CS4102 Algorithms

Spring 2021 – Floryan and Horton

Module 4, Day 3: Recorded Lecture

Roadmap: Where We're Going and Why

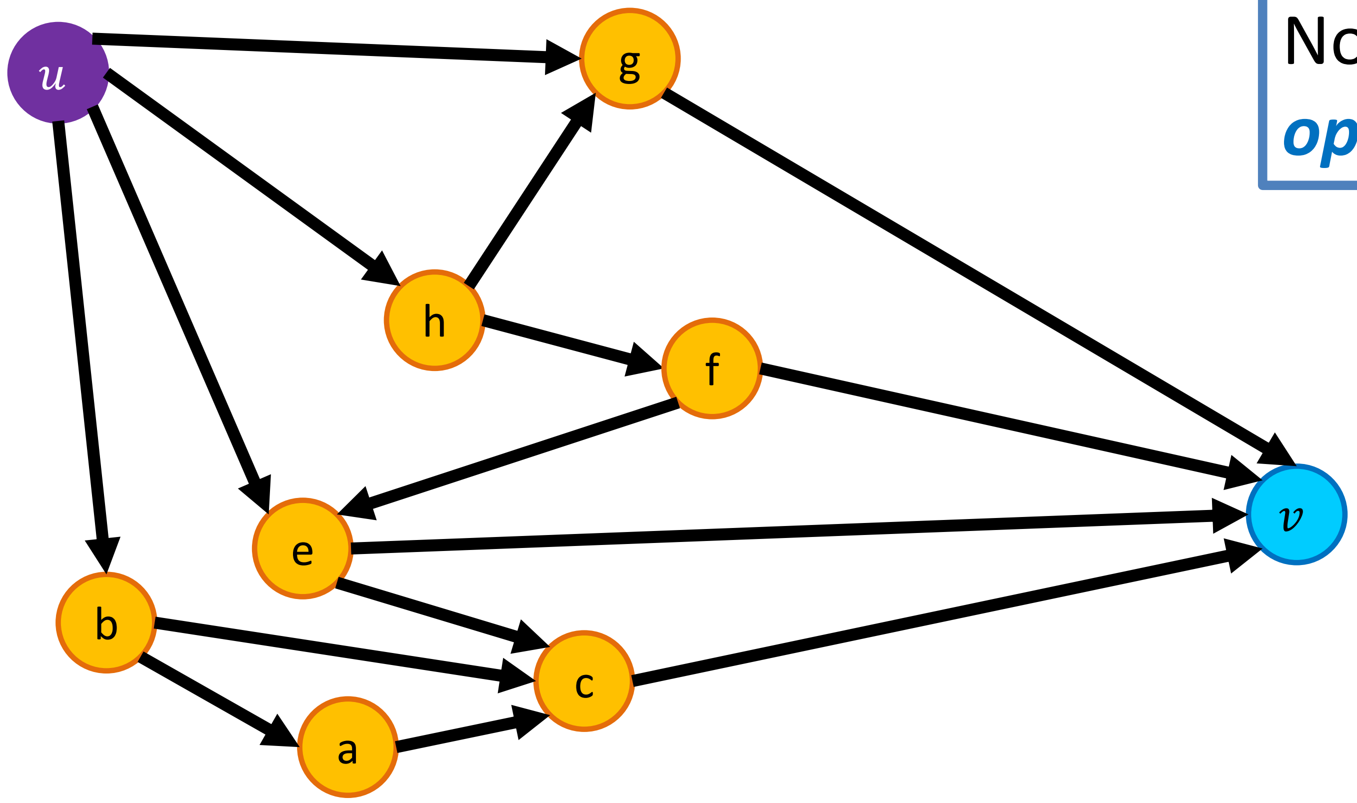
- ***Reductions*** between problems
  - Why? Can be a practical way of solving a new problem
  - Also: A proof about one problem's complexity can be applied to another
  - Formal definition of a reduction
- Examples
  - Bipartite graphs, matching
  - Vertex cover and independent set

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

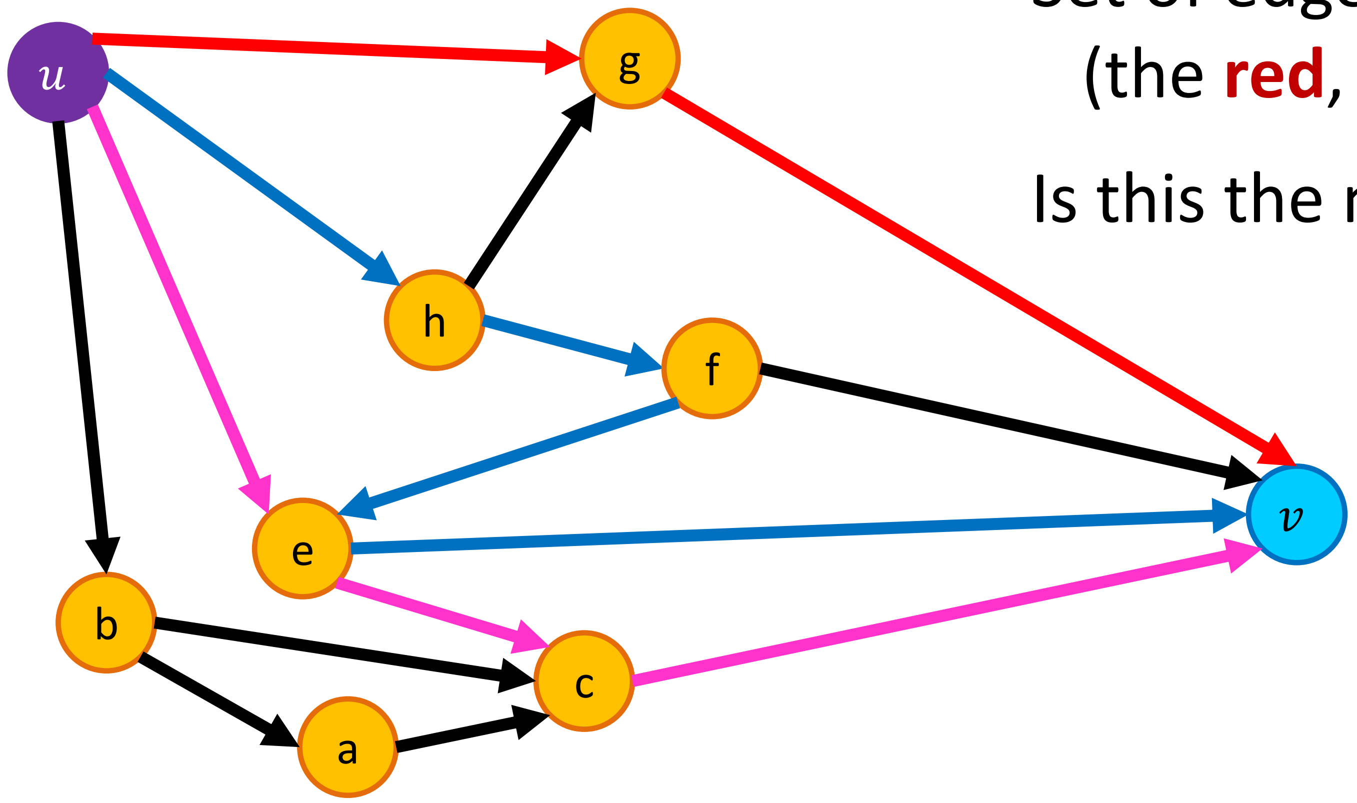
Given a graph $G = (V, E)$, a start node u and a destination node v, give the maximum number of paths from u to v which share no edges



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

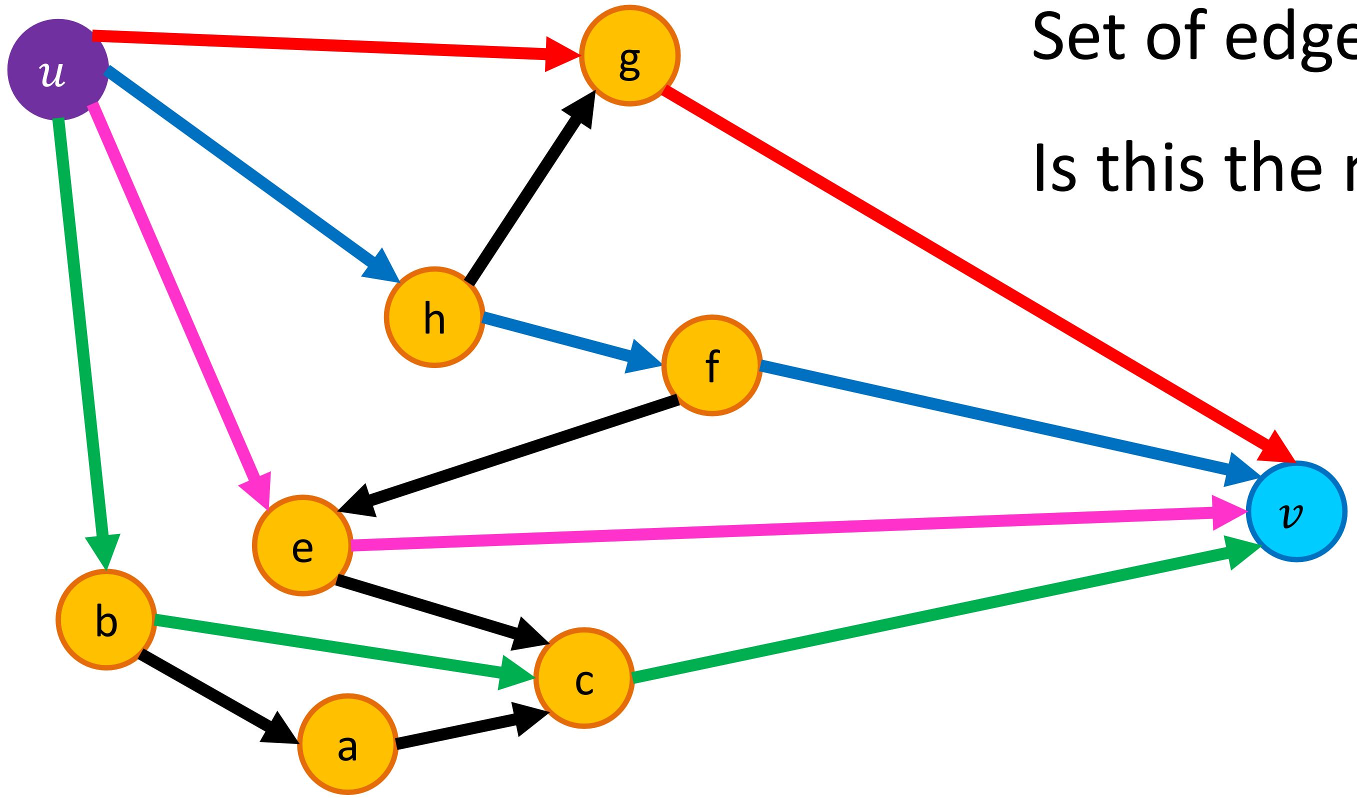


Set of edge-disjoint paths of size 3 (the **red**, **blue**, **magenta** paths)

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



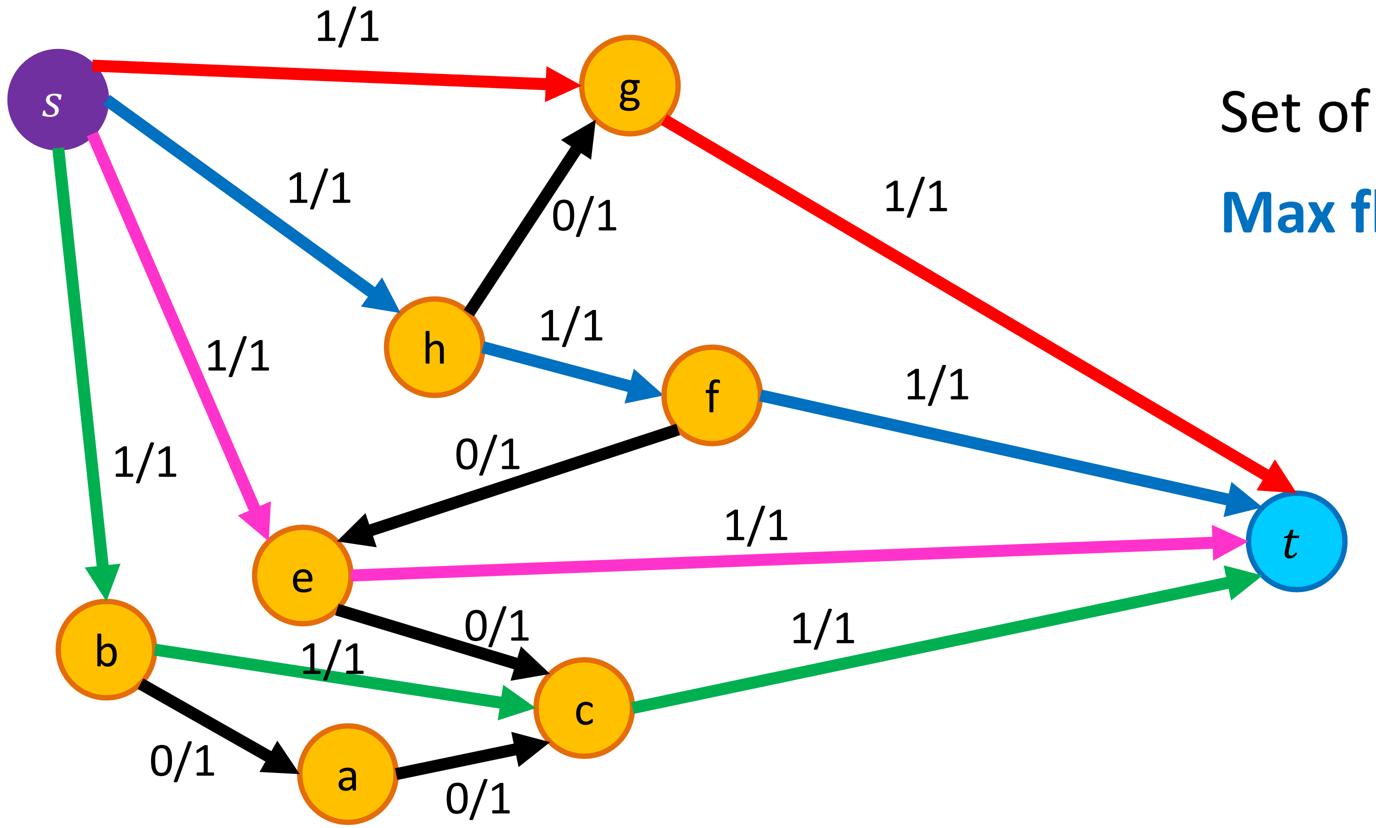
Set of edge-disjoint paths of size 4

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

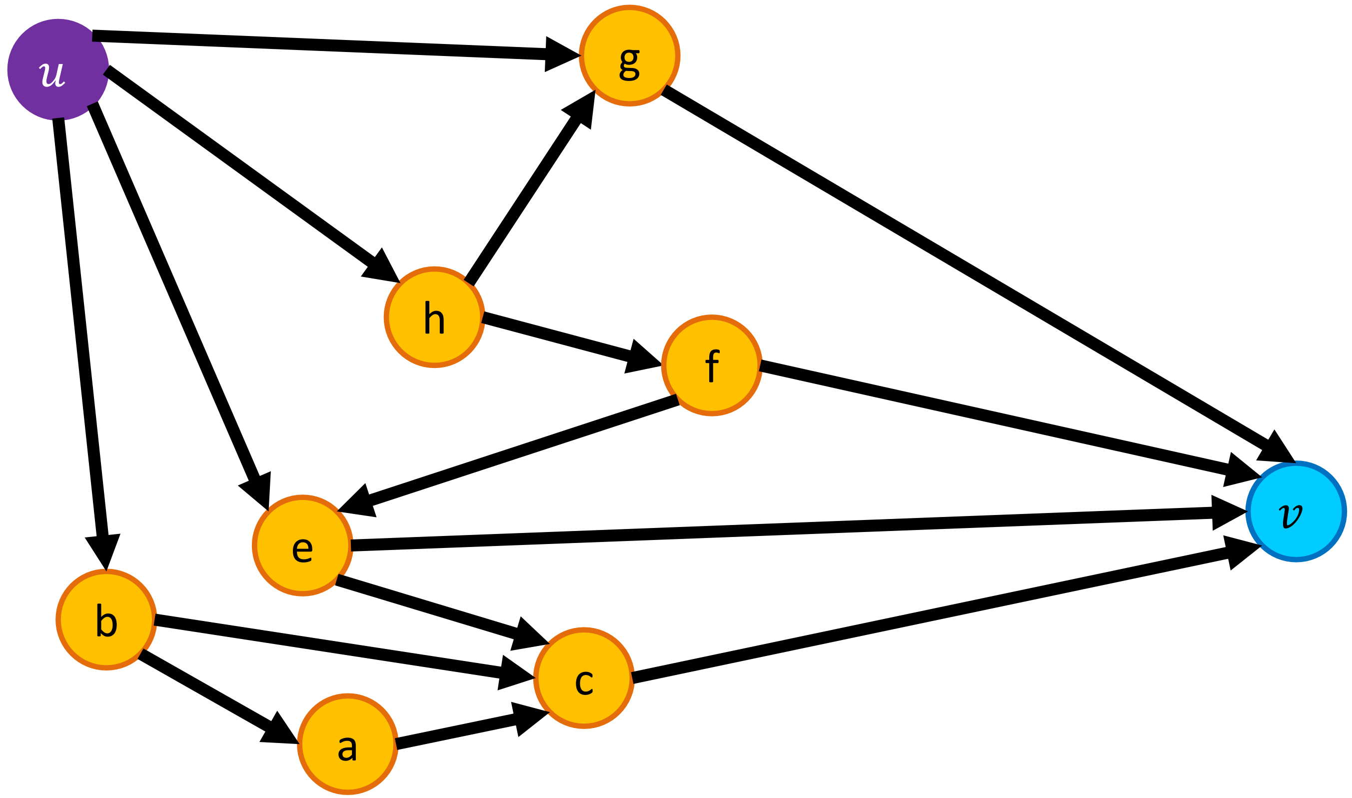
Why does this work?
We need to be able to make a valid argument that it always does.

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

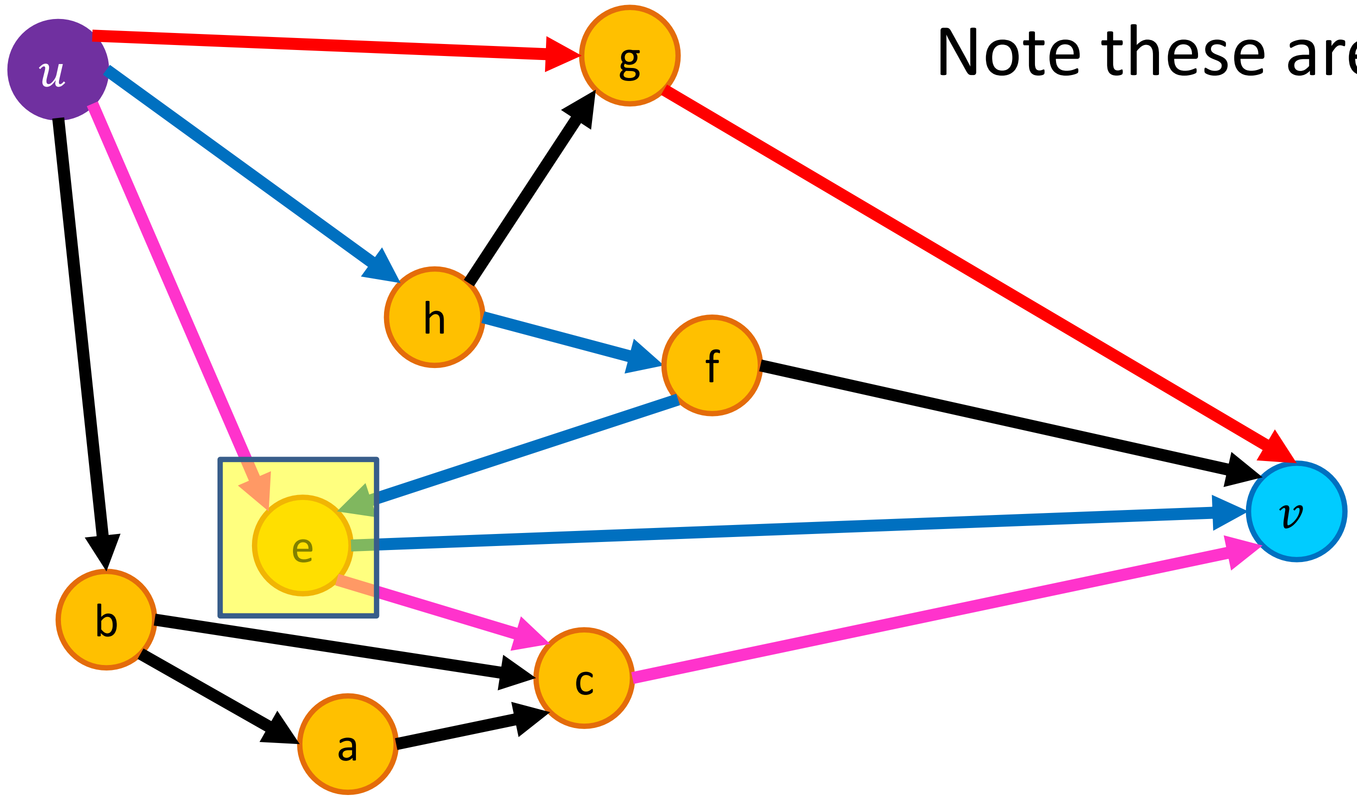
Vertex-Disjoint Paths

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 <u>vertices</u>



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.

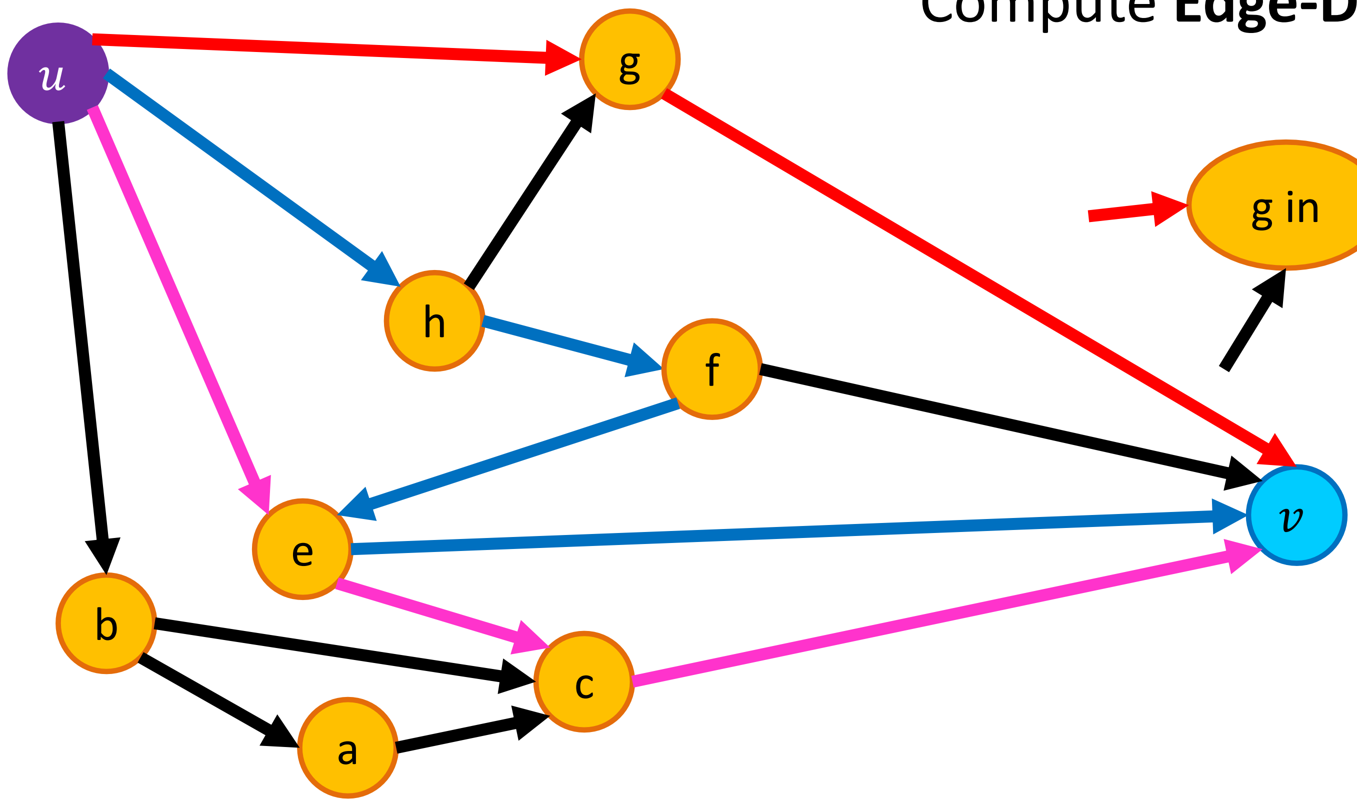
Note these aren't vertex-disjoint paths!

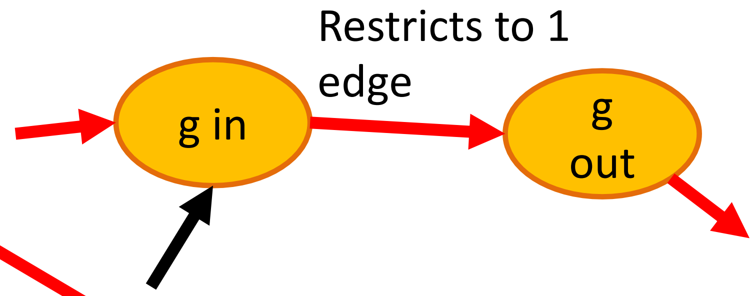
Vertex-Disjoint Paths Algorithm

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 <u>now</u>?

- 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. 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***

Reduction

- A ***reduction*** is a transformation of one problem into another problem
  - Min-cut is reducible to max-flow because we can use max-flow to solve min-cut
  - 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 in *polynomial time*
- If A is ***polynomial-time reducible*** to B, we denote this as:

  $\mathbf{A \leq_p B}$

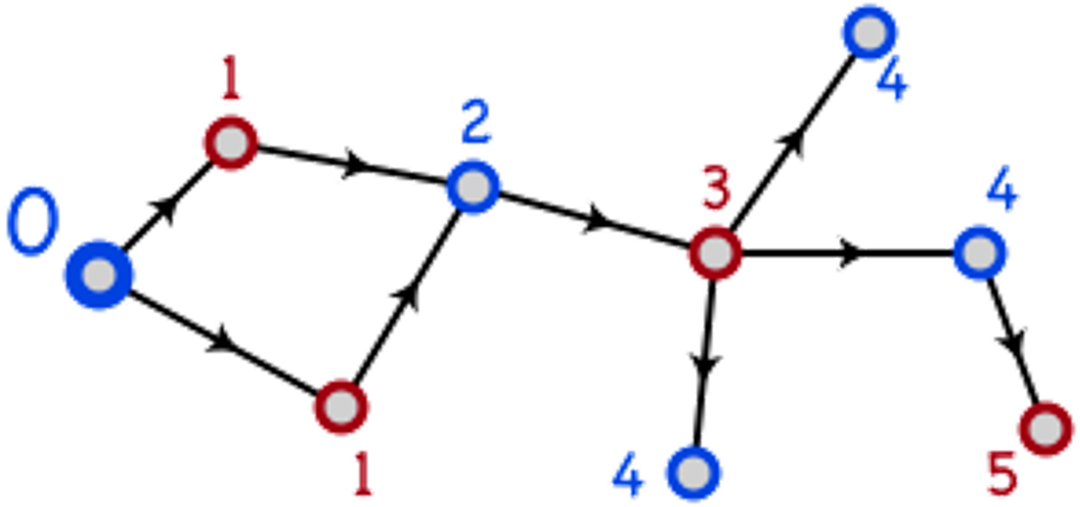
Reducing both ways

- It's easy to see that:
  - Min-cut $\leq_p$ max-flow
  - Max-flow $\leq_p$ min-cut
- Because they reduce both ways, they are ***polynomial-time equivalent***
  - If we find a polynomial solution for one, the other is also polynomial
  - What if we prove an exponential lower-bound for one?
    Is it possible that the other one could have a polynomial solution?

Bipartite Matching

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



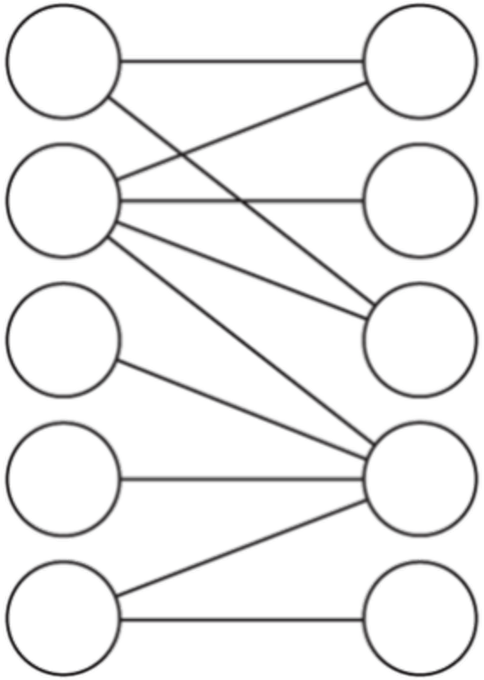
How to determine if G is bipartite?

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

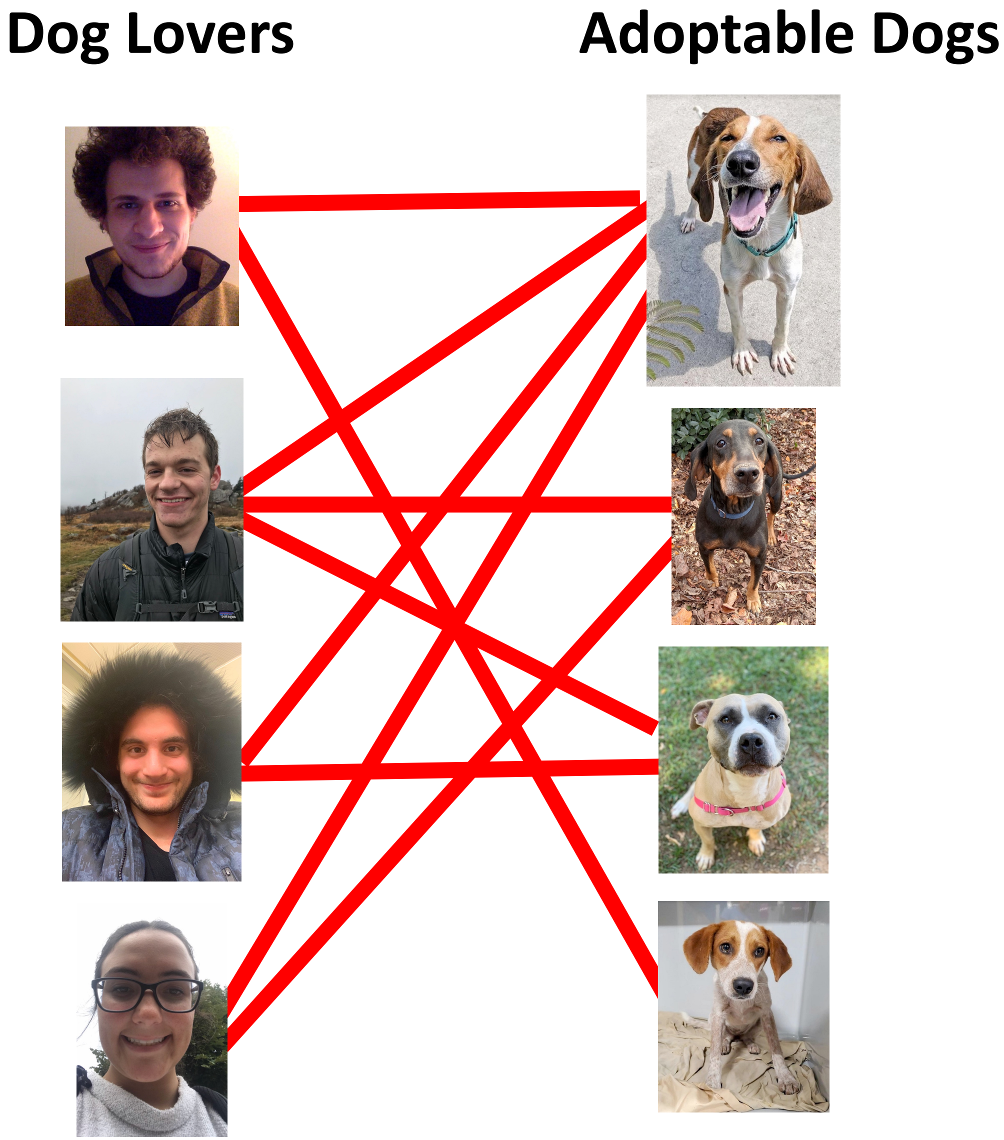
BFS or DFS, and label nodes by levels in tree.
Non-tree edge to node with same label means NOT bipartite.

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

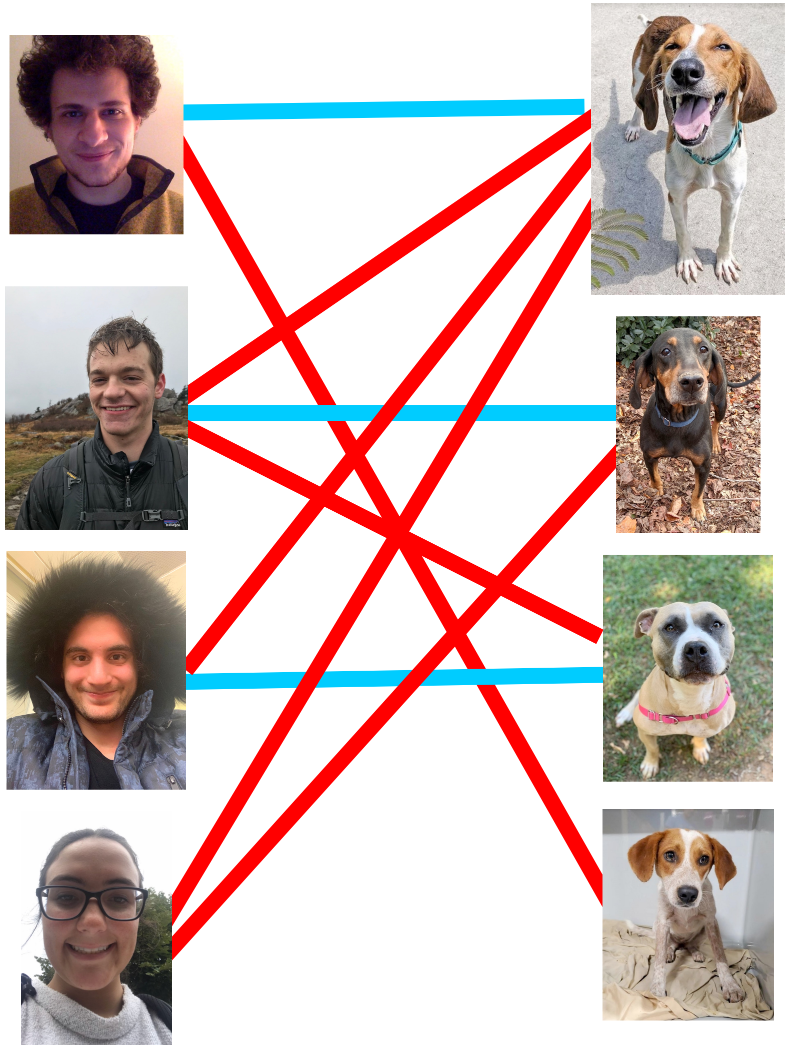


Maximum Bipartite Matching



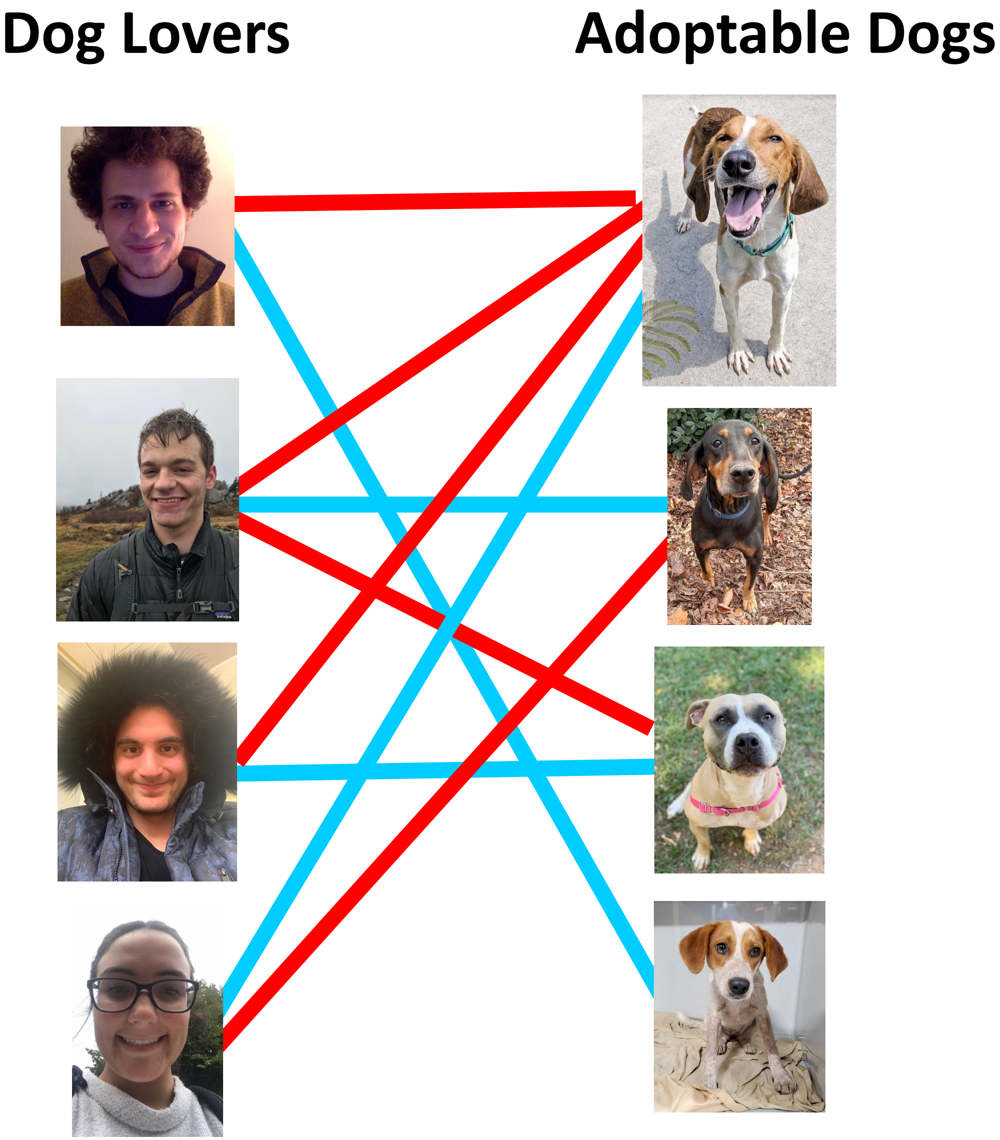
Maximum Bipartite Matching

Dog Lovers **Adoptable Dogs**



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 $G = (L, R, E)$

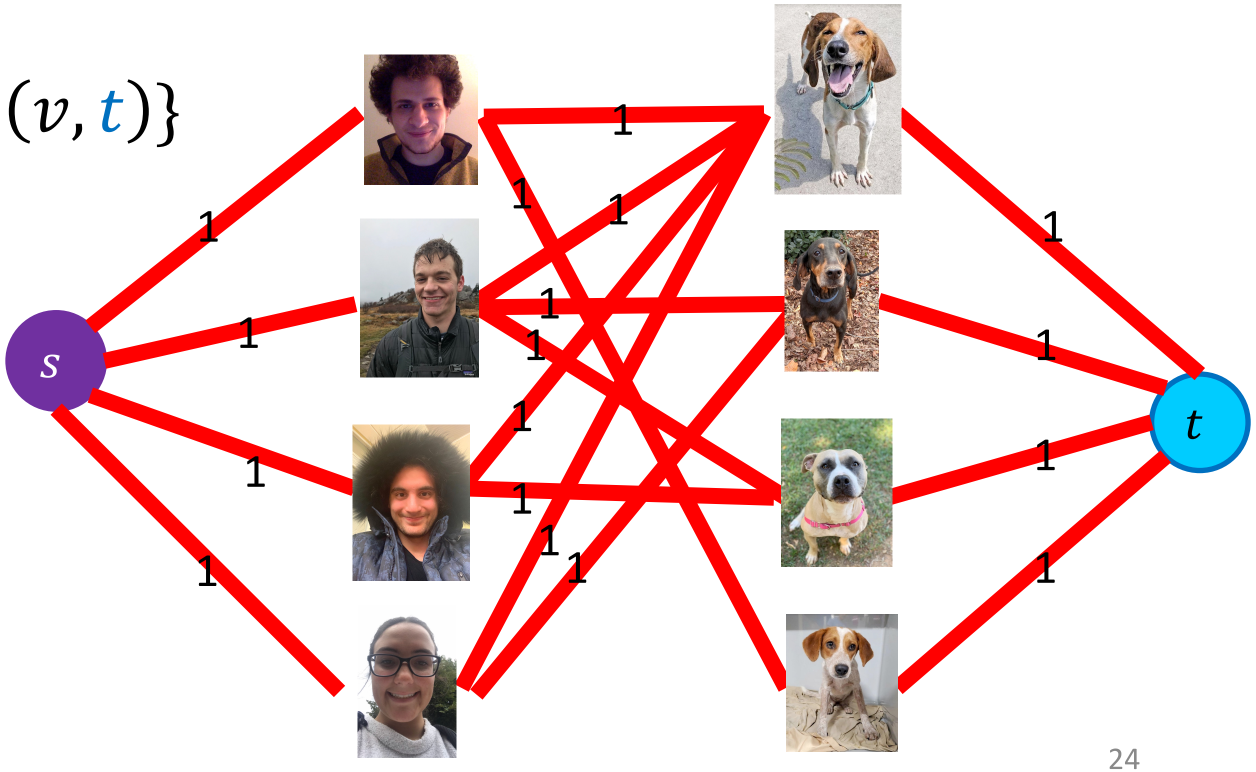
a set of left nodes, right nodes, and edges between left and right

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

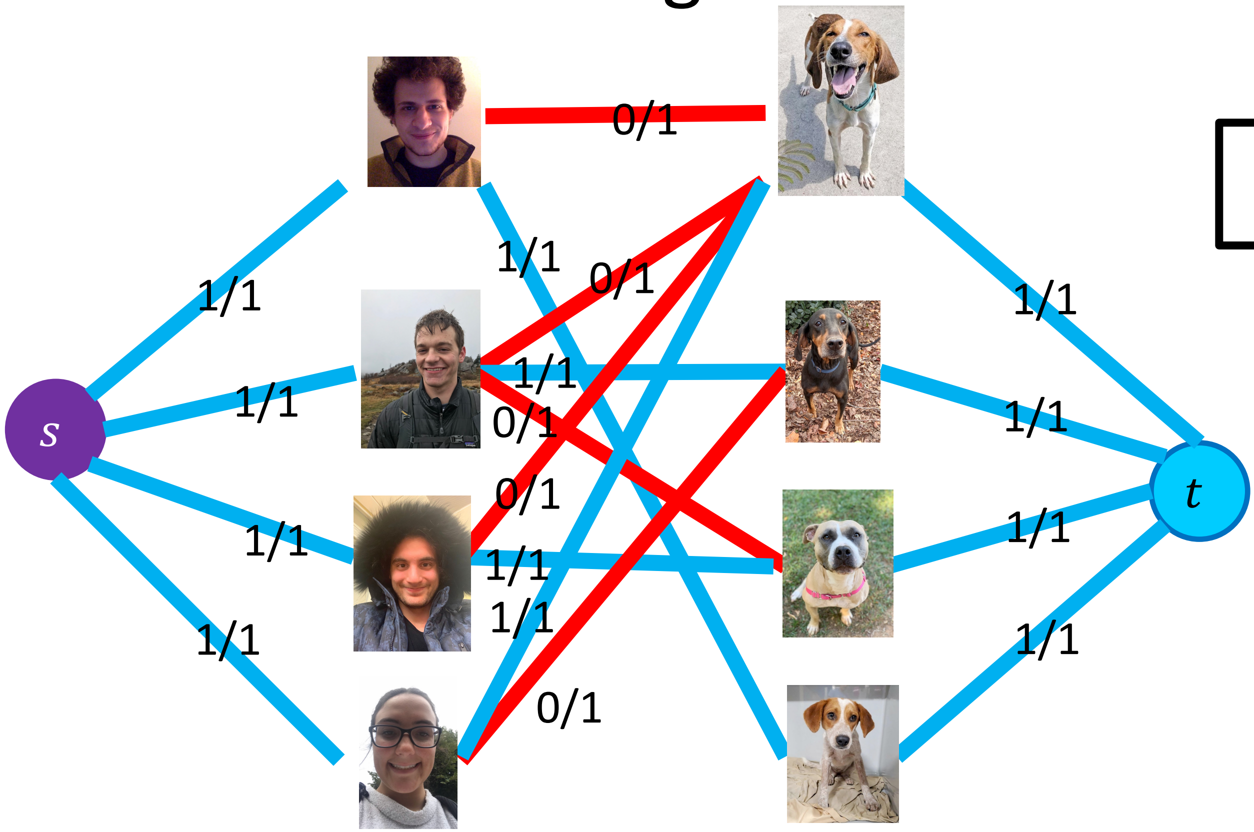
Make $G = (L, R, E)$ a flow network $G' = (V', E')$ by:

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

Roadmap: Where We've Been and Why

- ***Reductions*** between problems
  - Why? Can be a practical way of solving a new problem
  - **Coming soon:** A proof about one problem's complexity can be applied to another
  - Formal definition of a reduction
- Examples
  - Bipartite graphs, matching
- **Next:** example problems: vertex cover and independent set
  - Then, classes of problems: P, NP, NP-Hard, NP-complete