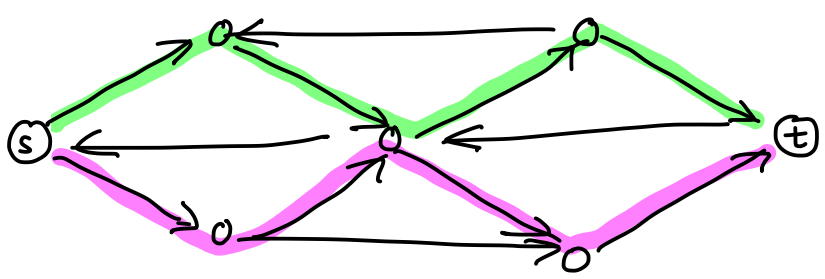
LECTURE 14 (October 16th)

APPLICATIONS OF MAX FLOW

Edge Disjoint Paths Problem

We are given a directed graph $G = (V, E)$, two vertices s and t and we want to find the largest number of paths from s to t where every edge in the graph appears in at most one of the paths



For example, the highlighted paths above are edge-disjoint

This is a fundamental problem that occurs in many real-world scenarios such as how few Google Streetview cars to send out to maximize street coverage.

The following is the algorithm to solve this problem:

- Assign capacity 1 to every edge
- Compute max (s,t)-flow f^*
- Value of the max flow is the number of edge disjoint paths

Why does this work? Since every capacity is an integer, the maximum flow value is an integer. This is the flow that Ford-Fulkerson computes. Since flow is an integer flow, every edge must have either 0 or 1 unit of flow through it. The edges with 1 unit of flow comprise the edge disjoint paths. In particular, if we leave at how many paths leave s, that will be the same as the net flow. The paths are edge disjoint because if two paths share an edge then that would be two units of flow which is not possible because the capacity is 1.

This would just give the maximum number of edge disjoint paths. If we want the paths themselves, we can add one more step

- Compute the flow decomposition of f^* into paths

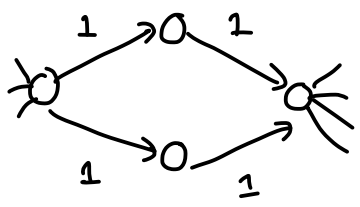
The running time of Ford-Fulkerson is $O(|E| \cdot |f^*|)$

But if you think about it $|f^*| \leq |V|$ since the maximum number of edge disjoint paths is at most the maximum out degree of any vertex which is atmost $|V|$.

So, the running time of Ford-Fulkerson here is $O(|V| \cdot |E|)$ which is the same as Orlin's algorithm. Also, computing the flow decomposition takes $O(|V| \cdot |E|)$ time

What if the graph is undirected ? Replace every undirected edge with two directed edges

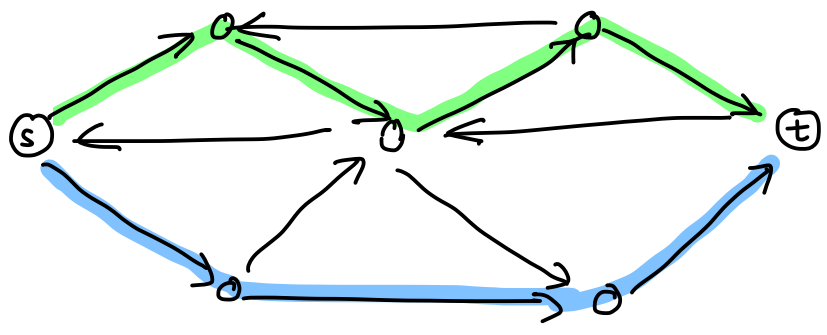
and run Ford-Fulkerson. It is possible that Ford-Fulkerson will compute a flow using both these edges but one can do a sanity check and remove such cycles in the flow decomposition. Alternatively, one can split the two directed edges further and then compute the flow.



This is still $O(|V| \cdot |E|)$ time.

Vertex disjoint paths

What if we want the maximum number of paths that do not share any vertices ?



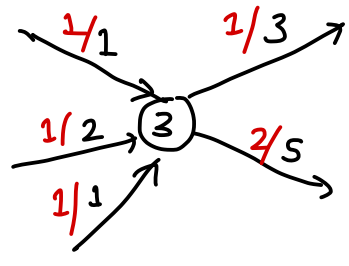
Here what we would want is also a capacity on the vertices somehow. In other words, a feasible flow f satisfies

$$0 \leq f(e) \leq c(e) \quad \text{for every edge } e$$

but also $$\sum_{u \to v} f(u \to v) \leq c(v) \quad \text{for every vertex } v$$

↳ By conservation, this is also the flow coming out of v

As a concrete example, this flow saturates the vertex with capacity 3 below



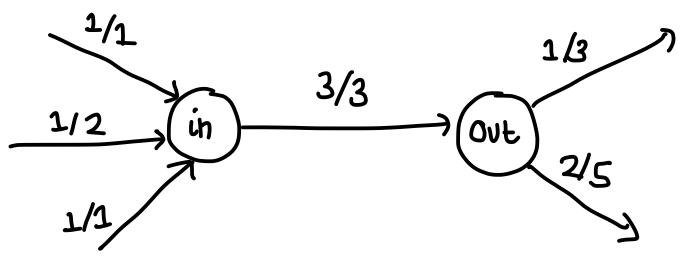
Now, there are two possibilities to design an algorithm here

① Change the flow algorithm to handle this case

This is not recommended! One has to carefully think about residual graph subject to vertex capacities and other details and make sure it works. Hint: this gets very difficult!

② Change the graph/flow network and solve the regular flow problem on this new graph

Here we will somehow construct a graph where each vertex capacity is represented as capacity of an edge. For example, the above vertex will have a new edge with capacity 3.



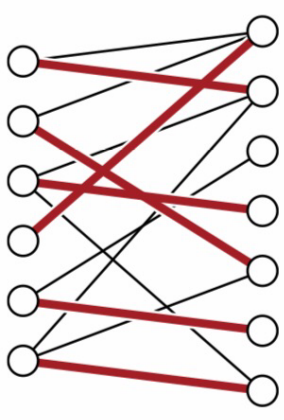
We can do this for every vertex in $O(|V|+|E|)$ time.

Now, we can just use the standard flow algorithm to find a flow in this new graph which easily gives us a flow in the original graph as well. We can also decompose the flow in the new graphs into paths and cycles and obtain a flow decomposition for the flow in the original graph as well. This also takes $O(|V|\cdot|E|)$ time.

Now, suppose we want at most two paths sharing an edge and at most three paths sharing a vertex. Exercise!

Bipartite Matching

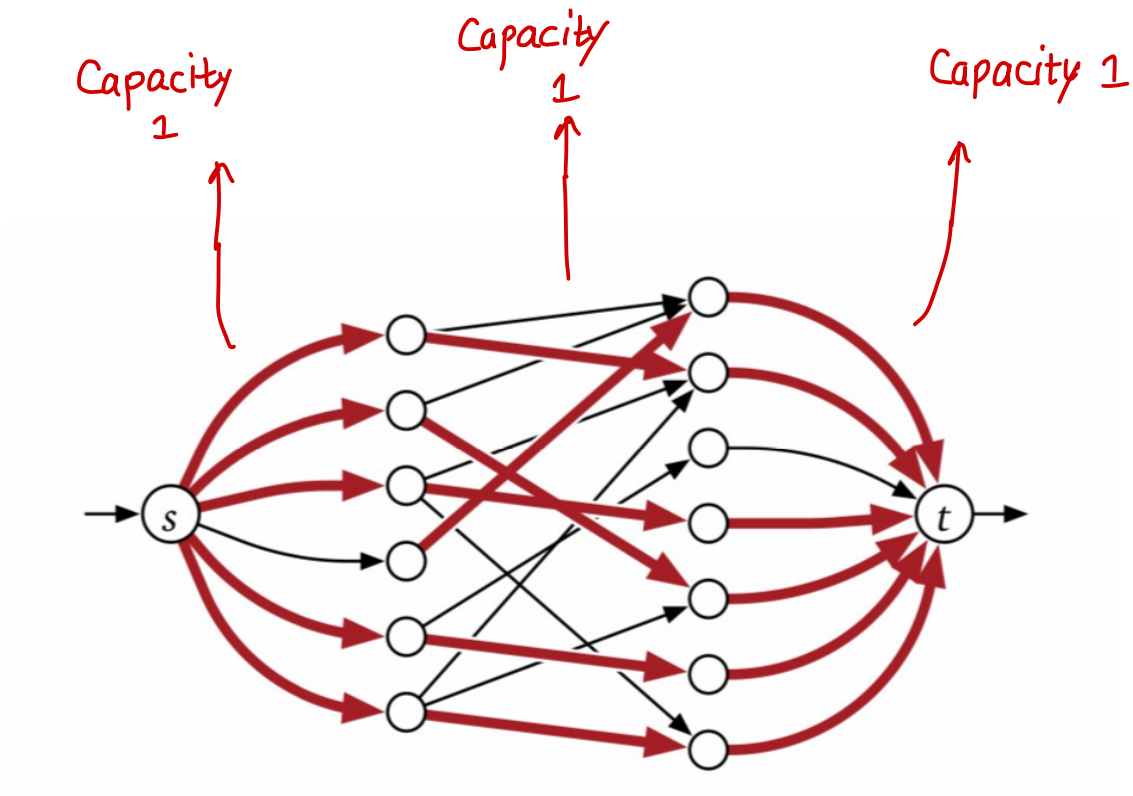
A bipartite graph is a graph where vertices can be divided into two parts L and R such that edges only go between L & R



A matching is a set of edges in the graph s.t. no two edges share any vertex. For example, the red edges highlighted above.

The goal in this problem is to find a maximum bipartite matching, i.e., a matching with the maximum number of edges.

How do we solve this problem? Since, we want to find edges not sharing vertices, we can consider the following graph, by adding a source s and a sink t, and edges from s to each vertex in L and from each vertex in R to t and directing the edges from L to R. Running time is $O(|V| \cdot |E|)$.

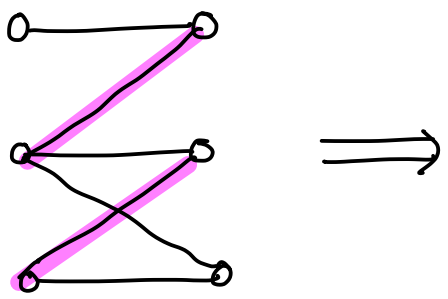


Vertex and Edge disjoint paths in this graph are the same! So, if run the flow algorithm on this graph, we get the size of the maximum matching. To get the matching, we compute the flow decomposition and remove the edges that touch s or t.

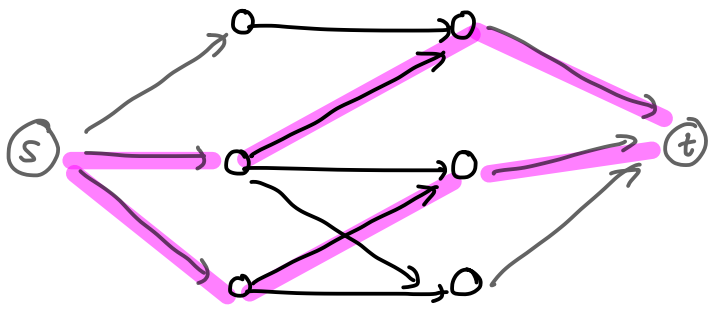
Another way of finding the matching is to see that the max flow is integral and the matching is the set of edges with flow value 1.

A useful way to think about what Ford-Fulkerson is doing here is the following.

Consider the following graph and a matching in it

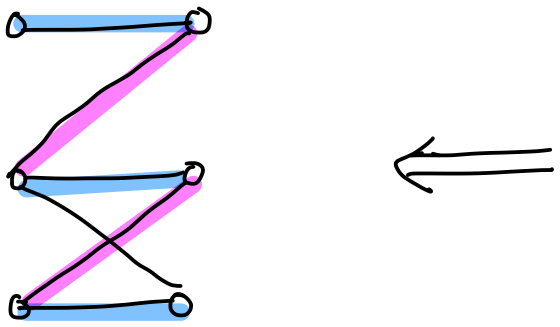


The flow on the graph we constructed looks like

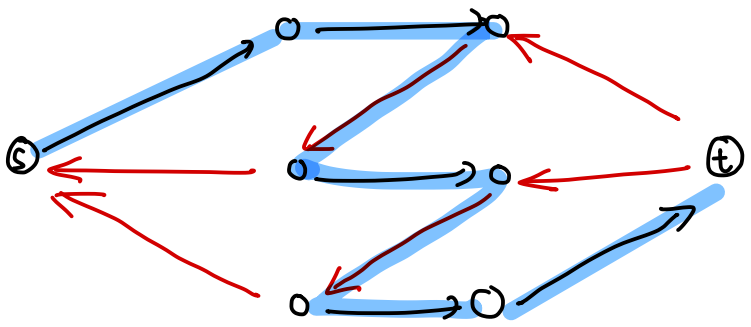


where the pink edges have flow value one and all other have flow value zero

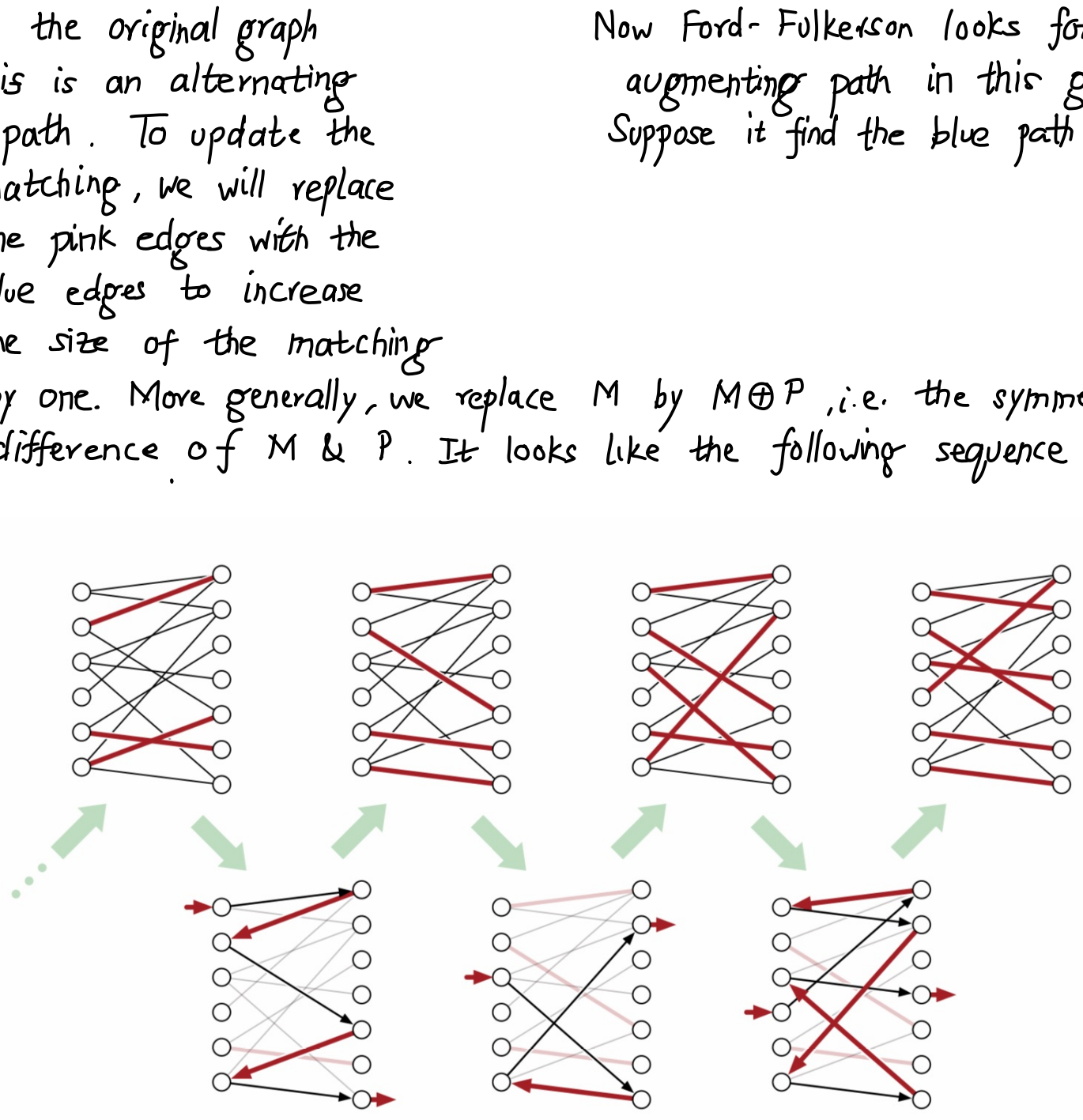
Residual graph of the flow



In the original graph this is an alternating path. To update the matching, we will replace the pink edges with the blue edges to increase the size of the matching by one. More generally, we replace M by $M \oplus P$, i.e. the symmetric difference of M & P. It looks like the following sequence



Now Ford-Fulkerson looks for an augmenting path in this graph. Suppose it find the blue path

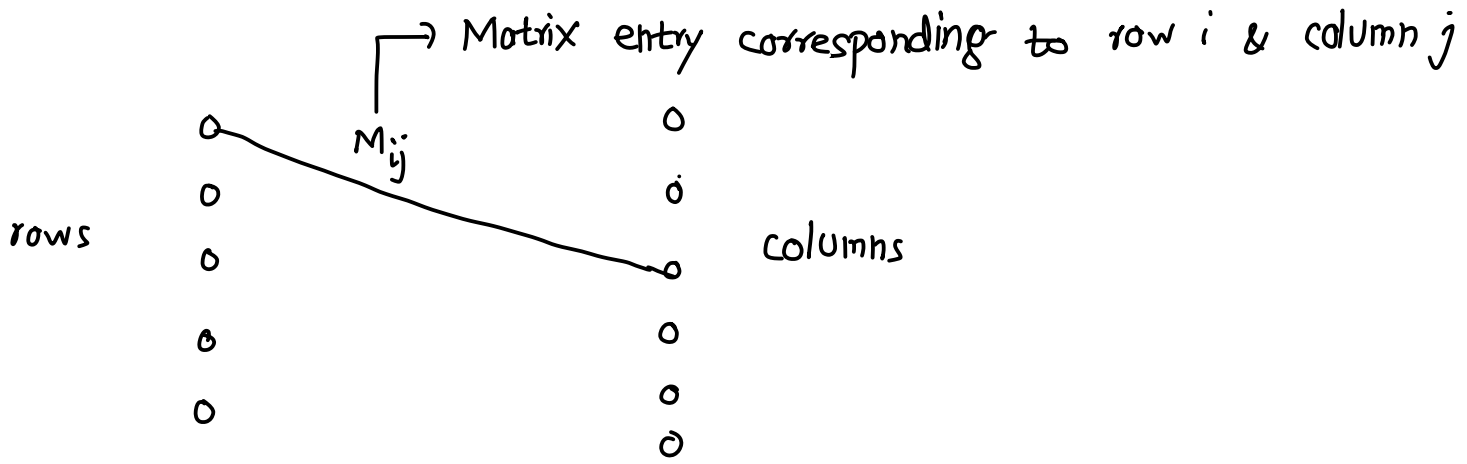


So, we don't need to explicitly compute the residual graph but only the alternating paths. [How would you find alternating paths?]

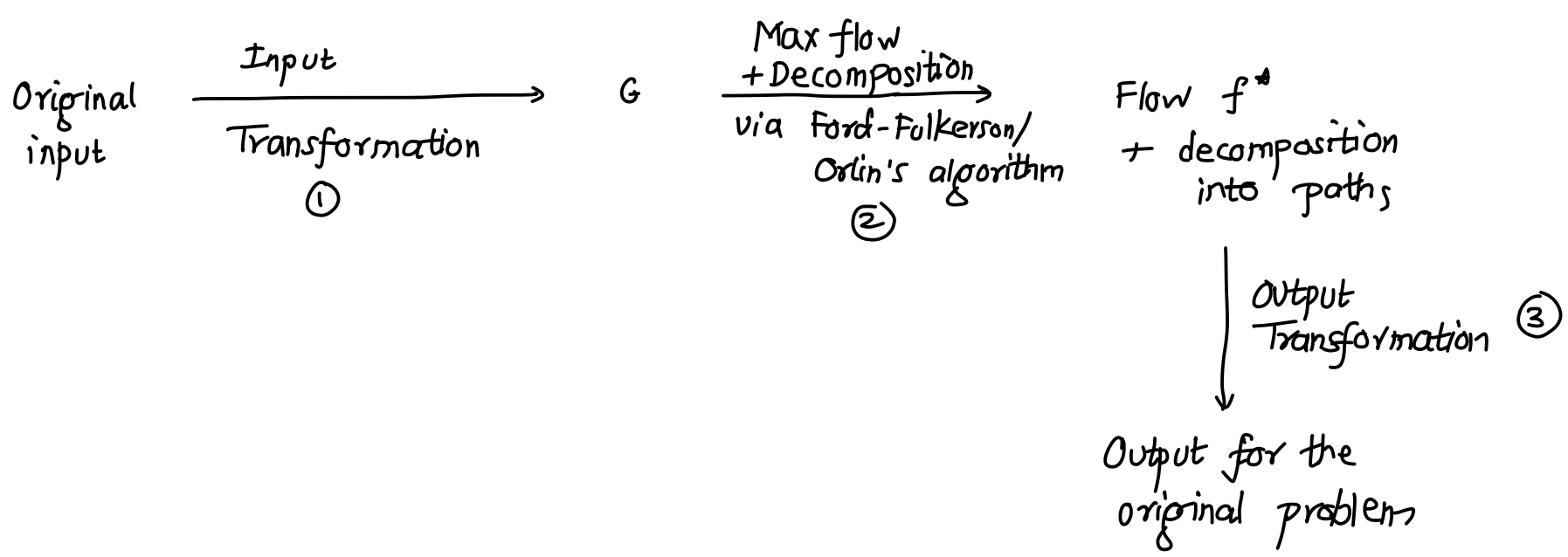
This notion of alternating paths was published by Berge in 1957 in what is considered as the first graph theory textbook, predating Ford-Fulkerson. But infact the notion goes back way earlier to Jacobi in 1836 who was trying to solve a problem about matrices originating in the theory of differential equations.

The matrix problem is the following :

Permute the rows and columns of a matrix to maximize the sum of values along the main diagonal. This is exactly the bipartite matching problem with weights. Jacobi gave an algorithm in narrative Latin to solve this using alternating paths ! This is also a polynomial time algorithm which was only invented in 1960s !



So, to solve a problem with max-flows, the standard pipeline is

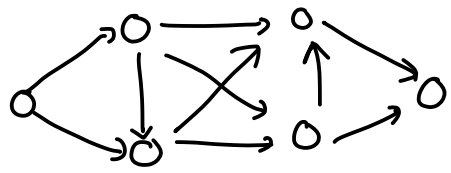


You need to describe ① & ③, since ② is a black-box
You also need to describe time analysis in terms of the original input as well as proof of correctness, i.e., why the flow output gives the solution to the original problem.

Typically what we are looking for is ways of relating the features in our solutions to paths in the graphs, e.g. in the matching problem we related it to paths.

Let's see another example of a flow problem

Disjoint Path Cover



The input is a DAG $G = (V, E)$
Previously we saw problems related to finding paths that don't collide in either vertices or edges. Such problems are called packing problems.

In this problem, we want a covering problem — Find the smallest collection of disjoint paths that cover all the vertices. As an example application of this problem, consider the following:

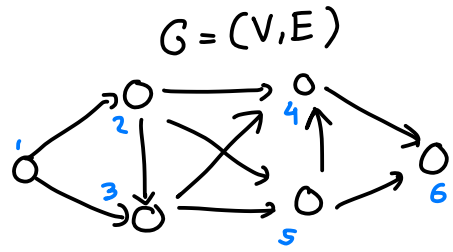
Given n envelopes with height h_i & width w_i, we can put i inside j if $h_i < h_j$ & $w_i < w_j$. We want to nest envelopes into as few stacks as possible. We can create a DAG where $i \to j$ occurs if envelope i can be nested inside envelope j. A path is a sequence of envelopes that can be nested and we want to find the smallest collection of such paths to minimize the number of stacks.

This is infact a matching problem. We are trying to assign to each envelope what envelope we put it into, i.e., for each vertex we are trying to assign a successor & the number of paths is the number of vertices that don't have a successor.

So, minimizing the number of vertices that do not have a successor is the same as maximizing the number of vertices that do have a successor, i.e. we are trying to MATCH nodes in G to their successor.

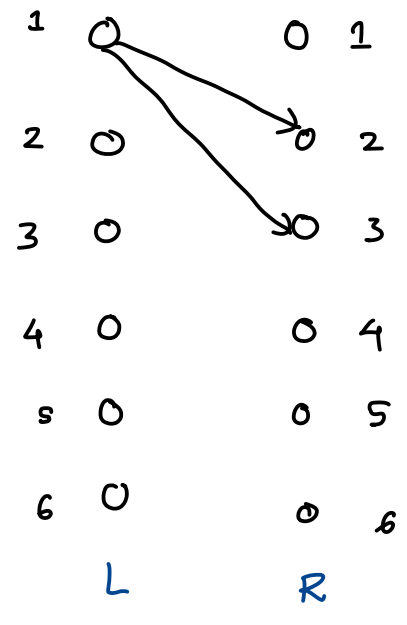
$G = (V, E)$

How do we set it up? We build a bipartite graph



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

where L = copy of V
R = copy of V
and $E' = (u \to v)$ if $u \to v$ is edge in G
($u \in L$, $v \in R$)

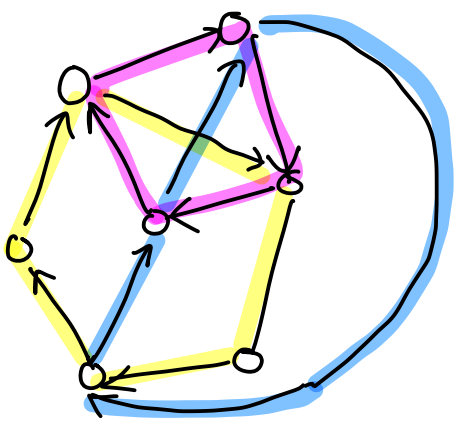


We want a max matching in G' which will give us the maximum number of nodes with a successor which gives us the minimum number of disjoint paths or envelopes needed via # paths = # V − # matching

Running time is $O(VE) = O(n^3)$ if there are n envelopes

Another example of this problem is the 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!

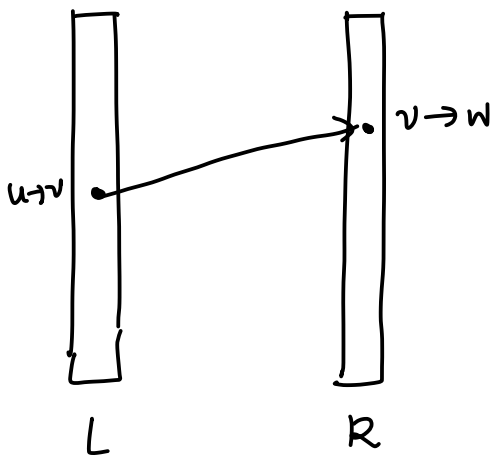


How can we turn this into a matching problem?

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

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