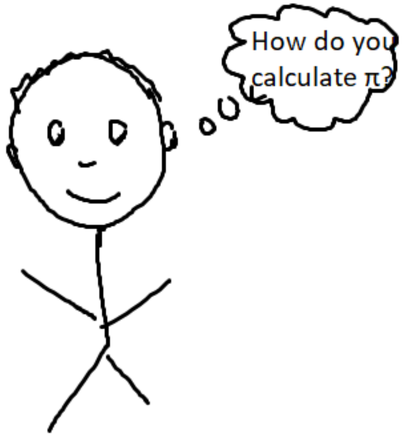
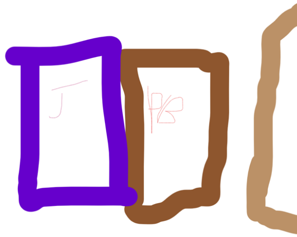
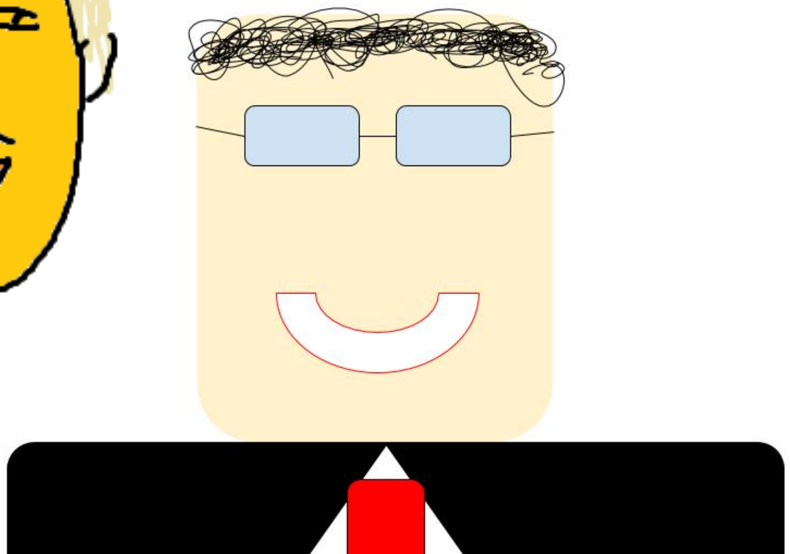
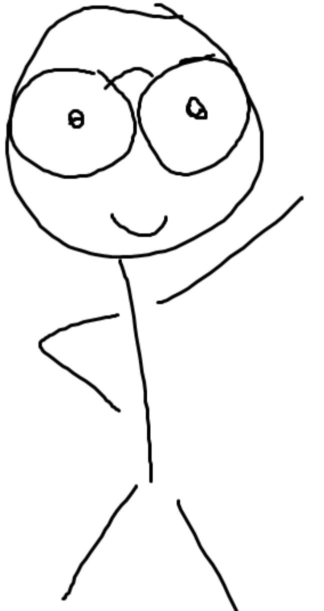
CS4102 Algorithms

Fall 2019

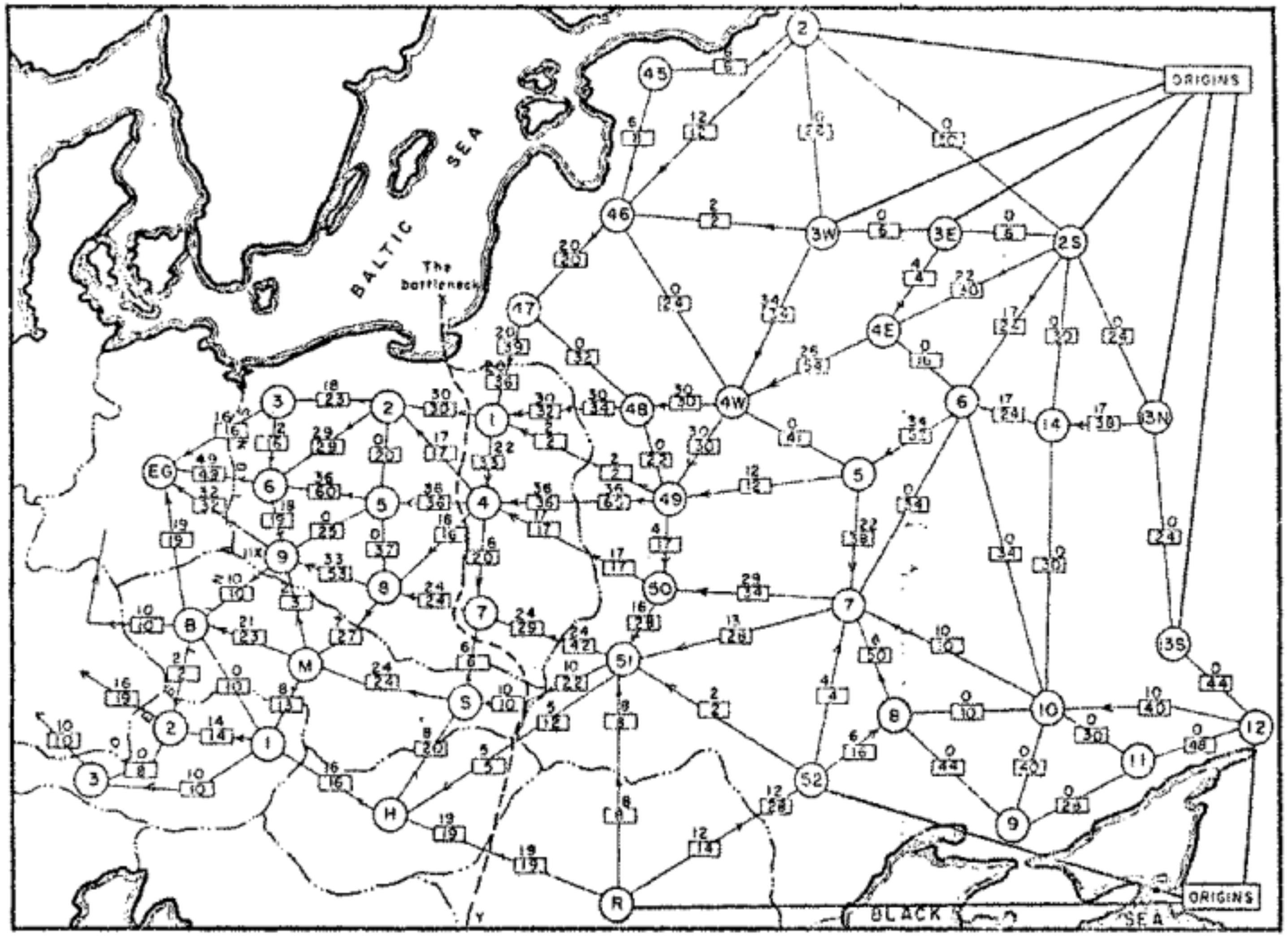






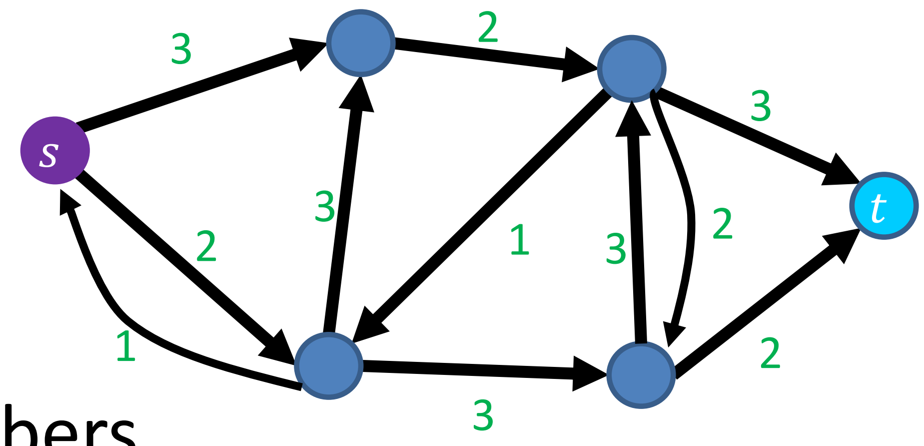


Max Flow / Min Cut



Railway map of Western USSR, 1955

Flow Network



Graph $G = (V, E)$

Source node $s \in V$

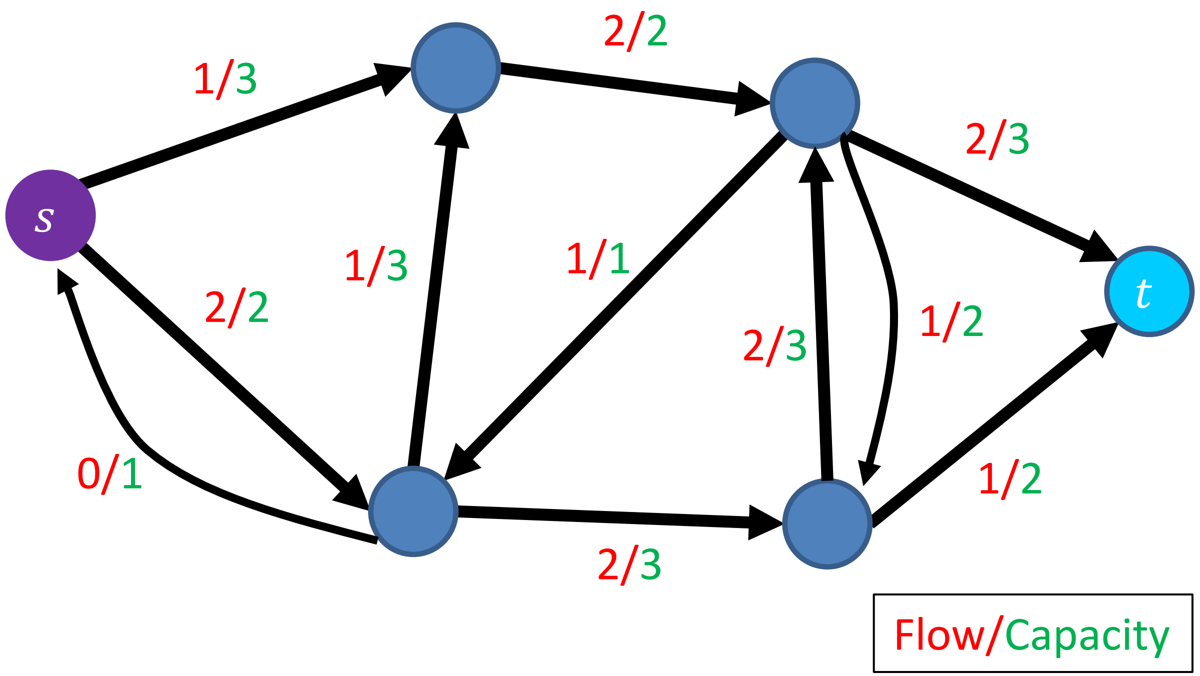
Sink node $t \in V$

Edge Capacities $c(e) \in$ Positive Real numbers

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

Flow

- Assignment of values to edges
  - $f(e) = n$
  - Amount of water going through that pipe
- Capacity constraint
  - $f(e) \leq c(e)$
  - Flow cannot exceed capacity
- Flow constraint
  - $\forall v \in V - \{s, t\}, inflow(v) = outflow(v)$
  - $inflow(v) = \sum_{x \in V} f(x, v)$
  - $outflow(v) = \sum_{x \in V} f(v, x)$
  - Water going in must match water coming out
- Flow of G: $|f| = outflow(s) - inflow(s)$
  - Net outflow of s



3 in example above

Max Flow

- Of all valid flows through the graph, find the one which maximizes:
  - $|f| = outflow(s) - inflow(s)$

Residual Graph G_f

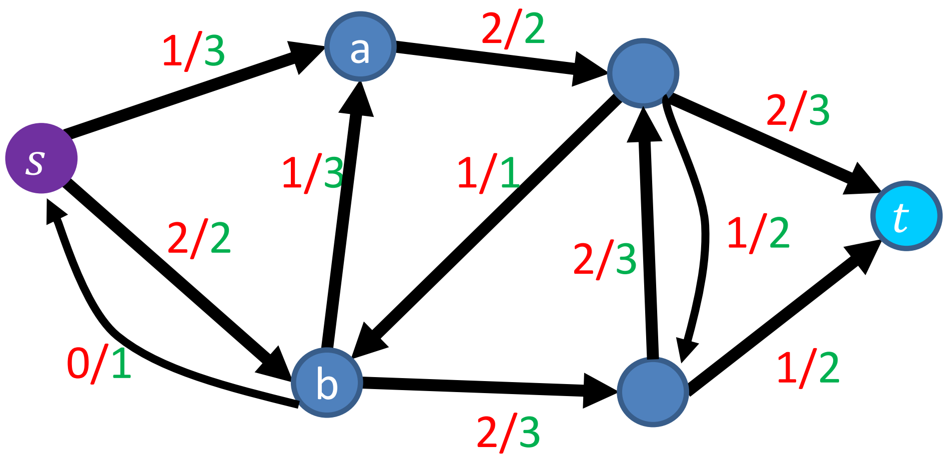
- Keep track of net available flow along each edge
- Forward edges: weight is equal to available flow along that edge in the flow graph
  - $w(e) = c(e) - f(e)$

Flow I *could* add

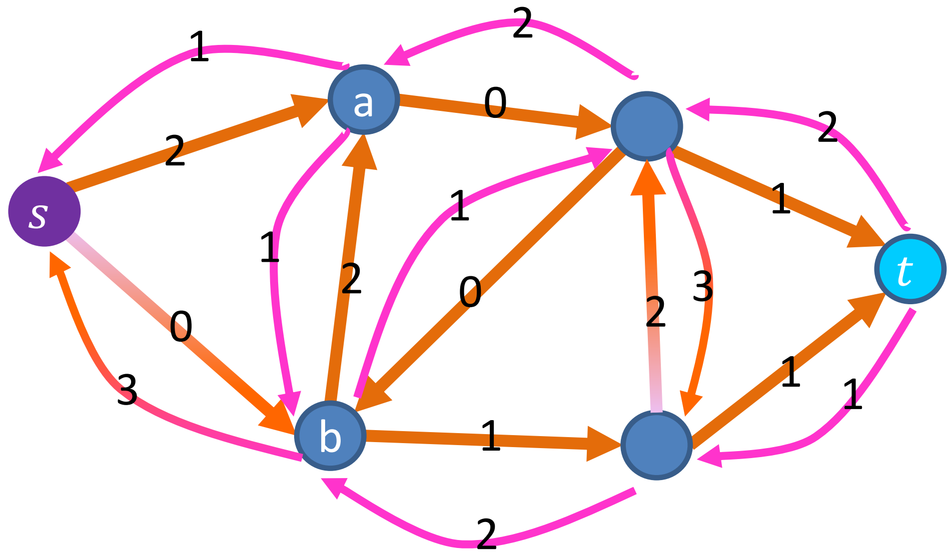
- Back edges: weight is equal to flow along that edge in the flow graph
  - $w(e) = f(e)$

Flow I *could* remove

Flow Graph G



Residual Graph G_f



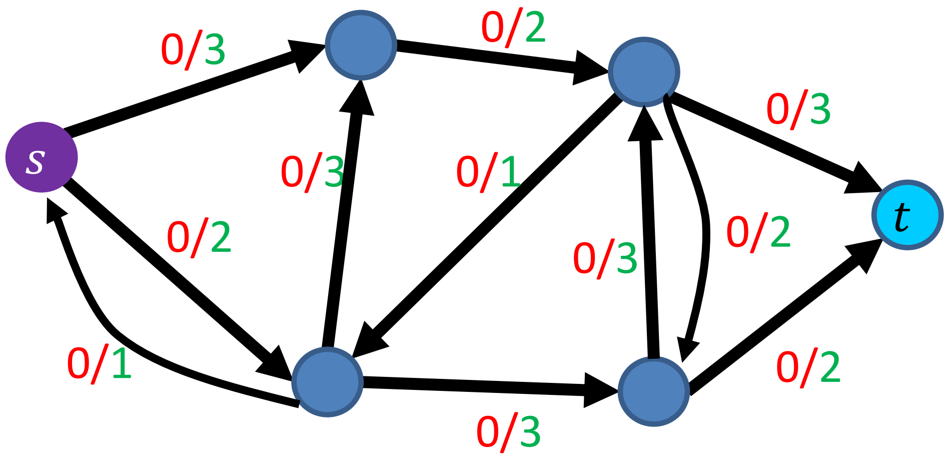
Ford-Fulkerson

- Augmenting Path: a path of positive-weight edges from s to t in the residual graph
- Algorithm: Repeatedly add the flow of any augmenting path

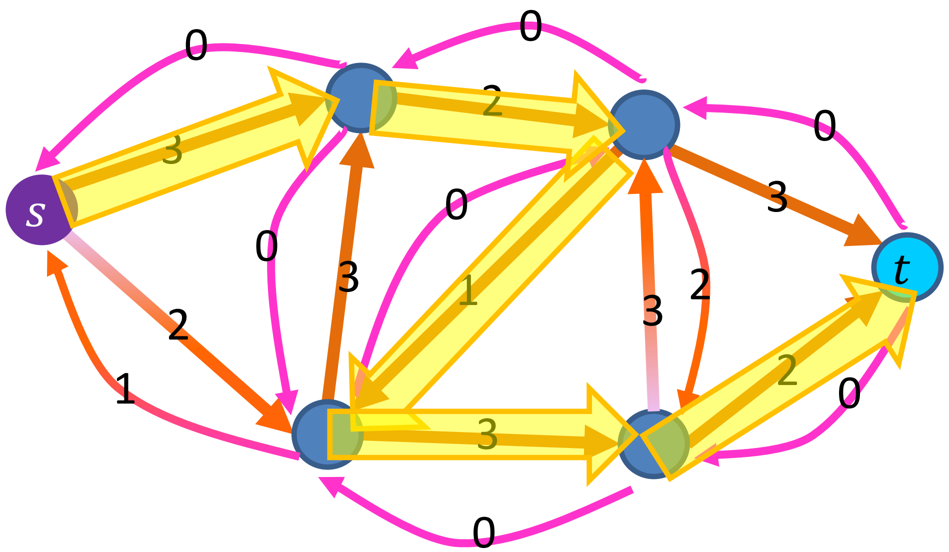
$\forall(u, v) \in E$ Initialize $f(u, v) = 0$
While there is an augmenting path p in G_f
  let $f = \min_{u,v \in p} c_f(u, v)$
  add f to the flow of each edge in p

Ford Fulkerson: example

Flow Graph G



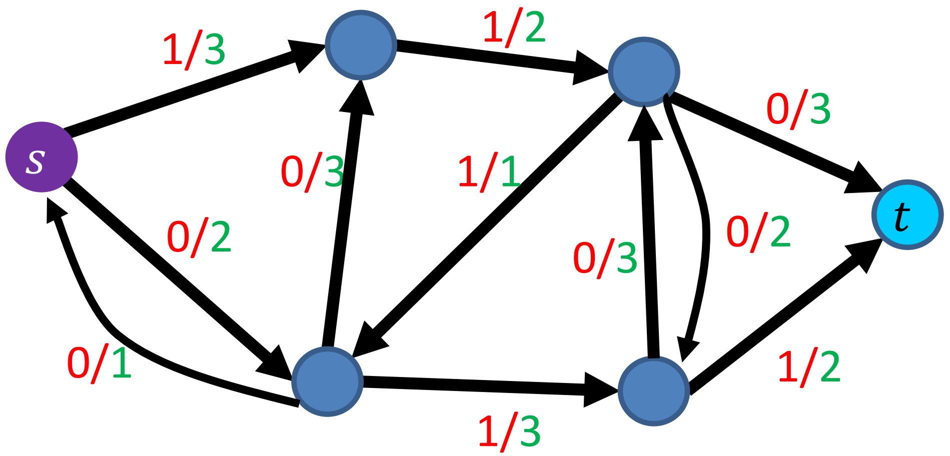
Residual Graph G_f



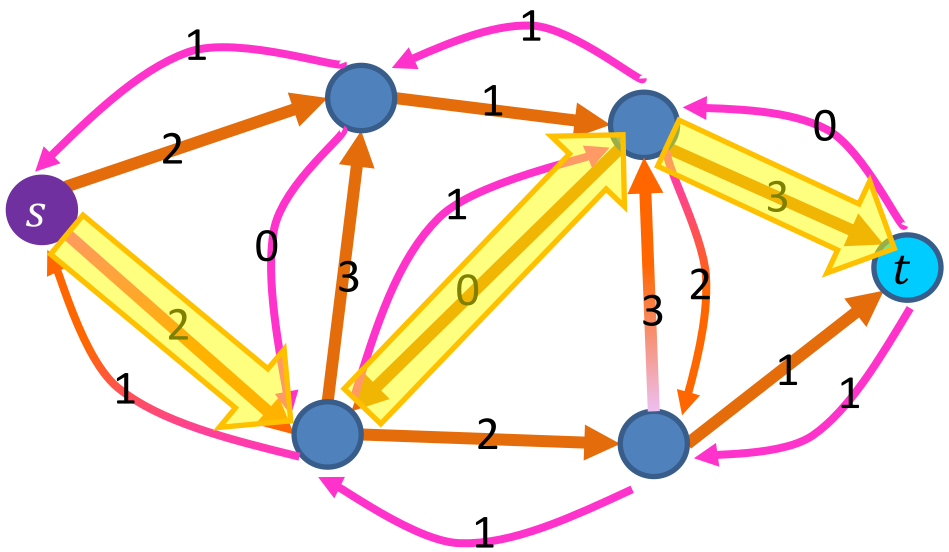
Add flow of 1 to this path

Ford Fulkerson: example

Flow Graph G



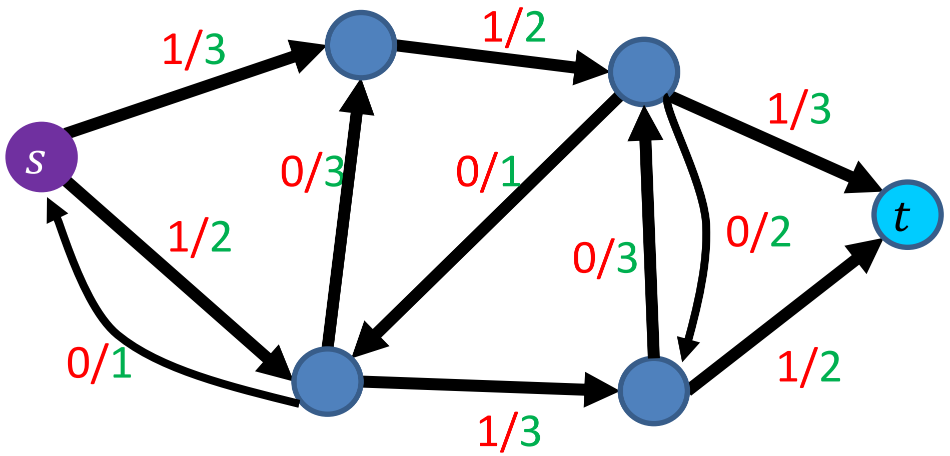
Residual Graph G_f



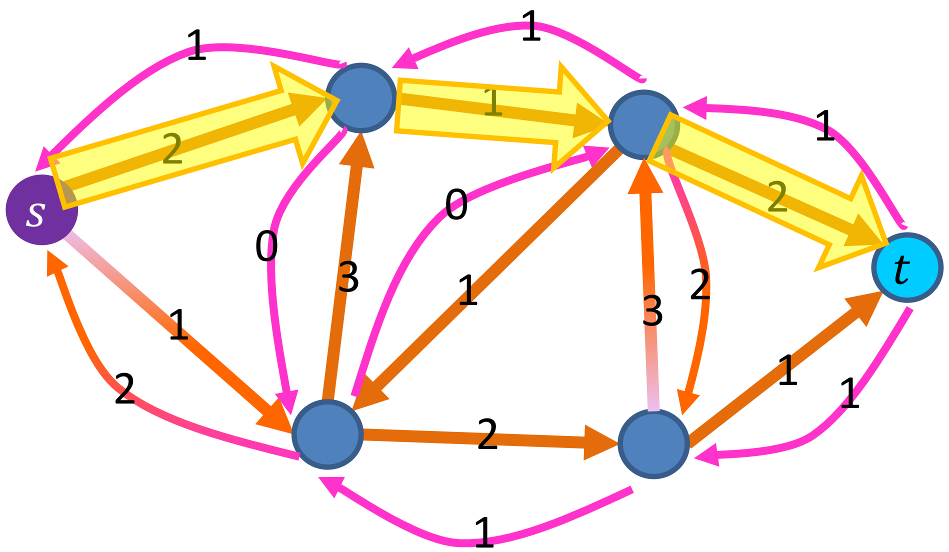
Add flow of 1 to this path

Ford Fulkerson: example

Flow Graph G



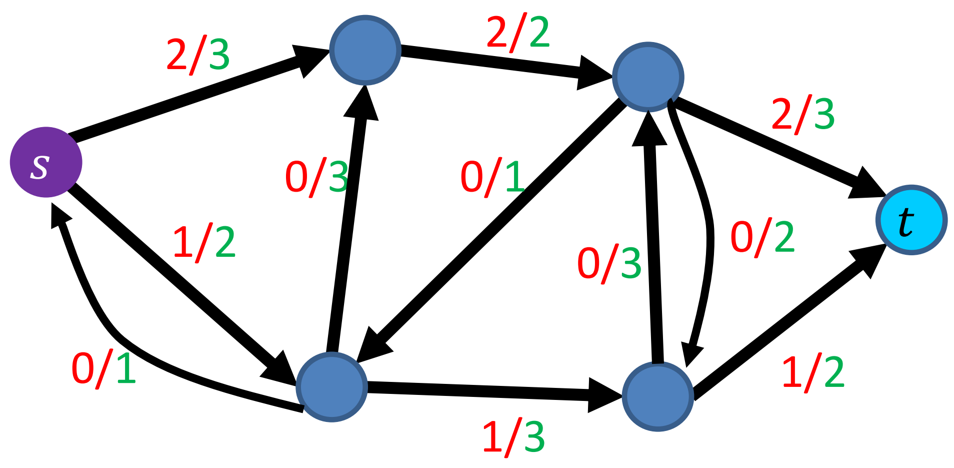
Residual Graph G_f



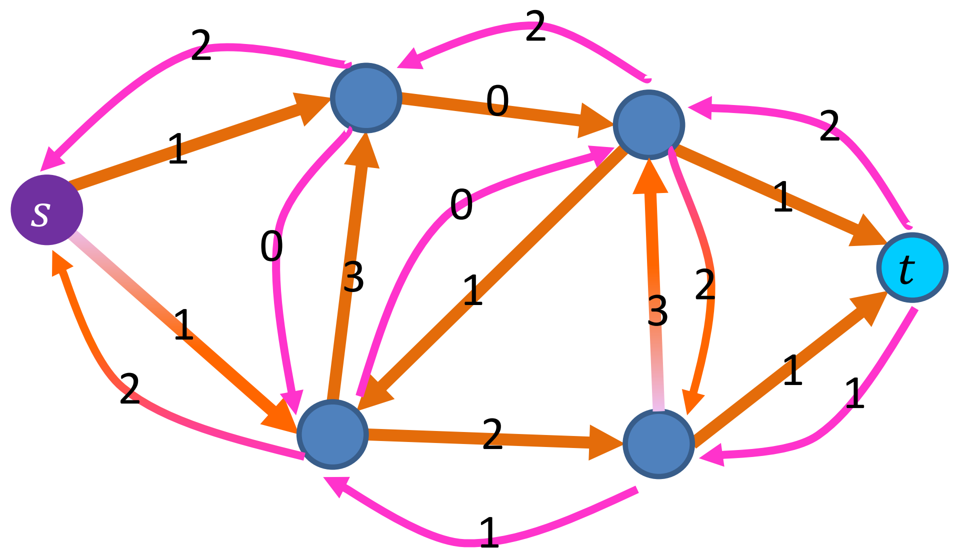
Add flow of 1 to this path

Ford Fulkerson: example

Flow Graph G

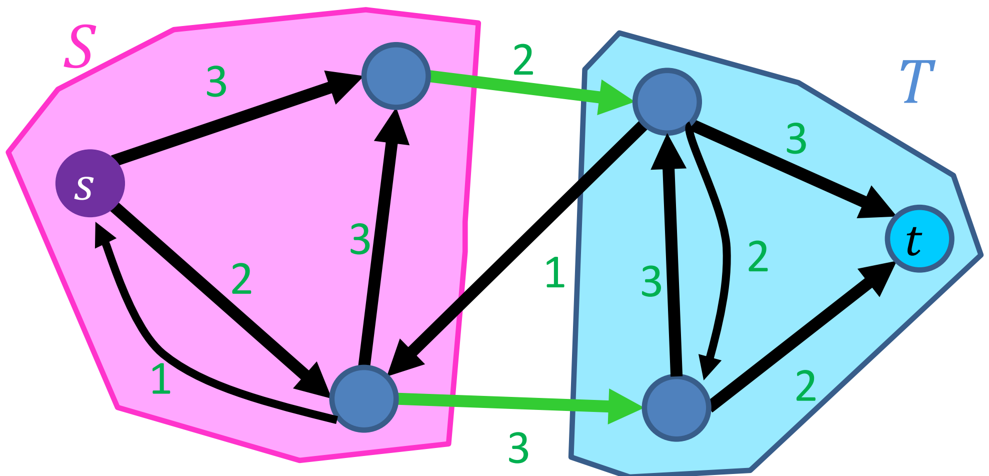


Residual Graph G_f



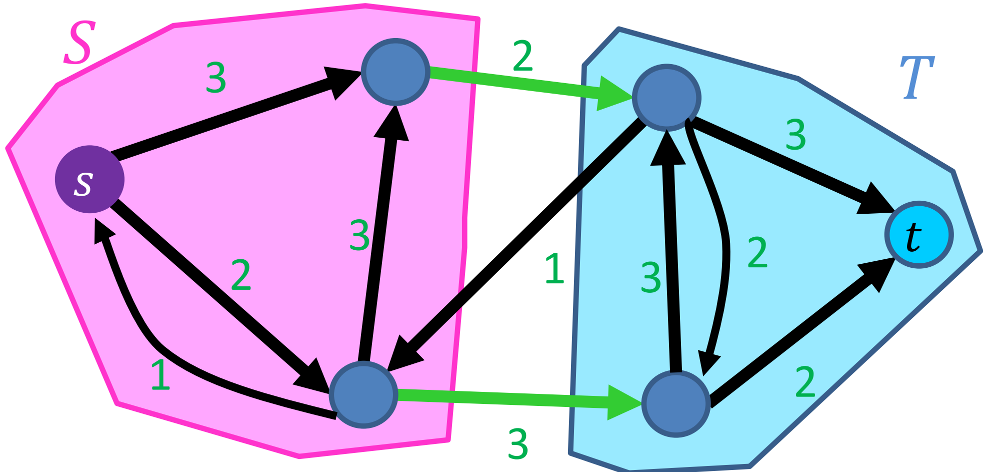
Showing Correctness of Ford-Fulkerson

- 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

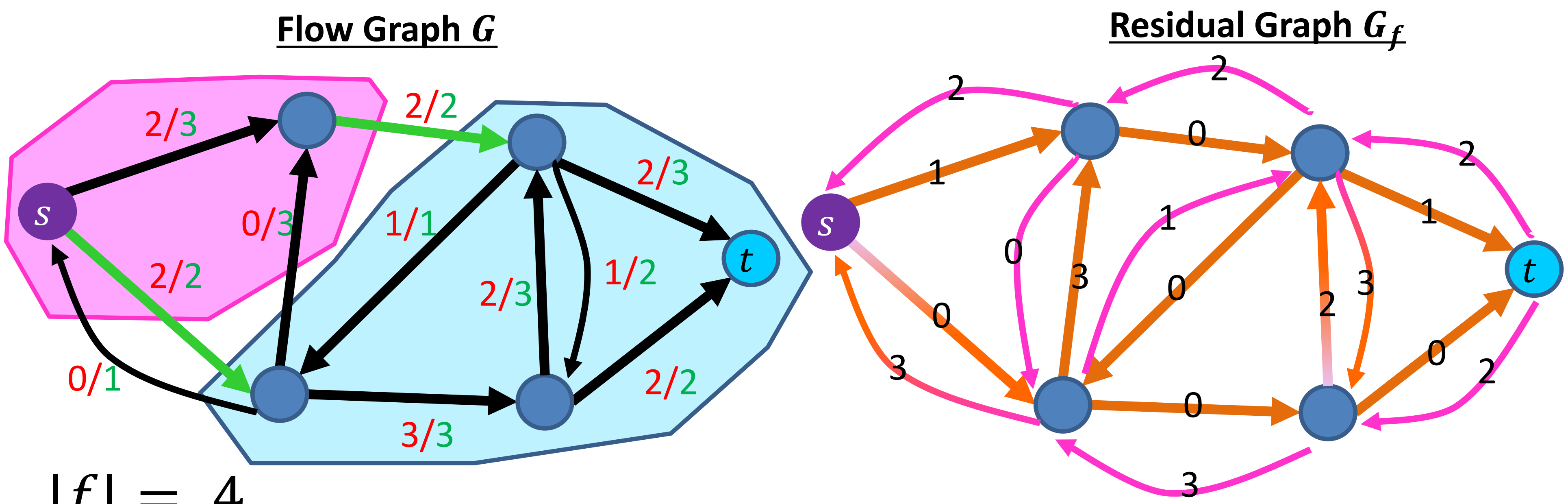
- Max flow upper bounded by any cut separating s and t
- Why? "Conservation of flow"
  - All flow exiting s must eventually get to t
  - To get from s to t, all "tanks" must cross the cut
- Conclusion: If we find the minimum-cost cut, we've found the maximum flow
  - $\max_{f} |f| \leq \min_{S,T} ||S,T||$



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



$|f| = 4$

$\left|\left|S, T\right|\right| = 4$

No Augmenting Paths

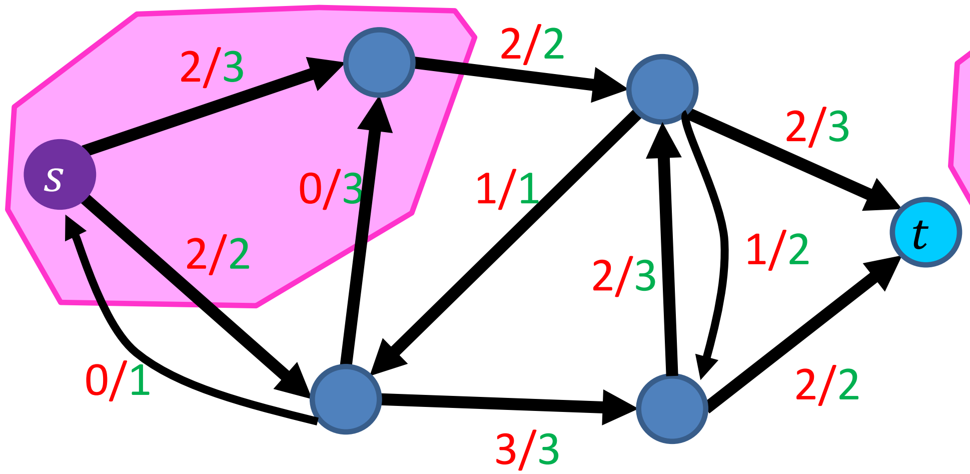
Idea: When there are no more augmenting paths, there exists a cut in the graph with cost matching the flow

15

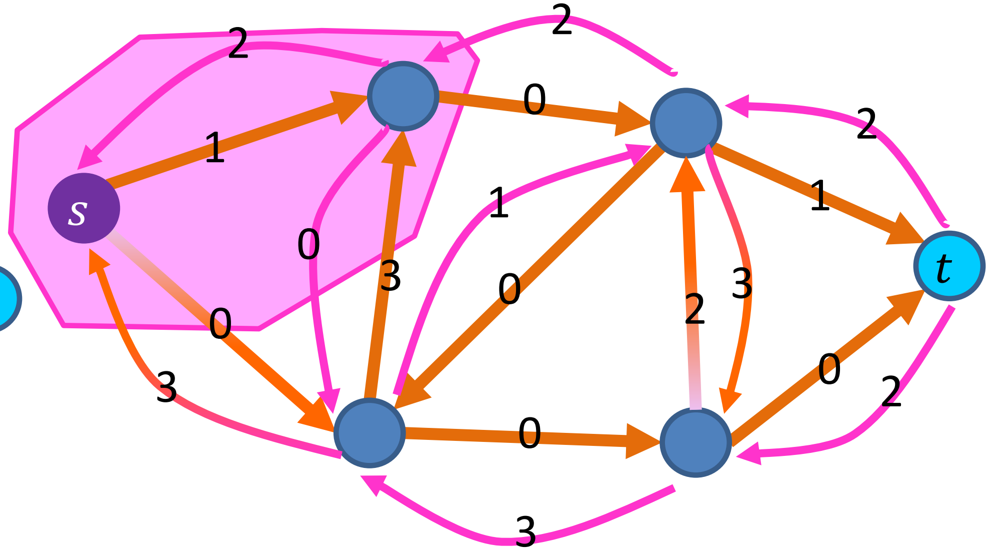
Proof: Maxflow/Mincut Theorem

- 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



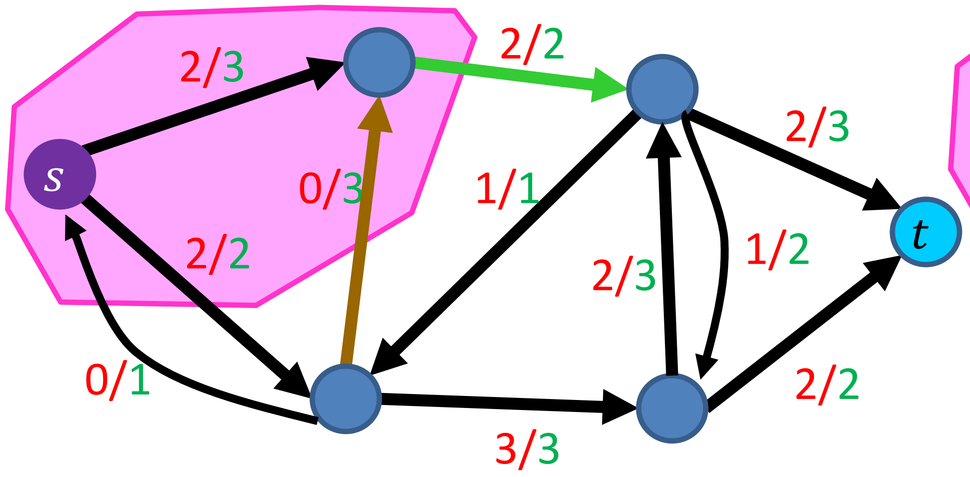
Residual Graph G_f



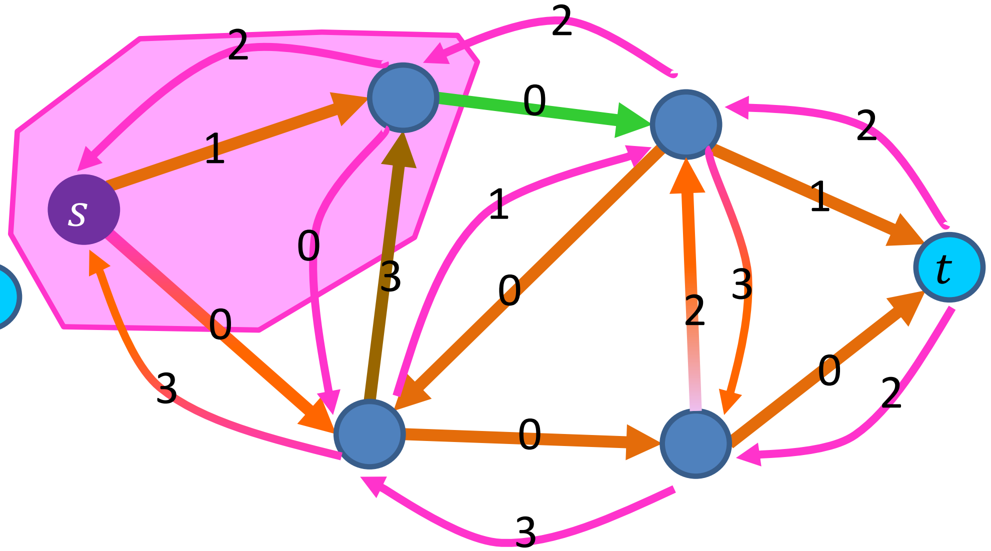
Proof: Maxflow/Mincut Theorem

- 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 $\in S$

Flow Graph G



Residual Graph G_f



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

Today’s Keywords

- Reductions
- Bipartite Matching
- Vertex Cover
- Independent Set

CLRS Readings

- Chapter 34

Homeworks

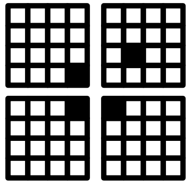
- HW8 due Thursday, 11/21, at 11pm
  - Python or Java
  - Marriage
- HW9 out Thursday, due Thursday 12/5 at 11pm
  - Reductions, Graphs
  - Written (LaTeX)
- HW10C out Thursday, due Thursday 12/5 at 11pm
  - Implement a problem from HW9
  - No late submissions

Final Exam

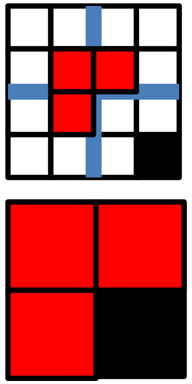
- Monday, December 9, 7pm in Maury 209 (our section)
  - Practice exam coming next week
  - Review session likely the weekend before
  - Conflict form coming by email
    (if you have another exam scheduled for 7pm)

Divide and Conquer*

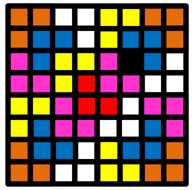
- **Divide**:



  – 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

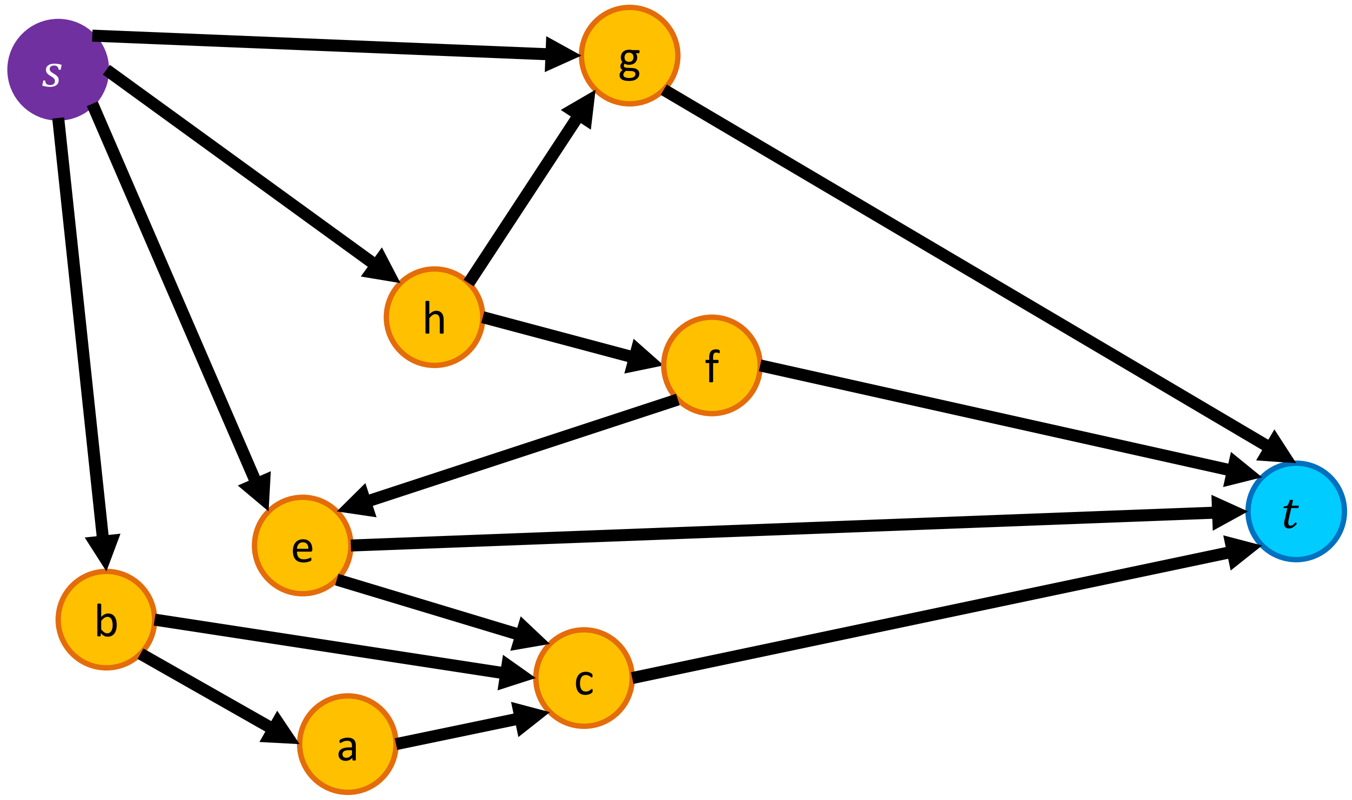
- Require Optimal Substructure
  - Solution to larger problem contains the solution to a smaller one
  - Only one subproblem to consider!
- Idea:
  1. Identify a greedy choice property
     - How to make a choice guaranteed to be included in some optimal solution
  2. Repeatedly apply the choice property 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

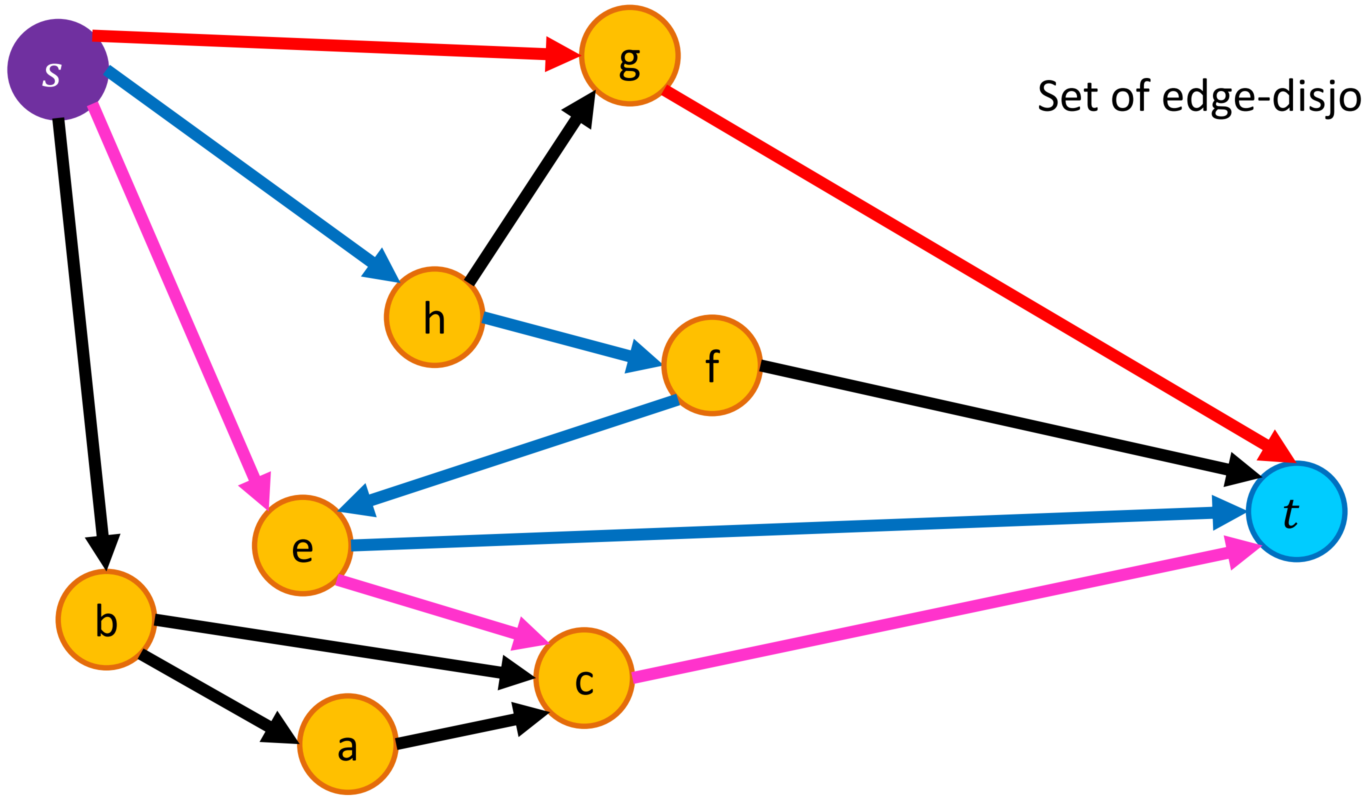
Edge-Disjoint Paths

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

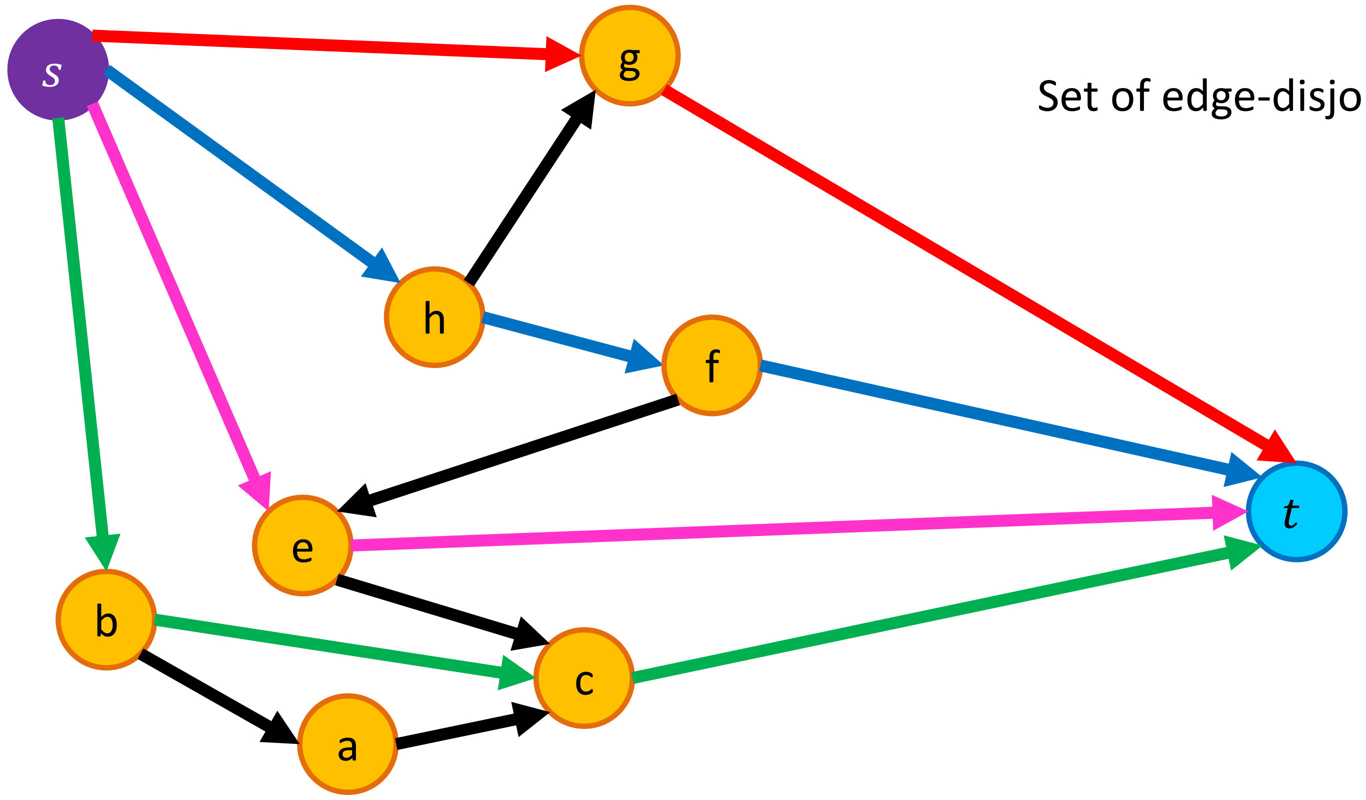
Given a graph $G = (V, E)$, a start node s and a destination node t, give the maximum number of paths from s to t which share no edges



Set of edge-disjoint paths of size 3

Edge-Disjoint Paths

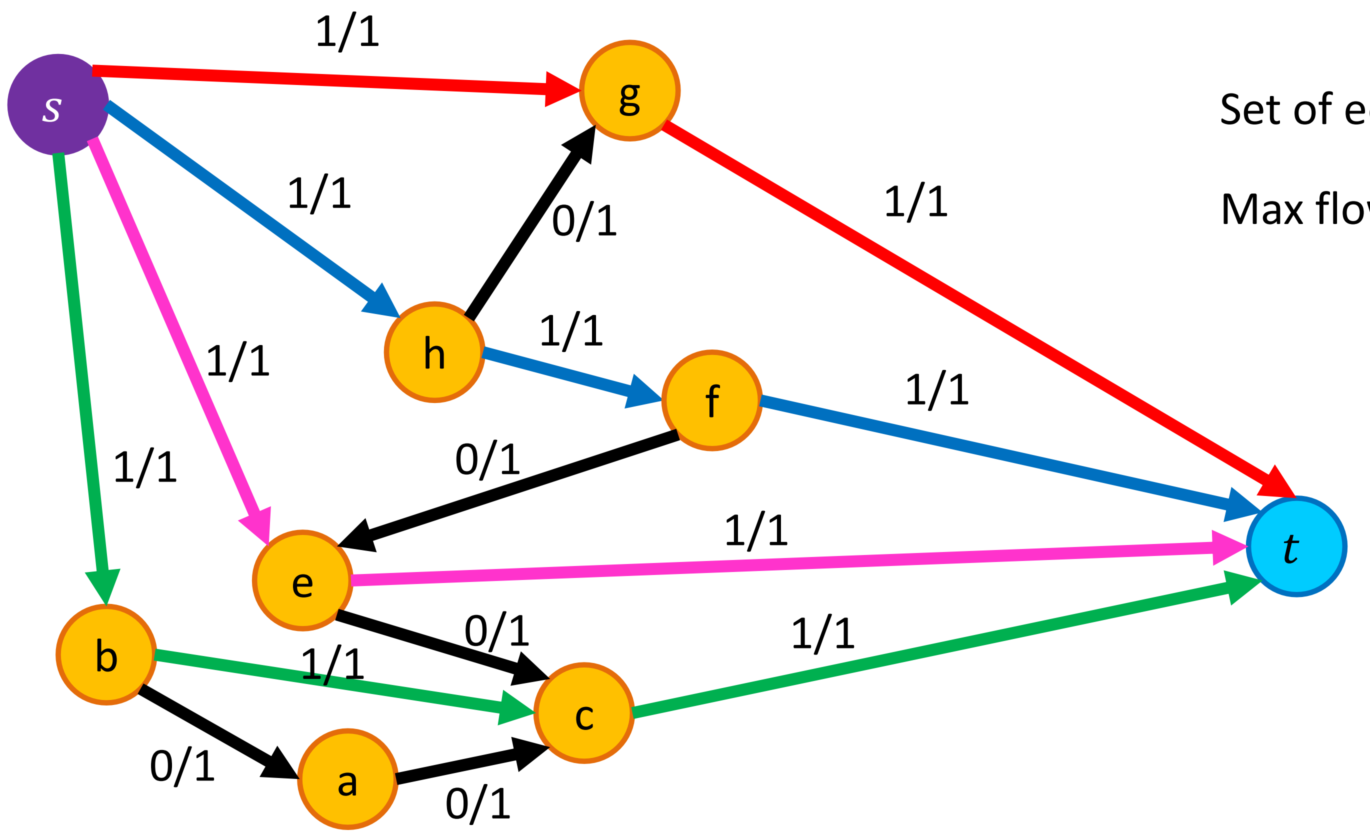
Given a graph $G = (V, E)$, a start node s and a destination node t, give the maximum number of paths from s to t which share no edges



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.

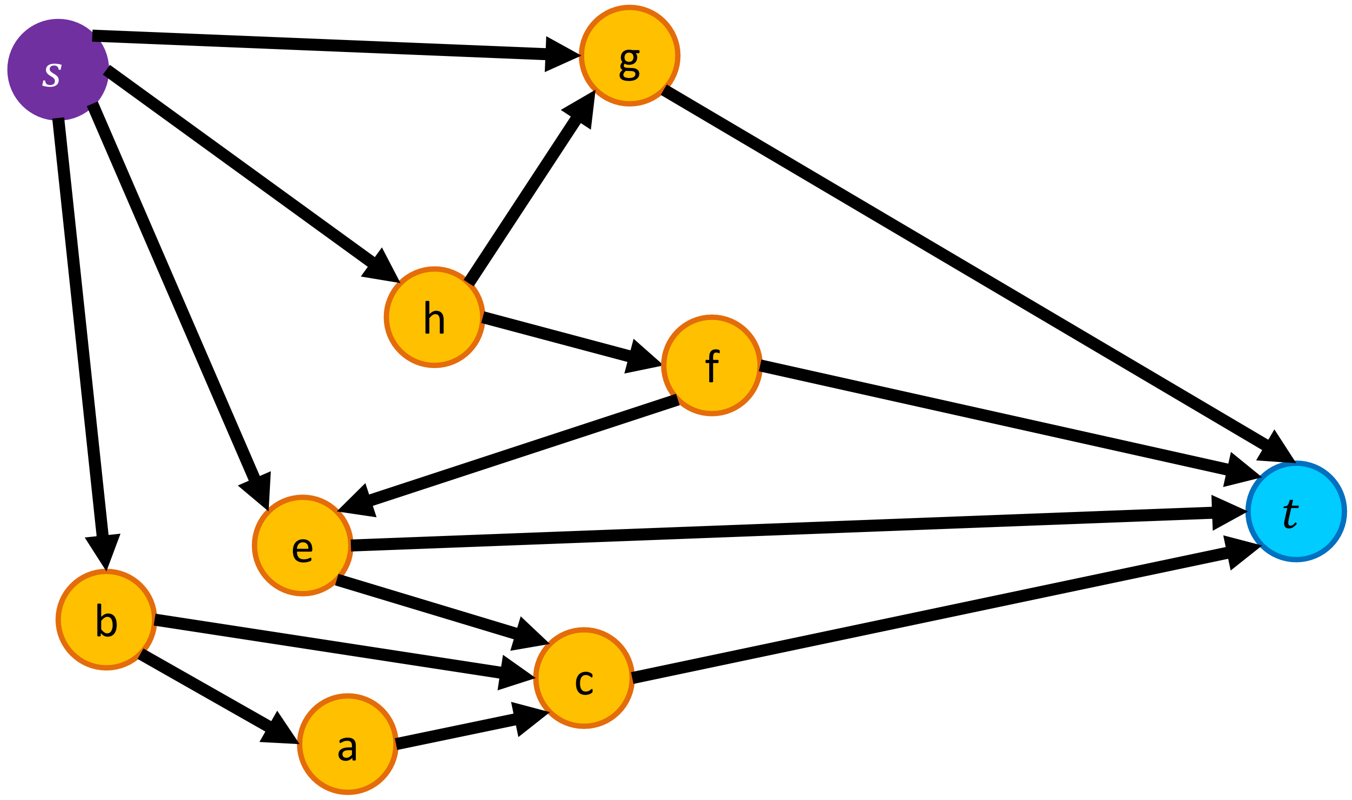


Set of edge-disjoint paths of size 4

Max flow = 4

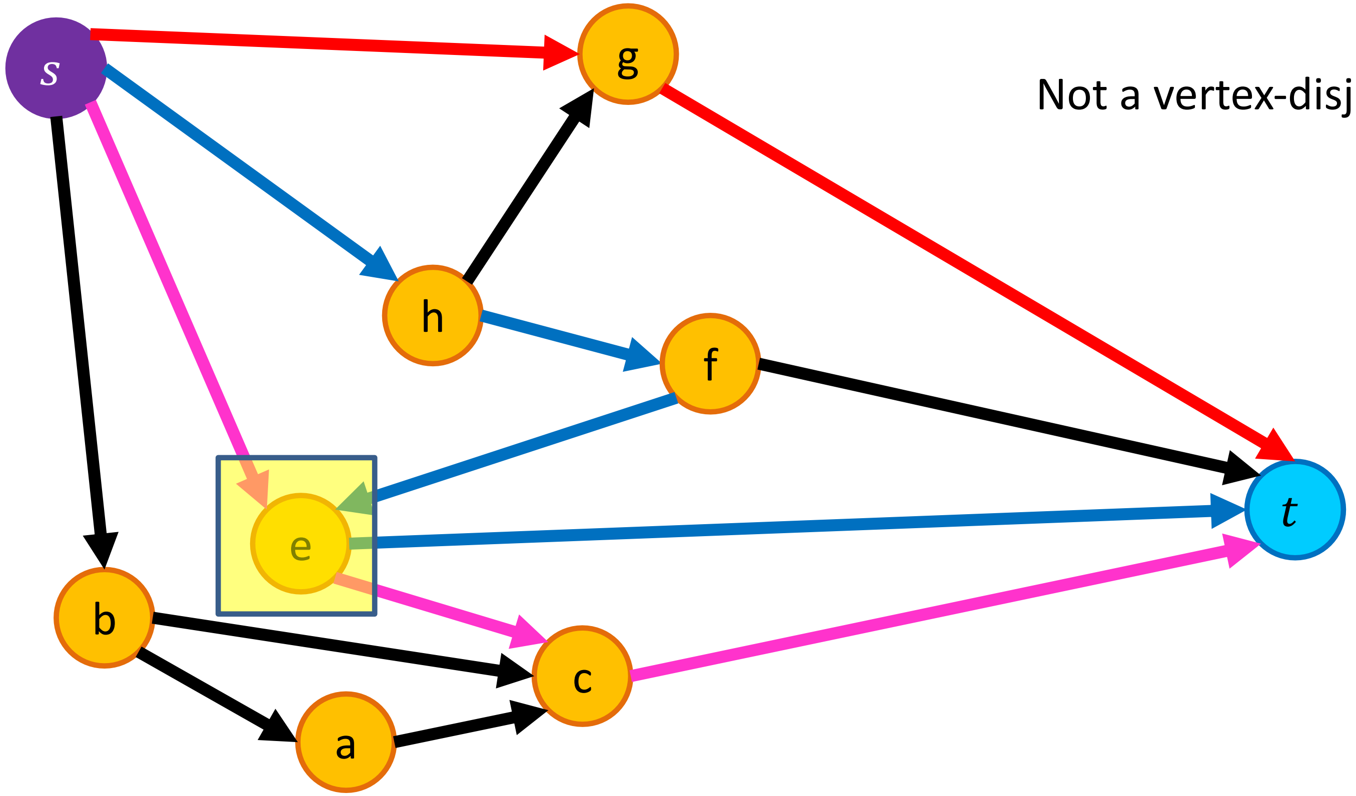
Vertex-Disjoint Paths

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



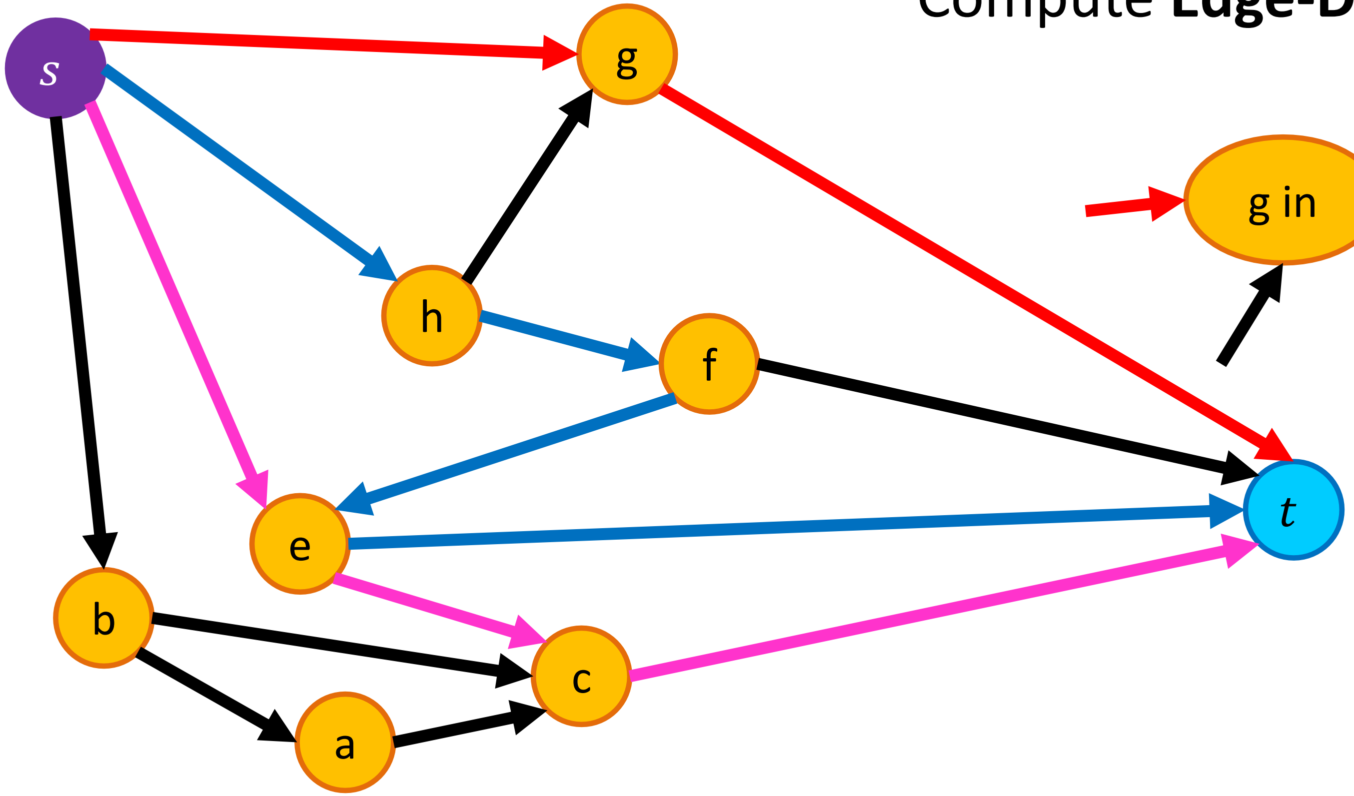
Not a vertex-disjoint path!

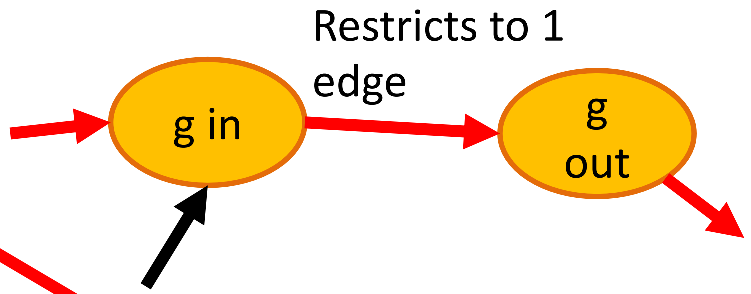
Vertex-Disjoint Paths Algorithm

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

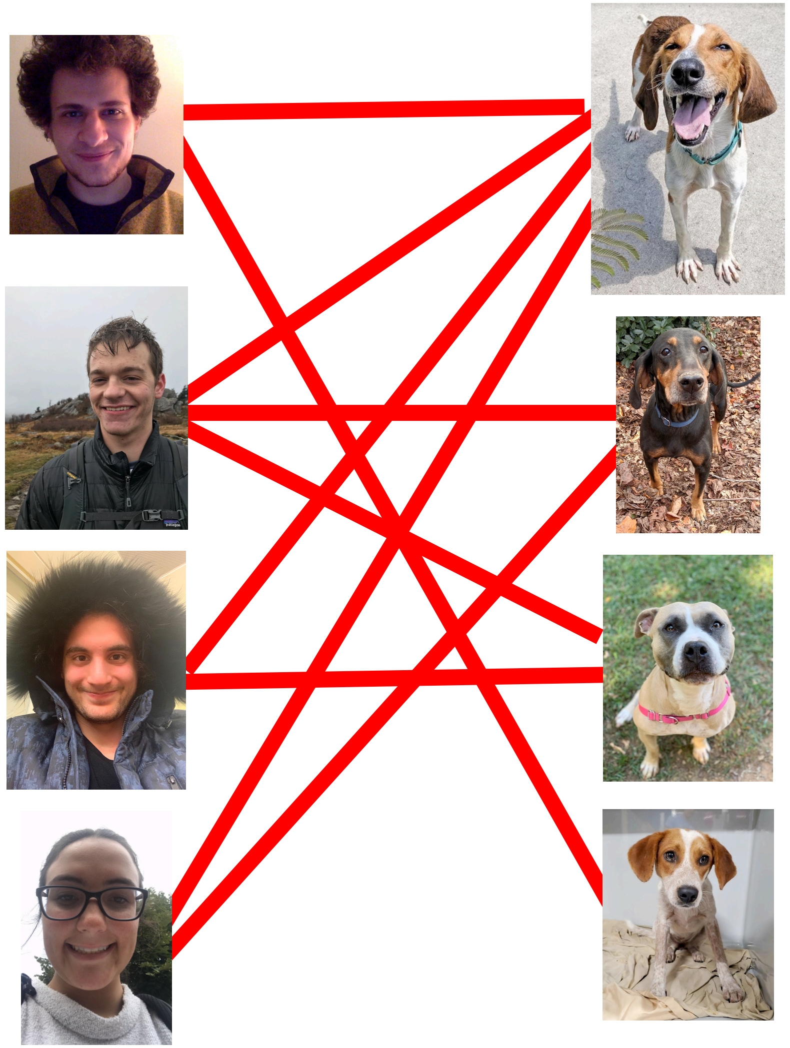
Compute **Edge-Disjoint Paths** on new graph





Maximum Bipartite Matching

Dog Lovers Dogs

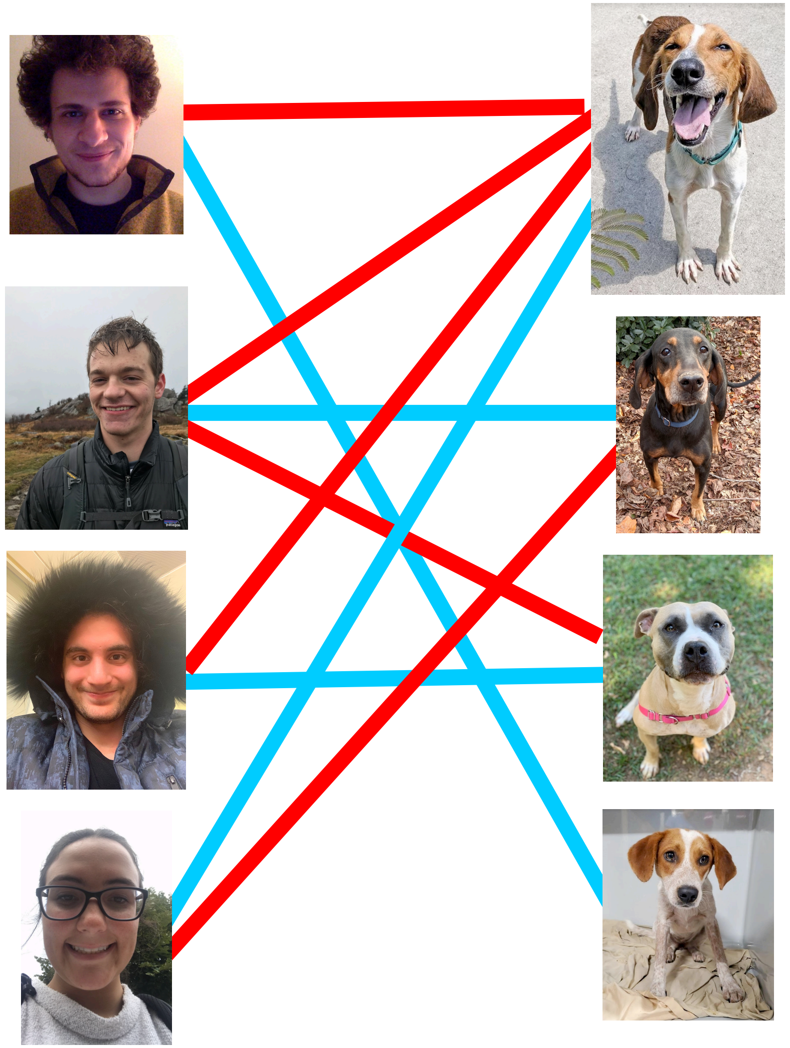


Maximum Bipartite Matching

Maximum Bipartite Matching

Dog Lovers

Dogs



Maximum Bipartite Matching

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

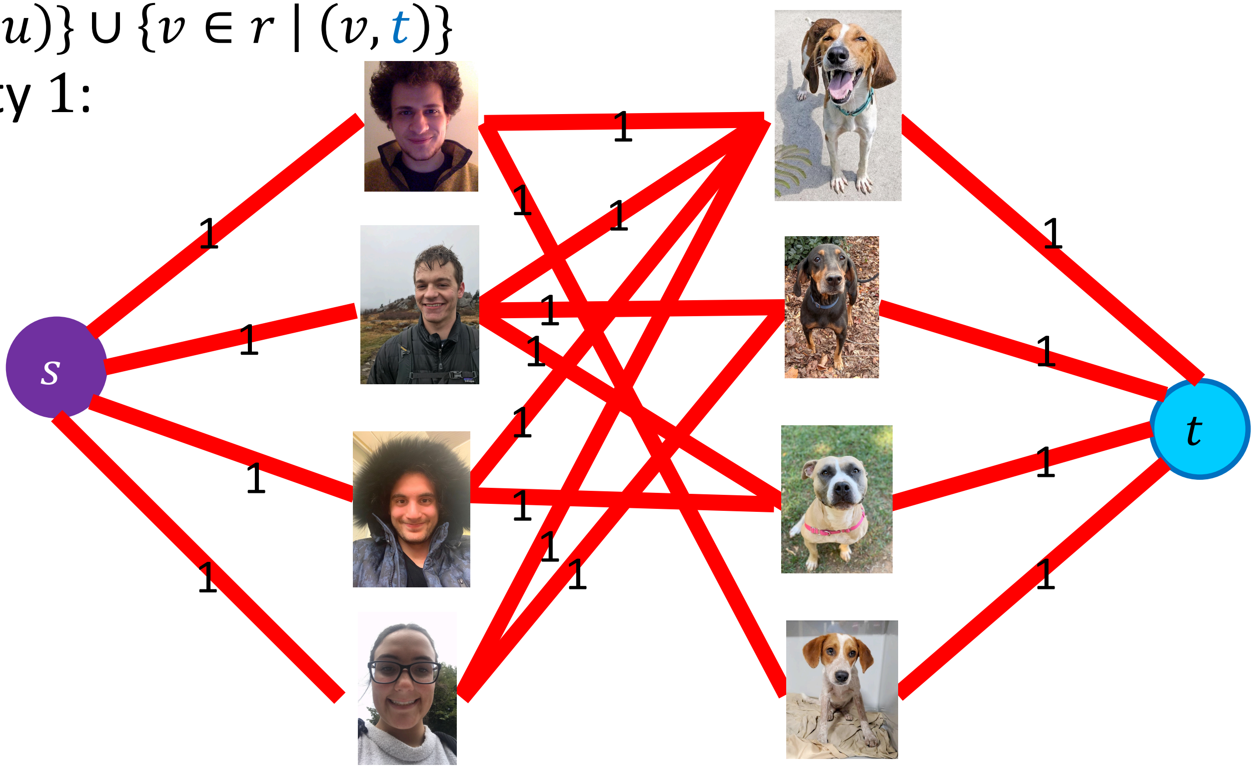
a set of left nodes, right nodes, and edges between left and right

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:

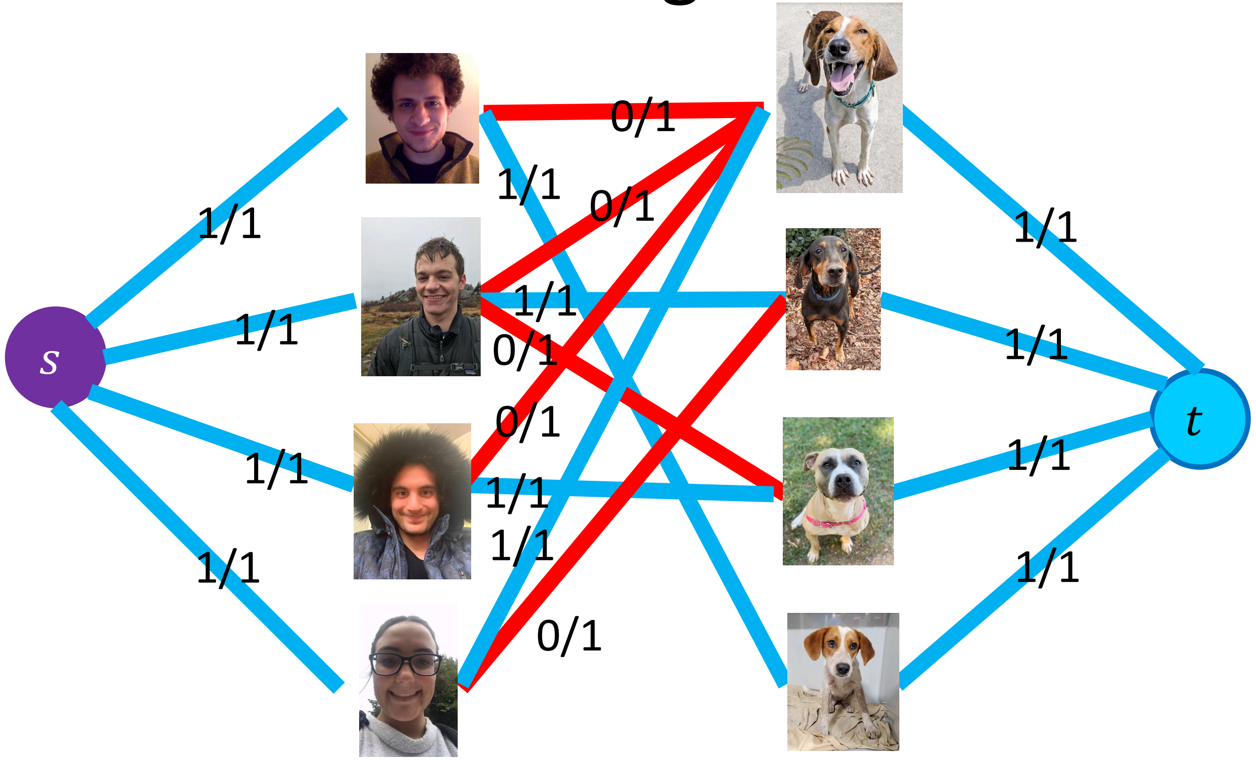
- Adding in a source and sink to the set of nodes:
  - $V' = L \cup R \cup \{s, t\}$
- Adding an edge from source to L and from R to sink:
  - $E' = E \cup \{u \in L \mid (s, u)\} \cup \{v \in r \mid (v, t)\}$
- Make each edge capacity 1:
  - $\forall e \in E', c(e) = 1$



Maximum Bipartite Matching Using Max Flow

$\Theta(E \cdot V)$

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 $\Theta(L + R)$



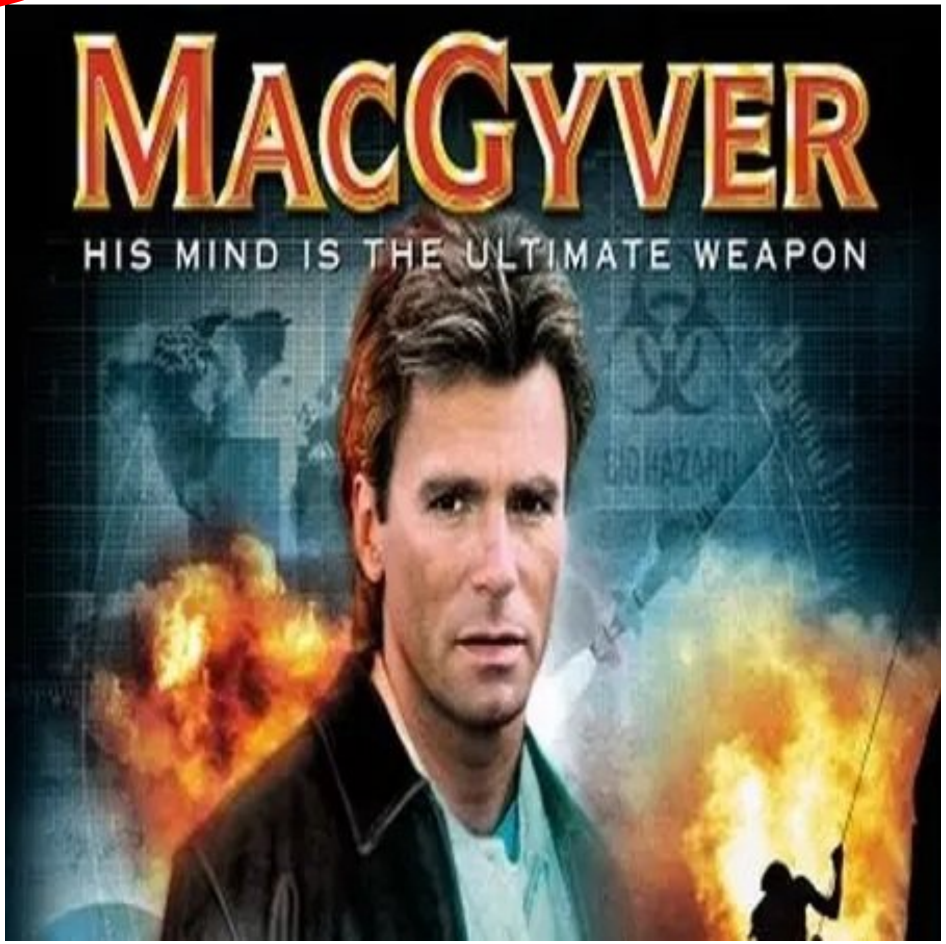
Reductions

- 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

A

Opening a door

Aim duct at door, insert keg

Problem we do know how to solve

B

Lighting a fire

How?

Solution for ***B***

Alcohol, wood, matches

Put fire under the Keg

Reduction

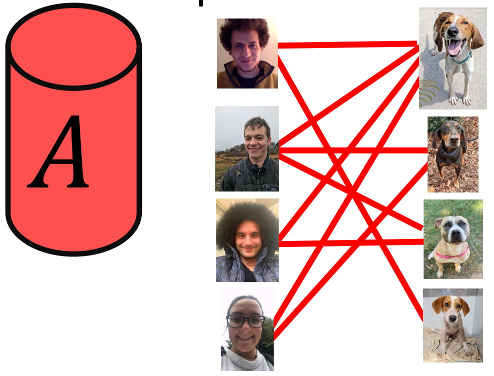
Solution for ***A***

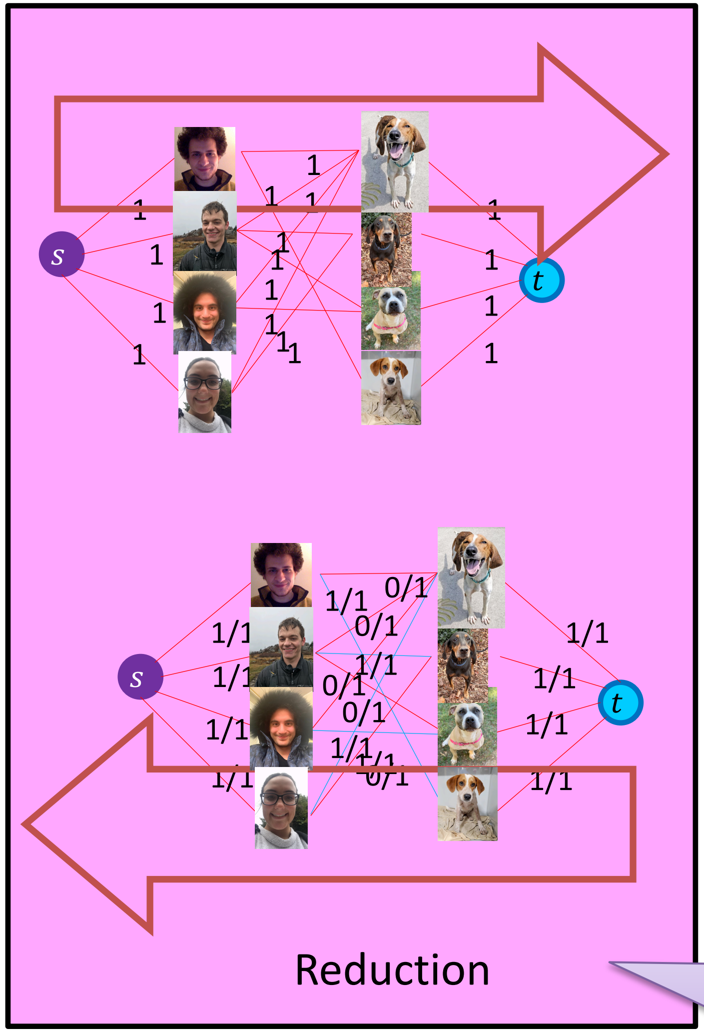
Keg cannon battering ram

Bipartite Matching Reduction

Problem we don't know how to solve

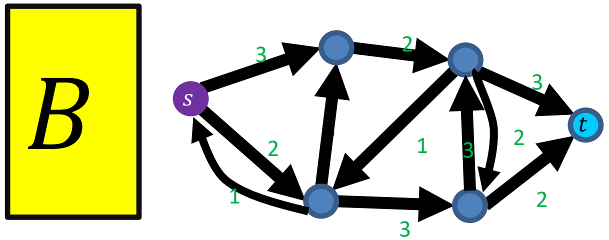
Bipartite Matching

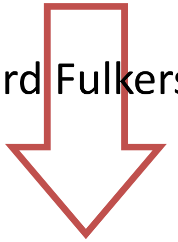




Problem we do know how to solve

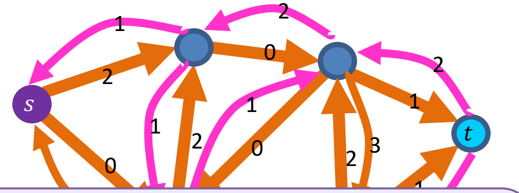
Max Flow



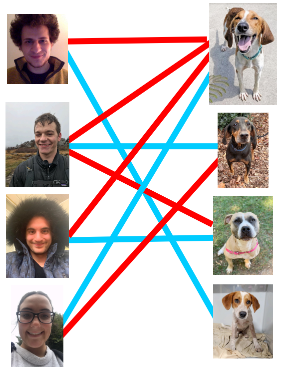


Ford Fulkerson

Solution for $\boldsymbol{B}$

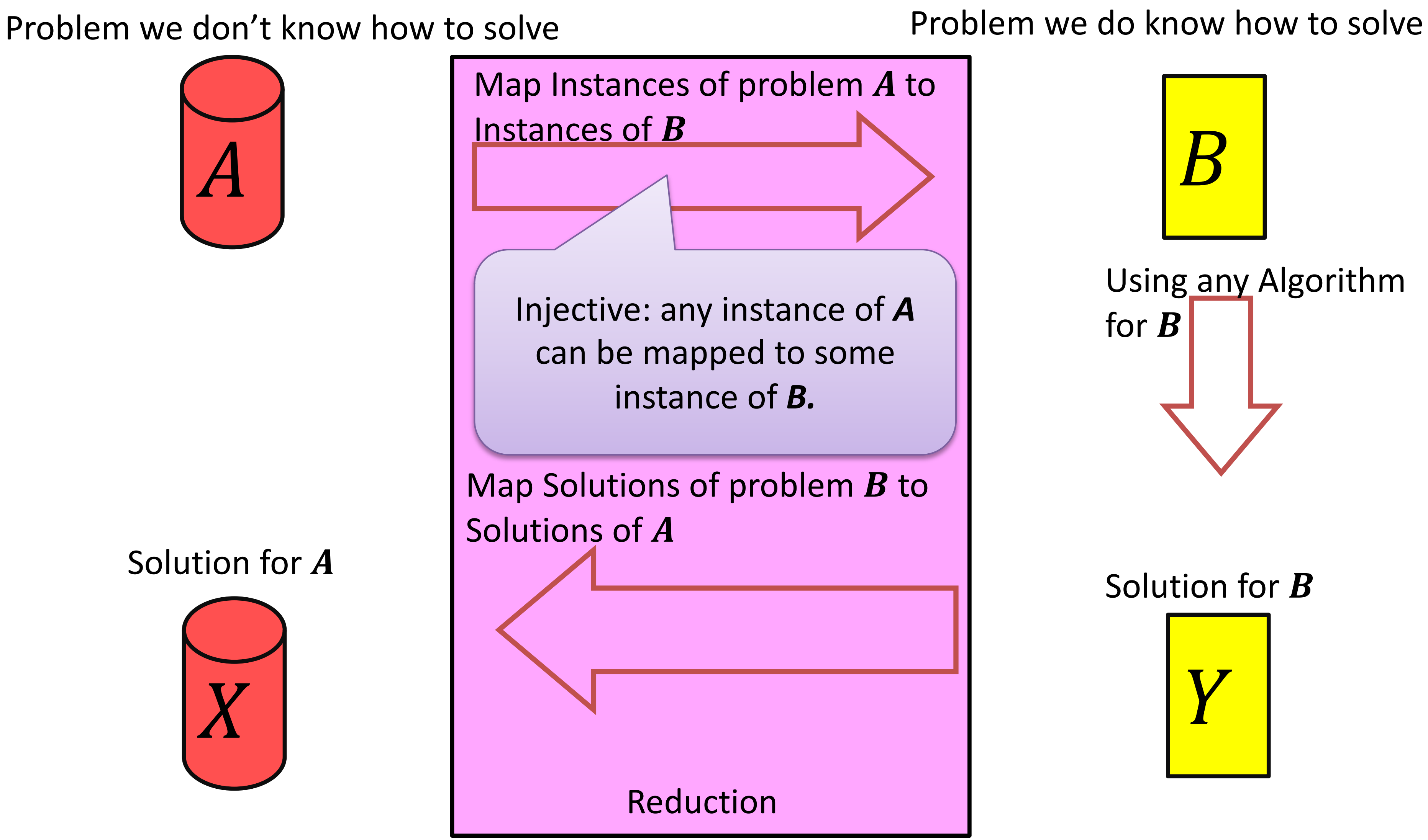


Solution for $\boldsymbol{A}$



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

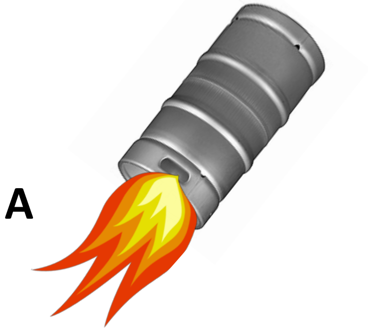
Algorithm for **B**

can be used to make

X

Algorithm for **A**

Keg cannon battering ram



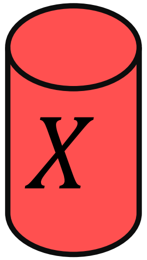
A is not a harder problem than B

$$A \leq B$$

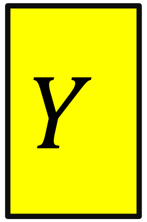
The name “reduces” is confusing: it is in the *opposite* direction of the making

Proof of Lower Bound by Reduction

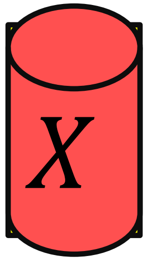
<u>To Show: Y is slow</u>



1. We know X is slow (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