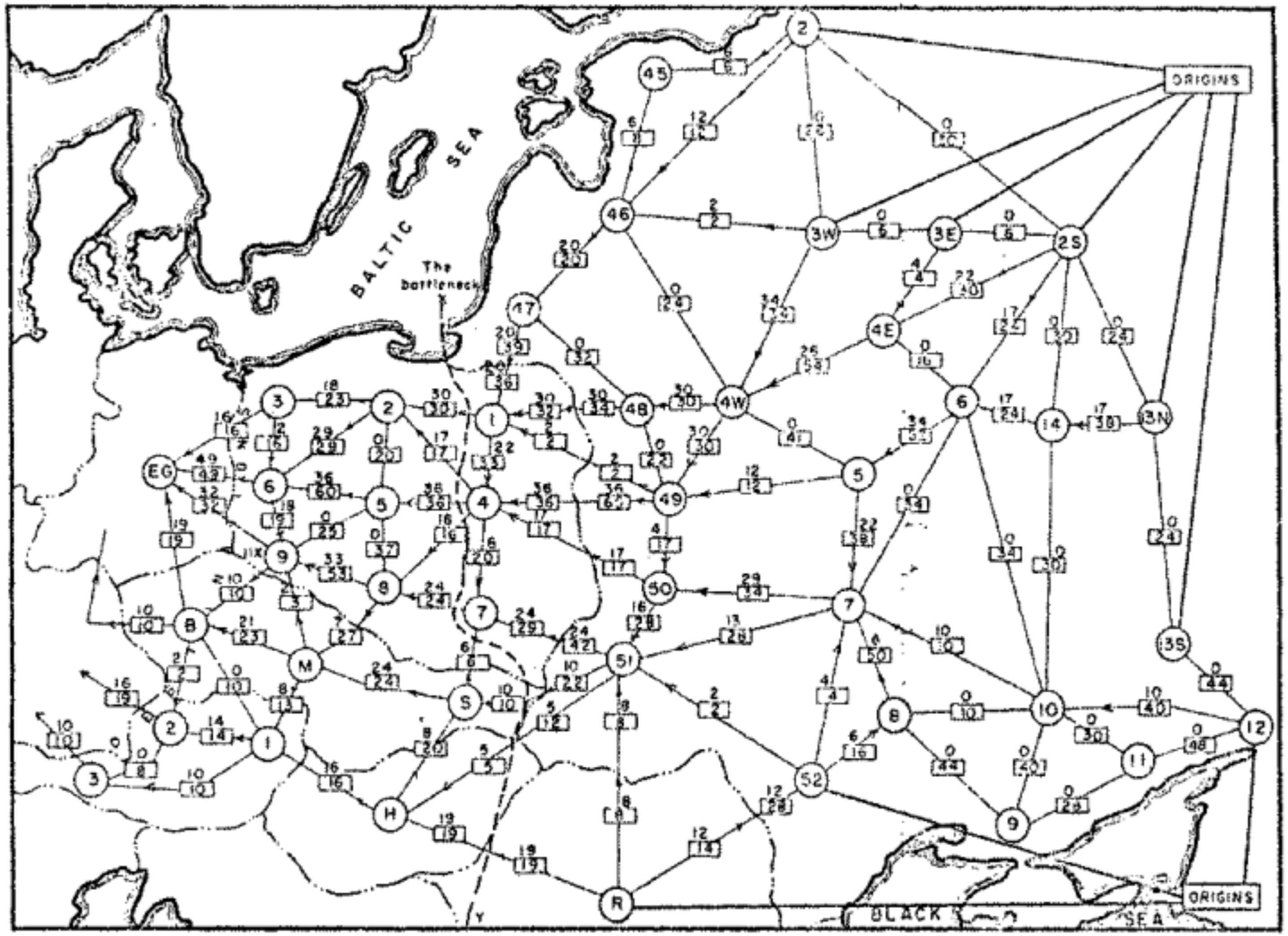
Network Flow



Railway map of Western USSR, 1955

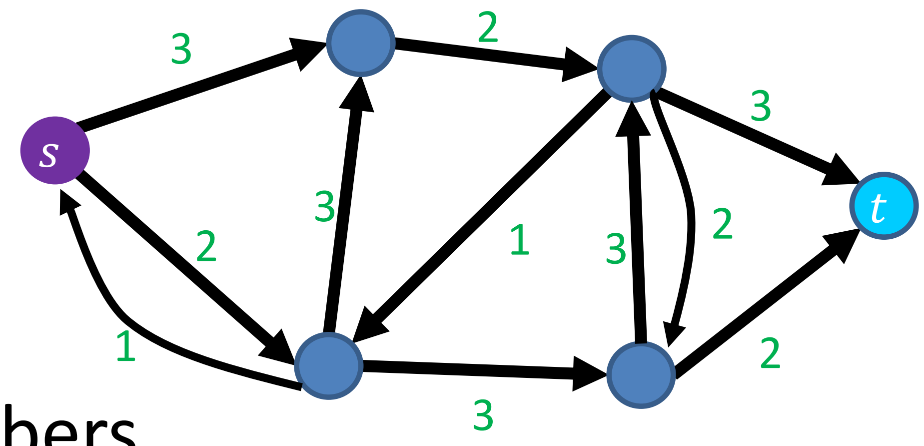
Question: What is the maximum throughput of the railroad network?

Today’s Keywords

- Max Flow, Min Cut
- Reductions
- Bipartite Matching
- Vertex Cover
- Independent Set

- CLRS Chapter 34

Flow Network



Graph $G = (V, E)$

Source node $s \in V$

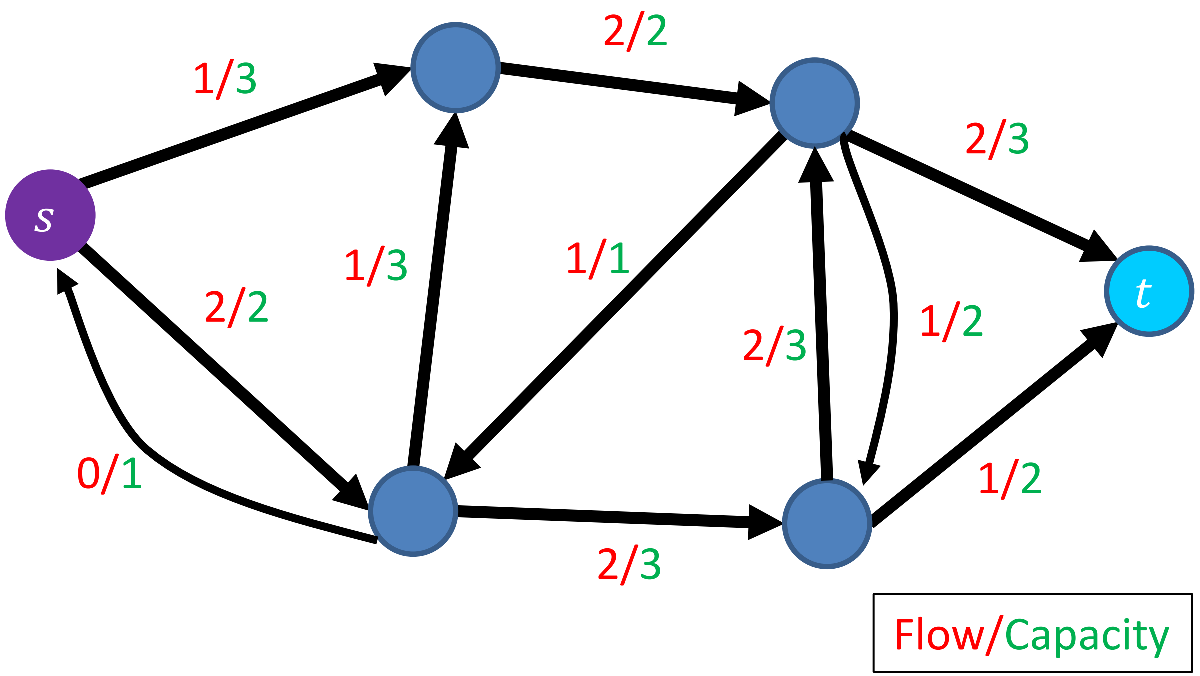
Sink node $t \in V$

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

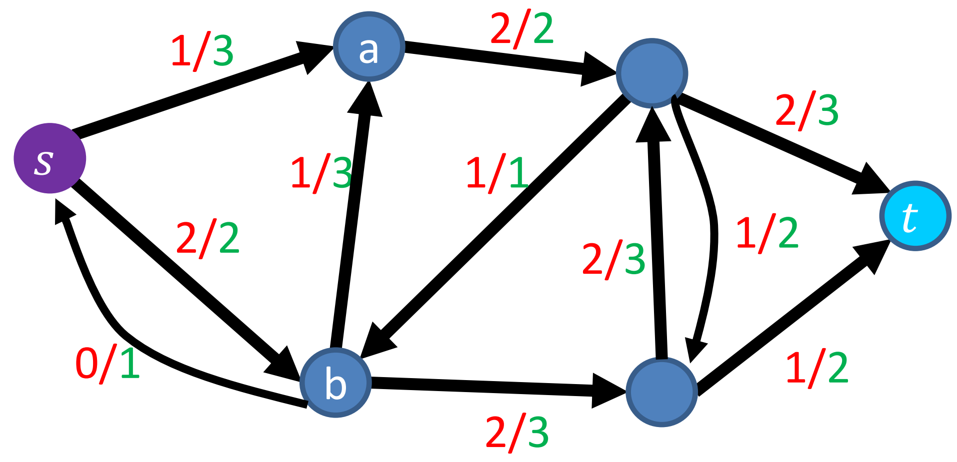
- Keep track of net available flow along each edge
- Forward edges: weight is equal to available flow along that edge in the flow graph
  - $w(e) = c(e) - f(e)$

Flow I *could* add

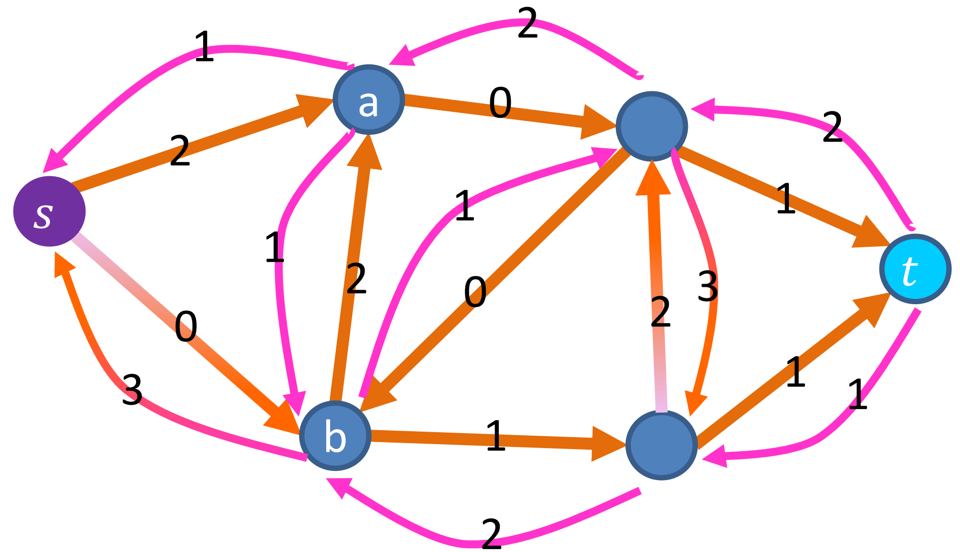
- Back edges: weight is equal to flow along that edge in the flow graph
  - $w(e) = f(e)$

Flow I *could* remove

Flow Graph G

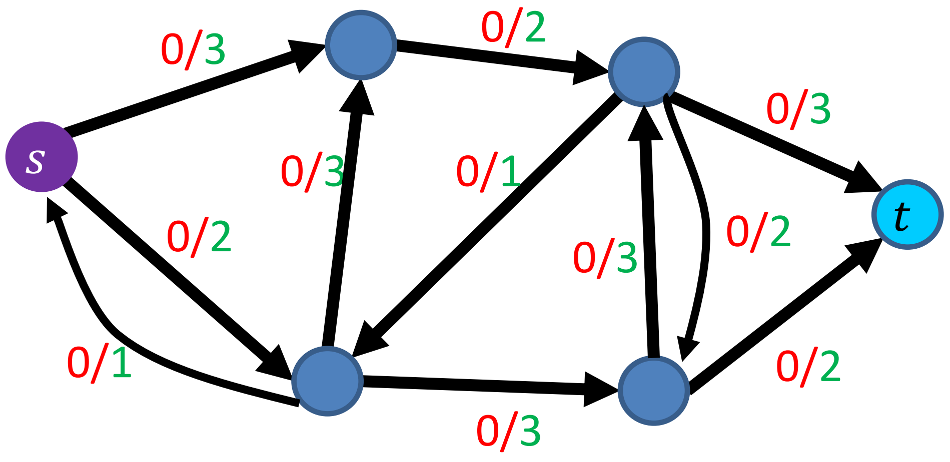


Residual Graph G_f

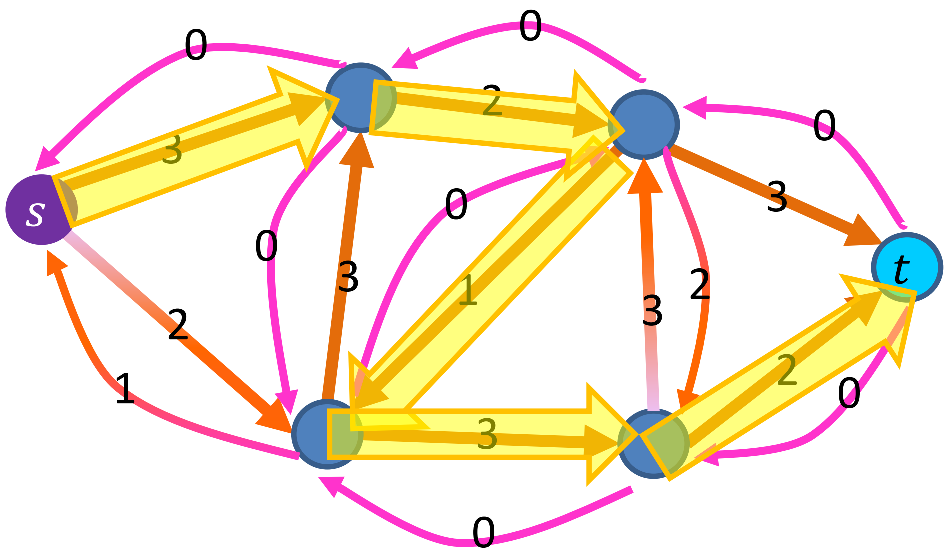


Ford Fulkerson: example

Flow Graph G

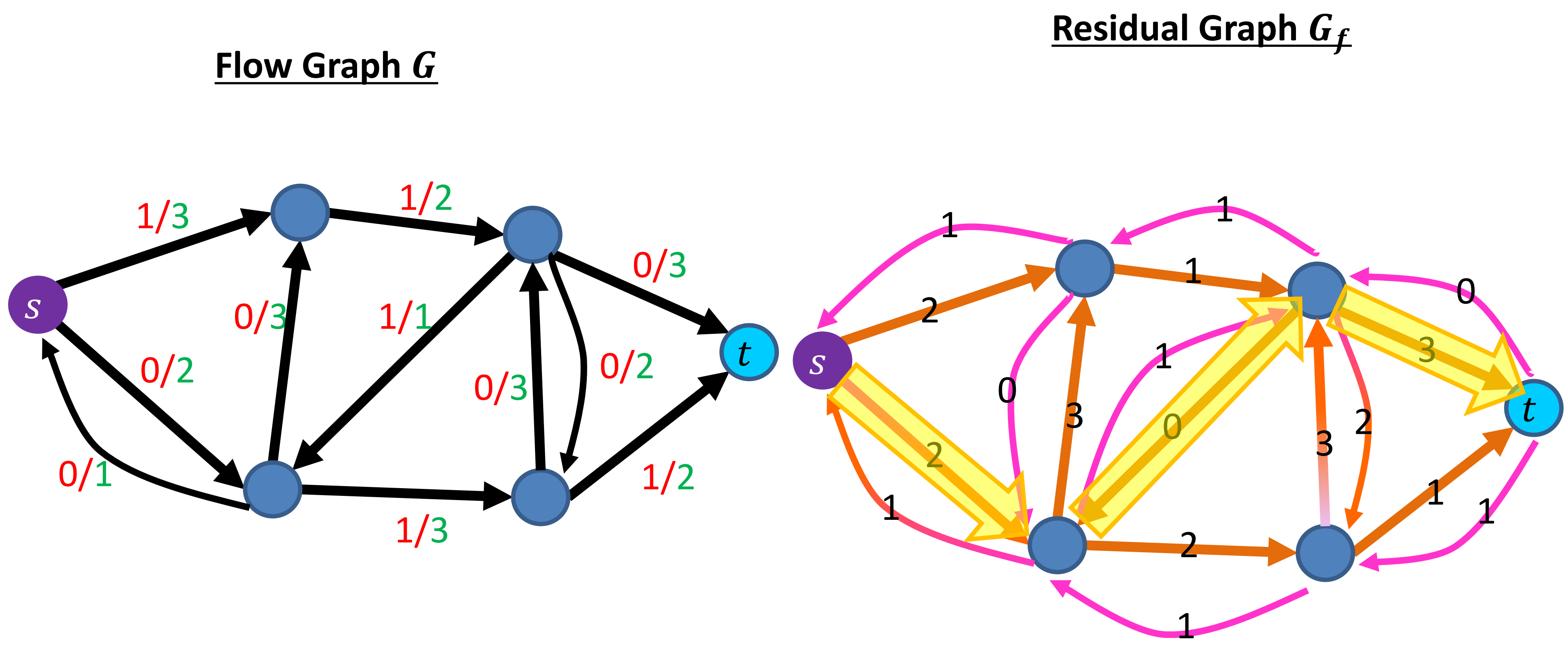


Residual Graph G_f



Add flow of 1 to this path

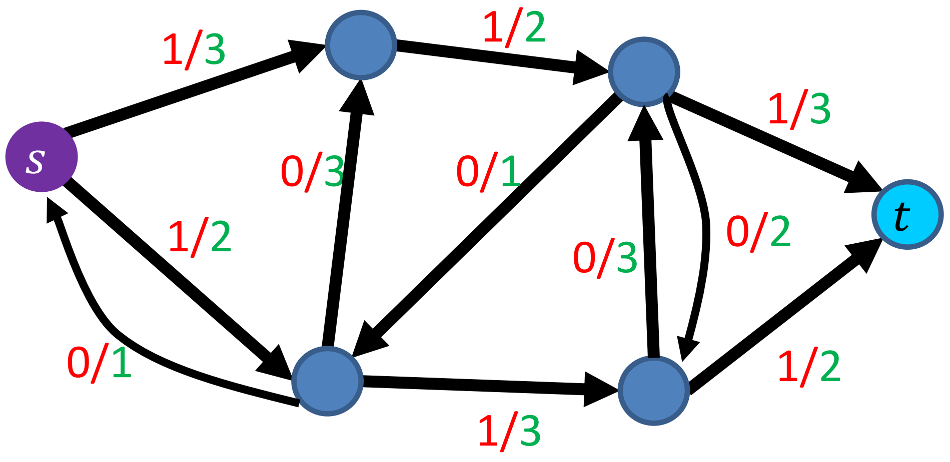
Ford Fulkerson: example



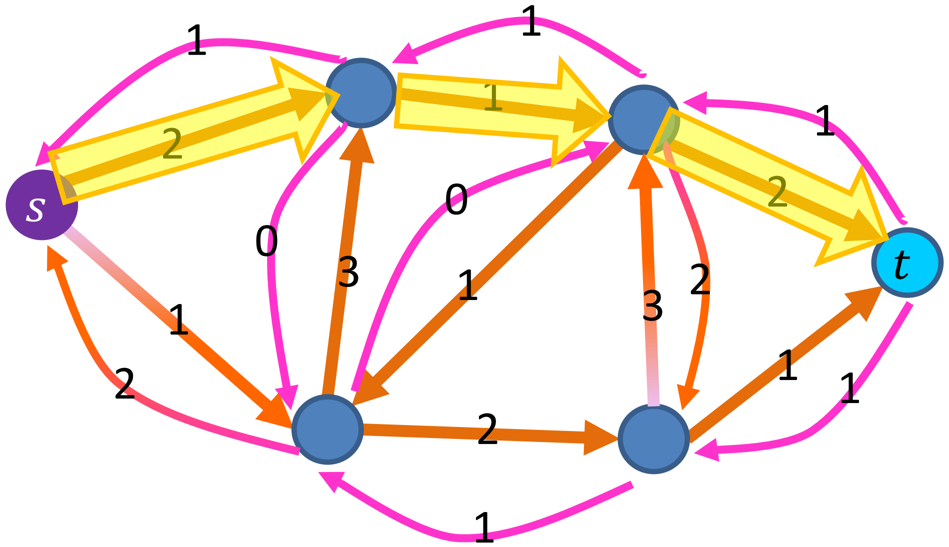
Add flow of 1 to this path

Ford Fulkerson: example

Flow Graph G



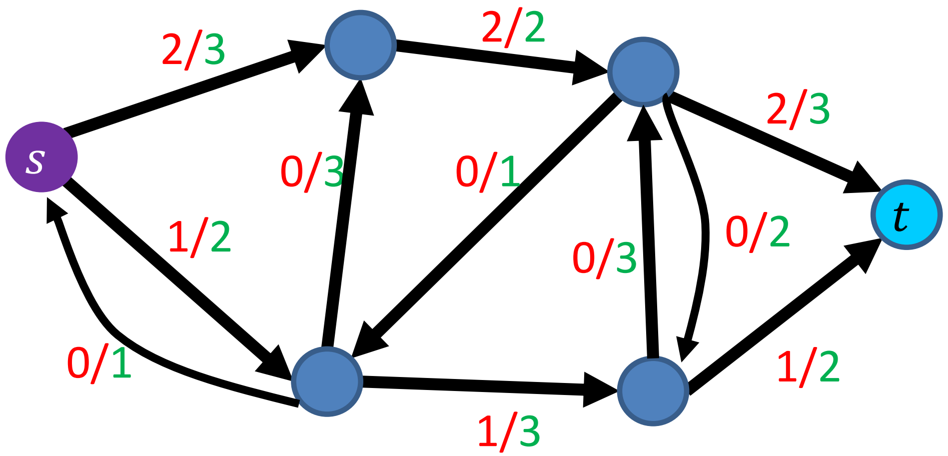
Residual Graph G_f



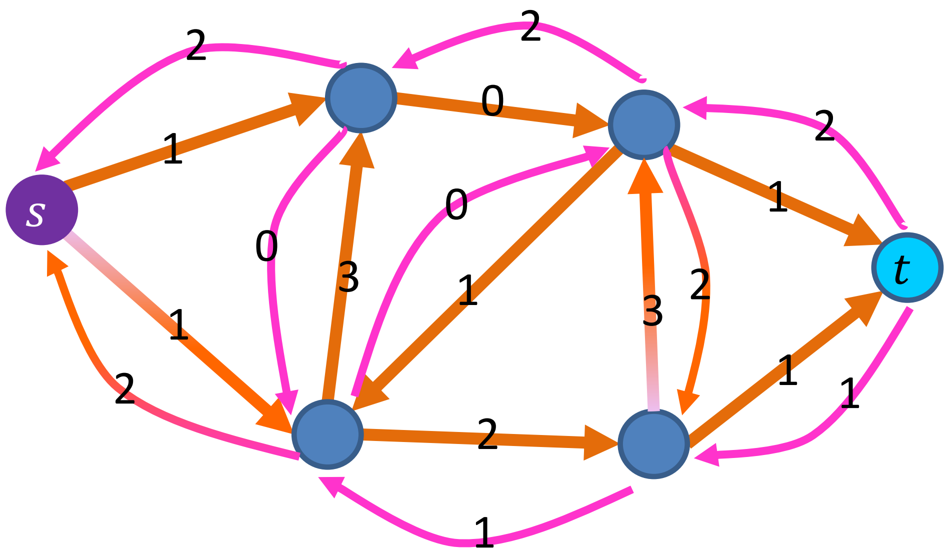
Add flow of 1 to this path

Ford Fulkerson: example

Flow Graph G



Residual Graph G_f



Ford-Fulkerson Running Time

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 $c = \min_{e \in E} c_f(e)$ ($c_f(e)$ 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

Initialization: $O(|E|)$

Ford-Fulkerson Running Time

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 $c = \min_{e \in E} c_f(e)$ ($c_f(e)$ 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

Initialization: $O(|E|)$

Construct residual network: $O(|E|)$

Ford-Fulkerson Running Time

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 $c = \min_{e \in E} c_f(e)$ ($c_f(e)$ is the weight of edge e in the residual network G_f)
  - Add c units of flow to G based on the augme
  - Update the residual network G_f for the upda

We only care about nodes reachable from the source s (so the number of nodes that are "relevant" is at most $|E|$)

Initialization: $O(|E|)$

Construct residual network: $O(|E|)$

Finding augmenting path in residual network: $O(|E|)$ using BFS/DFS

Ford-Fulkerson Running Time

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

How many iterations are needed?

- For integer-valued capacities, min-weight of each augmenting path is 1, so number of iterations is bounded by $|f^*|$, where $|f^*|$ is max-flow in G
- For rational-valued capacities, can scale to make capacities integer
- For irrational-valued capacities, algorithm may never terminate!

Initialization: $O(|E|)$

Construct residual network: $O(|E|)$

Finding augmenting path in residual network: $O(|E|)$ using BFS/DFS

Ford-Fulkerson Running Time

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 algorith

- Initialize $f(e) = 0$ for all e
- Construct the residual net
- While there is an augmen
  - Let $c = \min_{e \in E} c_f(e)$ $(c_f($
  - Add c units of flow to
  - Update the residual n

Initialization: $O(|E|)$

Construct residual network: $O(|E|)$

Finding augmenting path in residual network: $O(|E|)$ using BFS/DFS

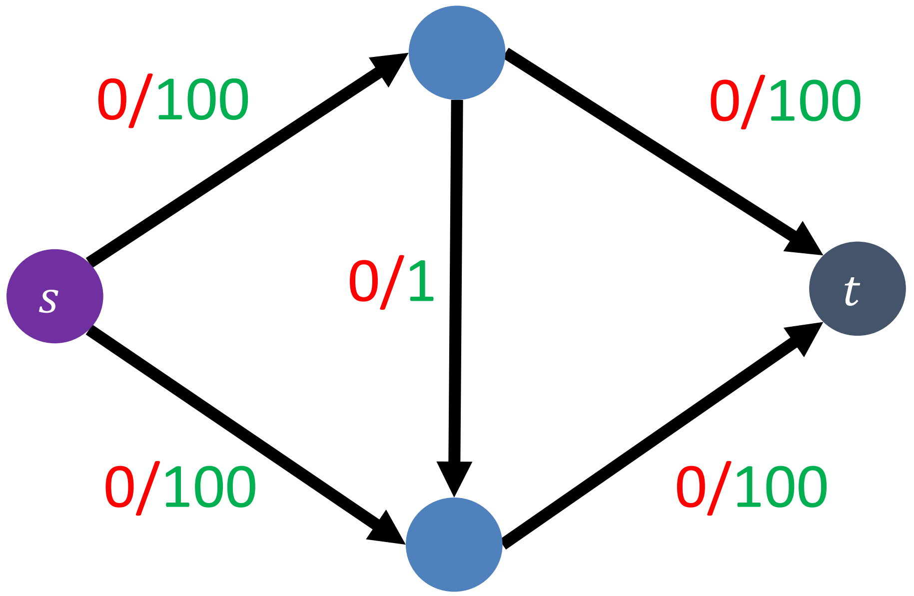
For graphs with integer capacities, running time of Ford-Fulkerson is

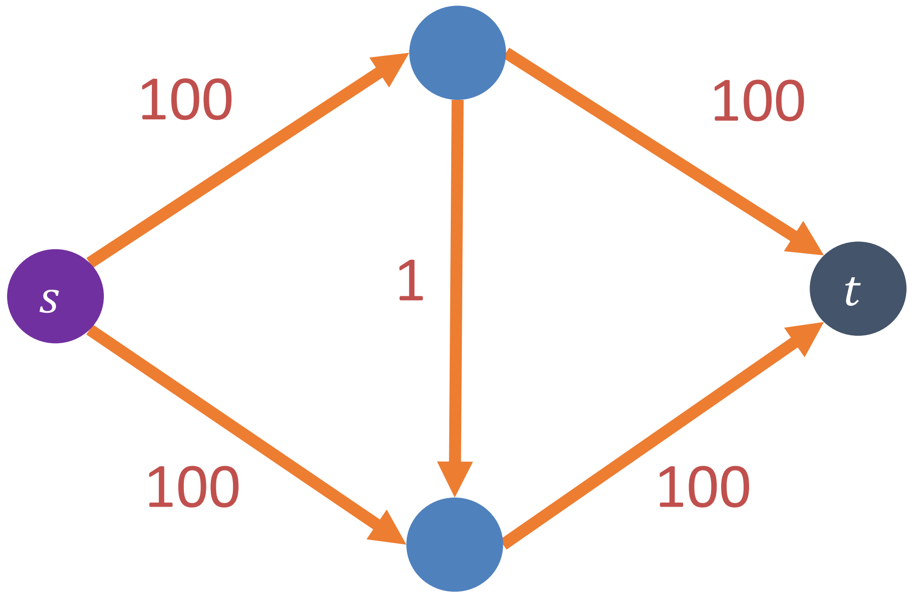
$$O(|f^*| \cdot |E|)$$

Highly undesirable if $|f^*| \gg |E|$ (e.g., graph is small, but capacities are $\approx 2^{32}$)

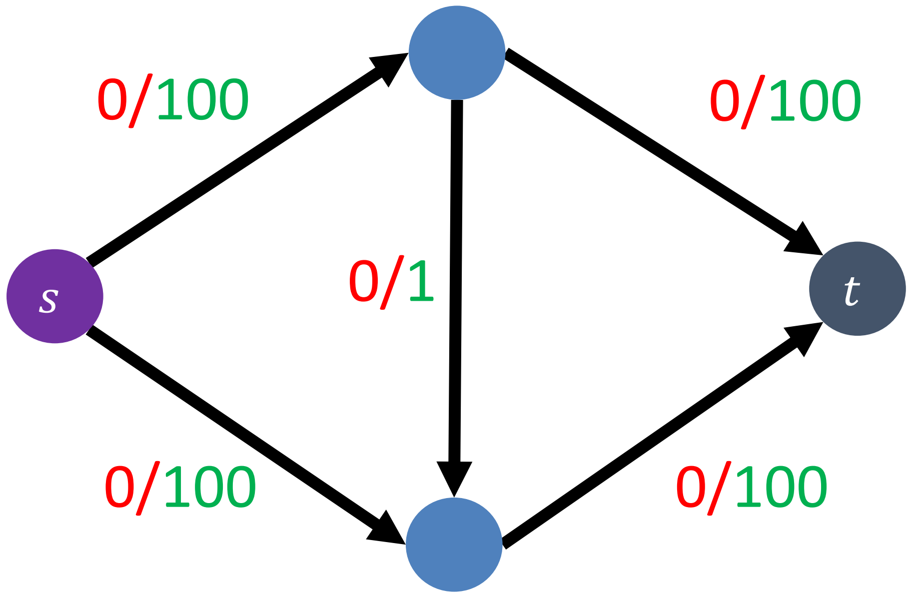
As described, algorithm is not polynomial-time!

Worst-Case Ford-Fulkerson

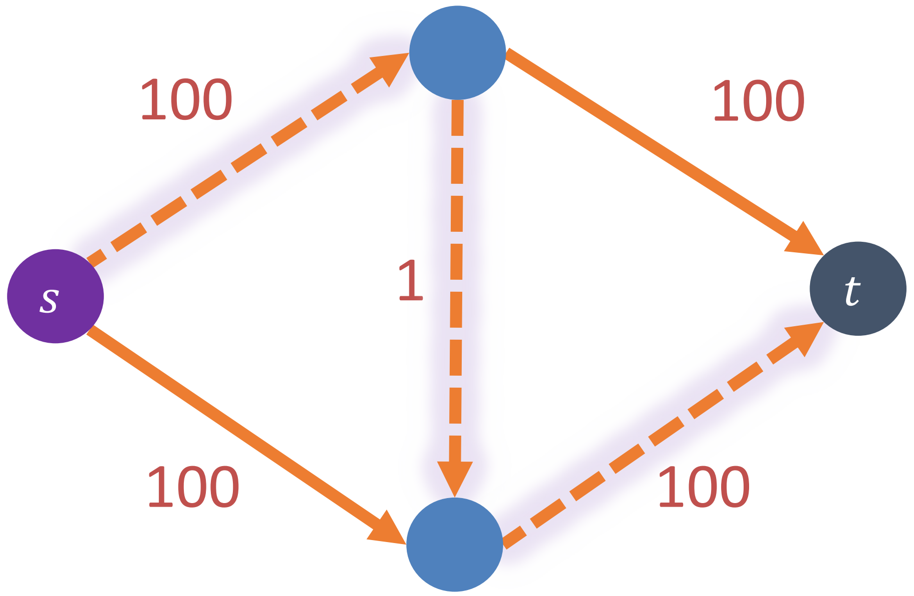




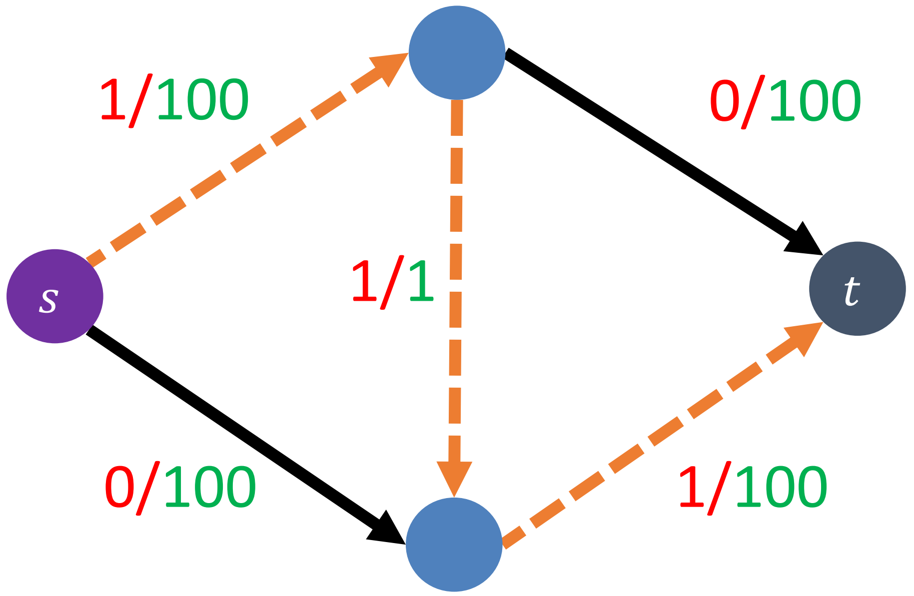
Worst-Case Ford-Fulkerson



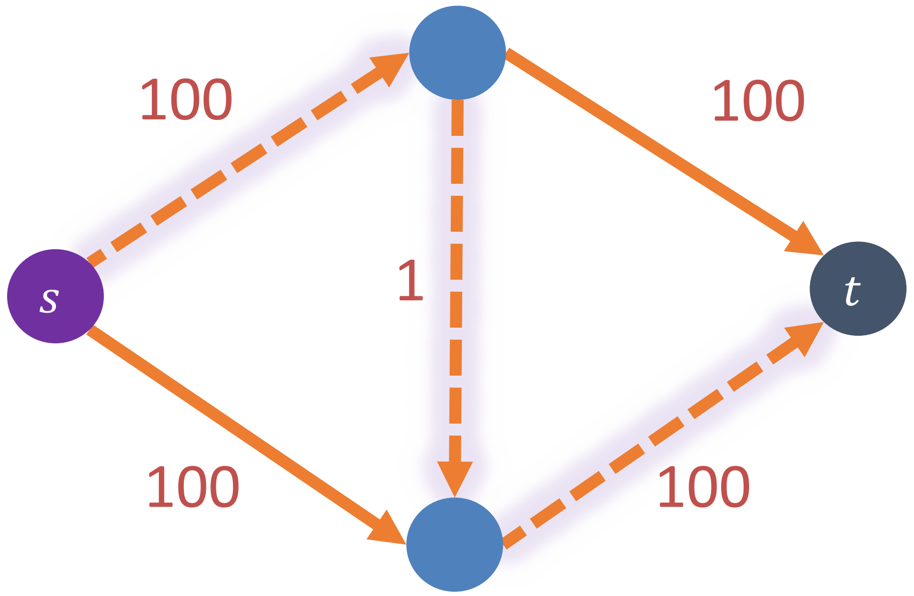
Increase flow by 1 unit



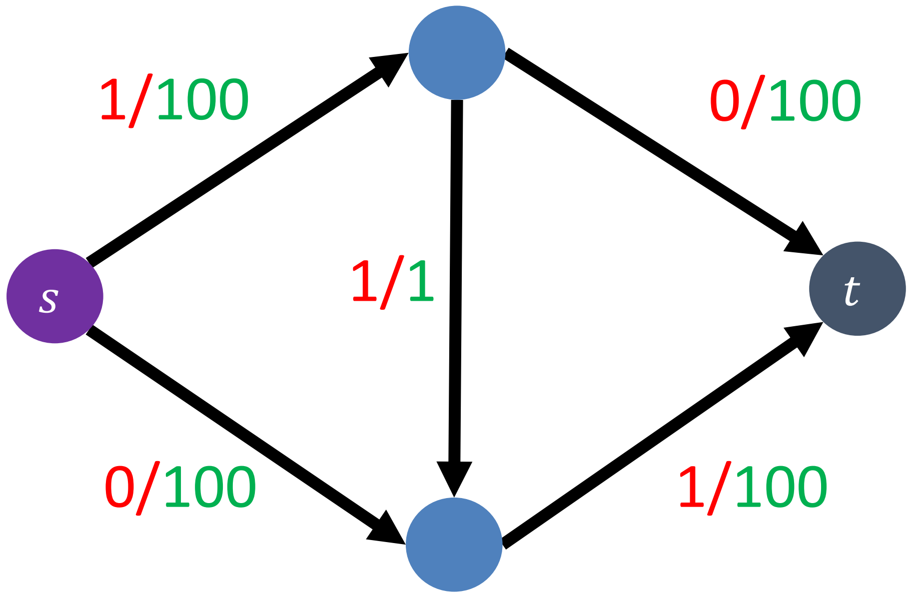
Worst-Case Ford-Fulkerson

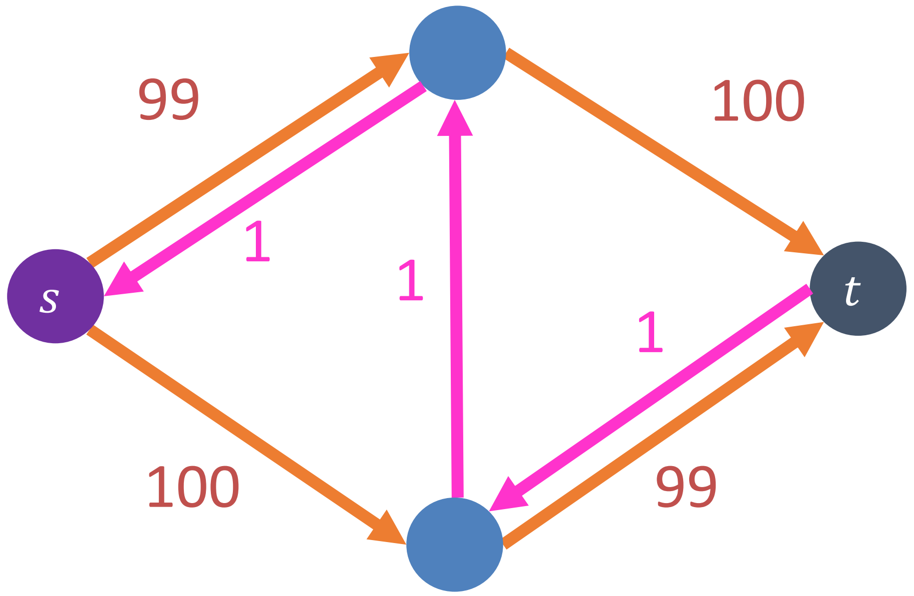


Increase flow by 1 unit

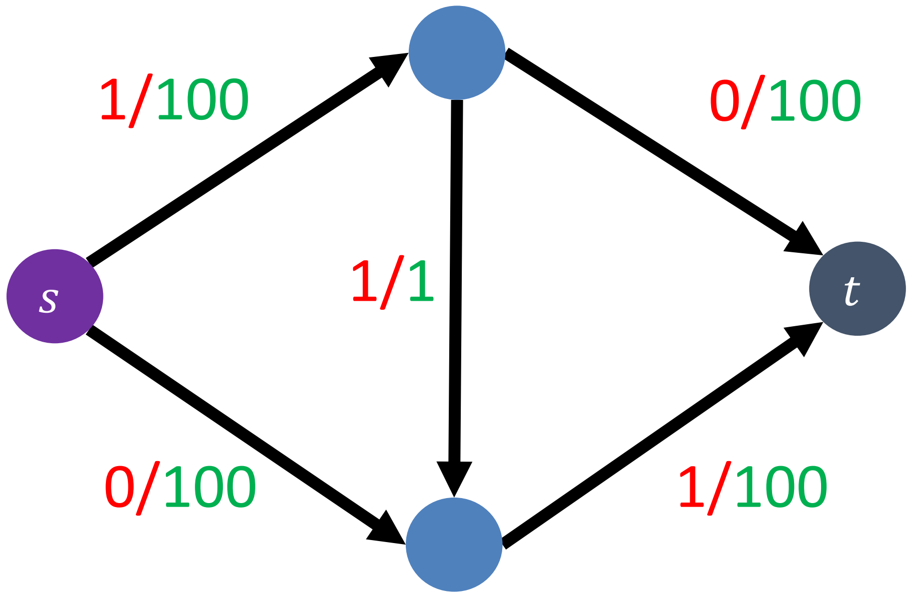


Worst-Case Ford-Fulkerson

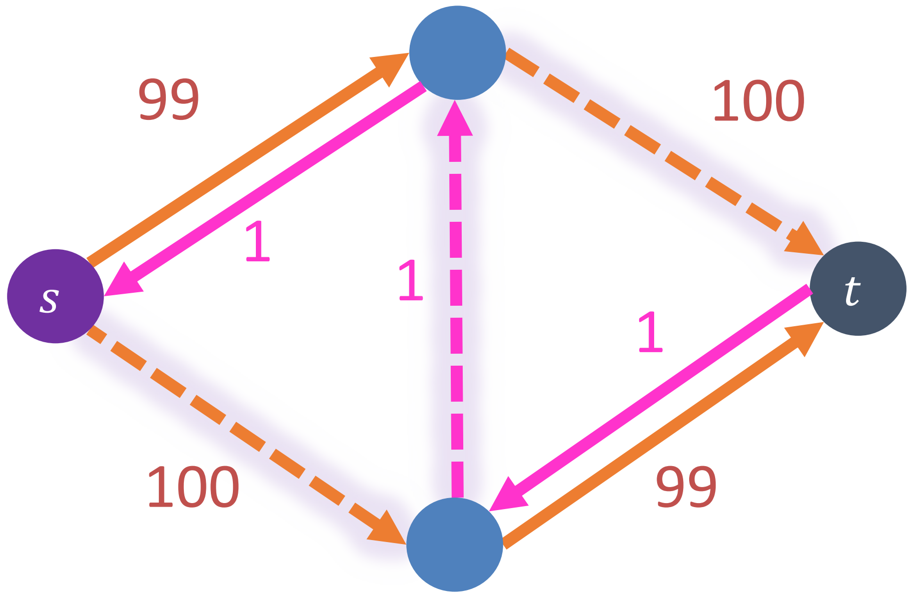




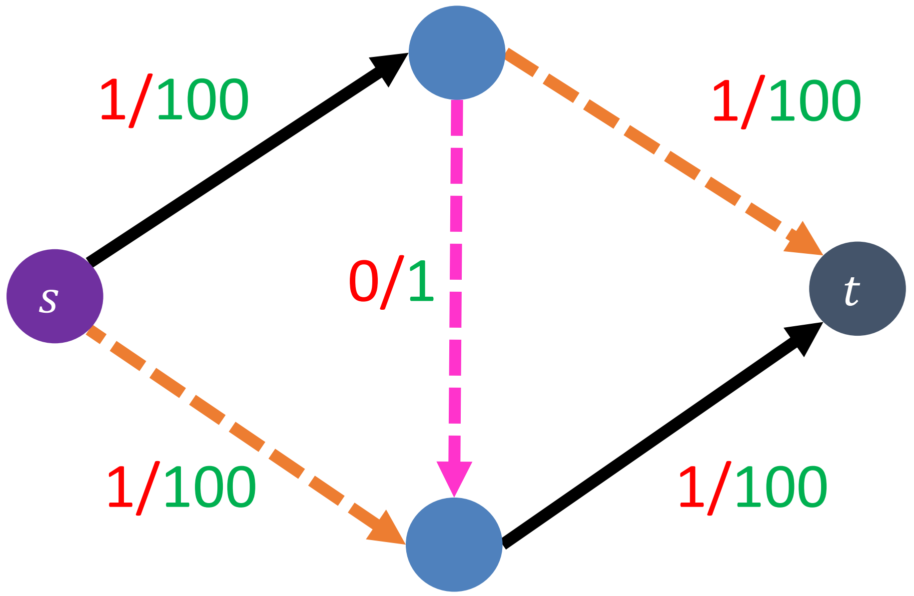
Worst-Case Ford-Fulkerson



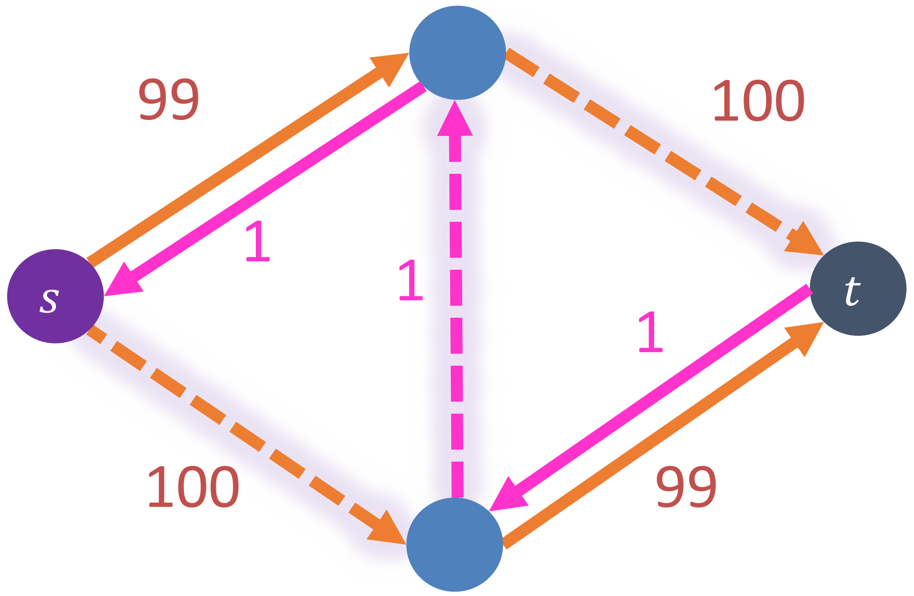
Increase flow by 1 unit



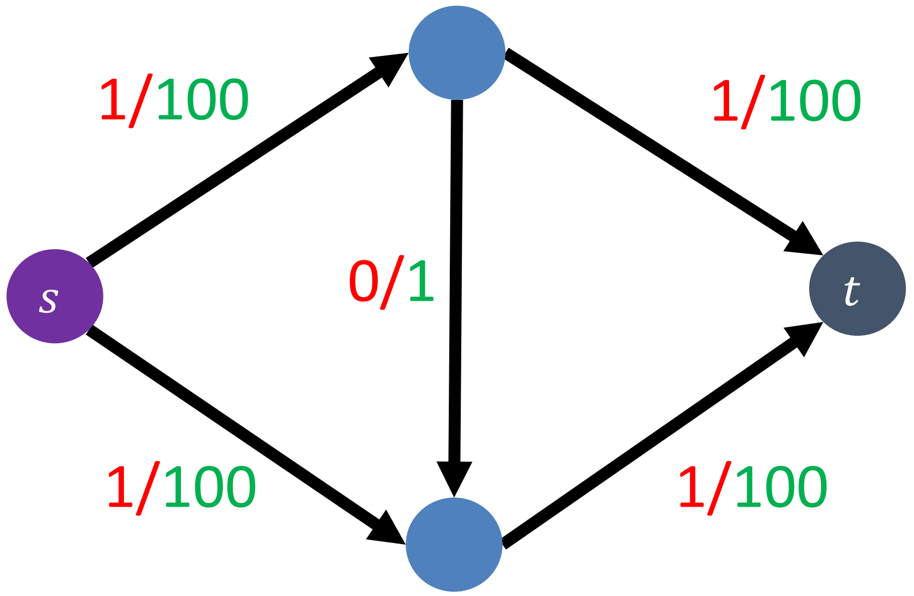
Worst-Case Ford-Fulkerson

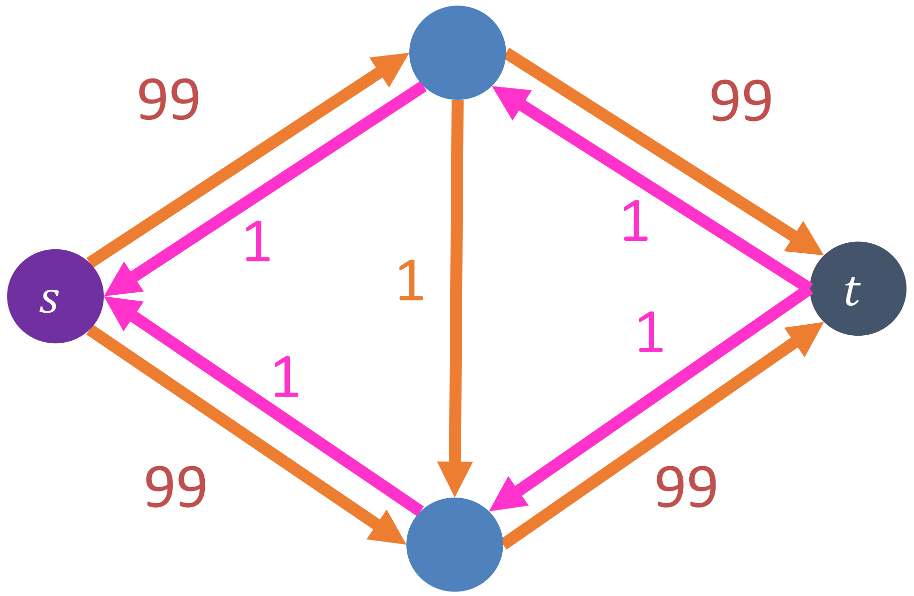


Increase flow by 1 unit

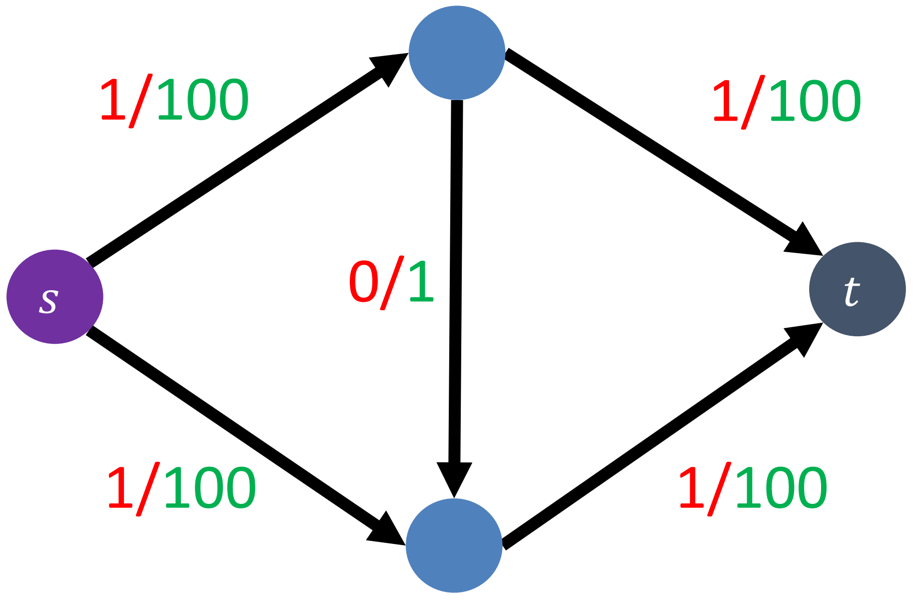


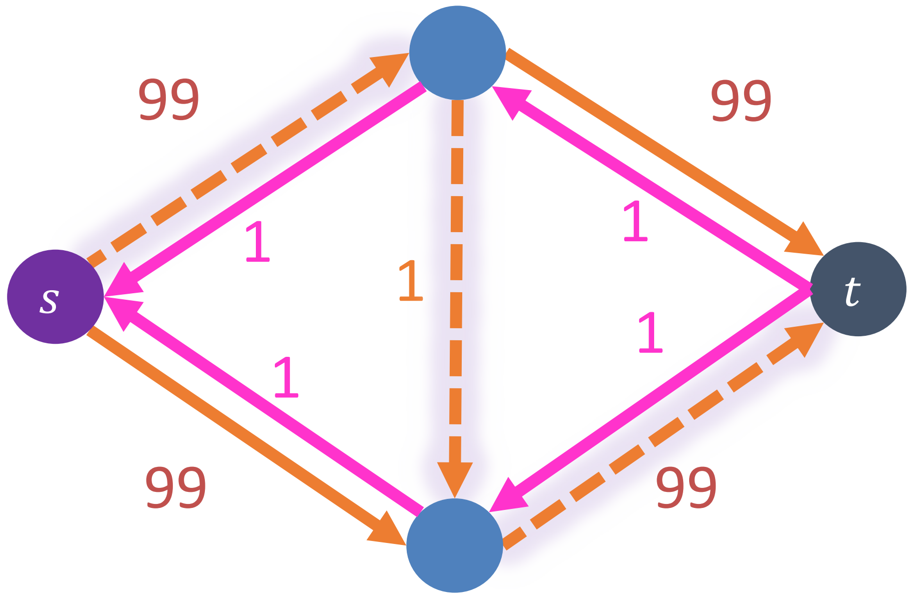
Worst-Case Ford-Fulkerson





Worst-Case Ford-Fulkerson





Observation: each iteration increases flow by 1 unit

Total number of iterations: $|f^*| = 200$

Can We Avoid this?

- **Edmonds-Karp Algorithm:** choose augmenting path with fewest hops
- **Running time:** $\Theta(\min(|E||f^*|, |V||E|^2))$

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

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

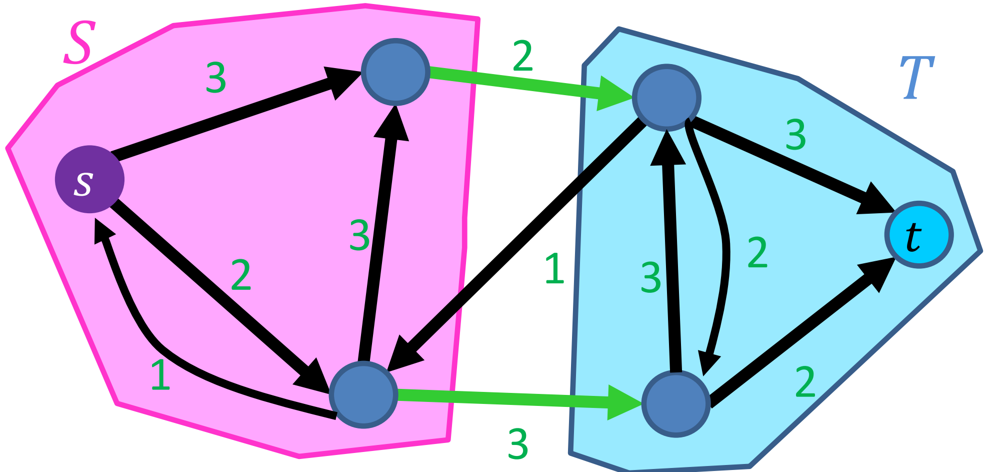
Ford-Fulkerson max-flow algorithm:

- 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

Proof: See CLRS (Chapter 26.2)

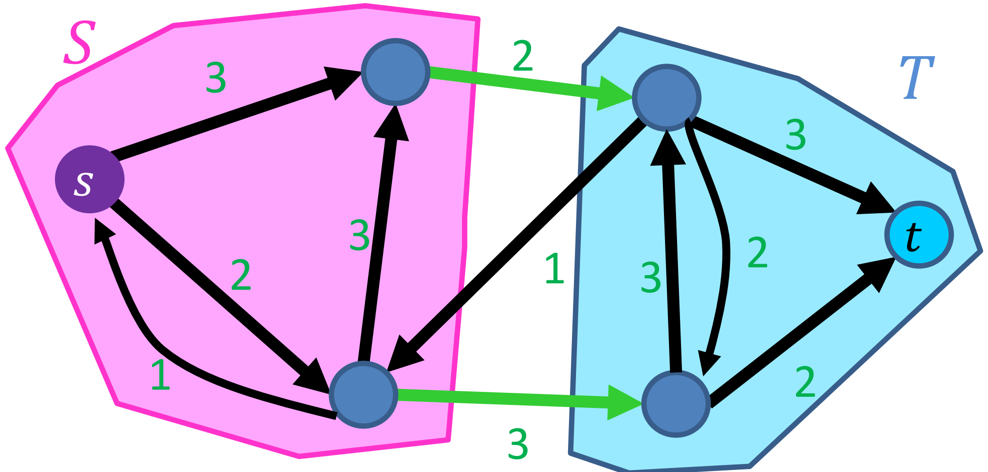
Showing Correctness of Ford-Fulkerson

- 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
  - $\max_f |f| \leq \min_{S,T} ||S,T||$

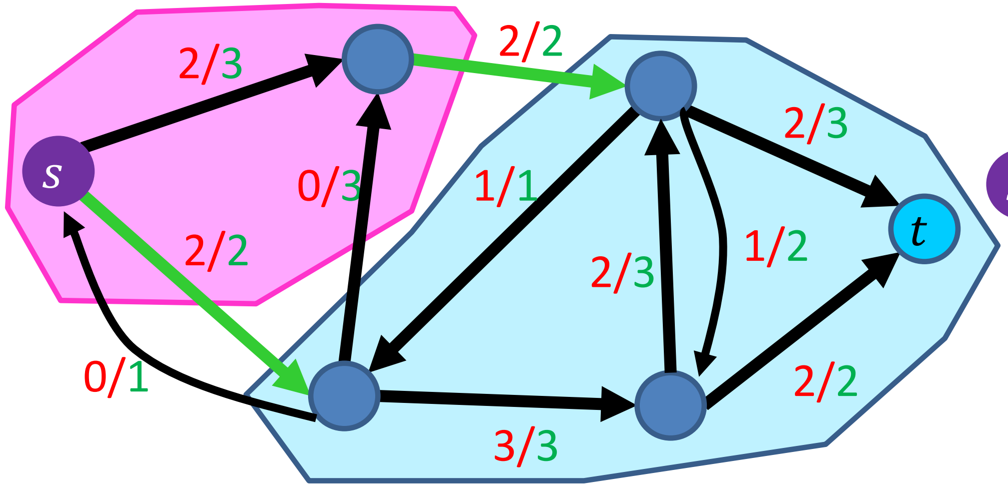


Maxflow/Mincut Theorem

- 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

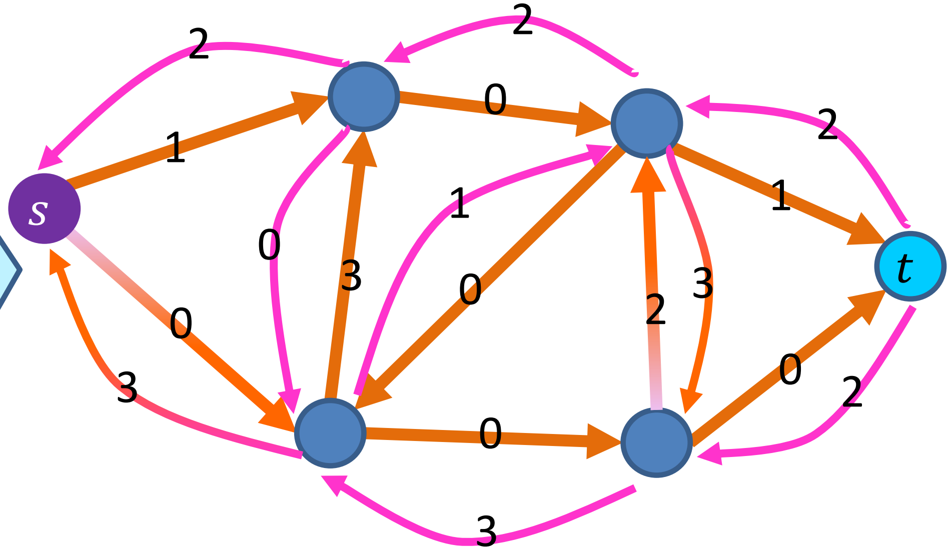
Flow Graph G



$$|f| = 4$$

$$||S, T|| = 4$$

Residual Graph G_f



No Augmenting Paths

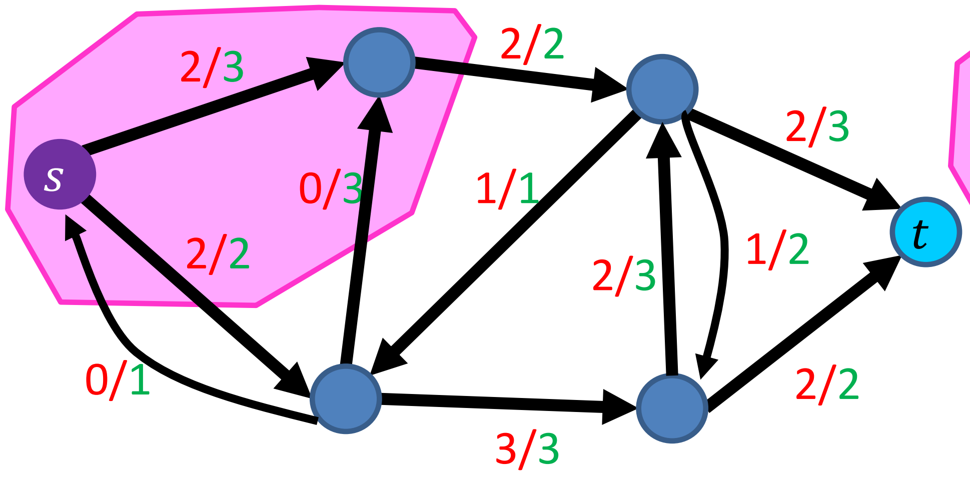
Idea: When there are no more augmenting paths, there exists a cut in the graph with cost matching the flow

28

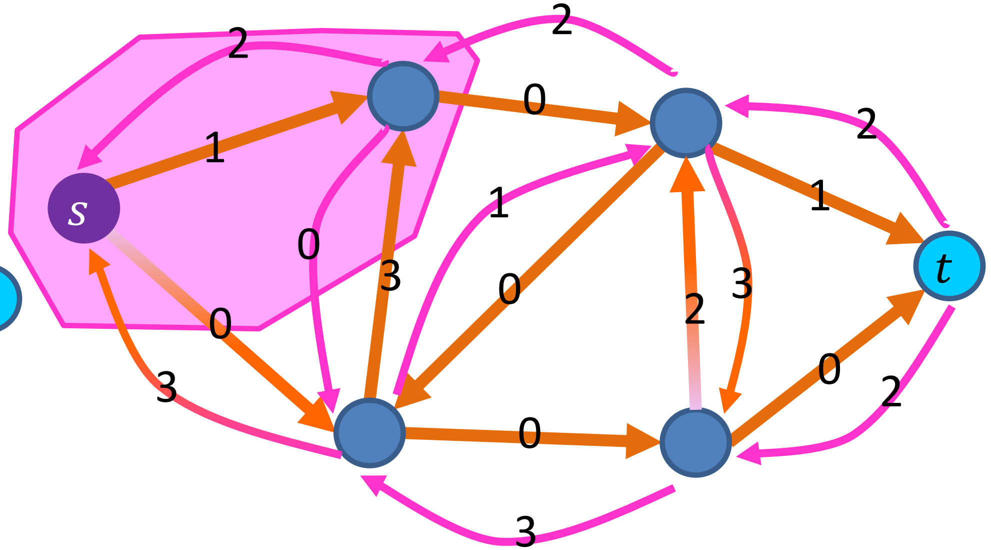
Proof: Maxflow/Mincut Theorem

- 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



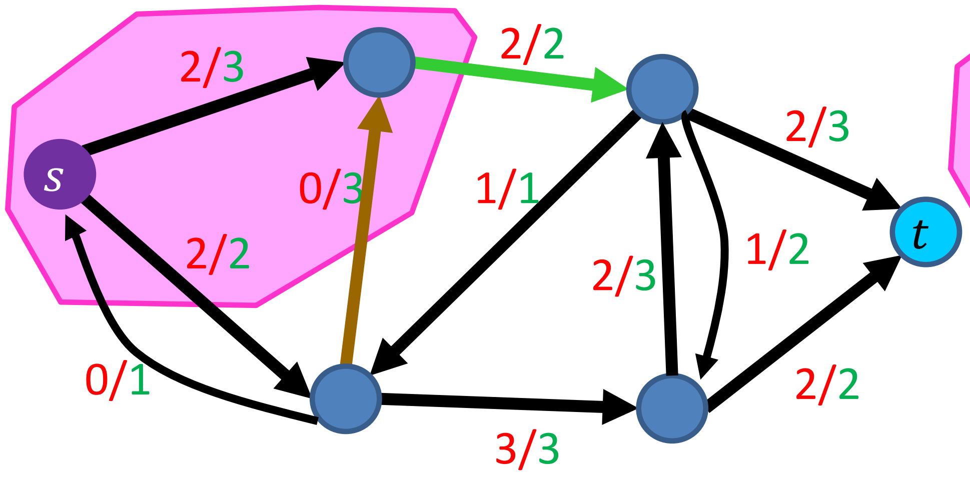
Residual Graph G_f



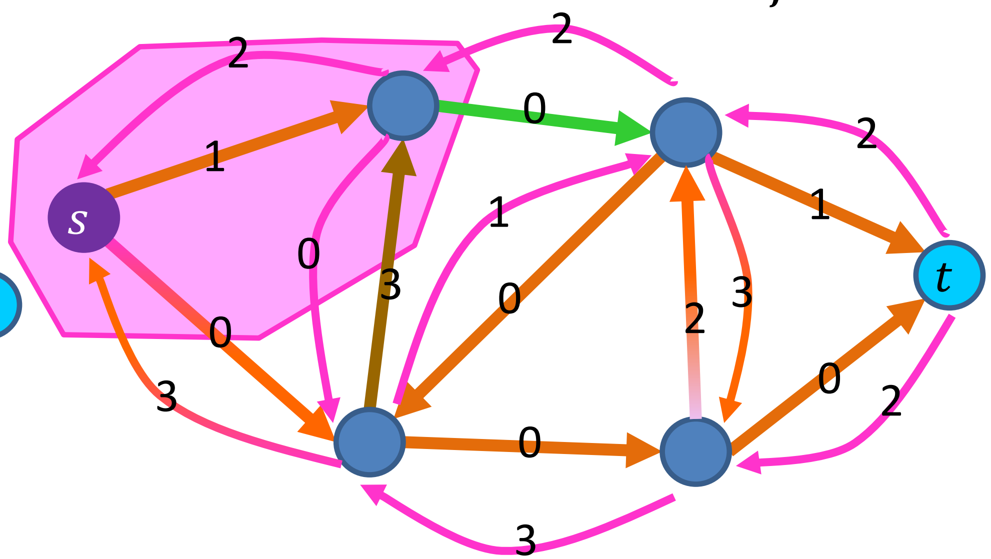
Proof: Maxflow/Mincut Theorem

- 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

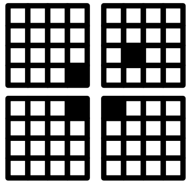
3. When there are no more augmenting paths in G_f then we can define a cut $S =$ nodes reachable from source node s by positive-weight edges in the residual graph

4. The sum of edge capacities crossing this cut must match the flow of the graph

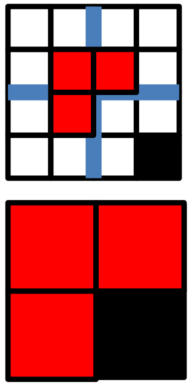
5. Therefore this flow is maximal

Divide and Conquer*

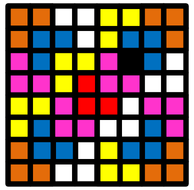
- **Divide**:



  – 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

Roadmap: Where We're Going and Why

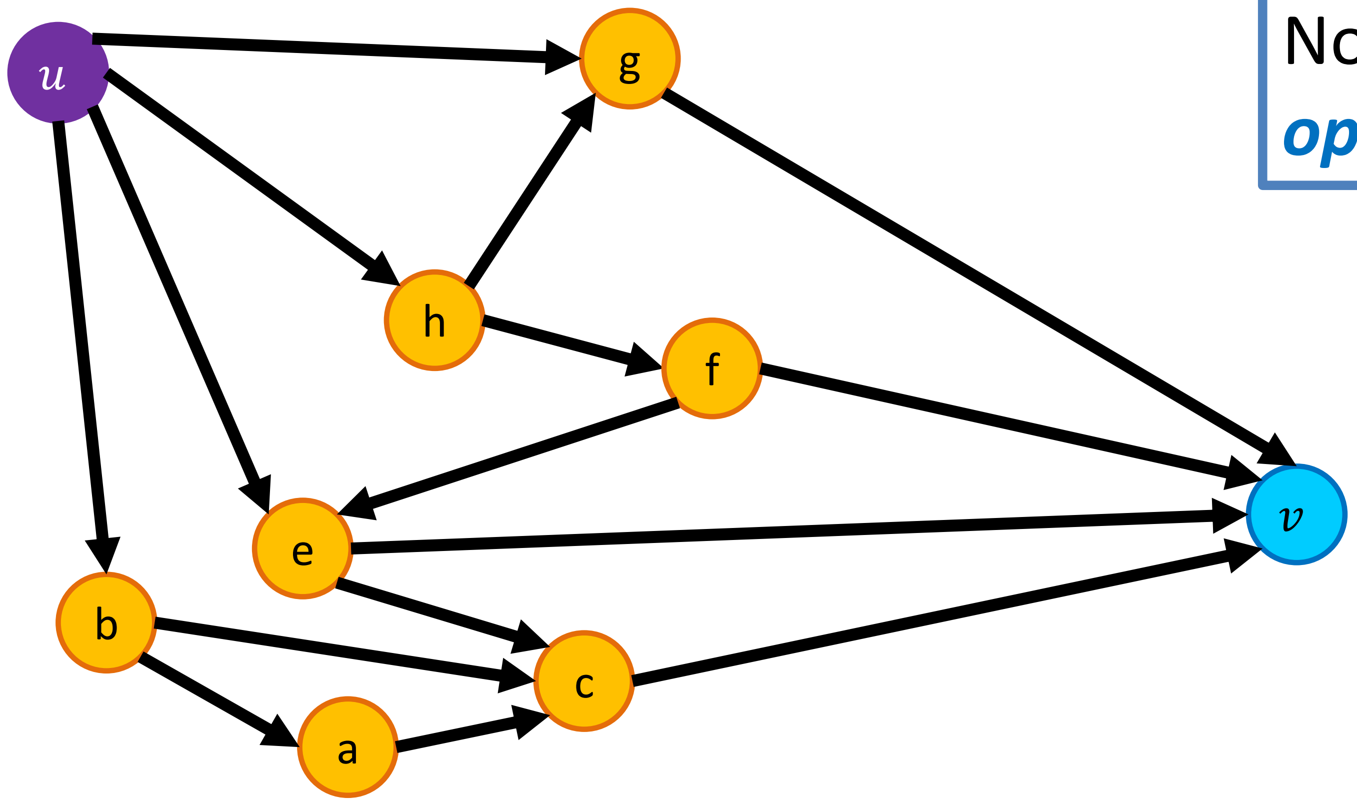
- ***Reductions*** between problems
  - Why? Can be a practical way of solving a new problem
  - Also: A proof about one problem's complexity can be applied to another
  - Formal definition of a reduction
- Examples
  - Bipartite graphs, matching
  - Vertex cover and independent set

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

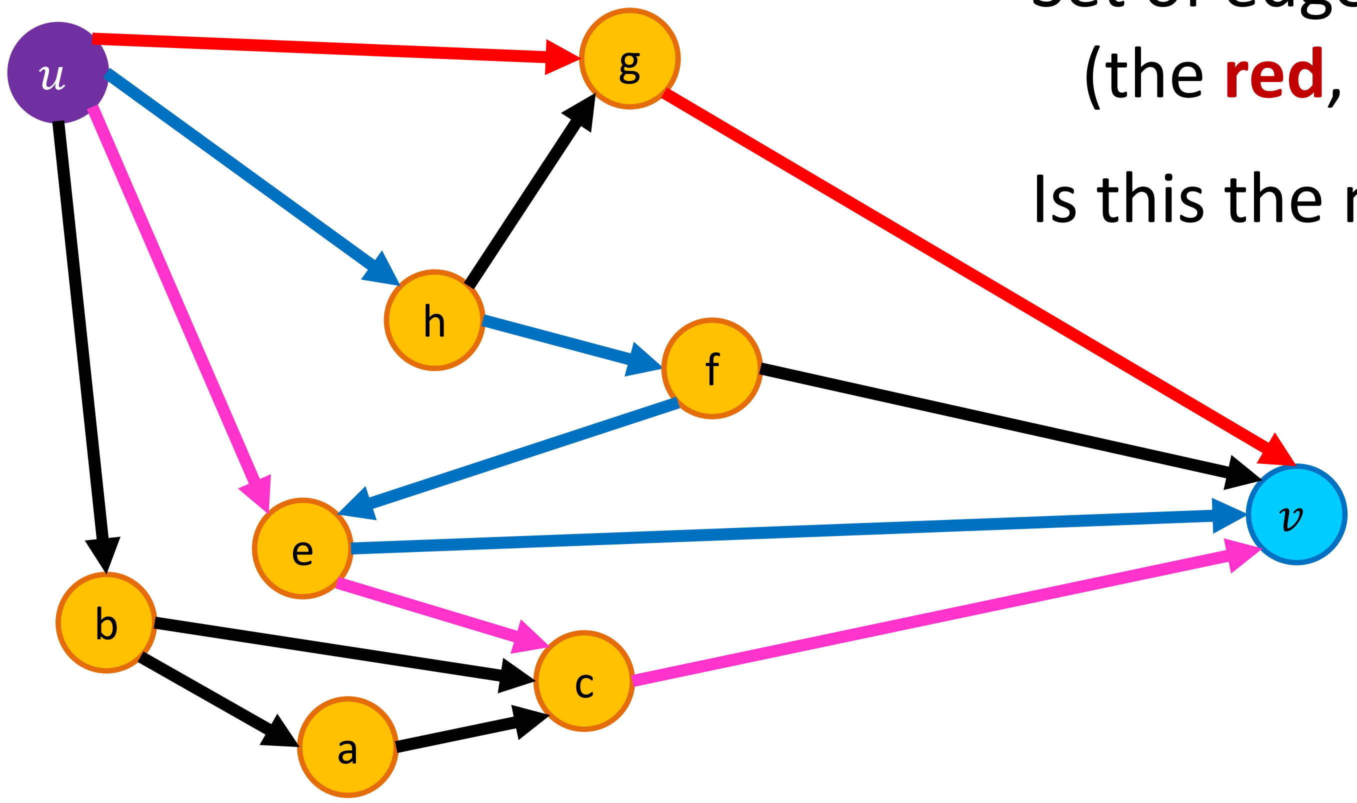
Given a graph $G = (V, E)$, a start node u and a destination node v, give the maximum number of paths from u to v which share no edges



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

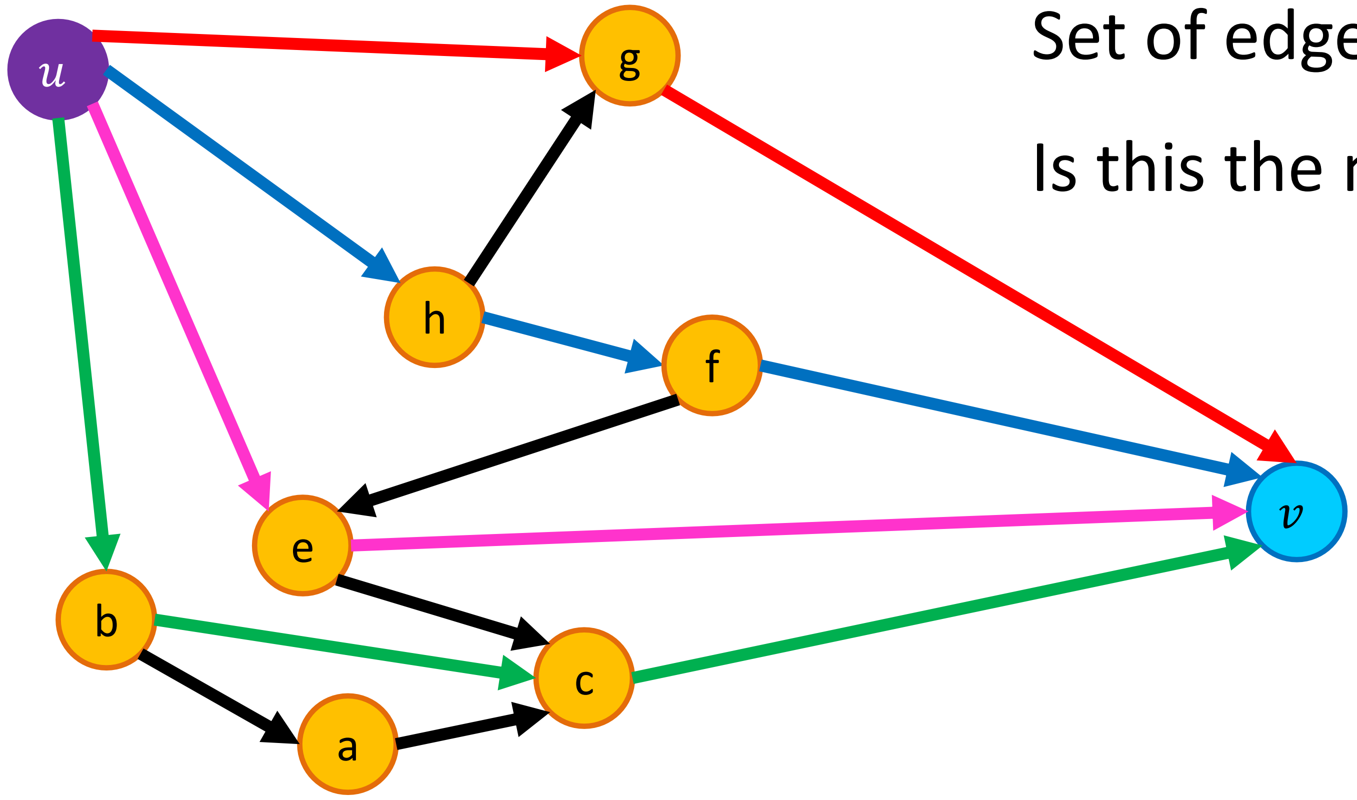


Set of edge-disjoint paths of size 3 (the **red**, **blue**, **magenta** paths)

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



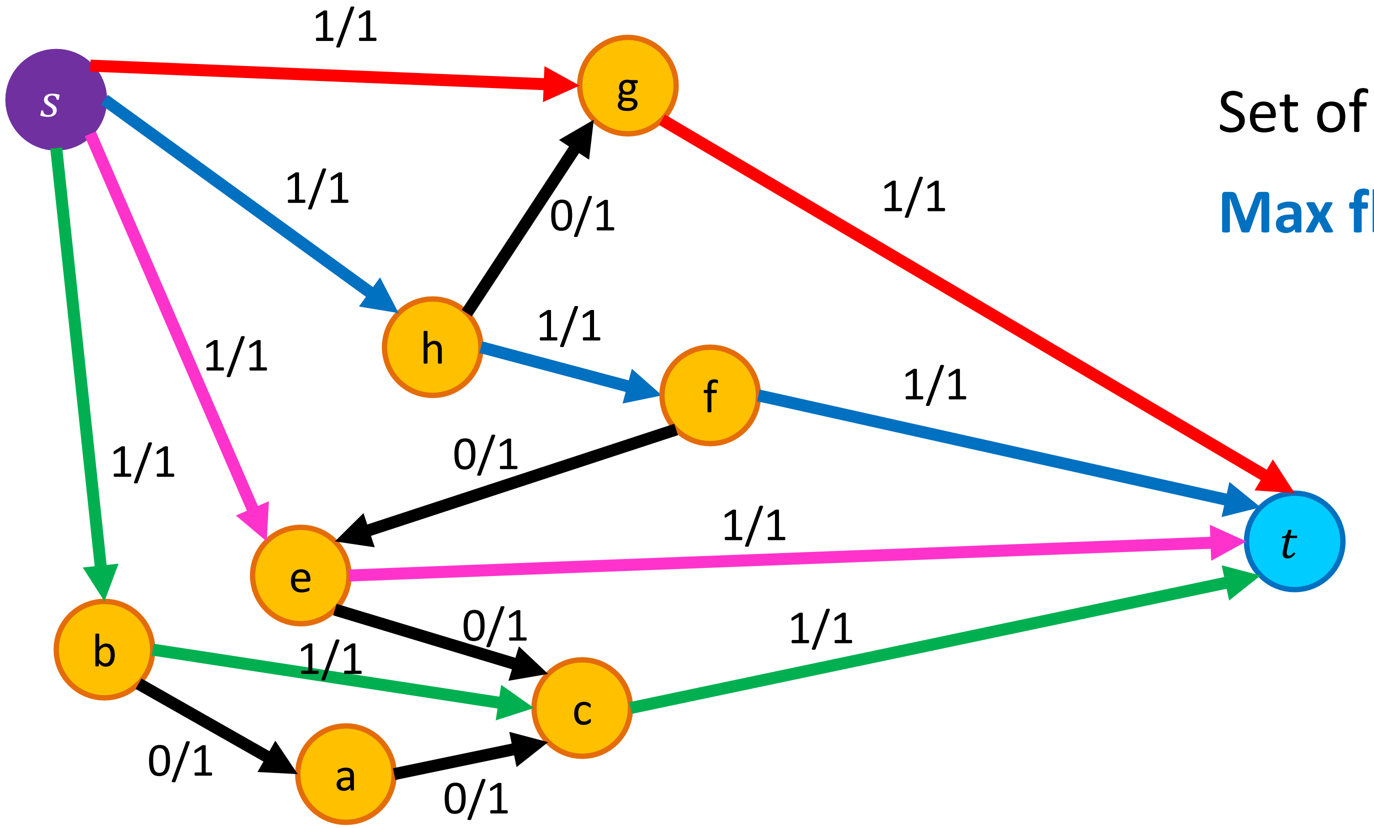
Set of edge-disjoint paths of size 4

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

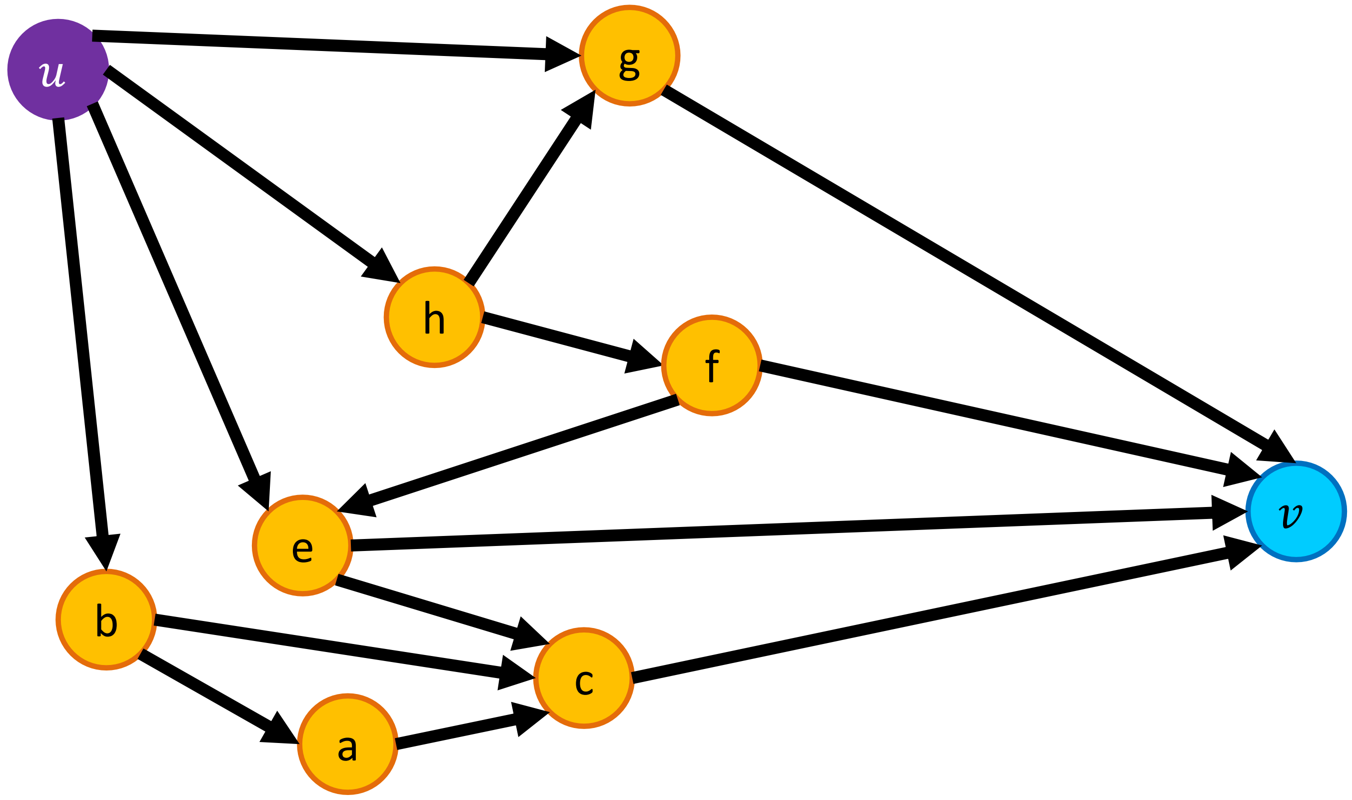
Why does this work? We need to be able to make a valid argument that it always does.

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

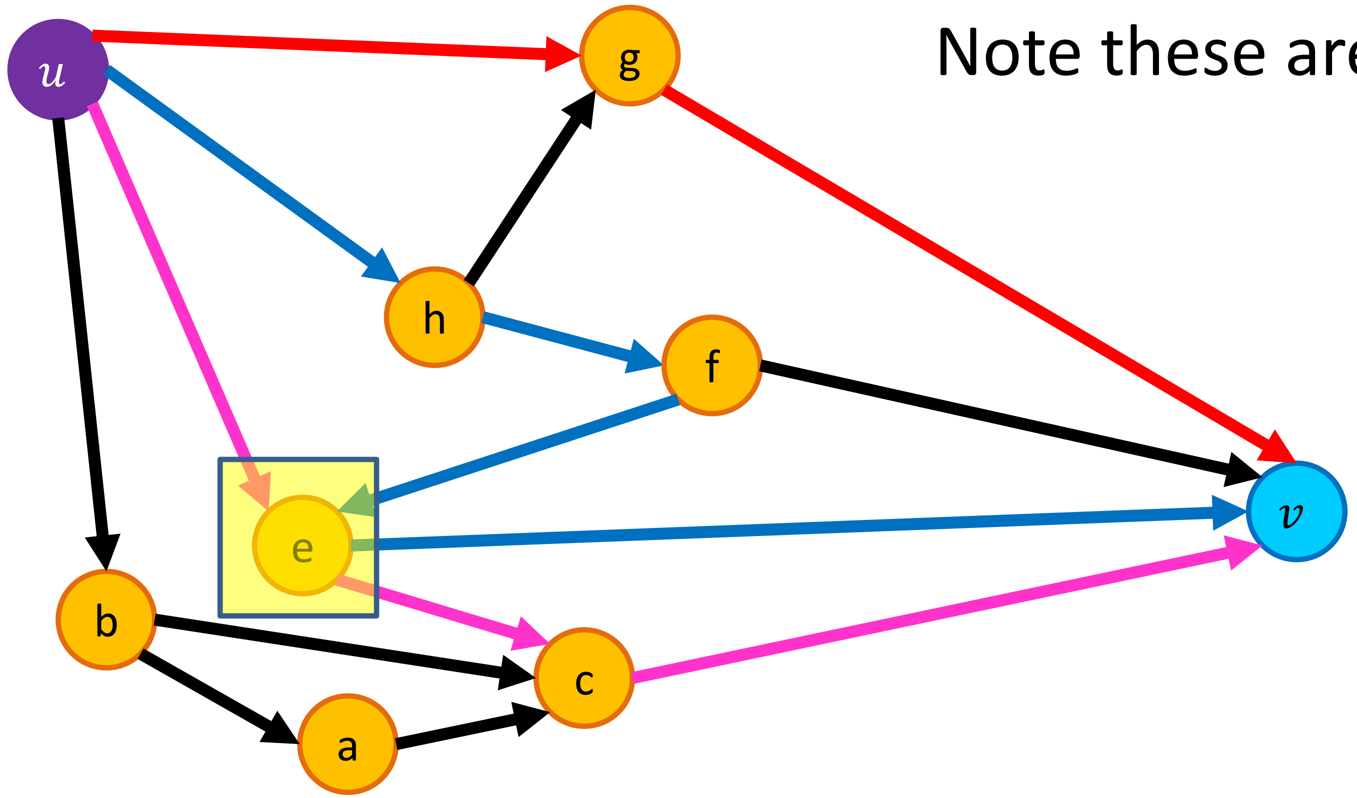
Vertex-Disjoint Paths

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 <u>edge</u>-disjoint paths.

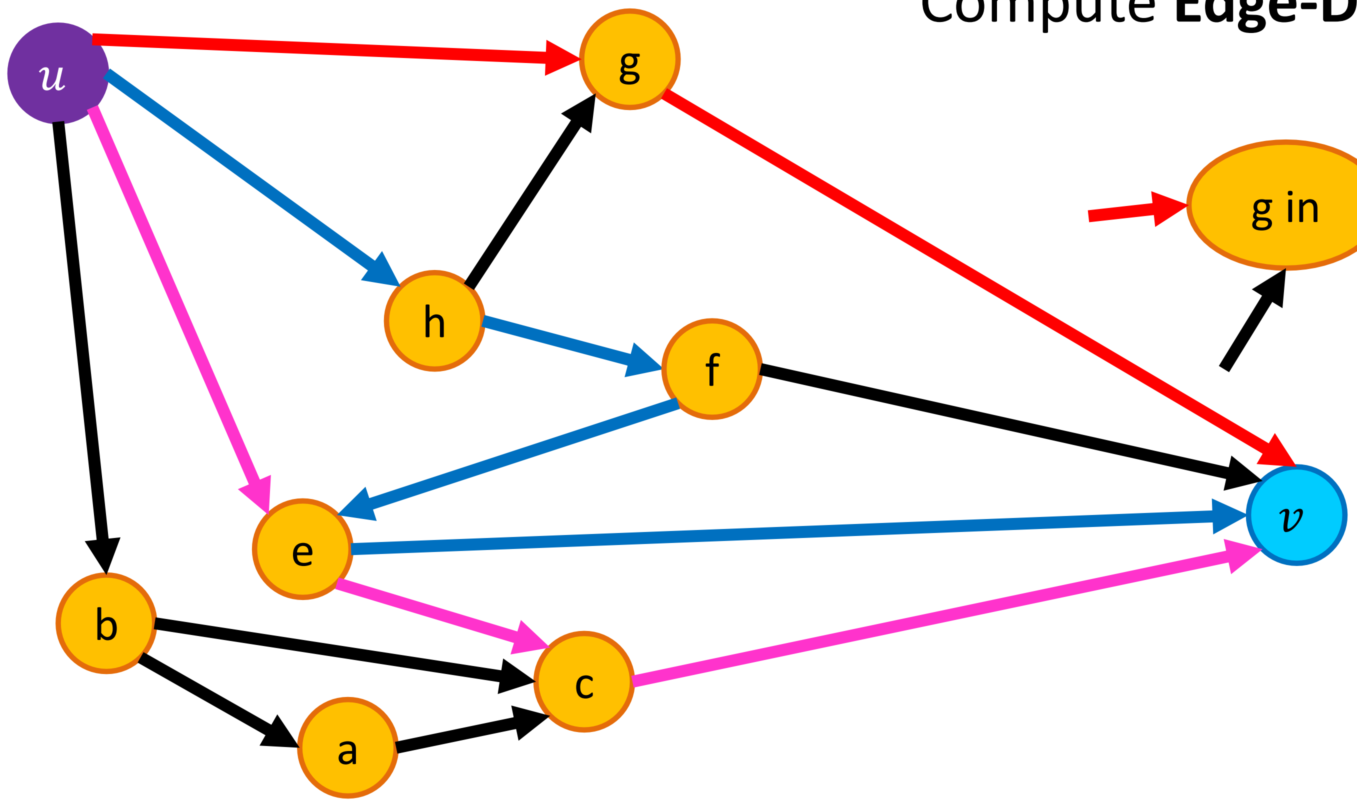
Note these aren't <u>vertex</u>-disjoint paths!

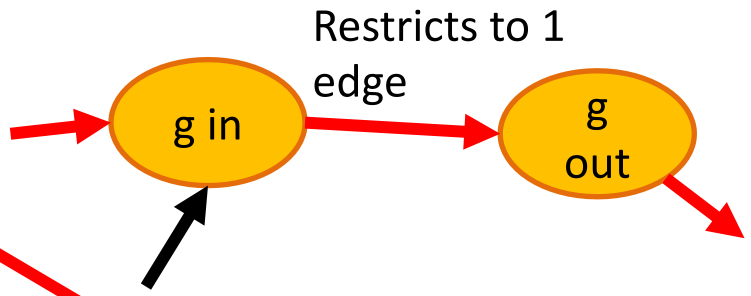
Vertex-Disjoint Paths Algorithm

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





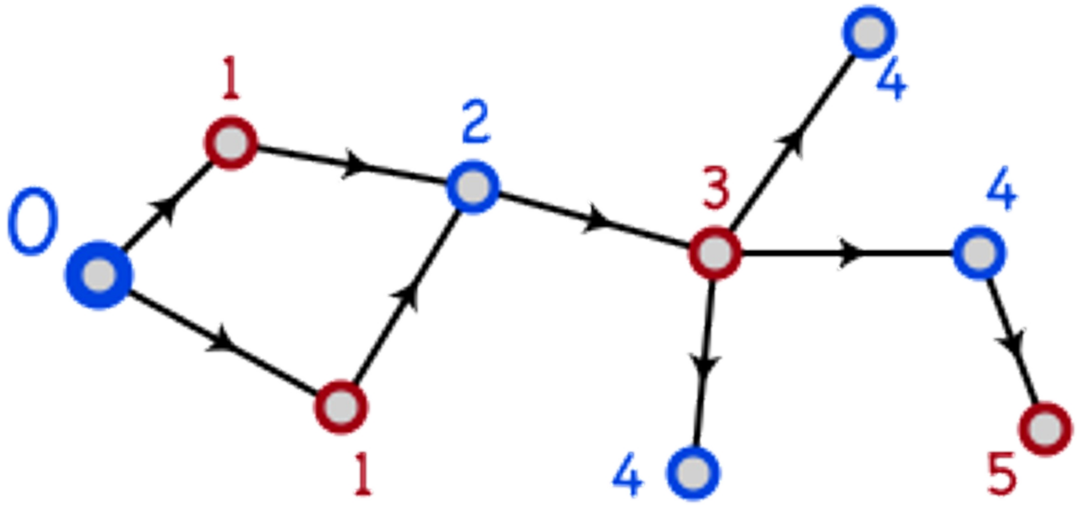
Why does this work?
We need to be able to make a valid argument that it always does.

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

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



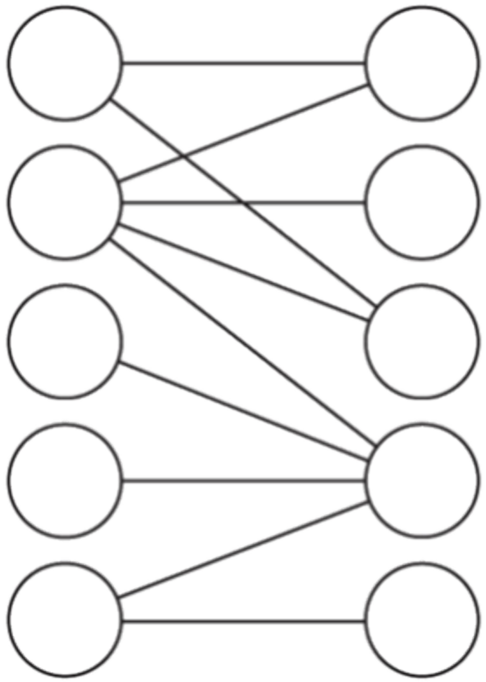
How to determine if G is bipartite?

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

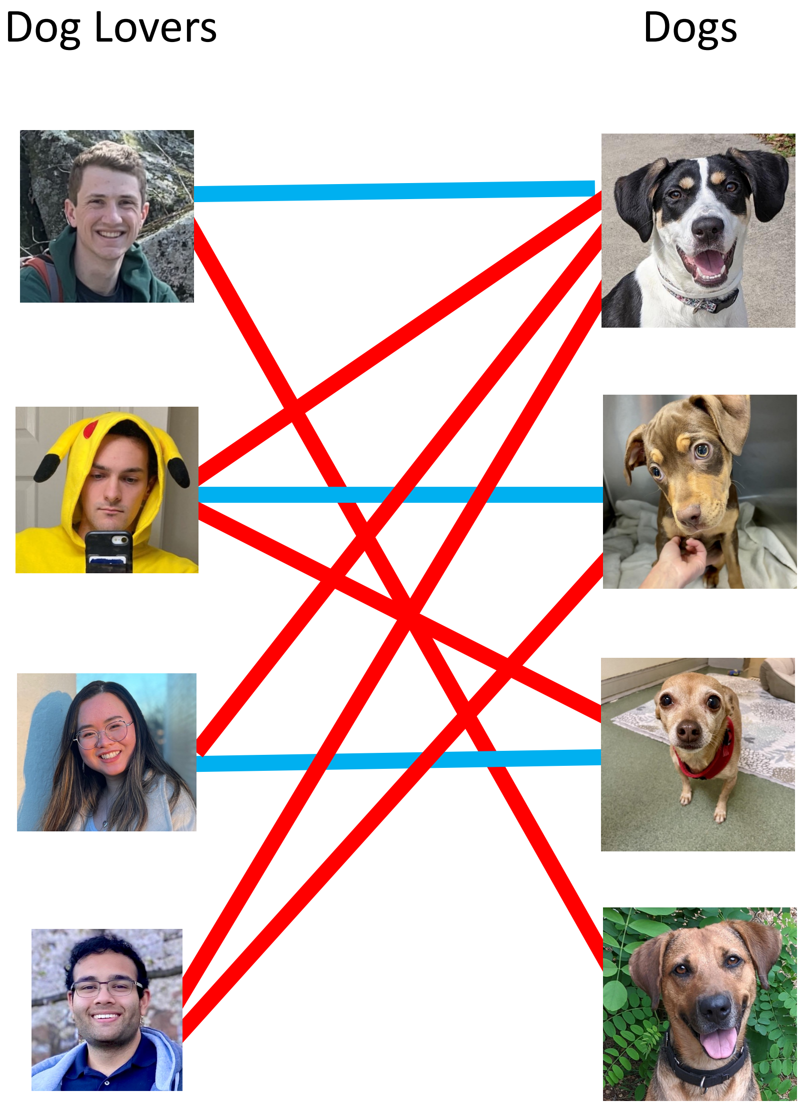
- Pick a starting vertex, color it red
- Color all adjacent nodes blue
  – And all nodes adjacent to that red
  – Etc.
- If you ever need to color an already red-node to be blue (or vice versa), then the graph is not bipartite

- Does this algorithm sound familiar?

Maximum Bipartite Matching

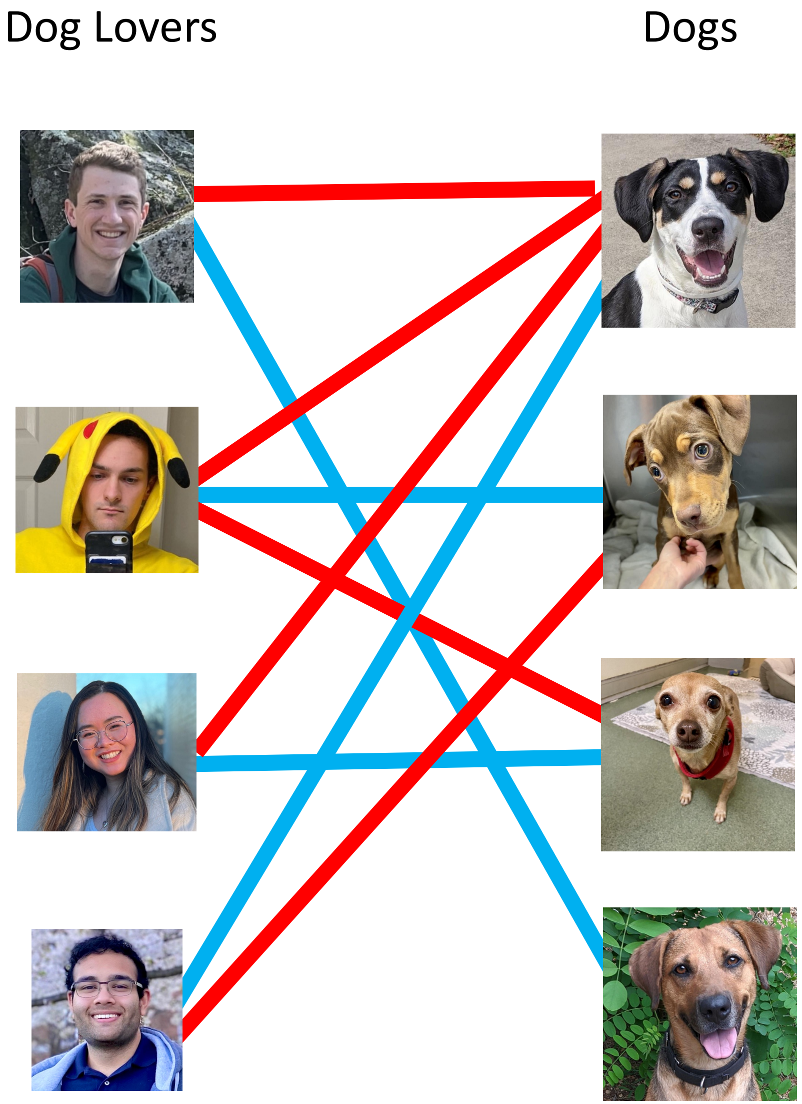
Dog Lovers **Adoptable Dogs**

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.

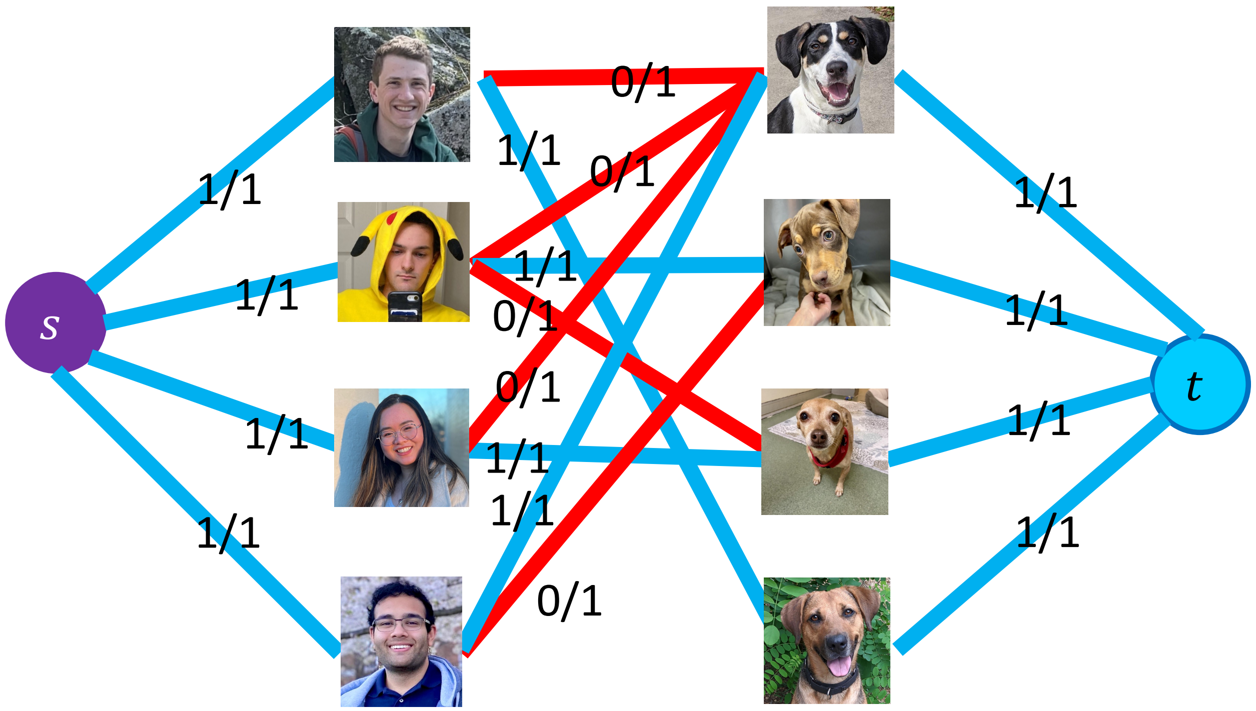
Maximum Bipartite Matching Using Max Flow

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' $\Theta(E \cdot V)$ $|f| \le L$
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"

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)