CS 3100
Data Structures and Algorithms 2

Lecture 20: Network Flow

Co-instructors: Robbie Hott and Ray Pettit

Spring 2024

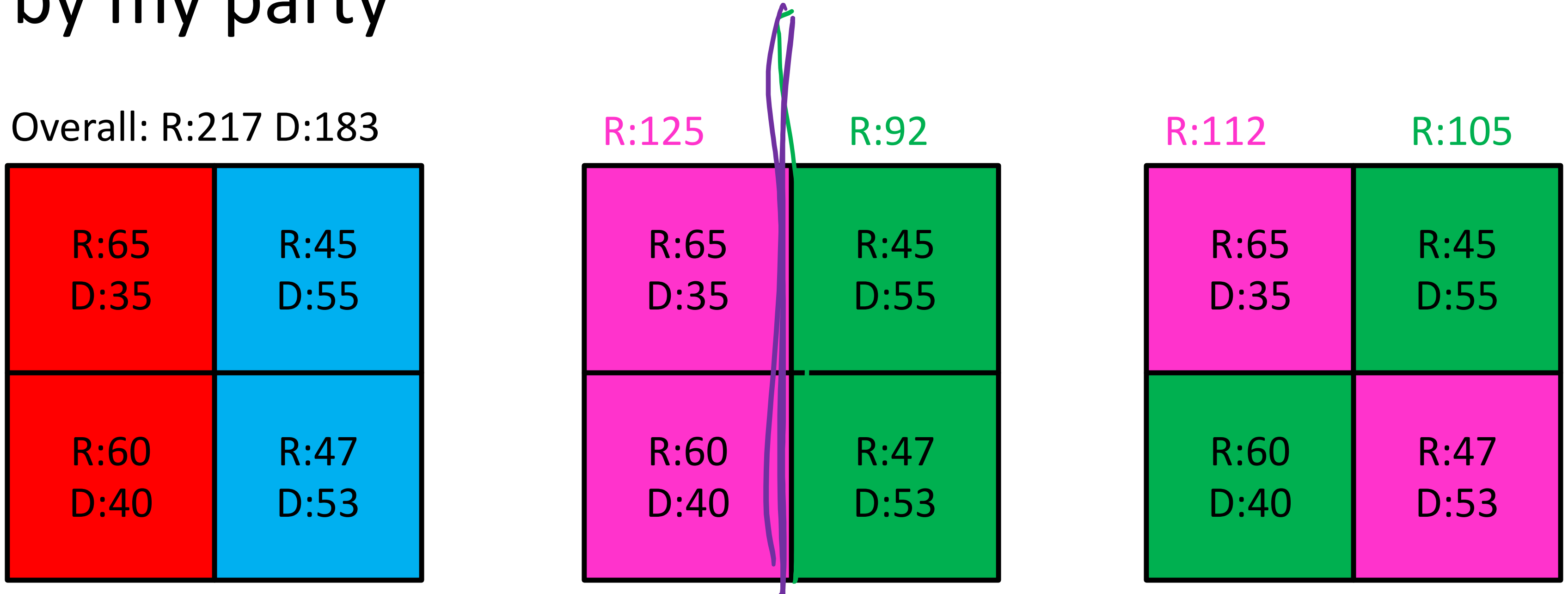
Readings from CLRS 4th Ed:
Chapter 24

Announcements

- PS9 available today
- Quizzes 3-4 next week
  - If you have SDAC, please schedule ASAP
  - More information about quiz security on Tuesday
  - Look for information about a review session early next week
- Office hours updates
  - Prof Hott Office Hours:
    - Back to normal starting Friday
    - Monday: slightly earlier 10-11am

How does it work?

- States are broken into precincts
- All precincts have the same size
- We know voting preferences of each precinct
- Group precincts into districts to maximize the number of districts won by my party



Gerrymandering Problem Statement

- Given:
  - A list of precincts: $p_1, p_2, \ldots, p_n$
  - Each containing m voters
- Output:
  - Districts $D_1, D_2 \subset \{p_1, p_2, \ldots, p_n\}$
  - Where $|D_1| = |D_2|$
  - $R(D_1) > \frac{mn}{4}$ and $R(D_2) > \frac{mn}{4}$
    - $R(D_i)$ gives number of “Regular Party” voters in D_i
    - $R(D_i) > \frac{\text{mn}}{4}$ means D_i is majority “Regular Party”
  - “failure” if no such solution is possible

Valid Gerrymandering!

$m \cdot \frac{n}{2} \cdot \frac{1}{2}$

Consider the last precinct

After assigning the first $n-1$ precincts $p_1, p_2, \dots, p_{n-1}$

D_1
k precincts
x voters for R

D_2 $(n-1)-k$
$n-k-1$ precincts
y voters for R

$n-1, k, x, y$

p_n

If we assign p_n to D_1

D_1
$k+1$ precincts
$x+R(p_n)$ voters for R

Valid gerrymandering if:

$k+1=\frac{n}{2},$

$x+R(p_n), y > \frac{mn}{4}$

If we assign p_n to D_2

D_2
$n-k$ precincts
$y+R(p_n)$ voters for R

Valid gerrymandering if:

$\mathrm{n}-k=\frac{n}{2},$

$x, y+R(p_n) > \frac{mn}{4}$

World One

D_1
$k+1$ precincts
$x+R(p_n)$ voters for R

D_2
$n-k-1$ precincts
y voters for R

World Two

D_1
k precincts
x voters for R

D_2
$n-k$ precincts
$y+R(p_n)$ voters for R

Define Recursive Structure

$S(j, k, x, y) =$ True if from among the first $\boldsymbol{j}$ precincts:
$\boldsymbol{k}$ are assigned to D_1
exactly $\boldsymbol{x}$ vote for R in D_1
exactly $\boldsymbol{y}$ vote for R in D_2

$n \times n \times mn \times mn$

4D Dynamic Programming!!!

True here means that this is a valid state of the world; not a valid Gerrymander!

Two ways to satisfy $S(j,k,x,y)$:

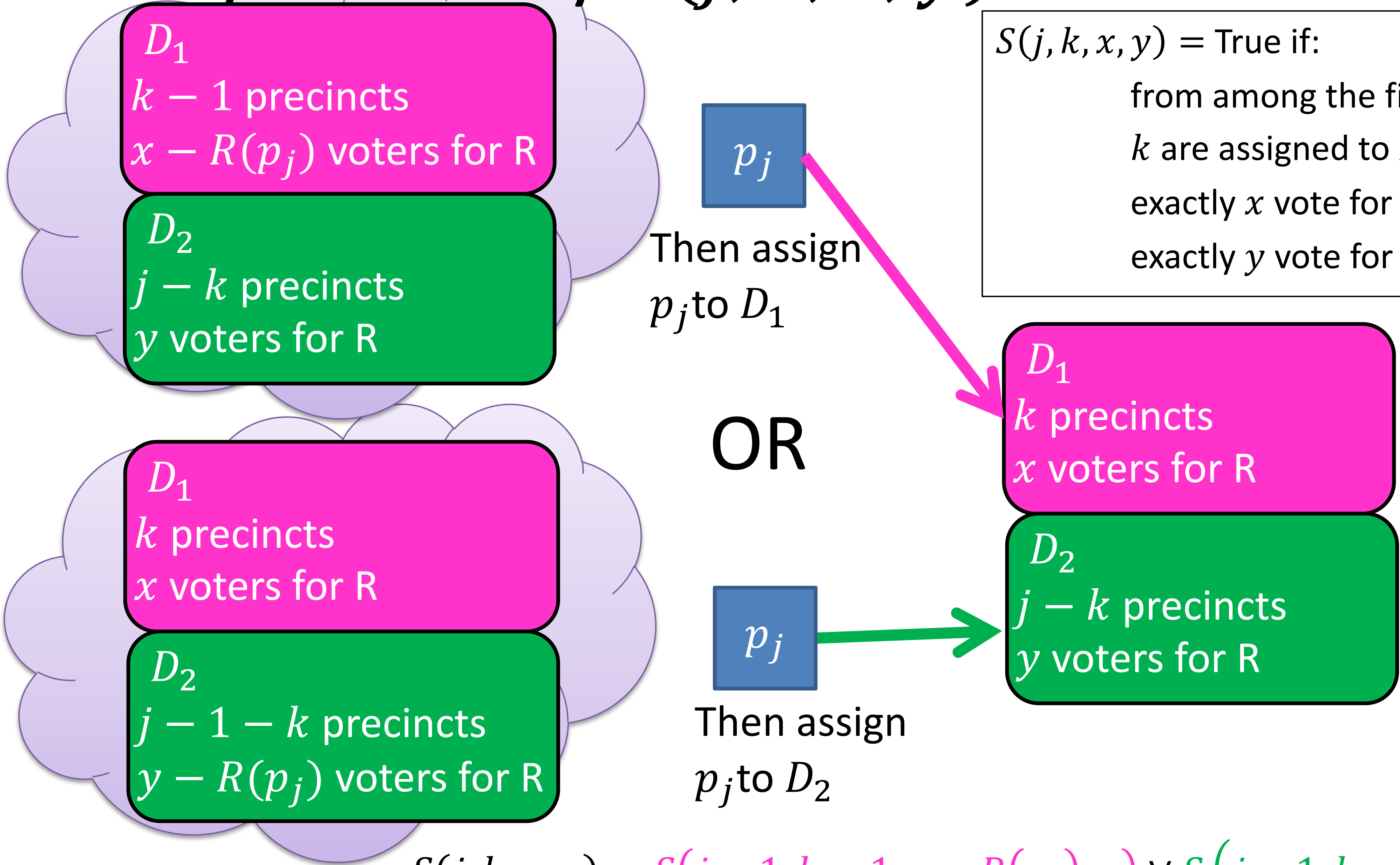
$S(j,k,x,y) =$ True if:

from among the first j precincts

k are assigned to D_1

exactly x vote for R in D_1

exactly y vote for R in D_2



$$S(j,k,x,y) = S\left(j-1, k-1, x-R(p_j), y\right) \vee S\left(j-1, k, x, y-R(p_j)\right)$$

Final Algorithm

$$S(j,k,x,y) = S(j-1,k-1,x-R(p_j),y) \vee S\left(j-1,k,x,y-R(p_j)\right)$$

Initialize $S(0,0,0,0) = \text{True}$

for $j = 1,\ldots,n$:

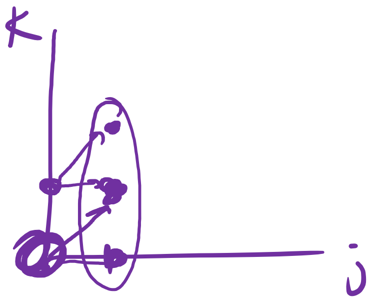
  for $k = 1,\ldots,\min(j,\frac{n}{2})$:

    for $x = 0,\ldots,jm$:

      for $y = 0,\ldots,jm$:

        $S(j,k,x,y) =$

$$S(j-1,k-1,x-R(p_j),y) \vee S\left(j-1,k,x,y-R(p_j)\right)$$

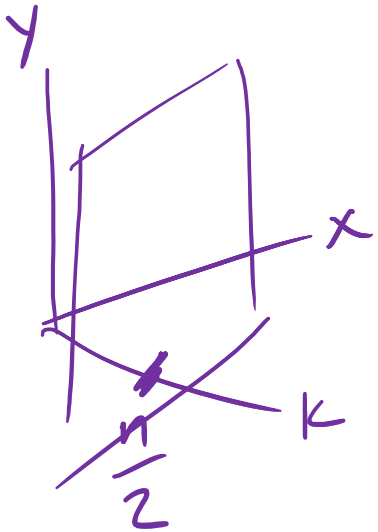


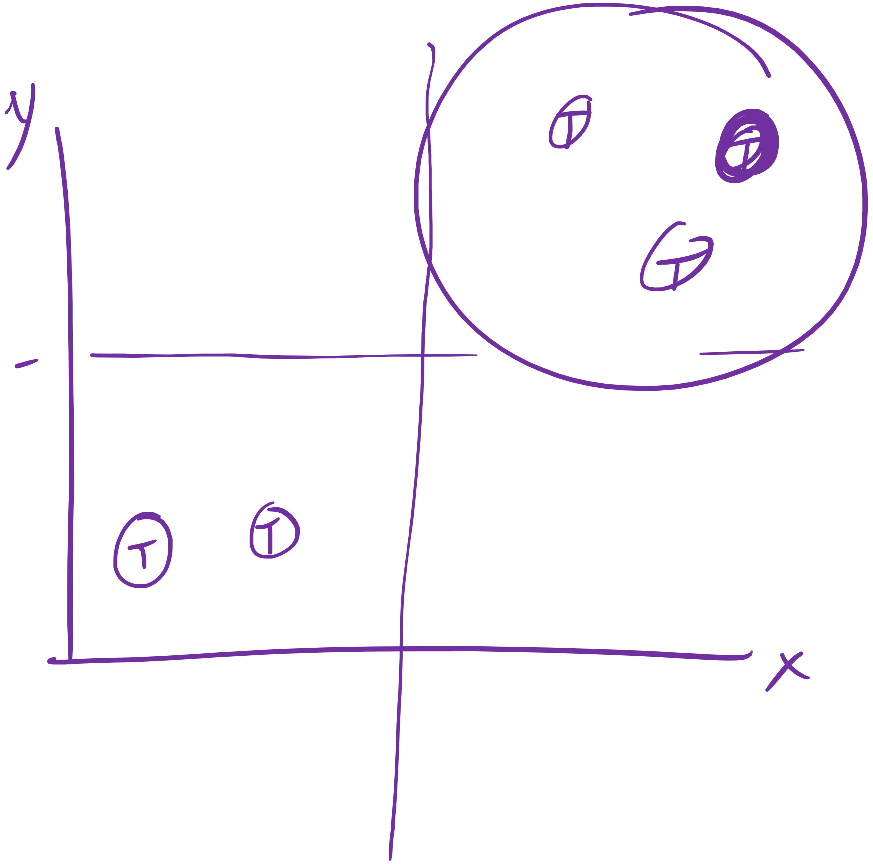
$S(j,k,x,y) = \text{True}$ if:

- from among the first j precincts
- k are assigned to D_1
- exactly x vote for R in D_1
- exactly y vote for R in D_2

Search for True entry at $S(n,\frac{n}{2},>\frac{mn}{4},>\frac{mn}{4})$

Where is Solution?





Run Time

$$S(j,k,x,y) = S(j-1,k-1,x-R(p_j),y) \vee S\left(j-1,k,x,y-R(p_j)\right)$$

Initialize $S(0,0,0,0) = \text{True}$

n for $j = 1, \dots, n$:

$\frac{n}{2}$ for $k = 1, \dots, \min(j, \frac{n}{2})$:

nm for $x = 0, \dots, jm$:

nm for $y = 0, \dots, jm$:

$S(j,k,x,y) =$

$S(j-1,k-1,x-R(p_j),y) \vee S\left(j-1,k,x,y-R(p_j)\right)$

Search for True entry at $S(n, \frac{n}{2}, > \frac{mn}{4}, > \frac{mn}{4})$

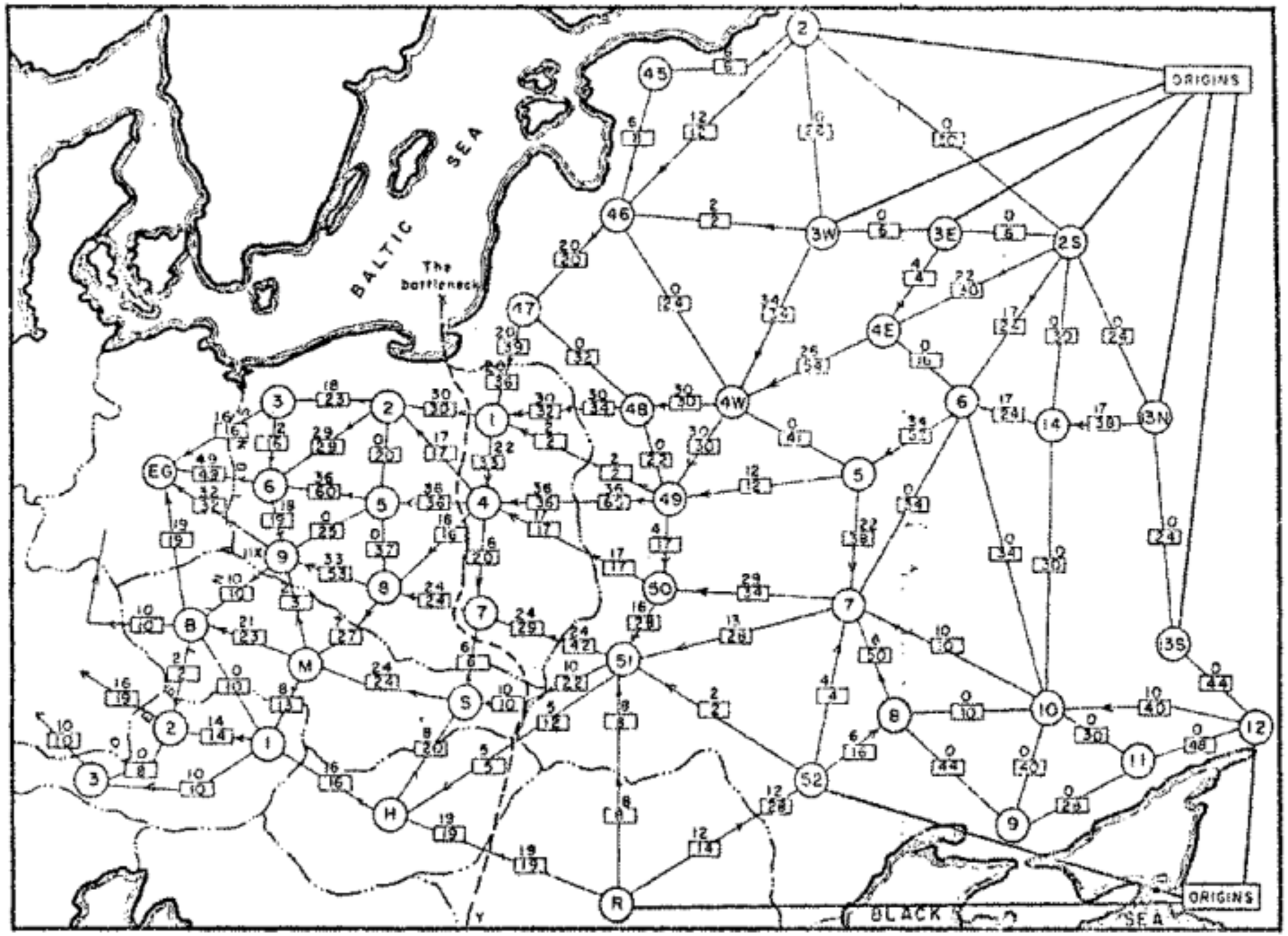
$\Theta(n^4 m^2)$

$\Theta(n^4 m^2)$

- Input: list of precincts (size n), number of voters (integer m)
- Runtime depends on the *value* of m, not *size* of m
  - Run time is exponential in *size* of input
  - Input size is $n + |m| = n + \log m$
- Note: Gerrymandering is NP-Complete

Network Flow

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



Railway map of Western USSR, 1955

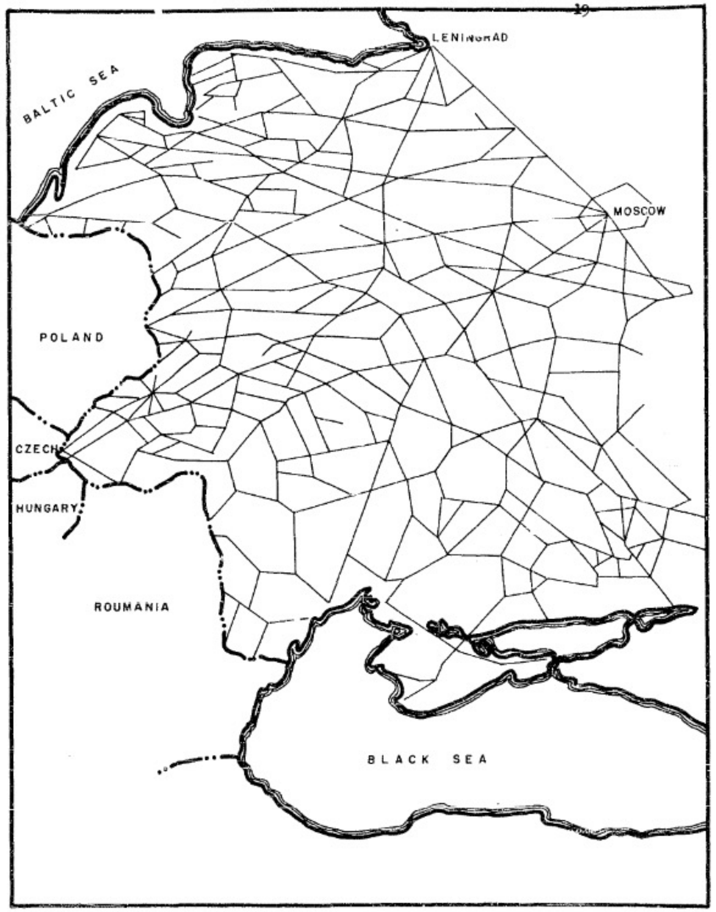
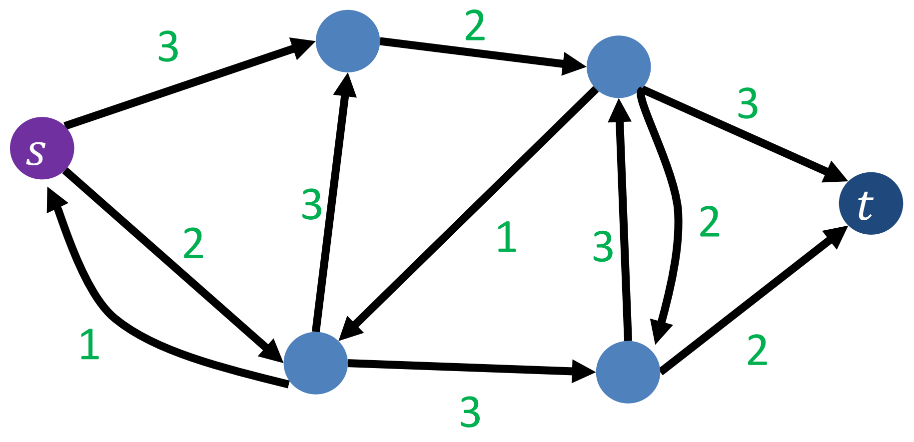


Fig. I—The railway system of western Russia

Flow Networks

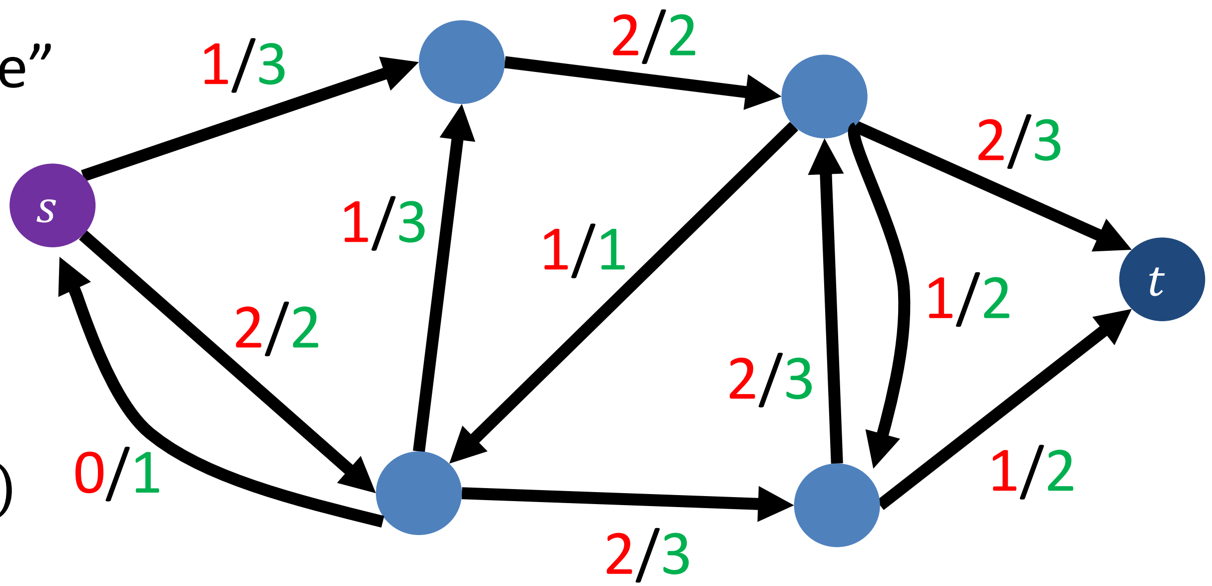
Graph $G = (V, E)$
Source node $s \in V$
Sink node $t \in V$
Edge capacities $c(e) \in \mathbb{R}^+$



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?

Network Flow

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



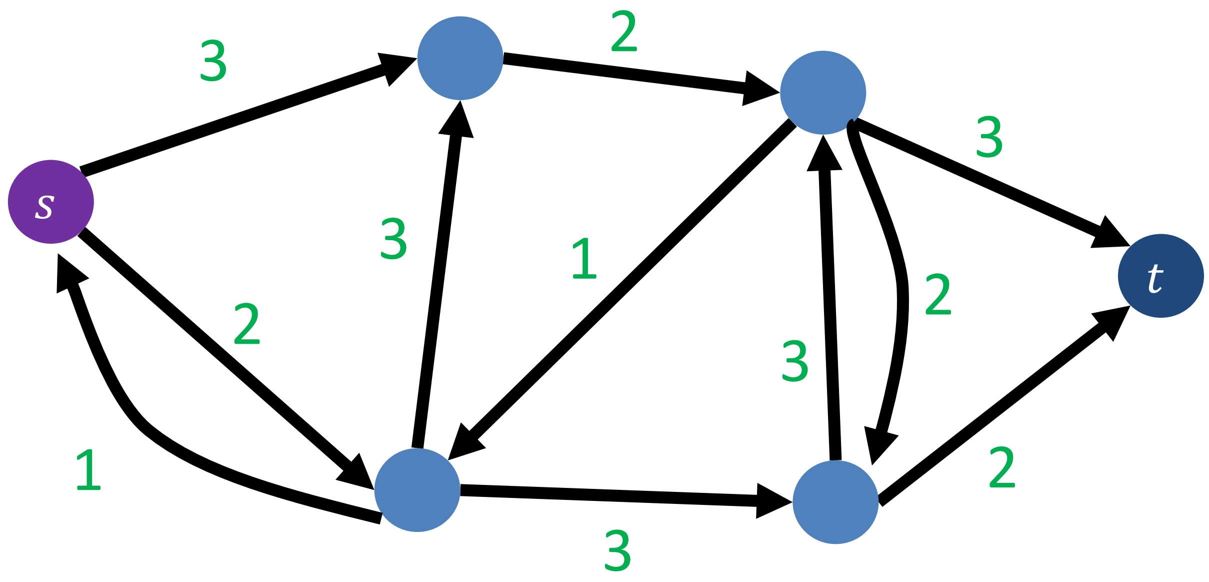
flow / capacity

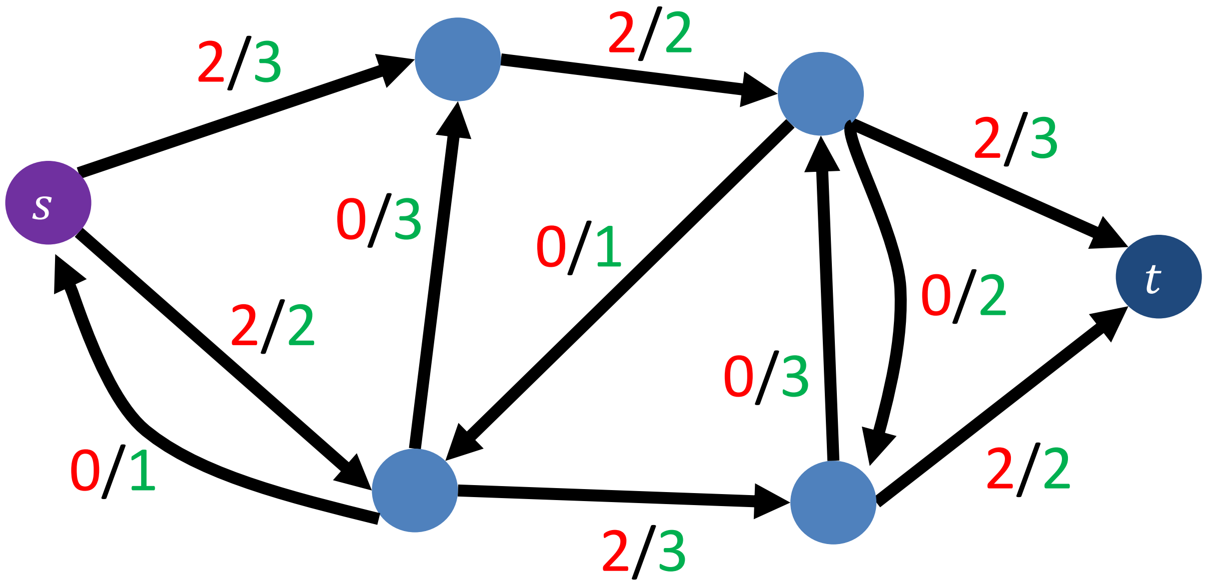
3 in this example

Maximum Flow Problem

- Of all valid flows through the graph, find the one that maximizes:

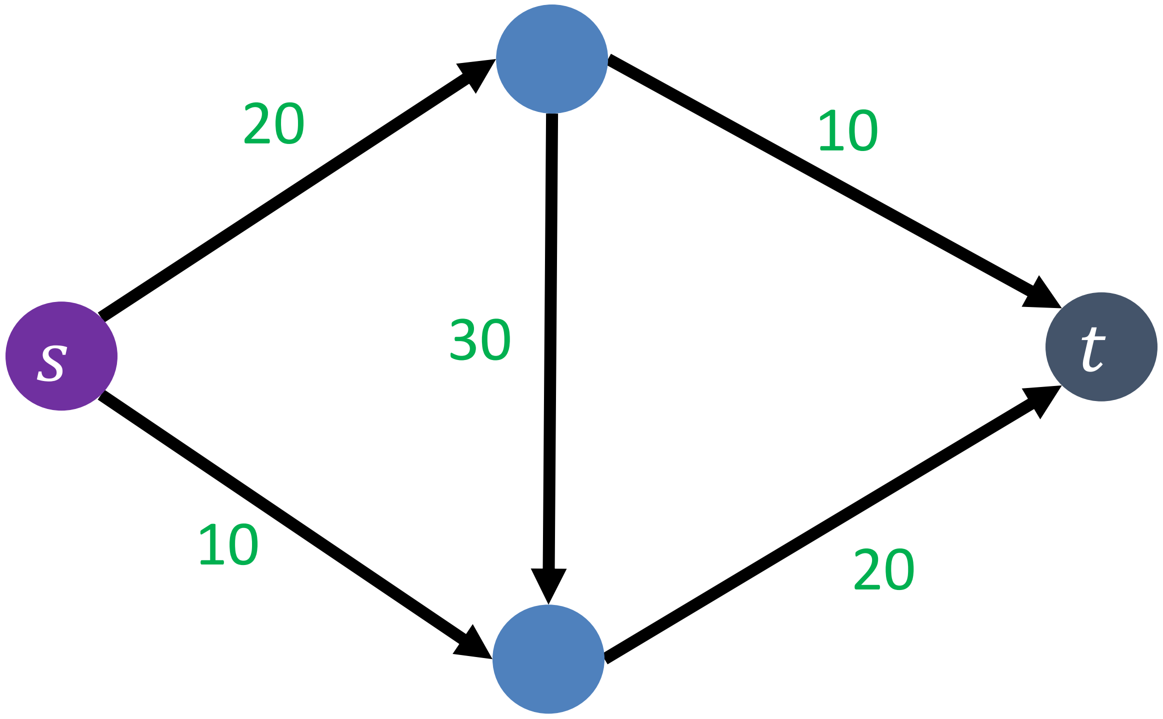
$$|f| = \text{outflow}(s) - \text{inflow}(s)$$





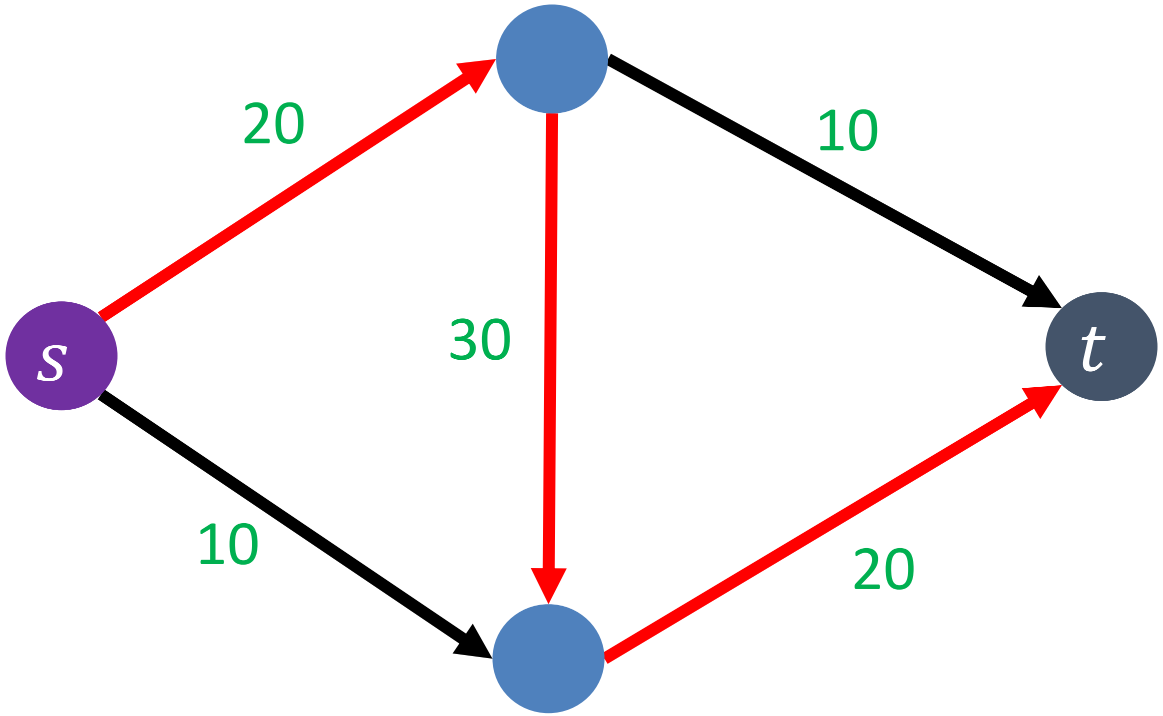
Greedy Approach

Greedy choice: saturate <u>highest</u> capacity path first



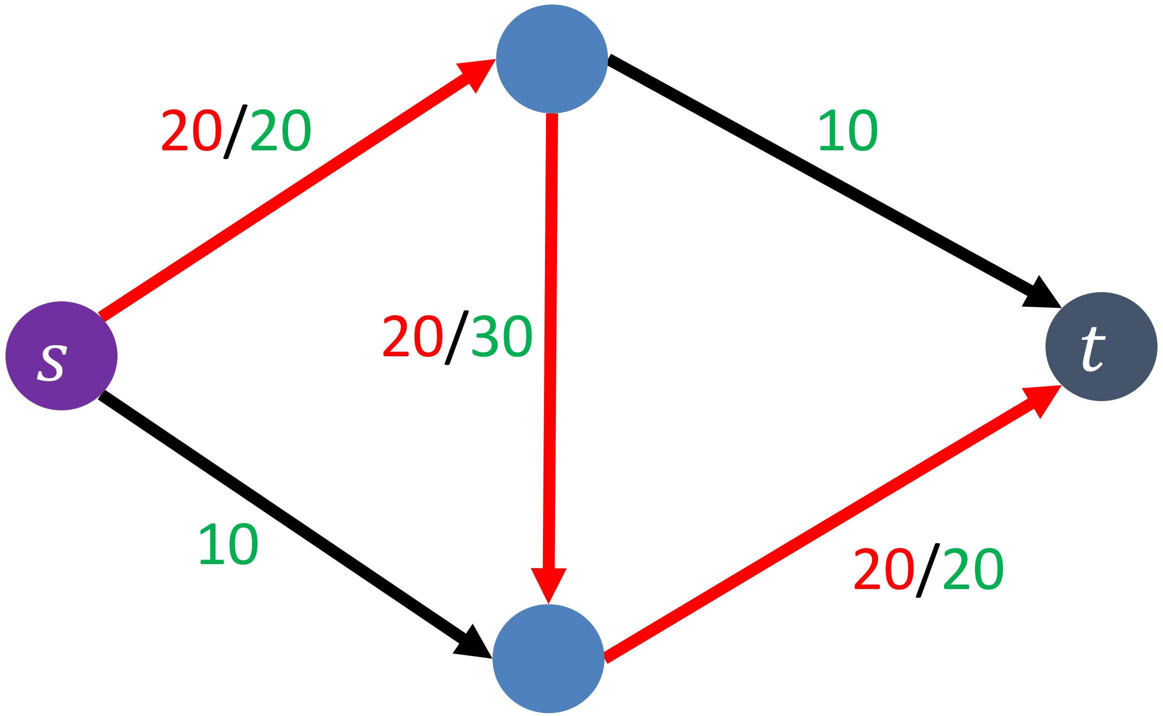
Greedy Approach

Greedy choice: saturate <u>highest</u> capacity path first



Greedy Approach

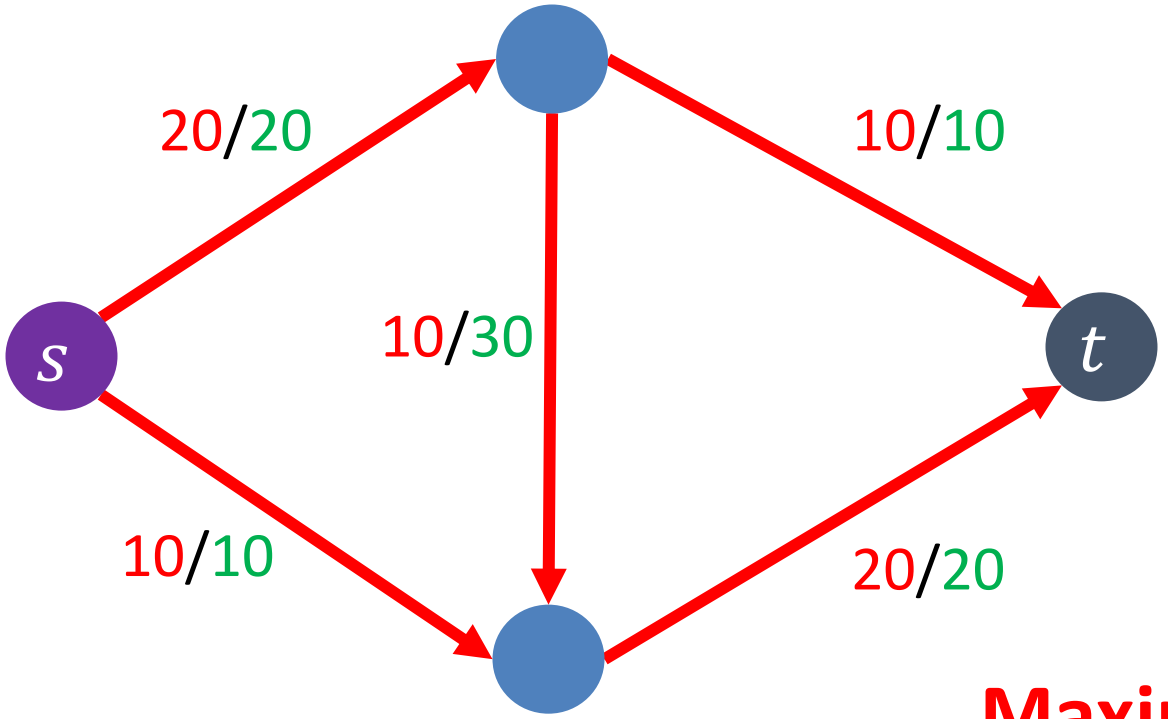
Greedy choice: saturate highest capacity path first



Flow: 20

Greedy Approach

Greedy choice: saturate highest capacity path first



Maximum Flow: 30

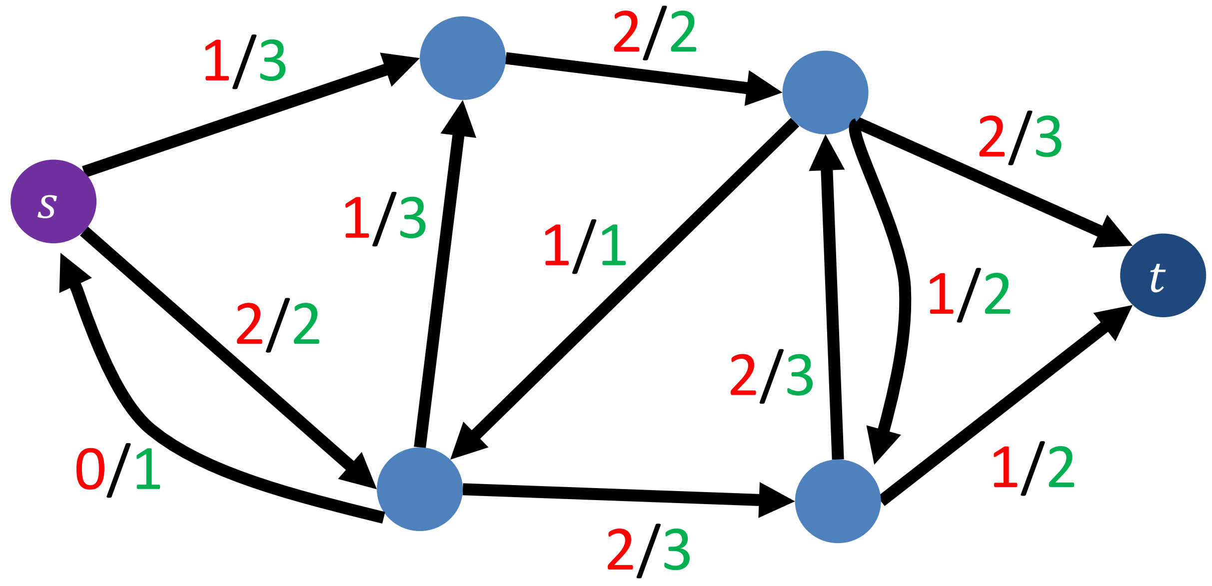
Observe: highest capacity path is not saturated in optimal solution

Residual Graphs

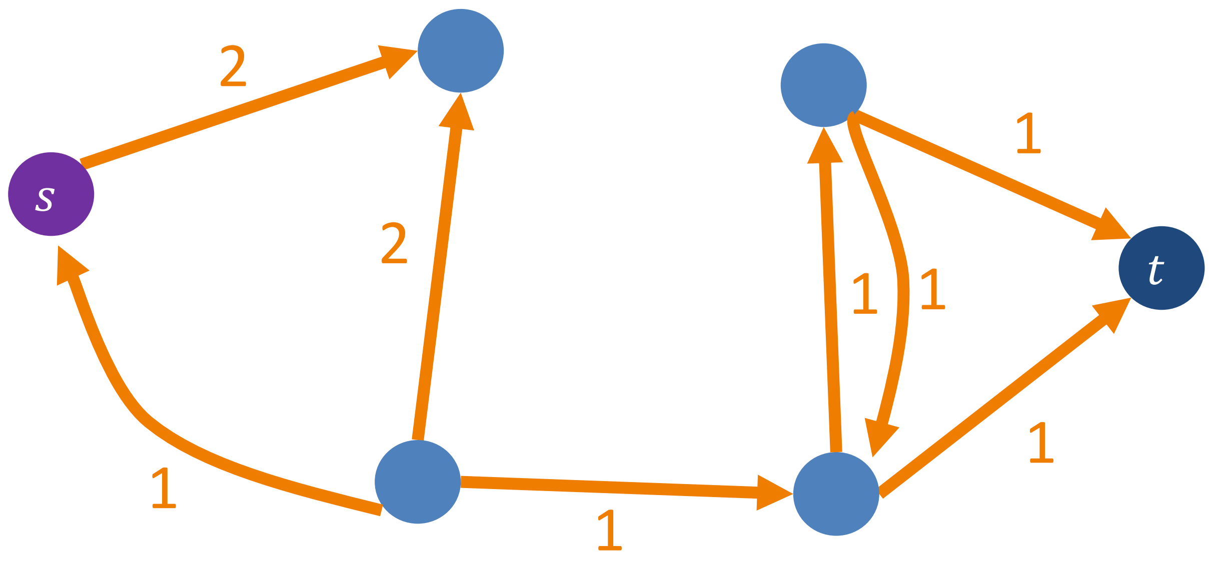
Given a flow f in graph G, the residual graph G_f models additional flow that is possible

- 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



Flow f in G

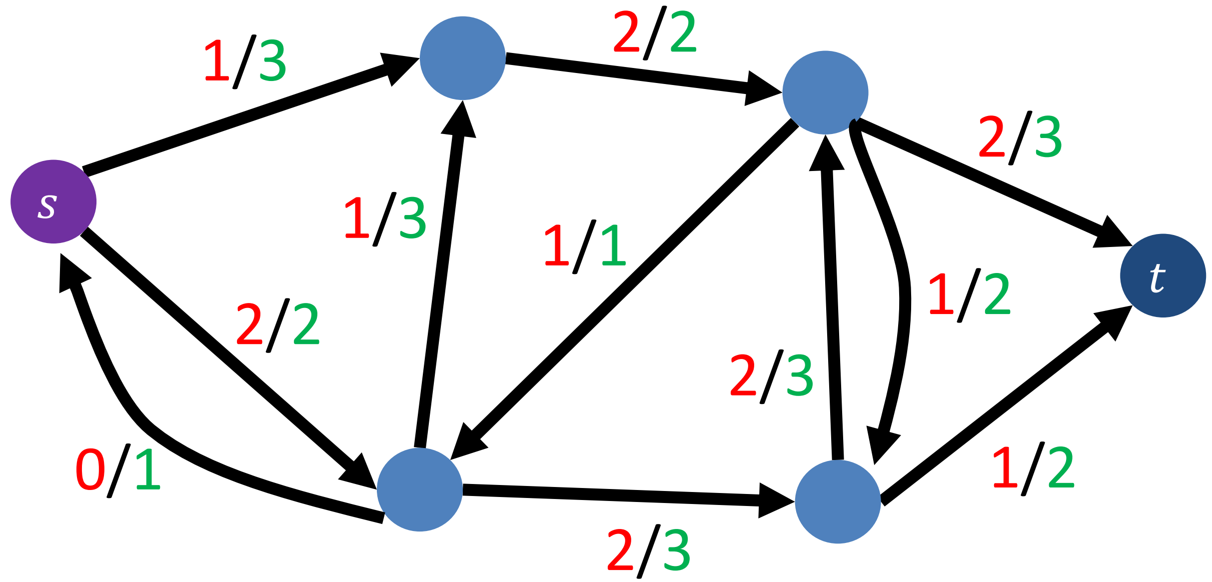


Residual graph G_f

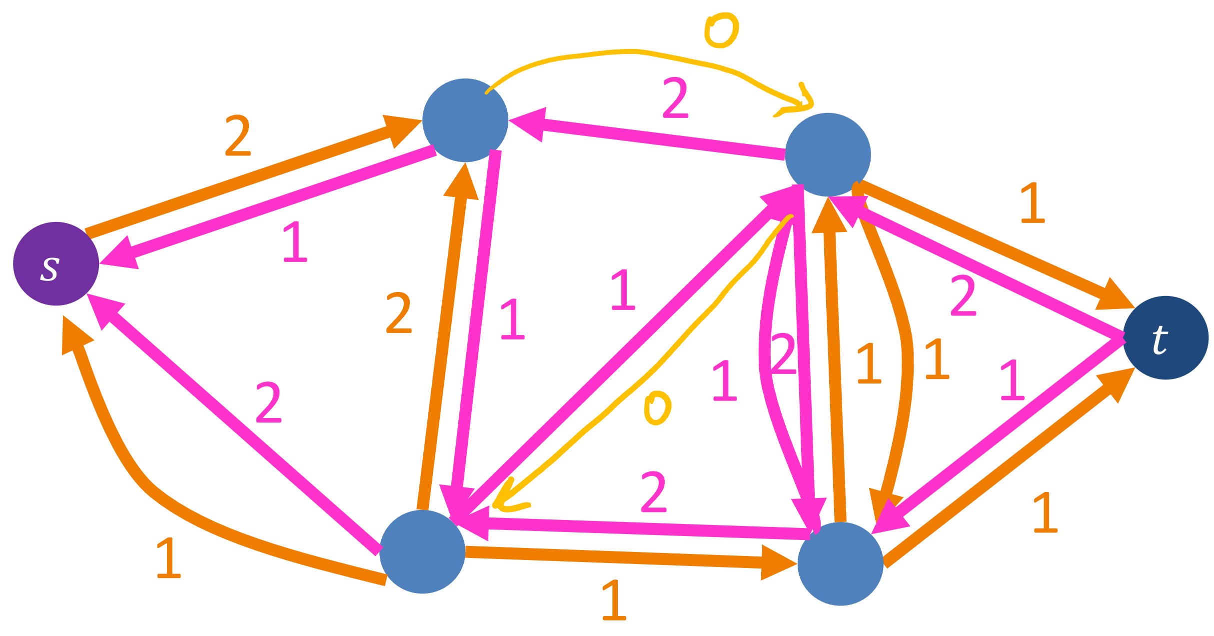
Residual Graphs

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

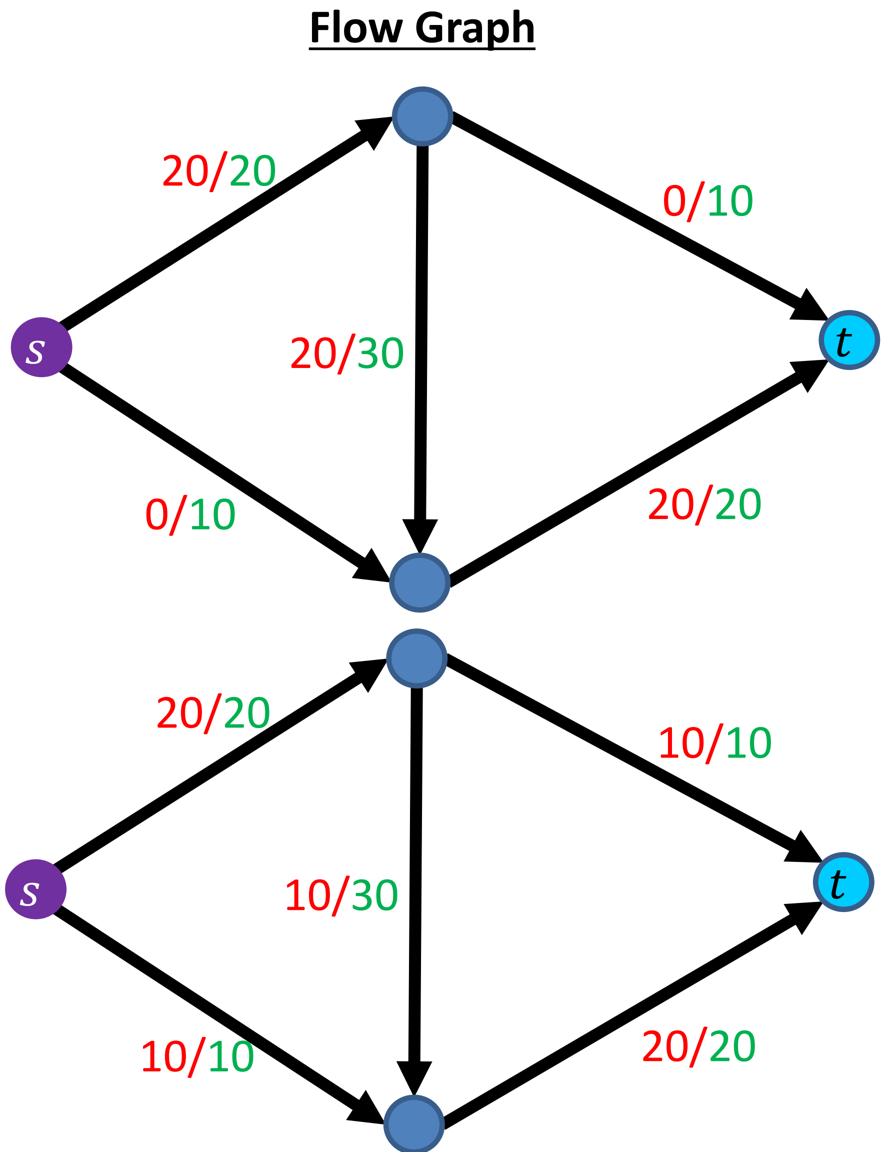


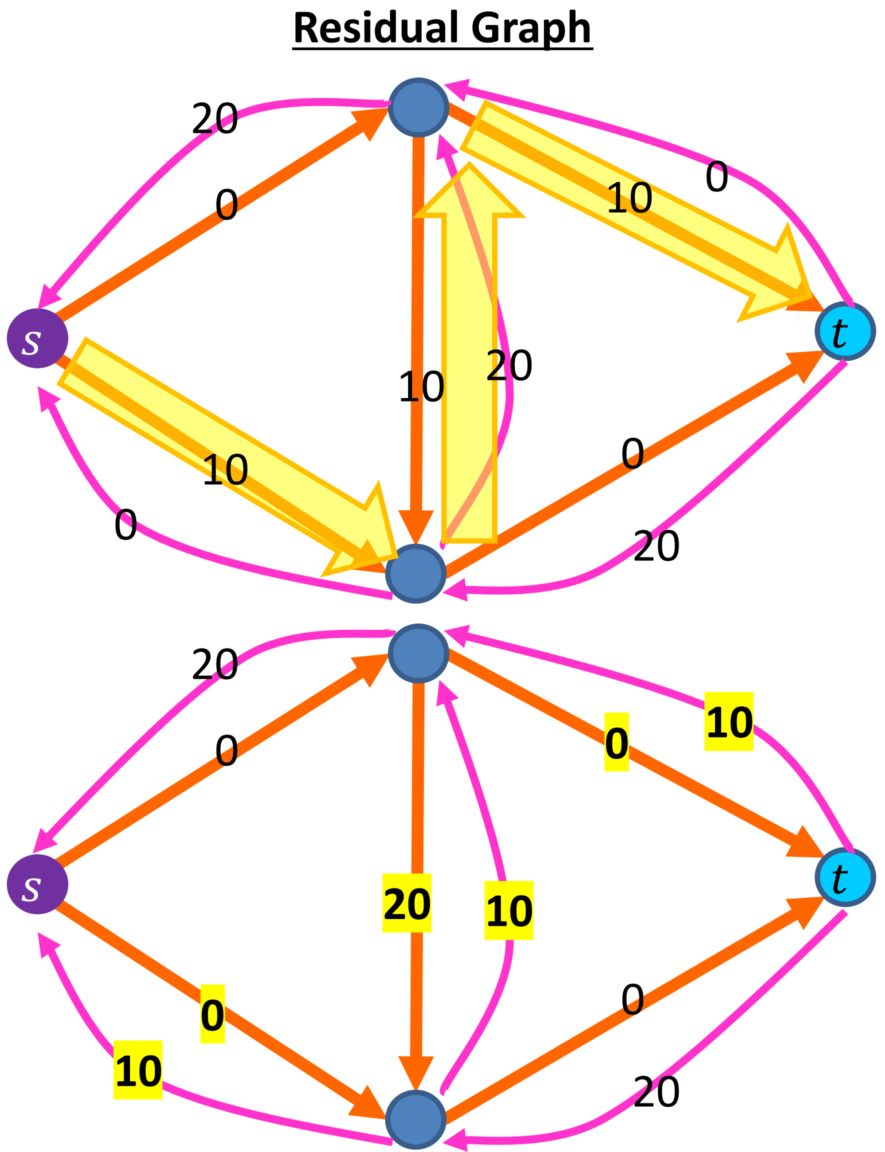
Flow f in G



Residual graph G_f

Residual Graphs Example

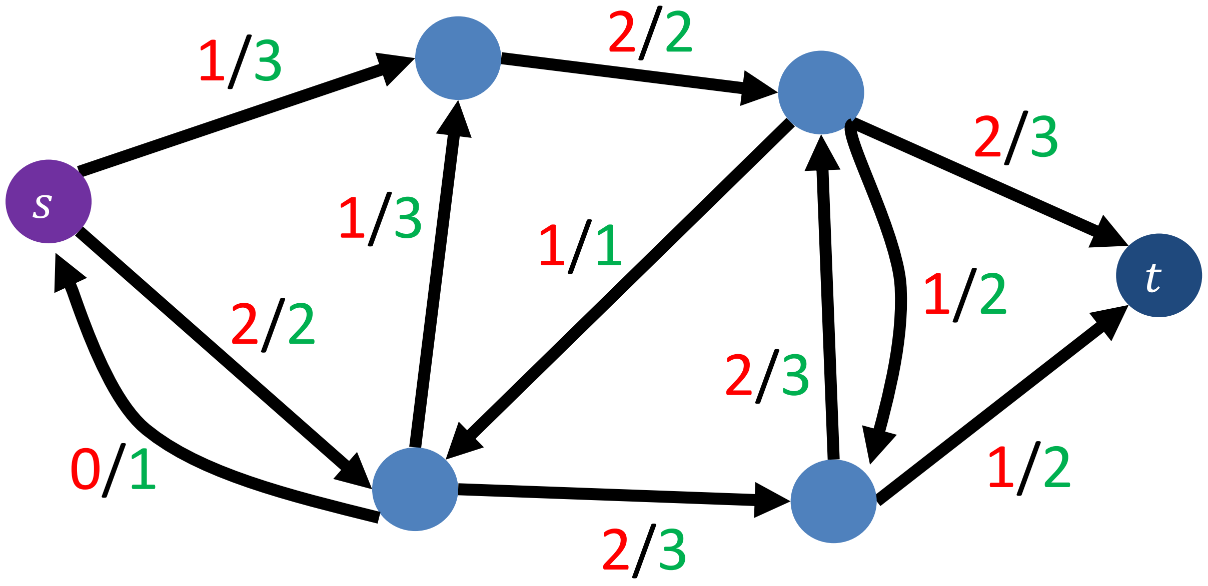




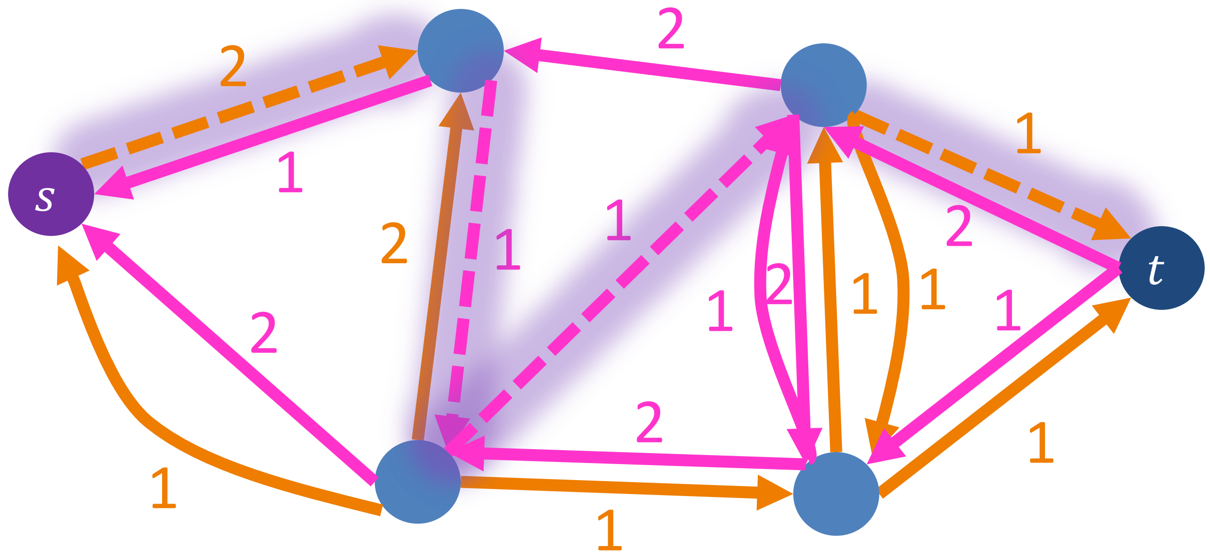
Residual Graphs

Consider a path from $s \to t$ in G_f using only edges with positive (non-zero) weight

Consider the minimum-weight edge e along the path: we can increase the flow by $w(e)$



Flow f in G



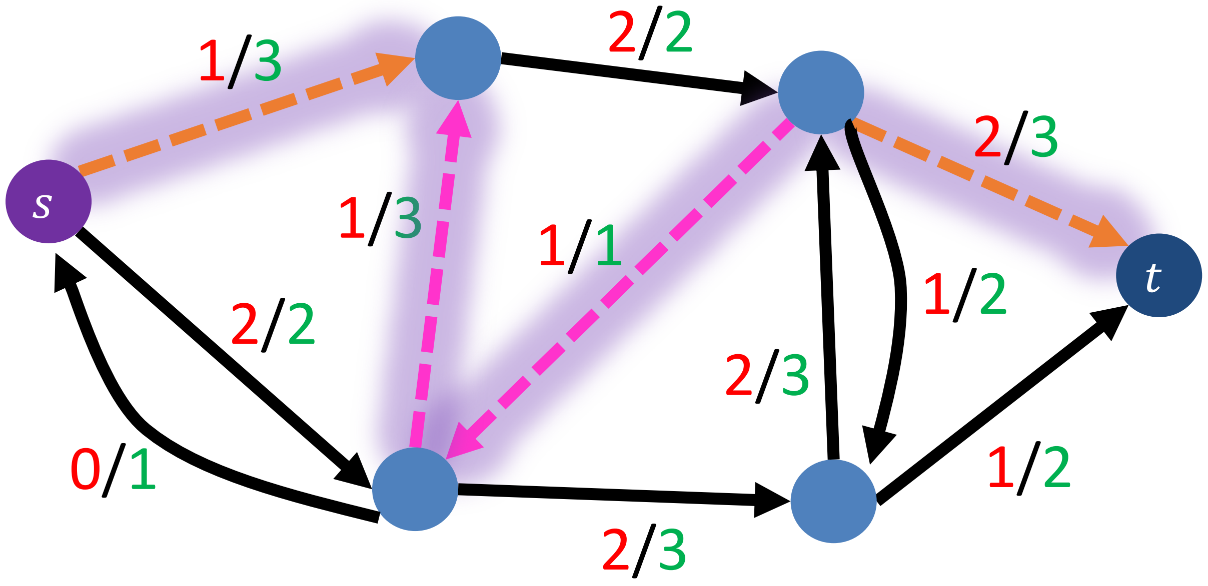
Residual graph G_f

Residual Graphs

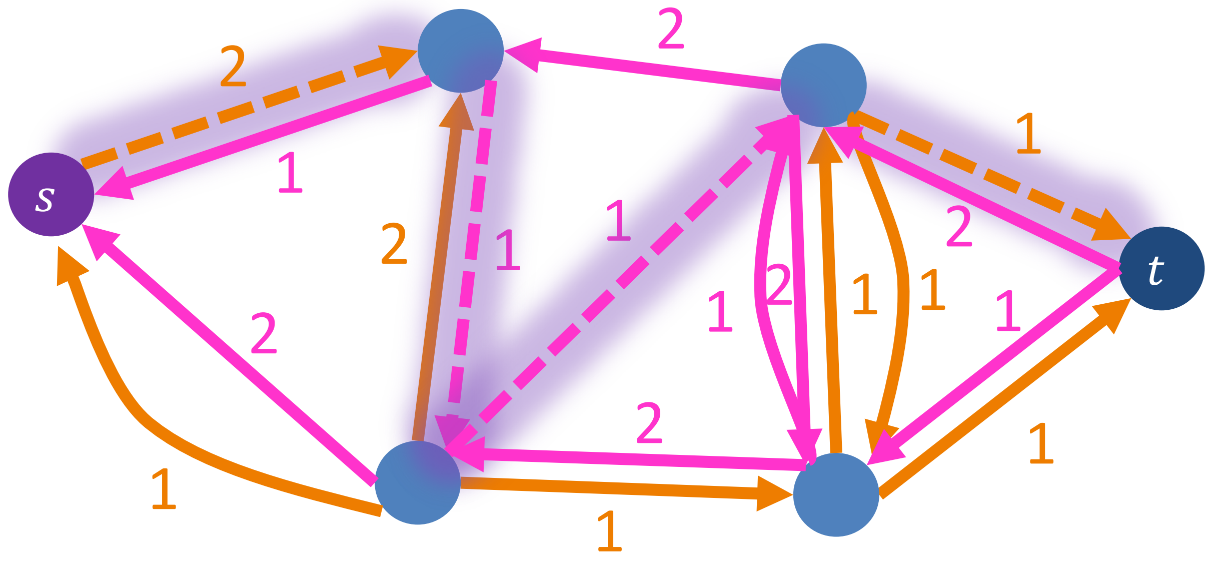
Consider a path from $s \to t$ in G_f using only edges with positive (non-zero) weight

Consider the minimum-weight edge e along the path: we can increase the flow by $w(e)$

- Send $w(e)$ flow along all forward edges (these have at least $w(e)$ capacity)
- Remove $w(e)$ flow along all backward edges (these contain at least $w(e)$ units of flow)



Flow f in G



Residual graph G_f

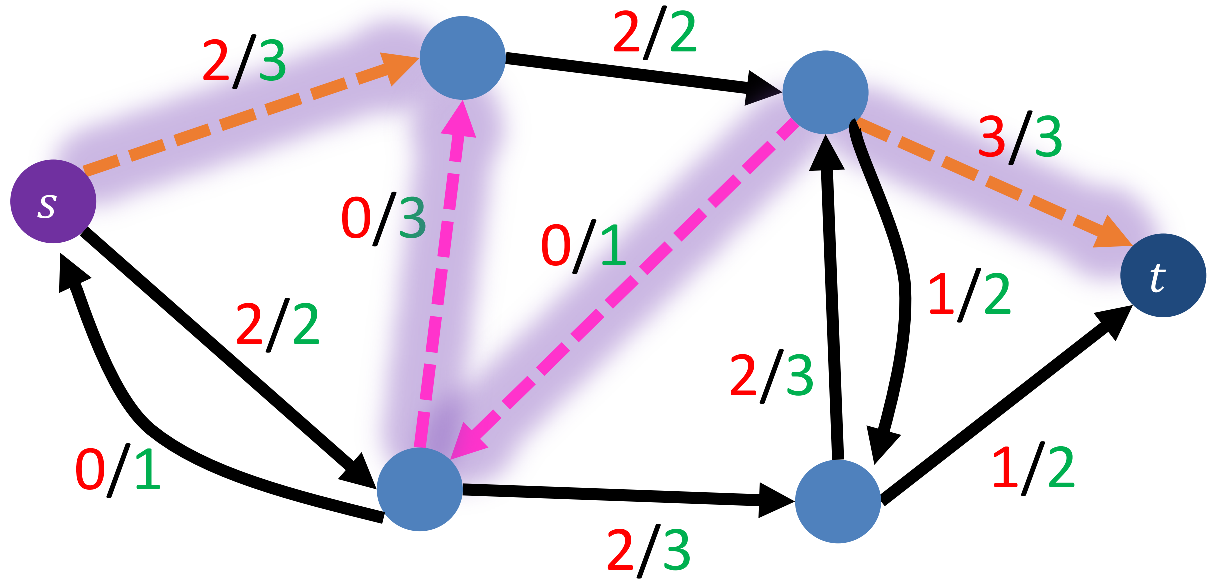
Residual Graphs

Consider a path from $s \rightarrow t$ in G_f using only edges with positive (non-zero) weight

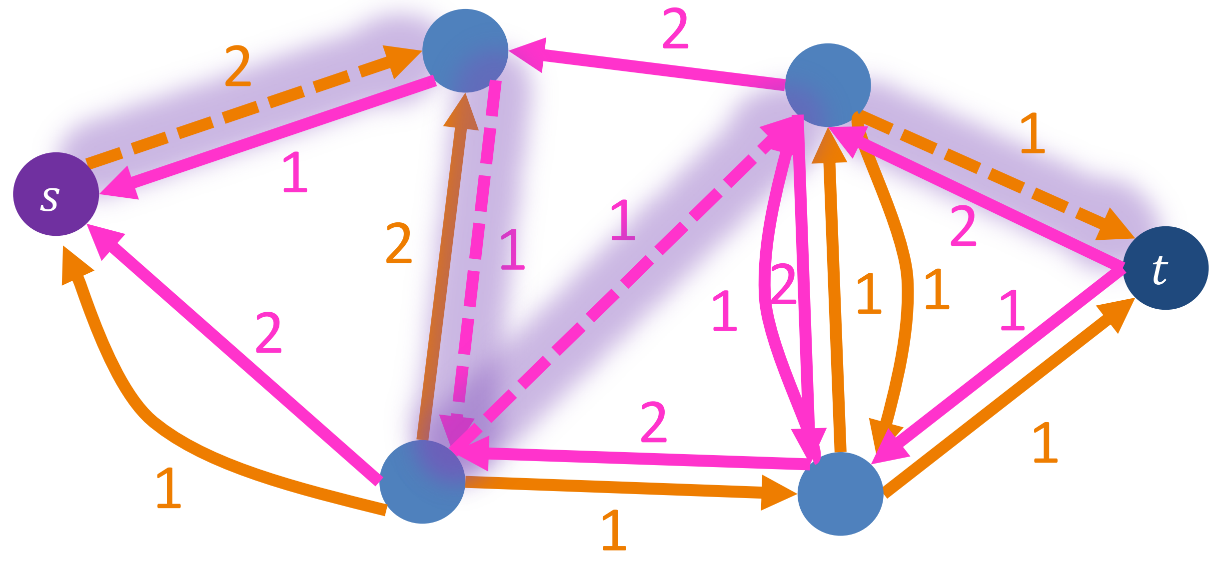
Consider the minimum-weight edge e along the path: we can increase the flow by $w(e)$

- Send $w(e)$ flow along all forward edges (these have at least $w(e)$ capacity)
- Remove $w(e)$ flow along all backward edges (these contain at least $w(e)$ units of flow)

Observe: Flow has increased by $w(e)$



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)

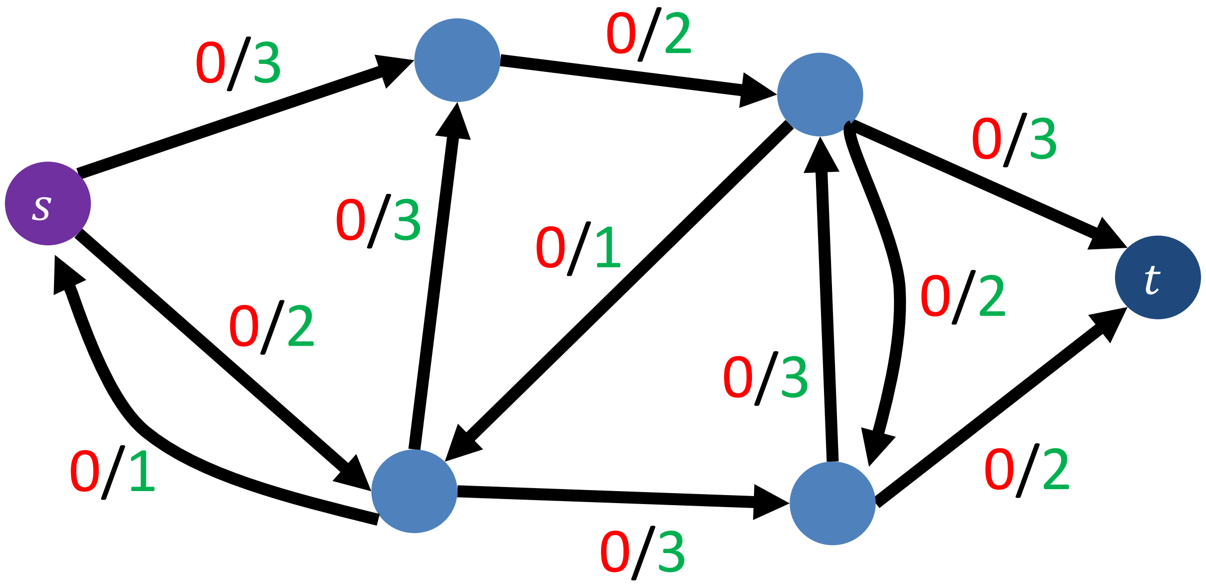
Ford-Fulkerson max-flow algorithm:

- 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

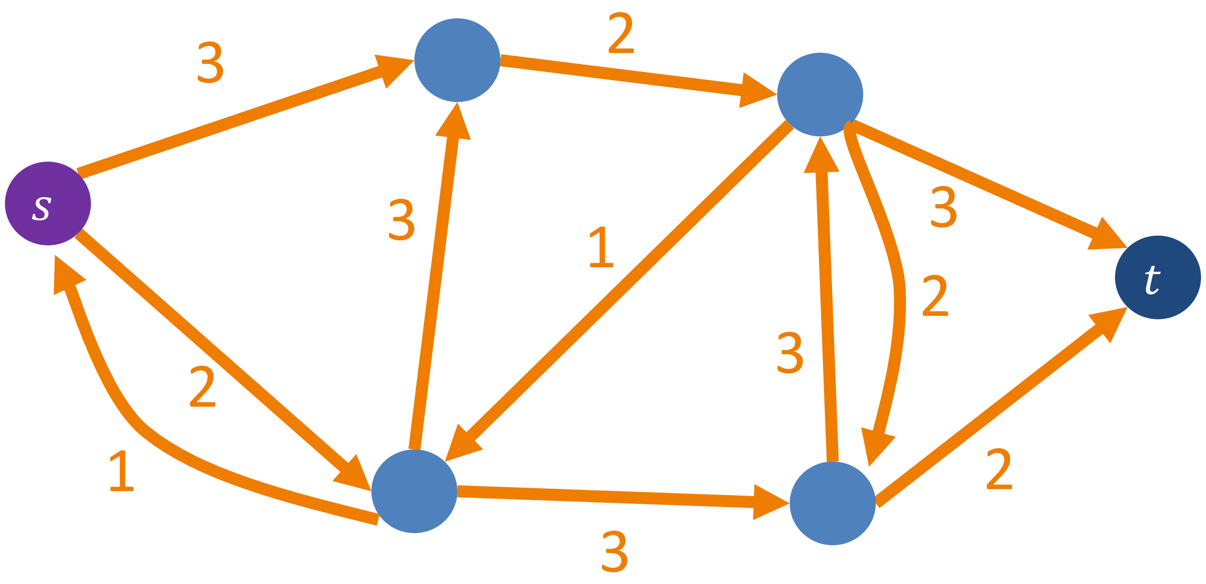
    ($c_f(e)$ is the weight of edge e in the residual network G_f)
  - Add c units of flow to G based on the augmenting path p
  - Update the residual network G_f for the updated flow

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

Ford-Fulkerson Example

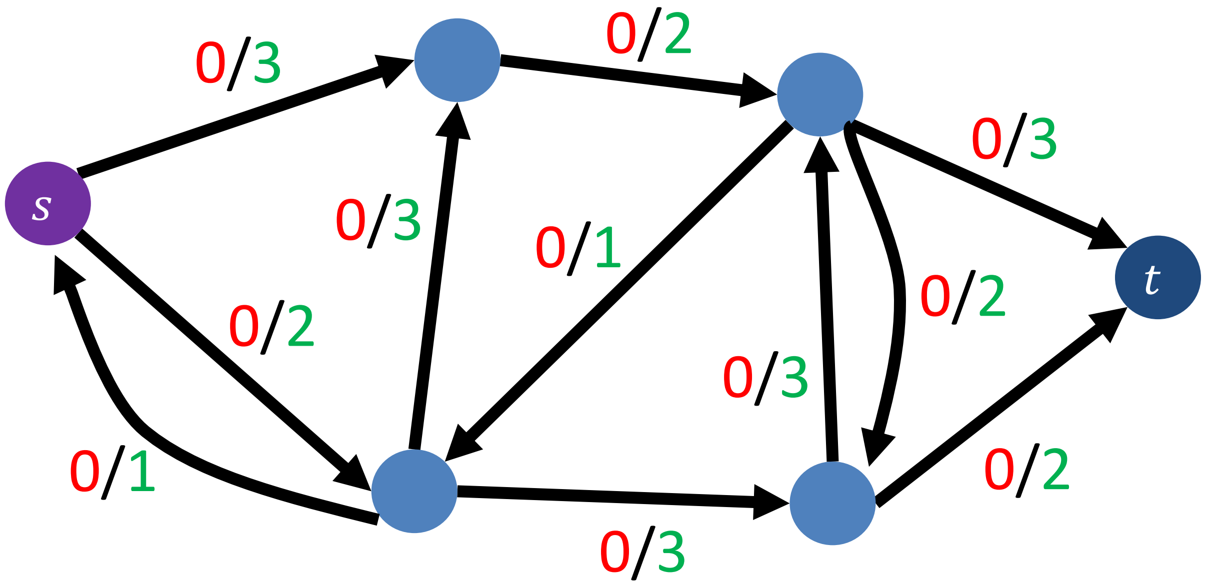


Initially: $f(e) = 0$ for all $e \in E$

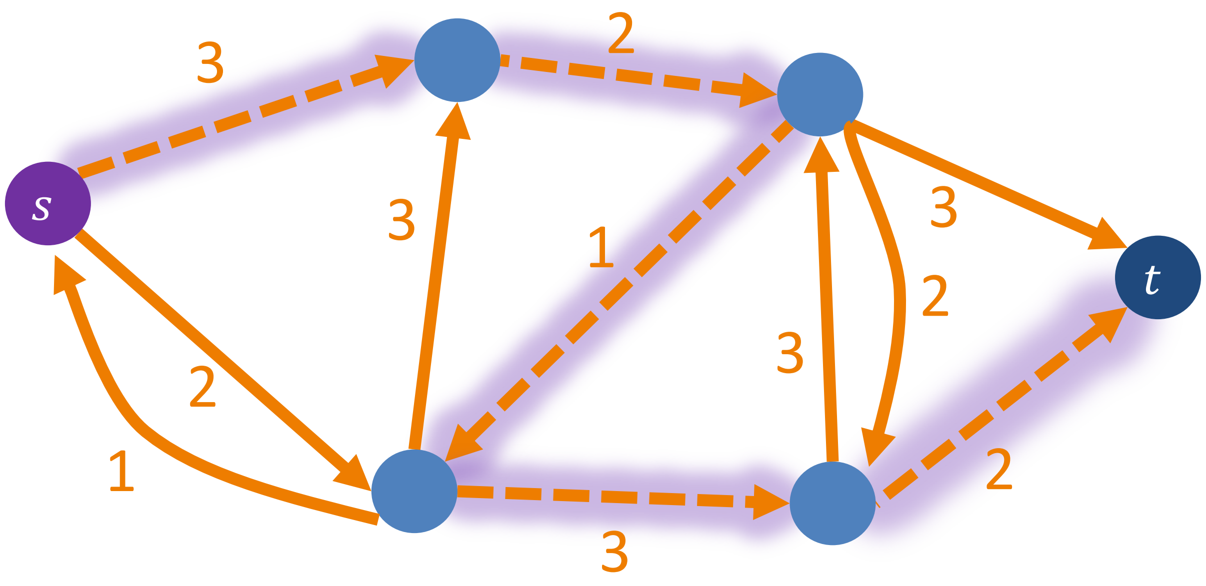


Residual graph G_f

Ford-Fulkerson Example

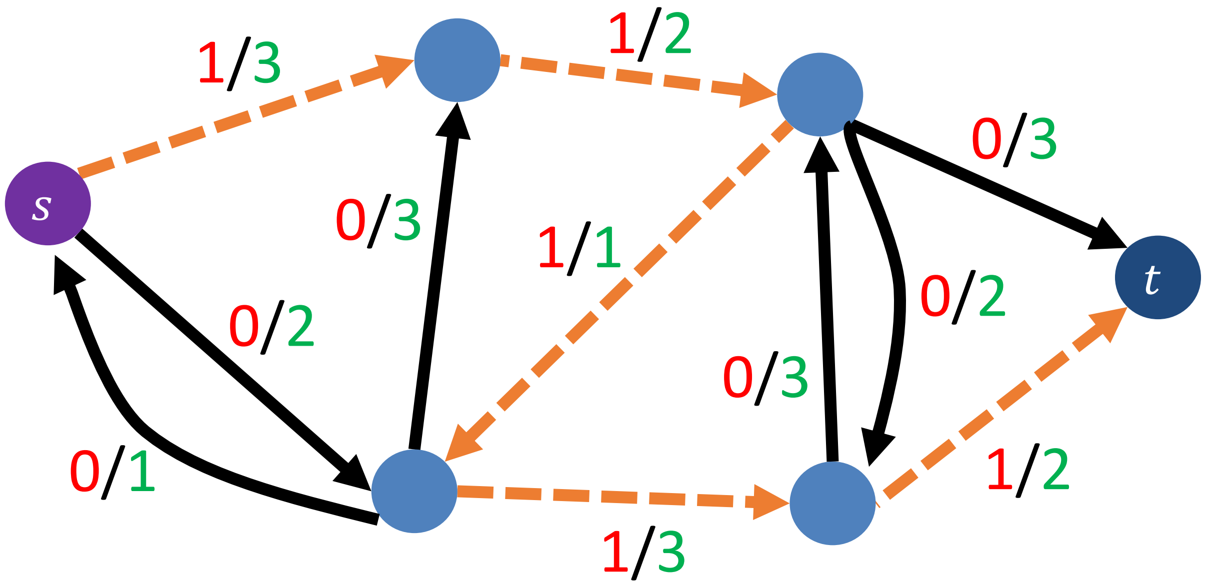


Increase flow by 1 unit

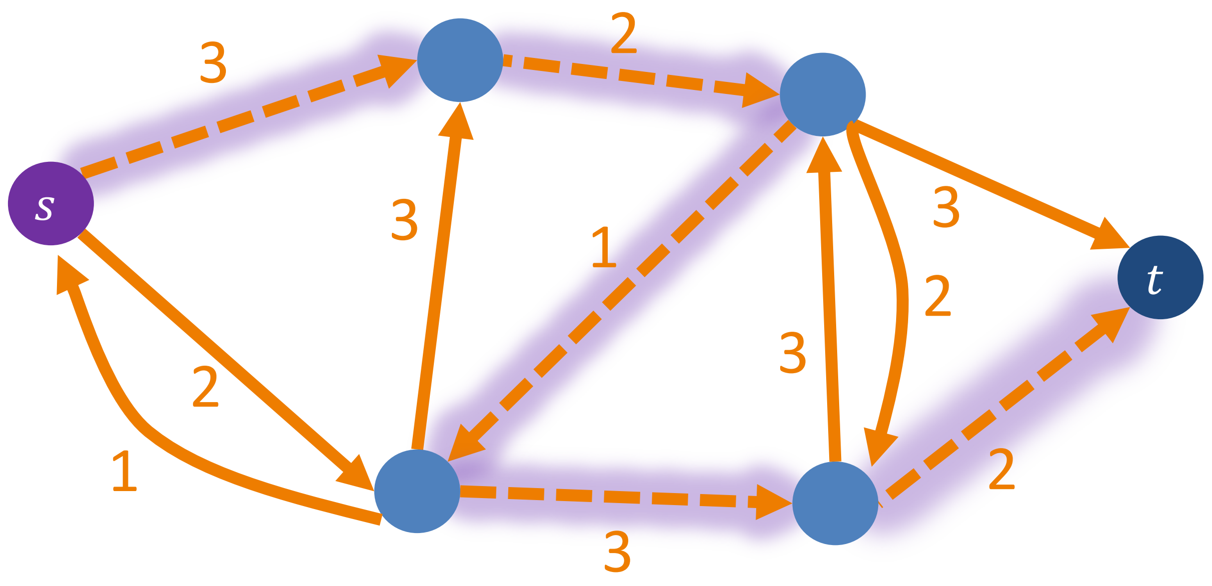


Residual graph G_f

Ford-Fulkerson Example

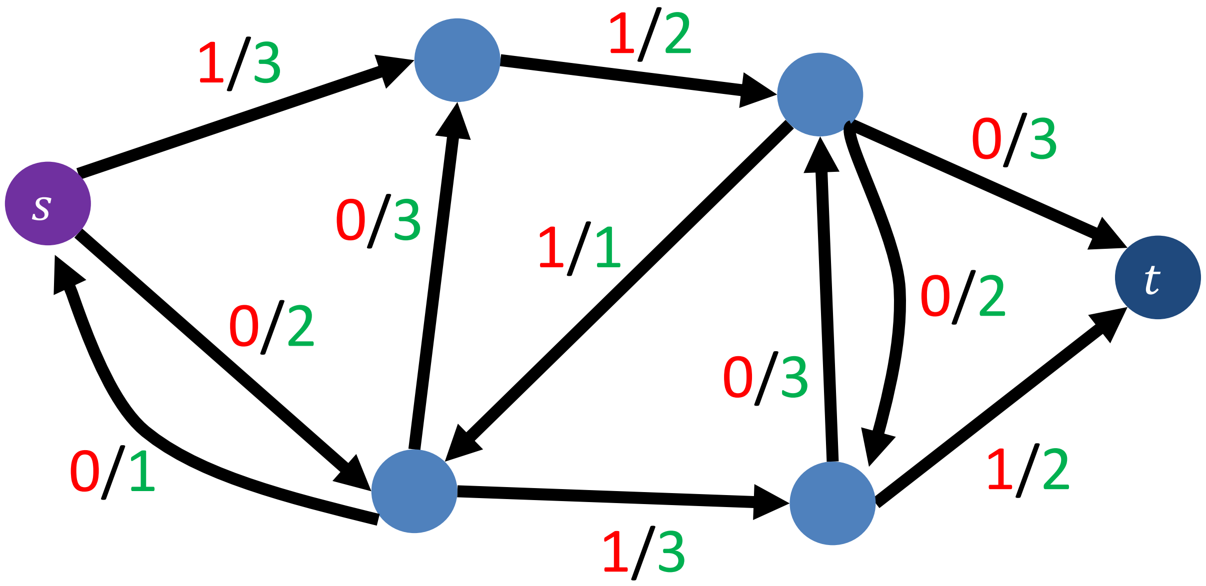


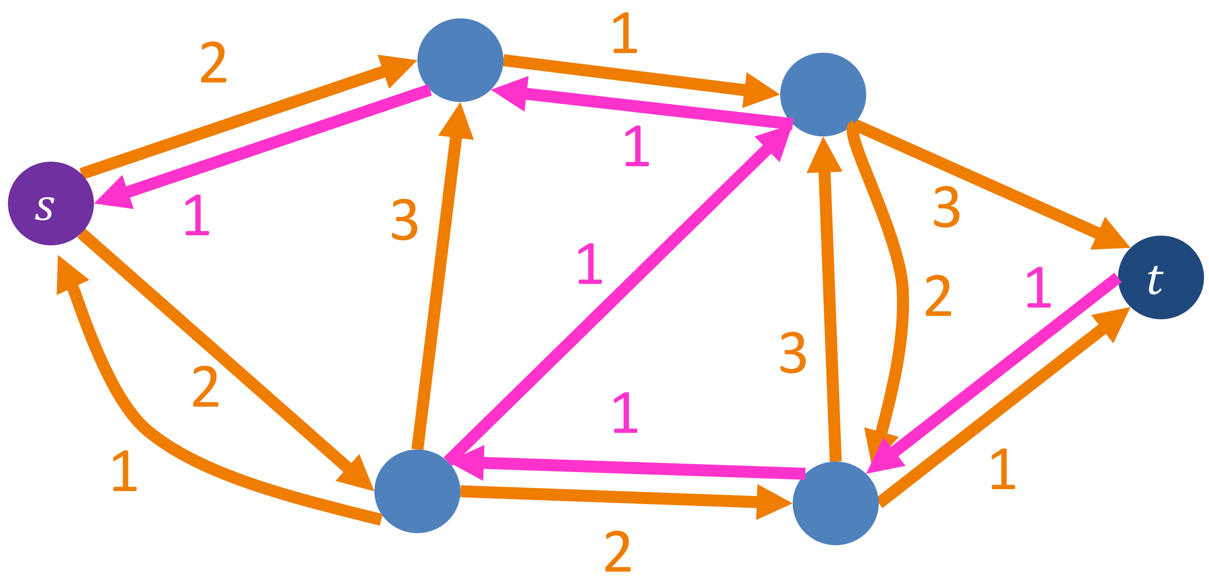
Increase flow by 1 unit



Residual graph G_f

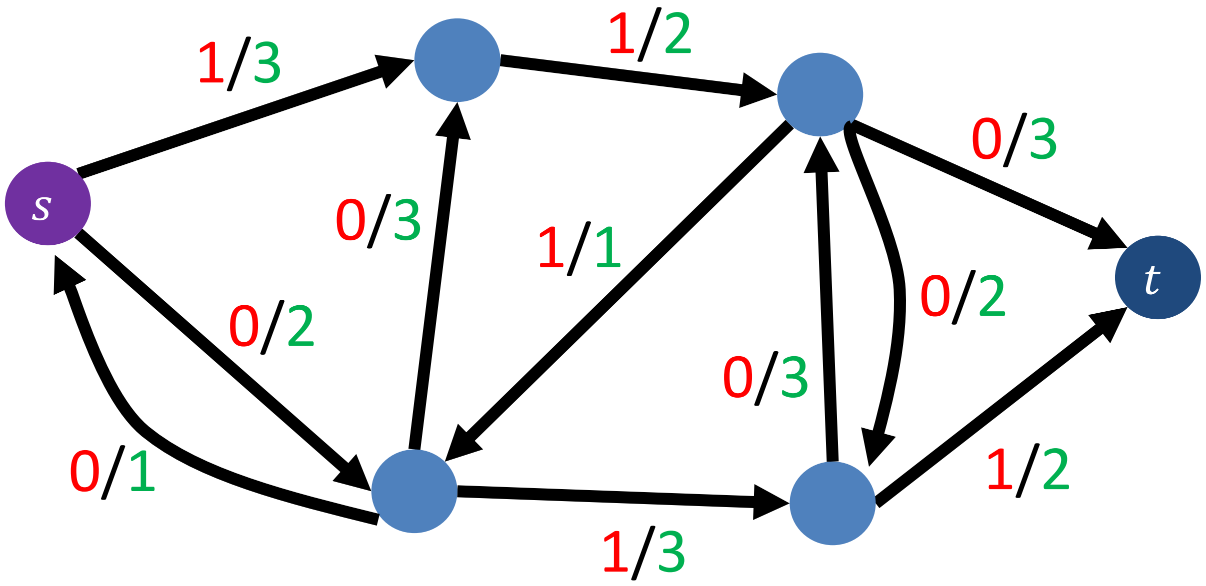
Ford-Fulkerson Example



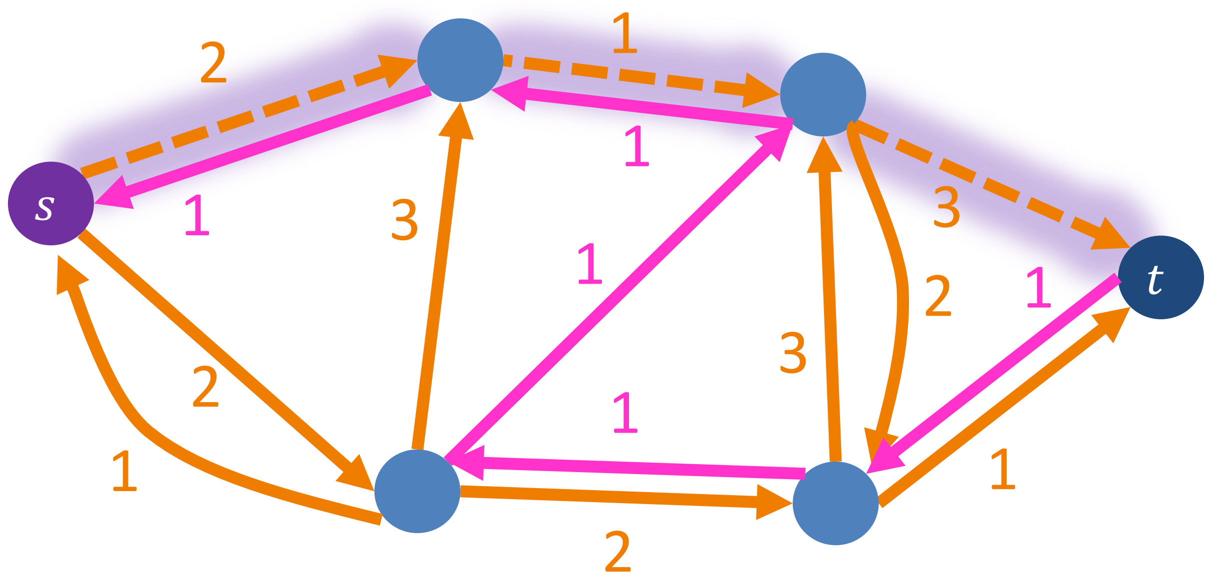


Residual graph G_f

Ford-Fulkerson Example



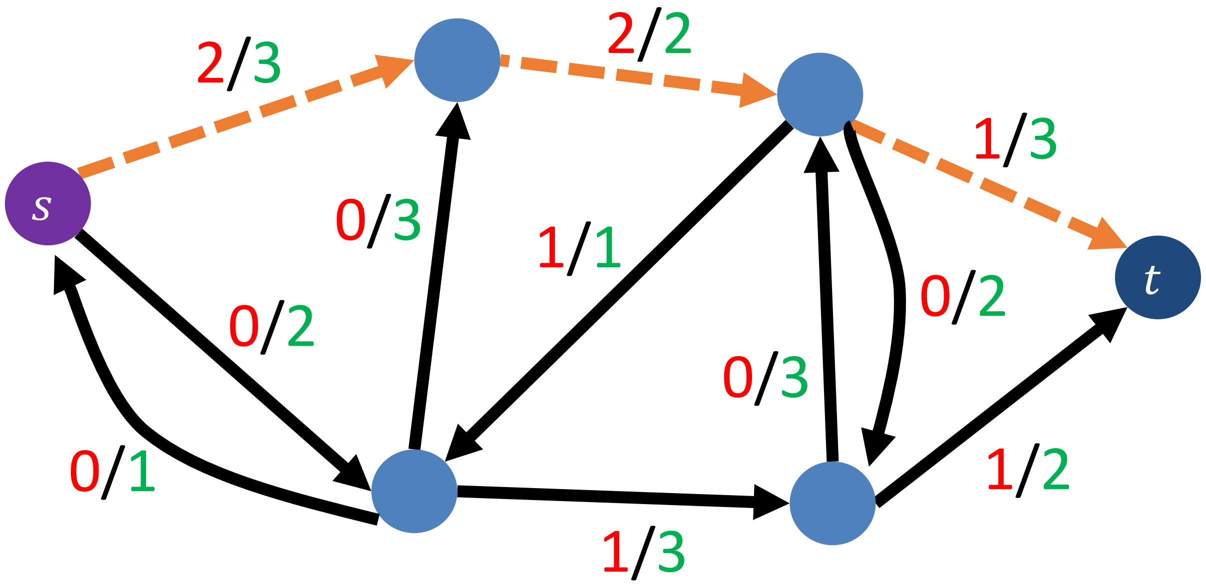
Increase flow by 1 unit

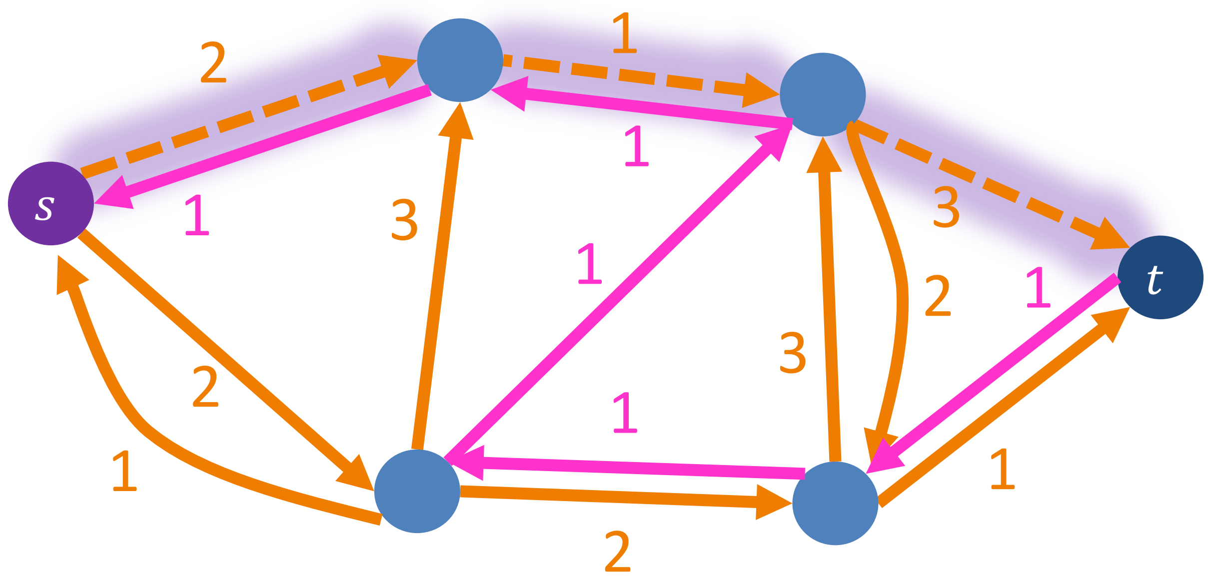


Residual graph G_f

Ford-Fulkerson Example

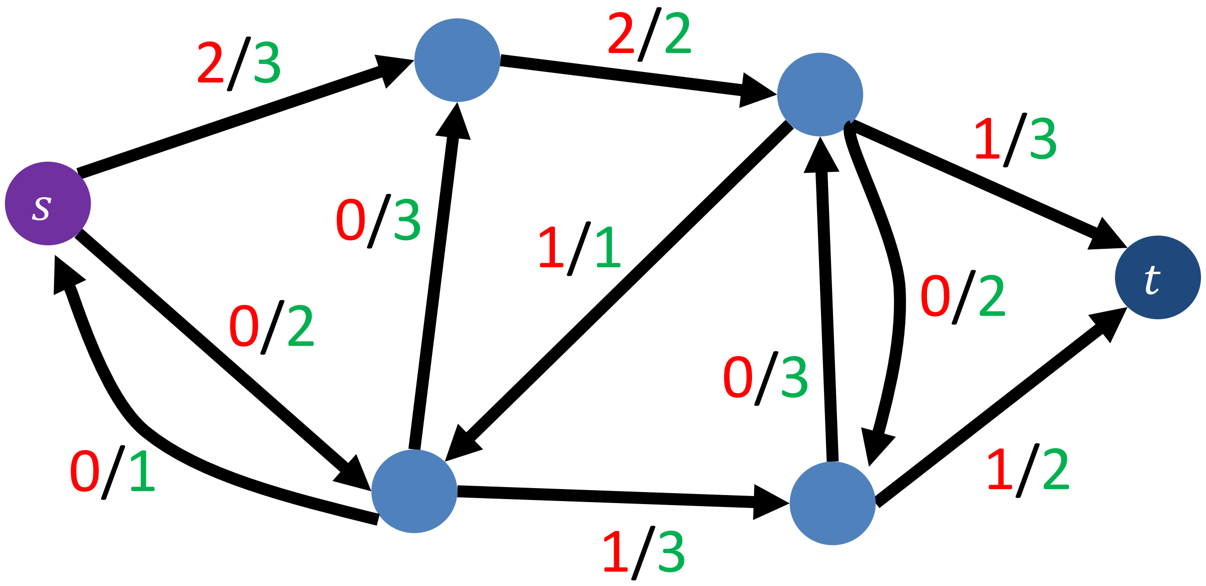
Increase flow by 1 unit

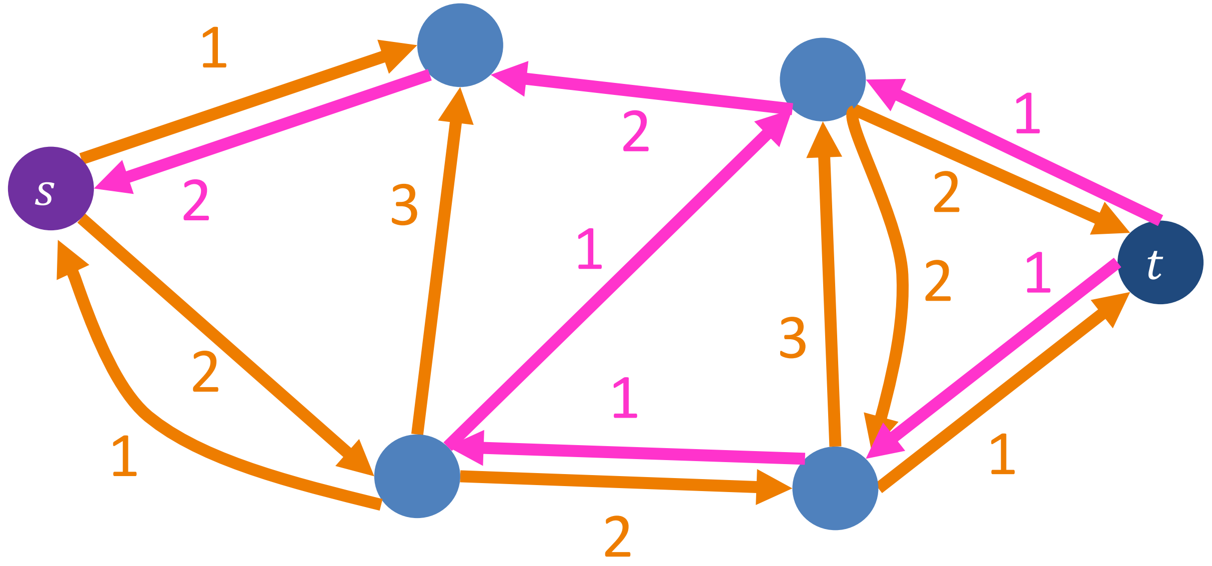




Residual graph G_f

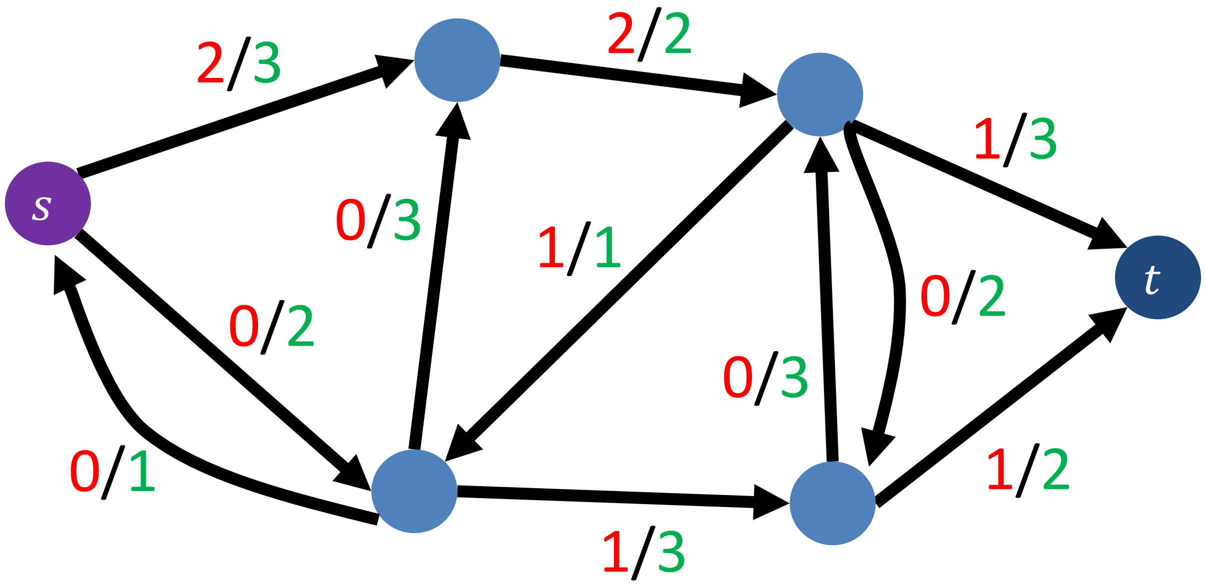
Ford-Fulkerson Example



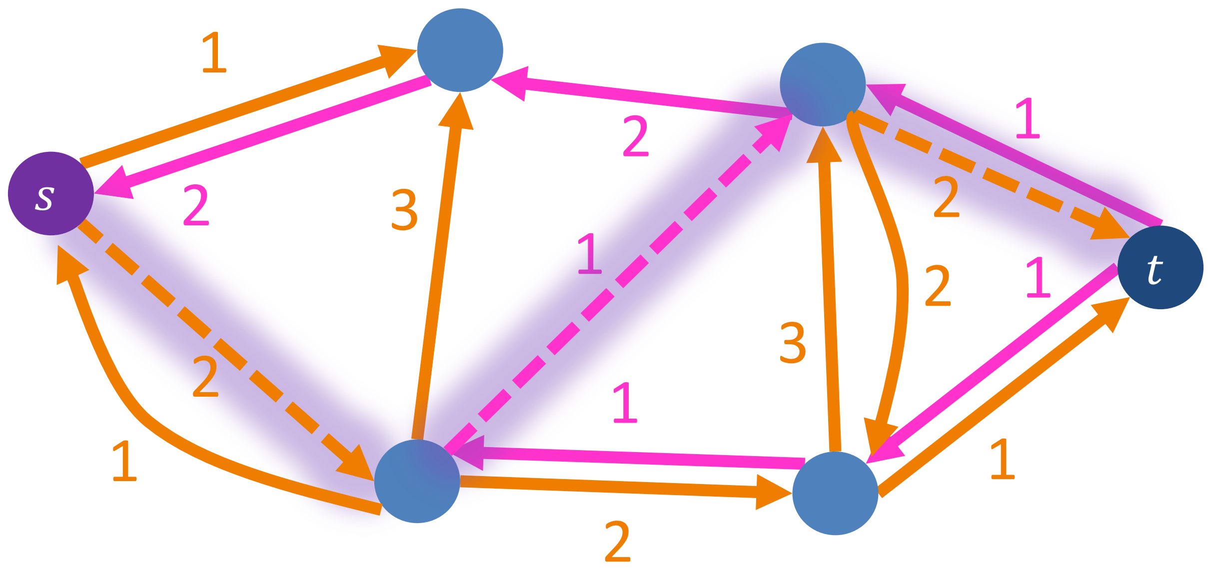


Residual graph G_f

Ford-Fulkerson Example



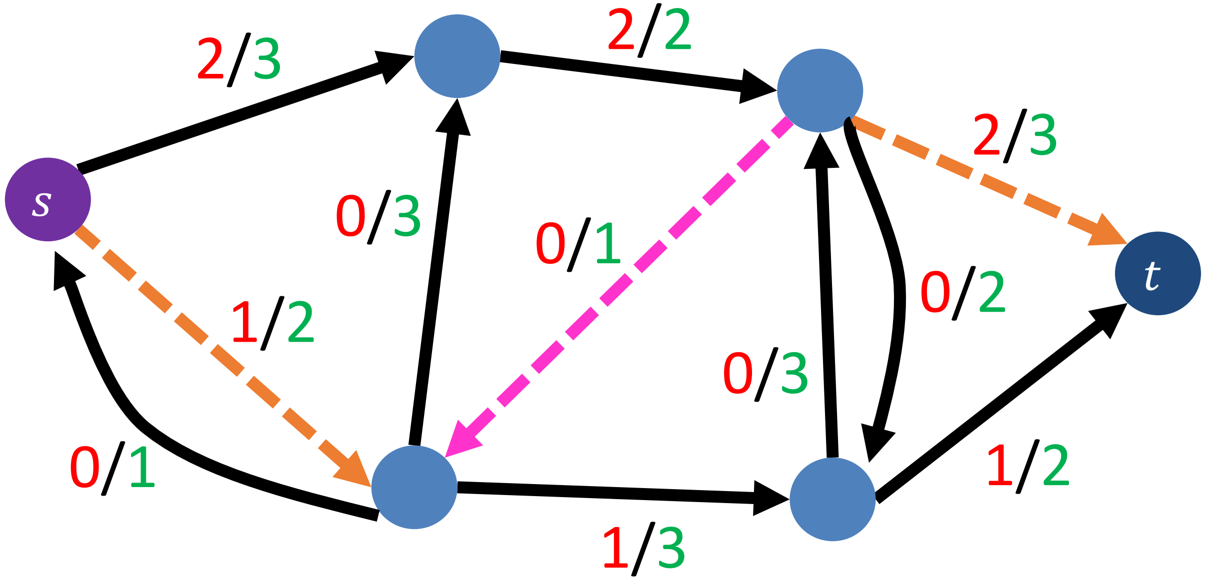
Increase flow by 1 unit

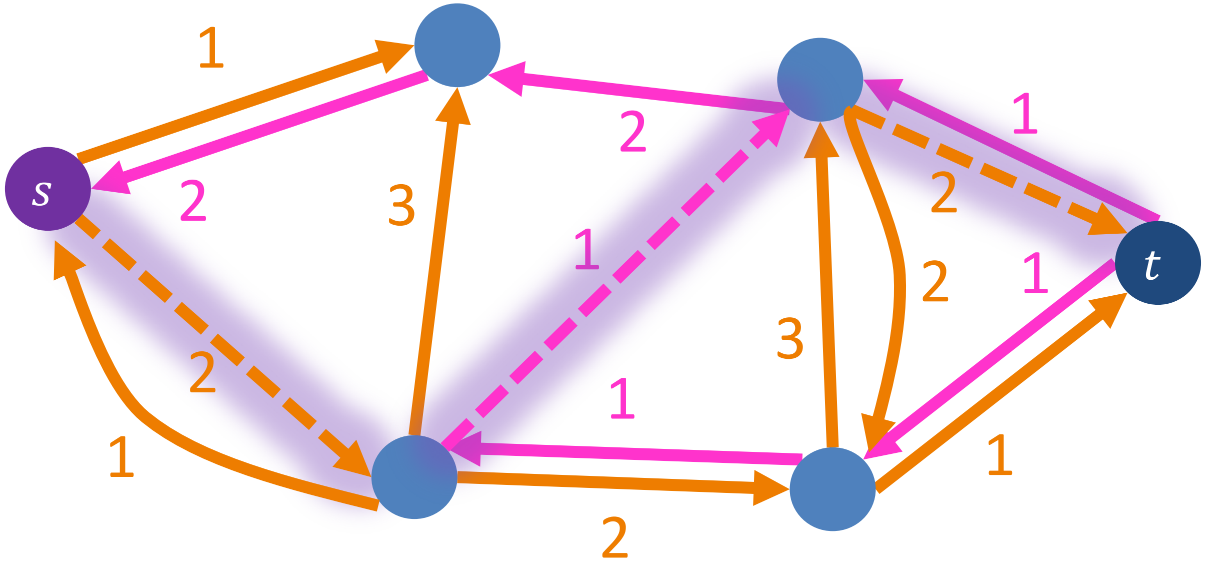


Residual graph G_f

Ford-Fulkerson Example

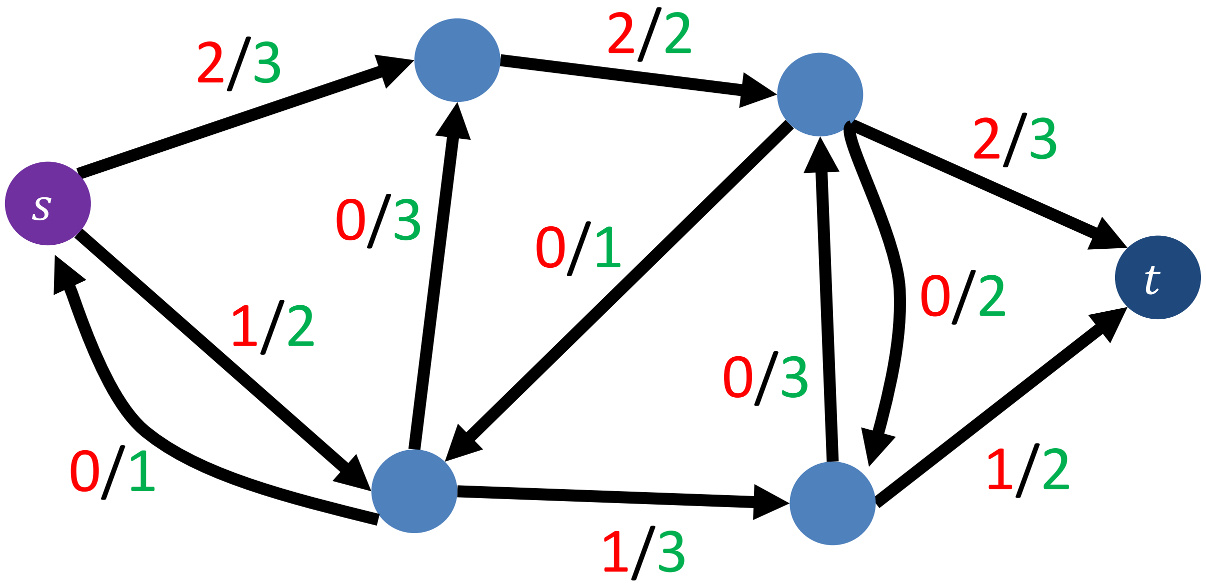
Increase flow by 1 unit

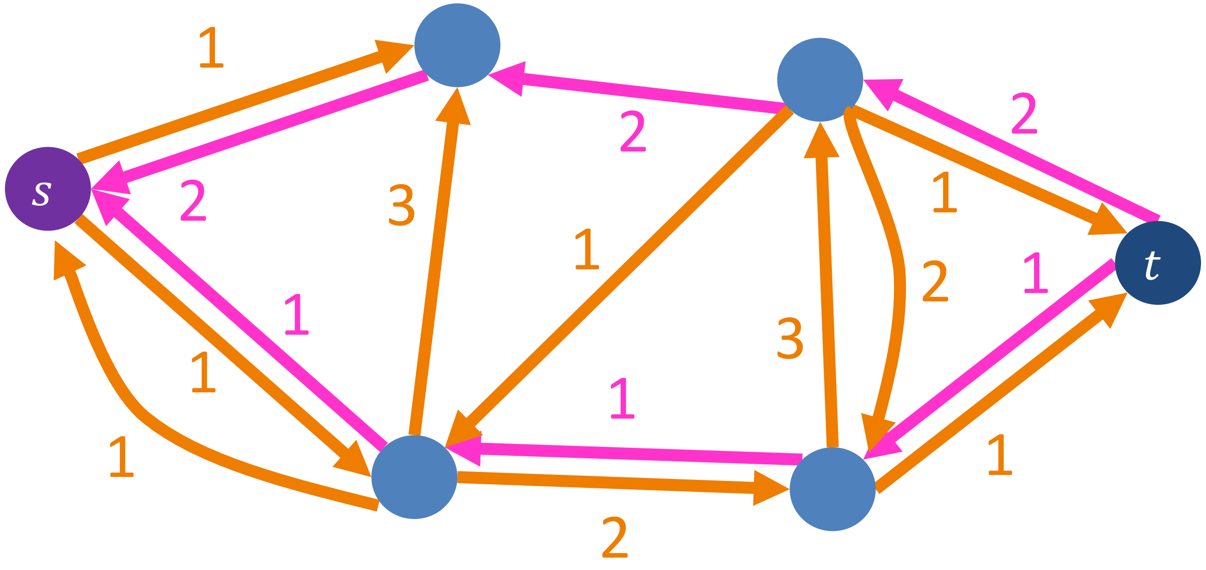




Residual graph G_f

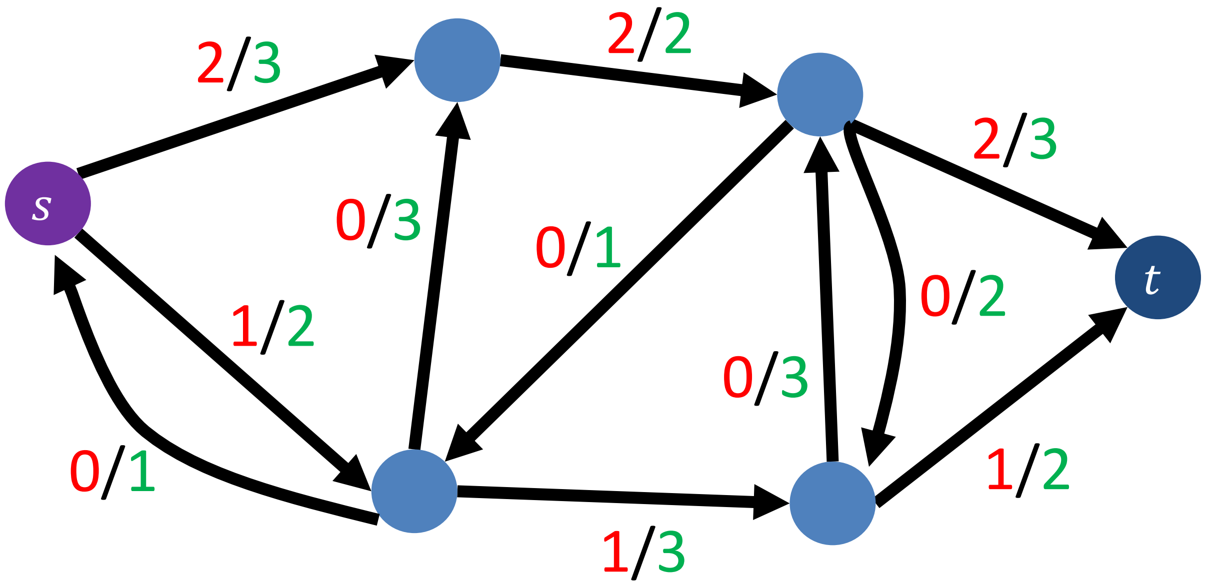
Ford-Fulkerson Example



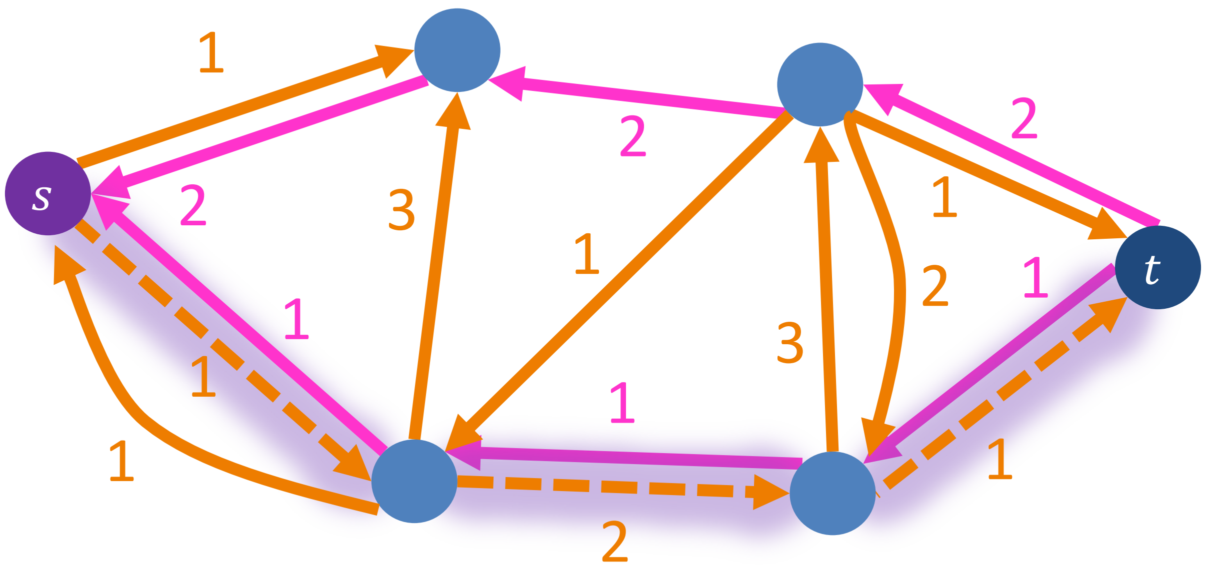


Residual graph G_f

Ford-Fulkerson Example

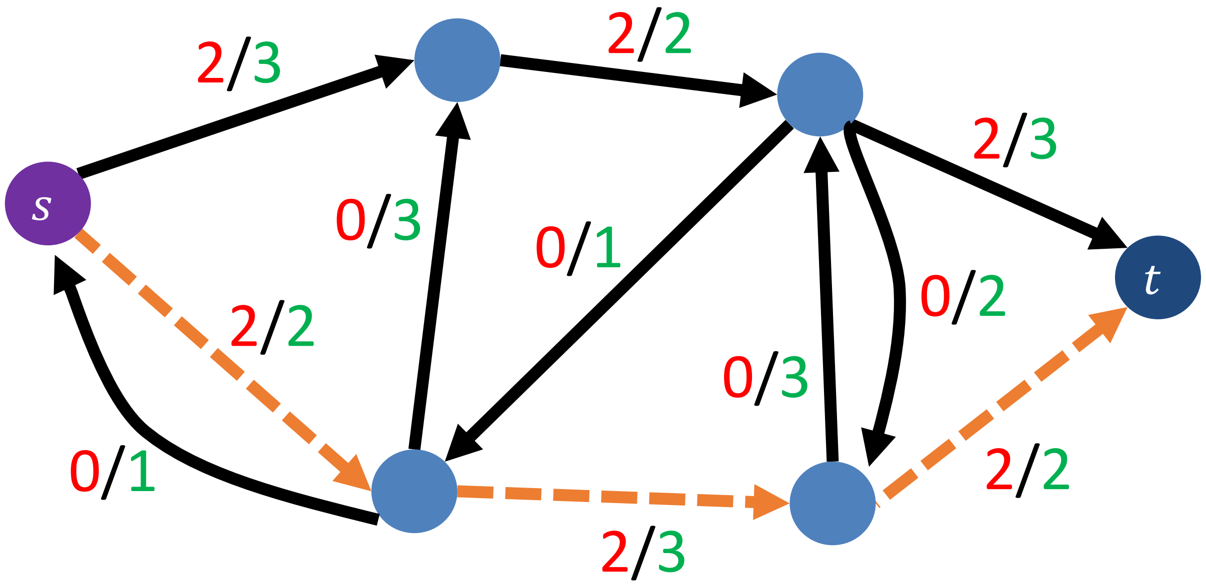


Increase flow by 1 unit

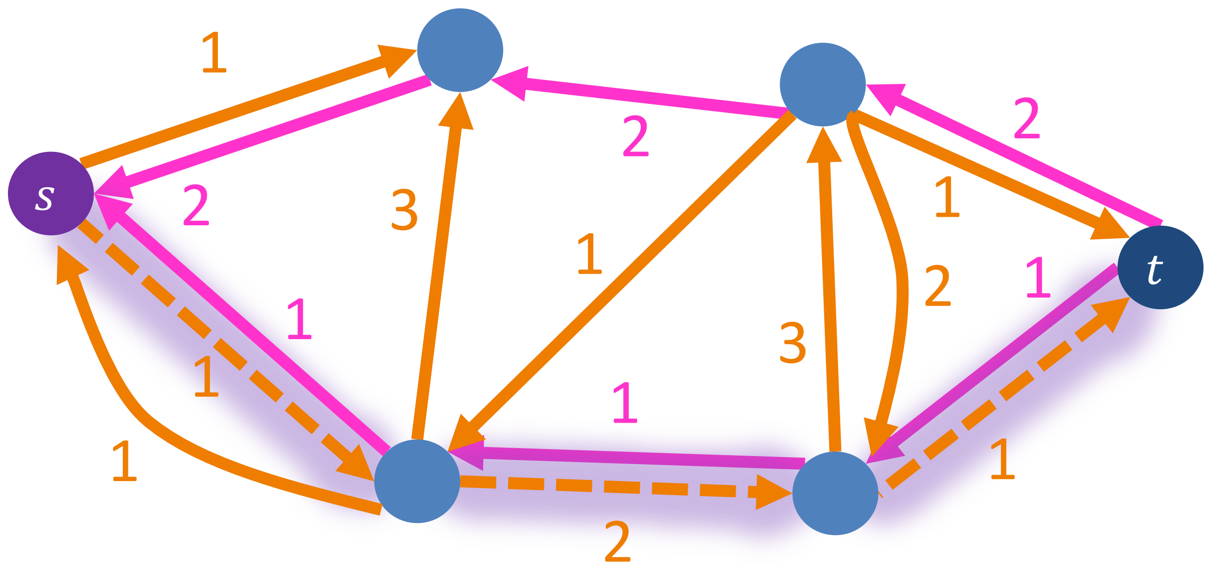


Residual graph G_f

Ford-Fulkerson Example



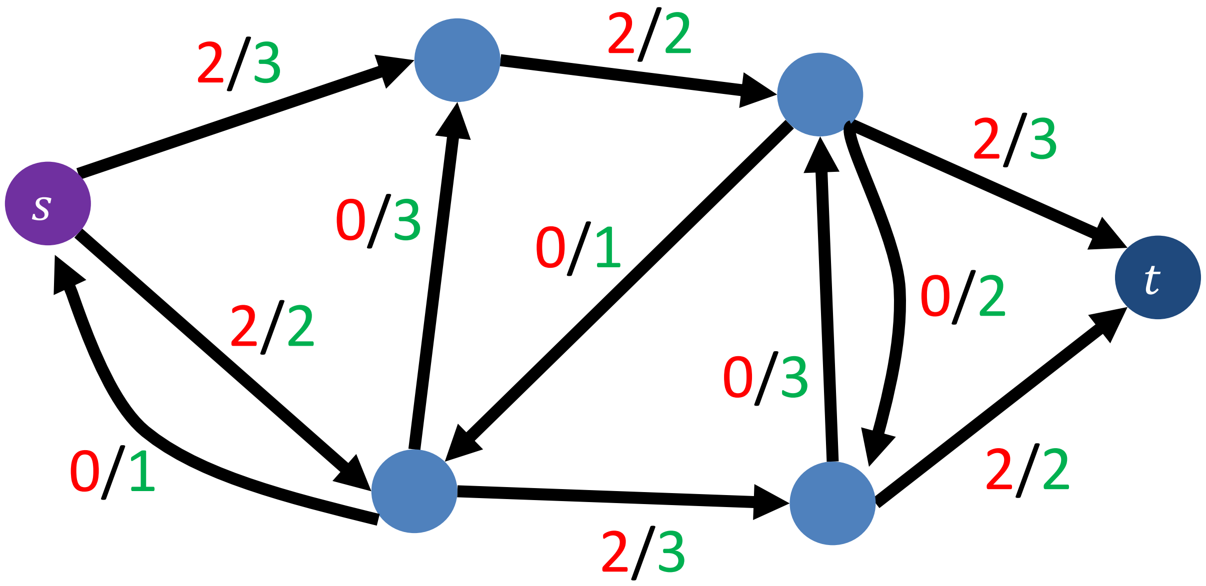
Increase flow by 1 unit

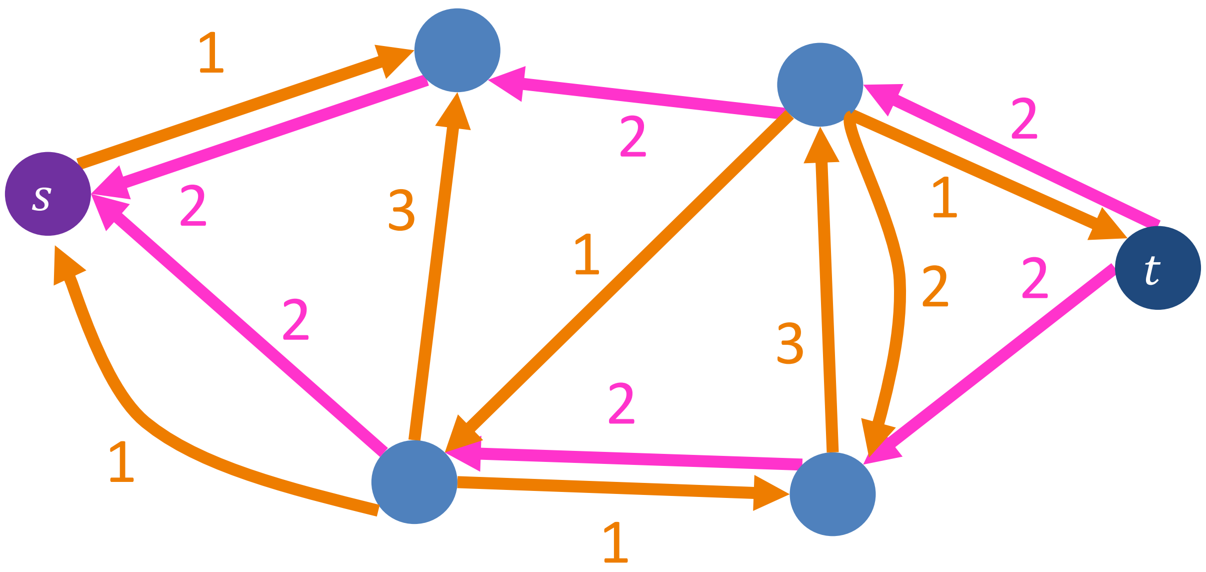


Residual graph G_f

Ford-Fulkerson Example

No more augmenting paths





Residual graph G_f

Maximum flow: 4

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

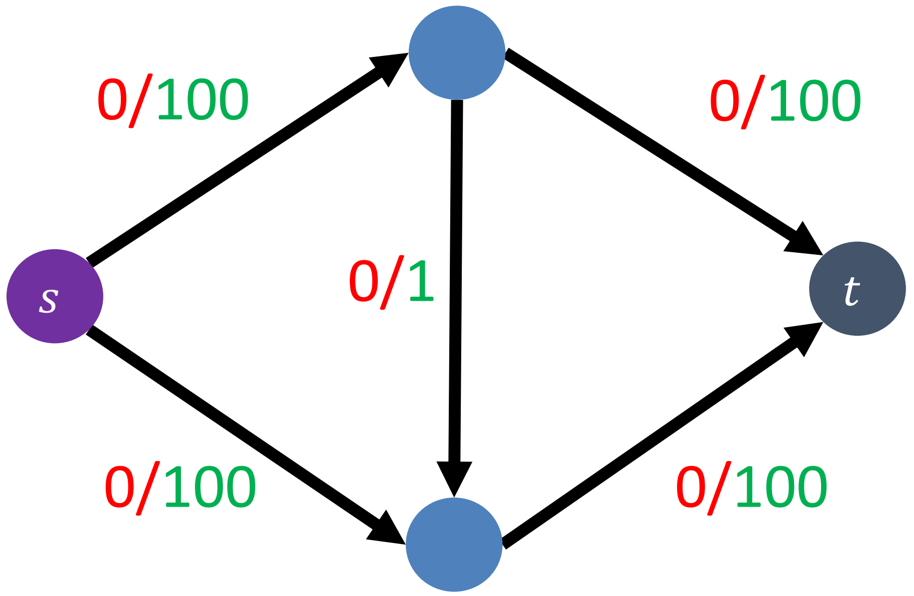
Define an augmenting path to be an $s \rightarrow t$ path in the residual graph G_f (using edges of non-zero weight)

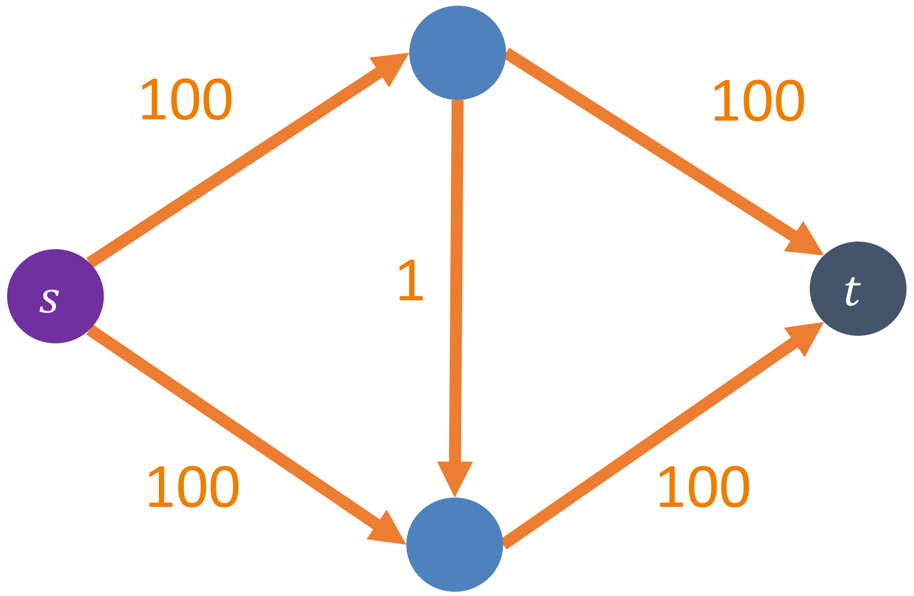
How many iterations are needed?

- 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

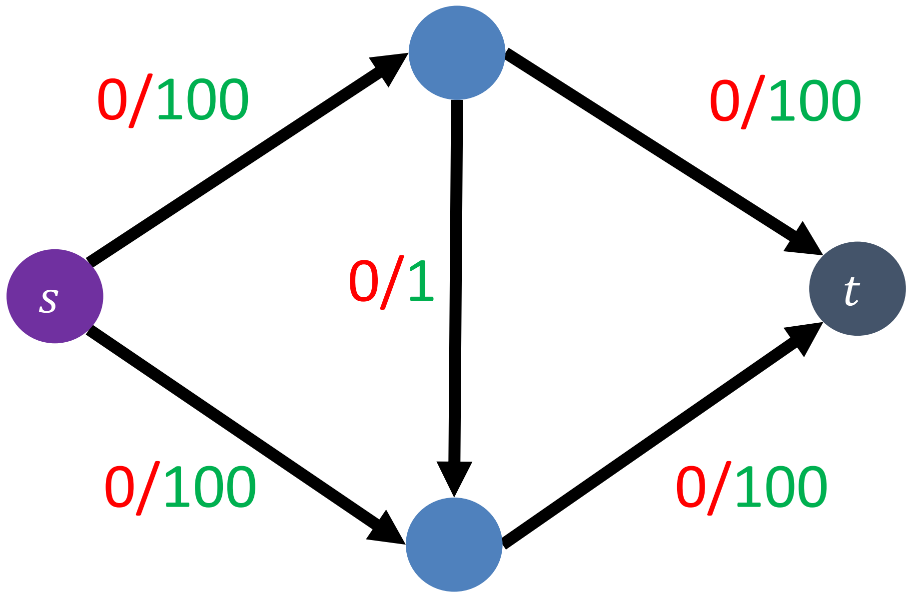
Initialization: $O(|E|)$
Construct residual network: $O(|E|)$
Finding augmenting path in residual network: $O(|E|)$ using BFS/DFS

Worst-Case Ford-Fulkerson

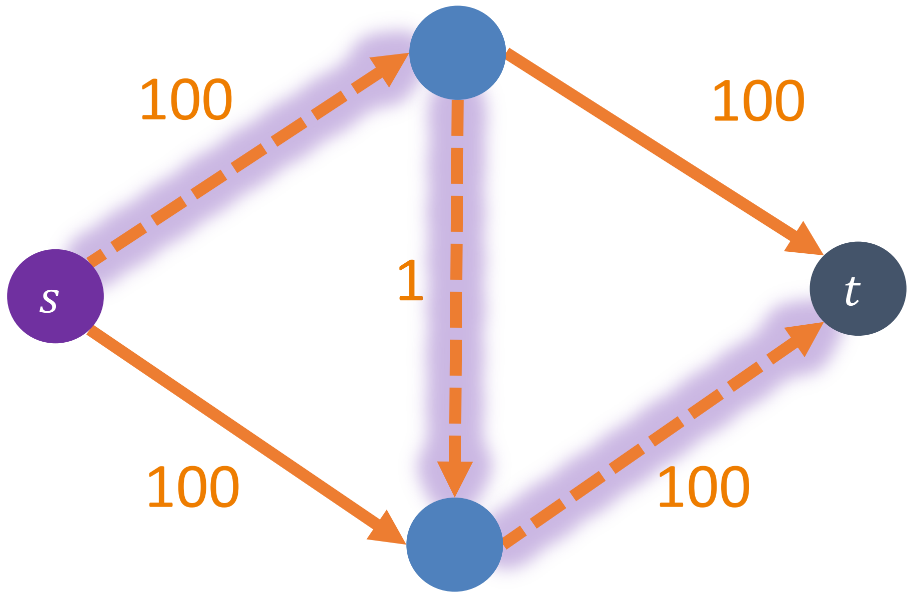




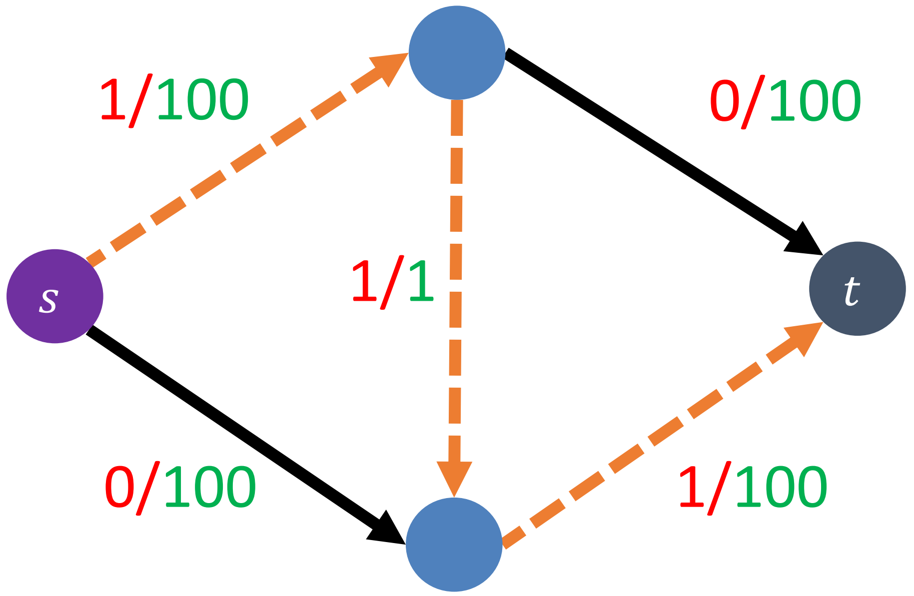
Worst-Case Ford-Fulkerson



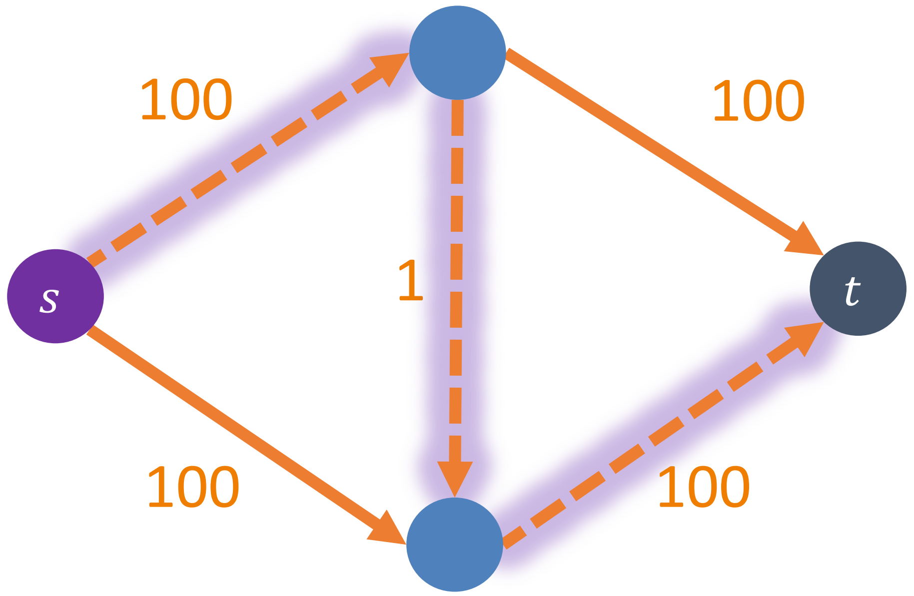
Increase flow by 1 unit



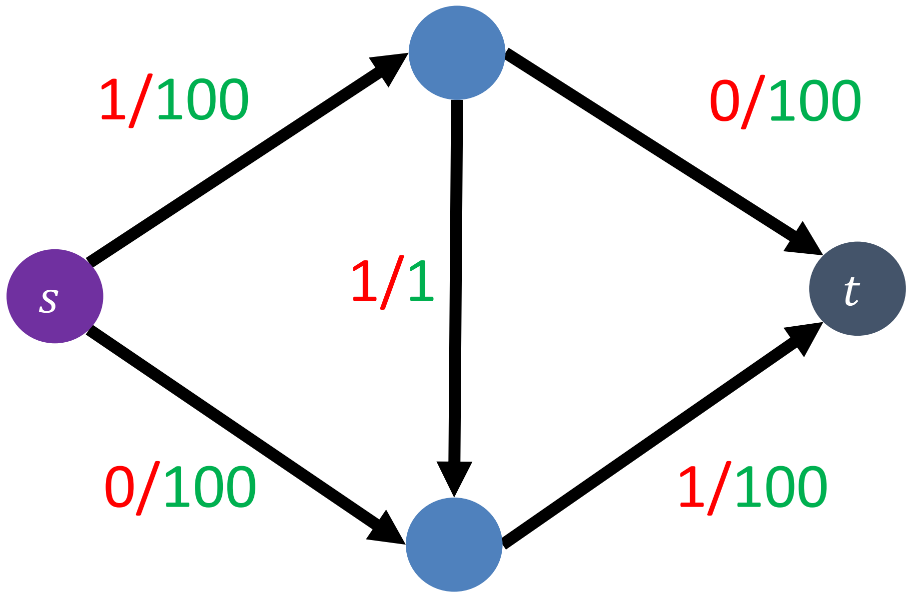
Worst-Case Ford-Fulkerson

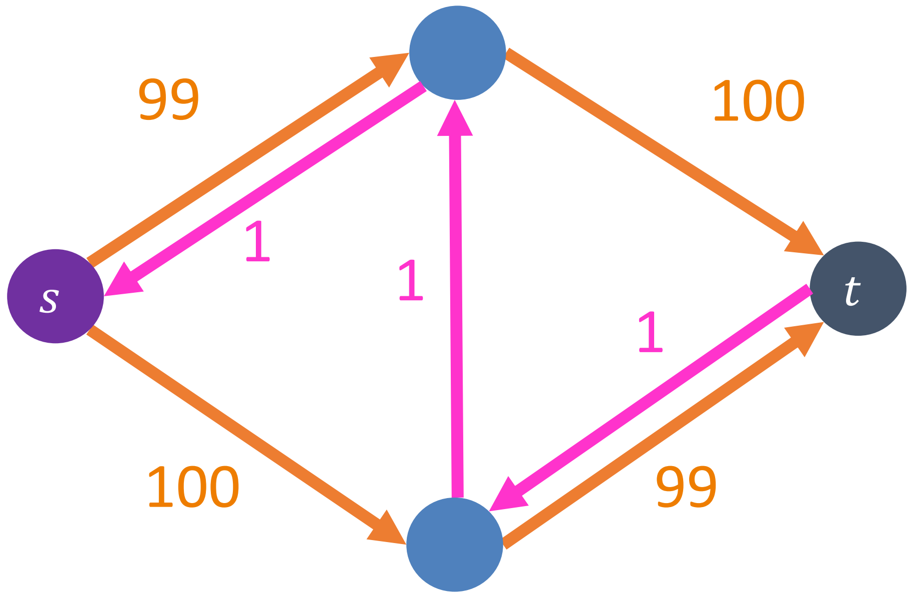


Increase flow by 1 unit

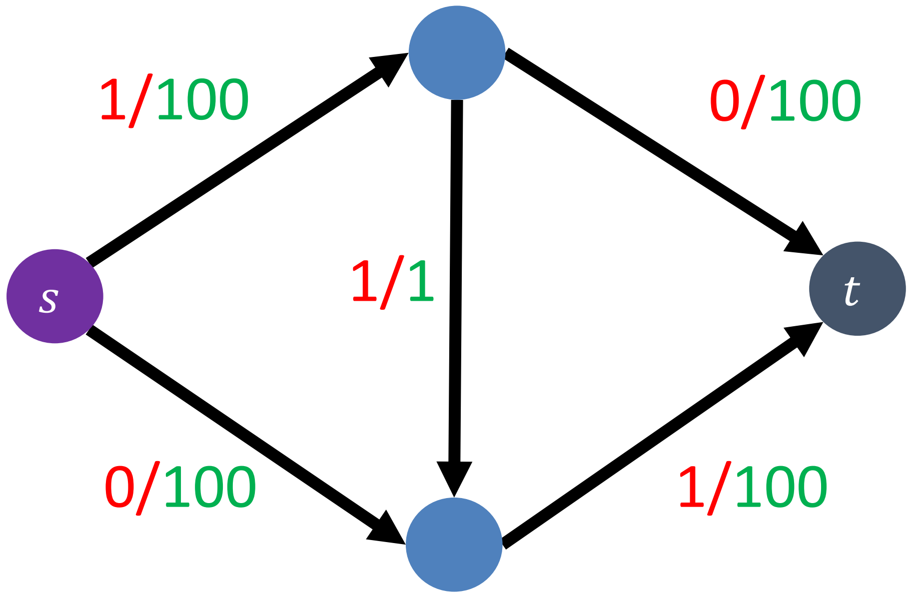


Worst-Case Ford-Fulkerson

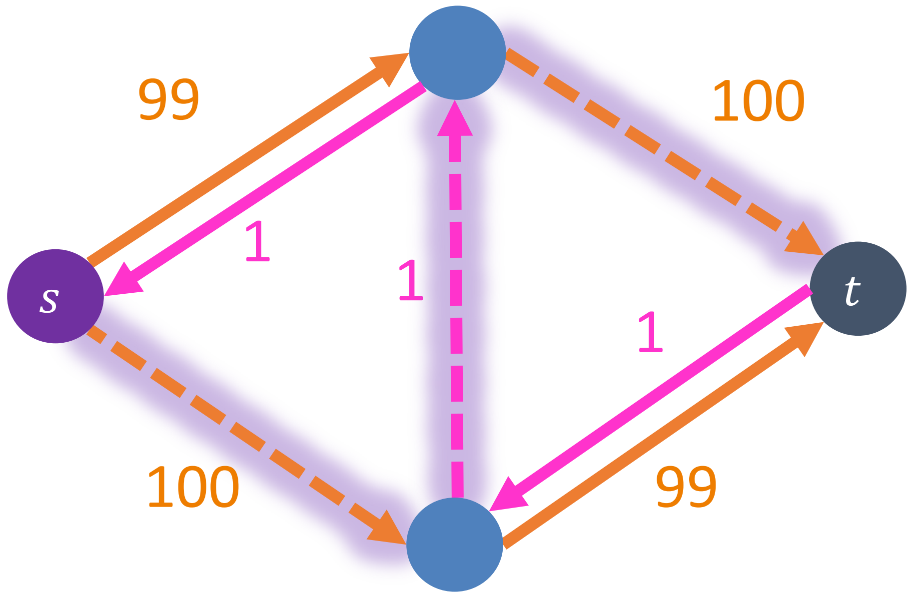




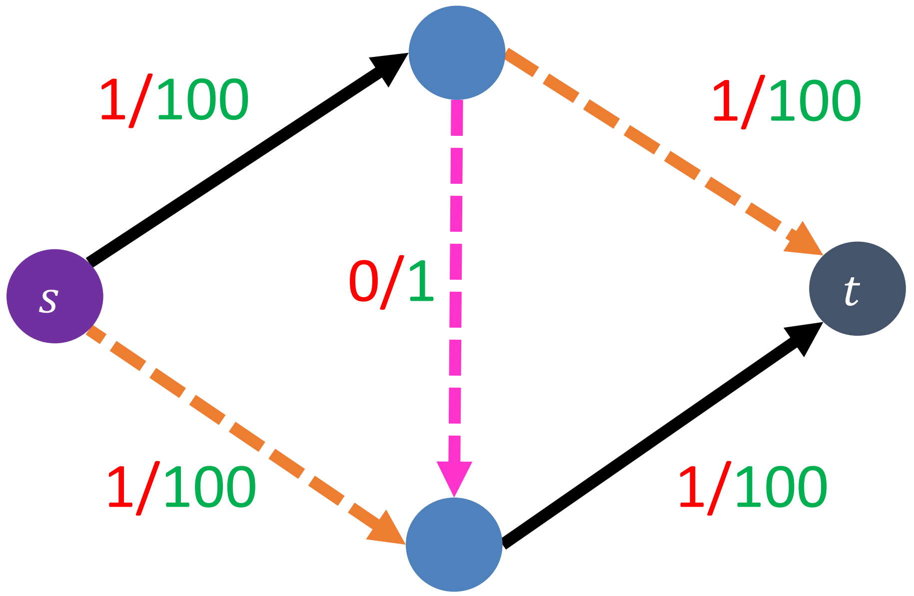
Worst-Case Ford-Fulkerson



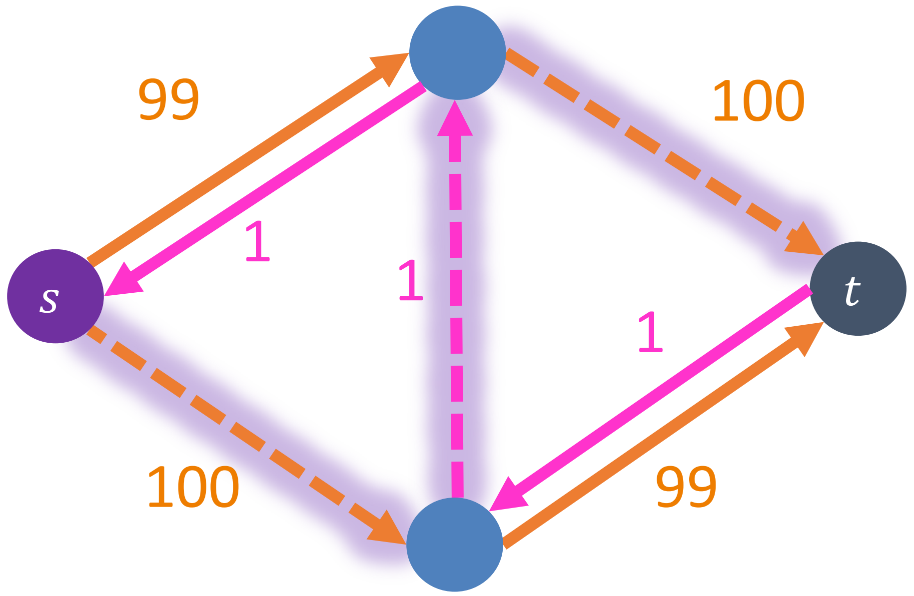
Increase flow by 1 unit



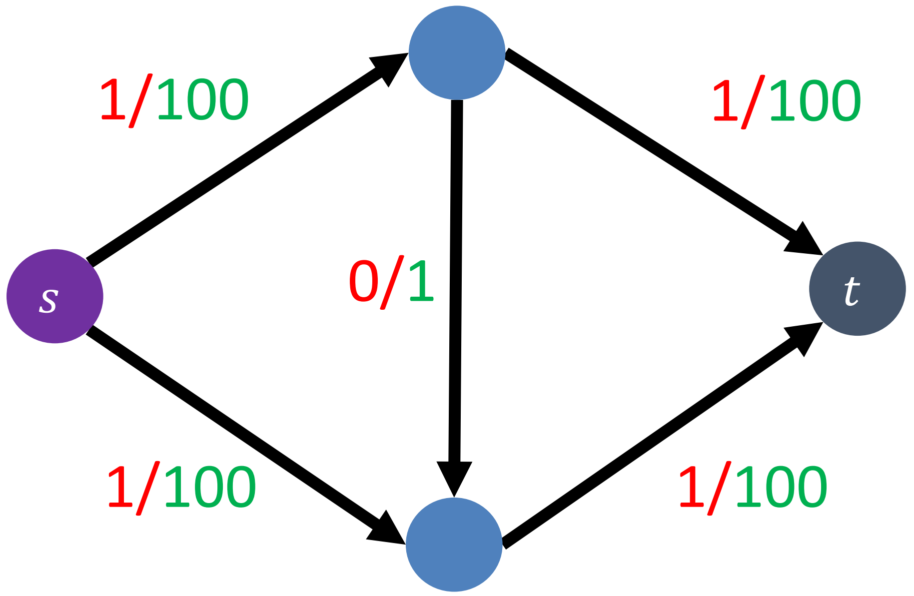
Worst-Case Ford-Fulkerson

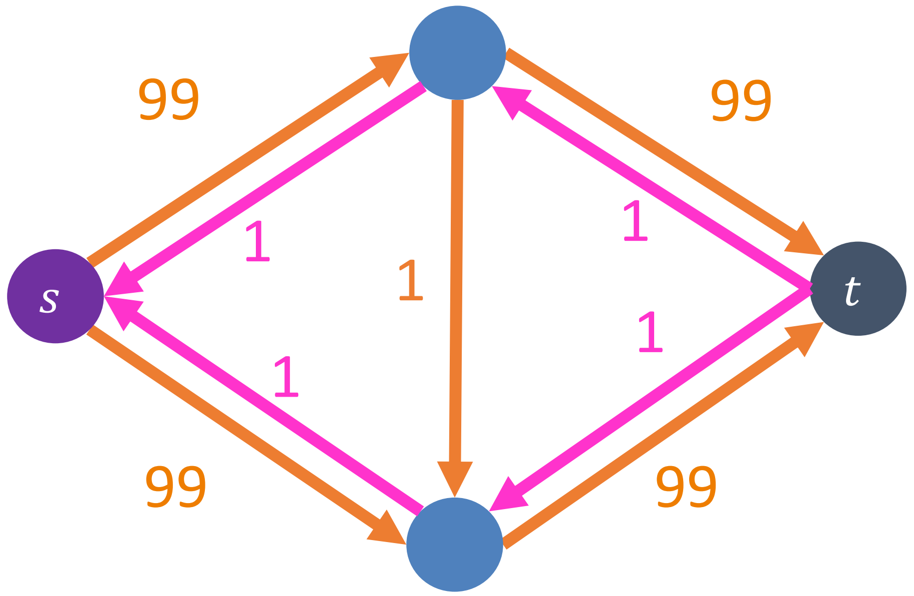


Increase flow by 1 unit

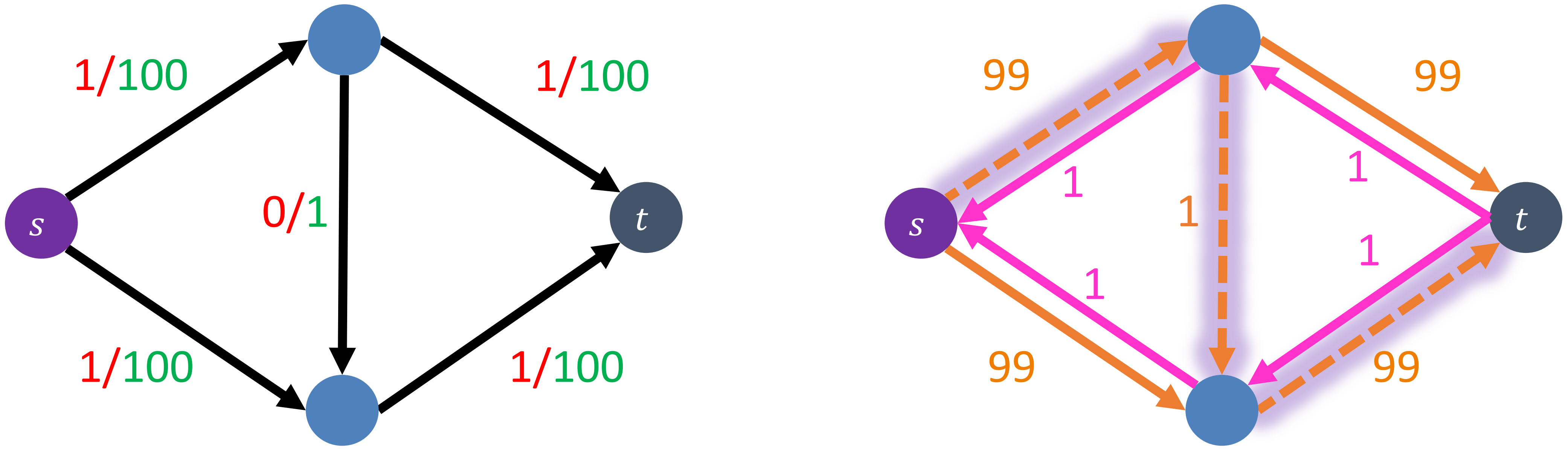


Worst-Case Ford-Fulkerson





Worst-Case Ford-Fulkerson



Observation: each iteration increases flow by 1 unit

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

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 <u>not</u> polynomial-time!

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

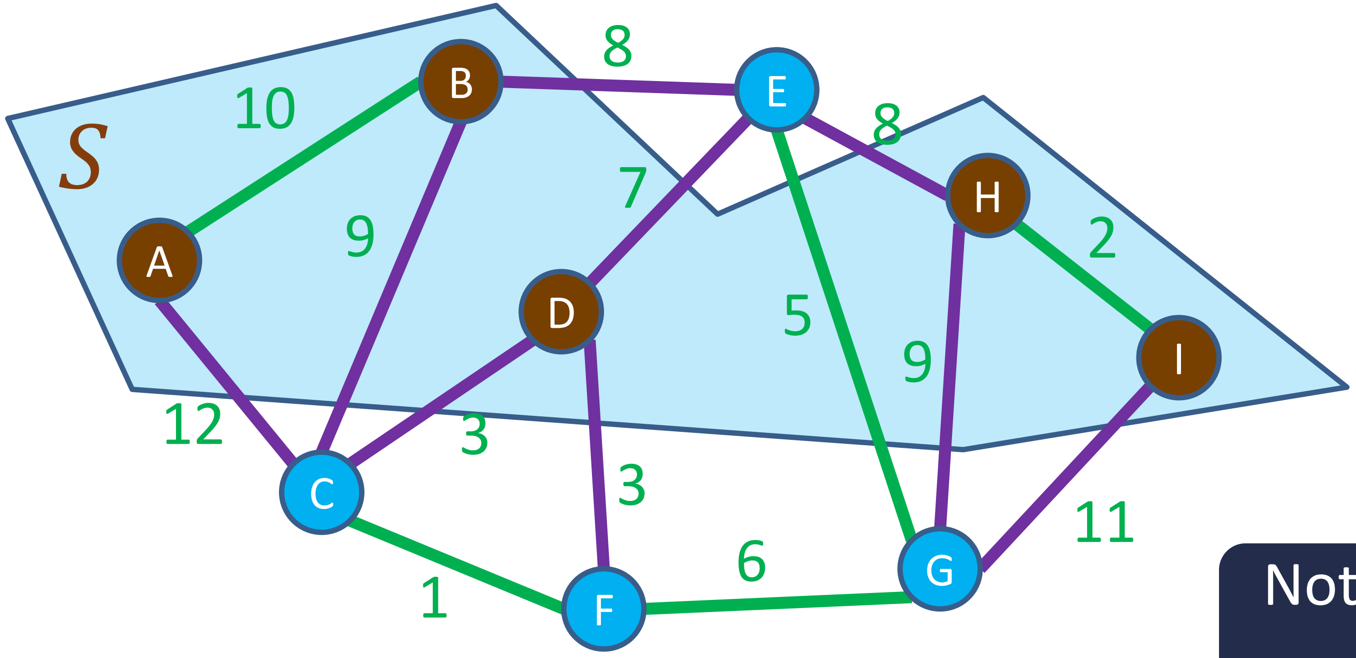
Ford-Fulkerson max-flow algorithm:

- 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 $V - S$



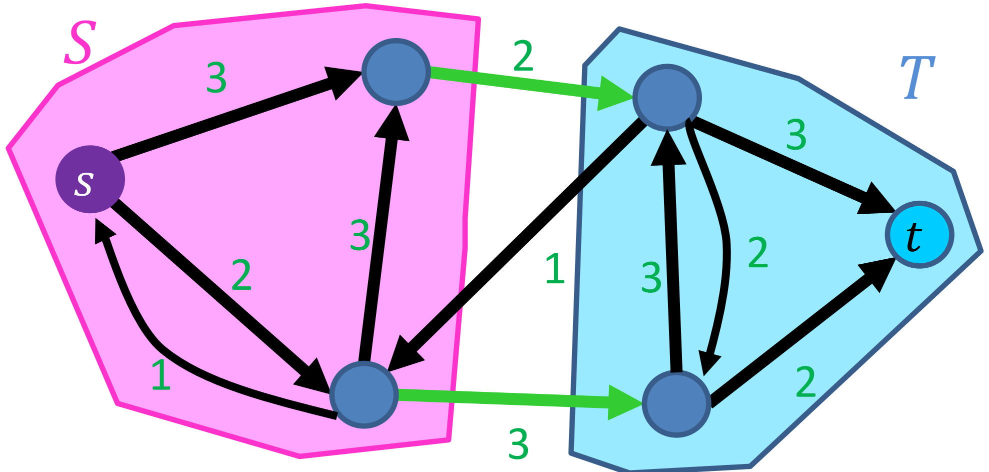
Notion extends naturally to a set of edges

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 $v_1, v_2 \in V - S$

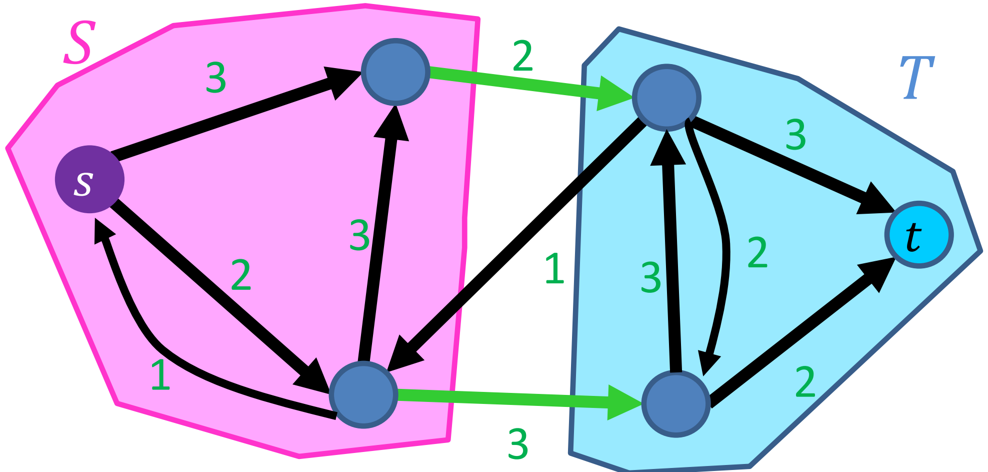
Showing Correctness of Ford-Fulkerson

- 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

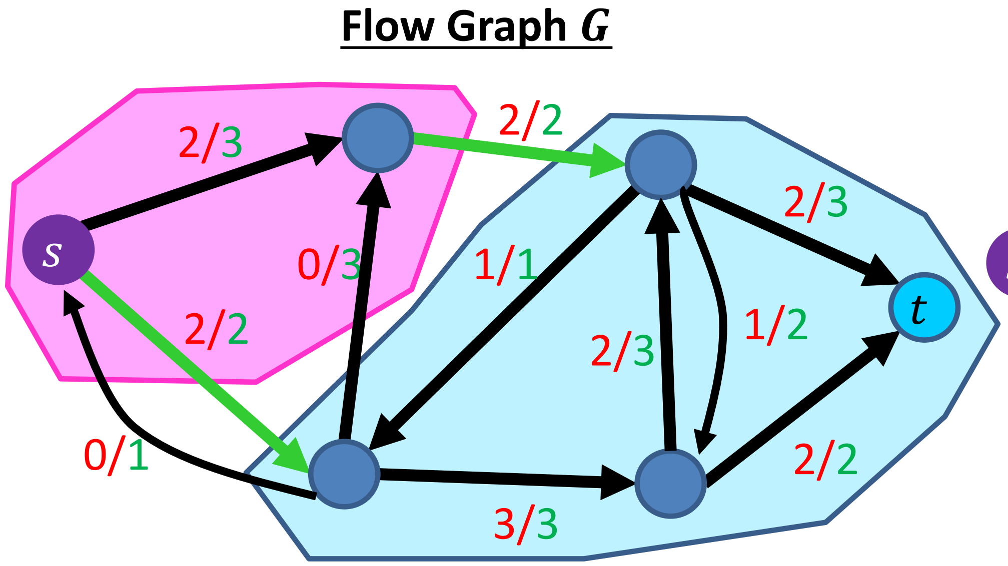
- Max flow upper bounded by any cut separating s and t
- Why? "Conservation of flow"
  - All flow exiting s must eventually get to t
  - To get from s to t, all "tanks" must cross the cut
- Conclusion: If we find the minimum-cost cut, we've found the maximum flow
  - $\max_f |f| \leq \min_{S,T} ||S,T||$



Maxflow/Mincut Theorem

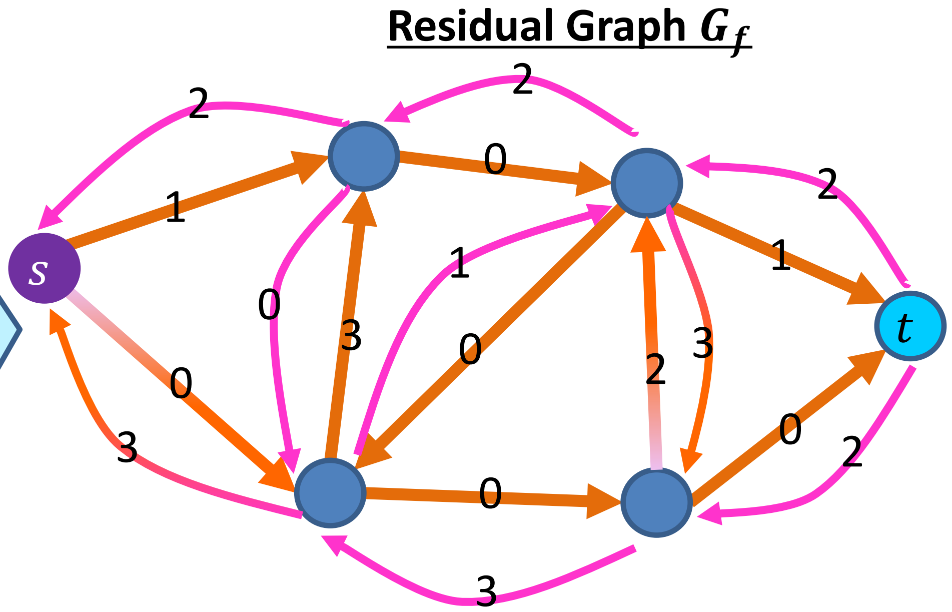
- To show Ford-Fulkerson is correct:
  - Show that when there are no more augmenting paths, there is a cut with cost equal to the flow
- Conclusion: the maximum flow through a network matches the minimum-cost cut
  - $\max_{f} |f| = \min_{S,T} ||S, T||$
- Duality
  - When we've maximized max flow, we've 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

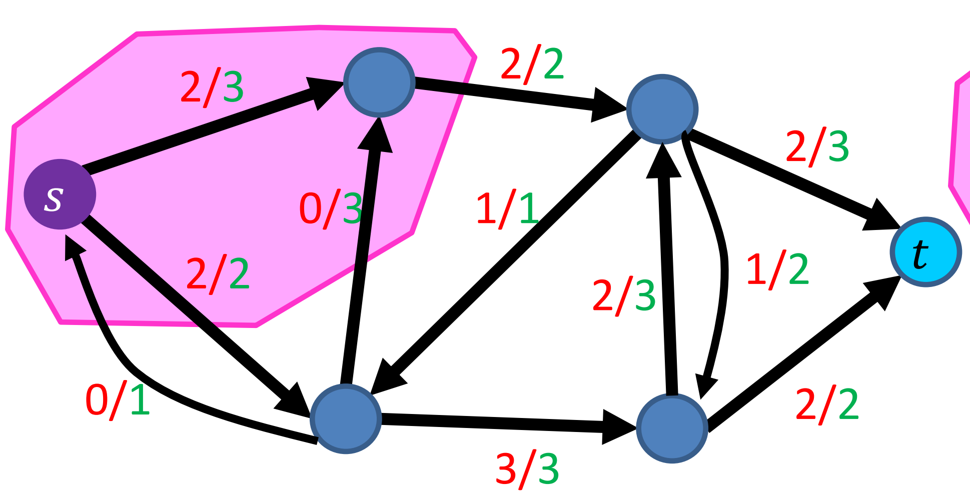
Idea: When there are no more augmenting paths, there exists a cut in the graph with cost matching the flow

61

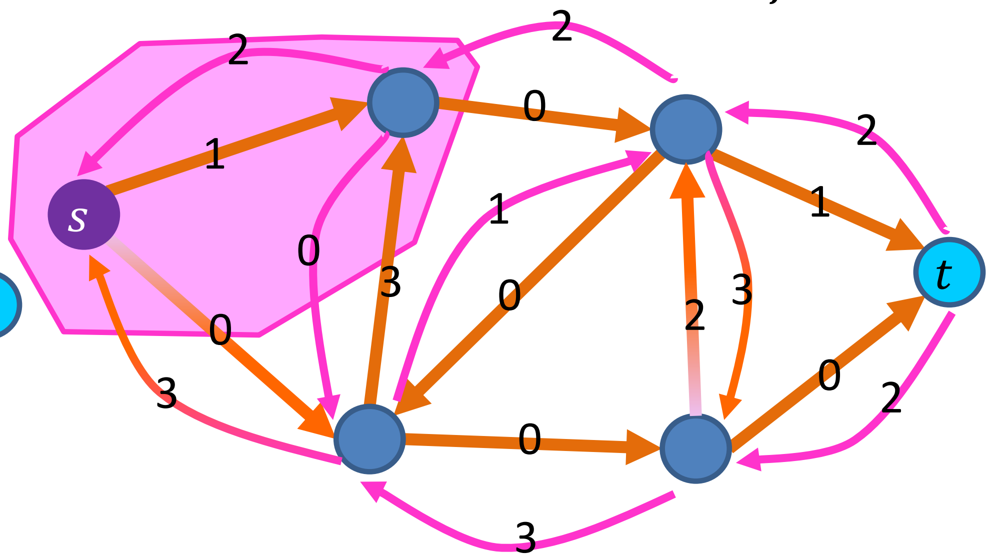
Proof: Maxflow/Mincut Theorem

- 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



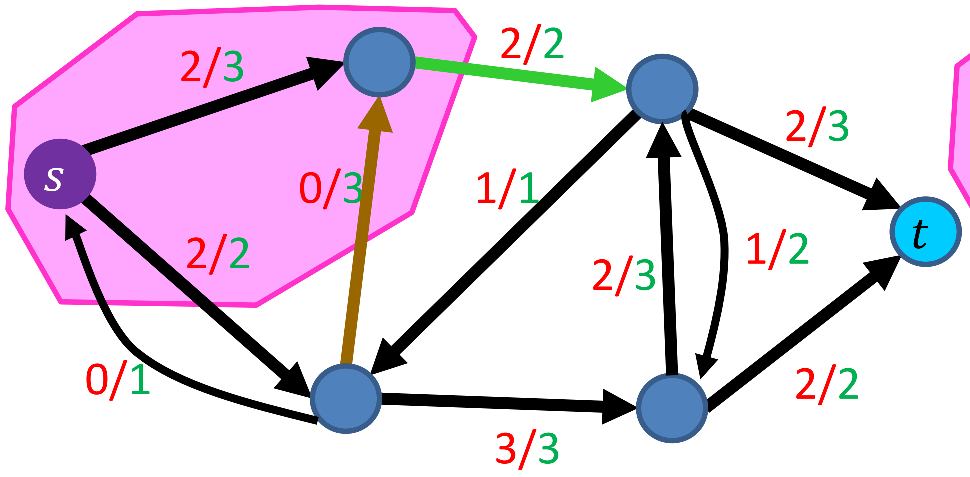
Residual Graph G_f



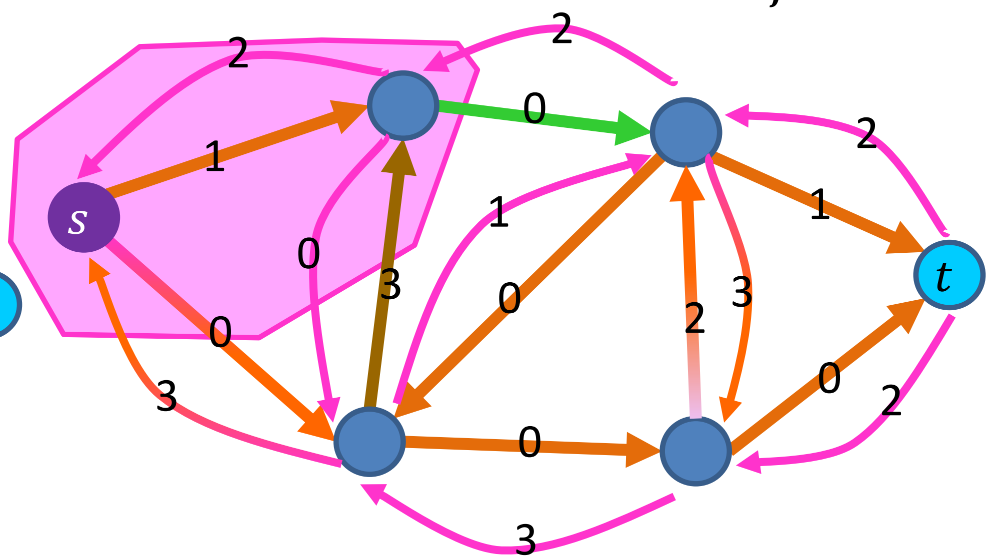
Proof: Maxflow/Mincut Theorem

- 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

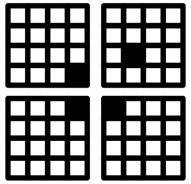
2. When Ford-Fulkerson terminates, there are no more augmenting paths in G_f

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

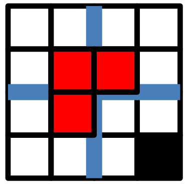
4. The sum of edge capacities crossing this cut must match the flow of the graph

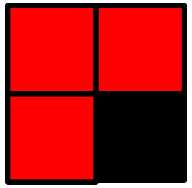
5. Therefore this flow is maximal

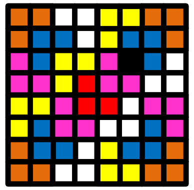
Divide and Conquer



- **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







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

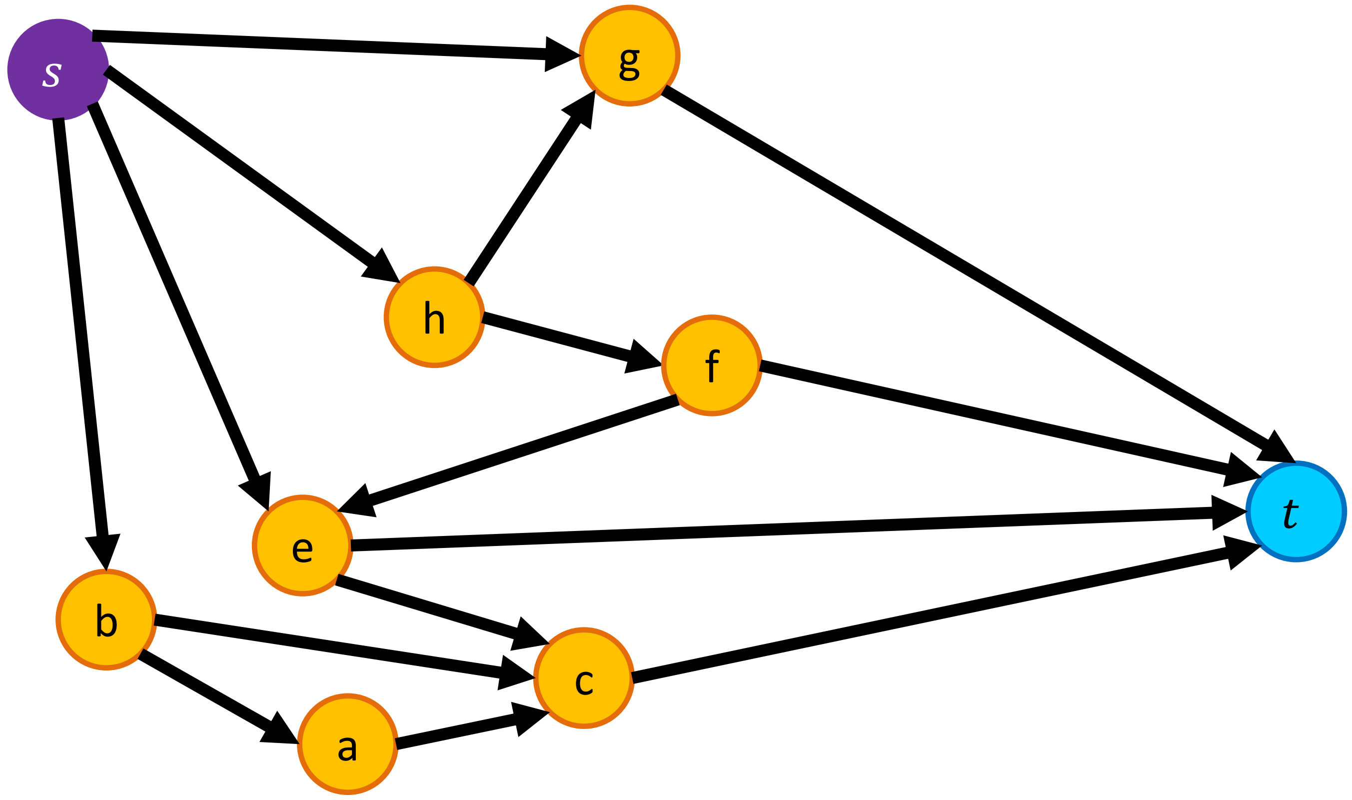
- Require Optimal Substructure
  - Solution to larger problem contains the solution to a smaller one
  - Only one subproblem to consider!
- Idea:
  1. Identify a greedy choice property
     - How to make a choice guaranteed to be included in some optimal solution
  2. Repeatedly apply the choice property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

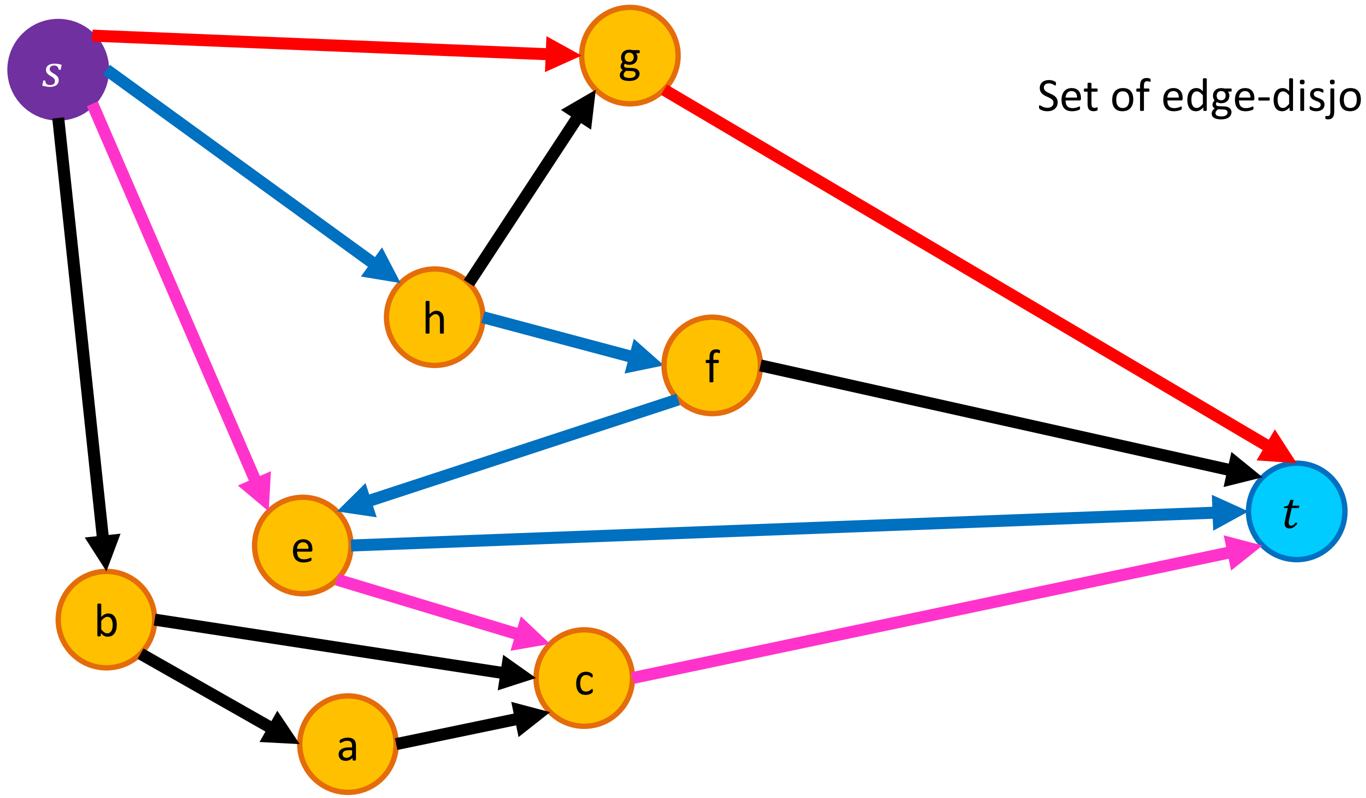
Edge-Disjoint Paths

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

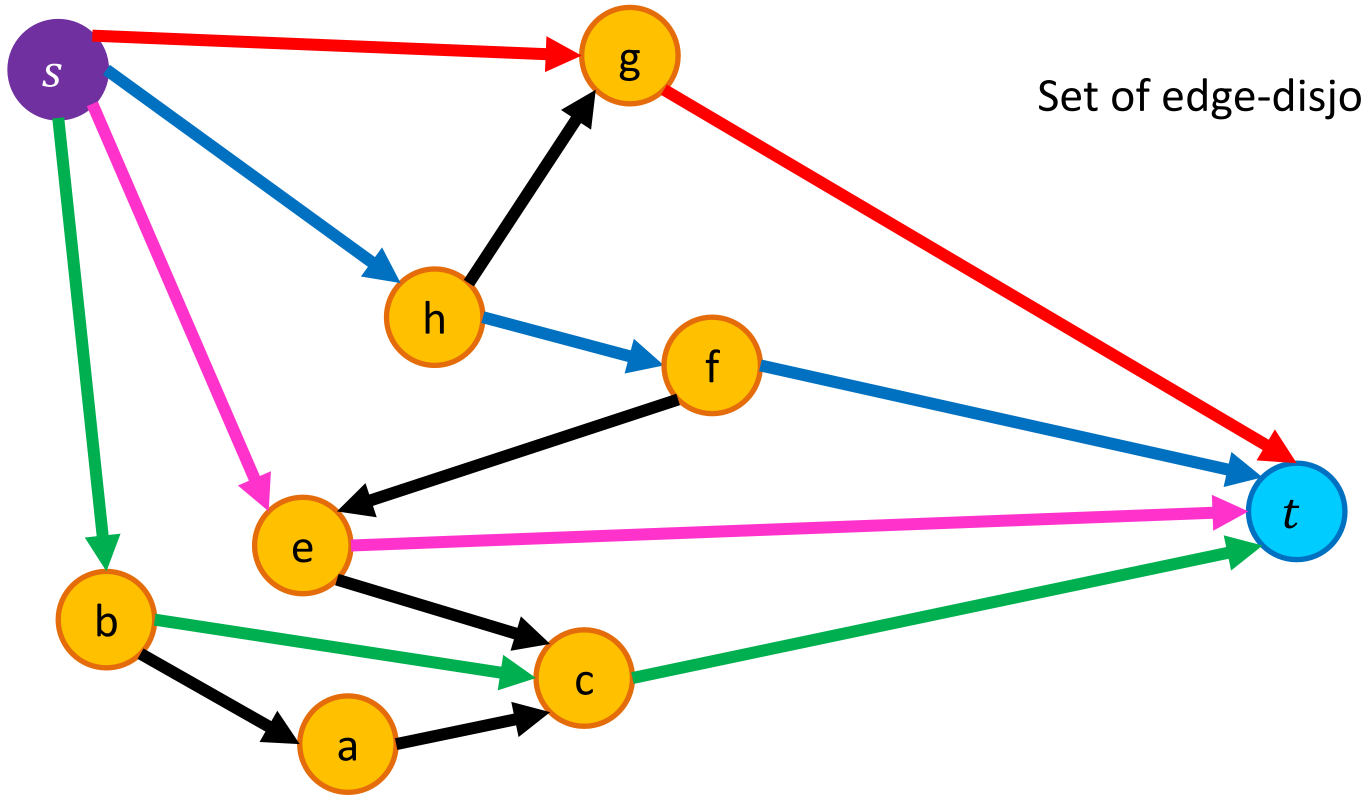
Given a graph $G = (V, E)$, a start node s and a destination node t, give the maximum number of paths from s to t which share no edges



Set of edge-disjoint paths of size 3

Edge-Disjoint Paths

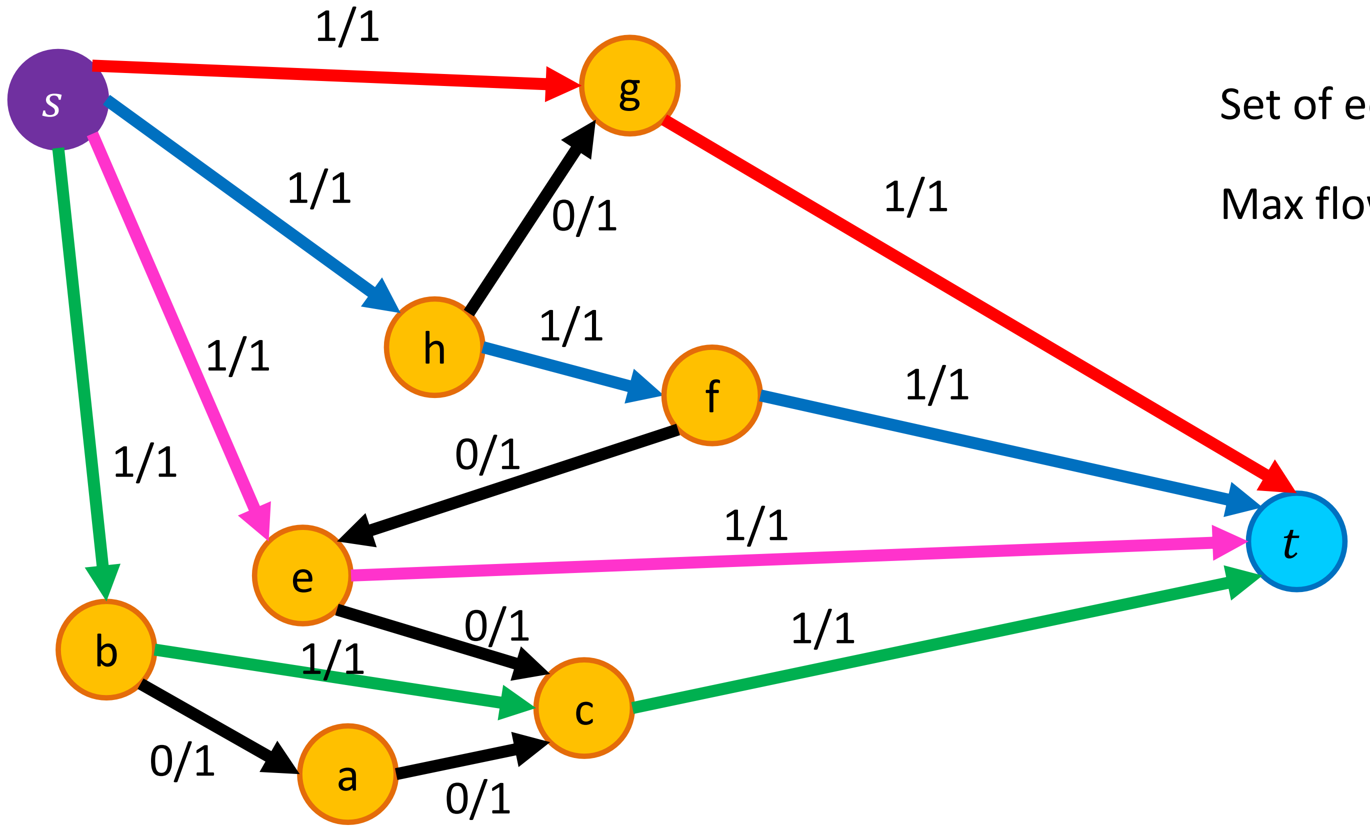
Given a graph $G = (V, E)$, a start node s and a destination node t, give the maximum number of paths from s to t which share no edges



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.

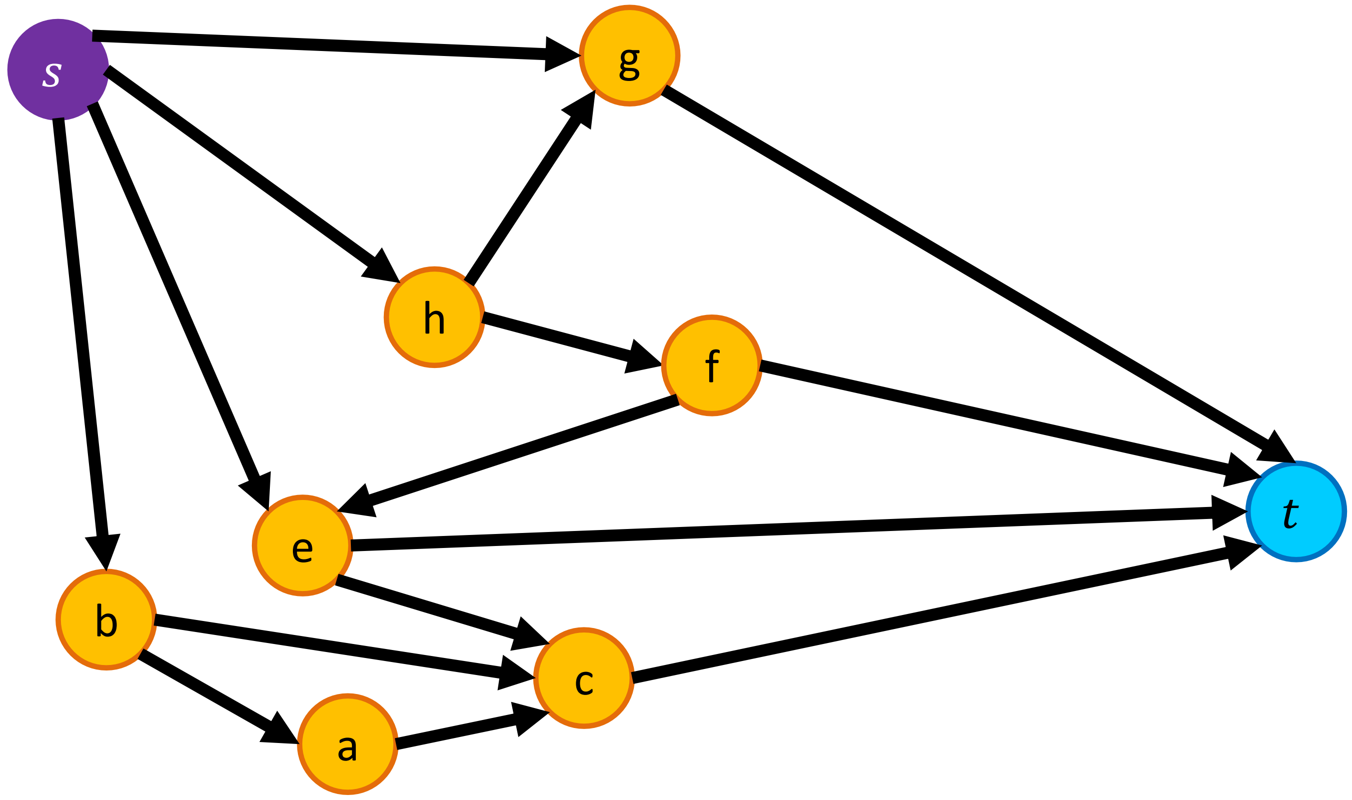


Set of edge-disjoint paths of size 4

Max flow = 4

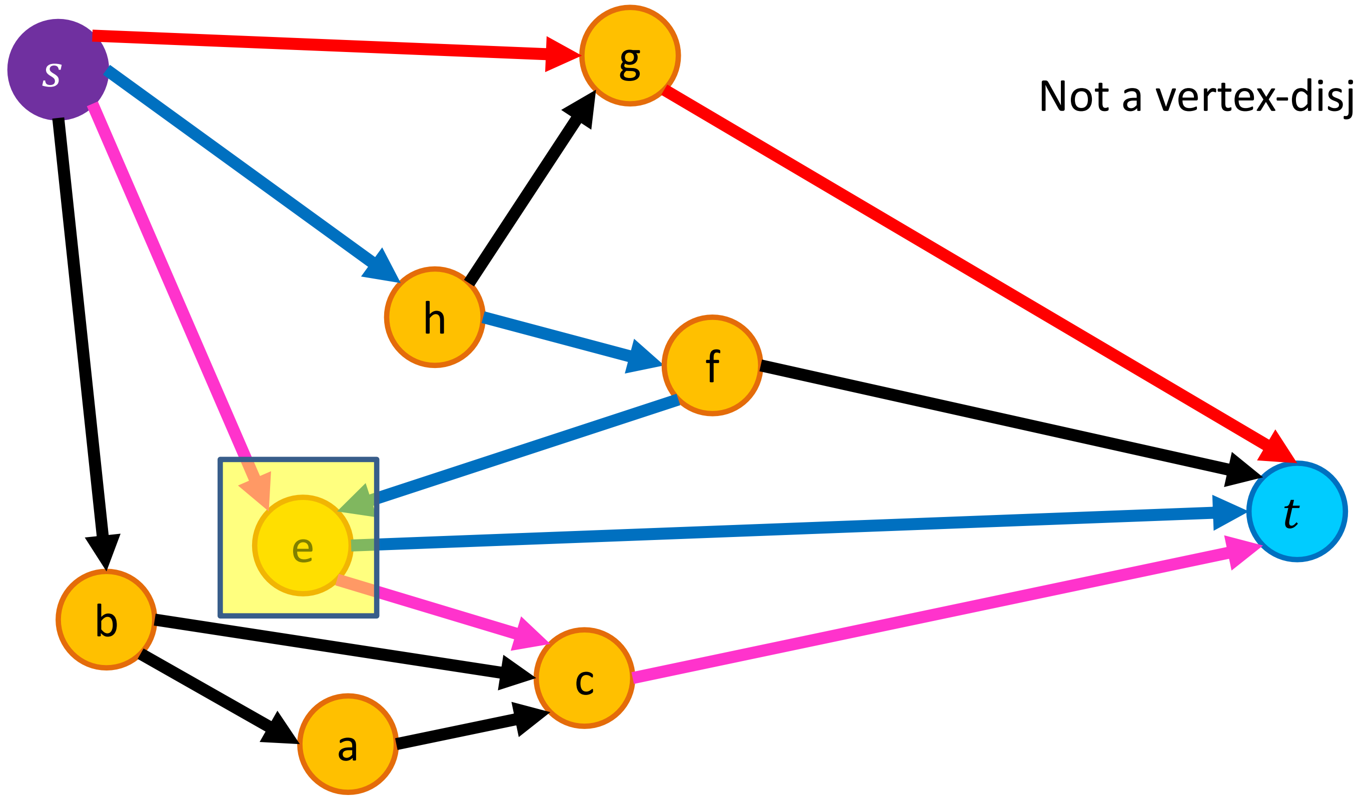
Vertex-Disjoint Paths

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