equals the head on the right side., i.e.
$\{(u \to v), (v \to w)\}$ is an edge in E'
and those are all the edges.

vertices = $2|E|$
edges $\leq |E| \cdot |V|$

To find a cycle cover, we need to match all the vertices in L to R, i.e. we need to find a matching that matches all vertices in L. This is called a perfect matching. The way to do this is to compute the maximum matching and make sure every vertex in G' is matched

The Running time of this algorithm is

$$O((|L| + |R|) \cdot |E'|) = O(|E| \cdot |E| \cdot |V|)$$
$$= O(|E|^2 \cdot |V|)$$

Baseball Elimination Problem

Here is another problem that is partly motivated by the real-world.

When is a baseball team mathematically eliminated?

Various teams play over 100 games in the course of a year against each other and the idea is that when the main part of the season ends, you want to come out on top because that puts you into at least the competition for the world series. So, teams keep very careful statistics. One of the statistics that people keep track of is how many games has each team won and lost during the course of the season so far. And also how many games has each team left to play against other teams.

For example, here are the statistics for some of the teams from 1996

Team	**Won–Lost**	**Left**	NYY	BAL	BOS	TOR	DET
New York Yankees	75–59	28		3	8	7	3
Baltimore Orioles	71–63	28	3		2	7	4
Boston Red Sox	69–66	27	8	2		0	0
Toronto Blue Jays	63–72	27	7	7	0		0
Detroit Tigers	49–86	27	3	4	0	0	

Here Detroit is clearly behind but the question is it even possible for Detroit to come out on top. For example, if Detroit wins all of their remaining 27 games, they win 76 games in total, so it seems like the answer should be yes. But for this to happen, NY can only win at most one game, Baltimore at most 5 and so on. If you notice more carefully then, there are not enough games left here. For example, NY & Boston have to play 8 games and one of them will win. So, it is not clear if Detroit can come out on top and in fact one can make an ad hoc argument like this to see that Detroit can not come out on top.

But here we will try to resolve this question systematically. We are just going to set it up as a maximum flow problem, run Ford-Fulkerson and the answer will fall out if we have set this up correctly.

The input here is for each team

Wins [i] → How many games has team i won in the past?
Games [i,j] → How many future games are left between teams i and j?

The output is TRUE if team n could end the season in the 1st place (may be tied)

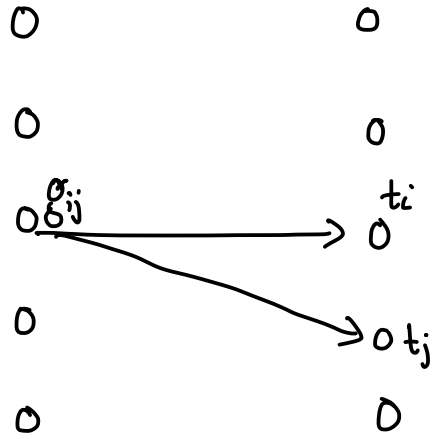
OR FALSE o/w

Let us also define Left [i] = $\sum_j$ Games [i,j] to be the total number of game team i has left

We are also going to assume that team n wins all of its remaining games.

So, team n wins the season iff every team i wins atmost Wins [n] + Left [n] - Wins [i] future games. In other words, we are trying to imagine for all of the future games if there is a way to assign a winner to each of those games so that this condition holds. Every other team plays a certain number of games but all of the games must be played and a winner assigned to every one of those games.

So, this seems like a bipartite matching problem since we are assigning winners to games. With that intuition, let us build a bipartite graph where left vertices represents games and right vertices represent teams.

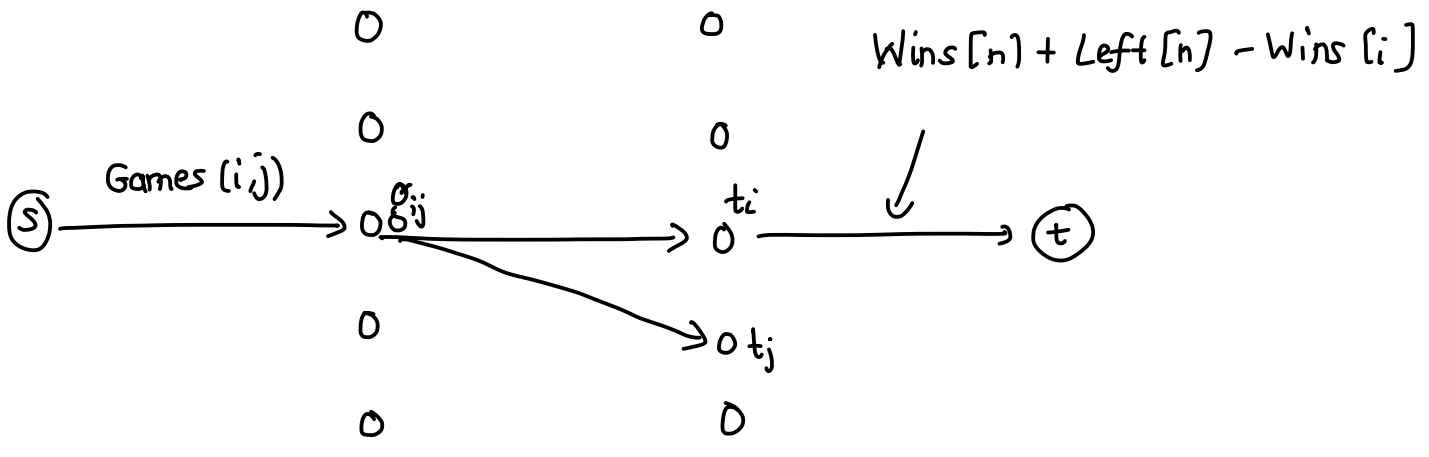


There are $\leq \binom{n-1}{2}$ game pairs

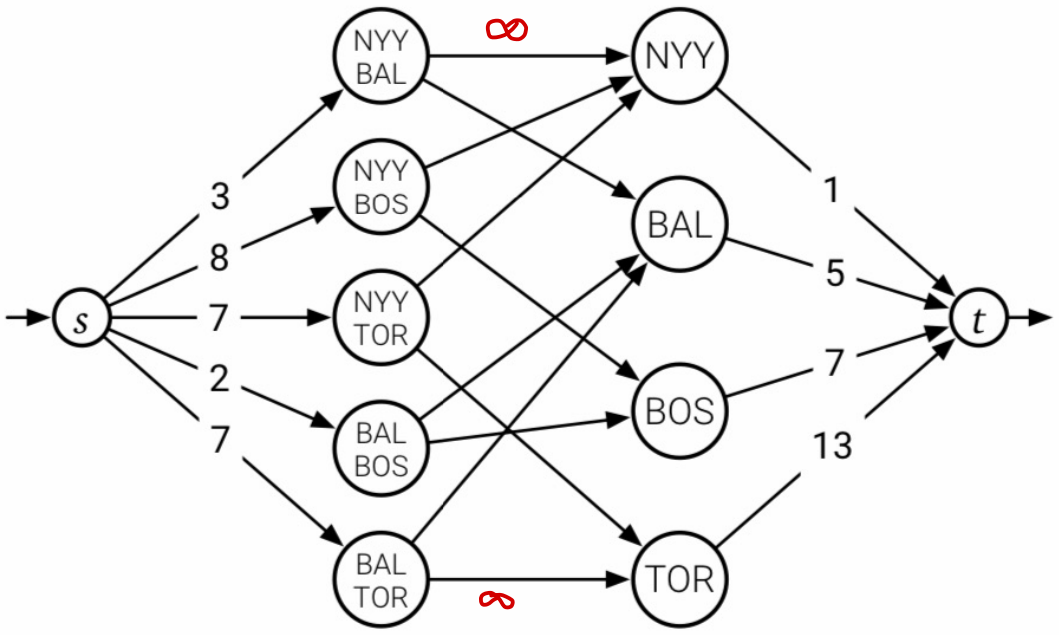
There are n-1 teams

For each of the future game, we want to assign a winner. Said differently, for all of the future games between i and j, we need to split them up. Some of these will be won by team i and the rest by team j
So, we connect each node g_{ij} to nodes t_i & t_j in the right layer

We also add a source s & a sink t and add edges with capacity Games[i,j] between s & g_{ij} and with capacity Wins[n] + Left[n] − Wins[i] between t_i & t



For the above example, the network looks like the following:



→ This has to be the max flow

Theorem Team n can win season ⟺ There is a flow in G that saturates every edge out of s

Proof (⇐) Suppose f flow saturates all edges out of s
We decompose f into paths. The flow decomposition tells us that f is a weighted sum of paths where the weights are integers. We are going to split them into paths of unit flow. Each of those paths is going to represent one game being played between a pair of teams and one of those teams winning the game.

The total number of paths that go through an edge of the team (i.e the edge $t_i \to t$) is the total number of games that team wins, so the capacities on the edges from $t_i \to t$ imply that no team overtakes team n. This means team n can come out on top, if all of wins/losses happens as given by the flow.

($\Leftarrow$) For the other direction, we are going to build a flow by adding one unit of flow along this path

$$s \longrightarrow g_{ij} \longrightarrow t_i \longrightarrow t$$

every time team i beats team j.

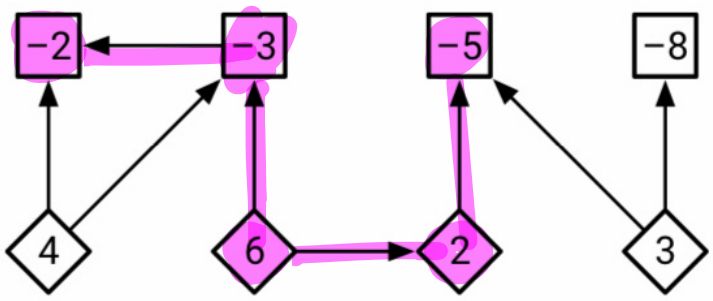
So, suppose there is some series of outcomes so that team n can come out on top. Each time t_i beats t_j we are going to push one unit of flow through the above path. This builds a flow such that every game is played which means that all edges $s \rightarrow g_{ij}$ are saturated and team n won implies that no edge $t_j \rightarrow t$ overflows, so all the capacities are also met. So, this is a valid flow through the network.

The running time of this algorithm is $O(n^2 \cdot n^2) = O(n^4)$ time

We can also reduce the above problem directly to bipartite matching by making copies of all the nodes, e.g. Baltimore_1 ... Baltimore_5 corresponding to the scenario where Baltimore wins 1 or 2 or 3 or 4 or 5 games among the games it has left. [Exercise – work out the details] But the size of the graph might increase if you do this, so this may not be faster.

Project Selection / Open-pit mining

Imagine that we are given as input a sequence of n projects that are arranged in some sort of DAG where the edges of the graph represents dependencies. For example, if there is an edge $u \rightarrow v$, then it means that u depends on v for instance, v needs to be completed before u can start. Each of these projects has a numerical value attached to it which you can think of as profit.



For example, if we select the highlighted jobs, the total profit is $-\$2$.
If we add the job ⟨4⟩, then the total profit is $\$2$.
So, the Question is:

Which jobs should we choose to maximize the total profit?

So, the output is a subset S of projects that satisfies

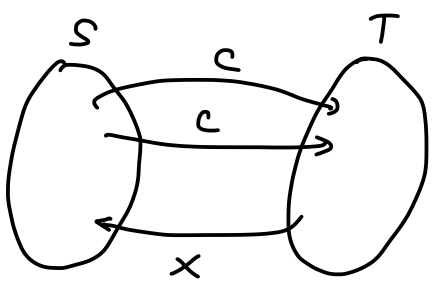
① Downwards closed $(u \in S, u \to v \in E \Rightarrow v \in S)$

② $\max \sum_{v \in S} \$(v)$

One way to think about this is that we are trying to partition the projects into two categories S & $T - S$ being the projects we select and T being the projects we turn down subject to some optimization constraints.

Recall that such a partition is called a cut. We want to find the best possible cut in this graph but we know how to do minimum cuts.

The way minimum cut is defined we only look at the capacity of edges going from S to T but we ignore the edges going from T to S



We are trying to find the cut $S \cup T$ s.t.

$\sum_{u \in S} \sum_{v \in T} c(u \to v)$ is minimized

Now, there are two issues with formulating it as a minimum cut problem

[1] There do not seem to be any capacities associated with the edges in our original graph

[2] Project selection is a maximization problem

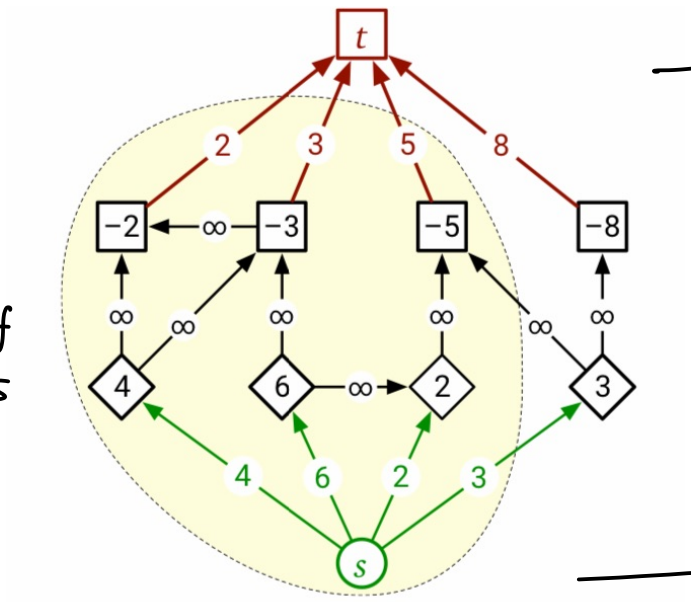
So, we need to transform the values on the vertices to capacities on edges and turn the maximization into a minimization problem.

Suppose we could do all the jobs that gave us money but none of the jobs that cost money. This is not possible but we would like to get as close to this as possible. In other words, let us define $P = \sum_{\substack{v \in V \\ \$(v) \geq 0}} \$(v)$.

Then, we want to minimize $P - \sum_{v \in S} \$(v)$. ← This takes care of the minimization vs maximization issue

How about the edge capacities ?

We will show this via an example what this construction looks like :



Capacity of this cut is

⟶ We add a vertex t and an edge $u \to t$ iff $\$(u) < 0$ with capacity $-\$(u)$

} All other edges have capacity ∞

⟶ We add a vertex s and an edge $s \to v$ iff $\$(v) \geq 0$ with capacity $\$(v)$

Consider any set S above that is downward closed, then the capacity of this cut is finite and vice-versa. This is because all the ∞ edges are only coming in and not going out. So,

S downward closed $\iff$ Capacity of S, T is finite

If we choose a cut that has an ∞ edge going out, it can not be the minimum cut and also it is not downward closed.

If we do not take any jobs, then the capacity of the cut is P so, the profit is zero.

So, we want to find the minimum cut above which gives us the maximum profit.

To formalize the above intuition Let us define

$$\text{cost}(S) = \sum_{\substack{u \in S \\ \$(u) < 0}} -\$(u) = \sum_{u \in S} c(u \to t)$$

$$\text{income}(S) = \sum_{\substack{v \in S \\ \$(v) > 0}} \$(v) = \sum_{v \in S} c(s \to v)$$

$$\text{Then, } P = \text{income}(V) = \sum_{v} c(s \to v) = \text{income}(S) + \text{income}(T)$$

So, $profit(S) = income(S) - cost(S)$

$$\Longrightarrow \quad \|S, T\| = income(T) + cost(S)$$
$$= P - profit(S)$$

So, to maximize the profit, we want to find the minimum cut.

Running time is $O(VE)$.