CS4102 Algorithms

Fall 2021 – Floryan and Horton

Module 10

Roadmap: Where We're Going and Why

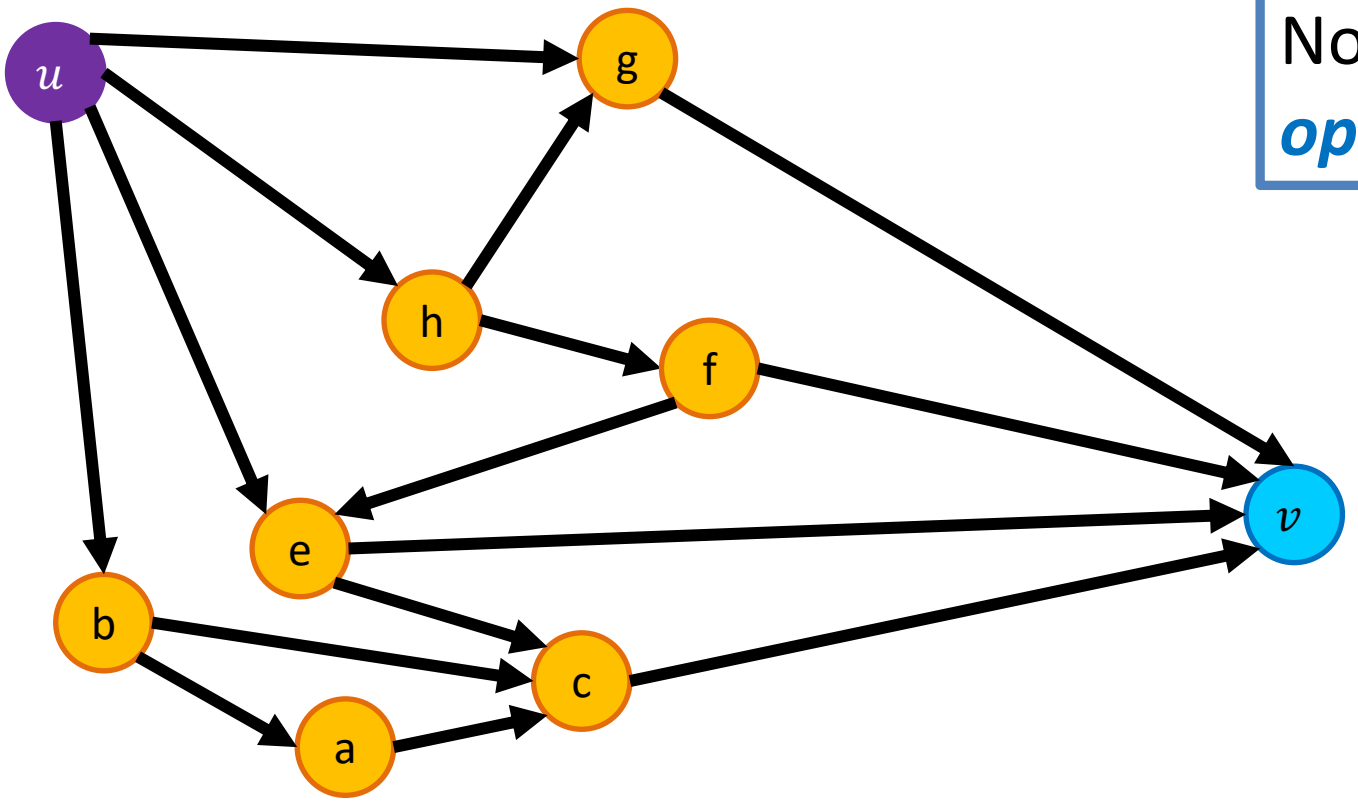
- ***Reductions*** between problems
  - Why? Can be a practical way of solving a new problem
  - Also: A proof about one problem's complexity can be applied to another
  - Formal definition of a reduction
- Examples
  - Bipartite graphs, matching
  - Vertex cover and independent set

Using One Solution to Solve Something Else

- Sometimes we can solve a “new” problem using a solution to another problem
  - We need to “re-cast” the “new” problem as an ***instance*** of the other problem
  - We may need to relate how the answer found for the other problem gives the answer for the “new” problem
- Some examples coming in this lecture:
  - We’ll see how to solve ***edge-disjoint path*** problem.
    Use that to solve ***vertex-disjoint path*** problem.
  - We know how to find ***max network flow***.
    Use that to solve ***bi-partite matching***.

Edge-Disjoint Paths

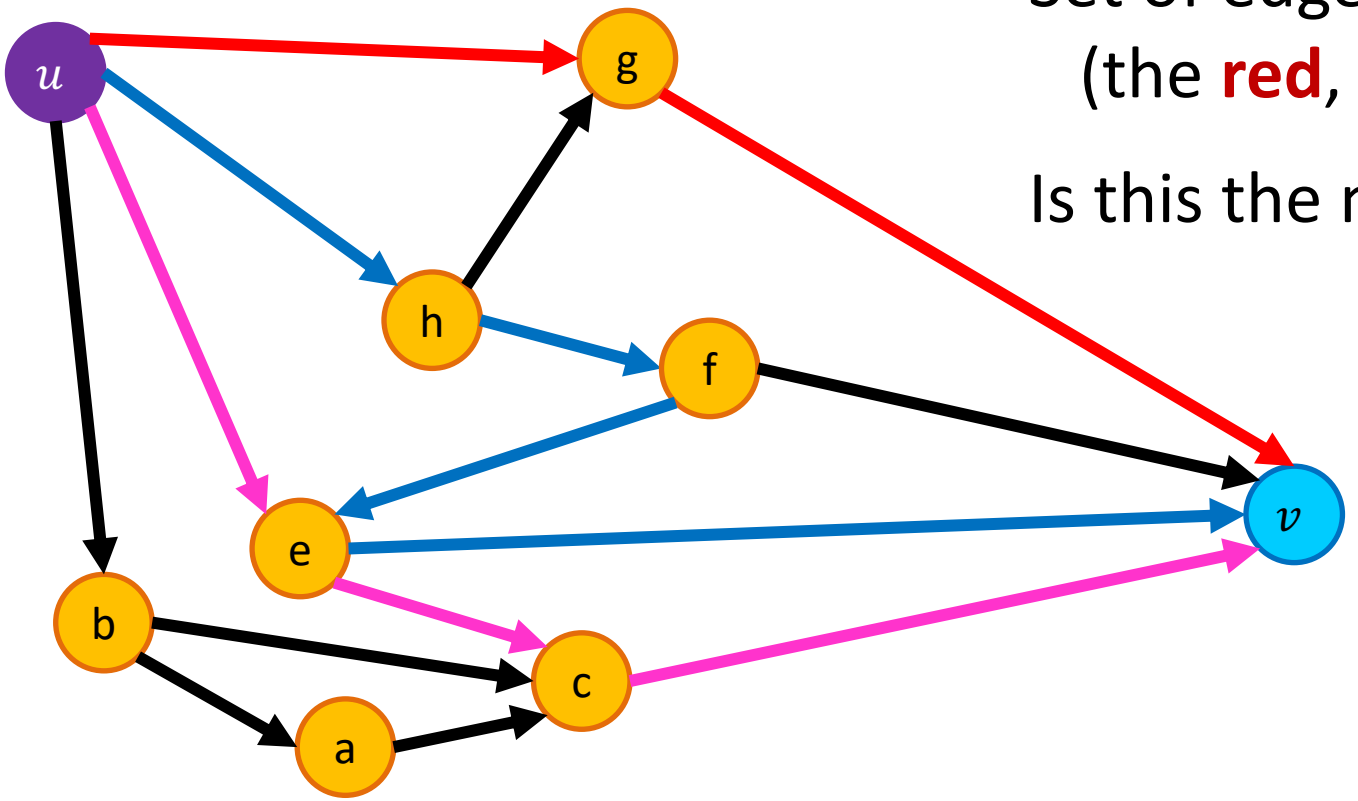
Given a graph $G = (V, E)$, a start node u and a destination node v, give the maximum number of paths from u to v which share no edges



Note this is an ***optimization problem.***

Edge-Disjoint Paths

Given a graph $G = (V, E)$, a start node u and a destination node v, give the maximum number of paths from u to v which share no edges

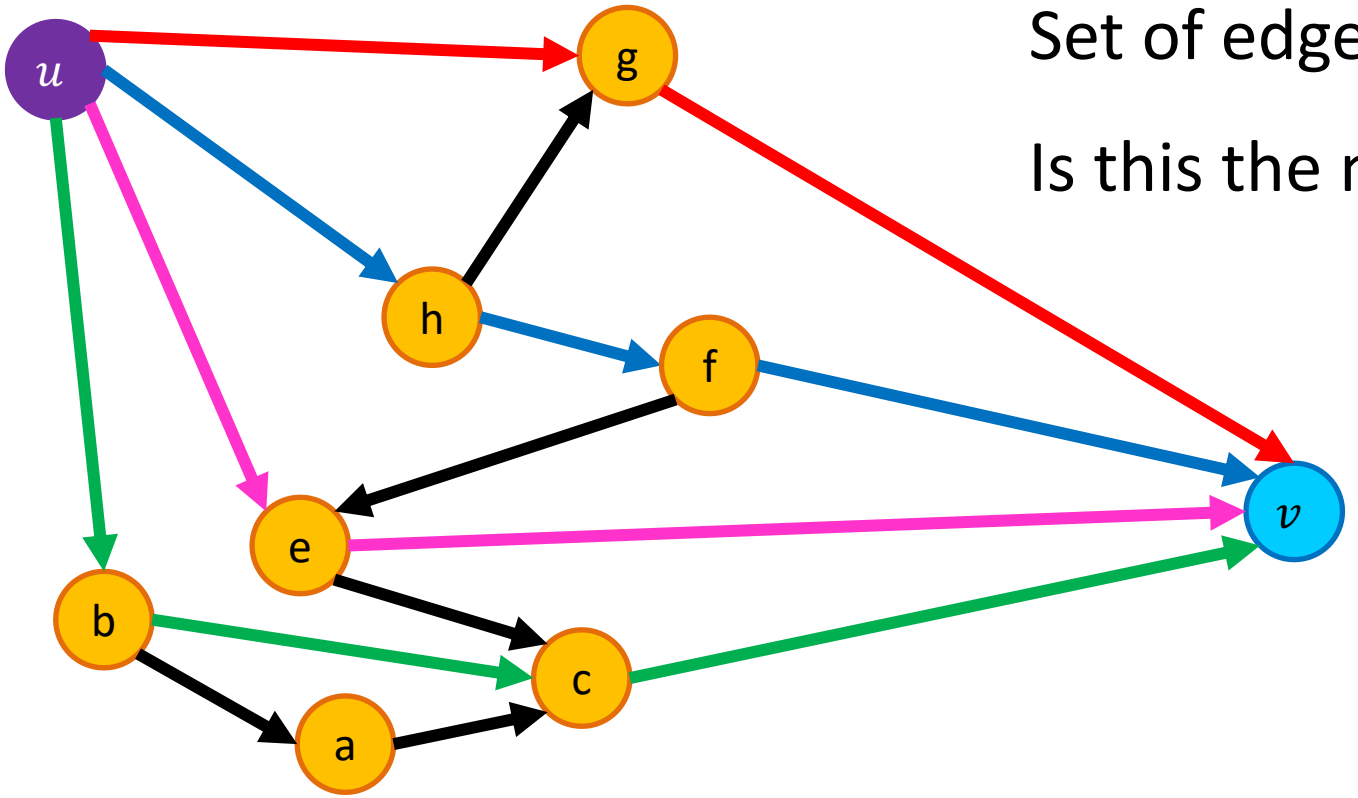


Set of edge-disjoint paths of size 3 (the **red**, **blue**, **magenta** paths)

Is this the max number?

Edge-Disjoint Paths

Given a graph $G = (V, E)$, a start node u and a destination node v, give the maximum number of paths from u to v which share no edges



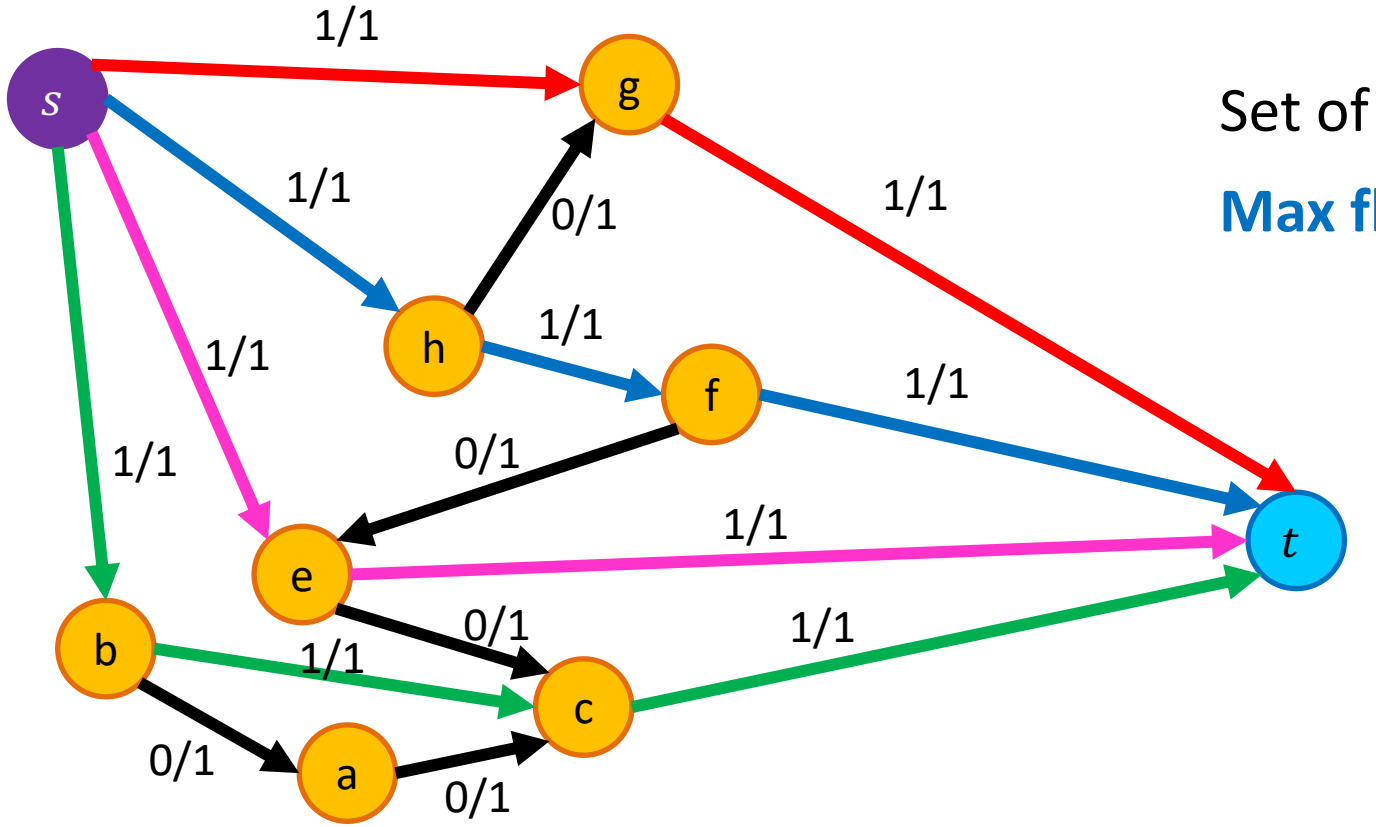
Set of edge-disjoint paths of size 4

Is this the max number?

Edge-Disjoint Paths Algorithm

Use a problem we know how to solve, ***max network flow***, to solve this!

Make u and v the source and sink, give each edge capacity 1, find the max flow.



Set of edge-disjoint paths of size 4

Max flow = 4

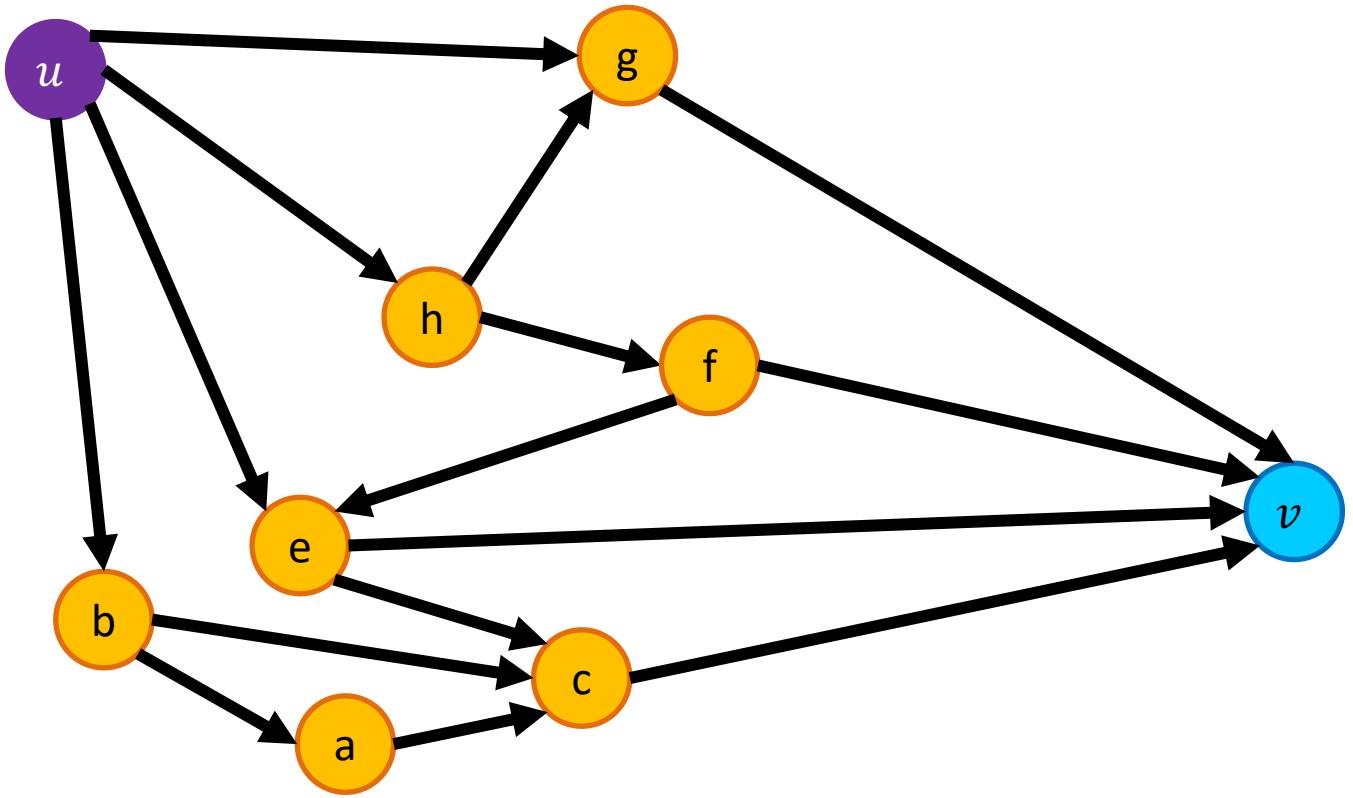
Why does this work?
We need to be able to make a valid argument that it always does.

What's the situation?

- Given an input I_1 for the ***max network flow problem*** (graph G with edge capacities), we can find the max flow for that input
- Given an input I_2 for ***edge-disjoint path problem***, we can:
  - Convert that input I_2 to make a valid input I_1 for ***network flow problem***, by using same graph G but adding capacity=1 for each edge
  - Solve ***max network flow problem*** for I_1 and get result R_1
  - Use R_1 to give the solution R_2 for ***edge-disjoint path*** for input I_2
    - In this case, **|f|** = **the number of paths**
- Next, let's solve another problem using our new *edge-disjoint path* solution

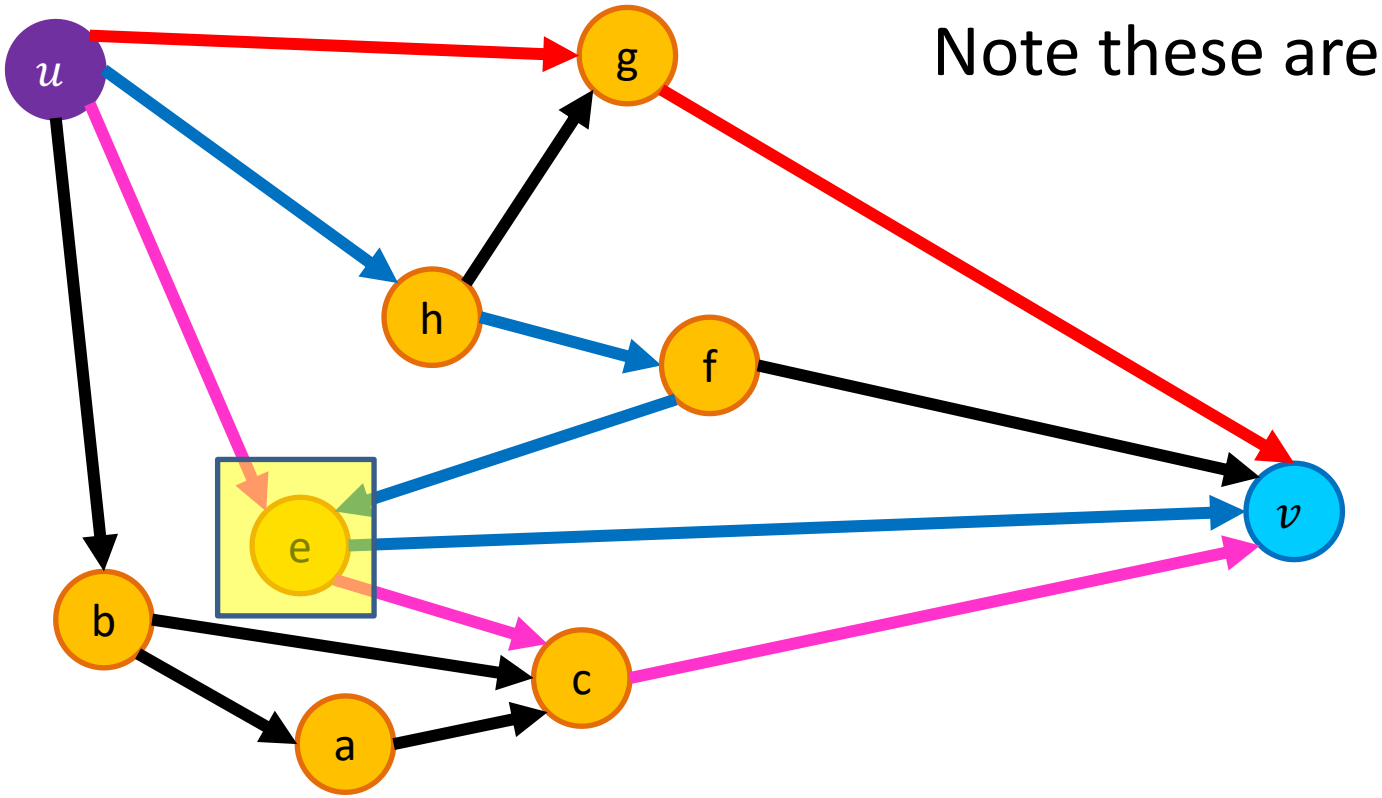
Vertex-Disjoint Paths

Given a graph $G = (V, E)$, a start node u and a destination node v, give the maximum number of paths from u to v which share no vertices



Vertex-Disjoint Paths

Given a graph $G = (V, E)$, a start node u and a destination node v, give the maximum number of paths from u to v which share no vertices



This shows 3 edge-disjoint paths.

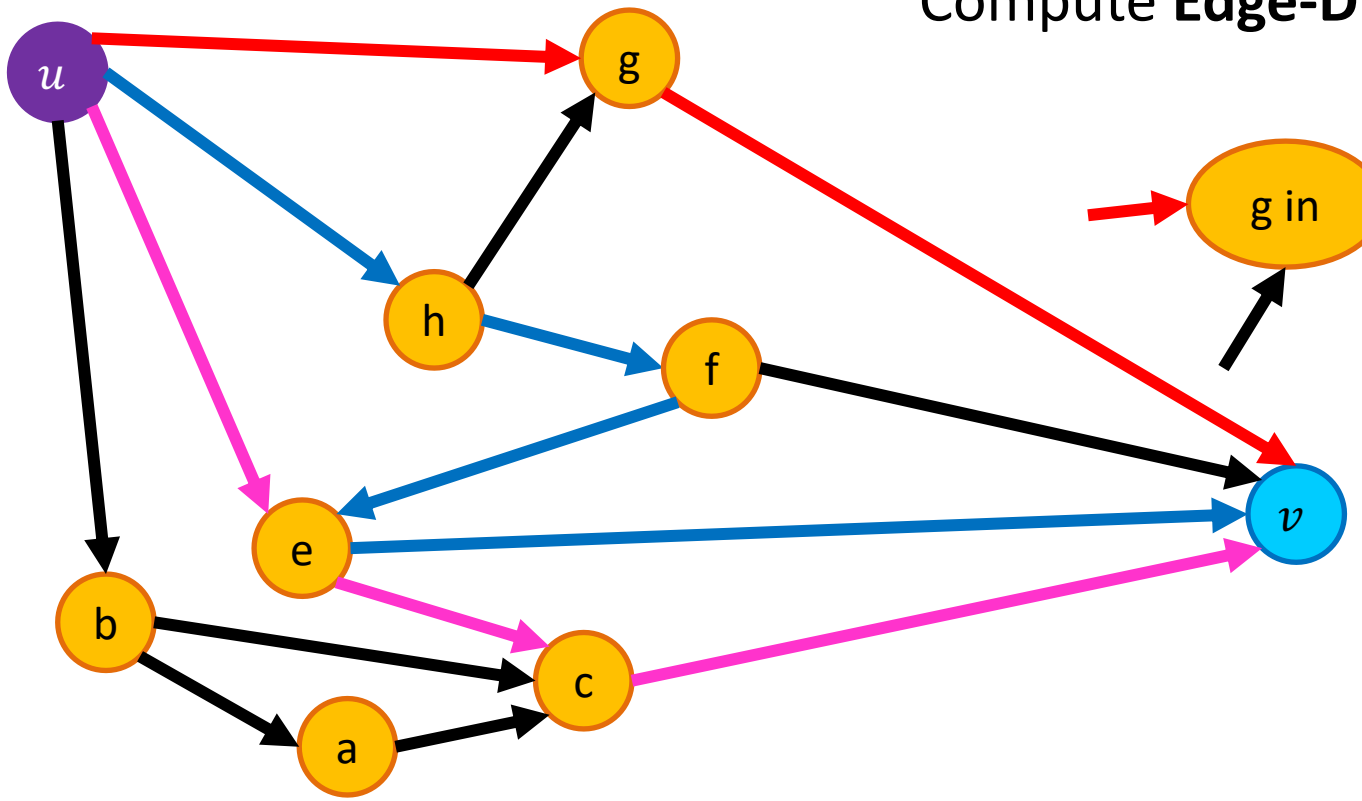
Note these aren't vertex-disjoint paths!

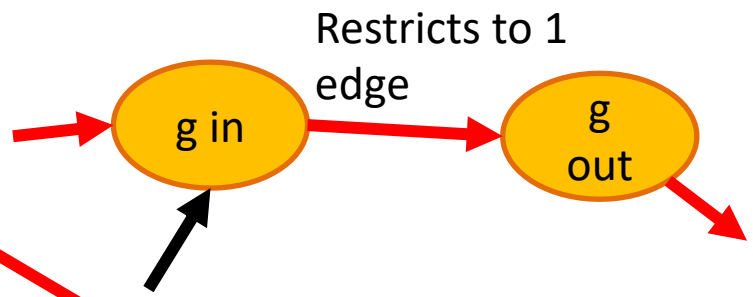
Vertex-Disjoint Paths Algorithm

Idea: Convert an instance of the vertex-disjoint paths problem into an instance of edge-disjoint paths

Make two copies of each node, one connected to incoming edges, the other to outgoing edges

Compute **Edge-Disjoint Paths** on new graph





Why does this work?
We need to be able to make a valid argument that it always does.

What's the situation now?

- Given an input I_1 for the ***max network flow problem*** (graph G with edge capacities), we can find max flow for that input
- Given an input I_2 for ***edge-disjoint path problem***, we can:
  - Convert that input I_2 to make a valid input I_1 for ***network flow problem***, and solve that to find **number of edge-disjoint paths**
- Given an input I_3 for ***vertex-disjoint path problem***, we can:
  - Convert that input I_3 to make a valid input I_2 for ***edge-disjoint path problem***
  - See above! Convert I_2 to I_1 and solve ***max network flow problem***
- This chain of "problem conversions" finds lets us solve ***vertex-disjoint path problem***
  - **Time complexity?** Cost of solving max network flow plus two conversions

Reductions

(We’re about to get interested in problems that seem to require exponential time…)

Max-flow vs. Edge-disjoint paths

- These two problems are “related”
  - Here we’re saying: if you can solve ***Max-flow***, you can solve ***Edge-disjoint path***
- Alternatively, we can say that one problem ***reduces*** to the other
  - The problem of finding Edge-disjoint paths ***reduces to*** the problem of finding max-flow
  - Maybe this ***reduction*** requires some work to “convert”
    - Could be nothing or minimal
  - For these problems, the cost of the conversion is ***hopefully small (more on this in a moment).***

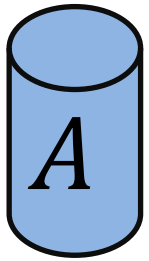
Reduction

- A ***reduction*** is a transformation of one problem into another problem
  - Edge-disjoint paths is reducible to max-flow because we can use max-flow to solve it
  - Formally, problem A is ***reducible*** to problem B if we can use a solution to B to solve A
- We're particularly interested in reductions that happen fast!
  - *Meaning the work to do the conversion is fast (how fast?)*
- If A is ***polynomial-time reducible*** to B, we denote this as:

  $A \leq_p B$
- **If the conversion takes linear time, we might say $A \leq_n B$**

In General: A Reduction

A problem we <u>don't</u> know how to solve

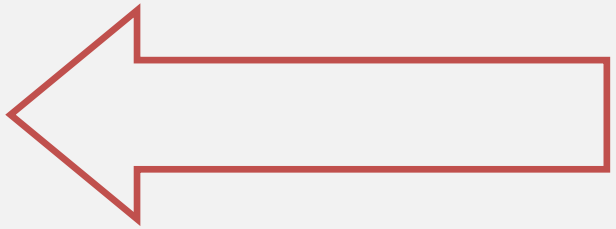


Solution for A

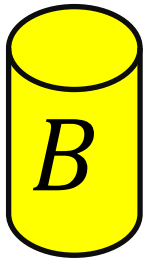
Reduction

Map Instances of problem A to Instances of B

Map Solutions of problem B to Solutions of A

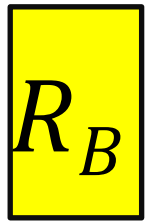


A problem we <u>do</u> know how to solve



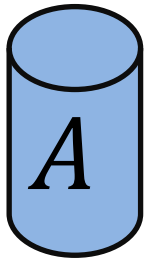
Using any Algorithm for B

Solution for B



In General: Reduction

Problem we <u>don't</u> know how to solve

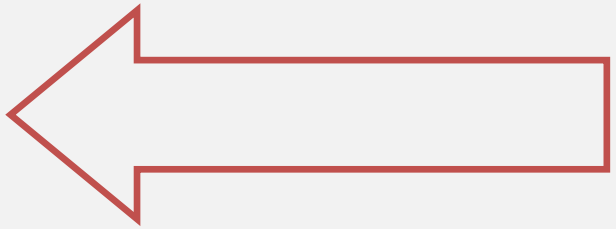


Solution for A

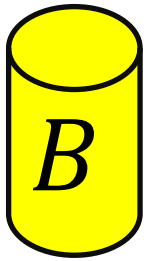
Reduction

Map Instances of problem A to Instances of B

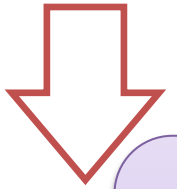
Map Solutions of problem B to Solutions of A



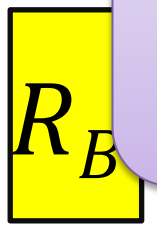
Problem we <u>do</u> know how to solve



Using any Algorithm for B



Solutio



For now: we are NOT focusing on an algorithm to solve one of these, <u>just on the reduction</u>!

Total Runtime

- The total runtime to solve A is:
  - ***Convert(A->B)***: Time it takes to convert problem A into problem B
  - ***Execute(B):*** Time it takes to execute algorithm to solve B
  - ***Solve(B->A)***: Time to convert solution of B back to solution of A

- ***Total Runtime: Convert(A->B) + Execute(B) + Solve(B->A)***

- Do you see why we want convert() and solve() to be FAST. We want the slowest part to be the actual algorithm that solves B if possible.

Relative Hard-ness

Total Runtime: Convert(A->B) + Execute(B) + Solve(B->A)

- Generally, we want execute() to be the slowest term, we then can say $A \leq_{\alpha} B$ where alpha is the runtime of convert() and solve()

- If Execute() IS the slowest part, then what does that mean about the relative difficulty of solving A versus B??
  - It means that B is as hard or harder than A. Why?
  - Because B provides an algorithm that solves A, so if algorithm to solve B gets faster, so does the algorithm that solves A.

Max-flow vs. min-cut

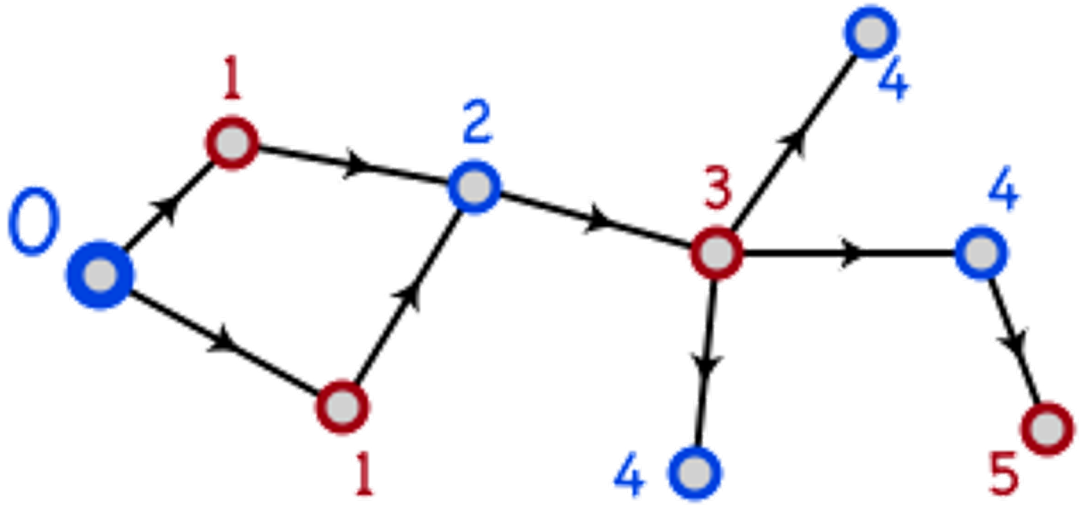
- These two problems are “equivalent”
  - Remember? ***max-flow min-cut theorem***
  - Here we’re saying: if you can solve one, you can solve the other
- Alternatively, we can say that one problem ***reduces*** to the other
  - The problem of finding min-cut ***reduces to*** the problem of finding max-flow
  - Maybe this ***reduction*** requires some work to “convert”
    - Could be nothing or minimal
  - For these problems, the cost of the conversion is ***polynomial***

Bipartite Matching

Another example of a reduction

Bipartite Graphs

- A graph is *bipartite* if node-set V can be split into sets X and Y such that every edge has one end in X and one end in Y
  - X and Y could be colored red and blue
  - Or Boolean true/false



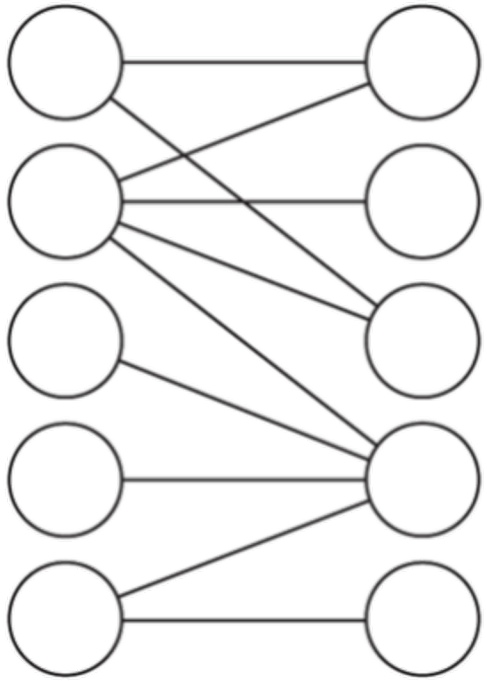
How to determine if G is bipartite?

The numbers and arrows on edges may give you a clue….

BFS or DFS, and label nodes by levels in tree.
Non-tree edge to node with same label means NOT bipartite.

Notes and assumptions

- We assume the graph is connected
  - Otherwise we will only look at each connected component individually
- A triangle cannot be bipartite
  - In fact, any graph with an odd length cycle cannot be bipartite

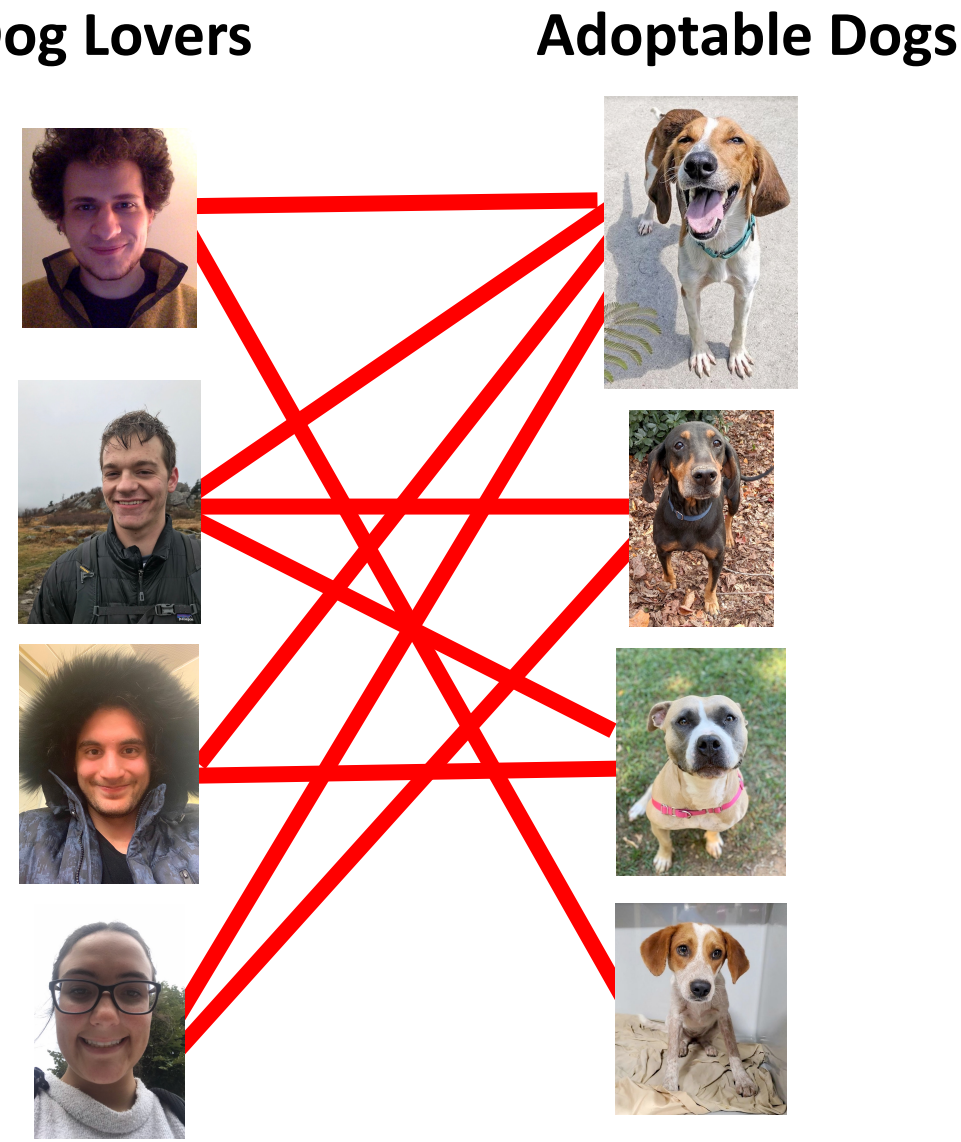


Bipartite Determination Algorithm

- Pick a starting vertex, color it red
- Color all adjacent nodes blue
  - And all nodes adjacent to that red
  - Etc.
- If you ever try coloring a red-node blue, or a blue-node red, then the graph is not bipartite

- Does this algorithm sound familiar?

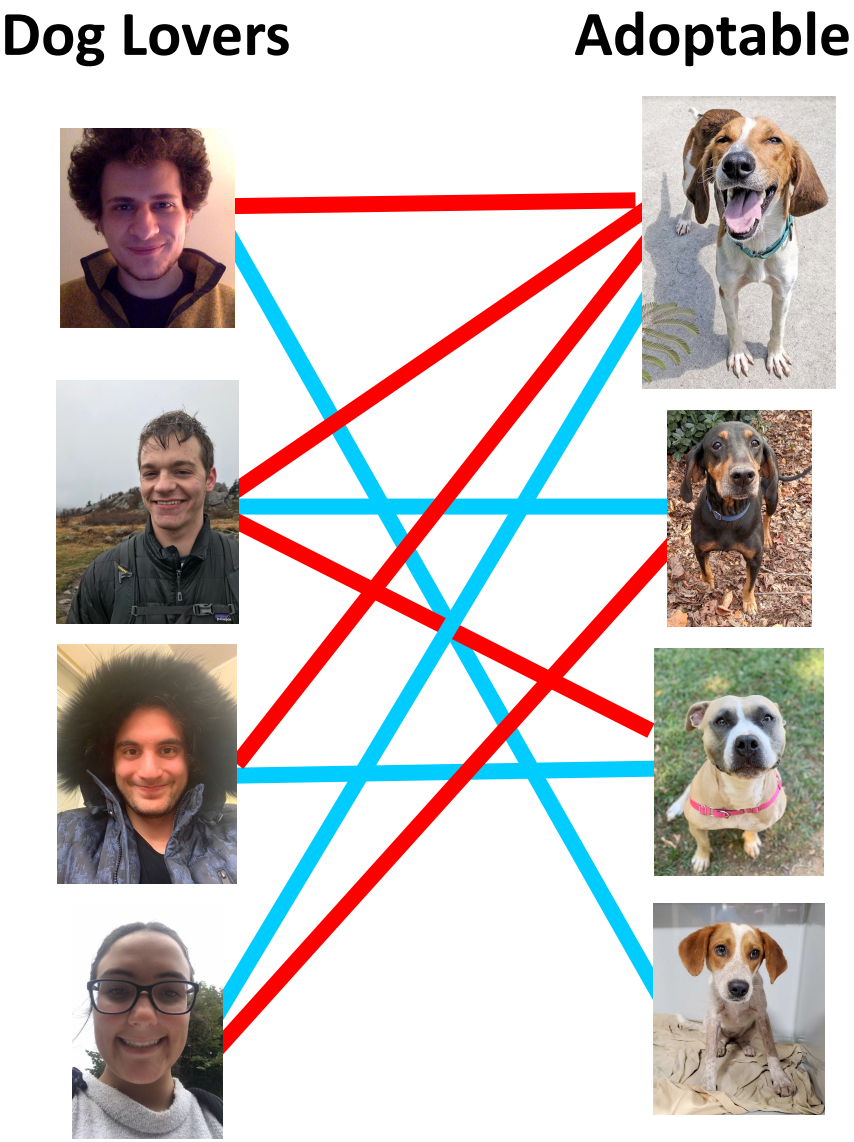
Maximum Bipartite Matching



Maximum Bipartite Matching

Is this the best possible?
The largest possible set of edges?

Maximum Bipartite Matching



Better! In fact, the maximum possible! How can we tell?

A ***perfect bipartite match***: Equal-sized left and right subsets, and all nodes have a matching edge

Maximum Bipartite Matching

Given a graph $G = (L, R, E)$

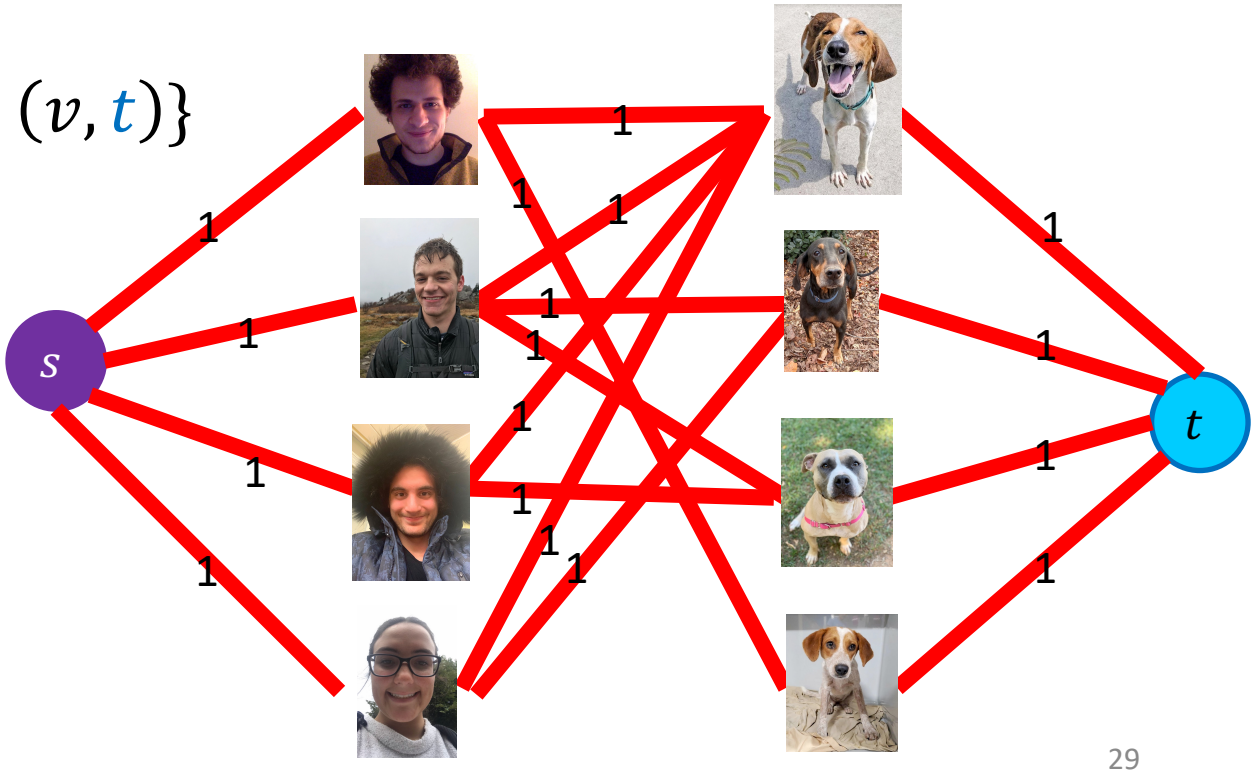
a set of left nodes, right nodes, and edges between left and right

Find the largest set of edges $M \subseteq E$ such that each node $u \in L$ or $v \in R$ is incident to at most one edge.

Maximum Bipartite Matching Using Max Flow

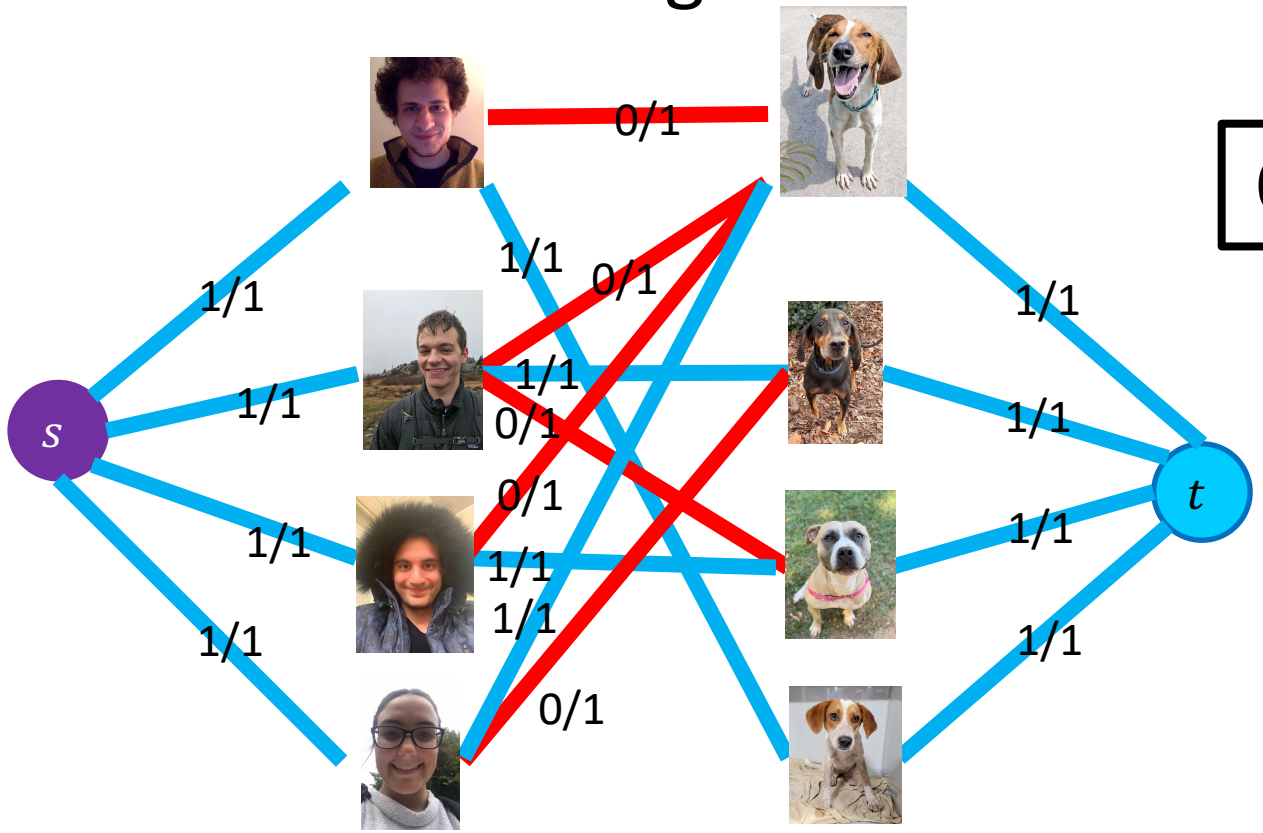
Make $G = (L, R, E)$ a flow network $G' = (V', E')$ by:

- Adding in a source and sink to the set of nodes:
  - $V' = L \cup R \cup \{s, t\}$
- Adding an edge from source to L and from R to sink:
  - $E' = E \cup \{u \in L \mid (s, u)\} \cup \{v \in R \mid (v, t)\}$
- Make each edge capacity 1:
  - $\forall e \in E', c(e) = 1$



Maximum Bipartite Matching Using Max Flow

1. Make G into G' $\Theta(L + R)$
2. Compute Max Flow on G' $\Theta(E \cdot V)$ $|f| \leq L$
3. Return M as all "middle" edges with flow 1 $\Theta(L + R)$



Overall: $\Theta(E \cdot V)$

Why does this work?

- Each node on the left can be in at most one matching
  - This is enforced by the edge of capacity one leading into it
- Likewise for each node on the right
- The bottleneck will be how it flows across the bipartite "barrier"

Reduction details

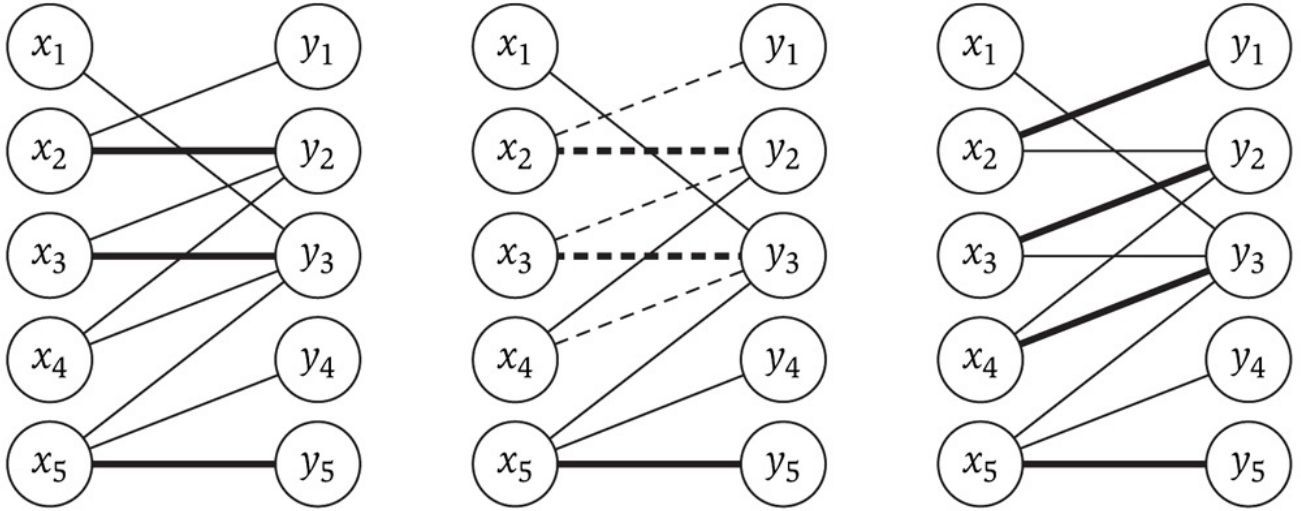
- We have transformed (in polynomial time) a bipartite matching problem into a maximal flow problem
- Specifically, bipartite-matching $\leq_p$ max-flow
  - Because we can transform bipartite matching to max-flow in polynomial time
- But is it the case that max-flow $\leq_p$ bipartite-matching?
  - Not so much: a solution to bipartite matching does not help us with a non-bipartite graph

Running time

- Max flow runs in O(E*f)
  - But the max flow is (at most) V/2 (where $V = L \cup R$)
    - If every node in the graph has flow through it, then there are V/2 units of flow moving through the graph
  - So the running time is equivalent to O(E*V)

Imperfect bipartite matchings

- These exist, and the algorithm may produce them, depending on the graph
  - The following shows an augmenting path (in the middle) used to achieve the maximal flow on the right



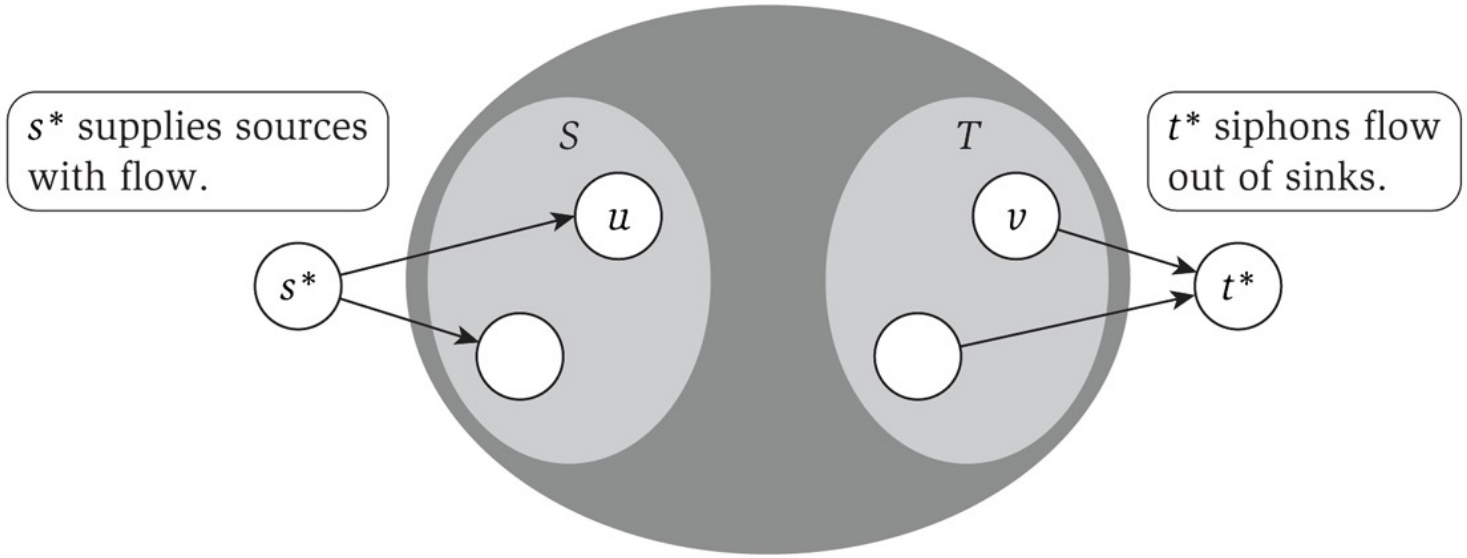
More Reductions
Max-Flow “Variations”

Finding a Circulation

- Real world applications don't have just one source and sink
  - Instead there are multiple ones: power production / consumption, etc.
- We designate a set S to be all the nodes that are sources
  - We can also view them has having negative demand
- Likewise, we designate a set T to be all the nodes that are sinks
  - They have positive demand
- Networks with multiple sources and sinks (modeled using demand) are called *circulation networks*

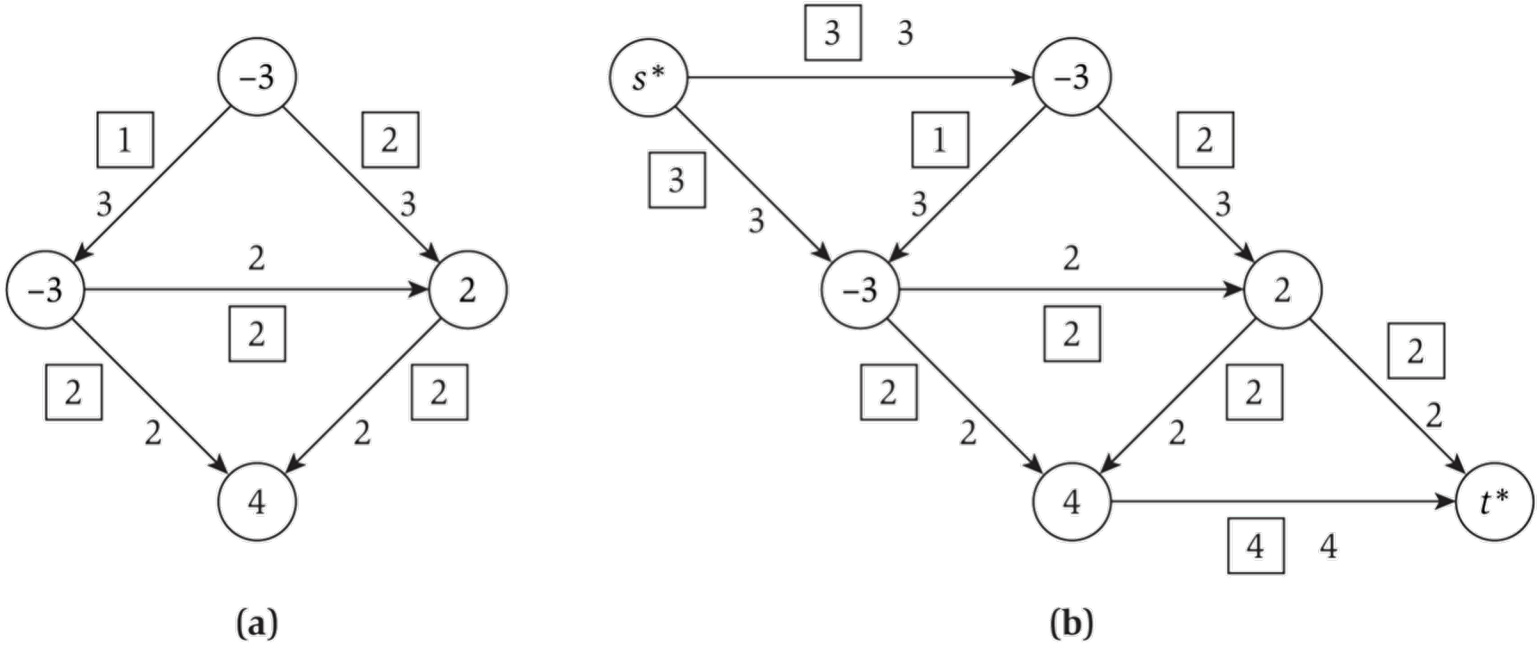
Reduction to max-flow

- With a few modifications, we can make this a max-flow problem:
  - Create a ‘super source’ s* with edges to each node in S
    - The capacity of that edge is the size of the source of the node in S
  - Likewi[illegible]



Conversion example

- Converting a graph with multiple sources and sinks to a single-source-single-sink max-flow problem:



Circulation notes

- A circulation problem is aiming for *feasibility*, not max flow
  - But we use max flow to solve it
- We set each edge from the super-source to each individual source to be the absolute value as the individual source's demand
- Max-flow is then run
- If the total amount leaving the single-source is the SAME as the capacity of each outgoing edge, then the circulation is feasible

Edge lower bounds

- So far, we have considered only the capacity of an edge: the upper bound on the flow
- We also want to consider a lower bound on the flow on an edge
  - i.e. forcing a certain amount of flow through an edge
- We will reduce this to a circulation problem
  - Which can then be reduced to a max-flow problem

Handling lower bounds

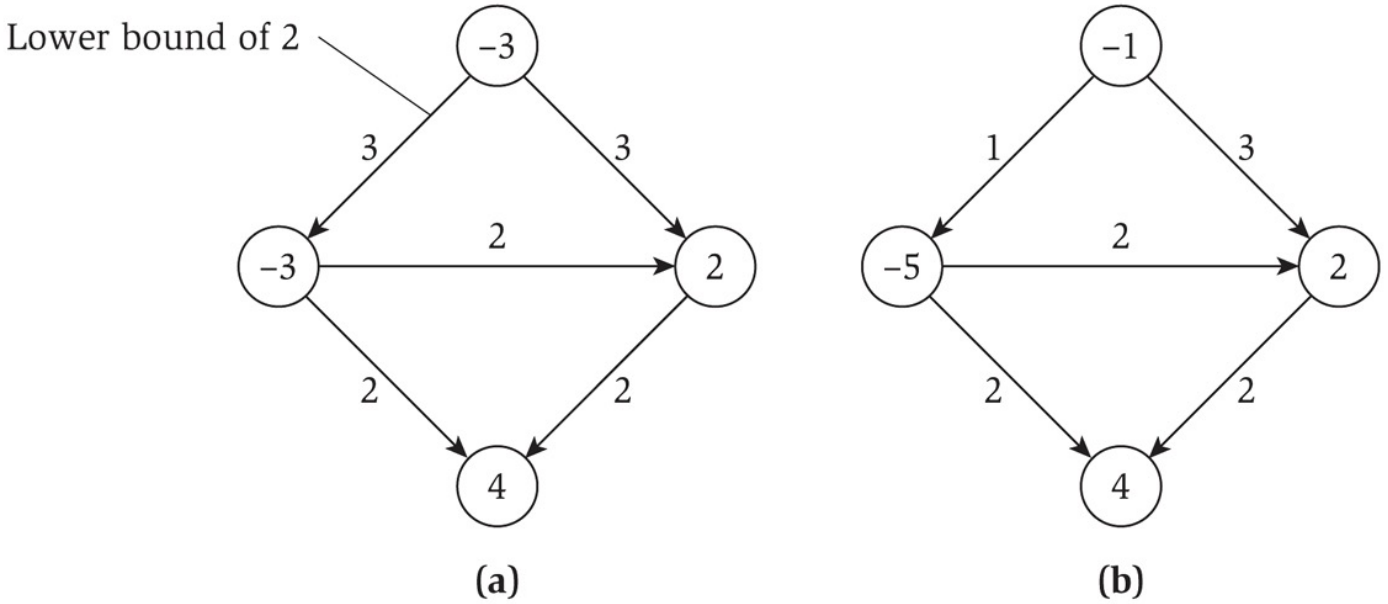
- A lower bound forces flow across an edge
  - Which increases demand at the start of the edge (to compensate for the flow across the edge)
  - And decreases demand at the terminus of the edge (as some flow is fulfilling the demand)

Solving a flow with lower bounds

- Given a circulation network G, construct a new graph G' such that for each edge e from u to v with a lower bound l_e:
  - We decrease the capacity on that edge by l_e
    - As that is the flow that is moving through the edge
  - We increase the demand at u by l_e
  - We decrease the demand at v by l_e
- Then solve G' as a circulation problem
  - i.e. add a super-sink and super-terminus, and solve as a max-flow problem

Eliminating a lower bound

- Diagrammatically…



Summary

What did we learn?

- Max-flow / min-cut
  - The problems, relationship between them, etc.
  - Ford-Fulkerson algorithm and related proofs that this approach is optimal
- Bi-partite matching
  - First example of a reduction. Use the algorithm from one problem to solve another problem.
- More reductions
  - Solving variations of max-flow by converting the problem into an instance of “normal” max-flow.