CS4102 Algorithms

Spring 2019

Just Kidding!

Come taste-test a cookie!
I baked cookies for you all this weekend.
Start with 2 cookies, come to office hours for more.

Reminder: Residual Graph G_f

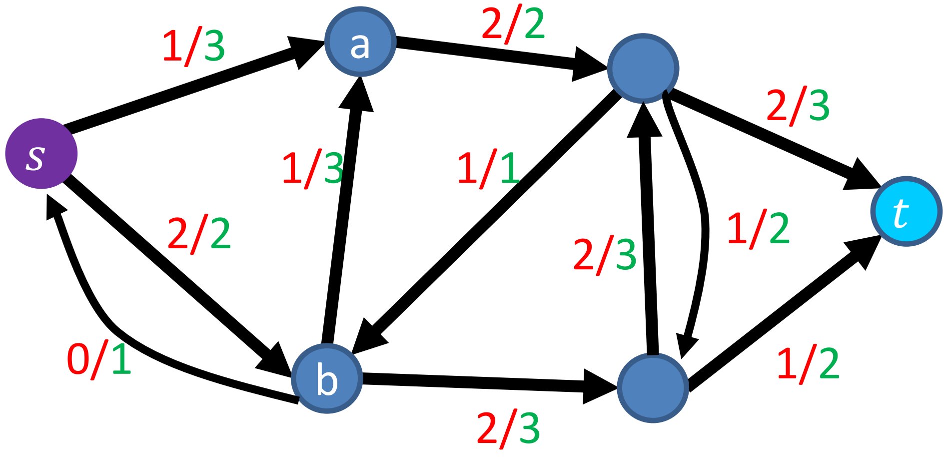
- Keep track of net available flow along each edge
- "Forward edges": weight is equal to available flow along that edge in the flow graph
  - $w(e) = c(e) - f(e)$

Flow I *could* add

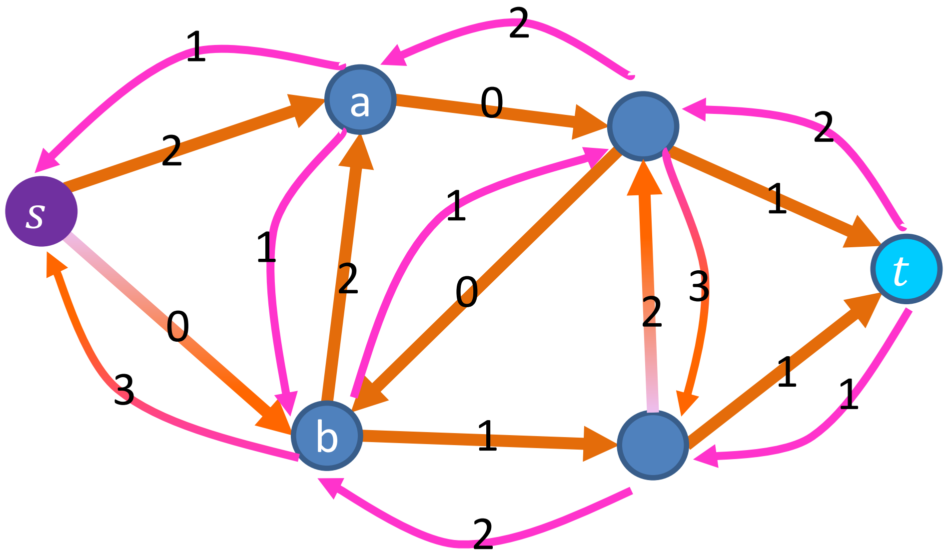
- "Back edges": weight is equal to flow along that edge in the flow graph
  - $w(e) = f(e)$

Flow I *could* remove

Flow Graph G



Residual Graph G_f



Today's Keywords

- Reductions
- Bipartite Matching
- Vertex Cover
- Independent Set

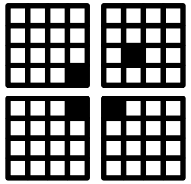
CLRS Readings

- Chapter 34

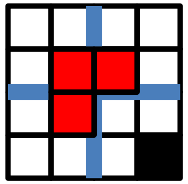
Homeworks

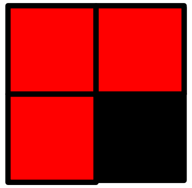
- HW8 due Tomorrow, 4/23, at 11pm
  - Python or Java
  - Tiling Dino
- HW9 out today, due Monday 4/29 at 11pm
  - Graphs, Reductions
  - Written (LaTeX)

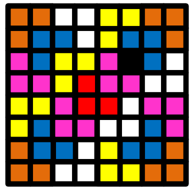
Divide and Conquer*



- **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

Dynamic Programming

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

Greedy Algorithms

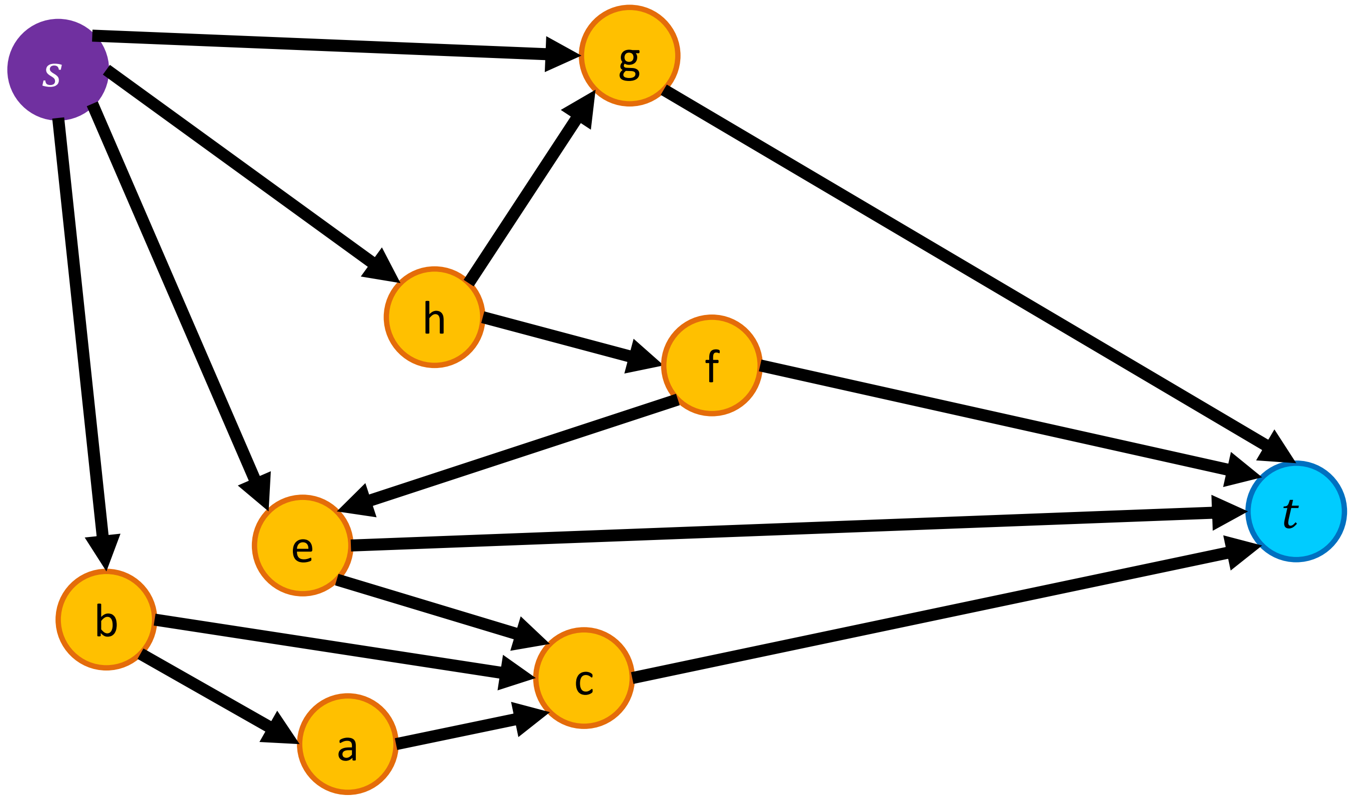
- Require Optimal Substructure
  - Solution to larger problem contains the solution to a smaller one
  - Only one subproblem to consider!
- Idea:
  1. Identify a greedy choice property
     - How to make a choice guaranteed to be included in some optimal solution
  2. Repeatedly apply the choice property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

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

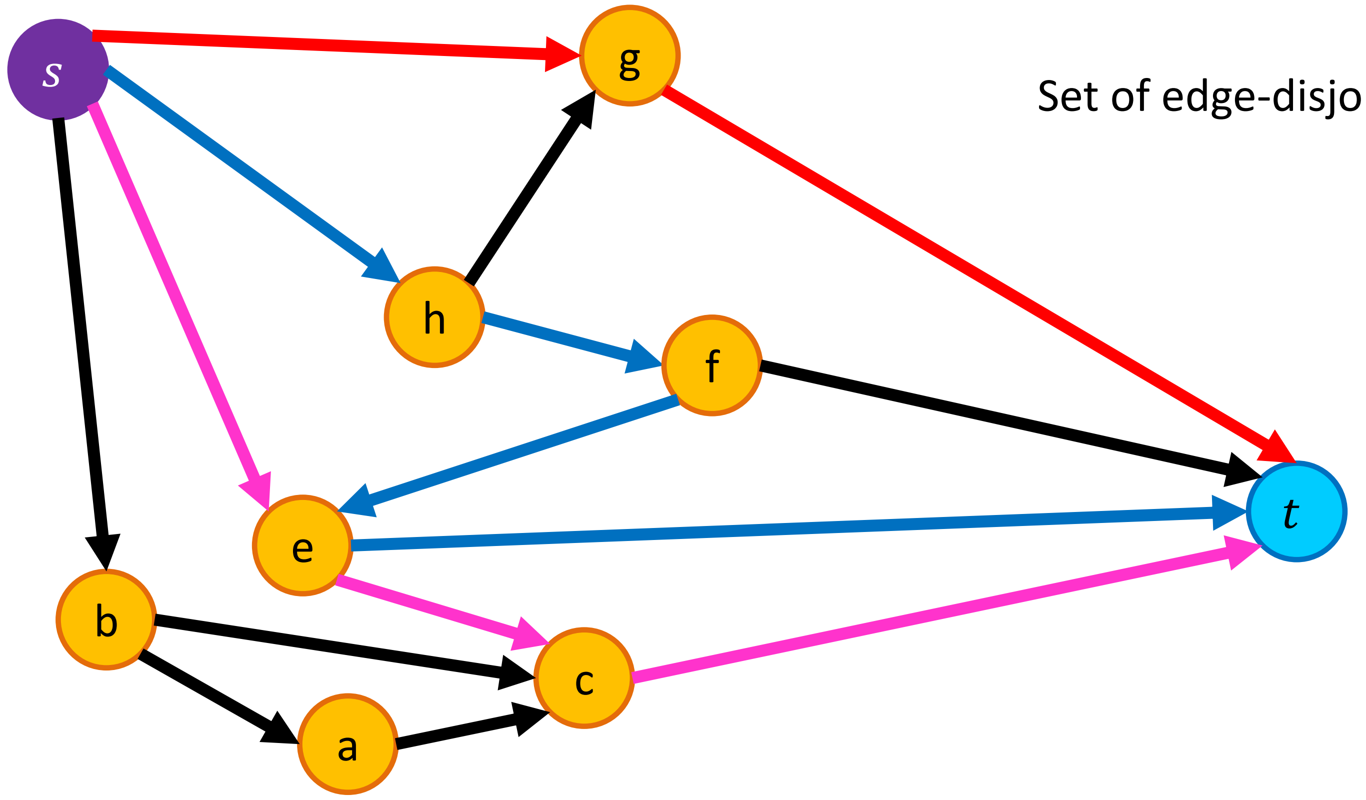
Edge-Disjoint Paths

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

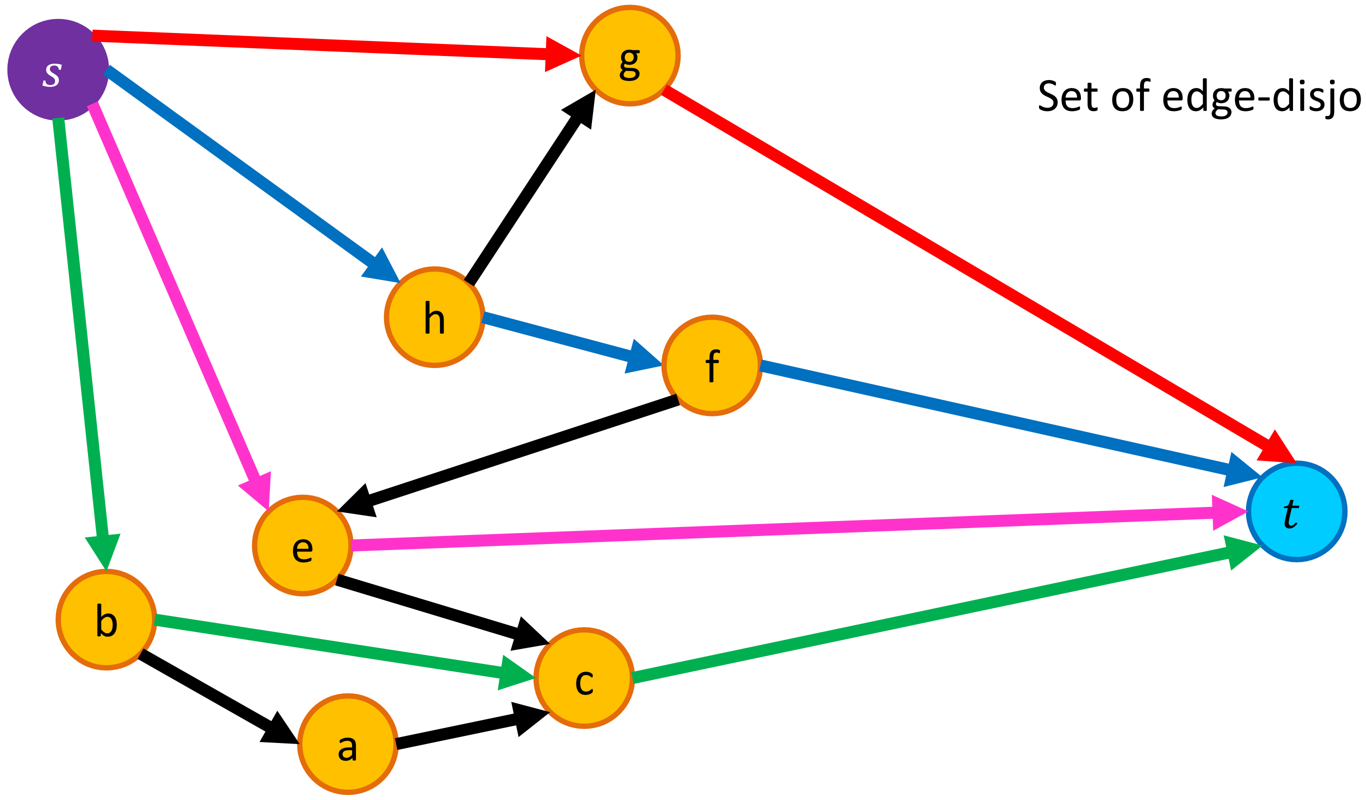
Given a graph $G = (V, E)$, a start node s and a destination node t, give the maximum number of paths from s to t which share no edges



Set of edge-disjoint paths of size 3

Edge-Disjoint Paths

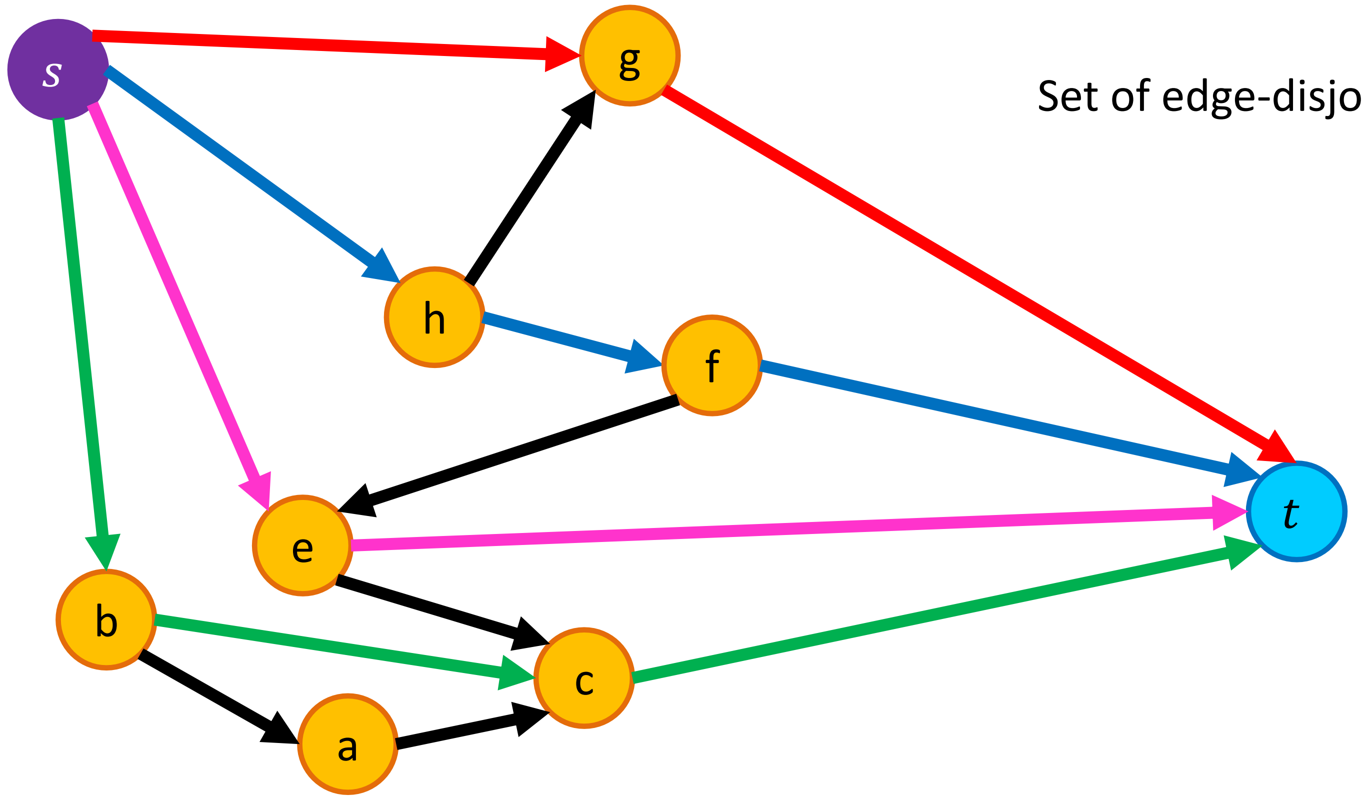
Given a graph $G = (V, E)$, a start node s and a destination node t, give the maximum number of paths from s to t which share no edges



Set of edge-disjoint paths of size 4

Edge-Disjoint Paths Algorithm

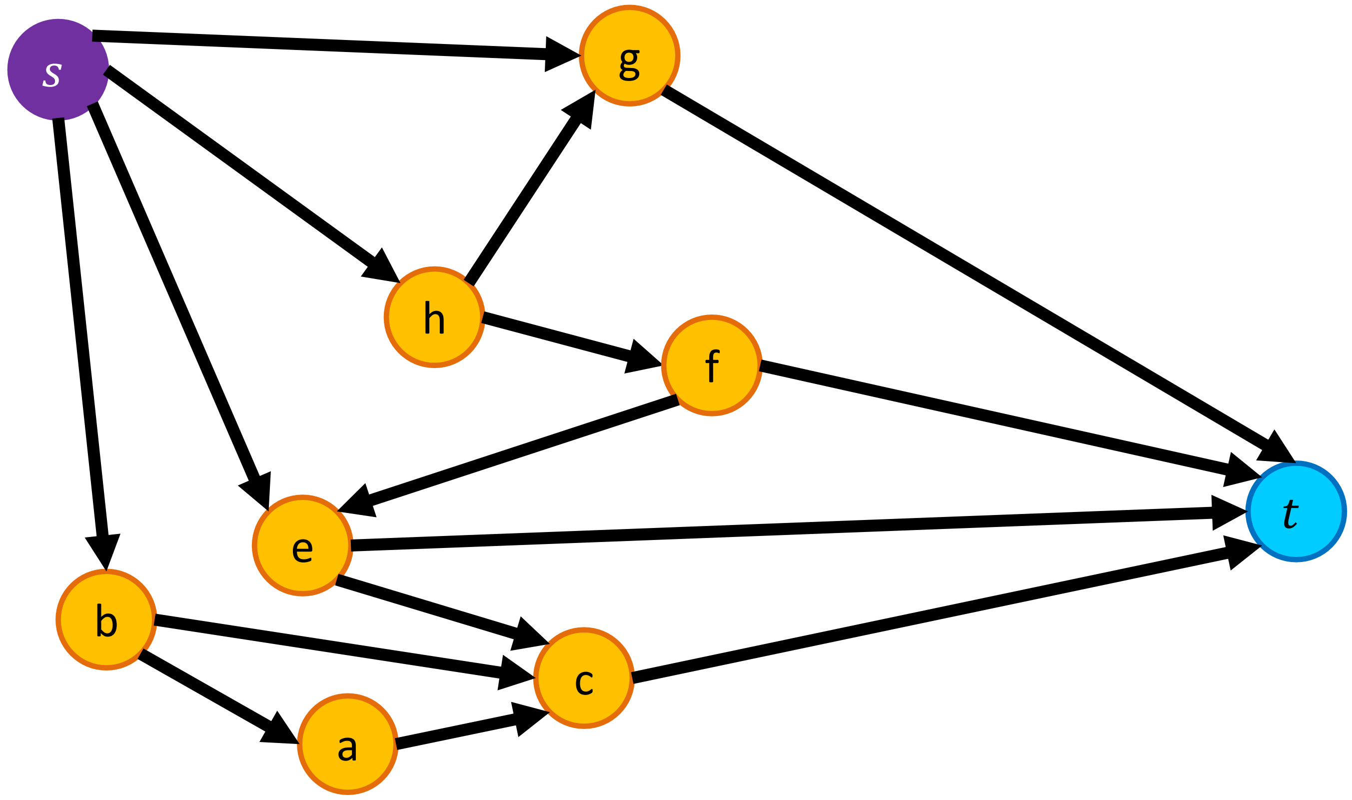
Make s and t the source and sink, give each edge capacity 1, find the max flow.



Set of edge-disjoint paths of size 4

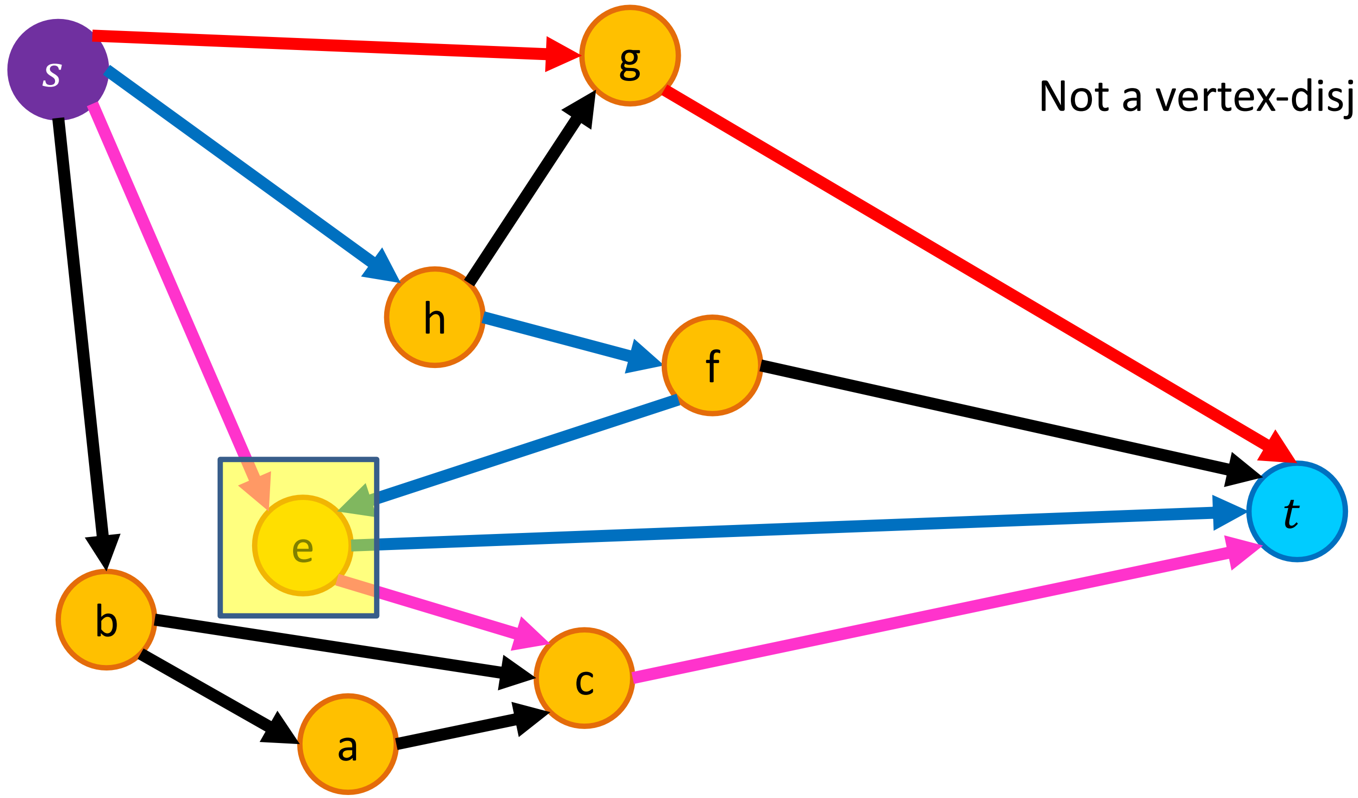
Vertex-Disjoint Paths

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



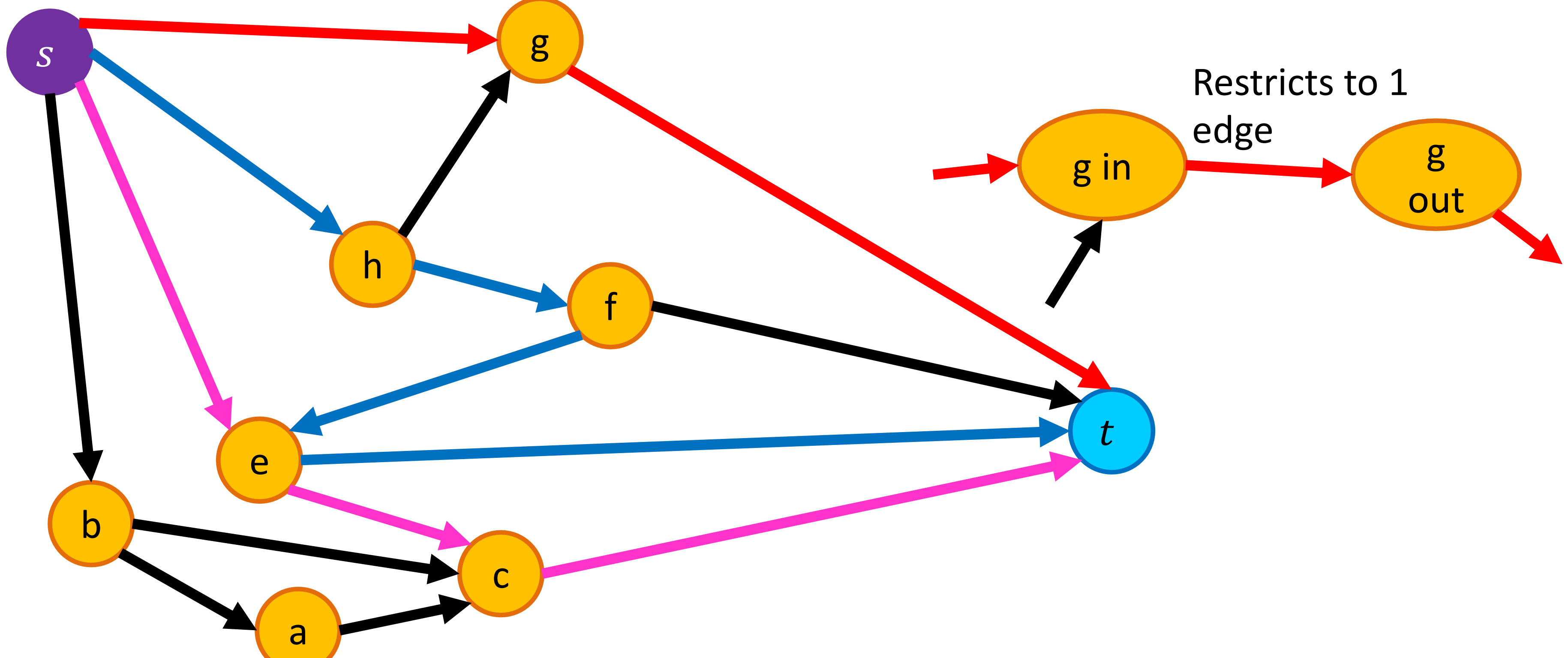
Not a vertex-disjoint path!

Vertex-Disjoint Paths Algorithm

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

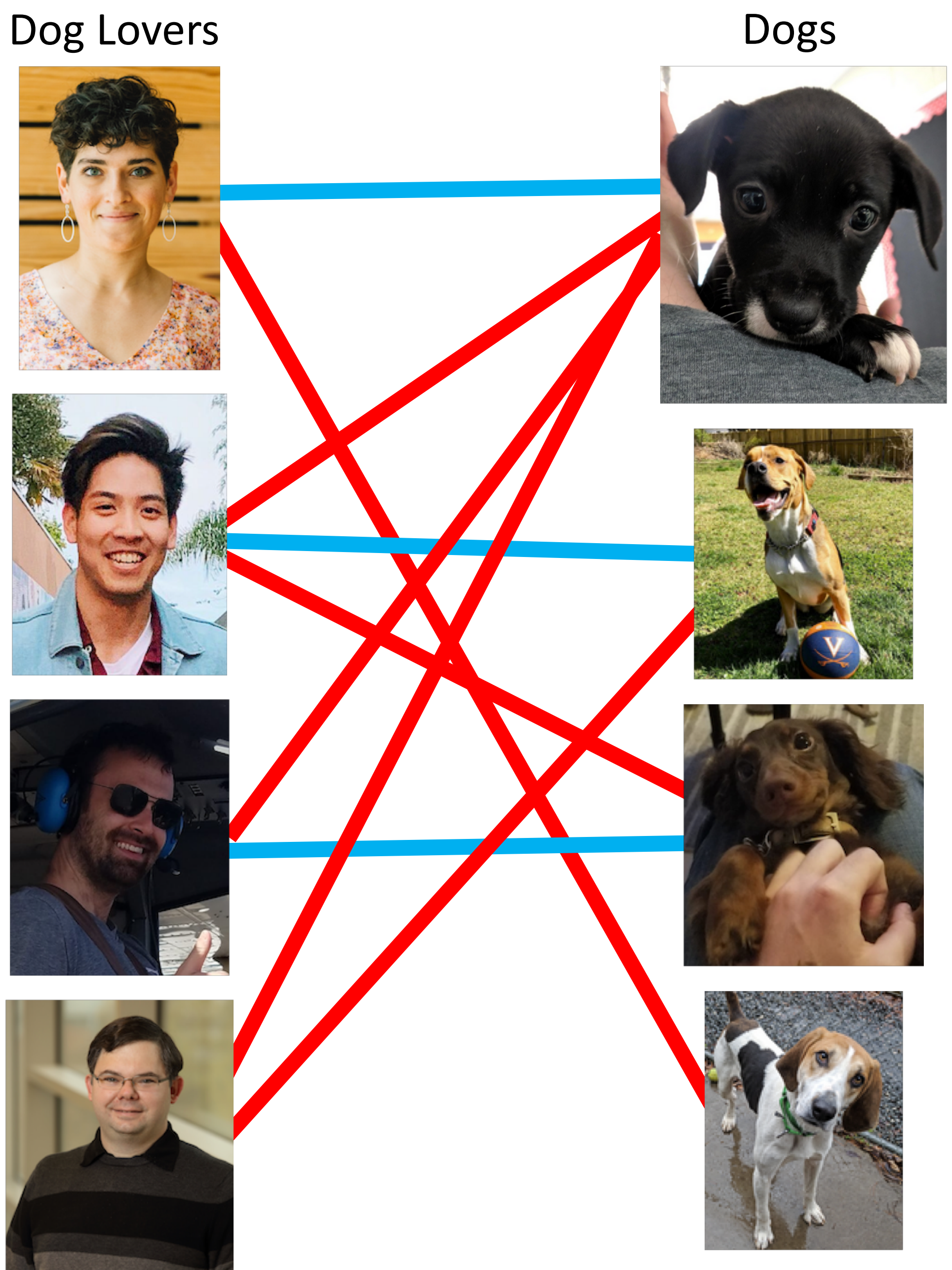
Make two copies of each node, one connected to incoming edges, the other to outgoing edges

Compute Edge-Disjoint paths on new graph

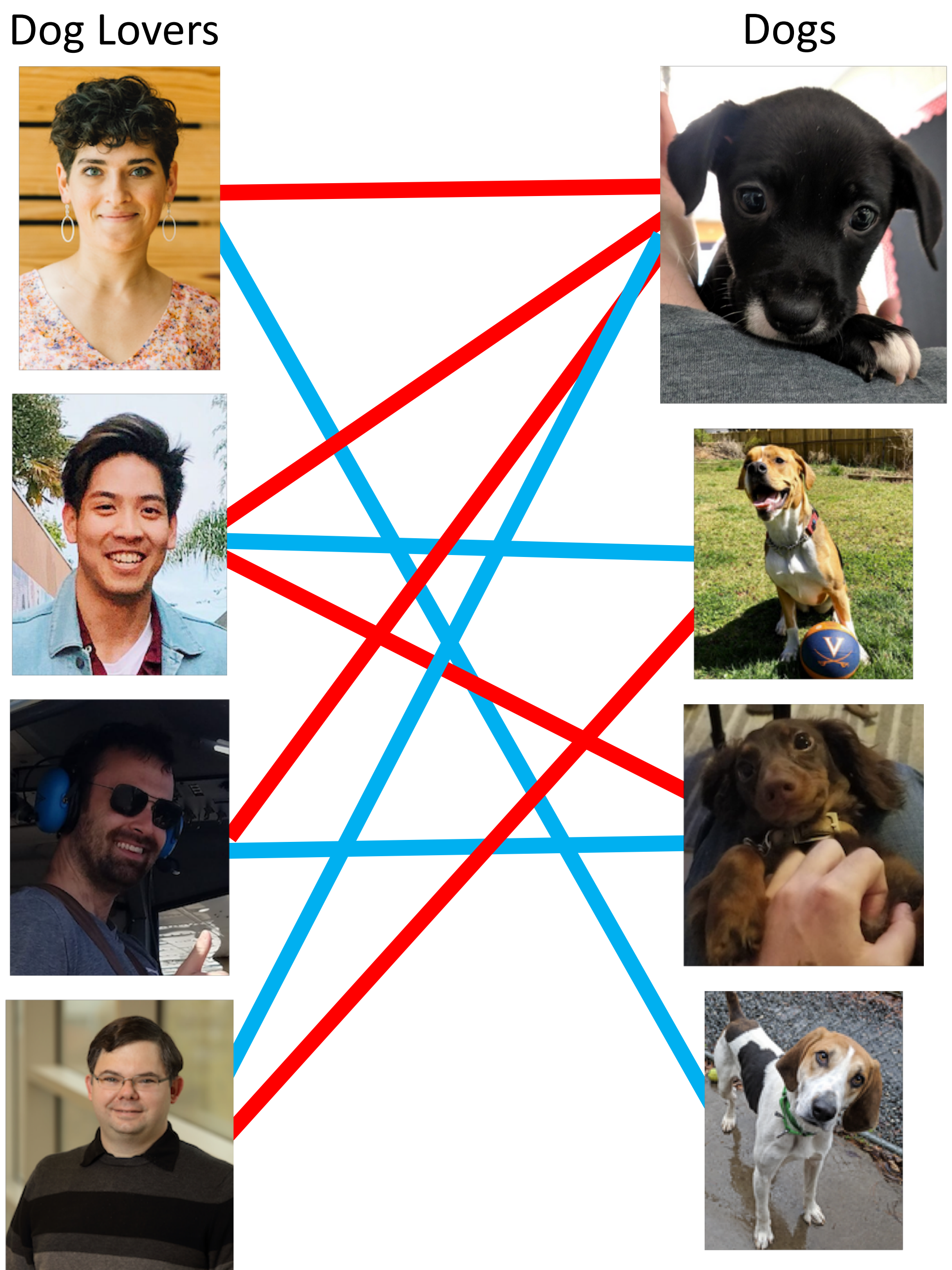


Maximum Bipartite Matching

Maximum Bipartite Matching



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 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:
  - $\forall e \in E', c(e) = 1$

Run Time $\Theta(E \cdot V)$

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 $\Theta(L + R)$

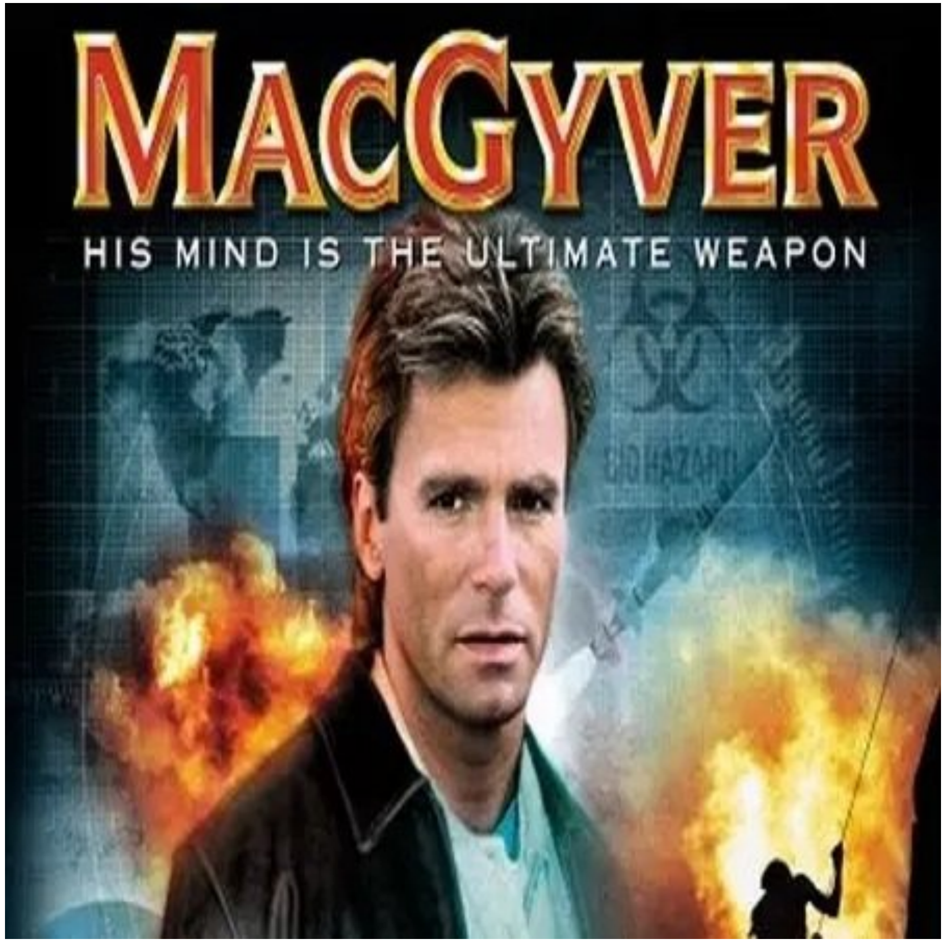
Reductions

- 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

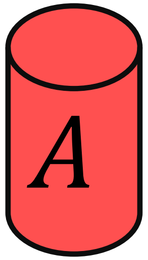
Shows how two different problems relate to each other

MOVIE TIME!



MacGyver's Reduction

Problem we don't know how to solve



Opening a door

Aim duct at door, insert keg

Problem we do know how to solve

B

Lighting a fire

How?

Solution for $\boldsymbol{A}$

Keg cannon battering ram

Put fire under the Keg

Reduction

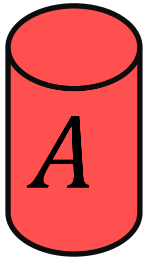
Solution for $\boldsymbol{B}$

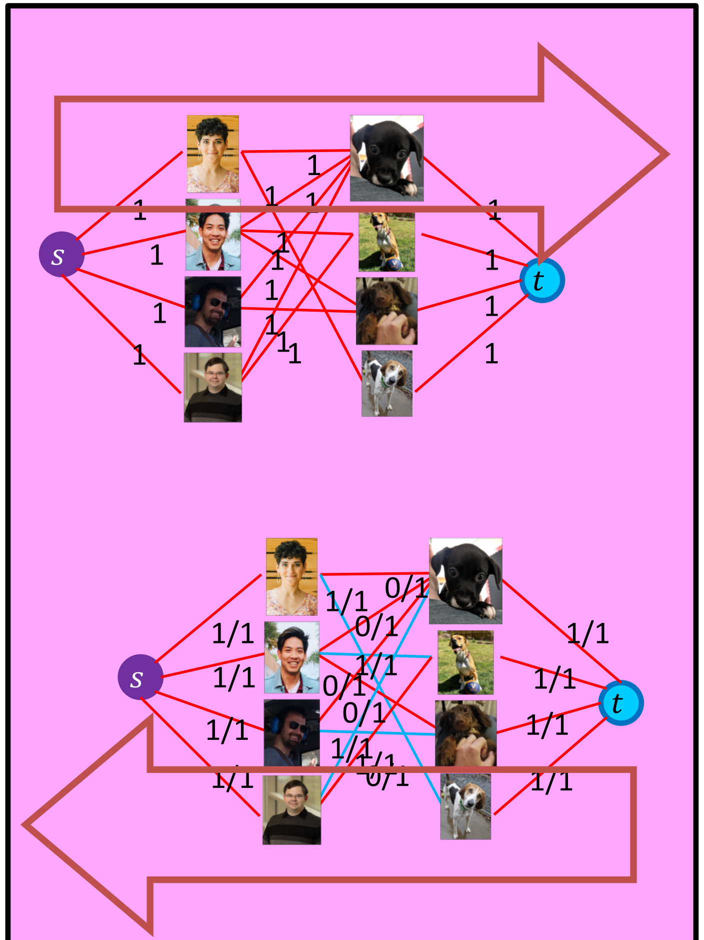
Alcohol, wood, matches

Bipartite Matching Reduction

Problem we don't know how to solve

Bipartite Matching

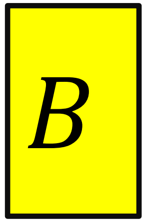


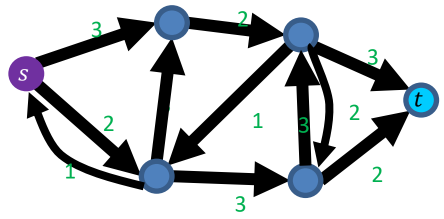


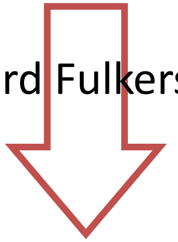
Reduction

Problem we do know how to solve

Max Flow

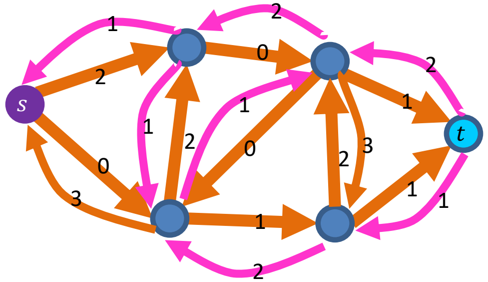




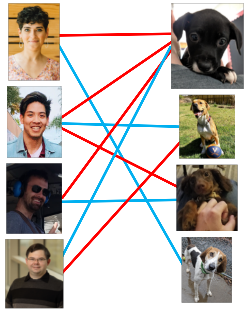


Ford Fulkerson

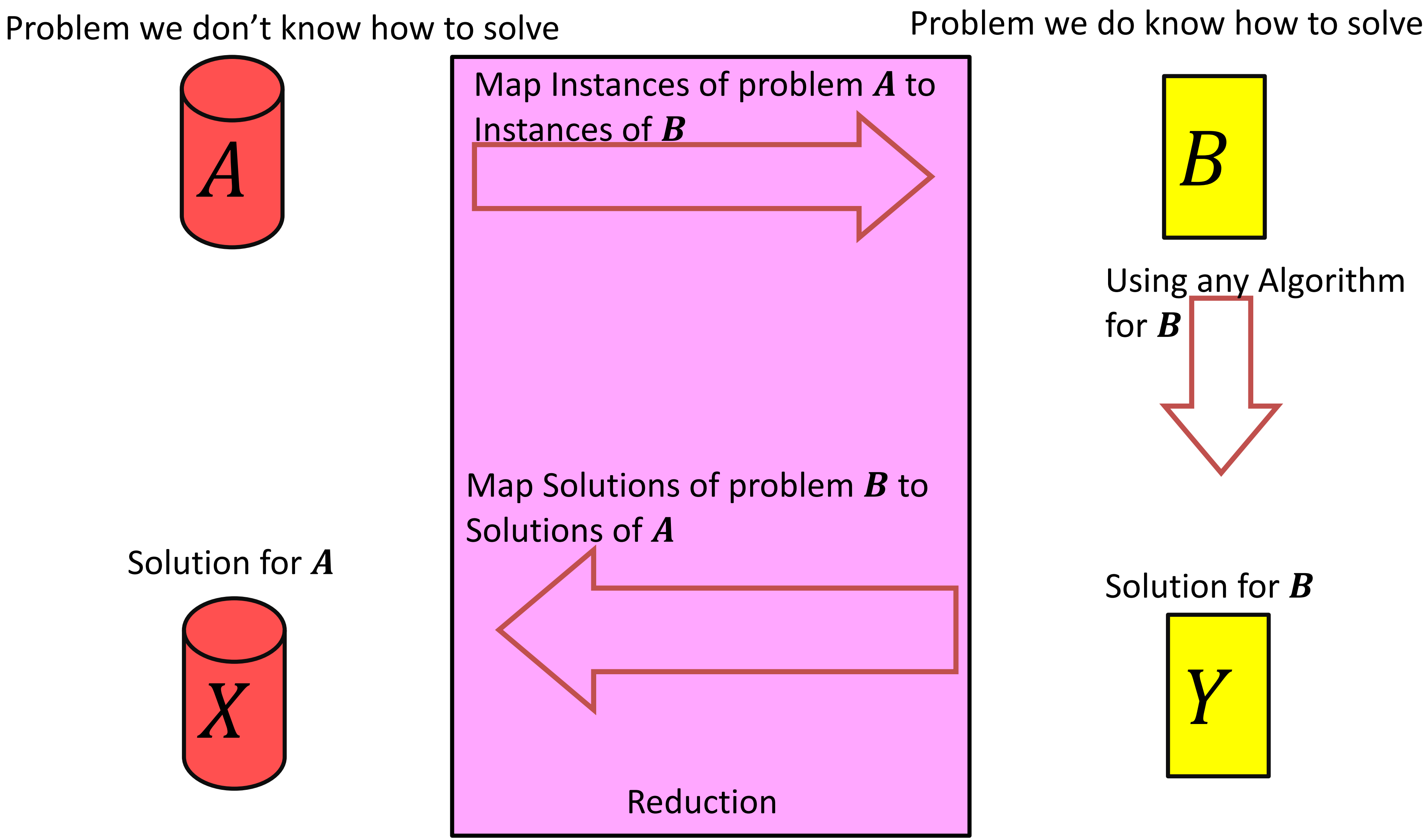
Solution for $\boldsymbol{B}$



Solution for $\boldsymbol{A}$



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

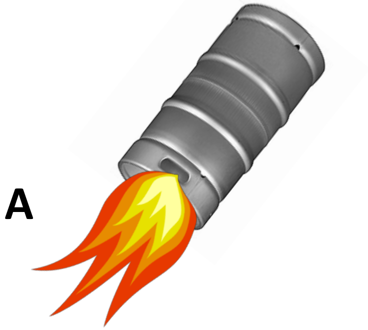
Algorithm for **B**

can be used to make

X

Algorithm for **A**

Keg cannon battering ram



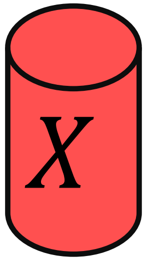
A is not a harder problem than B

$$A \leq B$$

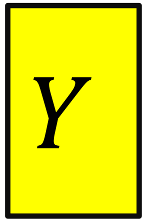
The name "reduces" is confusing: it is in the *opposite* direction of the making

Proof of Lower Bound by Reduction

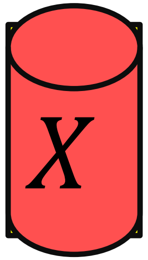
<u>To Show: Y is slow</u>



1. We know X is slow
(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