CS 3100
Data Structures and Algorithms 2
Lecture 21: Reductions, Bipartite Matching

Co-instructors: Robbie Hott and Ray Pettit

Spring 2024

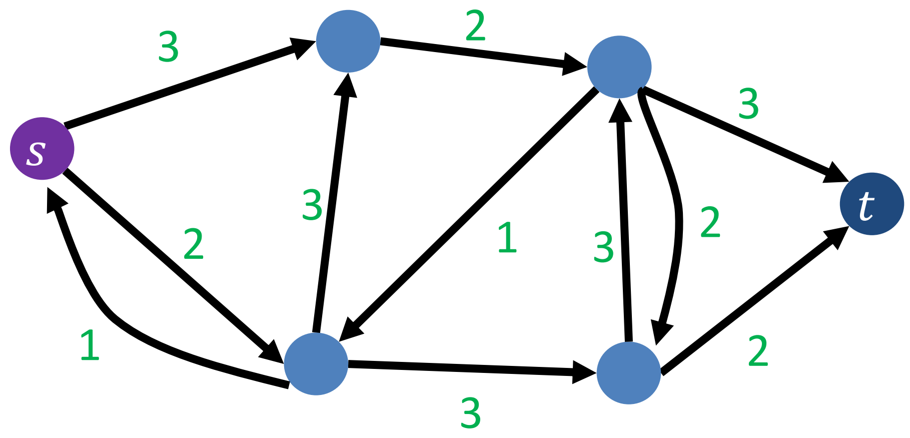
Readings from CLRS 4th Ed:
Chapter 24

Announcements

- Quizzes 3-4 Thursday
  - If you have SDAC, please schedule ASAP
  - Review Session: Tonight
  - Quiz Security
    - Arrive early to get your quiz, bring your ID with you
    - Your quiz will have your name on it
    - Do not sit next to your friends
- Office hours updates
  - Prof Hott Office Hours:
    - Today 2-3pm
    - Friday and Monday hours canceled this week (baby!)

Flow Networks

Graph $G = (V, E)$
Source node $s \in V$
Sink node $t \in V$
Edge capacities $c(e) \in \mathbb{R}^+$



Max flow intuition: If s is a faucet, t is a drain, and s connects to t through a network of pipes E with capacities $c(e)$, what is the maximum amount of water which can flow from the faucet to the drain?

Residual Graphs

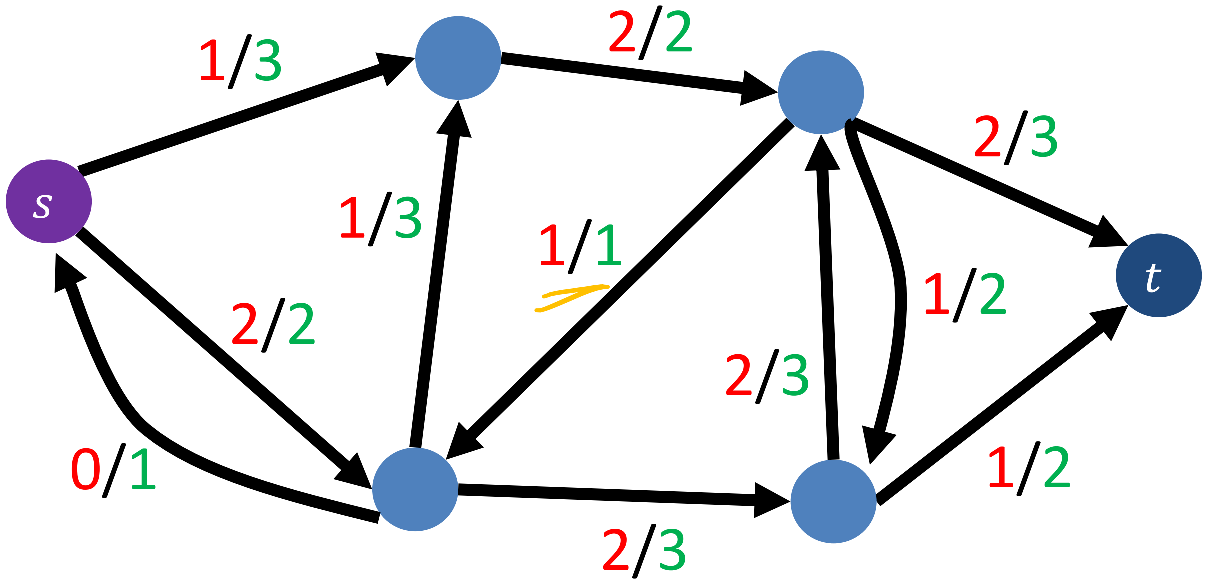
Given a flow f in graph G, the residual graph G_f models additional flow that is possible

- Forward edge for each edge in G with weight set to remaining capacity $c(e) - f(e)$
  - Models additional flow that can be sent along the edge

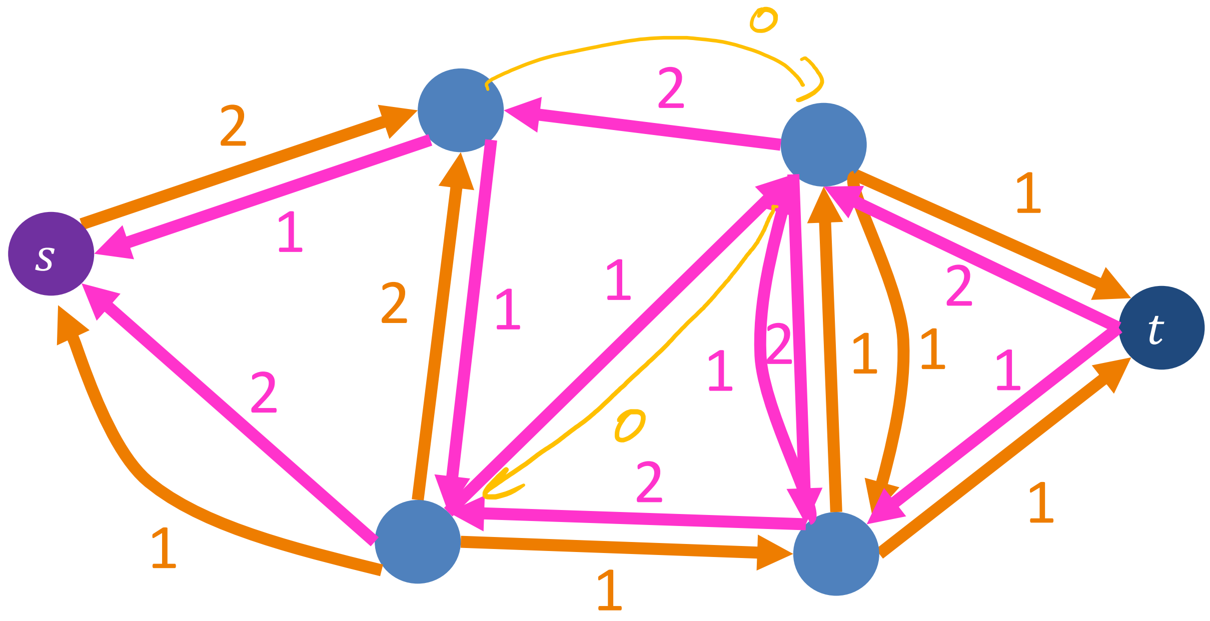
Flow I *could* add

- Backward edge by flipping each edge e in G with weight set to flow $f(e)$
  - Models amount of flow that can be removed from the edge

Flow I *could* remove



Flow f in G



Residual graph G_f

Ford-Fulkerson Algorithm

Define an augmenting path to be an $s \to t$ path in the residual graph G_f (using edges of non-zero weight)

Ford-Fulkerson max-flow algorithm:

- Initialize $f(e) = 0$ for all $e \in E$
- Construct the residual network G_f
- While there is an augmenting path p in G_f:
  - Let $c = \min_e c_f(e)$ along the path

    ($c_f(e)$ is the weight of edge e in the residual network G_f)
  - Add c units of flow to G based on the augmenting path p
  - Update the residual network G_f for the updated flow

Ford-Fulkerson approach: take any augmenting path (will revisit this later)

Can We Avoid this?

Edmonds-Karp Algorithm: choose augmenting path with fewest hops

Running time: $\Theta(\min(|E||f^*|, |V||E|^2)) = O(|V||E|^2)$

How to find this?
Use breadth-first search (BFS)!

Edmonds-Karp = Ford-Fulkerson using BFS to find augmenting path

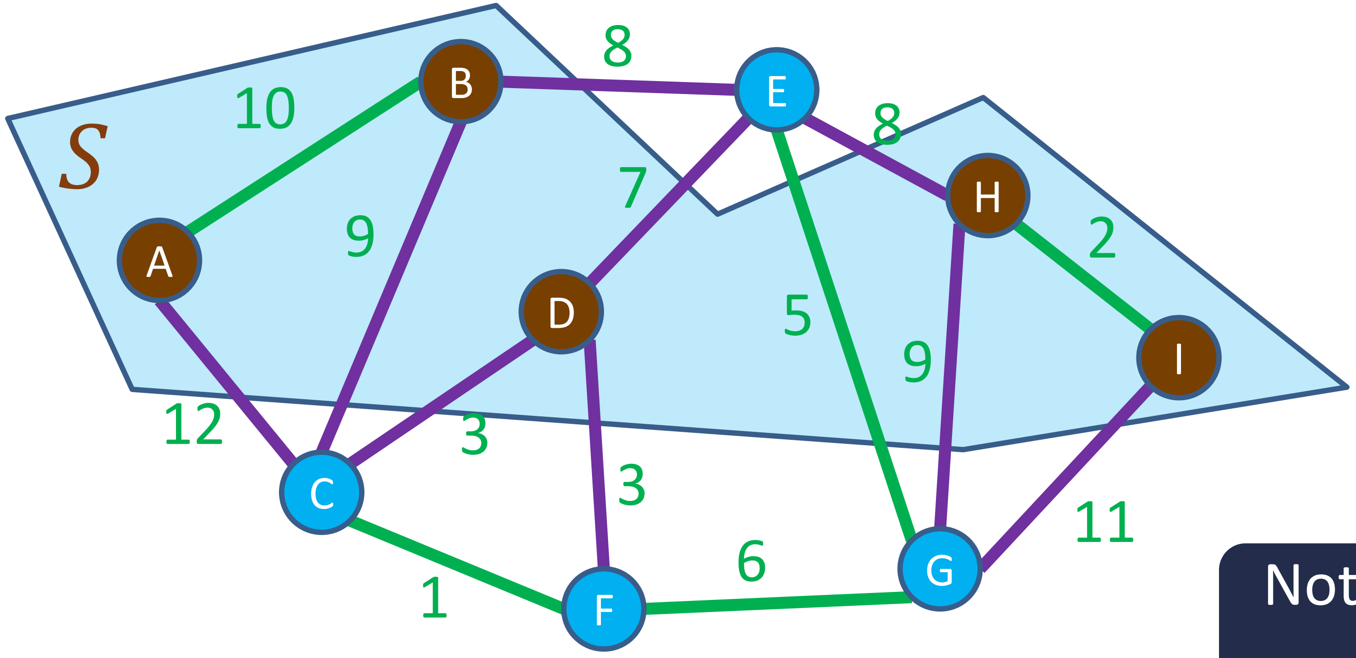
Ford-Fulkerson max-flow algorithm:

- Initialize $f(e) = 0$ for all $e \in E$
- Construct the residual network G_f
- While there is an augmenting path in G_f, let p be the path with fewest hops:
  - Let $c = \min_{e \in E} c_f(e)$ ($c_f(e)$ is the weight of edge e in the residual network G_f)
  - Add c units of flow to G based on the augmenting path p
  - Update the residual network G_f for the updated flow

See CLRS (Chapter 24)

Reminder: Graph Cuts

A **cut** of a graph $G = (V, E)$ is a partition of the nodes into two sets, S and $V - S$



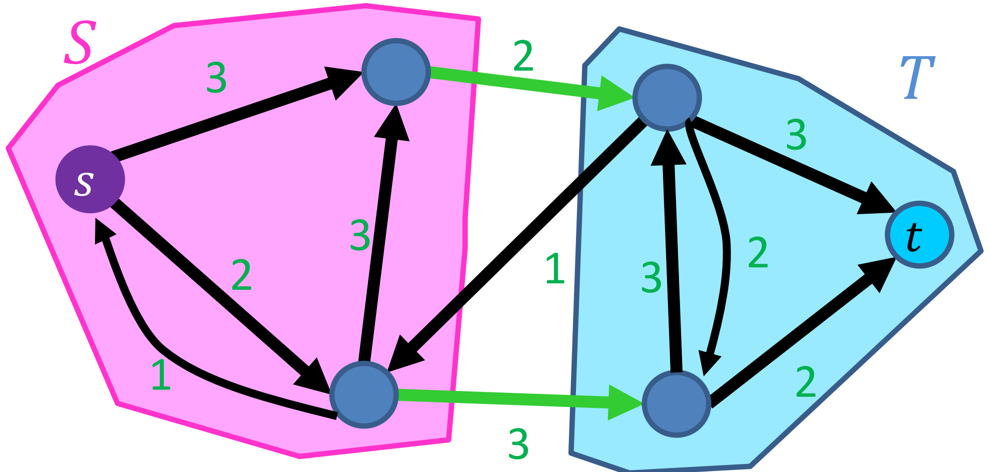
Notion extends naturally to a set of edges

An edge $(v_1, v_2) \in E$ crosses a cut if $v_1 \in S$ and $v_2 \in V - S$

An edge $(v_1, v_2) \in E$ respects a cut if $v_1, v_2 \in S$ or if $v_1, v_2 \in V - S$

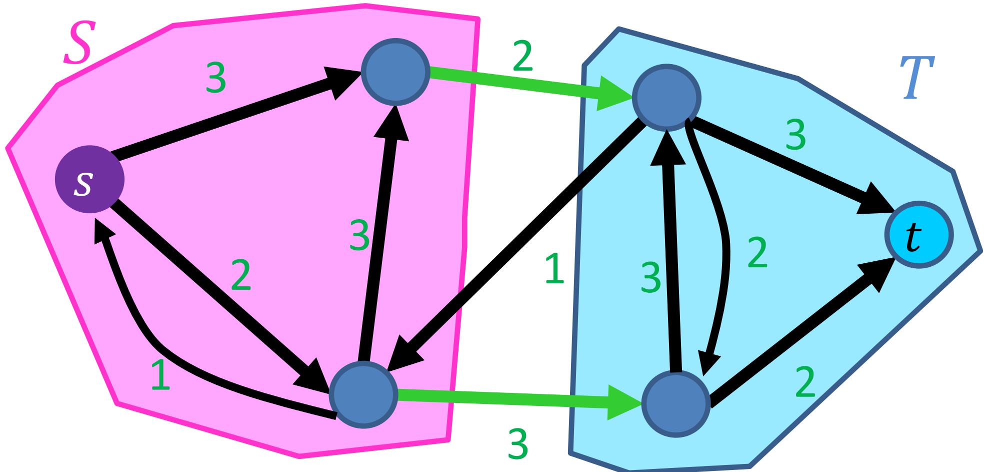
Showing Correctness of Ford-Fulkerson

- Consider cuts which separate s and t
  - Let $s \in S$, $t \in T$, s.t. $V = S \cup T$
- Cost of cut $(S, T) = ||S, T||$
  - Sum **capacities** of edges which go from S to T
  - This example: 5



Maxflow≤MinCut

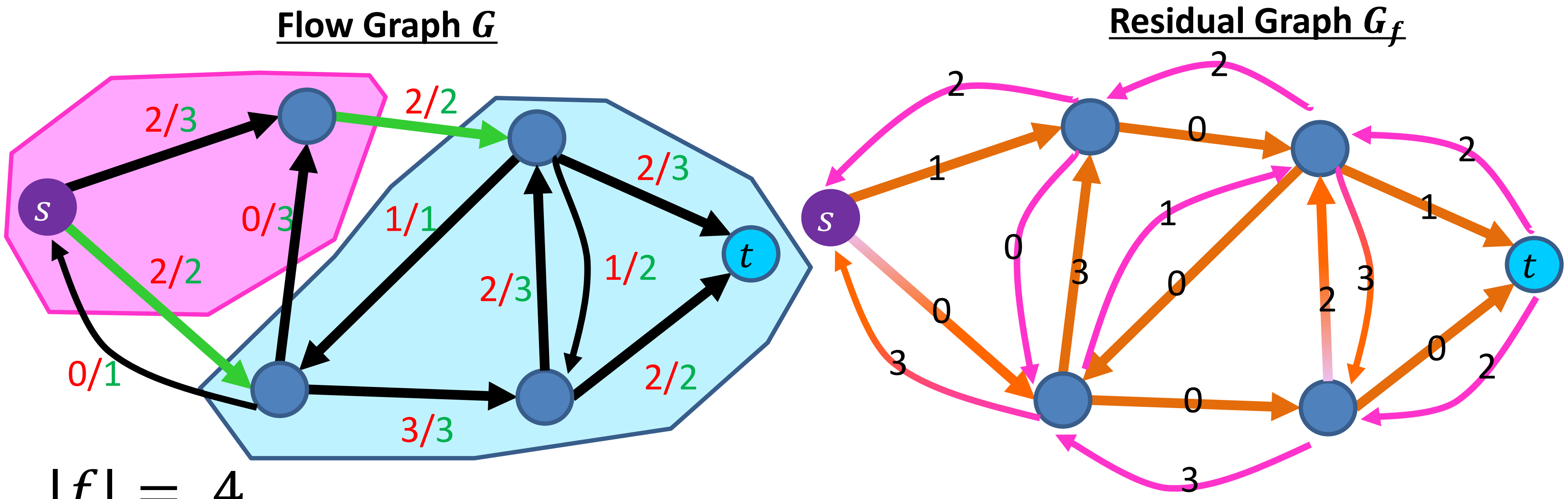
- Max flow upper bounded by any cut separating s and t
- Why? "Conservation of flow"
  - All flow exiting s must eventually get to t
  - To get from s to t, all "tanks" must cross the cut
- Conclusion: If we find the minimum-cost cut, we've found the maximum flow
  - $\max_f |f| \leq \min_{S,T} ||S,T||$



Maxflow/Mincut Theorem

- To show Ford-Fulkerson is correct:
  - Show that when there are no more augmenting paths, there is a cut with cost equal to the flow
- Conclusion: the maximum flow through a network matches the minimum-cost cut
  - $\max_f |f| = \min_{S,T} ||S, T||$
- Duality
  - When we've maximized max flow, we've minimized min cut (and vice-versa), so we can check when we've found one by finding the other

Example: Maxflow/Mincut



$|f| = 4$

$||S, T|| = 4$

No Augmenting Paths

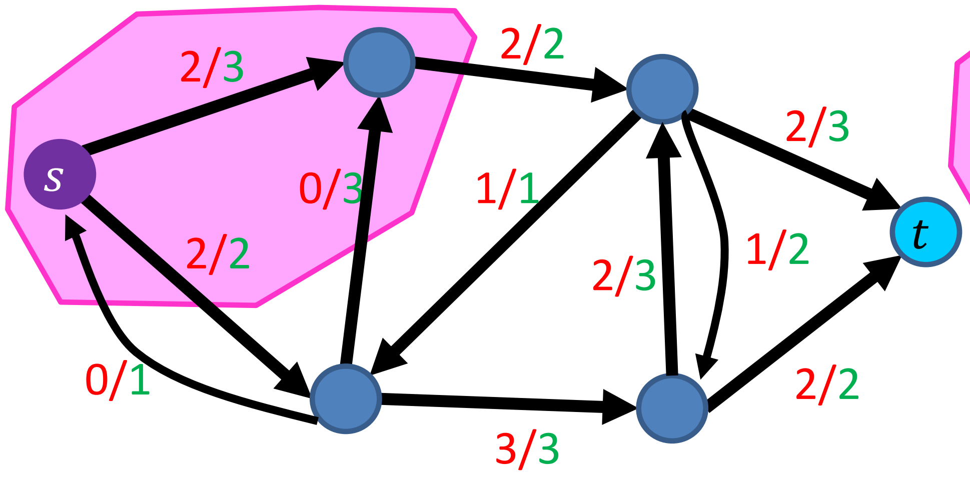
Idea: When there are no more augmenting paths, there exists a cut in the graph with cost matching the flow

11

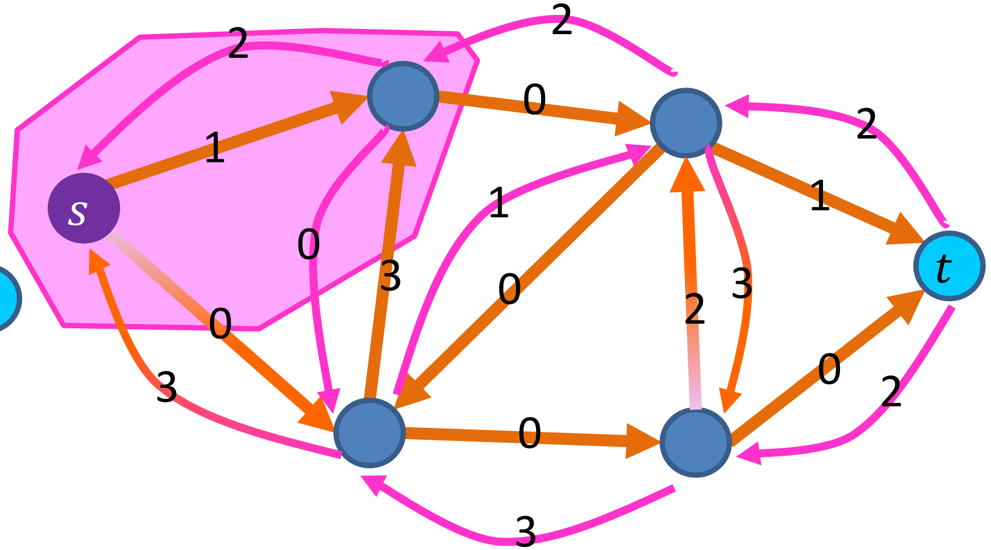
Proof: Maxflow/Mincut Theorem

- If $|f|$ is a max flow, then G_f has no augmenting path
  - Otherwise, use that augmenting path to "push" more flow
- Define S = nodes reachable from source node s by positive-weight edges in the residual graph
  - $T = V - S$
  - S separates s , t (otherwise there's an augmenting path)

Flow Graph G

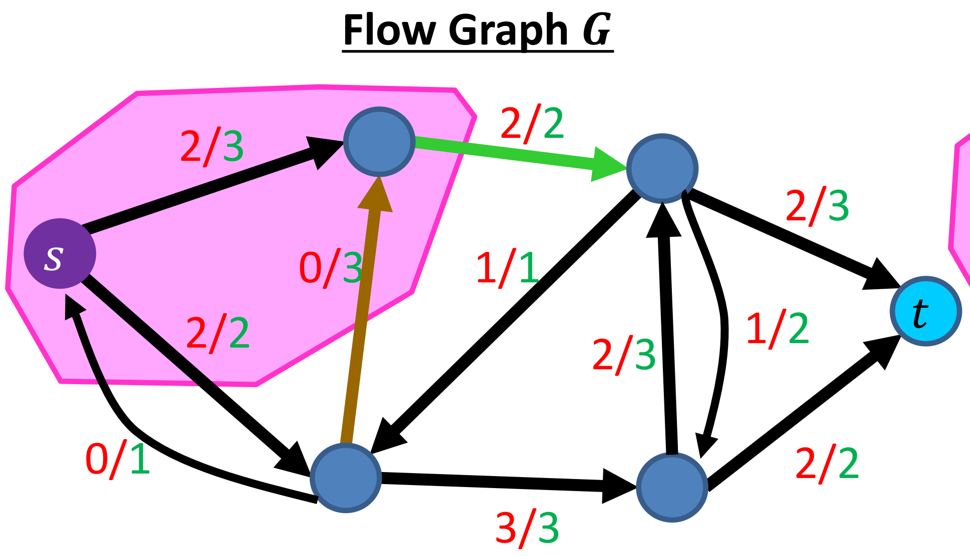


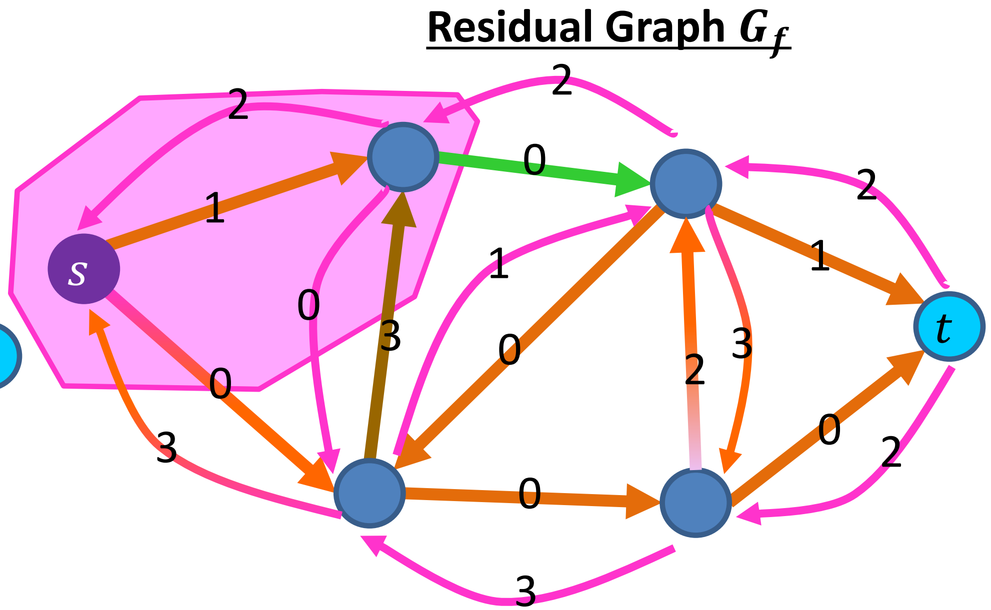
Residual Graph G_f



Proof: Maxflow/Mincut Theorem

- To show: $||S, T|| = |f|$
  – Weight of the cut matches the flow across the cut
- Consider edge (u, v) with $u \in S$, $v \in T$
  – $f(u, v) = c(u, v)$, because otherwise $w(u, v) > 0$ in G_f, which would mean $v \in S$
- Consider edge (y, x) with $y \in T$, $x \in S$
  – $f(y, x) = 0$, because otherwise the back edge $w(y, x) > 0$ in G_f, which would mean y $\in S$





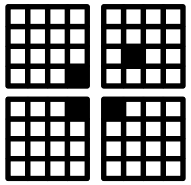
Proof Summary

1. The flow $|f|$ of G is upper-bounded by the sum of capacities of edges crossing any cut separating source s and sink t
2. When Ford-Fulkerson terminates, there are no more augmenting paths in G_f
3. When there are no more augmenting paths in G_f then we can define a cut $S =$ nodes reachable from source node s by positive-weight edges in the residual graph
4. The sum of edge capacities crossing this cut must match the flow of the graph
5. Therefore this flow is maximal

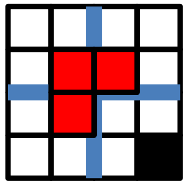
Moving on

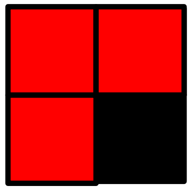
Divide and Conquer*

- **Divide**:

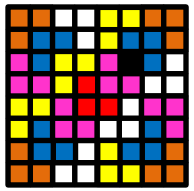


  – Break the problem into multiple subproblems, each smaller instances of the original
- **Conquer**:





  – If the suproblems are “large”:
    - Solve each subproblem recursively
  – If the subproblems are “small”:
    - Solve them directly (base case)
- **Combine**:



  – Merge together solutions to subproblems

*CLRS Chapter 4

Greedy Algorithms

- Require Optimal Substructure
  – Solution to larger problem contains the solution to a smaller one
  – Only one subproblem to consider!
- Idea:
  1. Identify a greedy choice property
     - How to make a choice guaranteed to be included in some optimal solution
  2. Repeatedly apply the choice property until no subproblems remain

Dynamic Programming

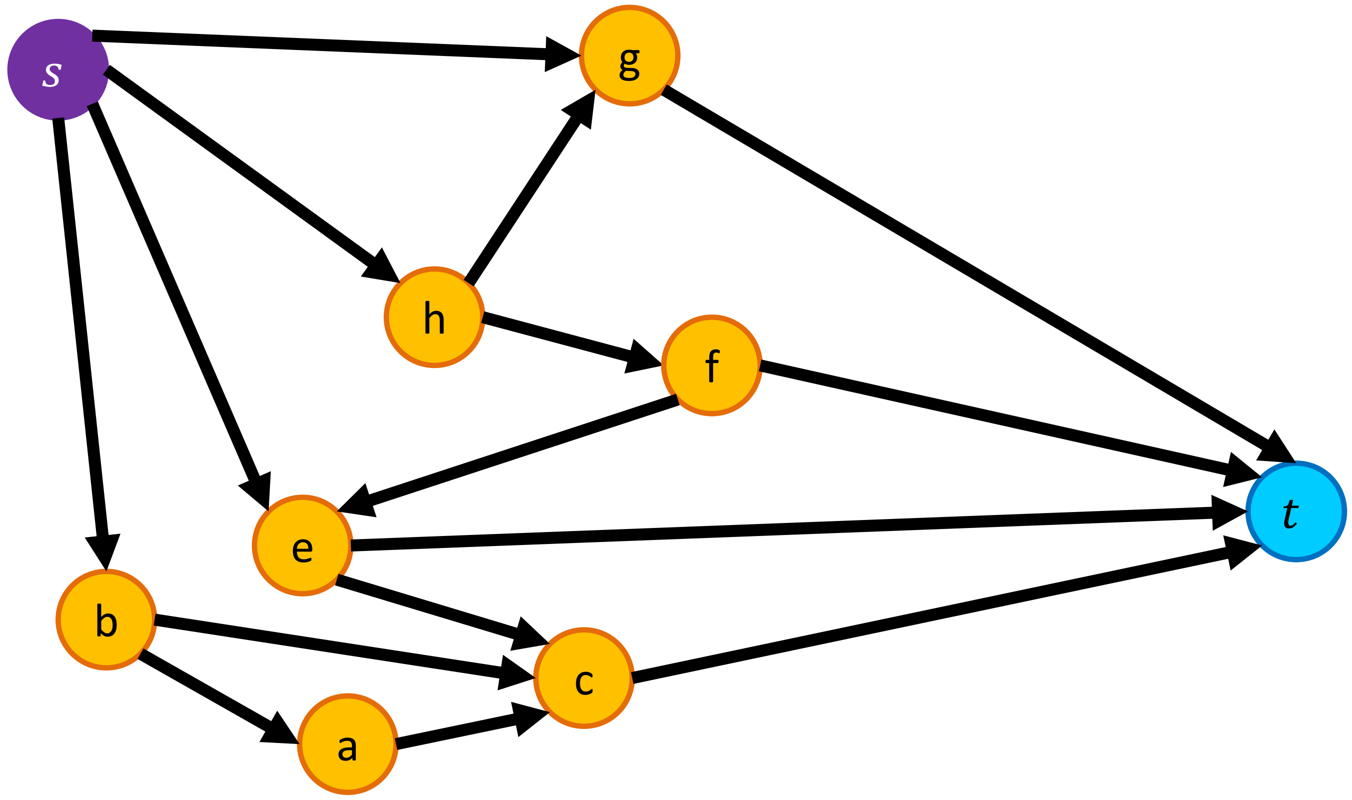
- Requires Optimal Substructure
  – Solution to larger problem contains the solutions to smaller ones
- Idea:
  1. Identify recursive structure of the problem
  2. Store solutions to subproblems in memory
  3. Select a good order for solving subproblems
     - Usually smallest problem first

So far

- Divide and Conquer, Dynamic Programming, Greedy
  - Take an instance of *Problem A*,
    relate it to smaller instances of *Problem A*
- Next:
  - Take an instance of *Problem A*,
    relate it to an instance of ***Problem B***

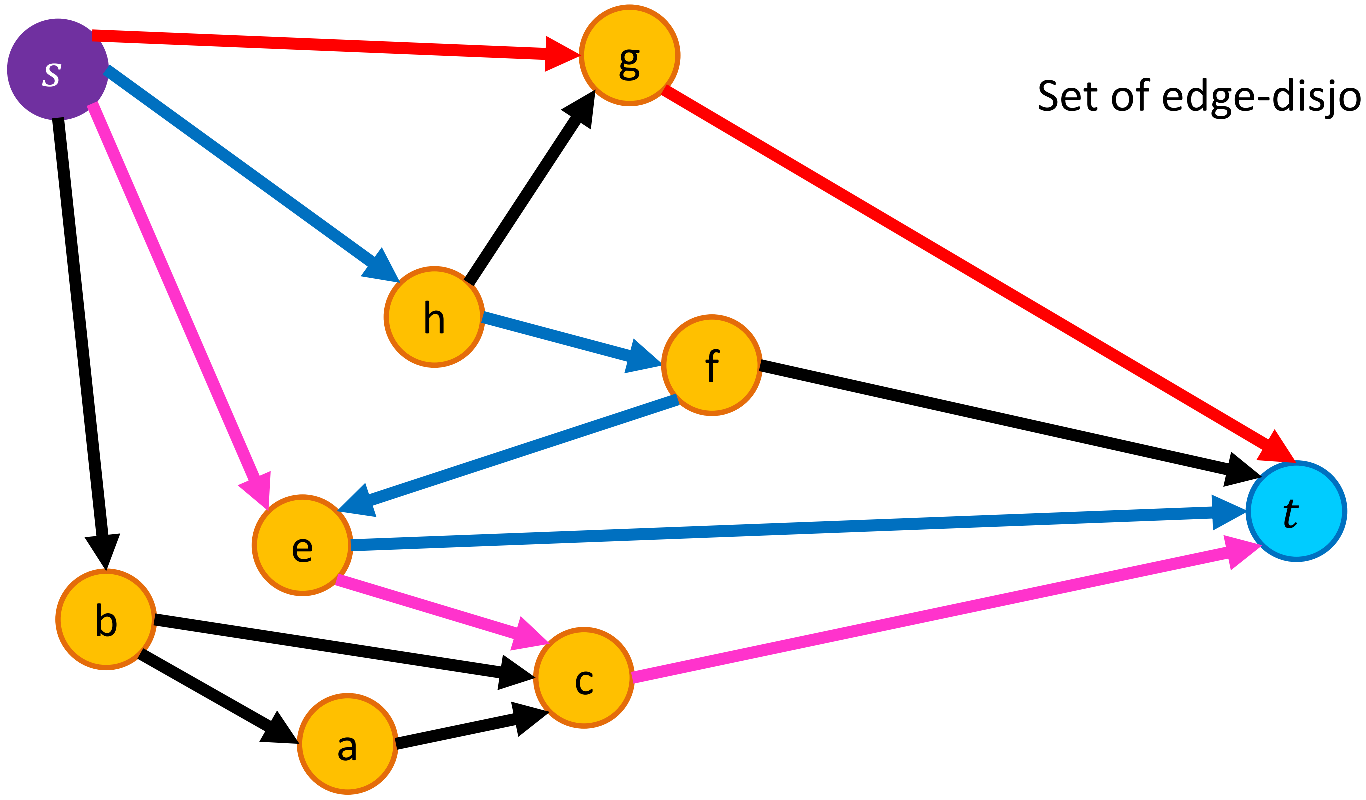
Edge-Disjoint Paths

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



Edge-Disjoint Paths

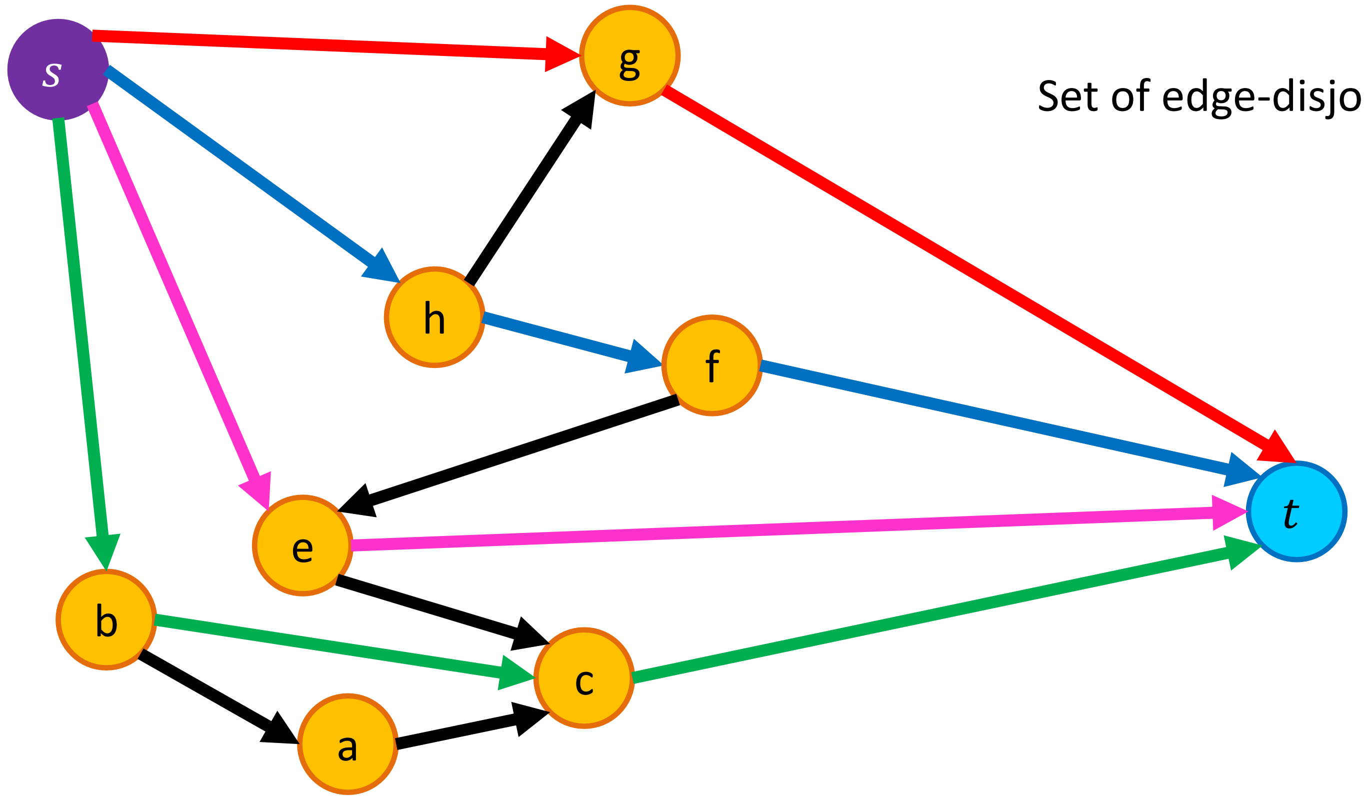
Given a graph $G = (V, E)$, a start node s and a destination node t, give the maximum number of paths from s to t which share no edges



Set of edge-disjoint paths of size 3

Edge-Disjoint Paths

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



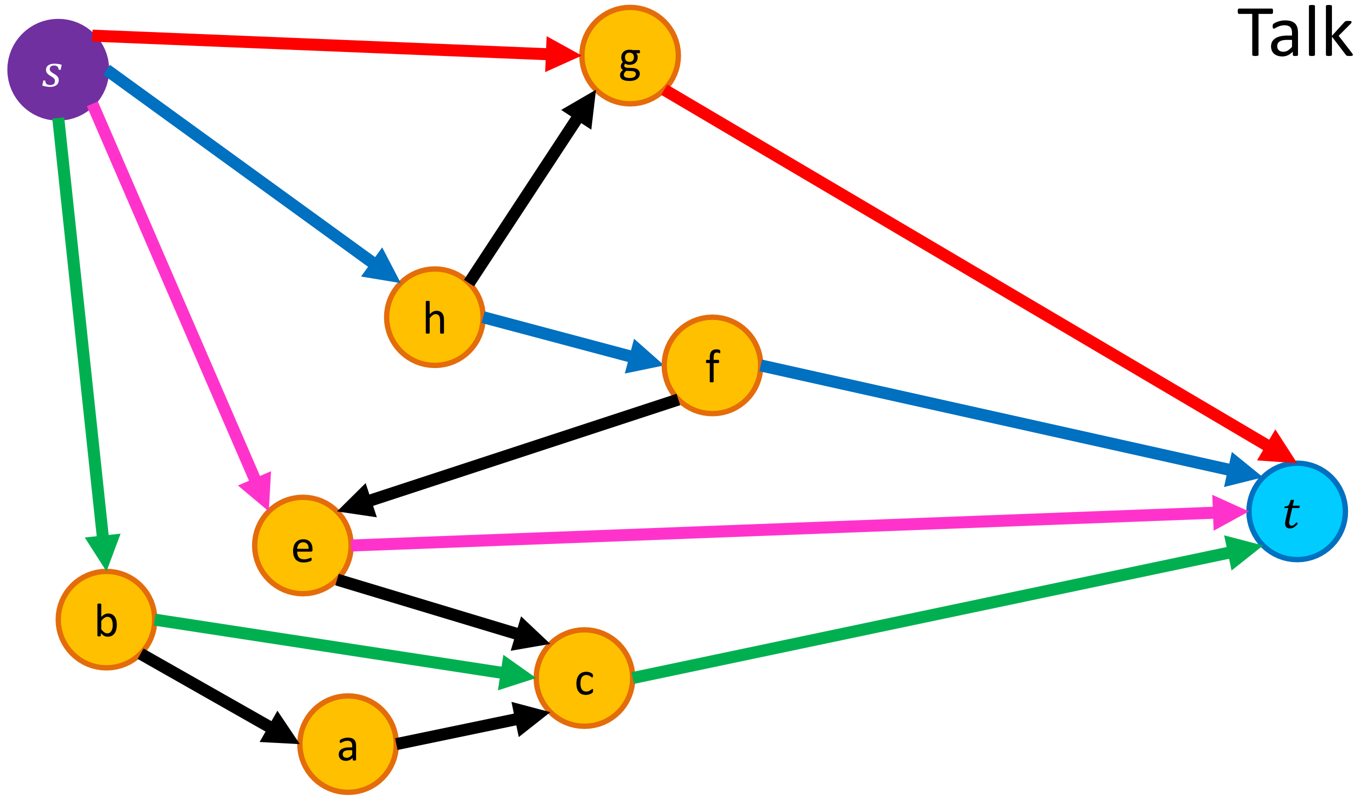
Set of edge-disjoint paths of size 4

Edge-Disjoint Paths

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

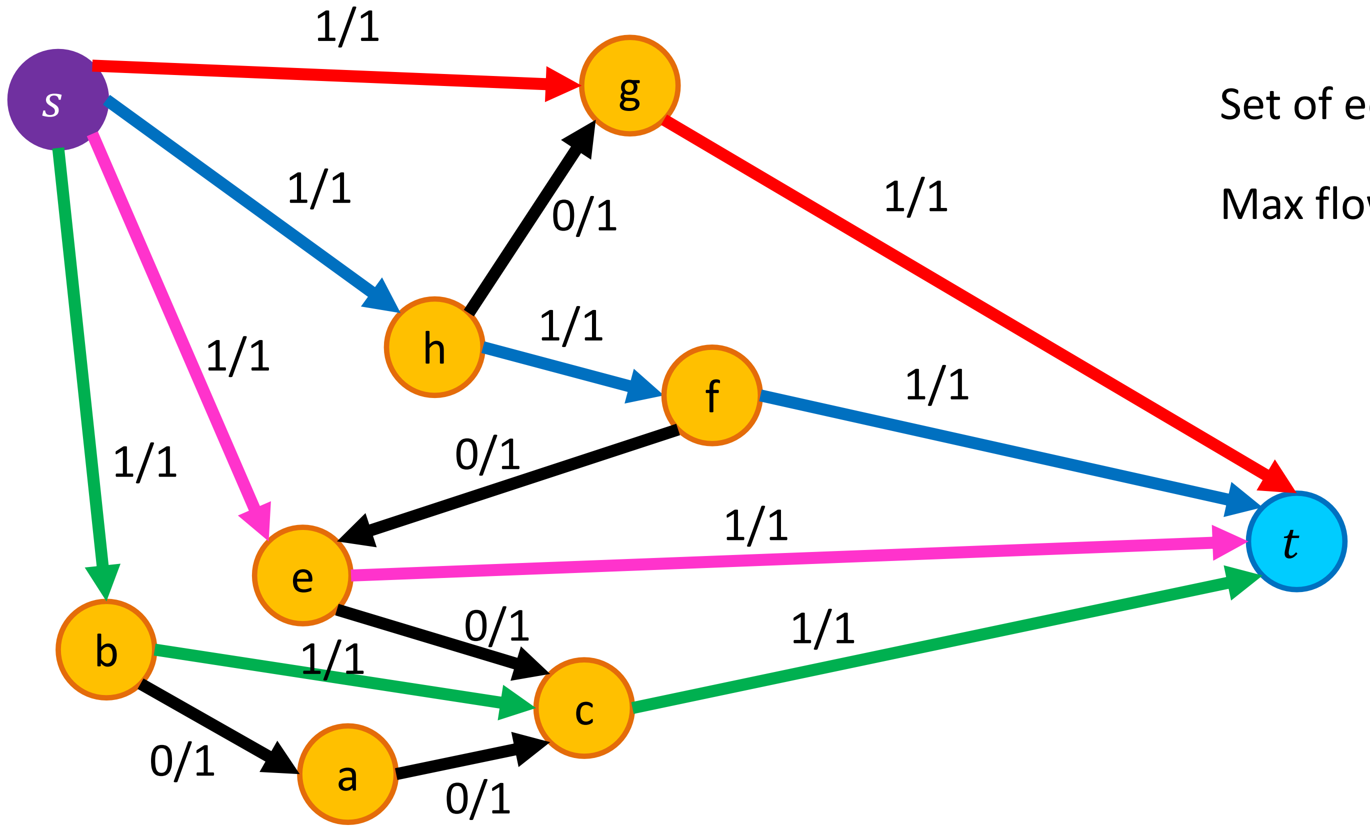
How could we solve this?

Talk with your neighbors!



Edge-Disjoint Paths Algorithm

Make s and t the source and sink, give each edge capacity 1, find the max flow.

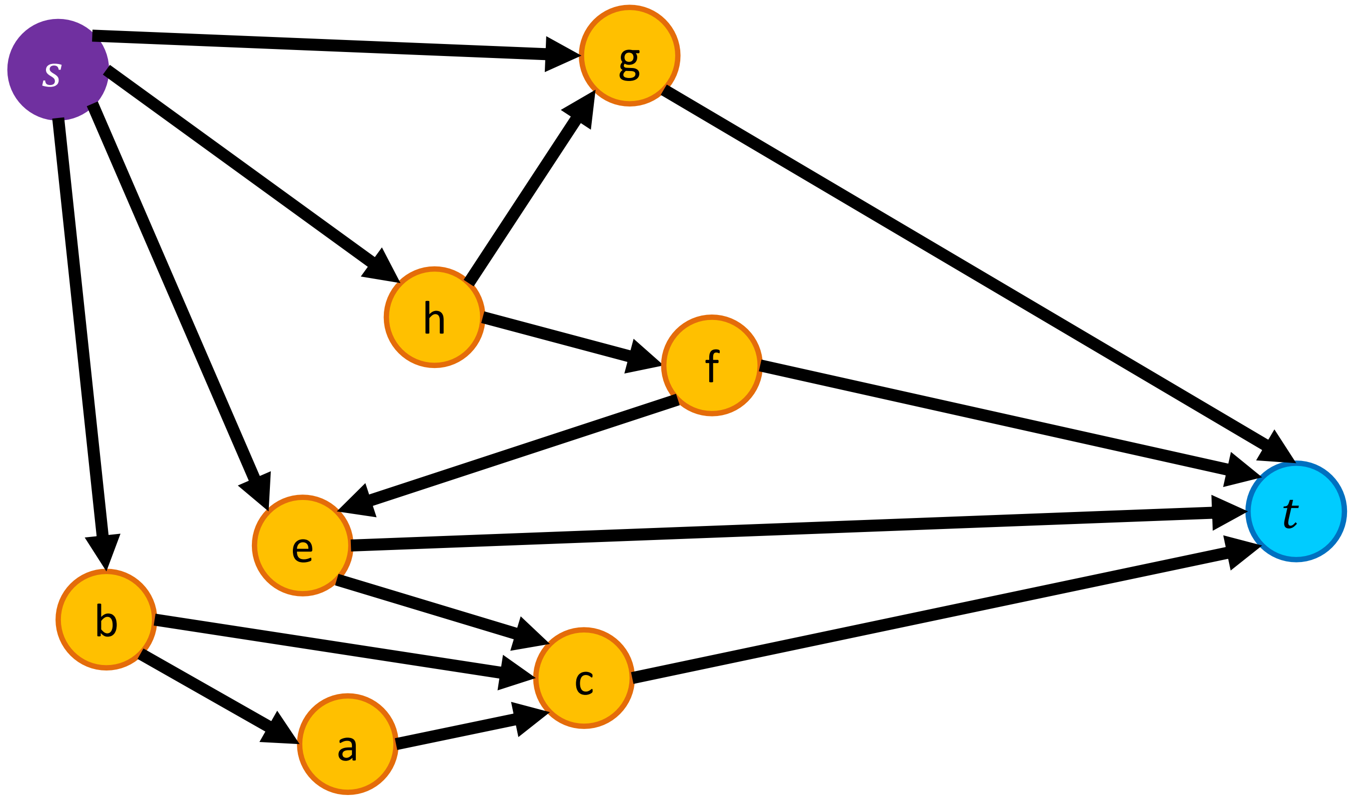


Set of edge-disjoint paths of size 4

Max flow = 4

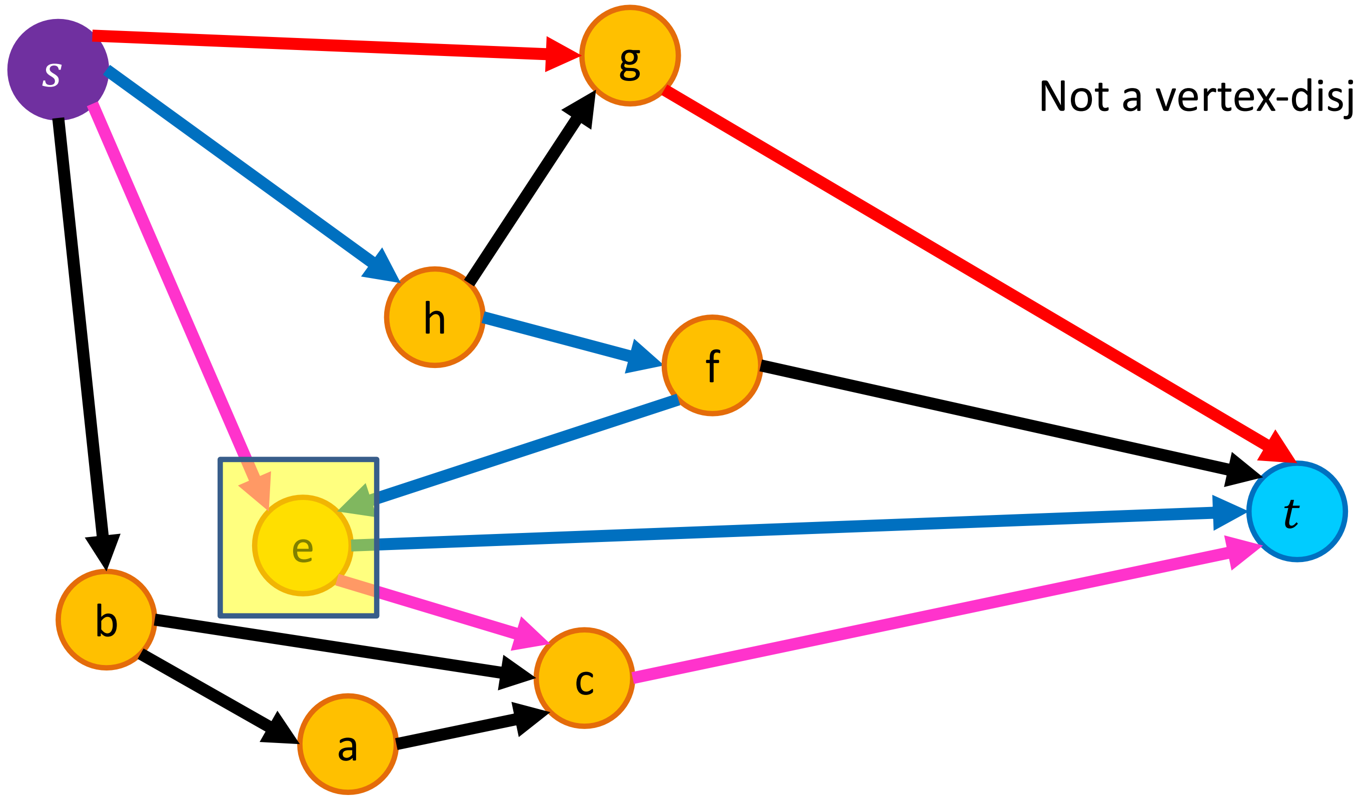
Vertex-Disjoint Paths

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



Vertex-Disjoint Paths

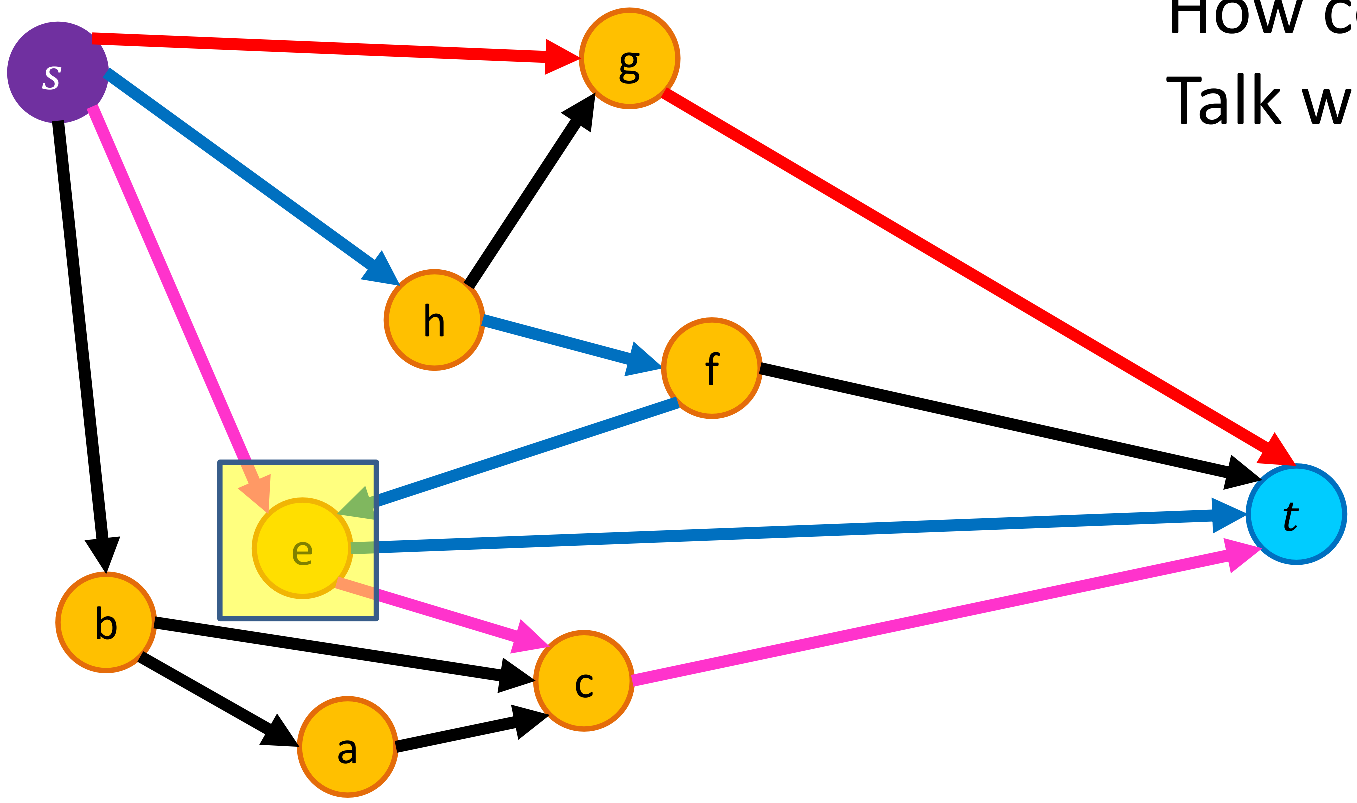
Given a graph $G = (V, E)$, a start node s and a destination node t, give the maximum number of paths from s to t which share no vertices



Not a vertex-disjoint path!

Vertex-Disjoint Paths

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



How could we solve this?

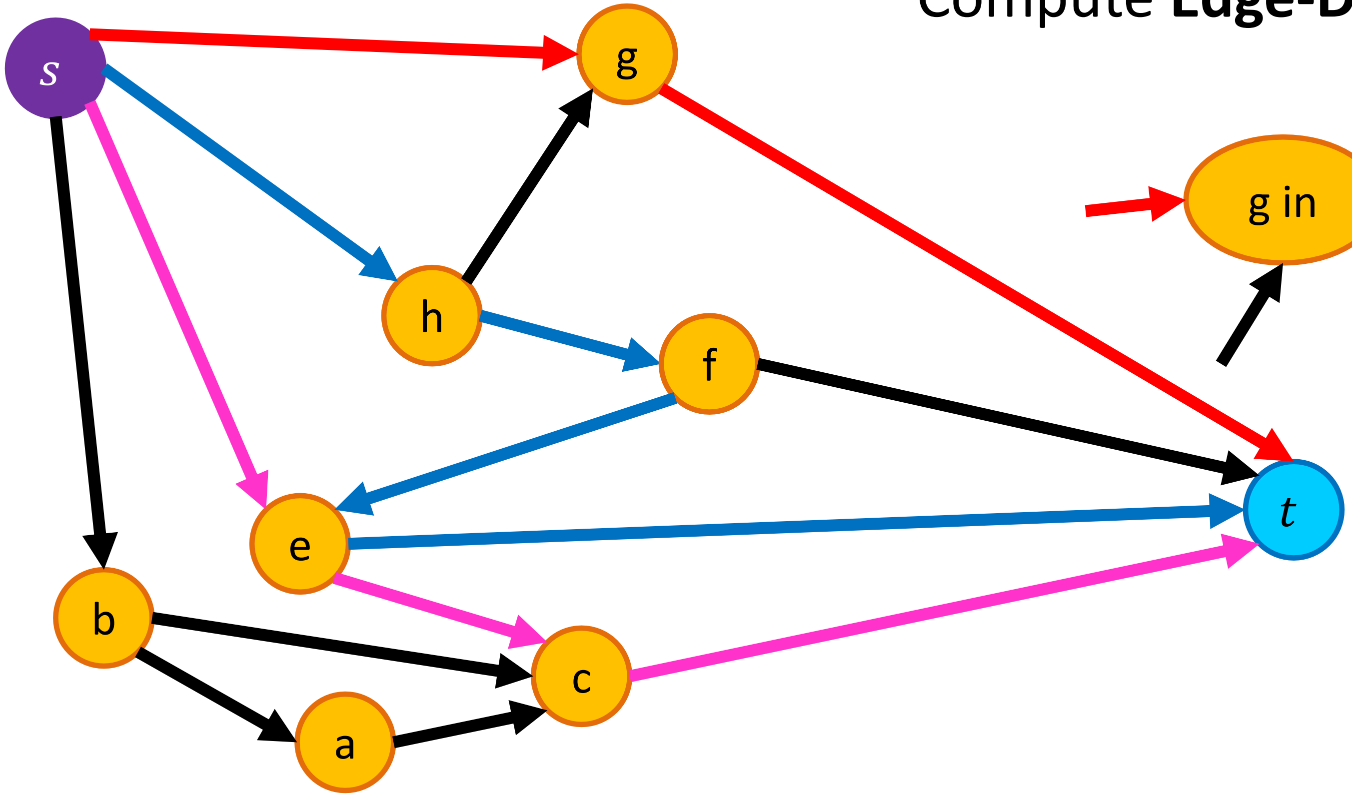
Talk with your neighbors!

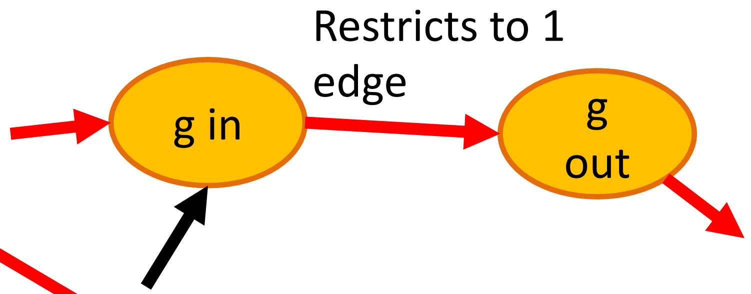
Vertex-Disjoint Paths Algorithm

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

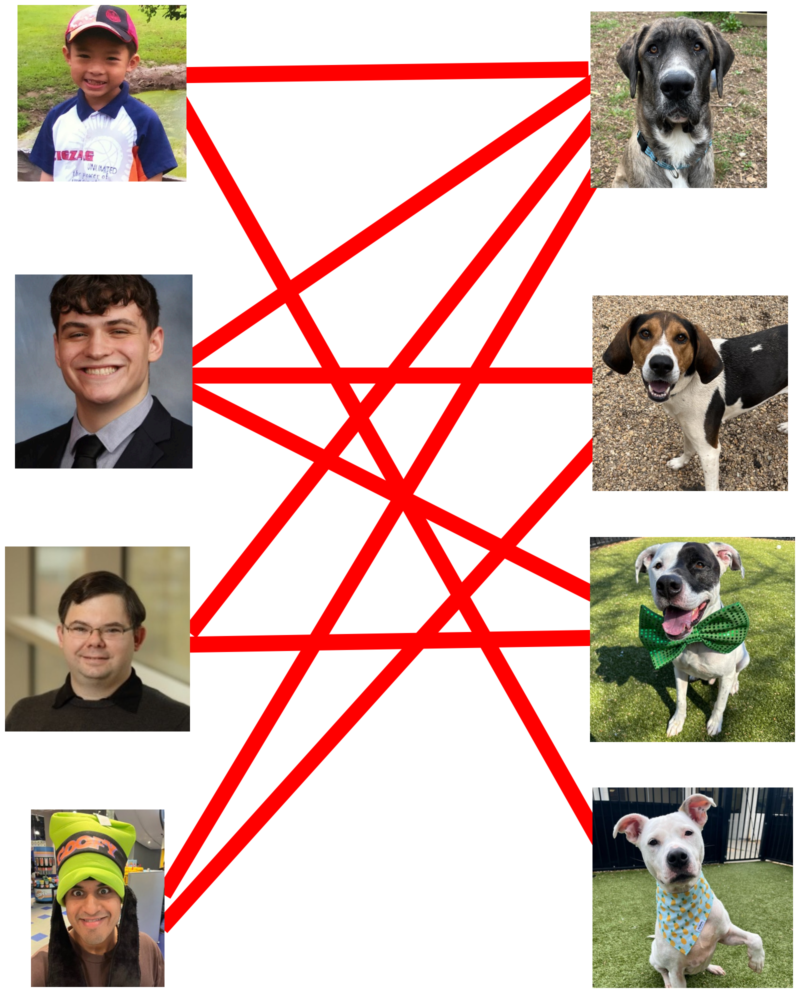




Maximum Bipartite Matching

Dog Lovers

Dogs

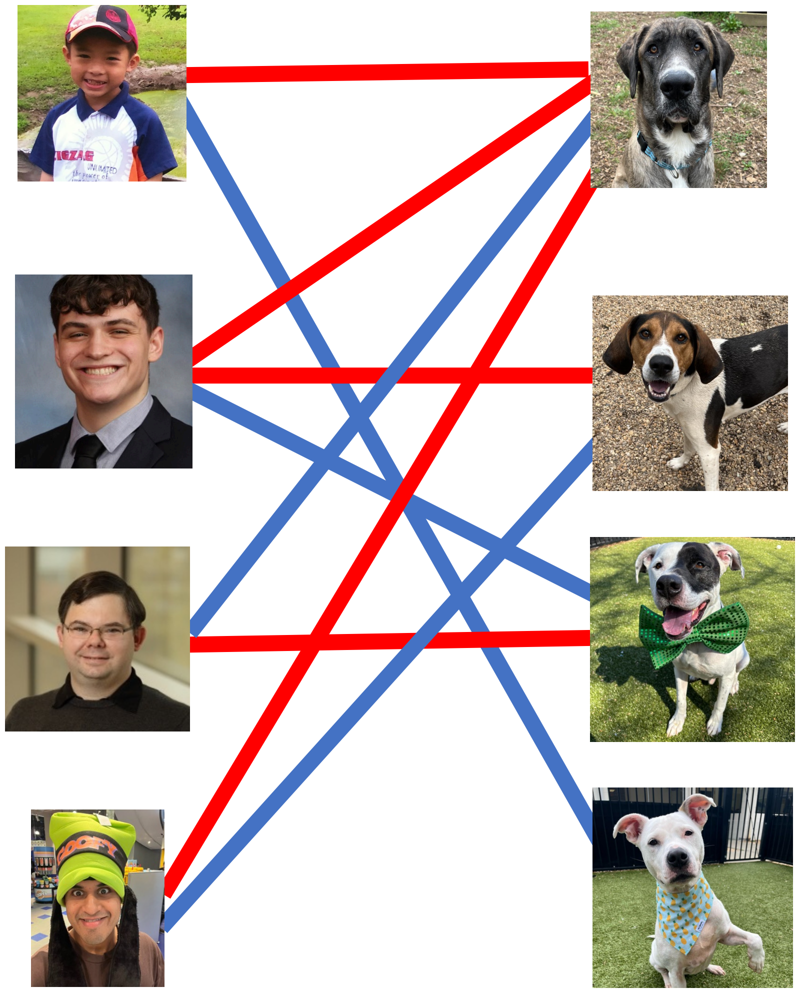


Maximum Bipartite Matching

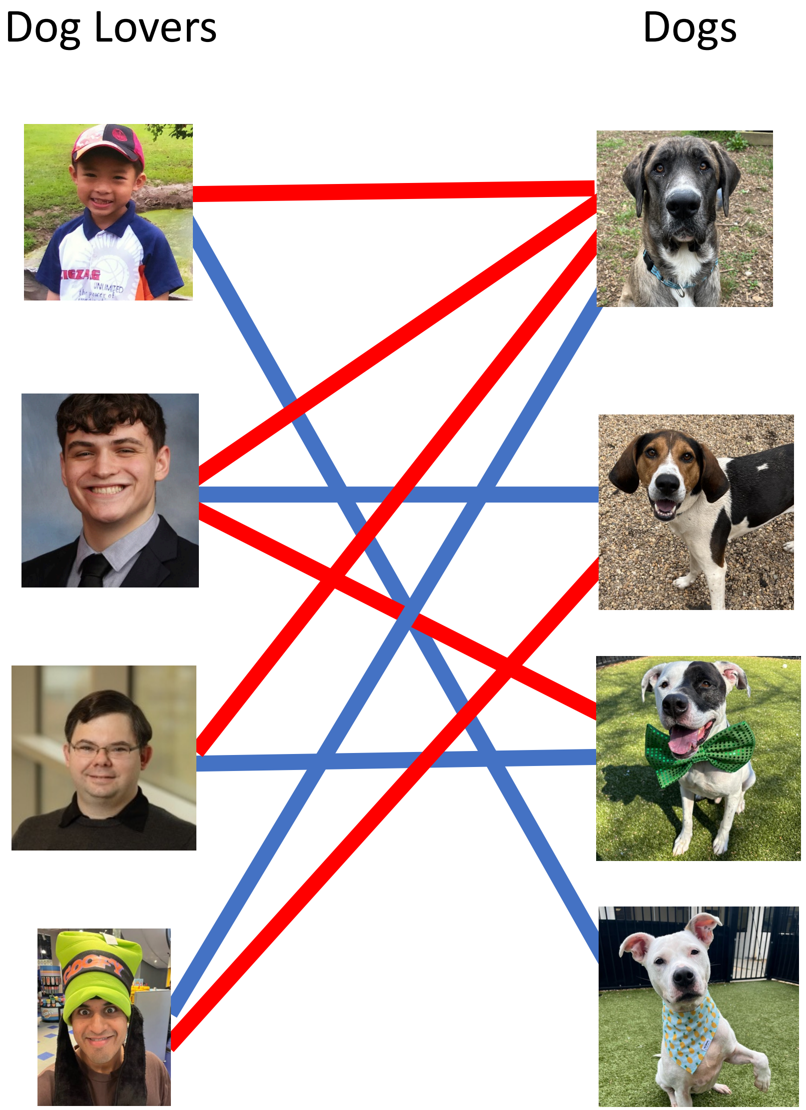
Dog Lovers Dogs

Maximum Bipartite Matching

Dog Lovers | Dogs



Maximum Bipartite Matching



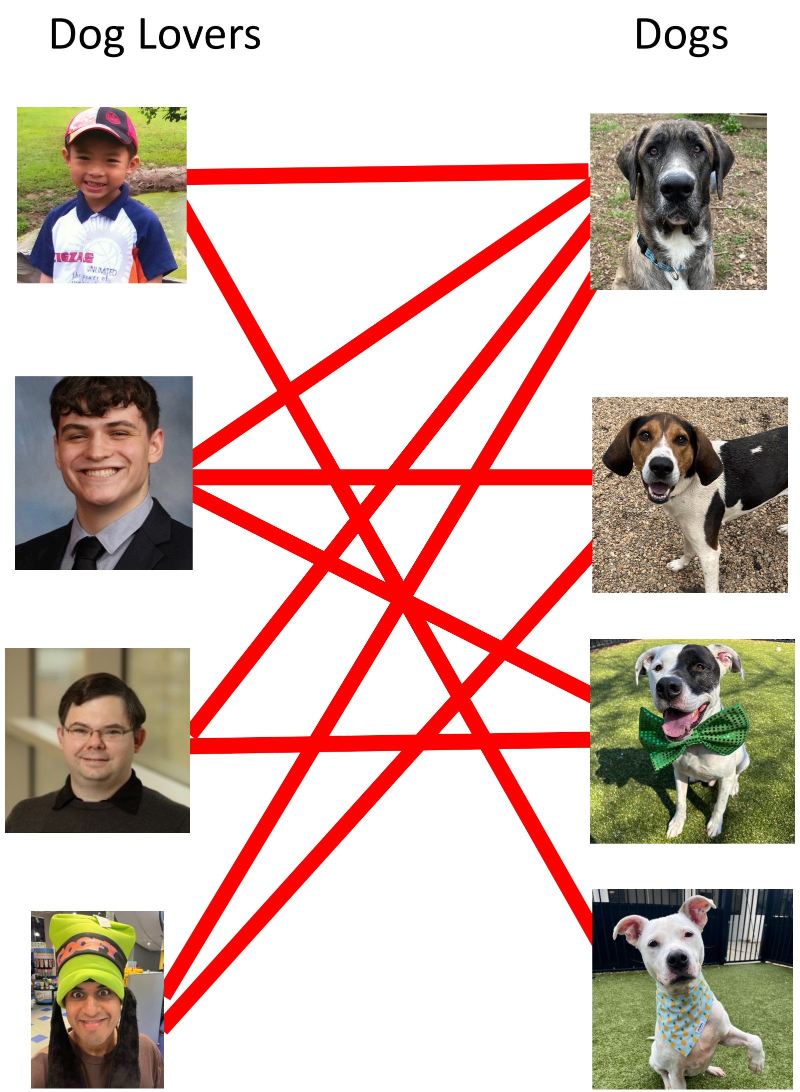
Maximum Bipartite Matching

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



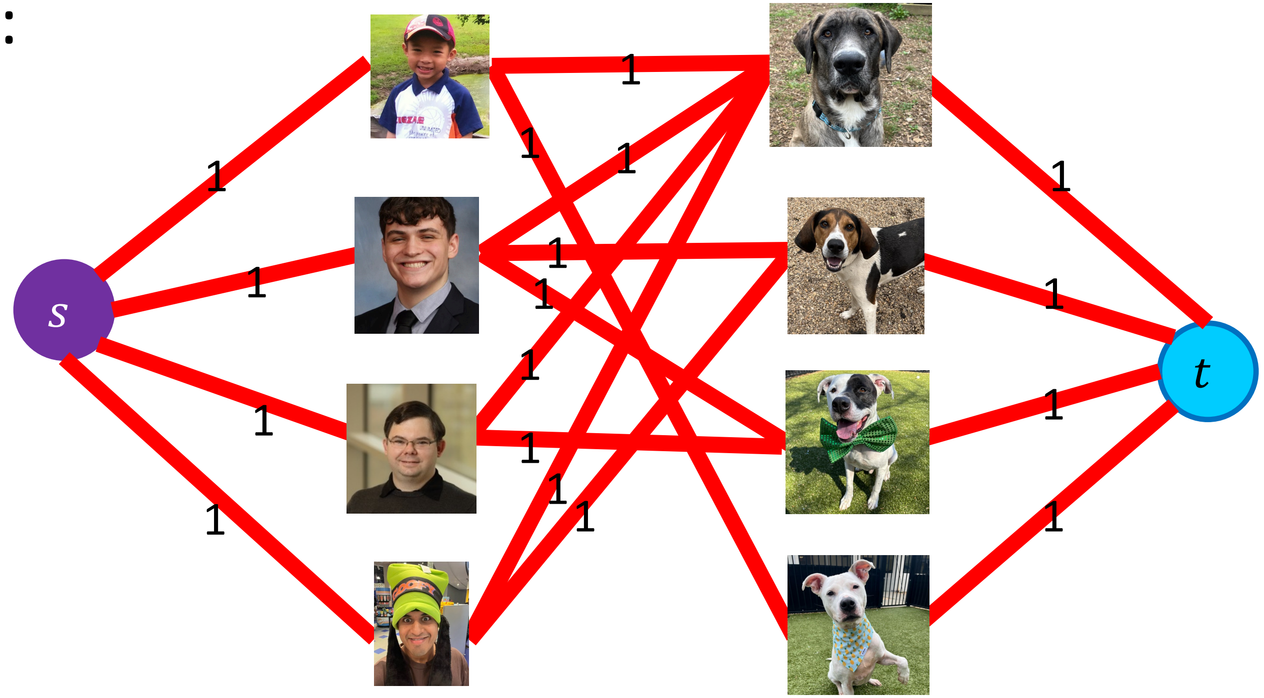
How could we solve this?

Talk with your neighbors!

Maximum Bipartite Matching Using Max Flow

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

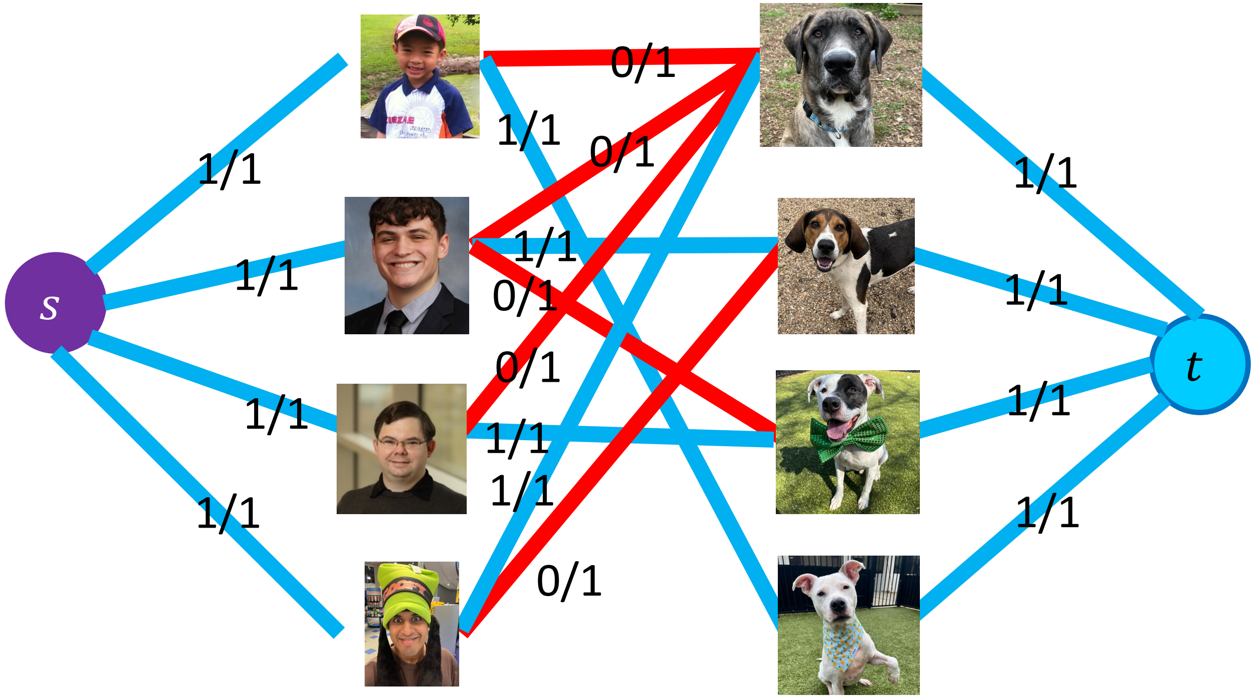
- Adding in a source and sink to the set of nodes:
  - $V' = L \cup R \cup \{s, t\}$
- Adding an edge from source to L and from R to sink:
  - $E' = E \cup \{u \in L \mid (s, u)\} \cup \{v \in r \mid (v, t)\}$
- Make each edge capacity 1:
  - $\forall e \in E', c(e) = 1$



Maximum Bipartite Matching Using Max Flow

$\Theta(E \cdot V)$

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



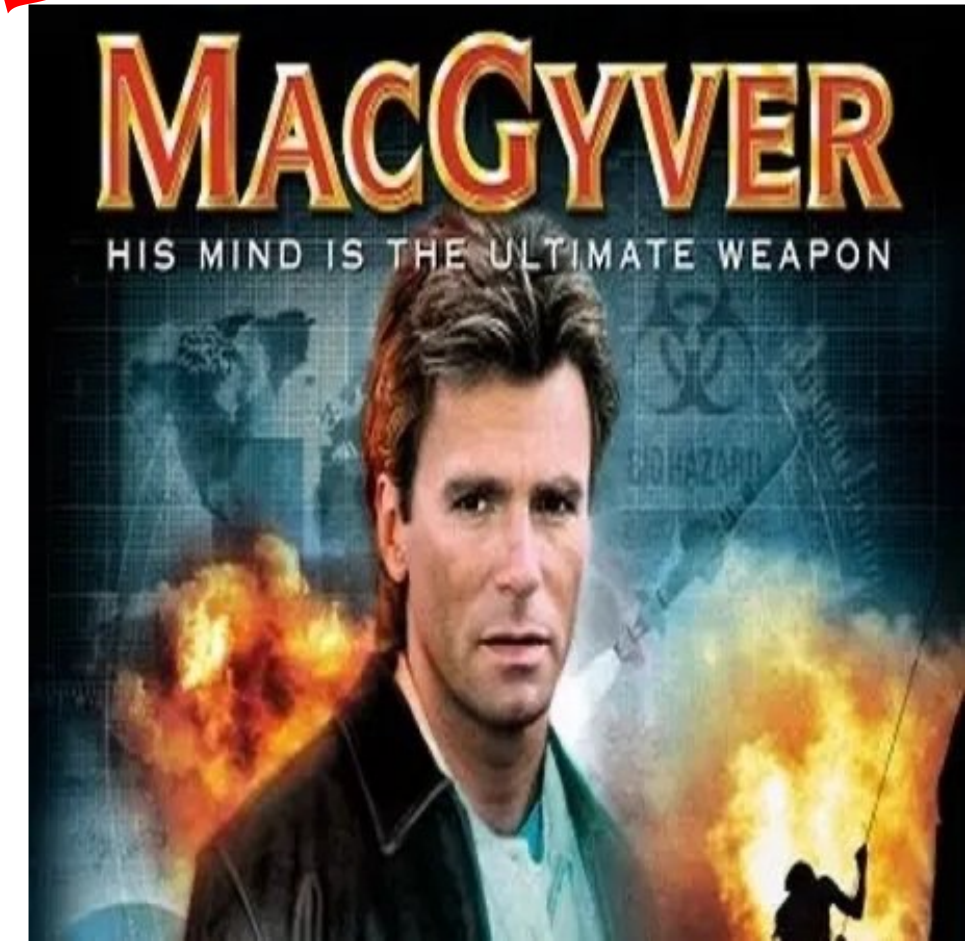
Reductions

- Algorithm technique of supreme ultimate power
- Convert instance of problem A to an instance of Problem B
- Convert solution of problem B back to a solution of problem A

Reductions

Shows how two different problems relate to each other

MOVIE TIME!



MacGyver's Reduction

Problem we don't know how to solve

A

Opening a door

Solution for $\boldsymbol{A}$

Keg cannon
battering ram

Aim duct at door,
insert keg

Put fire under the Keg

Reduction

Problem we do know how to solve

B

Lighting a fire

How?

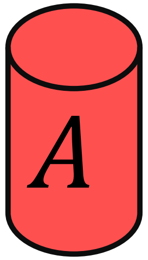
Solution for $\boldsymbol{B}$

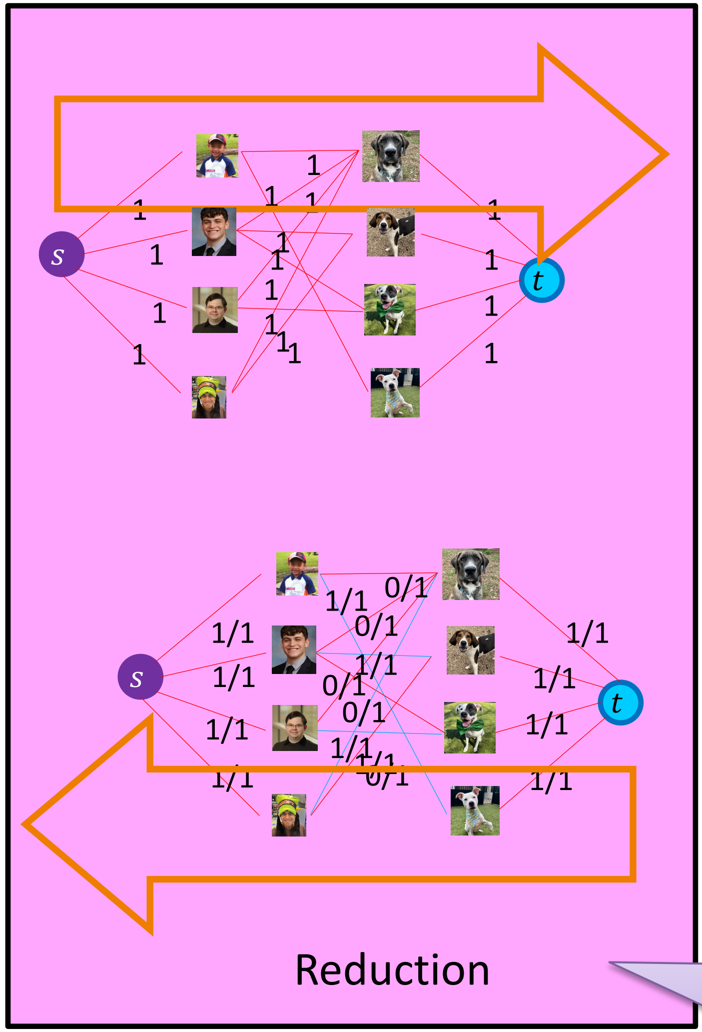
Alcohol, wood,
matches

Bipartite Matching Reduction

Problem we don't know how to solve

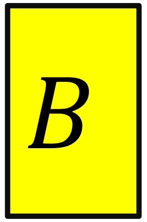
Bipartite Matching

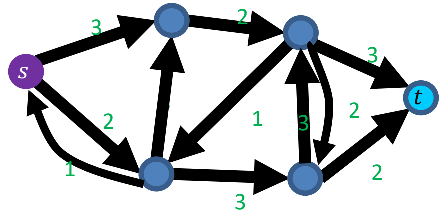




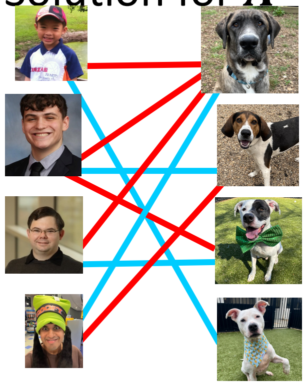
Problem we do know how to solve

Max Flow

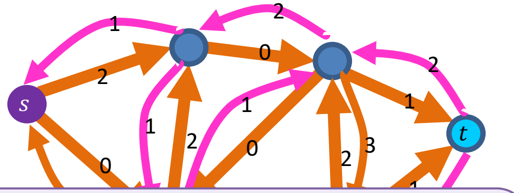




Solution for $\boldsymbol{A}$

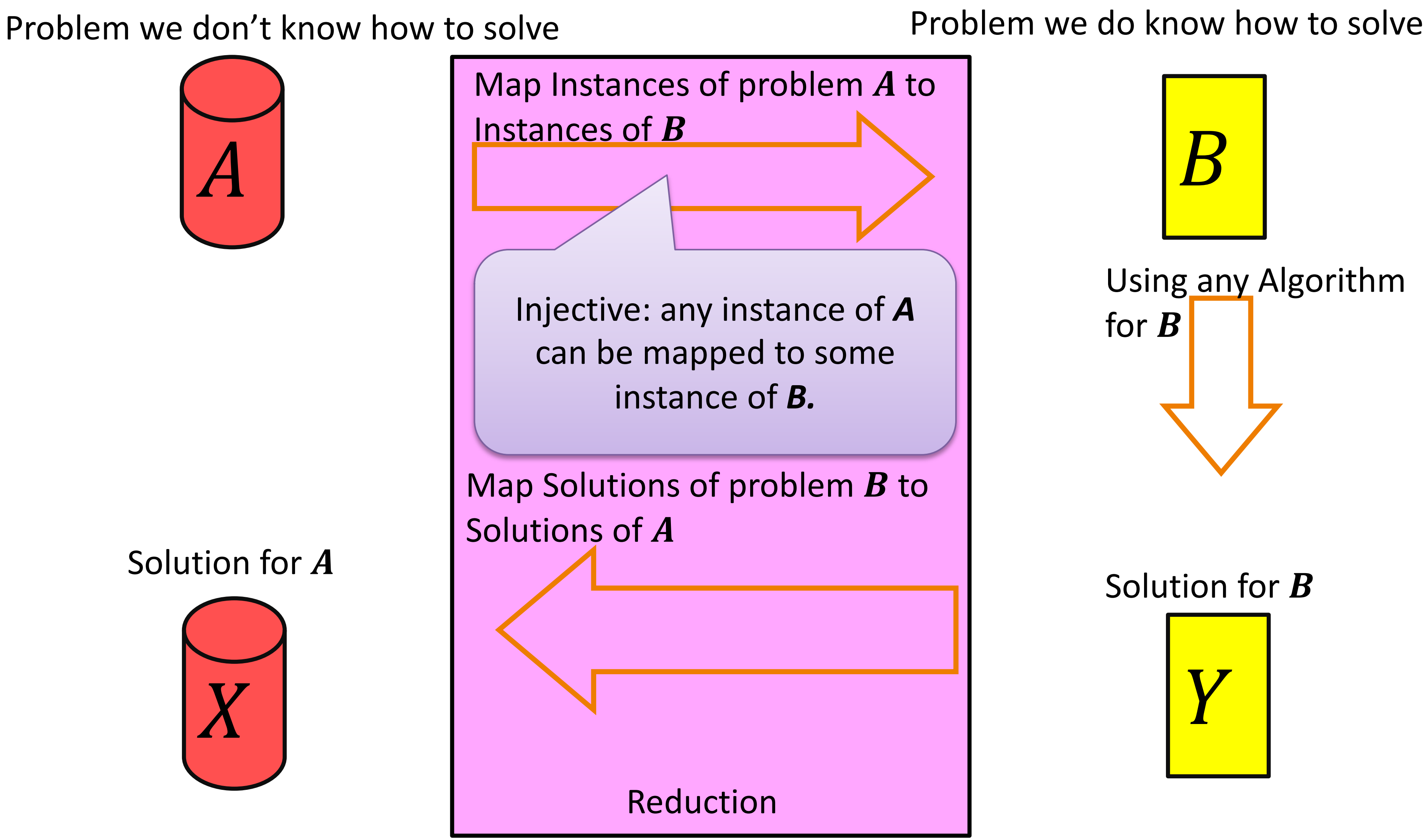


Solution for $\boldsymbol{B}$



Must show (prove):
1) how to make construction
2) Why it works

In General: Reduction



Worst-case lower-bound Proofs

Opening a door

A

Problem **A**

reduces to

B

Problem **B**

Lighting a fire

Alcohol, wood,
matches

Y

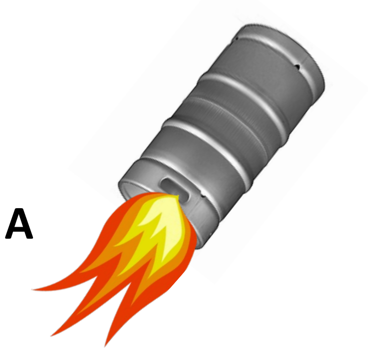
Algorithm for **B**

can be used
to make

X

Algorithm for **A**

Keg cannon
battering ram



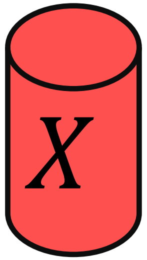
$\boldsymbol{A}$ is not a harder problem than $\boldsymbol{B}$

$$\boldsymbol{A \leq B}$$

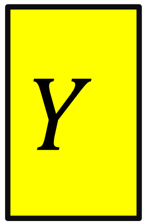
The name “reduces” is confusing: it is in the *opposite* direction of the making

Proof of Lower Bound by Reduction

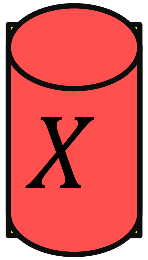
To Show: Y is slow



1. We know X is slow (by a proof)
(e.g., X = some way to open the door)



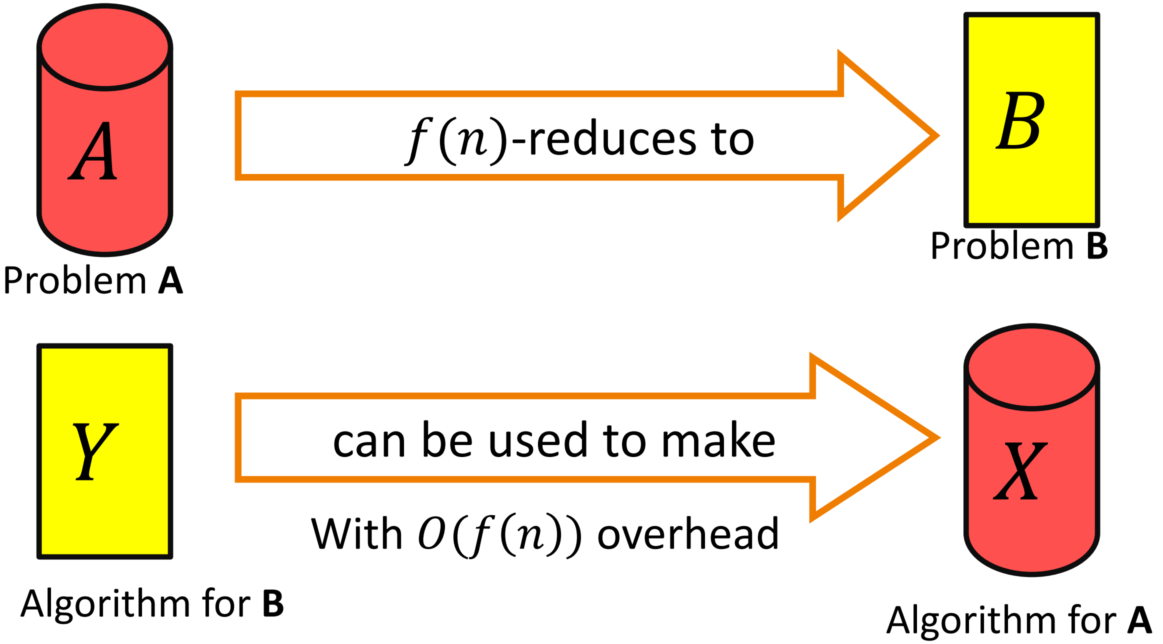
2. Assume Y is quick [toward contradiction]
(Y = some way to light a fire)



3. Show how to use Y to perform X quickly

4. X is slow, but Y could be used to perform X quickly
conclusion: Y must not actually be quick

Reduction Proof Notation



A is not a harder problem than B

$$A \leq B$$

If A requires time $\Omega(f(n))$ time then B also requires $\Omega(f(n))$ time

$$A \leq_{f(n)} B$$