So, we have a pseudo-poly time algo for Max Flow.
How do we design a polynomial time algorithm?

One algo that achieves polynomial running time is Edmonds-Karp

```
MaxFlowEK(G):
    Set f[e] = 0  ∀ e ∈ E

    While P = findShortestPath(s, t, ResidualGraph(G, f)):
        f = augment(f, P)
        UpdateResidual(G, f)

    return f
```

- So, what is the difference between FF and EK?

    - findPath → findShortestPath
- the only difference is the selection of the augmenting path
- Why does this lead to a different runtime analysis?
    - There is an explicit bound (mn) on the total # of bottleneck edges as EK runs. This limits the total number of iterations of the while loop and the overall run time.

Theorem: EK makes at most mn iterations of the while loop

Proof: Consider laying out the vertices of G in layers according to a BFS from s, and let t be at level d.

Keeping the layout fixed, consider the sequence of paths found in the resulting residual graph. If a shortest path uses only forward edges, each iteration will cause at least 1 forward edge to be saturated and removed from G_f, and only backward edges will be added.

This means that d does not decrease, and as long as d has not increased (so that only forward edges are being used), at least one forward edge is eliminated per iteration.

Forward edges will be removed at most m times before either (1) $d = \infty$ (graph is disconnected and EK terminates) or (2) A path with a non-forward edge is used (so that d increases by at least 1). We can re-layout G and apply the same argument again. This shows that the s-t distance of the selected path never decreases. Further, it increases by at least 1 every (at most) m iterations. The minimum path length cannot increase beyond n. This implies we have $\leq mn$ iterations of the while loop.

Running Time:

How long does each iteration take?

- We can identify the shortest path using BFS ($O(m+n)$) time. However, since we assume every vertex has at least one incident edge, $n \le 2m$ and $O(m+n) = O(m)$

- Given the path P, augment takes $O(m)$ time
- Thus, each iteration takes $O(m)$ time

EK takes $O(m \cdot mn) = O(m^2 n)$ time to find a maximum flow.

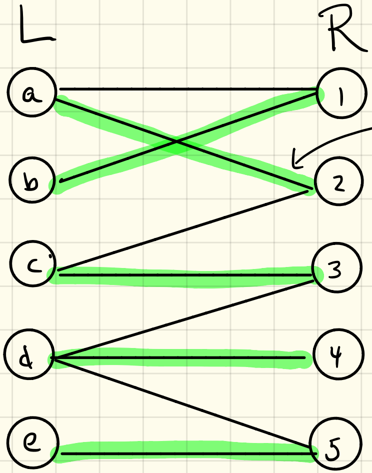
Note: This algorithm's runtime depends only on the specification of the graph and not the edge capacities. It is what we call a "strongly-polynomial" algorithm.

Now, let's look at how to use network flow to solve other interesting problems.

- Bipartite Matching

E.g. Set of people (L) and jobs (R)

- Each person is only qualified for some subset of jobs
- Each job is done by at most 1 person
- Can model this as a bipartite graph



Here, edges highlighted in green constitute a maximum matching.

- a matching gives an assignment of people to jobs
- Want to accomplish as many tasks as possible
- Each person is assigned 1 job
- So we want a maximum matching (containing as many edges as possible)

Problem: Maximum Bipartite Matching

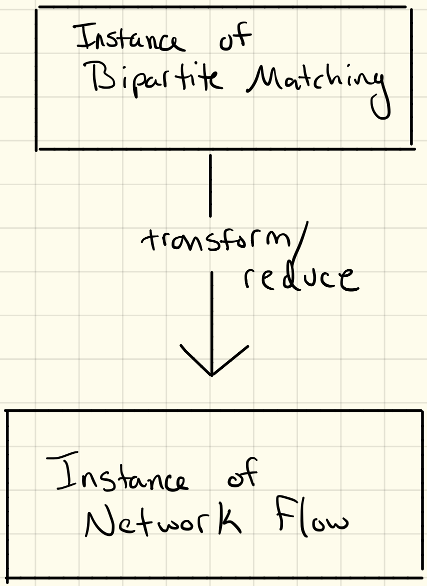
Given a bipartite graph $G = (A \cup B, E)$, find a matching $M \subseteq E$, such that any node appears in <u>at most one</u> edge in M, and such that M is as large as possible.

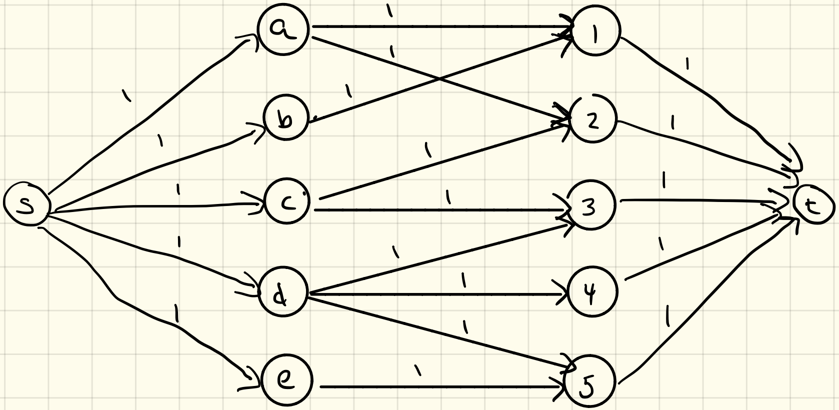
<u>Note:</u>

- We are given the bipartition; no need to find it
- S is a <u>perfect matching</u> if every vertex is matched
- Maximum is not maximal $\Rightarrow$ a greedy algo will give a <u>maximal</u> matching.

Key Concept: Reduction

- Given an instance of Maximum Bipartite Matching
- Create an instance of network flow such that the solution of the flow problem can be easily used to find the solution to the bipartite matching problem.





Transformation:

1) Given bipartite graph $G = (A \cup B, E)$, direct edges from A to B
2) Add new vertices s and t
3) Add an edge from s to every vertex in A
4) Add an edge from every vertex in B to t
5) Make all edge capacities 1
6) Solve Max Flow on this graph G'

Claim: The edges used in the Max Flow will correspond to the largest possible matching.

Important Notes:

- Since capacities are integers, flows will be integral
- Since capacities are all 1, every edge is used <u>completely</u> or <u>not at all</u>
- If M is the set of edges from A to B we use then
  1) M is a matching
  2) M is the largest possible matching

Theorem: The $A \to B$ edges of our flow constitute a maximum bipartite matching in G

Lemma: M is a matching

Proof: We can choose at most 1 edge leaving any node in A and at most 1 edge entering any node in B. Otherwise, we would not satisfy the balance constraints, and our solution wouldn't be a valid flow!

Lemma: M is of maximum size

Proof: If there is a matching of k edges, there is a flow of value k:

- f has 1 unit along each of the k edges, ≤ 1 unit leaves and enters every node but s & t.

If there is a flow of value k, there is a matching with k edges
- We find a maximum flow with (say) k edges.
- This corresponds to a matching of k edges
- If there were a matching with $k' > k$ edges, we would have found a flow with a value $> k$, contradicting that f was maximum.
- Hence, M is maximum.

Running Time?

- Consider the FF algo. The runtime is bounded by $O(m'C)$ where m' is the # of edges and

$$C = \sum_{e \text{ out of } s} c_e$$

- $C = |A| = n$
- the # of edges in G' is equal to the number of edges in G (m) plus $2n$

- So, the running time is $O((m+2n)n) = O(mn + n^2) = O(mn)$

This leads immediately to:

Theorem: We can find a maximum bipartite matching on a graph with n vertices and m edges in $O(mn)$ time.