UVA DATA SCIENCE

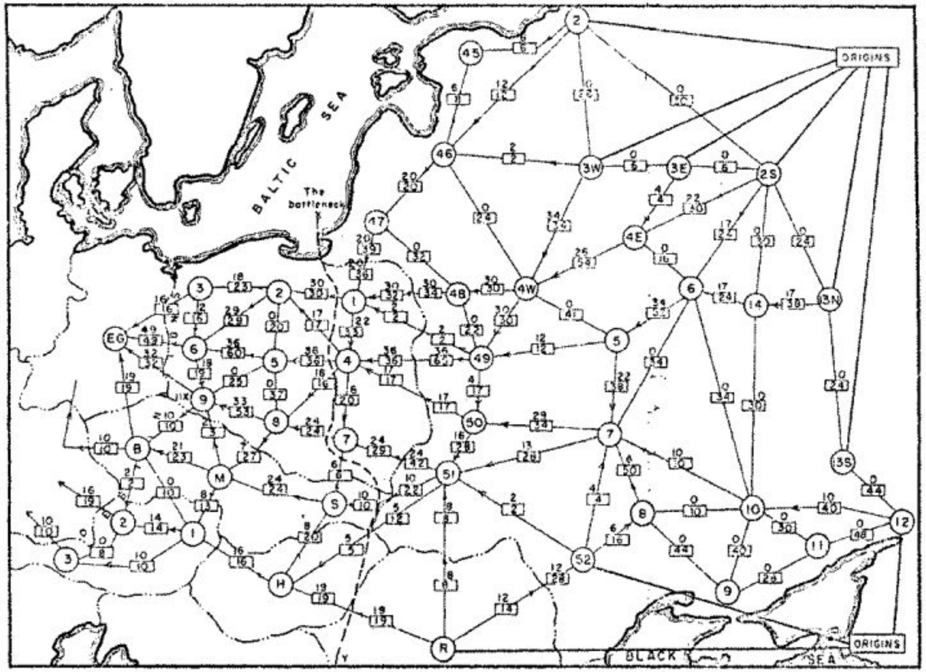
Foundation of Computer Science for Data Science

Classic CS
Maximum Flow

Mai Dahshan

November 11, 2024

Motivation

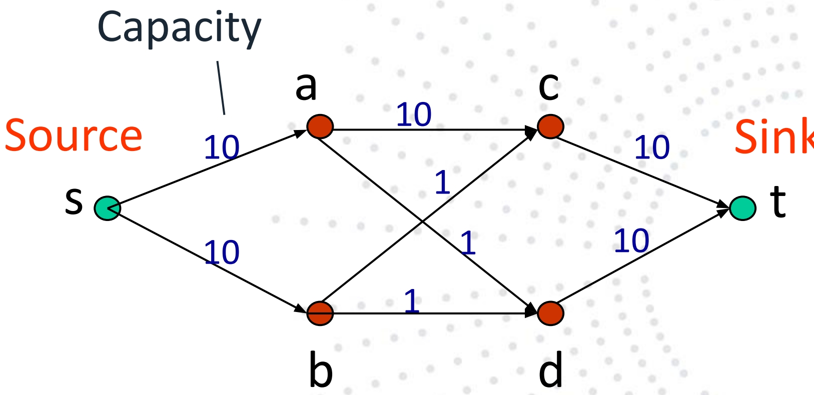


Soviet railway network, 1940

Find the **maximum** amount of cargo that can be **transported** from **sources** in the Western Soviet Union to **destinations** in Eastern Europe countries

Flow Network

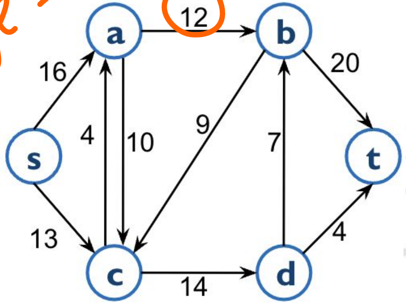
- A flow network is a directed graph **G**
- Edges represent pipes that carry flow
- Each edge **<u,v>** has a maximum capacity $\mathbf{c}_{<u,v>}$
- A source node **s** in which flow arrives
- A sink node **t** out which flow leaves



Flow Network

- The flow network problem is defined as follows:
  - Given a directed graph G with non-negative integer weights
    - Each edge stands for the capacity of that edge
  - Two different vertices, s and t, called the source and the sink
    - The source only has out-edges and the sink only has in-edges
  - Find the maximum amount of some commodity that can flow through the network from source to sink

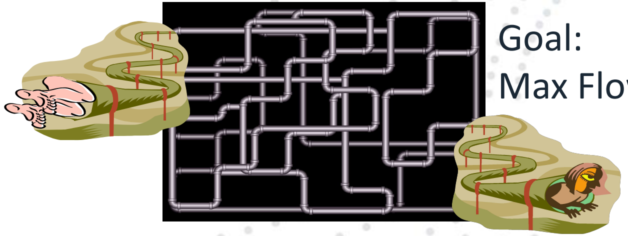
from s-a-b-t, the max you can transfer is the min edge



Each edge stands for the capacity of that edge.

Flow Network

- One way to imagine the situation is imagining each edge as a pipe that allows a certain flow of a liquid
  - The source is where the liquid is pouring from, and the sink is where it ends up.
  - Each edge weight specifies the maximal amount of liquid that can flow through that pipe per second.
  - Given that information, what is the most liquid that can flow from source to sink in the steady state?



Goal:
Max Flow

Flow Network Examples

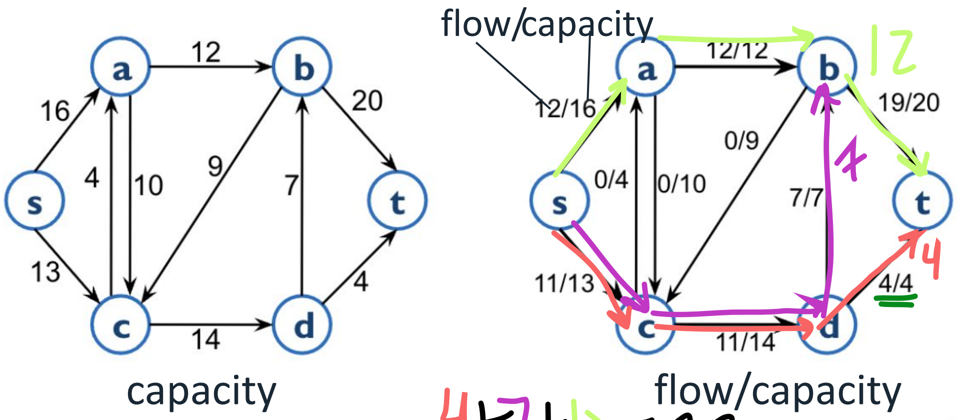
- **Transportation:** Modeling traffic on a network of roads, or the routing of packages by a company
- **Communication:** Routing packets in a communication network Air travel: Sequencing the legs of a flight
- **Railway systems:** Transporting goods across a railway system Vehicle routing: Finding the best routes for delivery trucks to minimize costs and time

Flow Graph

- **Flow graph** is a directed graph with distinguished vertices s (source) and t (sink)
- Capacities on the edges, c(e) >= 0
- Assign flows f(e) to the edges such that:
  - ***Capacity Rule:*** 0 <= f(e) <= c(e)
  - ***Conservation Rule:*** Flow is conserved at vertices other than s and t
    - Flow conservation: flow going into a vertex equals the flow going out
  - The flow leaving the source is a large as possible

Flow Graph

The ***flow of the network*** is defined as the flow from the source, or into the sink.



$4+7+12=23$

whatever comes in or out of the node needs to be the same

The network flow is 23

min Flow from s-a-b-t is 12

min flow from s-c-d-t is 4

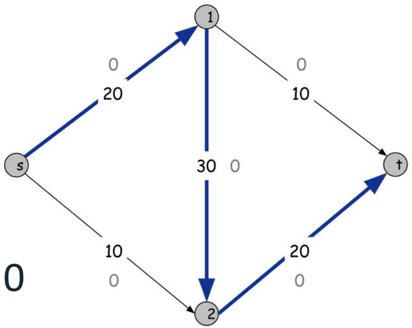
Flow Graph

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

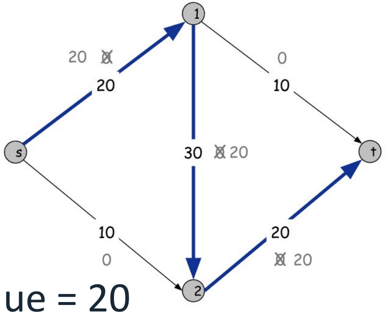
$$|f| = \text{outflow}(s) - \text{inflow}(s)$$

Greedy Approach

- Start with f(e) = 0 for all edge e ∈ E
- Find an s-t path P where each edge has max f(e) , where f(e) < c(e)
- Augment flow along path P
- Repeat until you get stuck



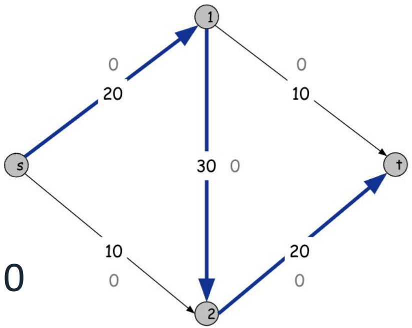
flow value = 0



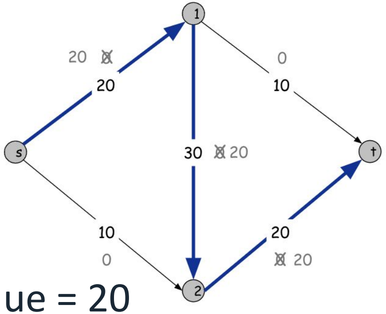
flow value = 20

Greedy Approach

- Start with f(e) = 0 for all edge e ∈ E
- Find an s-t path P where each edge has max f(e) , where f(e) < c(e)
- Augment flow along path P
- Repeat until you get stuck



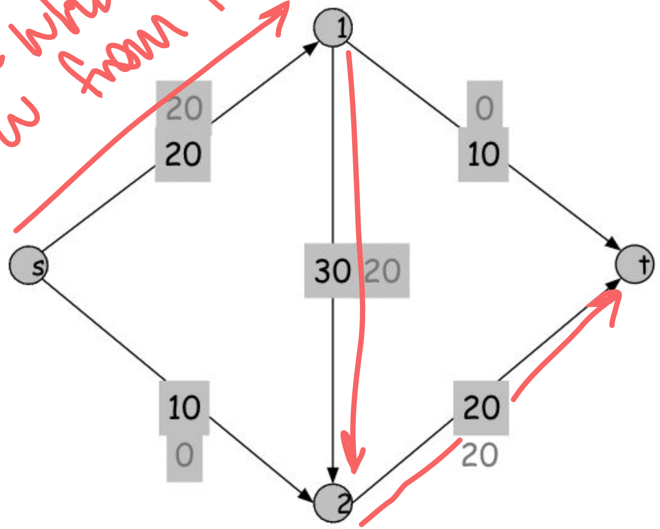
flow value = 0



flow value = 20

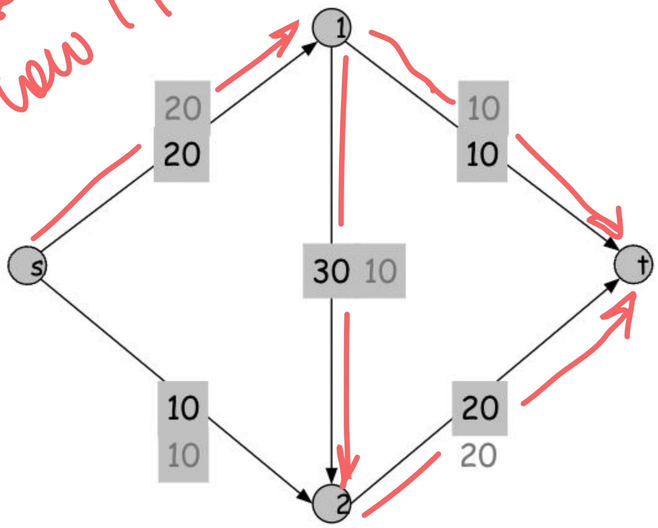
Greedy Approach

Pass the whole flow from 1 to 2



greedy = 20

Pass half the flow (1→2 & 1→t)



optimal (Max flow)= 30

locally optimality != global optimality

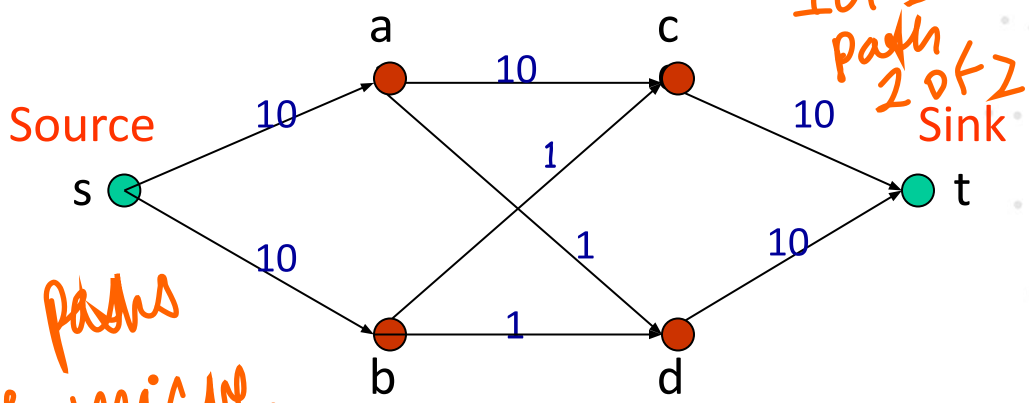
Greedy looks for local optimality, not global optimality

Greedy Approach

Using greedy

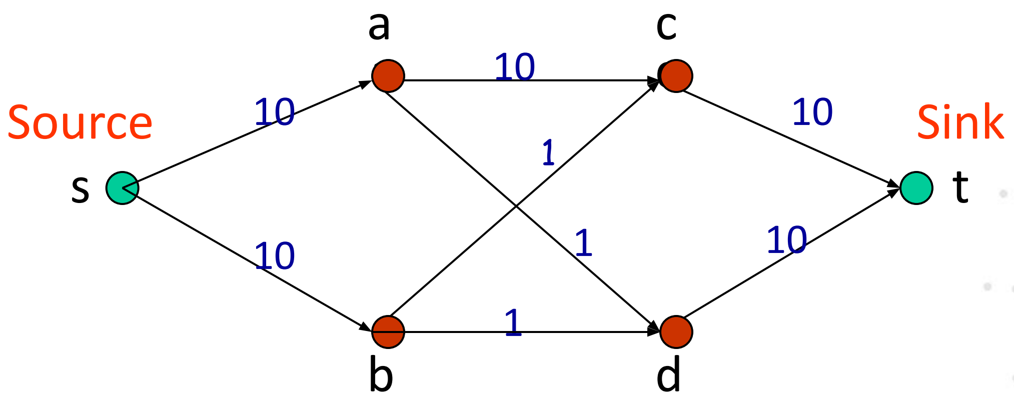
path 1 of 2 s-a-c-t } 10

path 2 of 2 s-b-d-t } 1

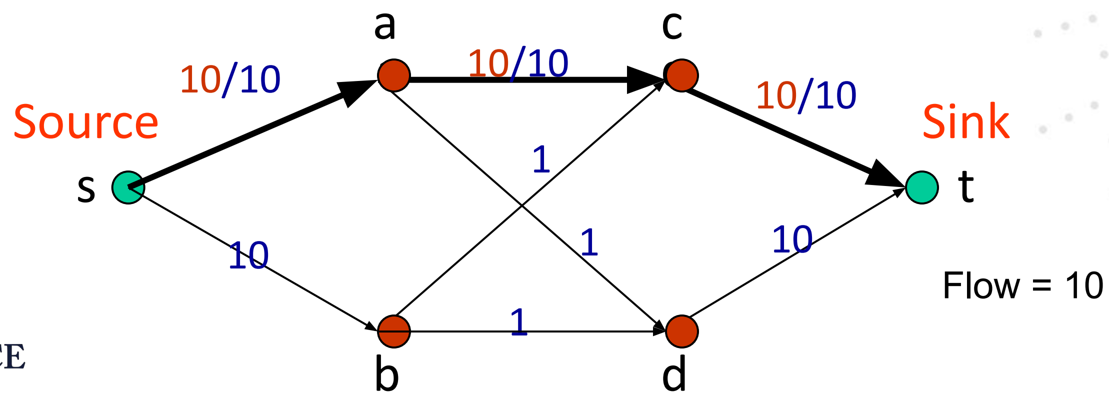


note: the paths have to be unique.
I wouldn't choose path s-b-c-t b/c c-t was already taken in s-a-c-t

Greedy Approach

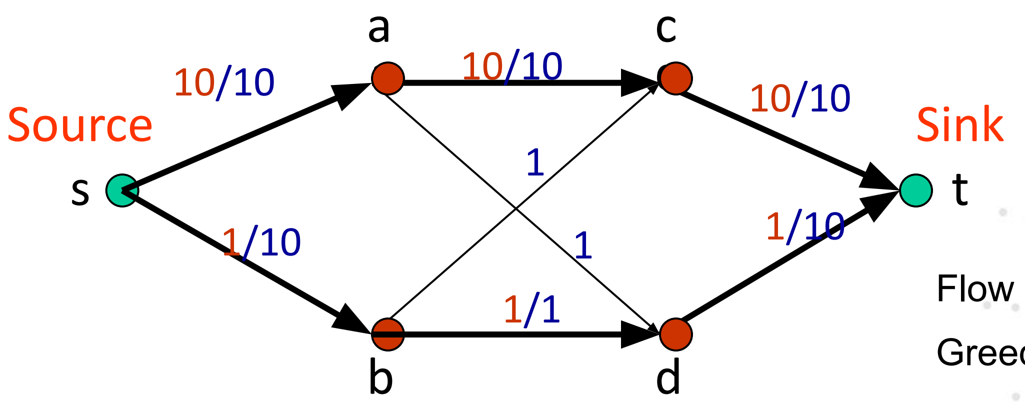


Step 1:



Greedy Approach

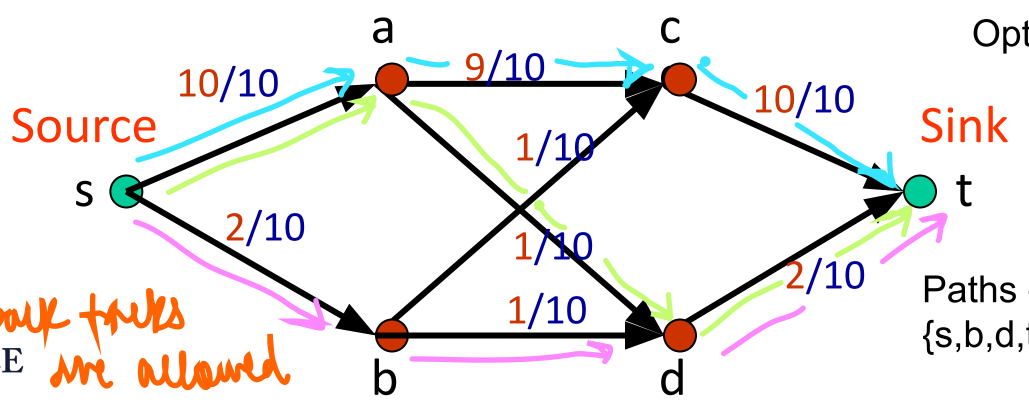
Step 2:



Flow = 10 + 1 = 11

Greedy = = 11

Max Flow:



Optimal = = 12

10 + 1 + 1 = 12

aka, back track

→ back tracks are allowed

Paths {s,a,c,t}, {s,a,d,t}, {s,b,c,t}, and {s,b,d,t}

Ford-Fulkerson Algorithm

- The **Ford-Fulkerson algorithm** is a classic method for finding the **maximum flow** in a flow network. It uses the concept of **augmenting paths** and operates on the **residual graph** to iteratively improve the flow until no more augmenting paths exist

Residual Graphs have forward & backward arrows

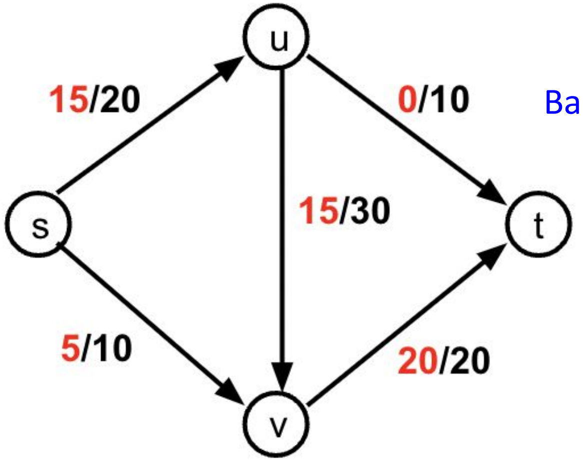
Ford-Fulkerson Algorithm

- Residual graph shows the remaining capacity
- 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 weights set to remaining capacity c(e)-f(e)

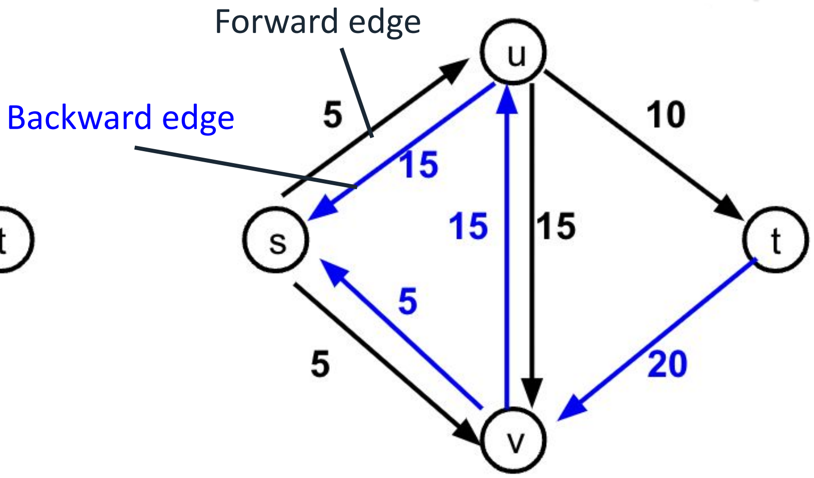
    forward: how many unused arrows are there?
    - models **additional flow** that can be sent along edge
  - **Backward edge** by flipping each edge e in *G* with weight set to the flow f(e) models amount of flow that can be removed from the edge

    backward: how many points can I remove from the graph

Ford-Fulkerson Algorithm



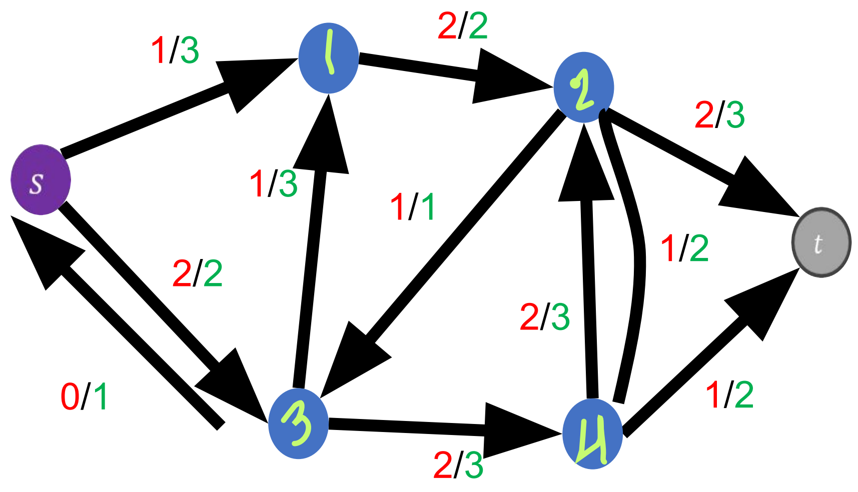
Flow graph G



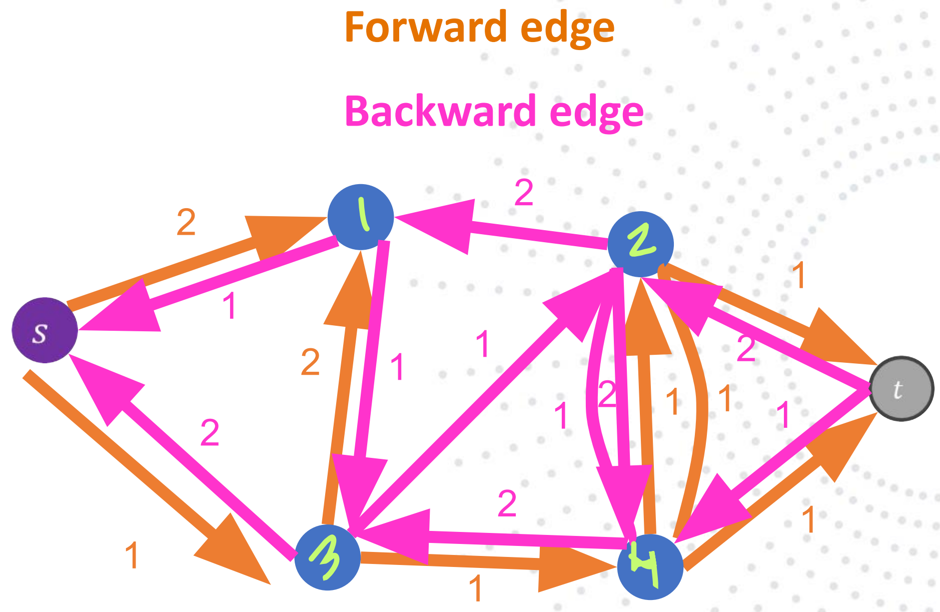
Residual Graph G_f

forward: $c(e) - f(e)$

Ford-Fulkerson Algorithm



Flow graph G



Residual Graph G_f

The residual graph tries to help you determine what is the max capacity in your graph

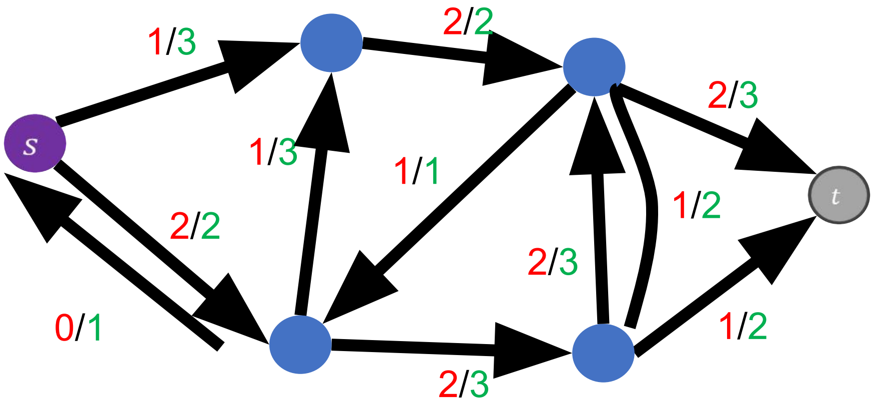
Ford-Fulkerson Algorithm

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

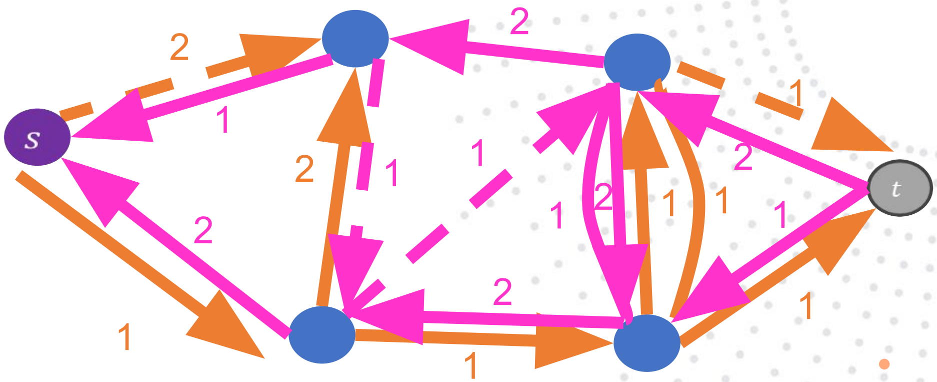
Consider the minimum-weight edge e along the path: we can increase the flow by $w(e)$

- Send $w(e)$ flow along all forward edges (these have at least $w(e)$ capacity)

augmented paths in residual graph (dashed arrows)



Flow graph G

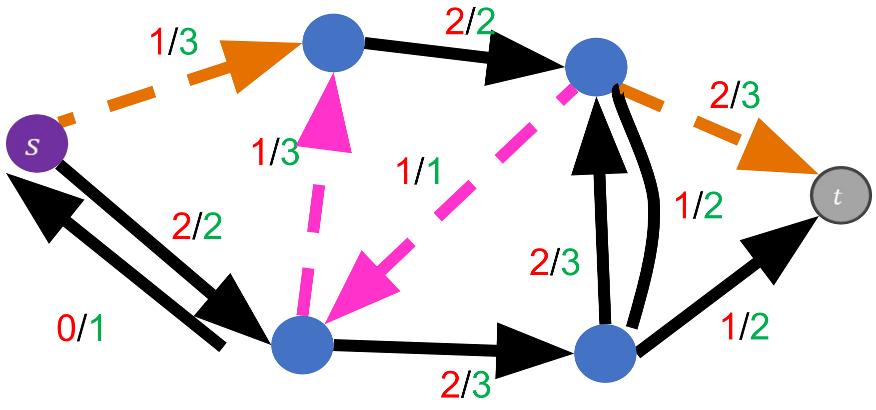


Residual Graph G_f

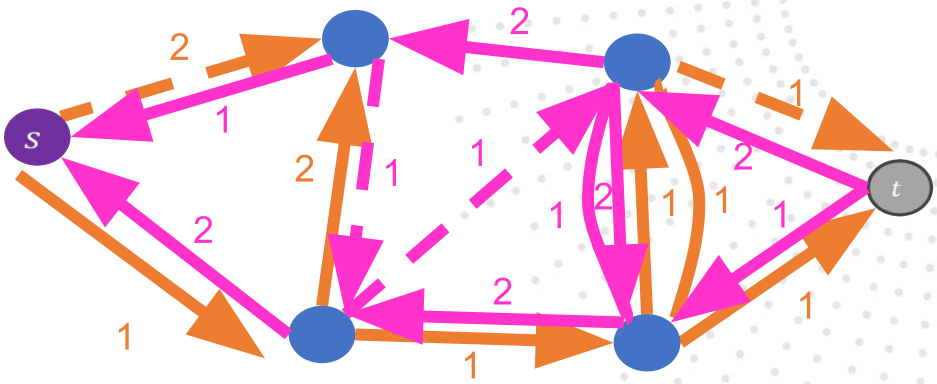
Ford-Fulkerson Algorithm

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)



Flow graph G



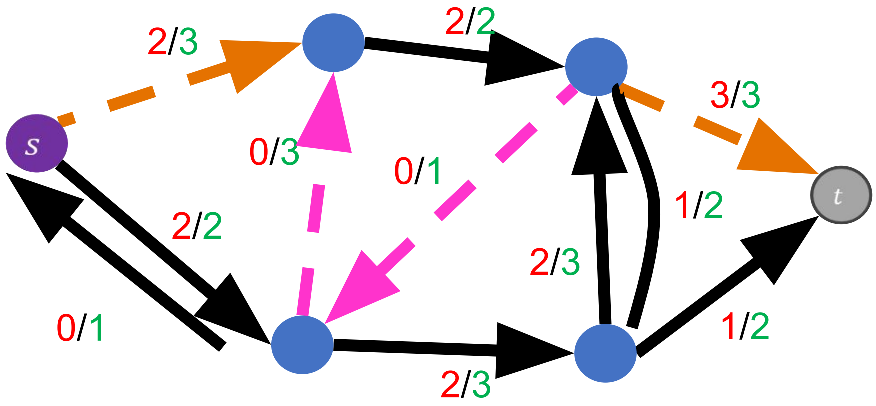
Residual Graph G_f

Ford-Fulkerson Algorithm

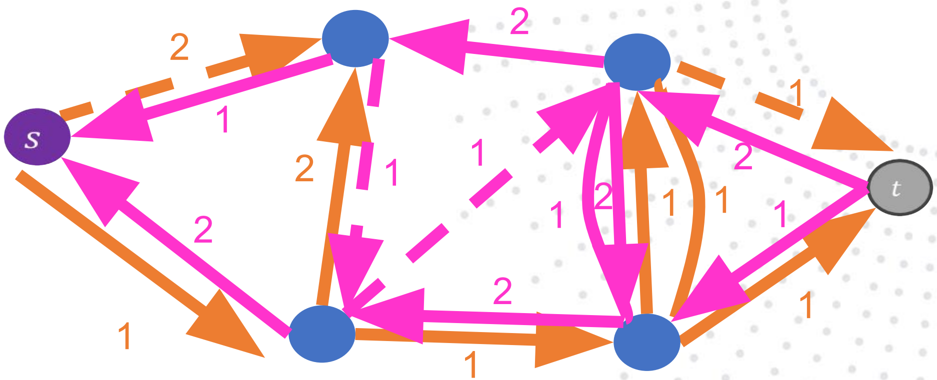
Consider a path from $s \rightarrow t$ in G_f using only edges with positive (non-zero) weight
Consider the minimum-weight edge e along the path: we can increase the flow by $w(e)$

- 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 graph 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)

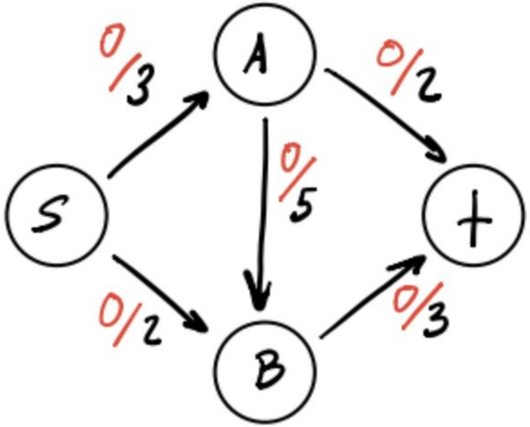
Ford-Fulkerson max-flow algorithm:

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

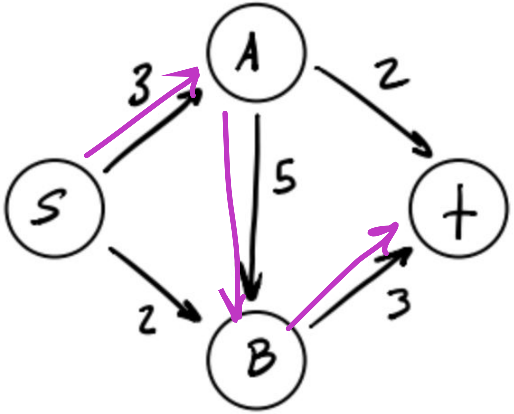
Ford-Fulkerson approach:
take any augmenting path

Ford-Fulkerson Algorithm- Example

- Initially set the flow along every edge to 0
- Construct a residual graph for this network. It should look the same as the input flow network



Flow graph



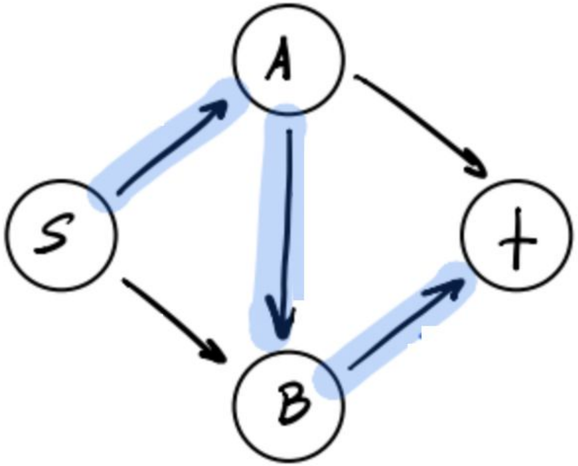
Residual graph

only forward edges

min flow is 3

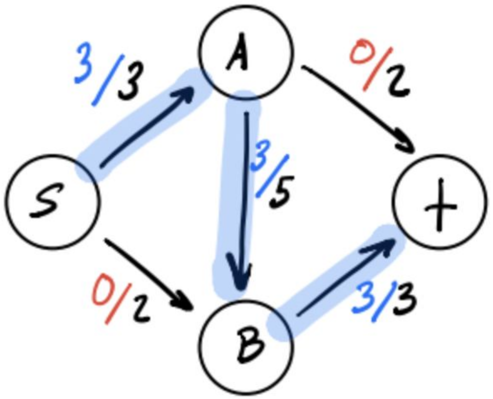
Ford-Fulkerson Algorithm- Example

- Use a pathfinding algorithm like (DFS) or (BFS) to find a path *P* from *s* to *t* that has available capacity **in the residual graph**



Ford-Fulkerson Algorithm- Example

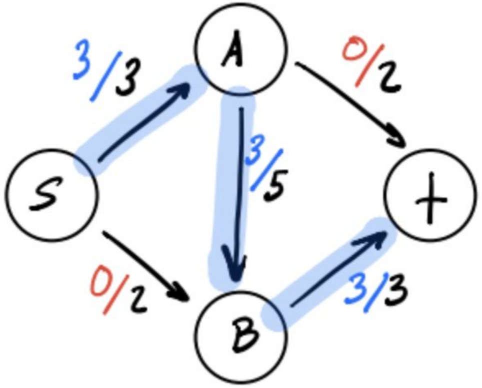
- Let *cap(P)* indicate the maximum amount of stuff that can flow along this path
  - To find the capacity of this path, we need to look at all edges *e* on the path and subtract their current flow, from their capacity. We'll set *cap(P)* to be equal to the smallest value since this will **bottleneck the path**

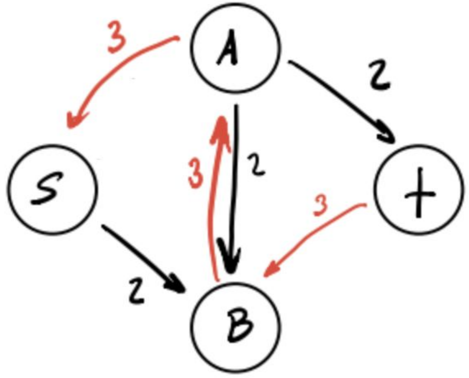


3 is the bottleneck for this the path

Ford-Fulkerson Algorithm- Example

- We then **augment the flow** across the forward edges in the path *P* by adding *cap(P)* value. For flow across the back edges in the residual graph, we subtract our *cap(P)* value
- Update the residual graph with these flow adjustments



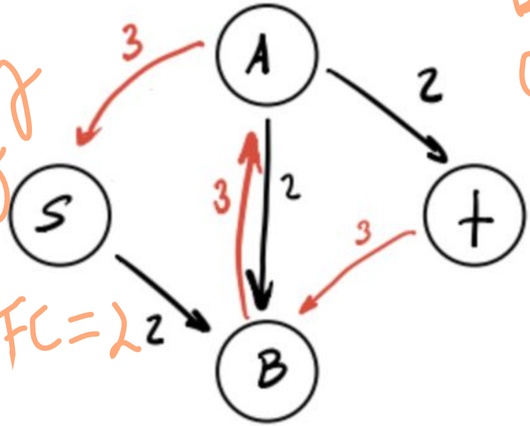


Ford-Fulkerson Algorithm- Example

- Search through the updated residual graph for a new *s-t* path. There are no forward edges available anymore, but we can use a back edge to augment the current flow

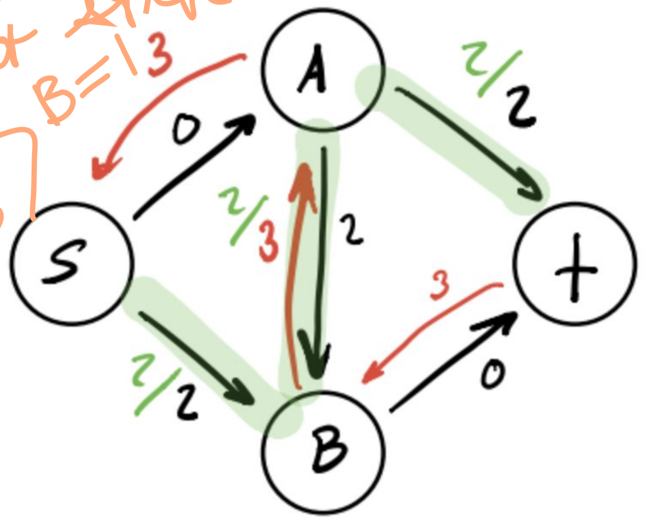
Forward remaining capacity

ex: 1/3 > FC=2



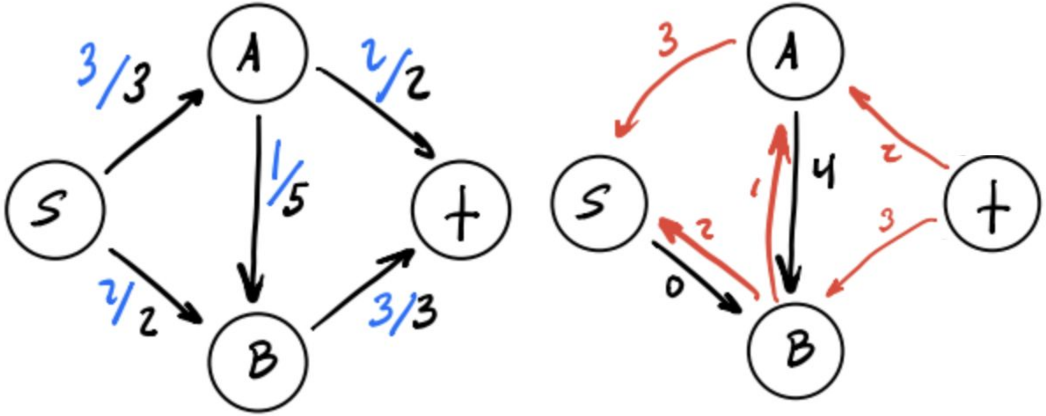
Backward: current ~~state~~

ex: 1/3] B=1

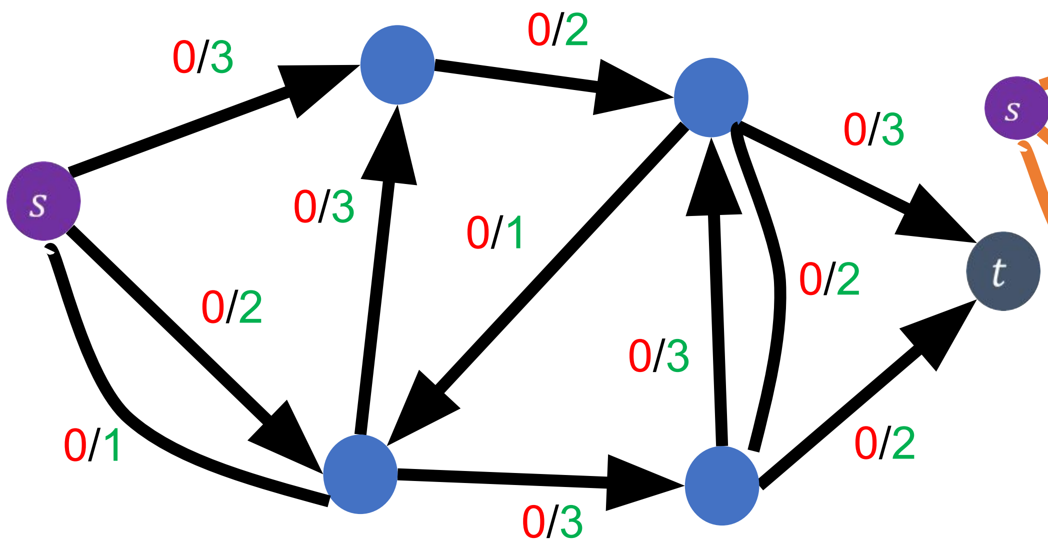


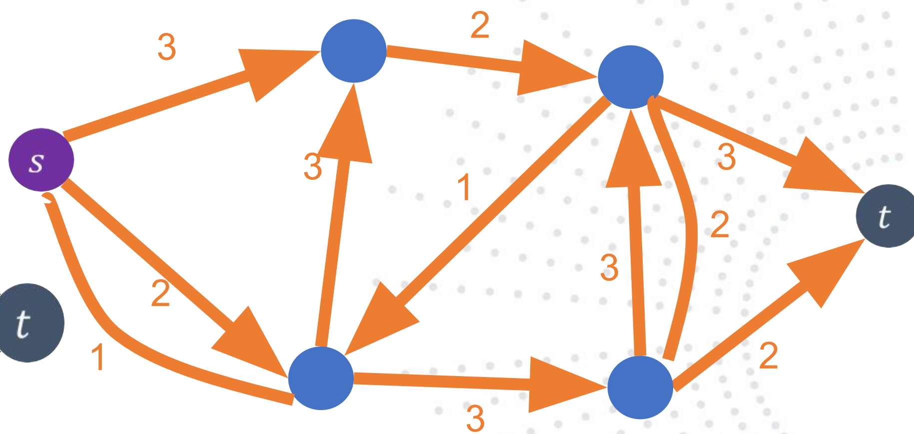
Ford-Fulkerson Algorithm- Example

- There are now no edges with available capacity that we can use to create a path from *s* to *t*. This means our run of the Ford-Fulkerson algorithm is complete and our max flow leading into *t* is 5



Ford-Fulkerson Algorithm- Example



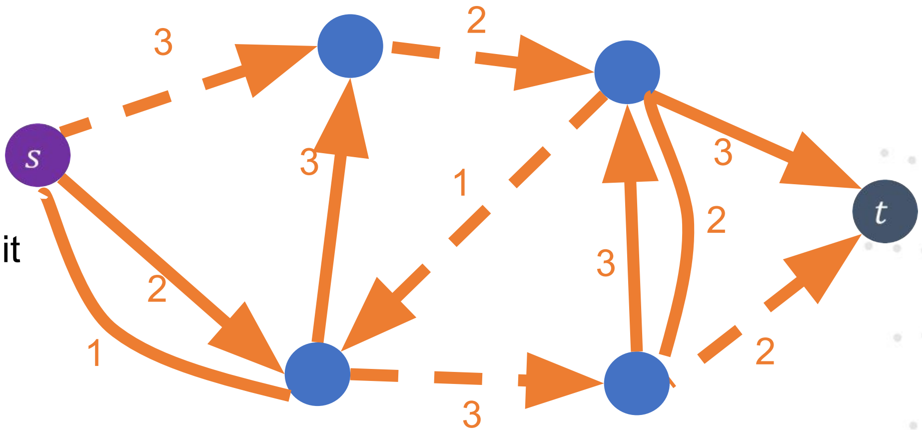


Residual graph G_f

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

Ford-Fulkerson Algorithm- Example

Increase flow by 1 unit

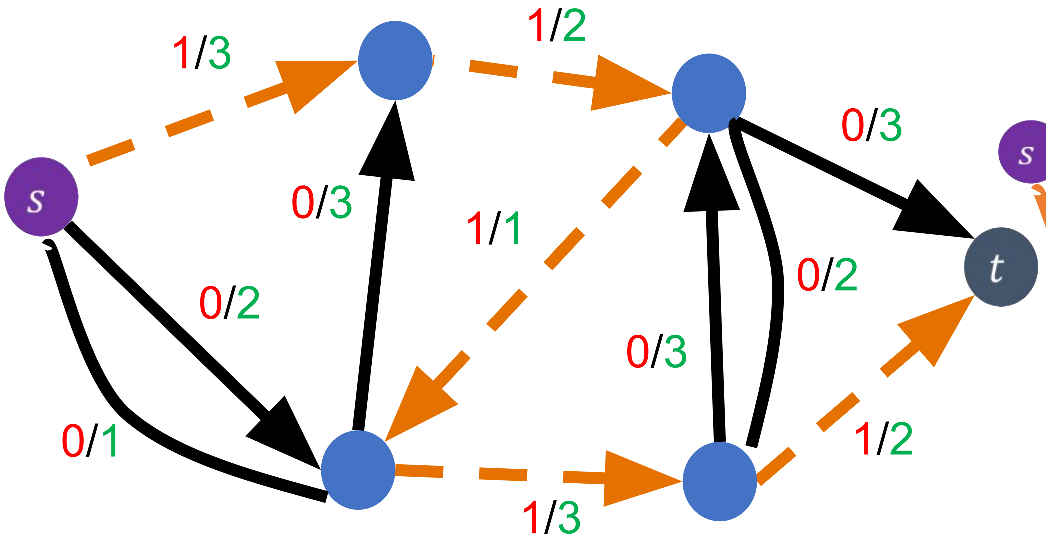


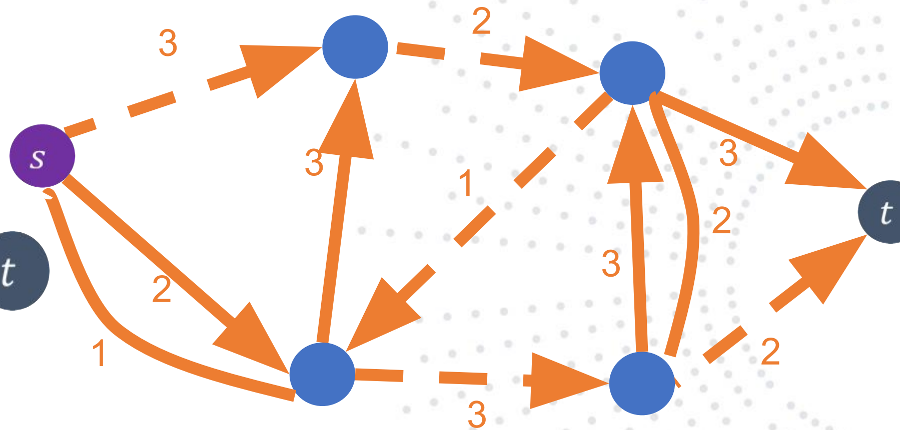
Residual graph G_f

31

Ford-Fulkerson Algorithm- Example

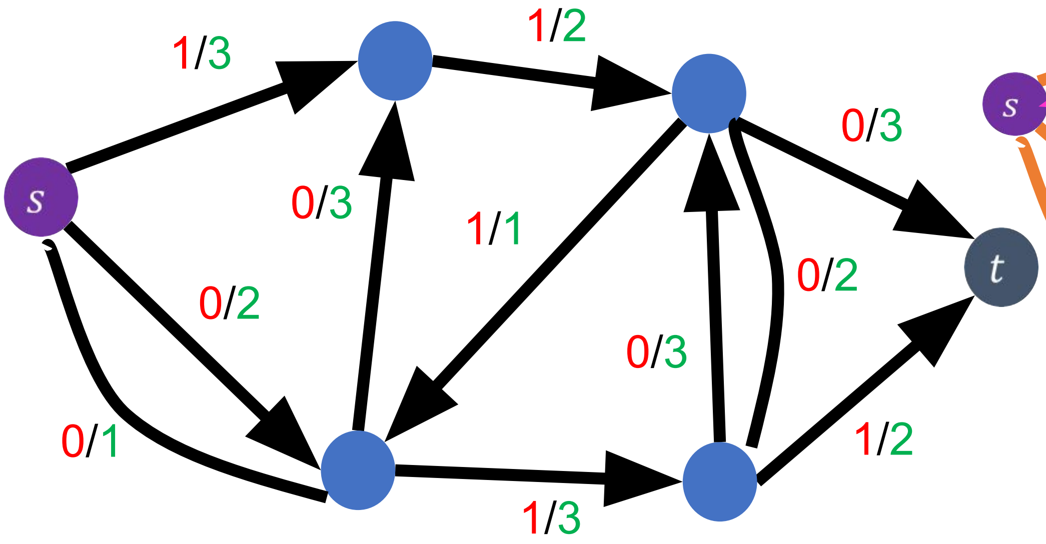
Increase flow by 1 unit

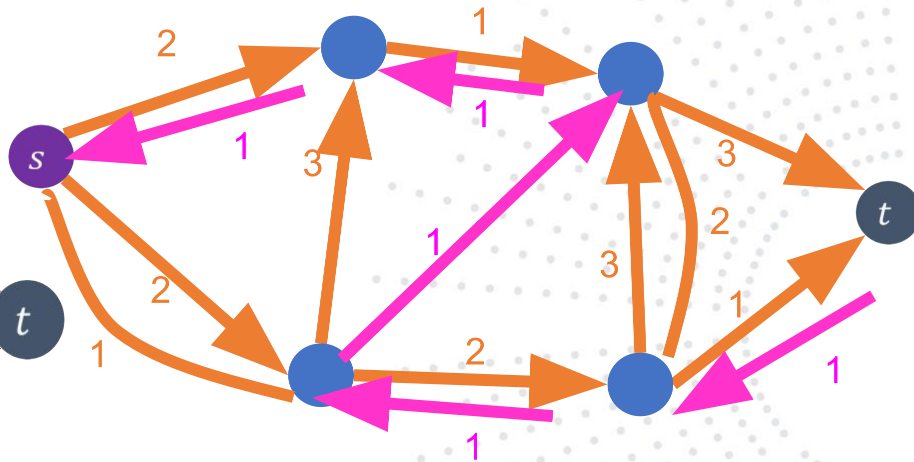




Residual graph G_f

Ford-Fulkerson Algorithm- Example



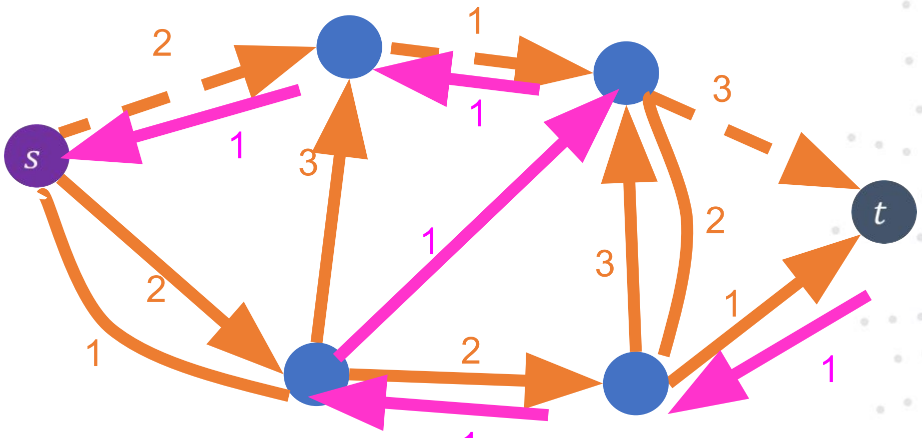


Residual graph G_f

33

Ford-Fulkerson Algorithm- Example

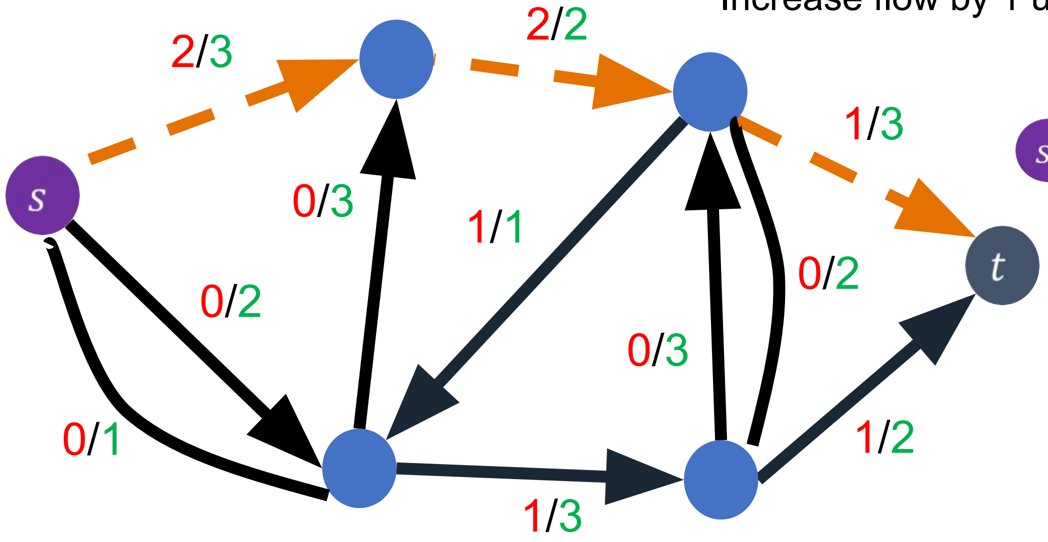
Increase flow by 1 unit

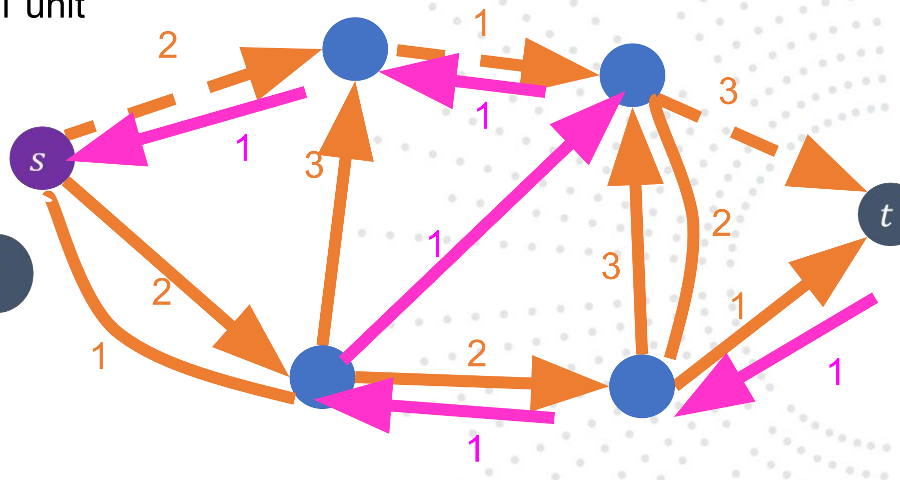


Residual graph G_f

Ford-Fulkerson Algorithm- Example

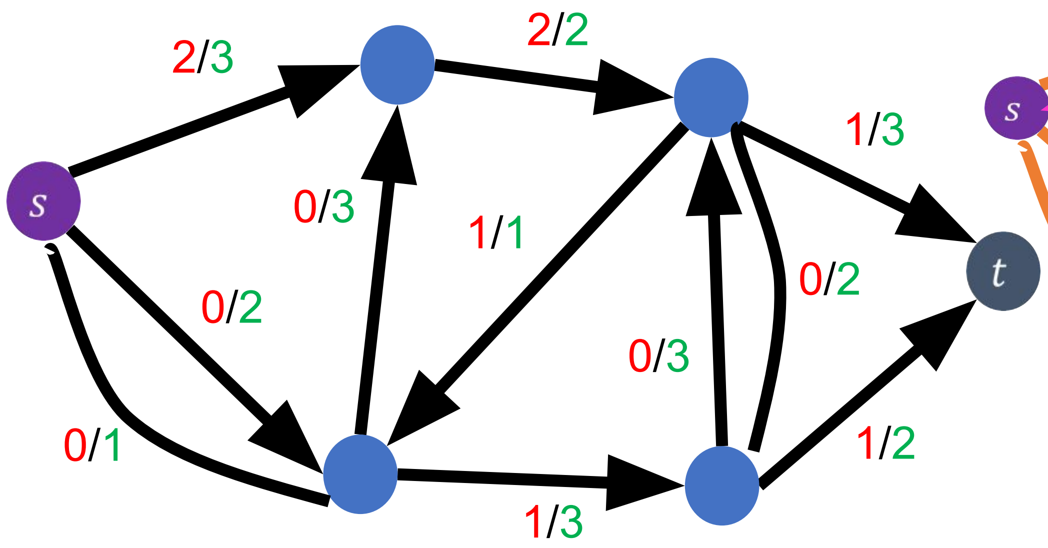
Increase flow by 1 unit

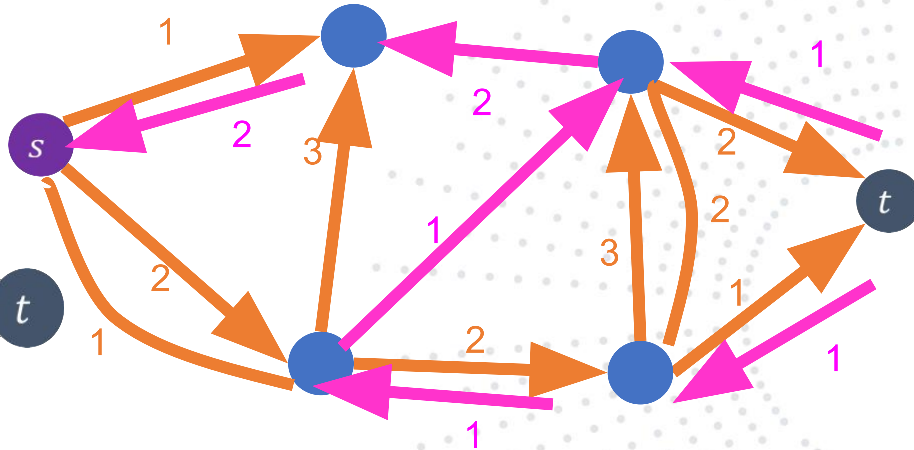




Residual graph G_f

Ford-Fulkerson Algorithm- Example



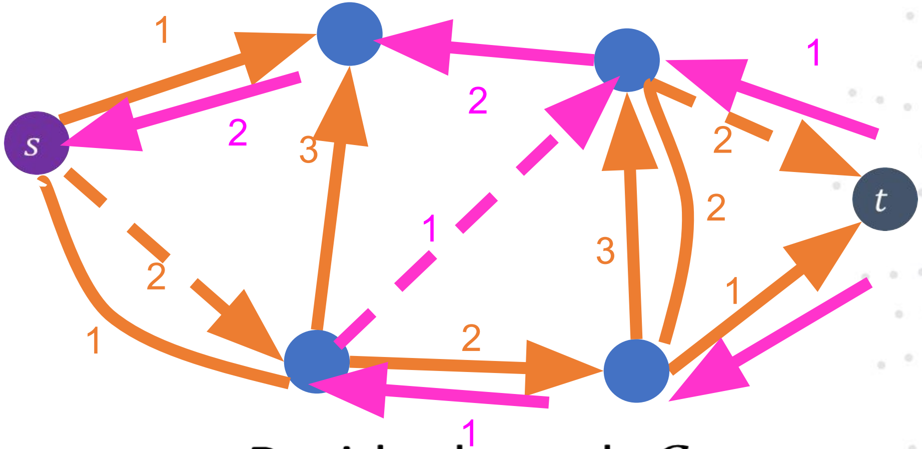


Residual graph G_f

36

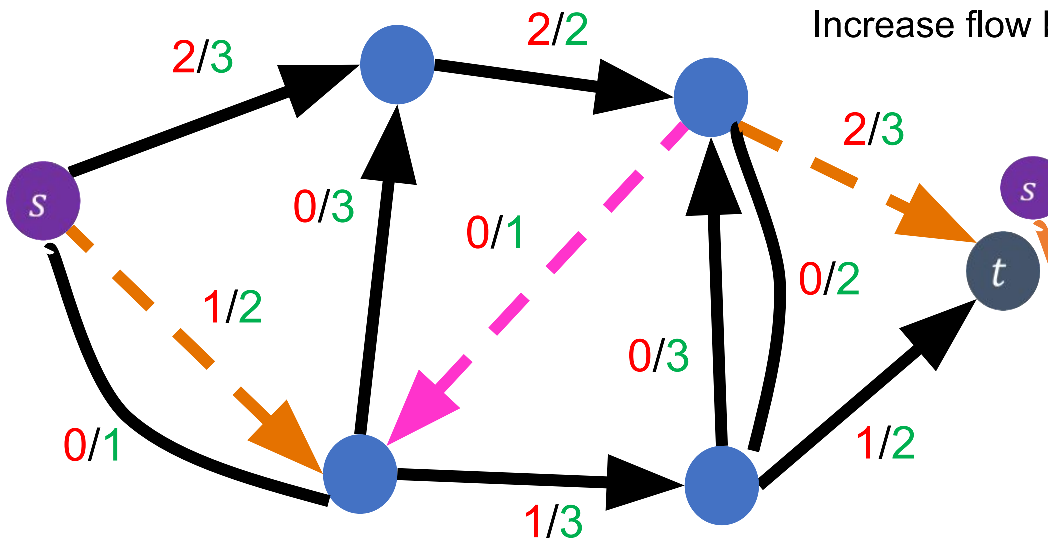
Ford-Fulkerson Algorithm- Example

Increase flow by 1 unit

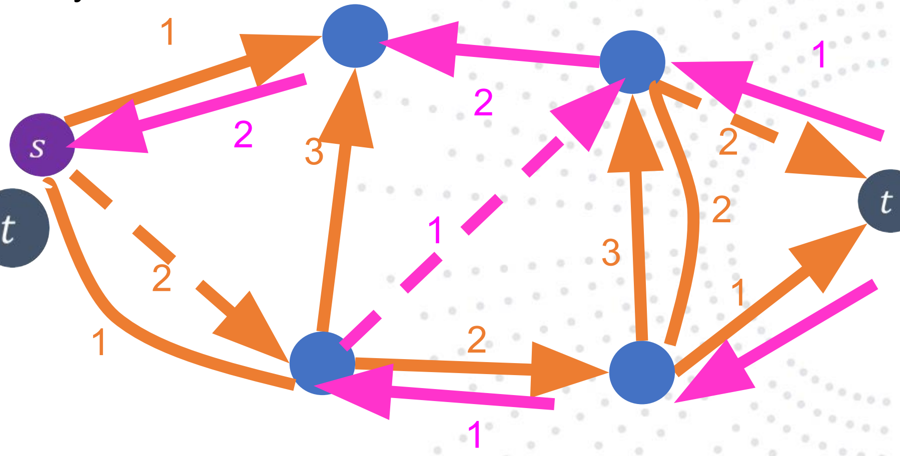


Residual graph G_f

Ford-Fulkerson Algorithm- Example

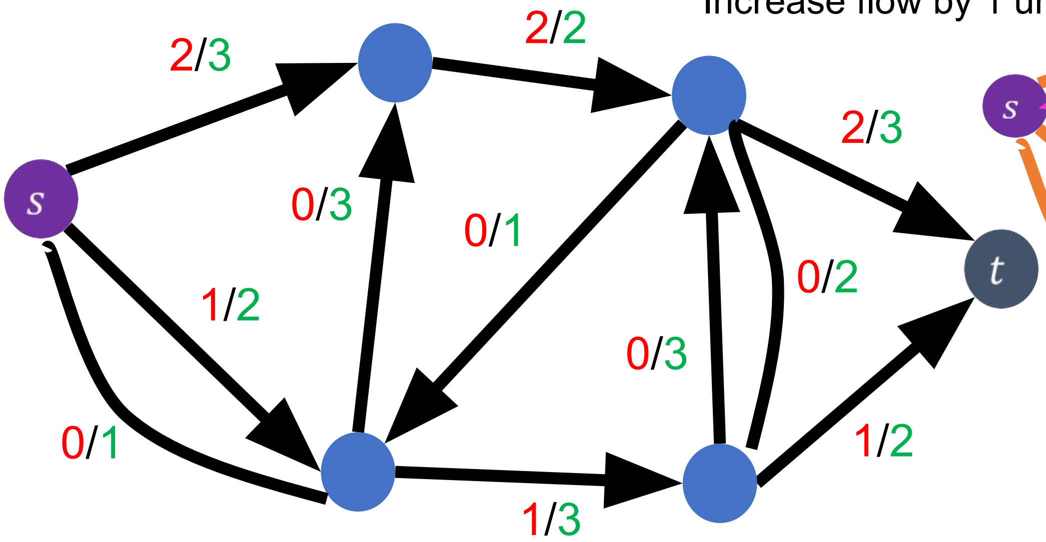


Increase flow by 1 unit

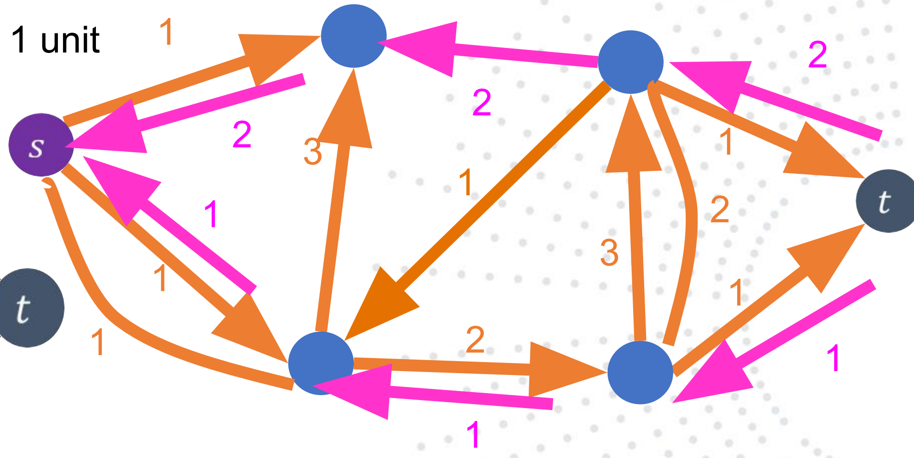


Residual graph G_f

Ford-Fulkerson Algorithm- Example

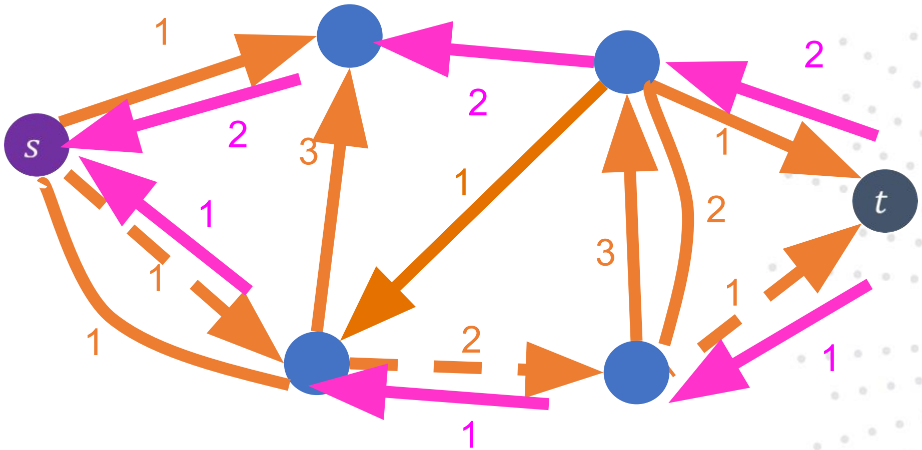


Increase flow by 1 unit



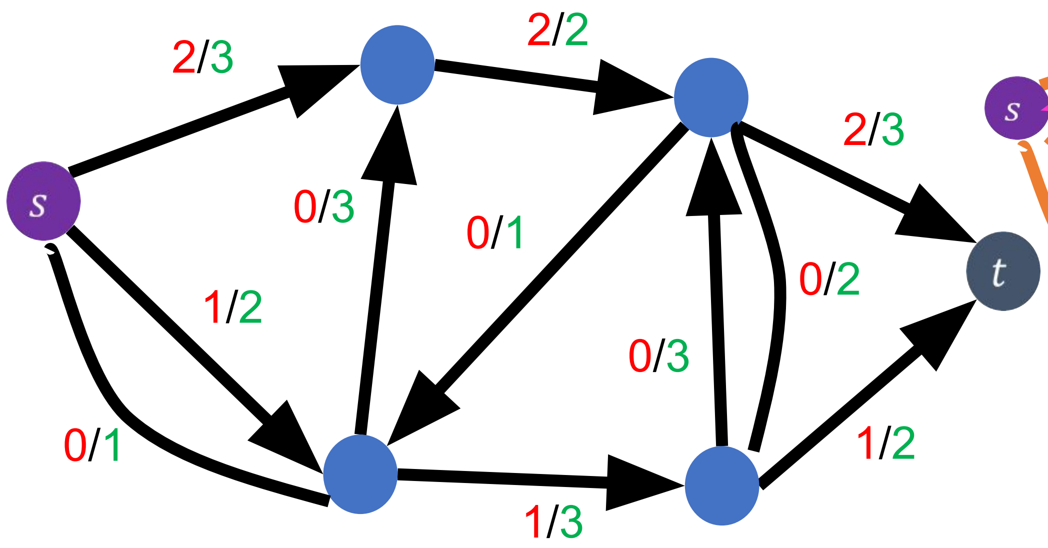
Residual graph G_f

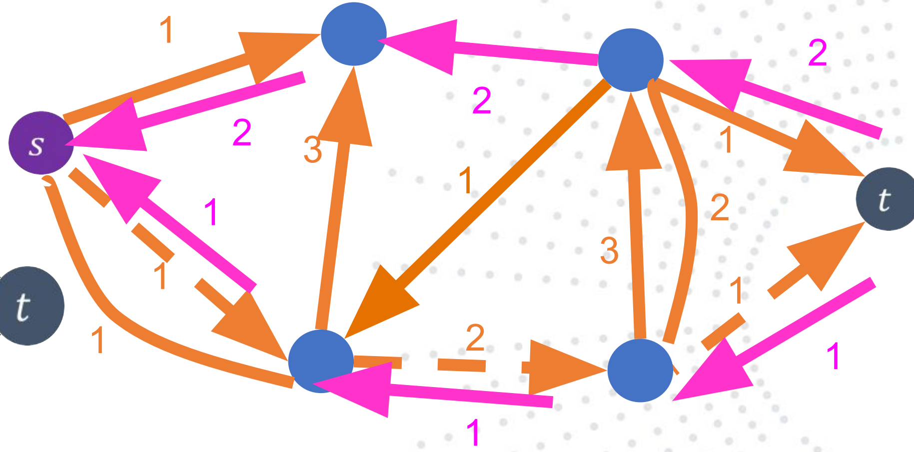
Ford-Fulkerson Algorithm- Example



Residual graph G_f

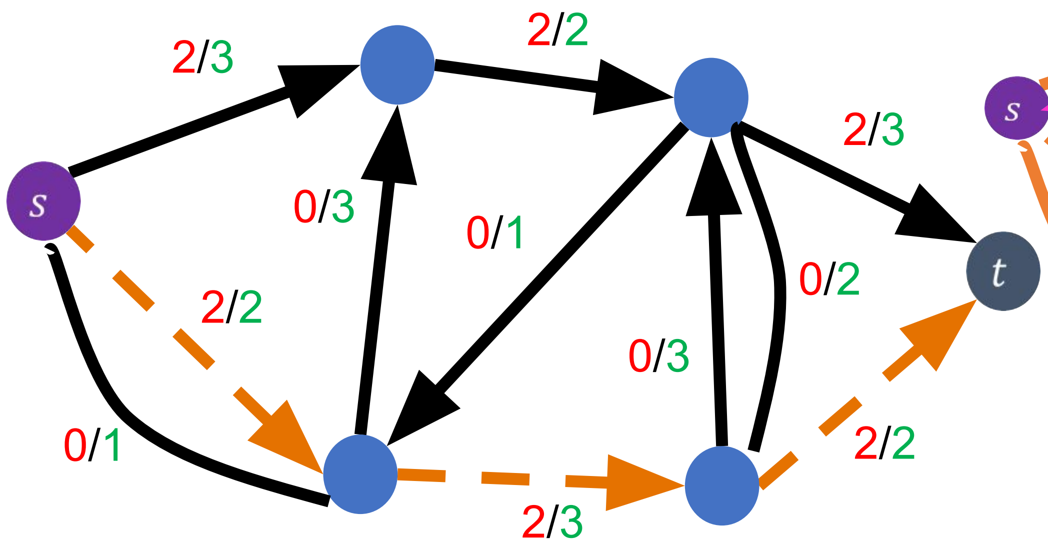
Ford-Fulkerson Algorithm- Example

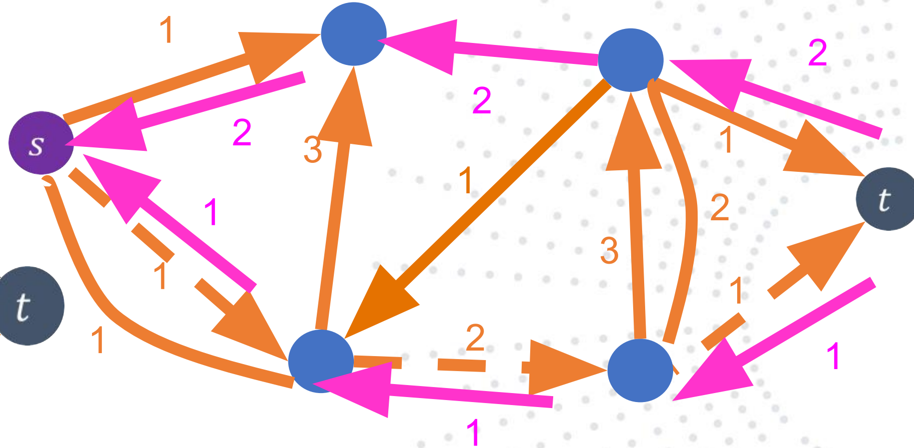




Residual graph G_f

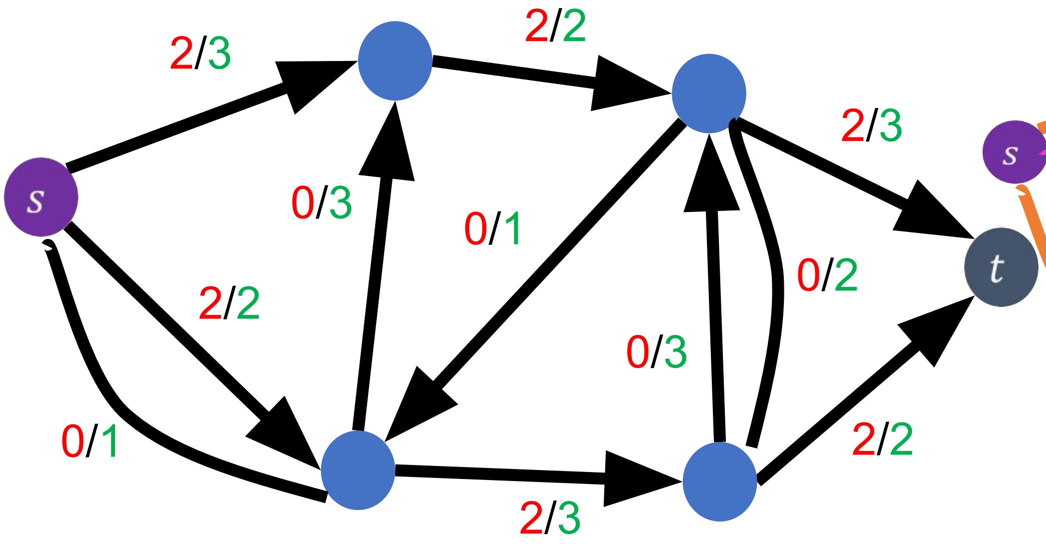
Ford-Fulkerson Algorithm- Example



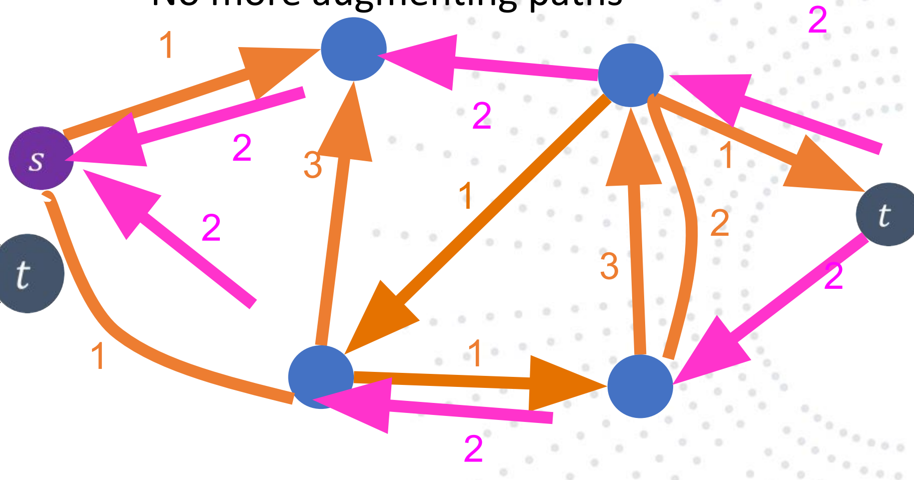


Residual graph G_f

Ford-Fulkerson Algorithm- Example



No more augmenting paths



Residual graph G_f

Maximum flow: 4

Ford-Fulkerson Algorithm

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

Ford-Fulkerson max-flow algorithm:

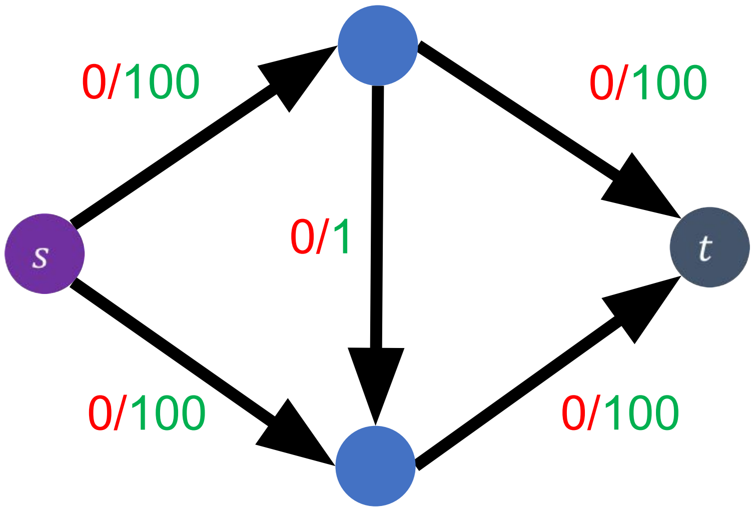
- Initialize $f(e) = 0$ for all $e \in E$
- Construct the residual network G_f
- While there is an augmenting path p in G_f:
  - Let $c = \min_{e \in E} c_f(e)$ ($c_f(e)$ is the weight of edge e in the residual network G_f)
  - Add c units of flow to G based on the augmenting path p
  - Update the residual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

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

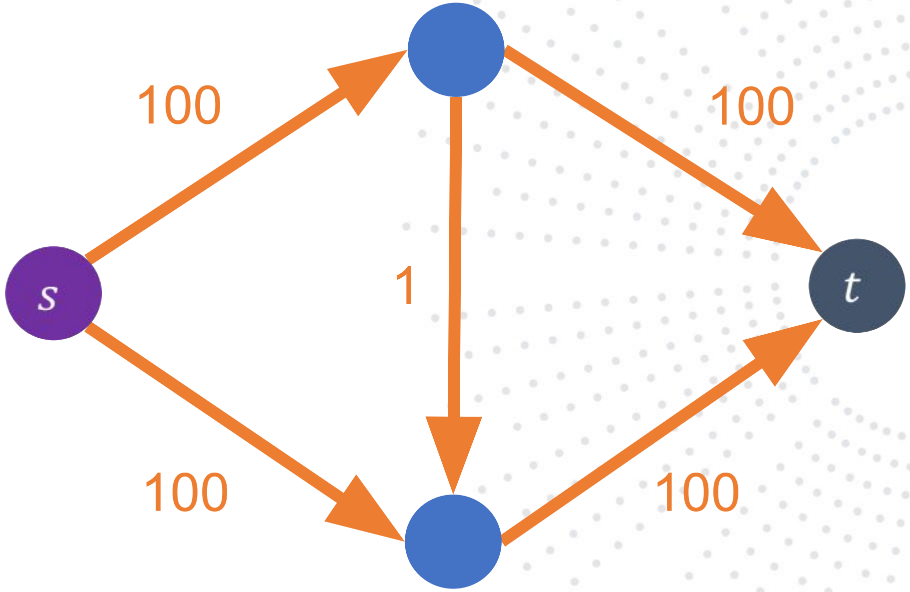
Worst-Case Ford-Fulkerson Algorithm

- The **worst-case time complexity** of the **Ford-Fulkerson algorithm** can arise in specific scenarios where the algorithm has to process a large number of augmenting paths to reach the maximum flow
  - Path augmentation is slow
  - Edge capacities are small

Worst-Case Ford-Fulkerson Algorithm

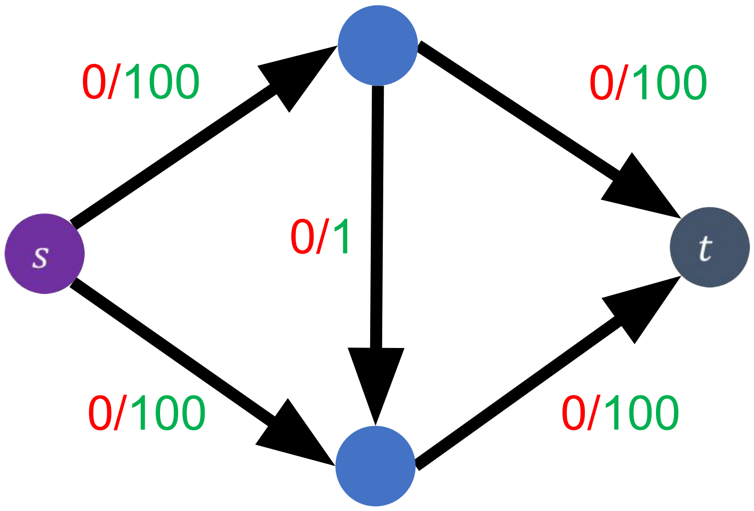


Flow graph



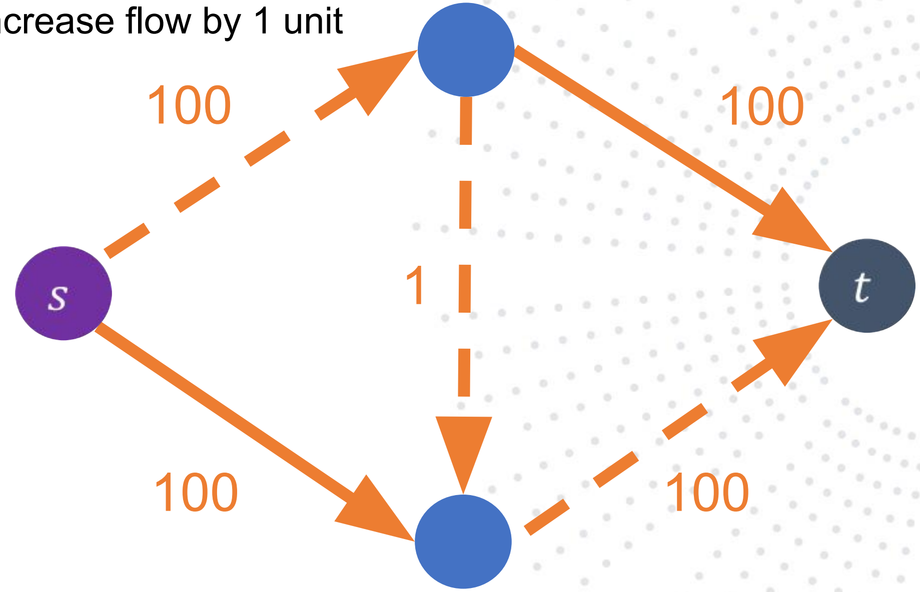
Residual graph

Worst-Case Ford-Fulkerson Algorithm



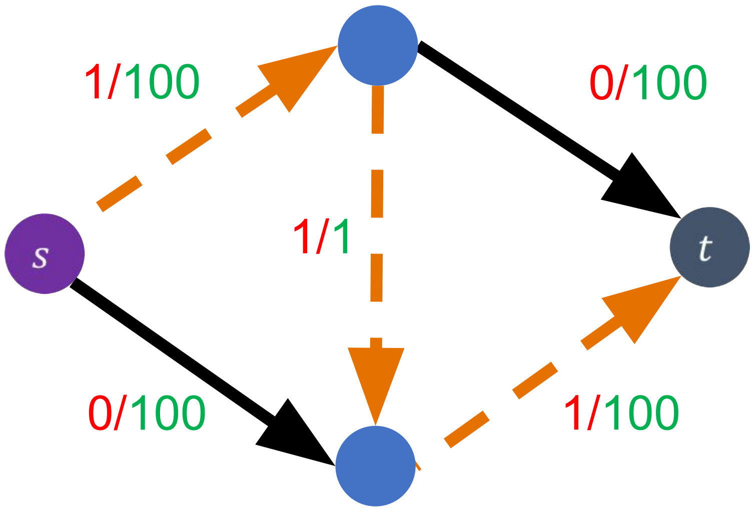
Flow graph

Increase flow by 1 unit



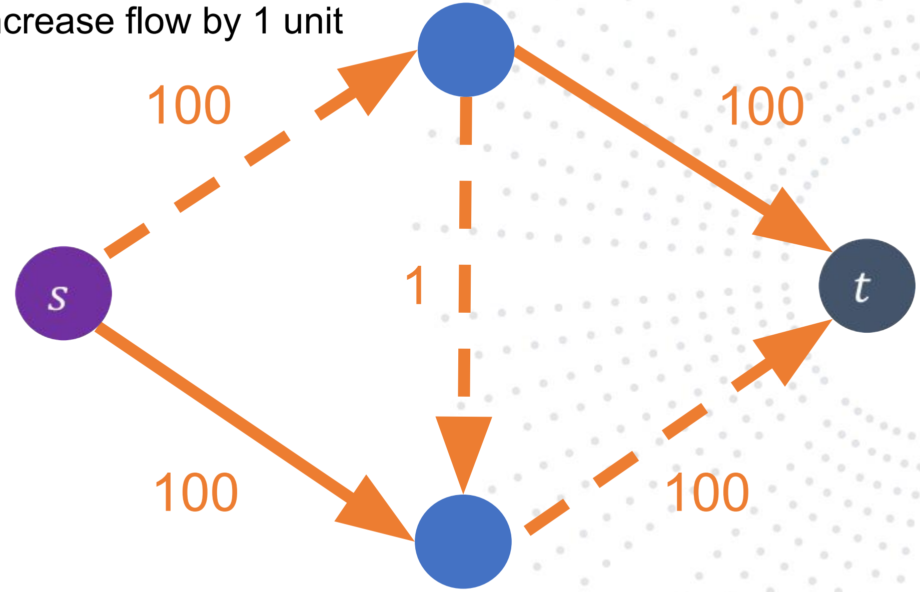
Residual graph

Worst-Case Ford-Fulkerson Algorithm



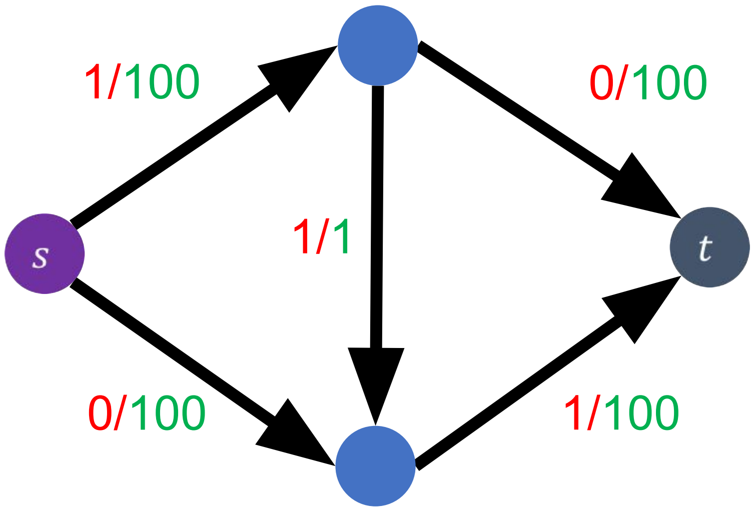
Flow graph

Increase flow by 1 unit

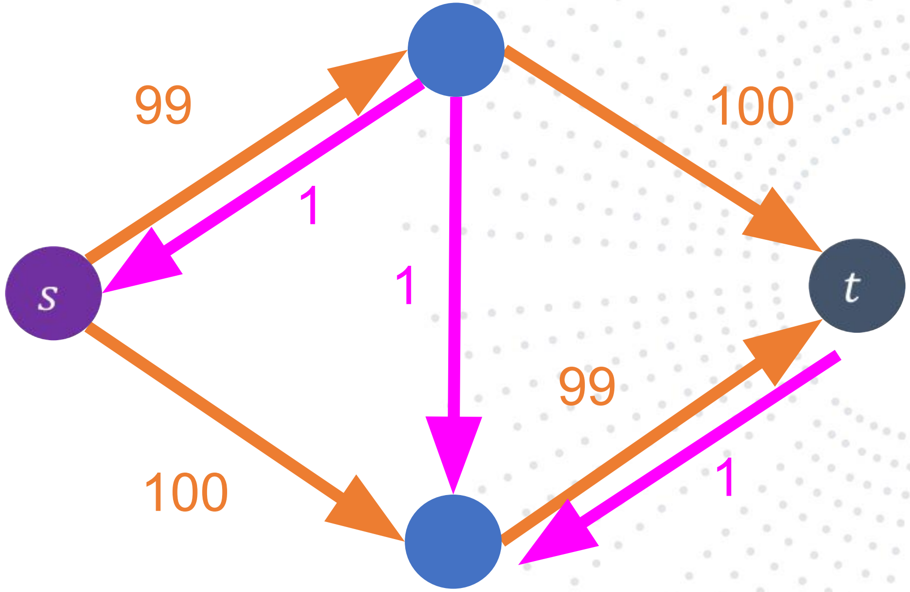


Residual graph

Worst-Case Ford-Fulkerson Algorithm

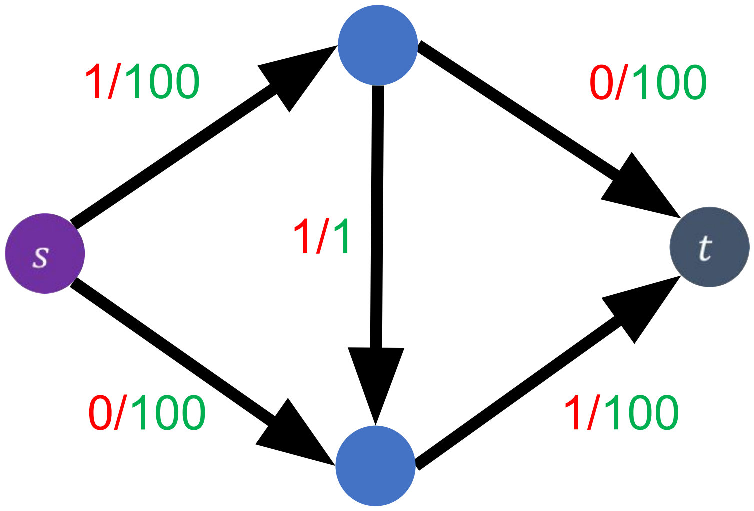


Flow graph



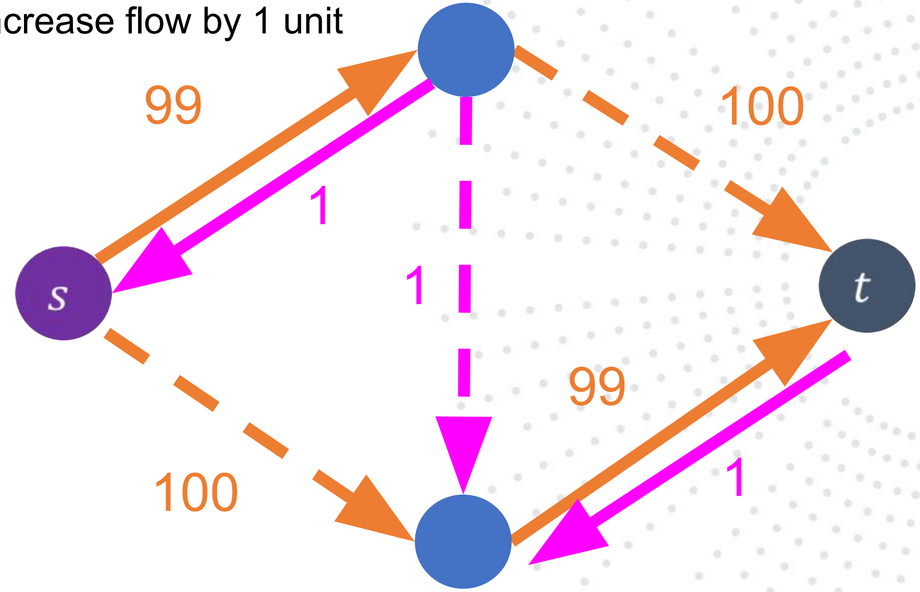
Residual graph

Worst-Case Ford-Fulkerson Algorithm



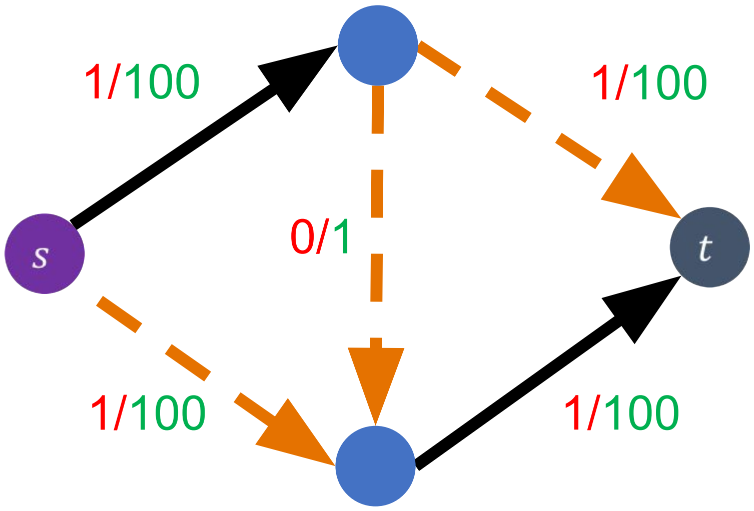
Flow graph

Increase flow by 1 unit



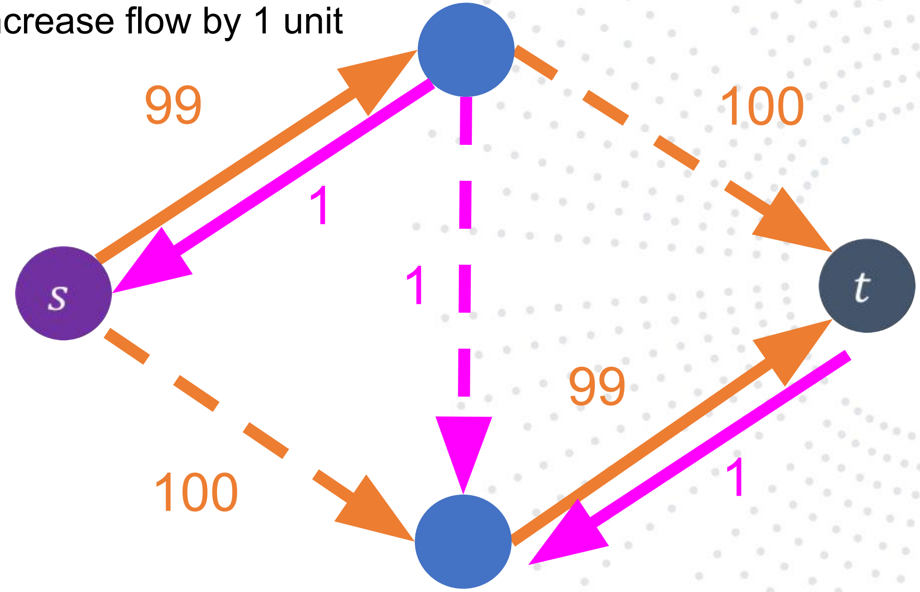
Residual graph

Worst-Case Ford-Fulkerson Algorithm



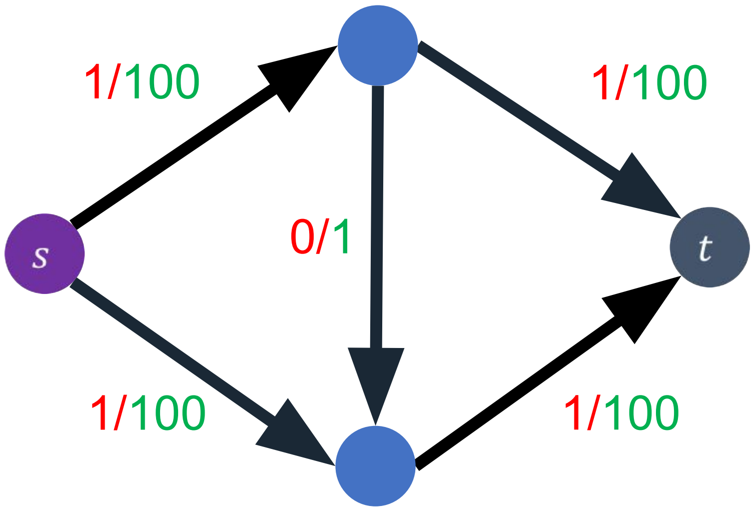
Flow graph

Increase flow by 1 unit

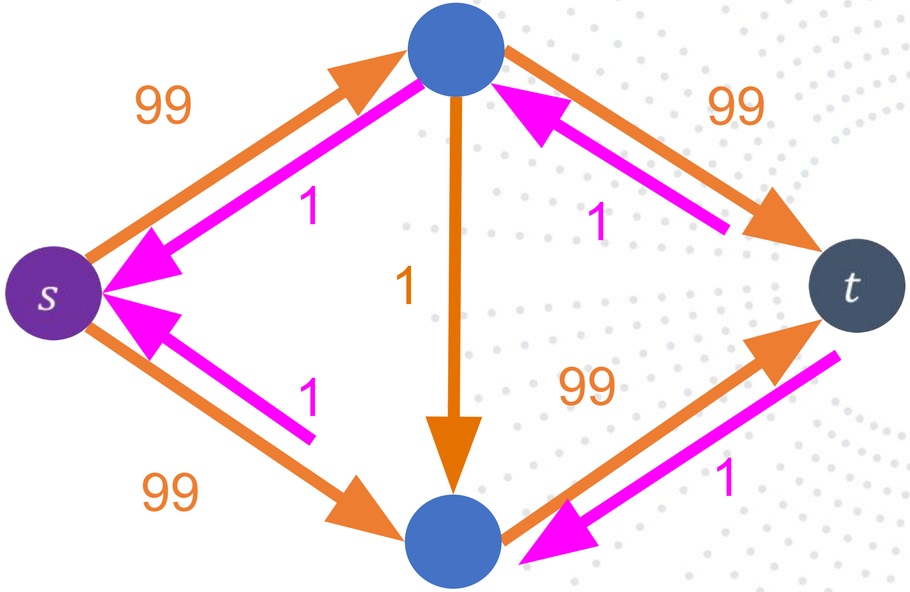


Residual graph

Worst-Case Ford-Fulkerson Algorithm

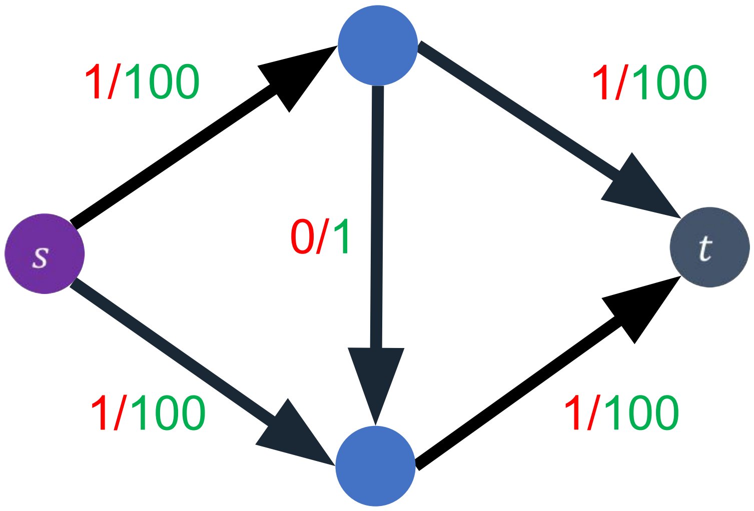


Flow graph



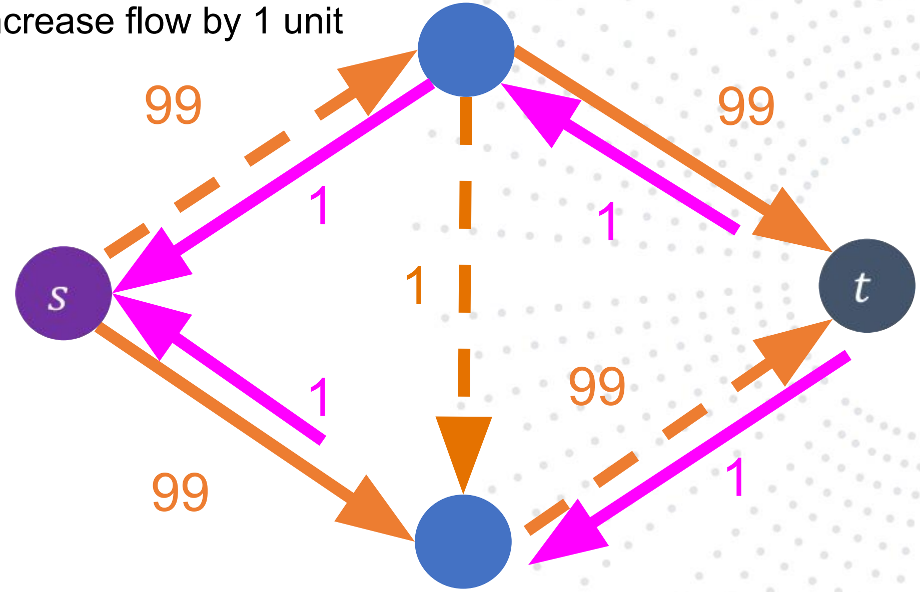
Residual graph

Worst-Case Ford-Fulkerson Algorithm



Flow graph

Increase flow by 1 unit



Residual graph

Ford-Fulkerson algorithm

- **Worst Case (Exponential Time Complexity)**:
    - $O(E \cdot \text{Max Flow})$, which can be exponential in some cases
- This happens when the algorithm uses **Depth-First Search (DFS)** to find augmenting paths, and capacities lead to tiny flow increments.
- For example, if each augmenting path adds only 1 unit of flow and the maximum flow is F, there can be F iterations. If E edges are checked in each iteration, the time complexity becomes $O(E \cdot F)$

Edmonds-Karp Algorithm

- The **Edmonds-Karp algorithm** is a specific implementation of the **Ford-Fulkerson method** for solving the **maximum flow problem** in a flow network. It improves the Ford-Fulkerson method by using **Breadth-First Search (BFS)** to find the shortest augmenting path in terms of the number of edges

Edmonds-Karp Algorithm

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)

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

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

Edmonds-Karp Algorithm

- **Edmonds-Karp algorithm** has a time complexity of $O(V \cdot E^2)$
  - BFS takes $O(V+E)$ time per iteration
  - Each augmenting path can increase the flow by a finite amount.
  - The maximum number of BFS iterations is $O(V \cdot E)$, because each edge is involved at most V times.
- Since each BFS takes $O(V+E)$, and BFS is called $O(V \cdot E)$ times, the total time complexity is:

$$O((V+E) \cdot V \cdot E) = O(V \cdot E^2)$$