UVA DATA SCIENCE

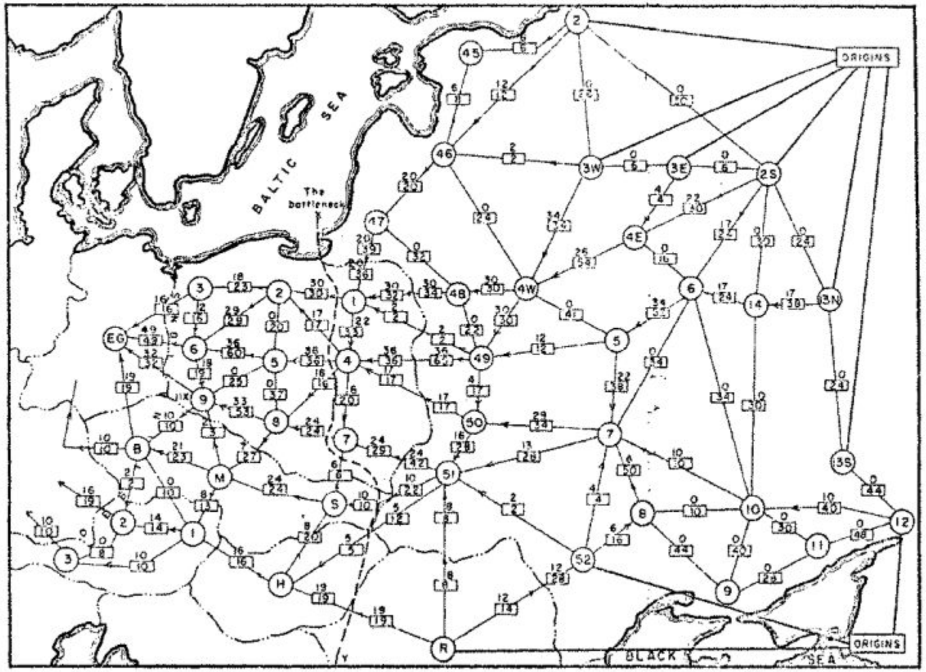
Foundation of Computer Science for Data Science

# Classic CS
# Maximum Flow

Mai Dahshan

updated November 12, 2024

# Motivation

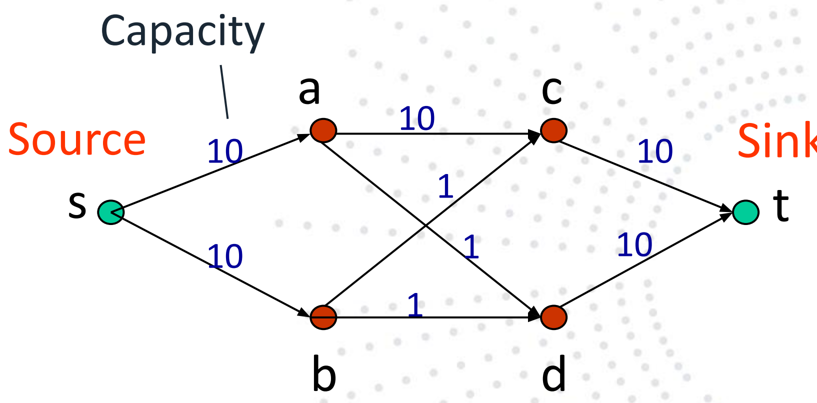



Soviet railway network, 1940

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

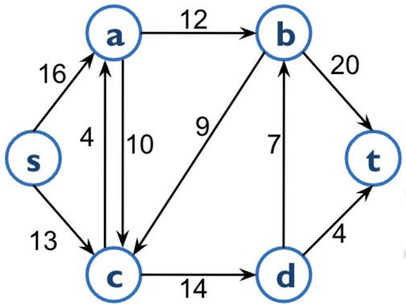
# Flow Network

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



# Flow Network

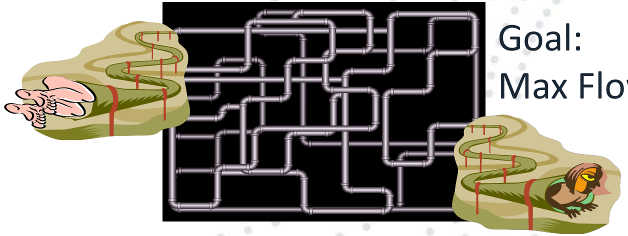
- The flow network problem is defined as follows:
  - Given a directed graph G with non-negative integer weights
    - Each edge stands for the capacity of that edge
  - Two different vertices, s and t, called the source and the sink
    - The source only has out-edges and the sink only has in-edges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




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

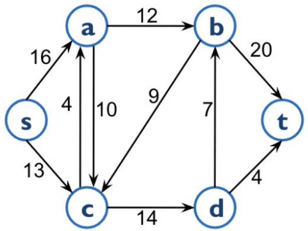
- **Transportation:** Modeling traffic on a network of roads, or the routing of packages by a company
- **Communication:** Routing packets in a communication network Air travel: Sequencing the legs of a flight
- **Railway systems:** Transporting goods across a railway system Vehicle routing: Finding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

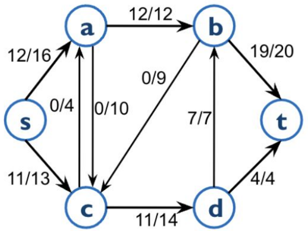
# Flow Graph

- **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

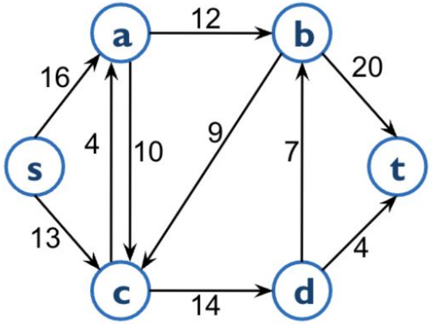
- A **flow** in a flow network is function *f* that assigns each edge *e* a non-negative integer value, namely the flow.
- The function has to fulfill the following two conditions:
  - 0 <= f(e) <= c(e)
  - The **sum of the incoming flow** of a vertex u has to be **equal to the sum of the outgoing flow** of except in the source and sink vertices



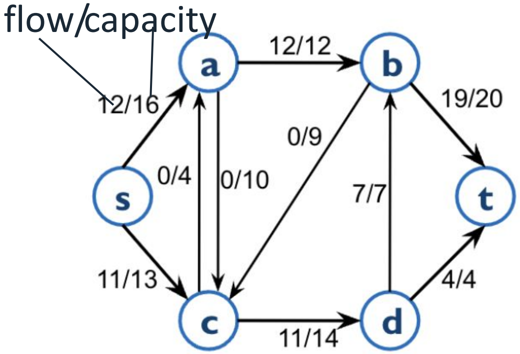



# Flow Graph

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




capacity

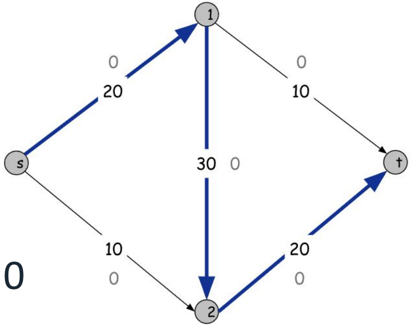



flow/capacity

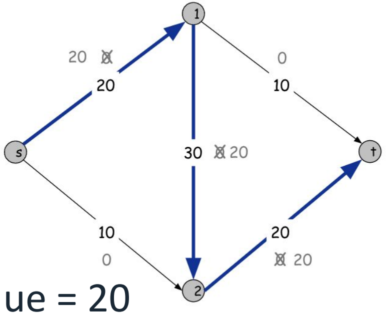
The network flow is 23

# Greedy Approach

- In a **greedy algorithm**, at each step, you would choose the edge that appears to maximize flow from the source to the sink, based on a local criterion (e.g., choosing the edge with the maximum available capacity)



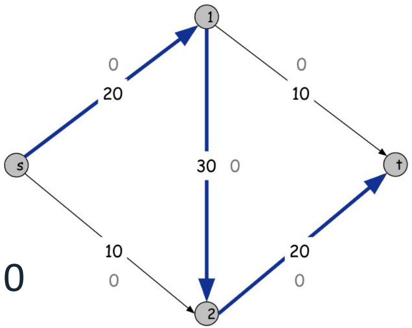

flow value = 0



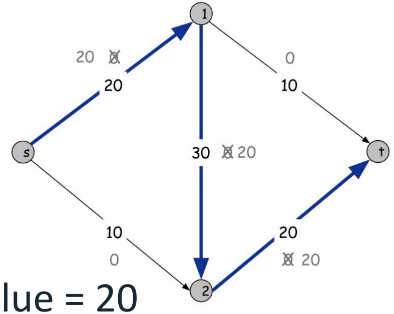

flow value = 20

# Greedy Approach

- Start with f(e) = 0 for all edge e $\in$ E
- Find an s-t path P where each edge maximize f(e) and f(e) < c(e)
- Augment flow along path P
- Repeat until you get stuck

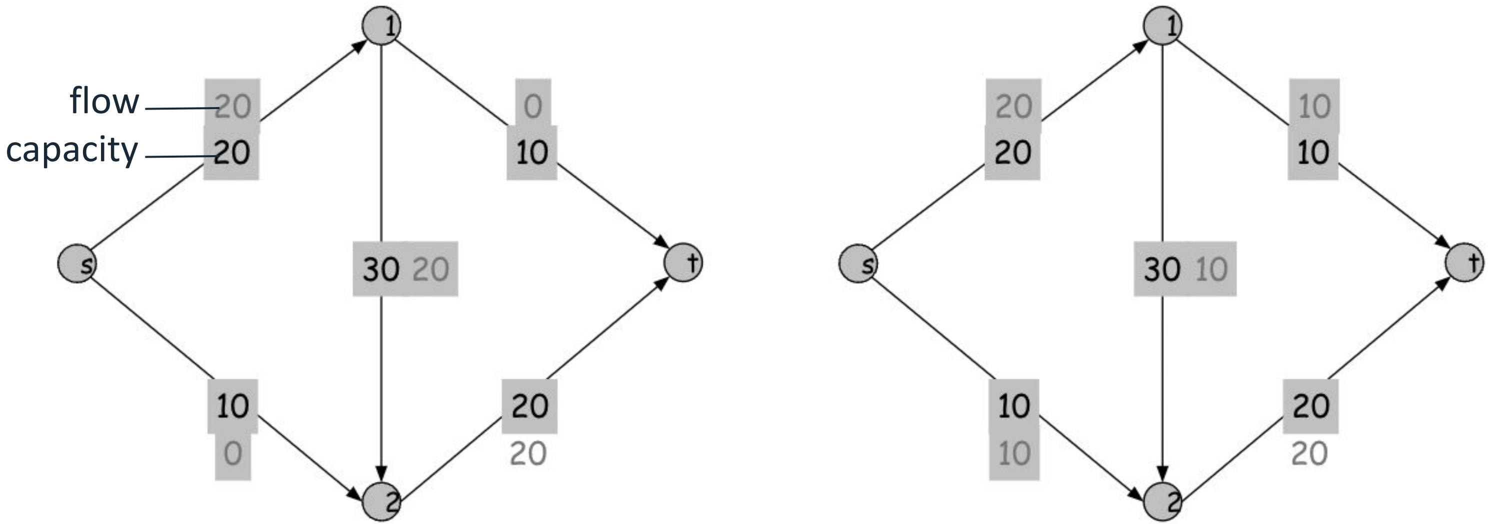



flow value = 0




flow value = 20

# Greedy Approach

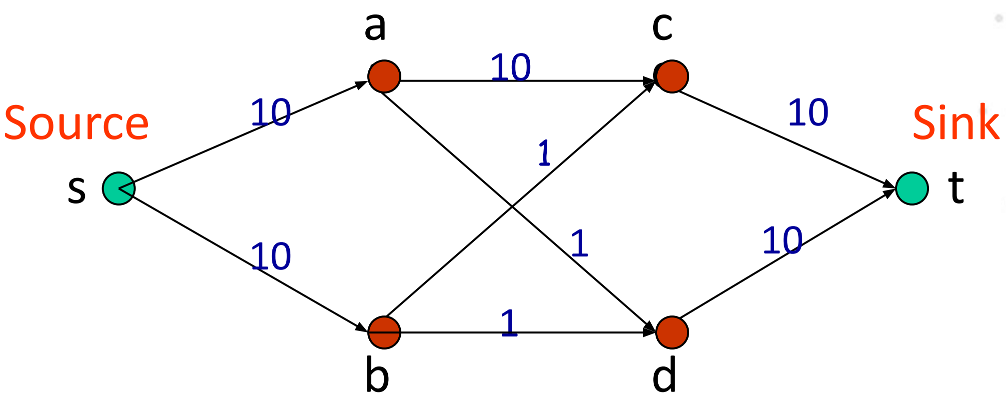



greedy = 20

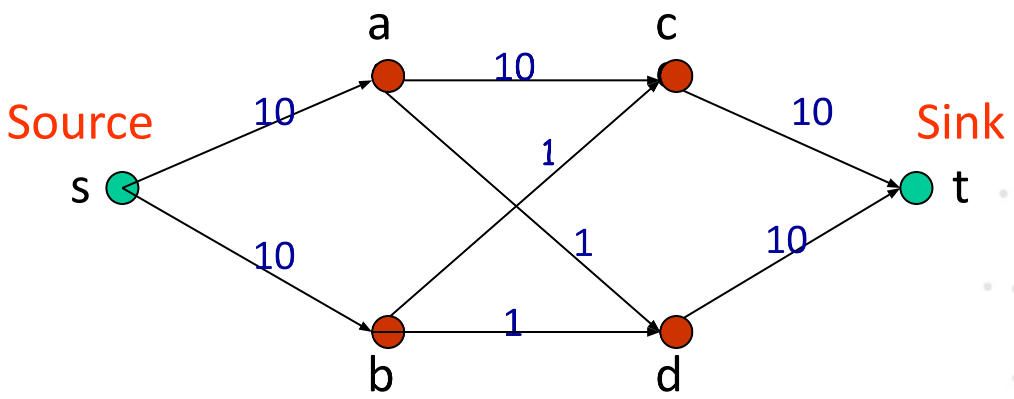
optimal (Max flow)= 30

locally optimality != global optimality

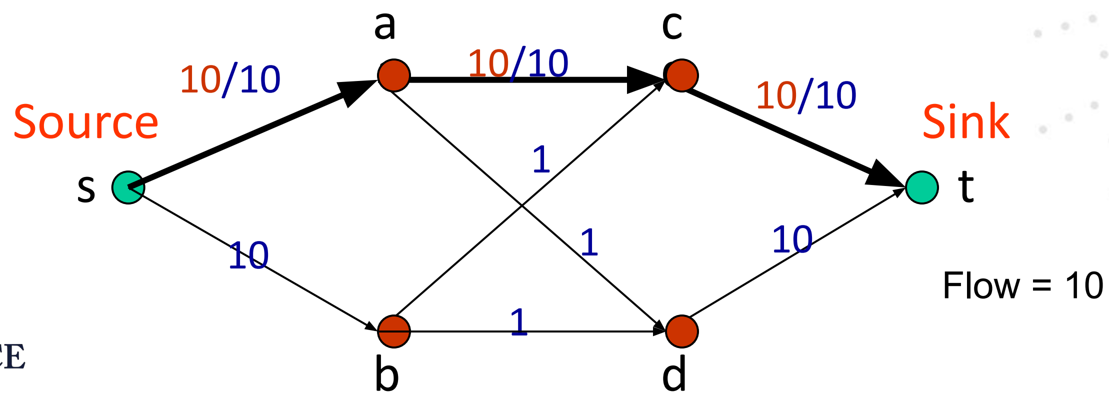
# Greedy Approach



# Greedy Approach

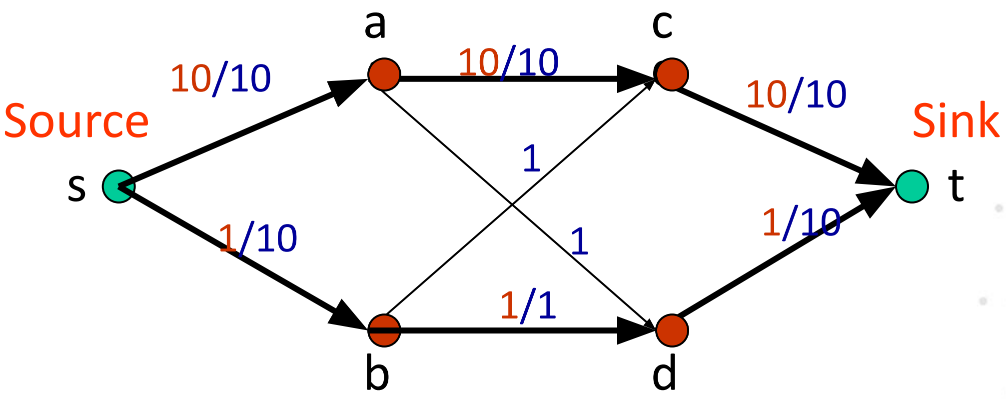



Step 1:



# Greedy Approach

Step 2:

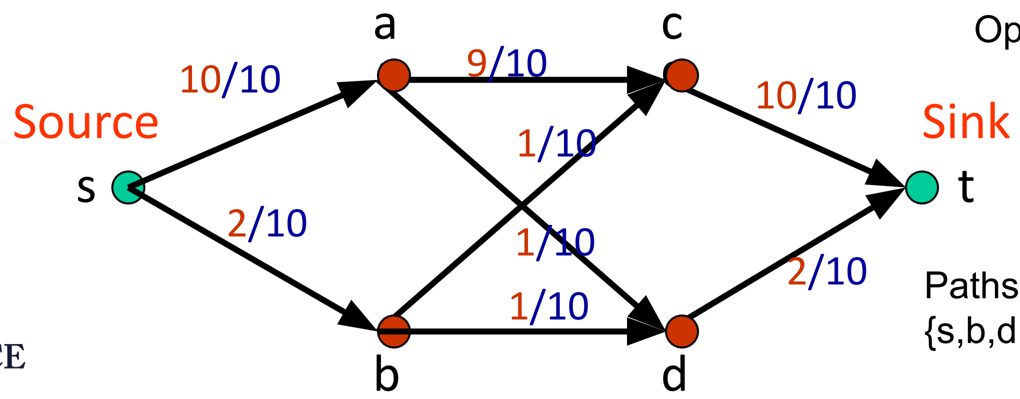



Flow = 10 + 1 = 11

Greedy = 11

Optimal = 12

Max Flow:




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

# Ford-Fulkerson Algorithm

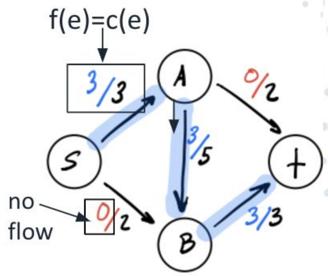
- The **Ford-Fulkerson algorithm** is a classic method for finding the **maximum flow** in a flow network. It uses the concept of **augmenting paths** and operates on the **residual graph** to iteratively improve the flow until no more augmenting paths exist

# Residual graph

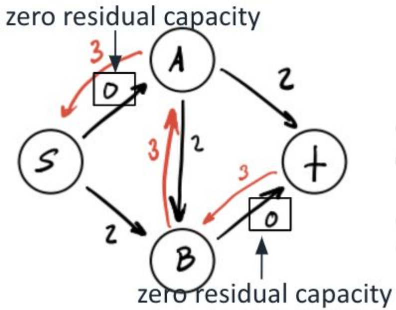
- Residual graph shows the remaining capacity
- It tracks the flow capacity and allows for flow adjustments during the process of finding maximum flow.
- Given a flow *f* in graph *G,* the residual graph $G_f$ models additional flow that is possible
  - **Forward edge** for each edge in *G* with weights set to remaining capacity c(e)-f(e)
    - 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

# Residual graph

- **Forward edge capacity** corresponds to the amount of flow that can still be pushed through that edge (c(e)-f(e))
- A **forward edge exists** if there is any **remaining capacity** for flow in the original edge.
- If the **flow on a particular edge reaches its capacity**, then the forward edge in the residual graph will have **zero residual capacity.**
- If **no flow** has been sent along the edge, the **forward edge will have the same capacity as the original edge** in the flow network



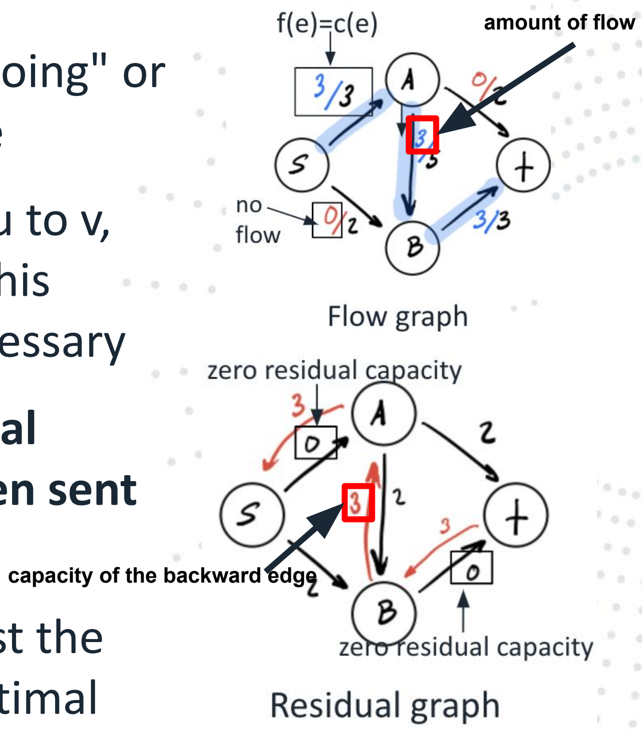

Flow graph



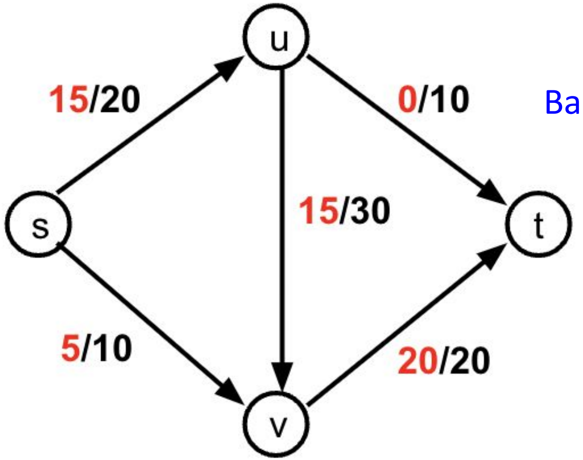

Residual graph

# Residual graph

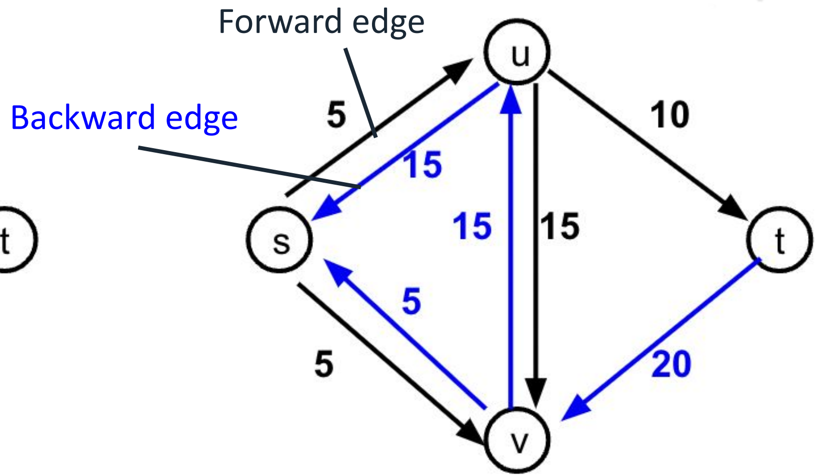
- **Backward Edge** represents the possibility of "undoing" or "reversing" the flow along a previously used edge
- It exists when there is flow already passing from u to v, and the flow can be "pushed back" from v to u. This backward edge allows us to adjust the flow if necessary
- The **capacity** of the **backward edge in the residual graph** is the **amount of flow that has already been sent along the forward edge in the original graph**
- The backward edge allows the algorithm to adjust the flow when a previously used path is no longer optimal



# Ford-Fulkerson Algorithm

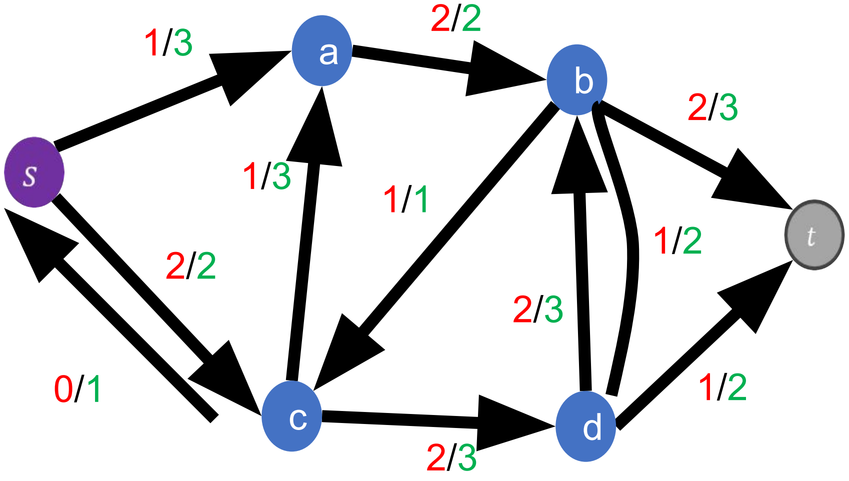



Flow graph G




Residual Graph $G_f$

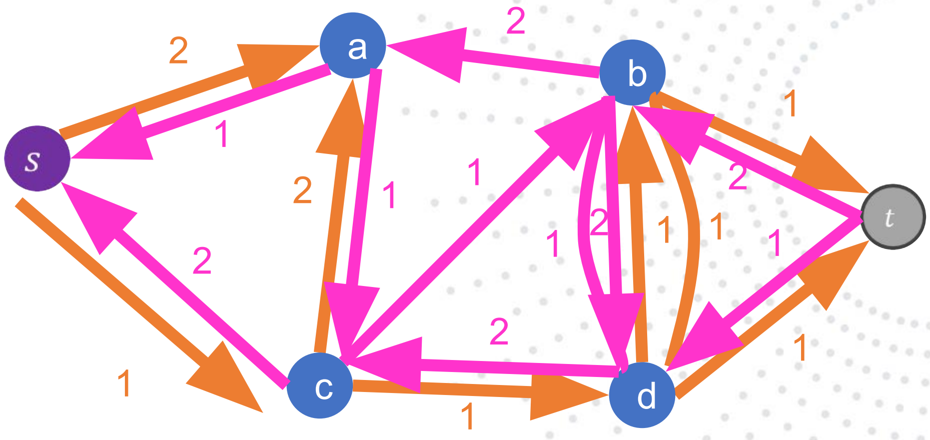
# Ford-Fulkerson Algorithm




Flow graph G

**Forward edge**

**Backward edge**

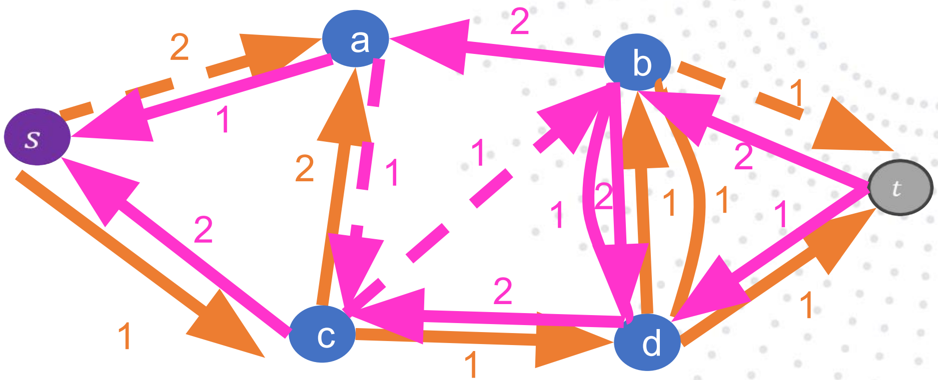



Residual Graph $G_f$

# Ford-Fulkerson Algorithm

- In a residual graph, an **augmented path** is a path from the source node s to the sink node t where each edge has **positive residual capacity** (either forward or backward). The path can mix both types of edges, as follows:
  - **Forward edges** (with positive residual capacity): These edges allow for additional flow to be pushed from the source to the sink
  - **Backward edges** (with positive residual capacity): These edges allow the possibility of undoing or adjusting flow by pushing flow in the reverse direction

**Augmented path: s-a-c-b-t**

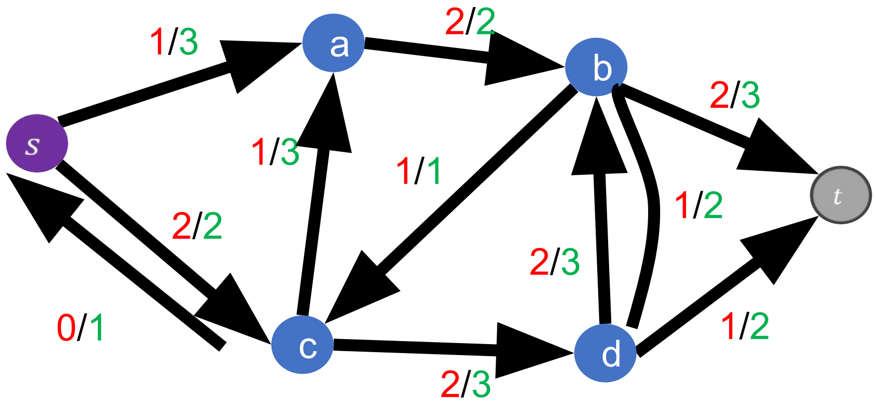


# Ford-Fulkerson Algorithm

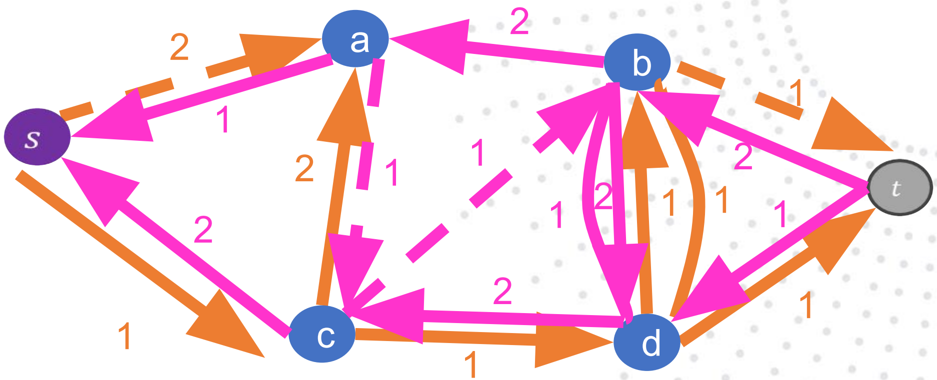
Consider a path from $s \rightarrow t$ in $G_f$ using only edges with positive (non-zero) weight

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)




Flow graph G

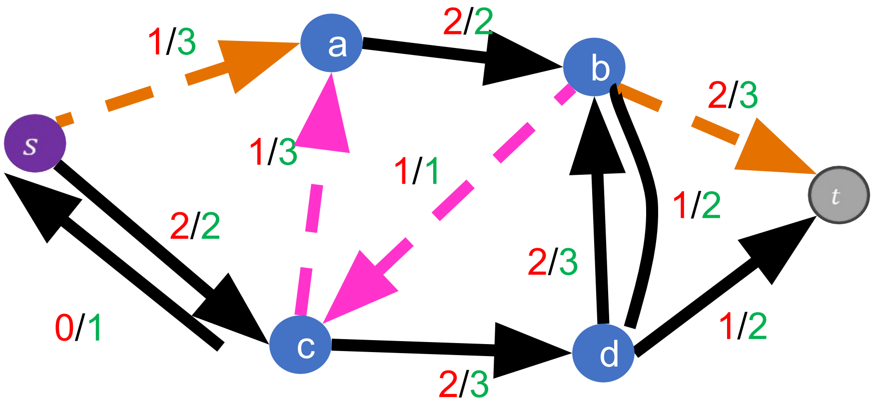



Residual Graph $G_f$

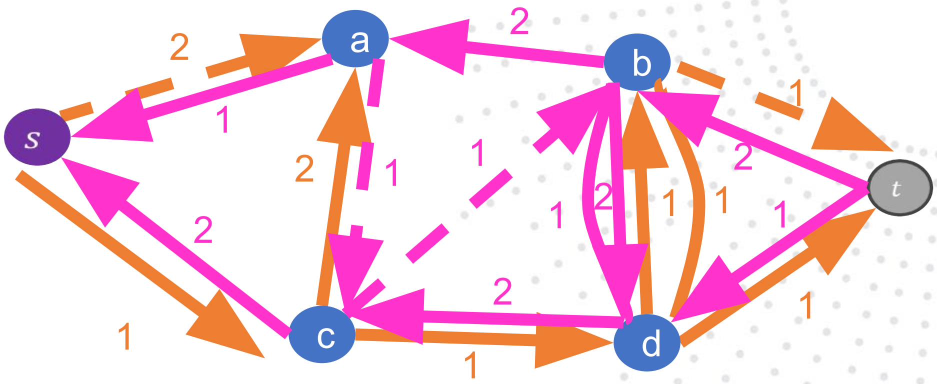
# Ford-Fulkerson Algorithm

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



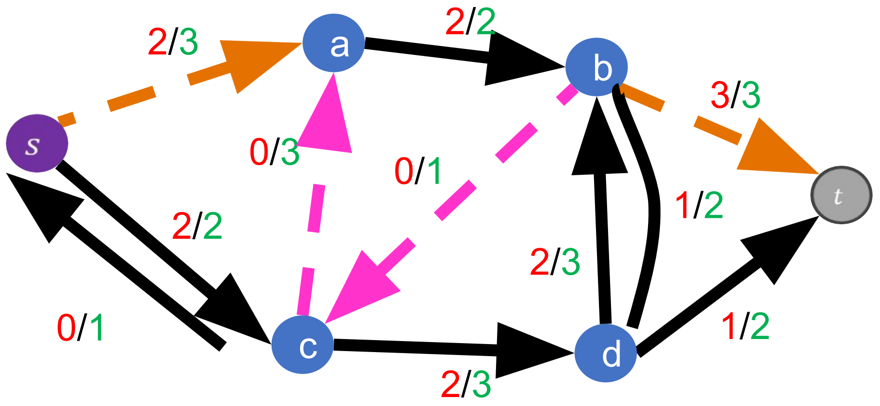

Residual Graph $G_f$

# Ford-Fulkerson Algorithm

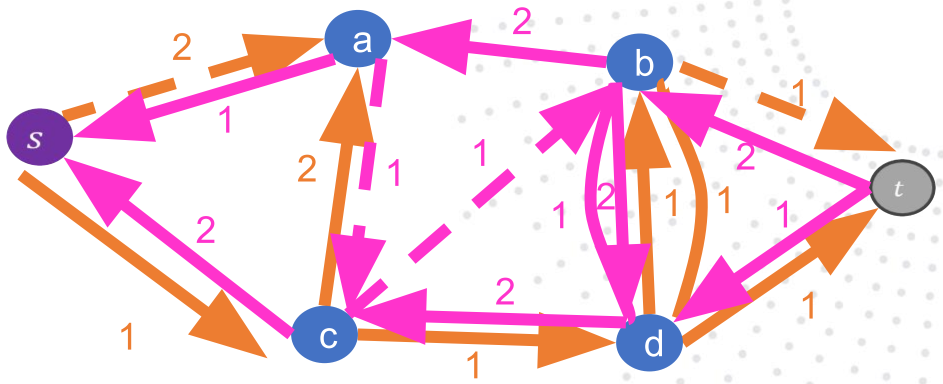
Consider a path from $s \to t$ in $G_f$ using only edges with positive (non-zero) weight
Consider the minimum-weight edge $e$ along the path: we can increase the flow by $w(e)$

- 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)

**Observe:** Flow has <u>increased</u> by $w(e)$




Flow graph G




Residual Graph $G_f$

# 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)

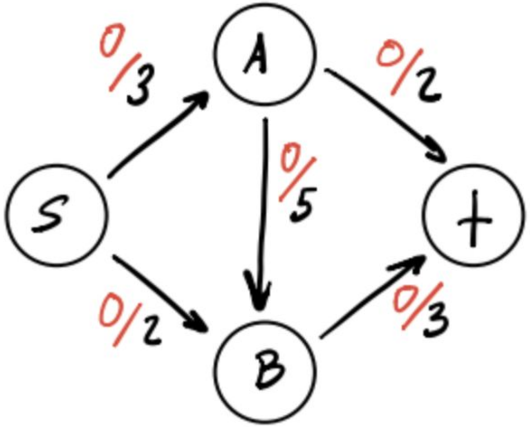
**Ford-Fulkerson approach:**
take any augmenting path

Ford-Fulkerson max-flow algorithm:

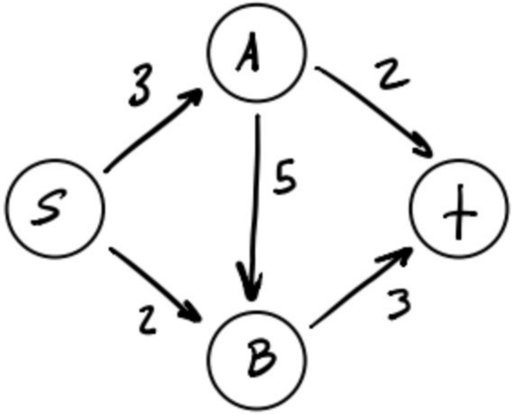
- Initialize $f(e) = 0$ for all $e \in E$
- Construct the residual network $G_f$
- While there is an augmenting path $p$ in $G_f$:
  - Let $c = \min_{e \in E} c_f(e)$ ($c_f(e)$ is the weight of edge $e$ in the residual network $G_f$)
  - Add $c$ units of flow to $G$ based on the augmenting path $p$
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

# Ford-Fulkerson Algorithm- Example 1

- 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



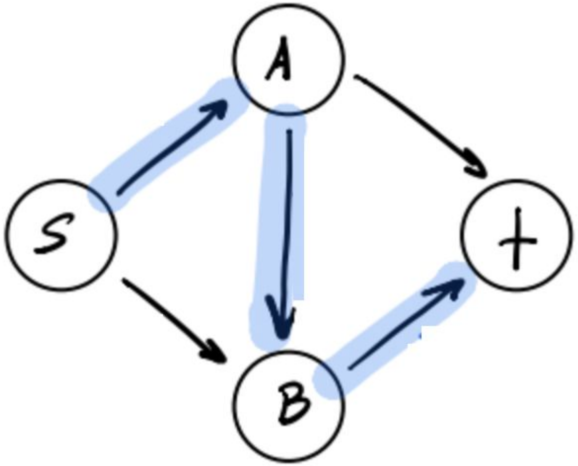

Flow graph




Residual graph

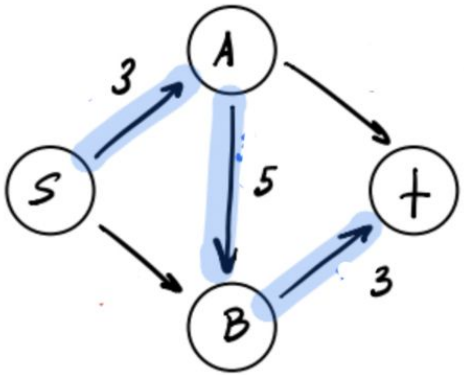
# Ford-Fulkerson Algorithm- Example 1

- 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 1

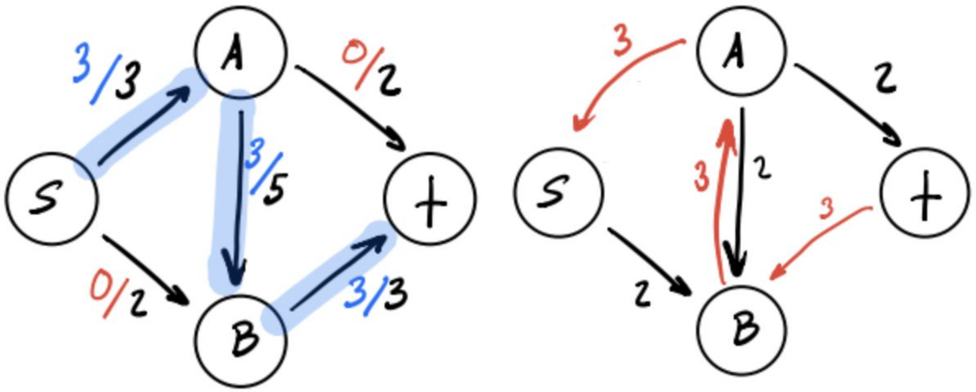
- Let *cap(P)* indicate the maximum amount of stuff that can flow along this path
    - To find the capacity of this path, we need to look at all edges *e* on the path and subtract their current flow, from their capacity. We'll set *cap(P)* to be equal to the smallest value since this will **bottleneck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




3 is the bottleneck for this the path

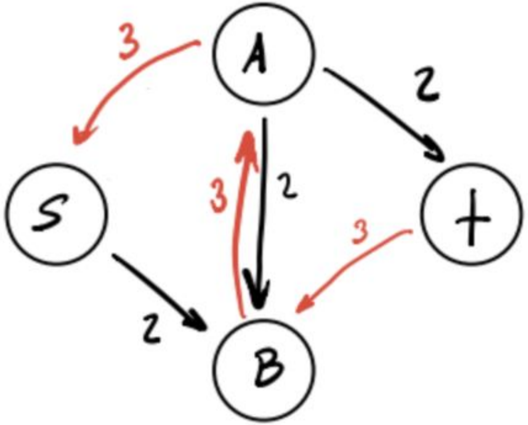
# Ford-Fulkerson Algorithm- Example 1

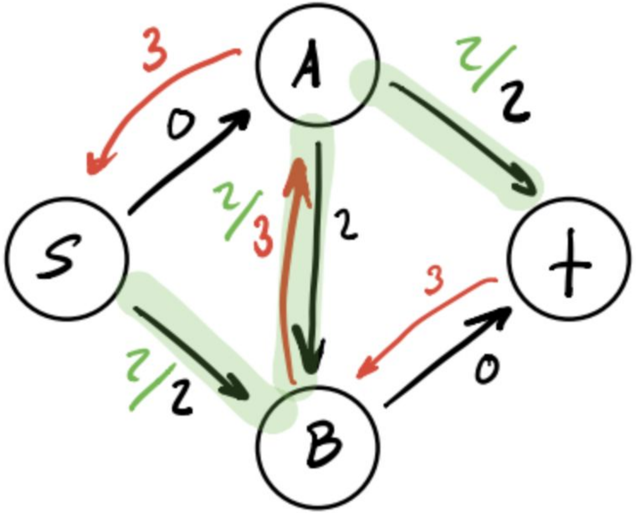
- We then **augment the flow** across the forward edges in the path *P* by adding *cap(P)* value. For flow across the back edges in the residual graph, we subtract our *cap(P)* value
- Update the residual graph with these flow adjustments



# Ford-Fulkerson Algorithm- Example 1

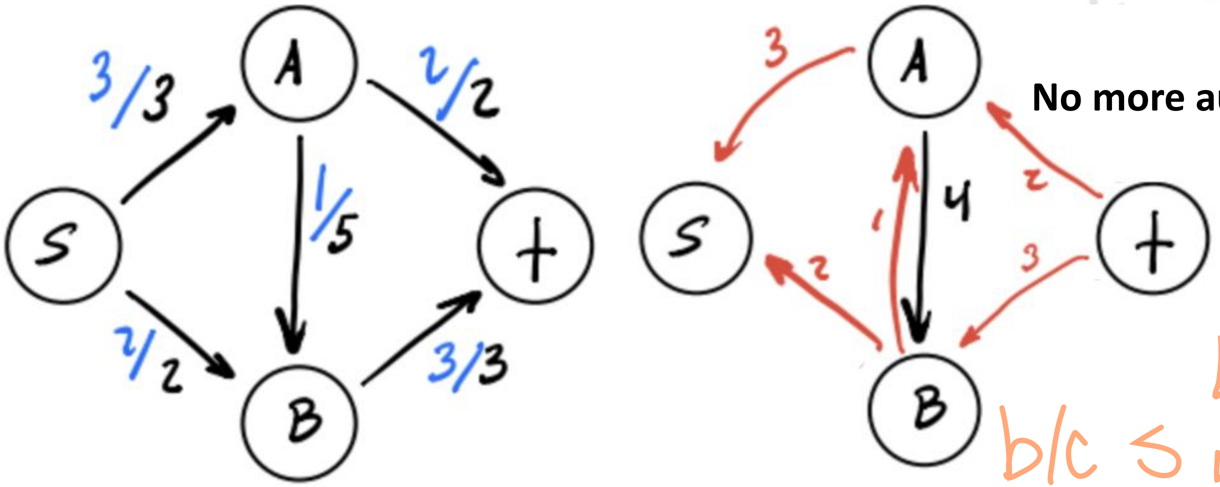
- Search through the updated residual graph for a new *s-t* path. There are no forward edges available anymore, but we can use a back edge to augment the current flow



# Ford-Fulkerson Algorithm- Example 1

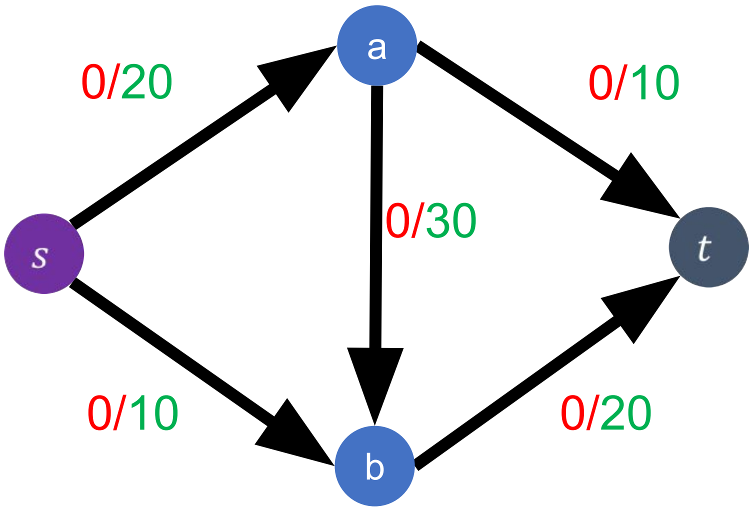
- There are now no edges with available capacity that we can use to create a path from *s* to *t*. This means our run of the Ford-Fulkerson algorithm is complete and our max flow leading into *t* is 5



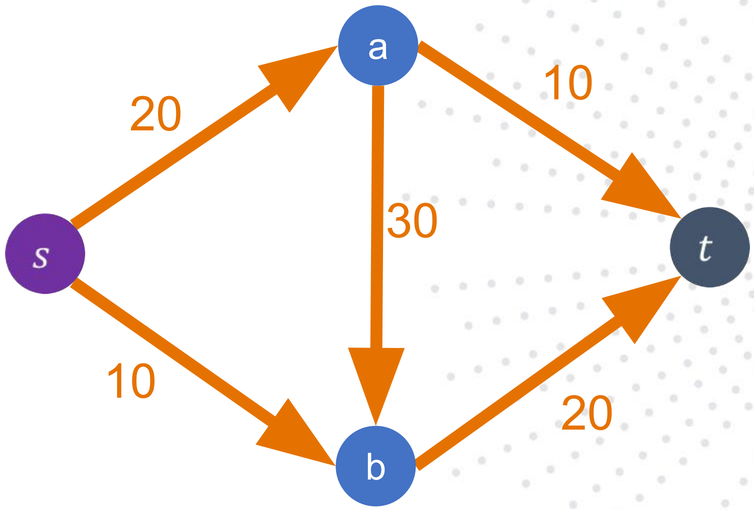

**No more augmented paths**

b/c S can't reach t

# Ford-Fulkerson Algorithm- Example 2

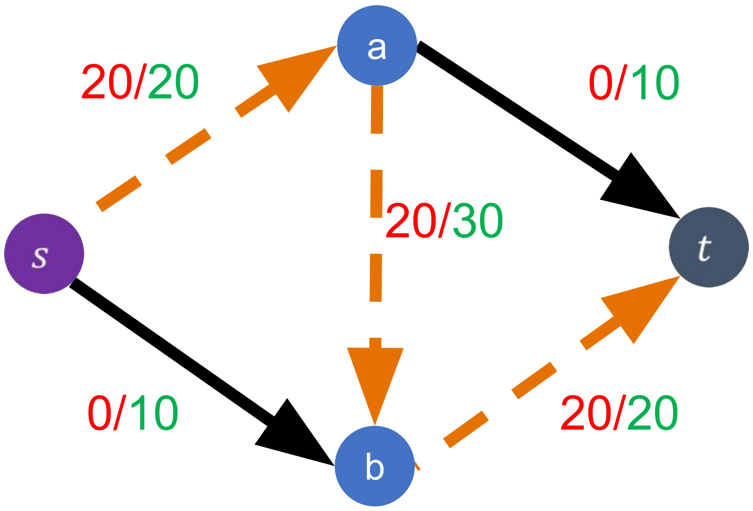


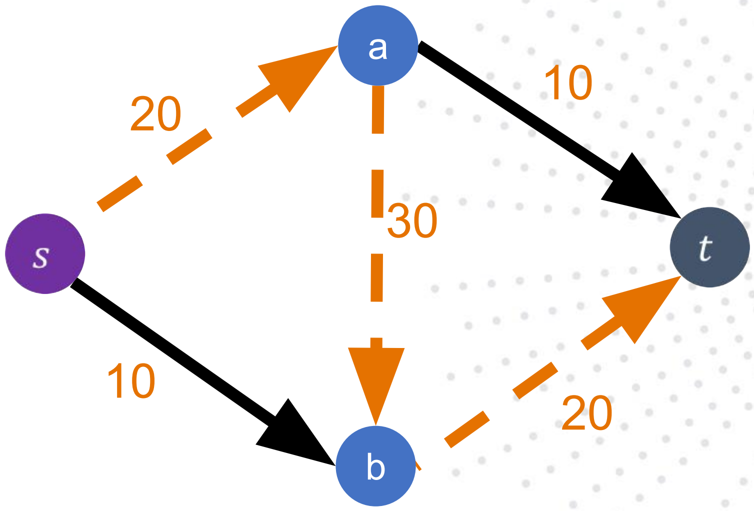

**Initially:** $f(e) = 0$ for all $e \in E$



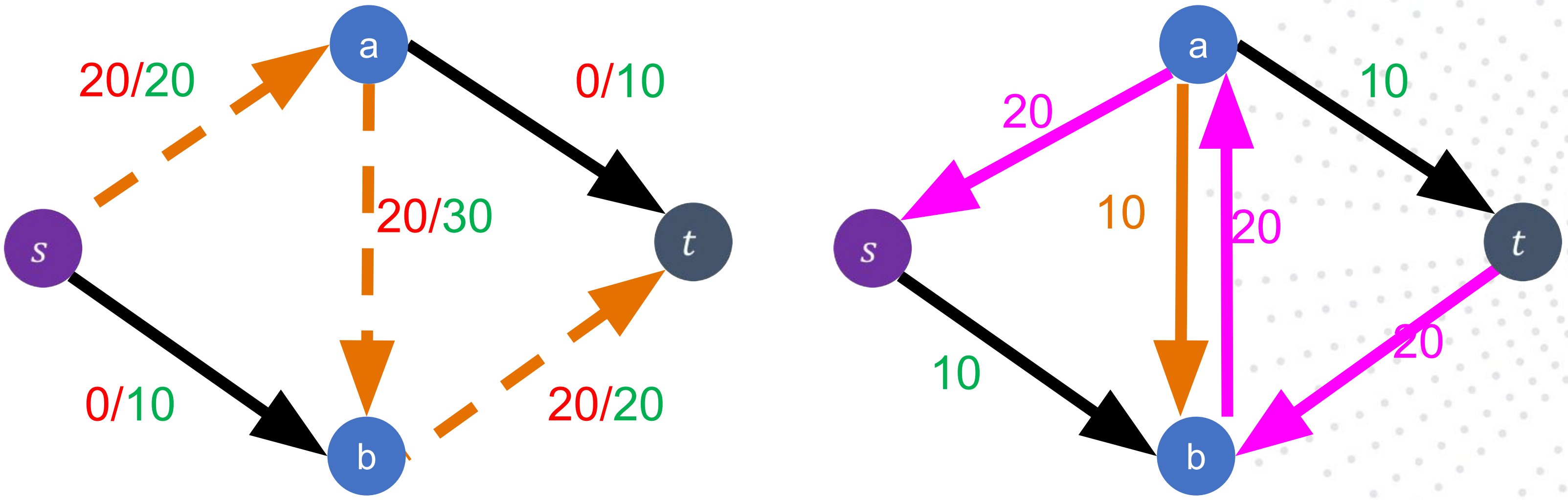

Residual graph $G_f$

# Ford-Fulkerson Algorithm- Example 2

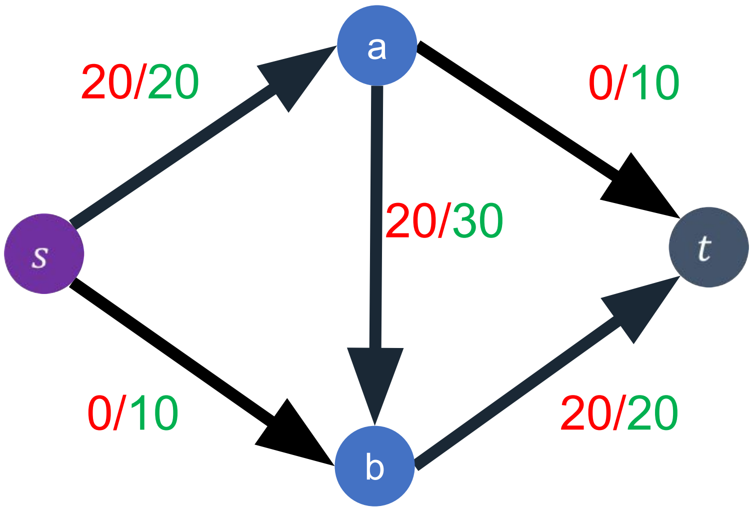


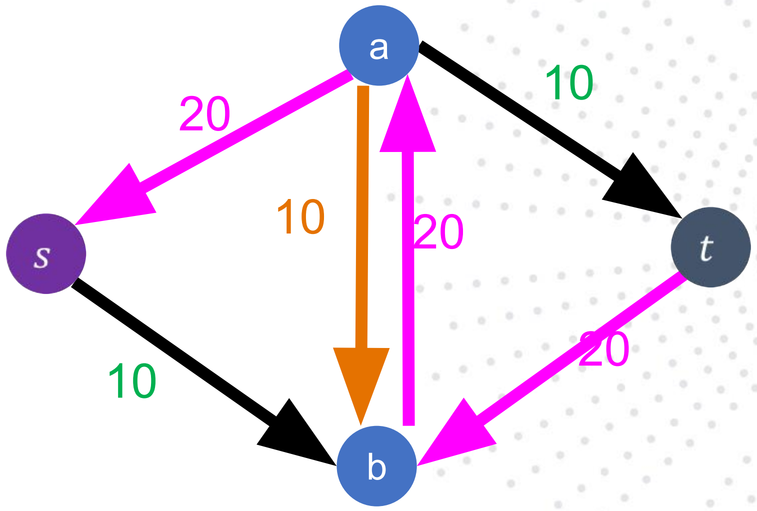



# Ford-Fulkerson Algorithm- Example 2

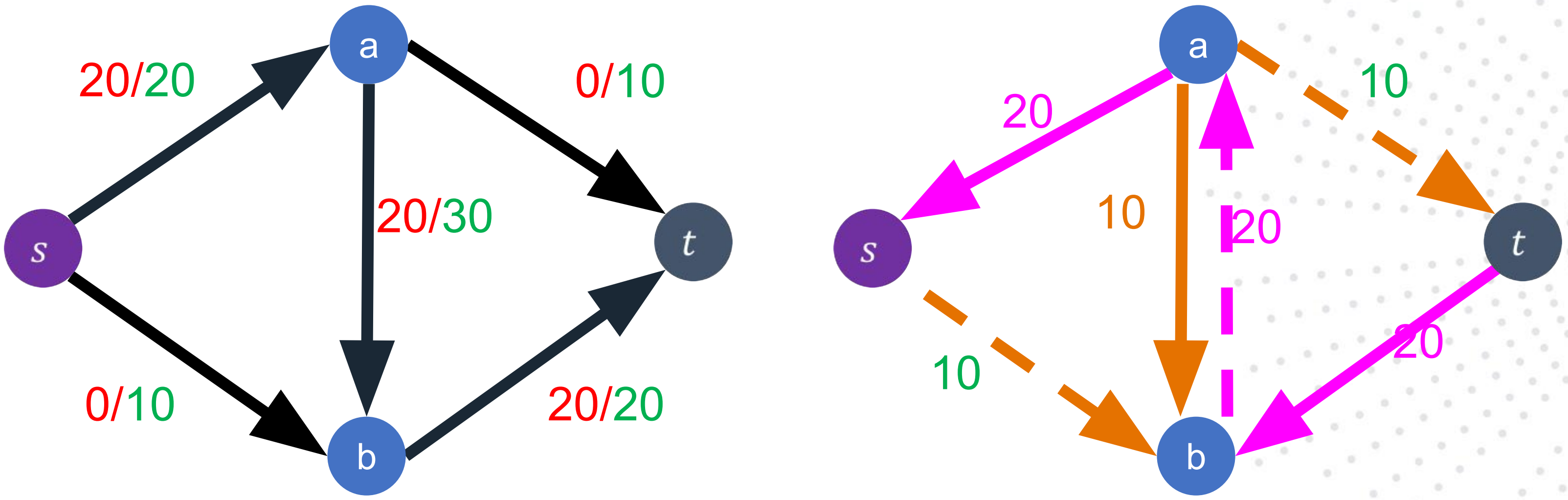


# Ford-Fulkerson Algorithm- Example 2

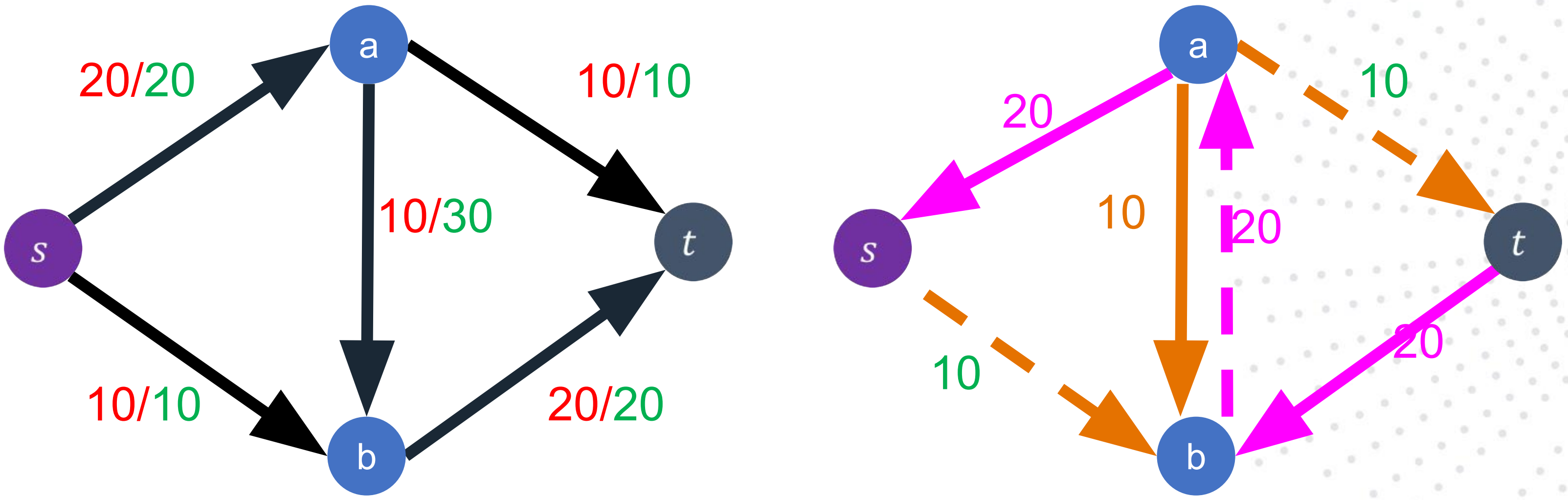


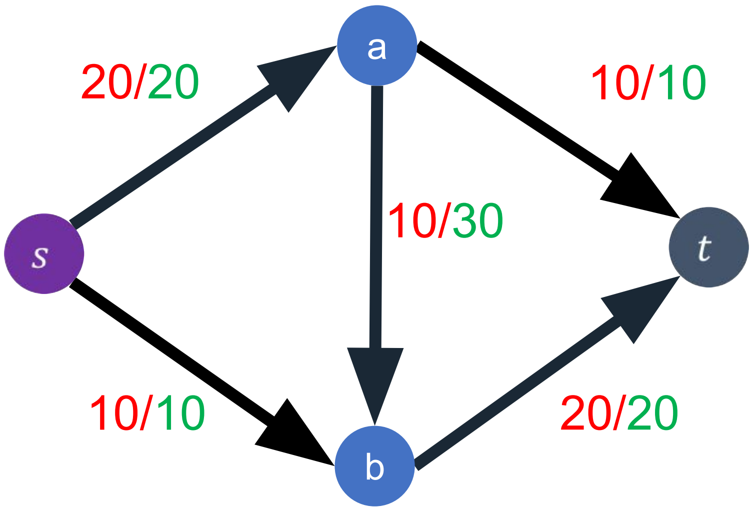
# Ford-Fulkerson Algorithm- Example 2



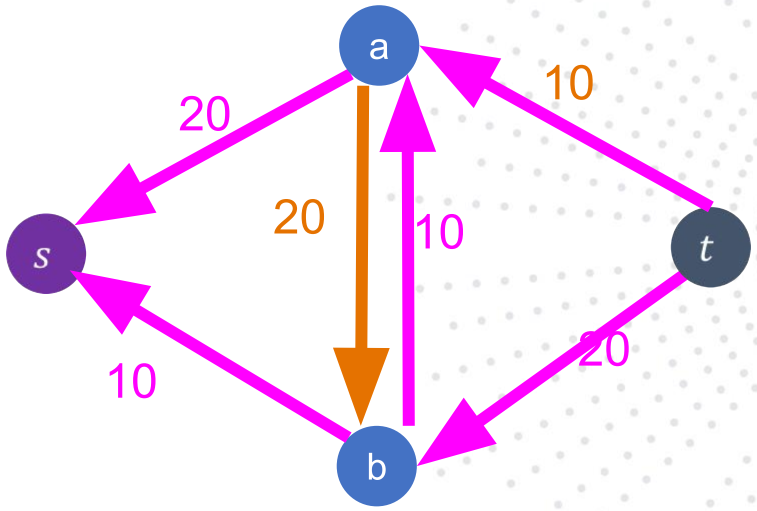
# Ford-Fulkerson Algorithm- Example 2



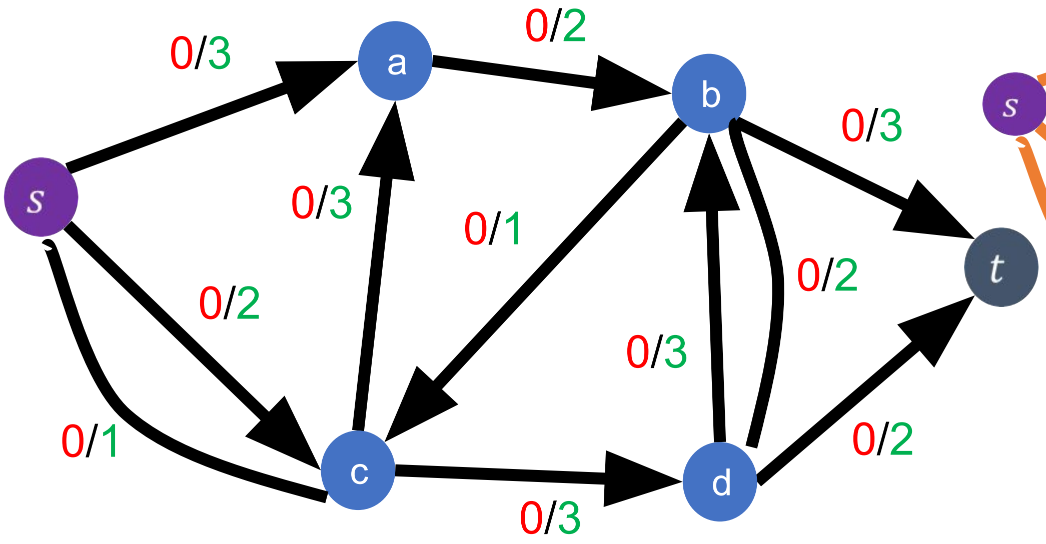
# Ford-Fulkerson Algorithm- Example 2

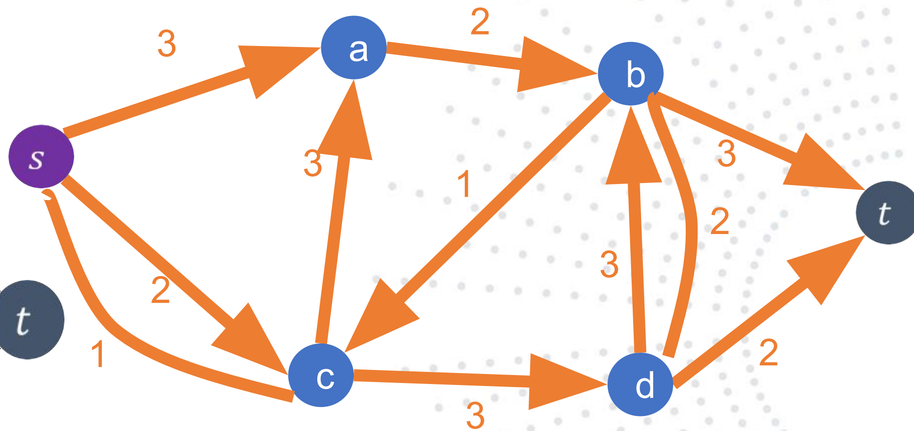



**No more augmented paths**



# Ford-Fulkerson Algorithm- Example 3 Sol.1



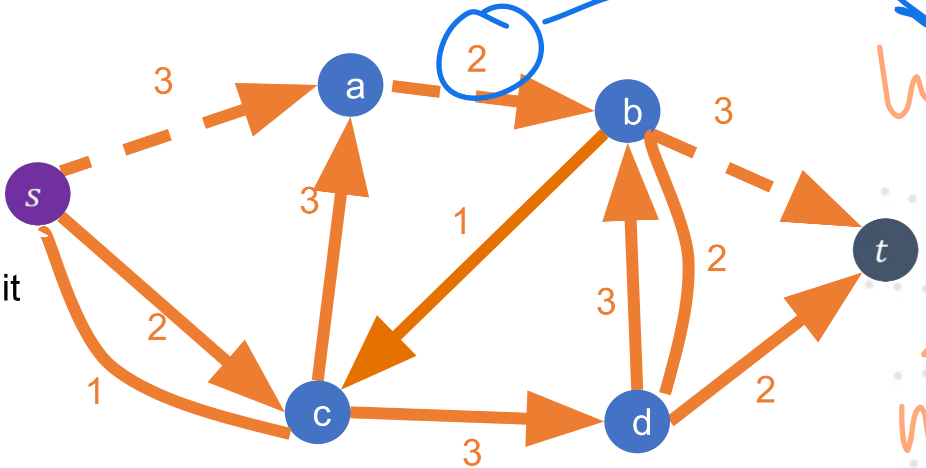

Residual graph $G_f$

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


40

# Ford-Fulkerson Algorithm- Example 3 Sol.1

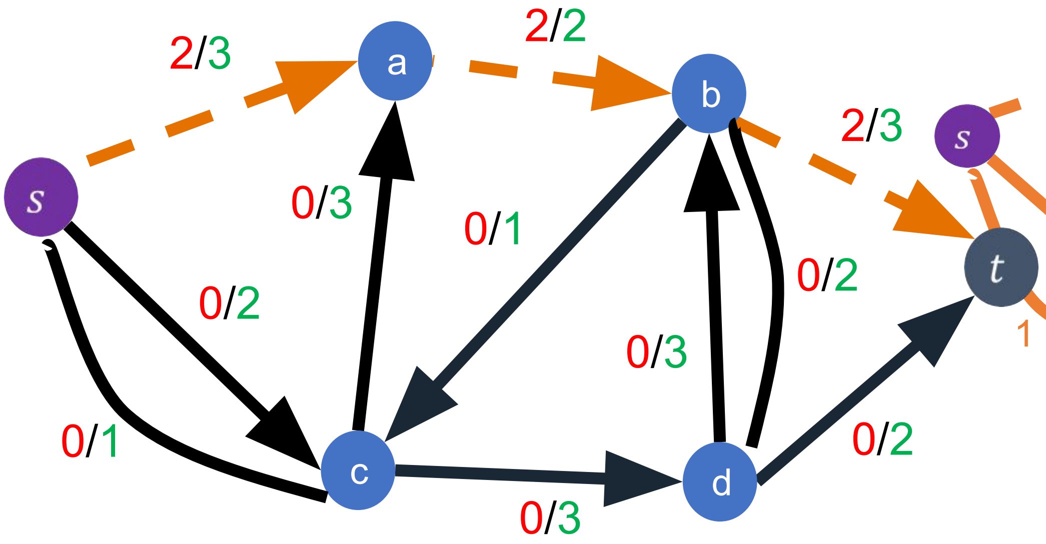
Increase flow by 2 unit



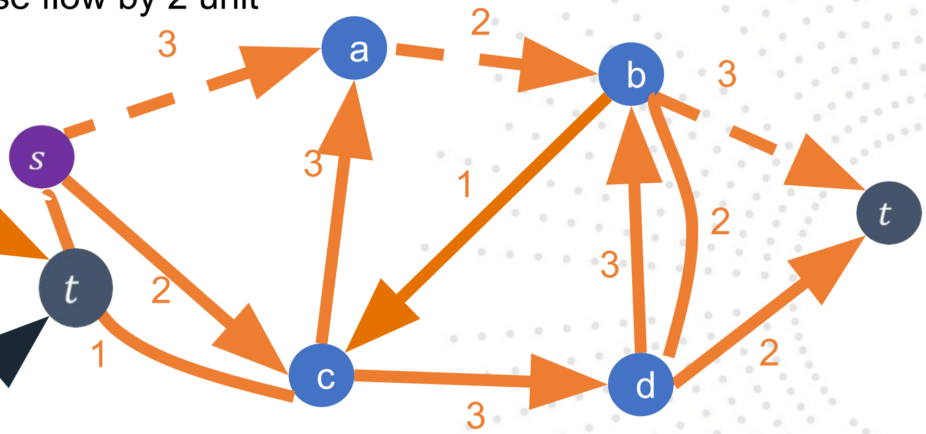

We choose s-a-b-t, and our max flow is 2 units

Residual graph $G_f$

# Ford-Fulkerson Algorithm- Example 3 Sol.1



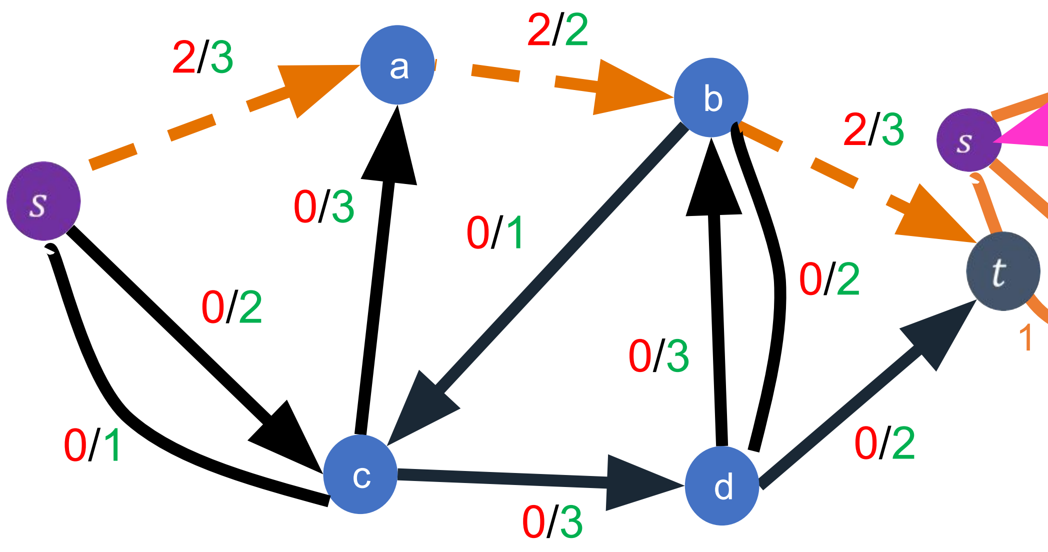

Increase flow by 2 unit



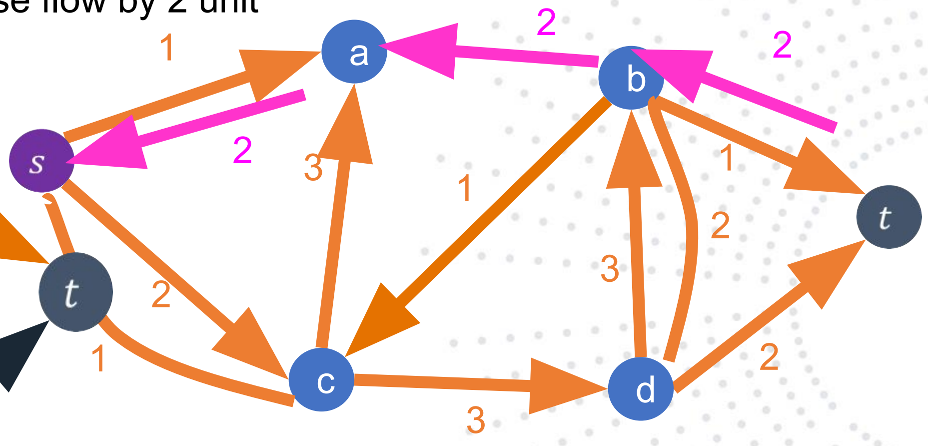

Residual graph $G_f$


42

# Ford-Fulkerson Algorithm- Example 3 Sol.1




Increase flow by 2 unit

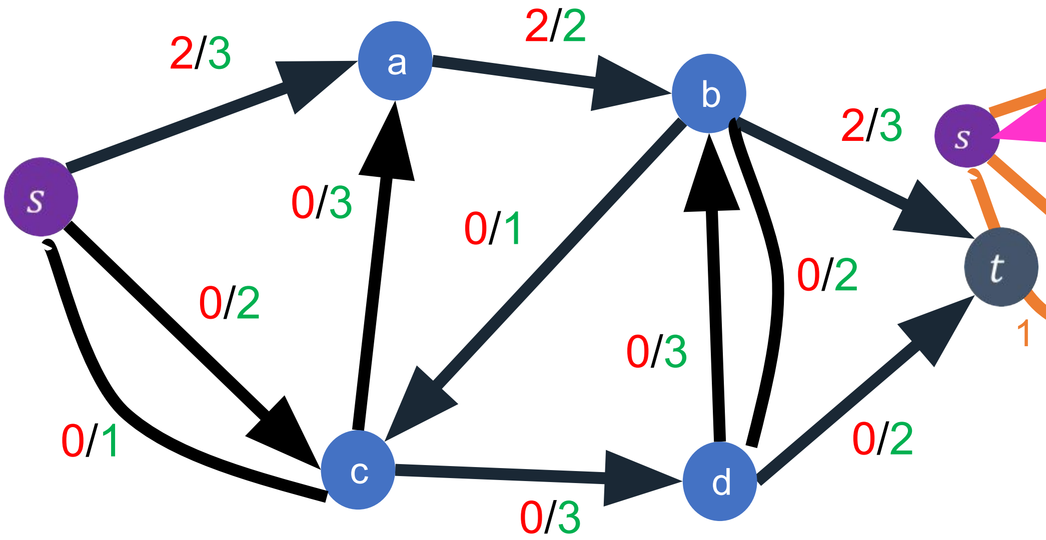


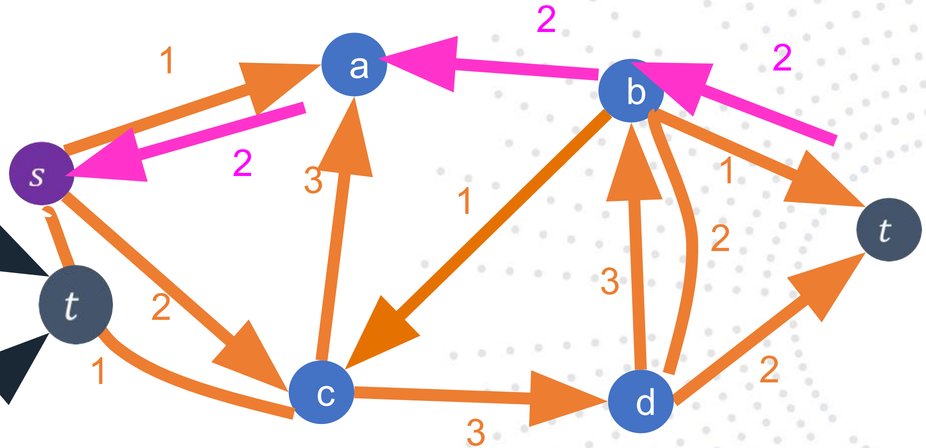

Residual graph $G_f$



43

# Ford-Fulkerson Algorithm- Example 3 Sol.1

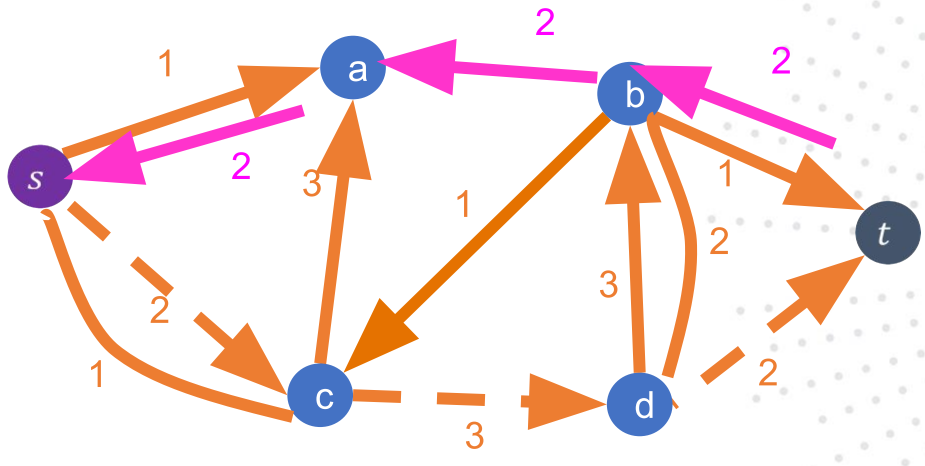



Residual graph $G_f$

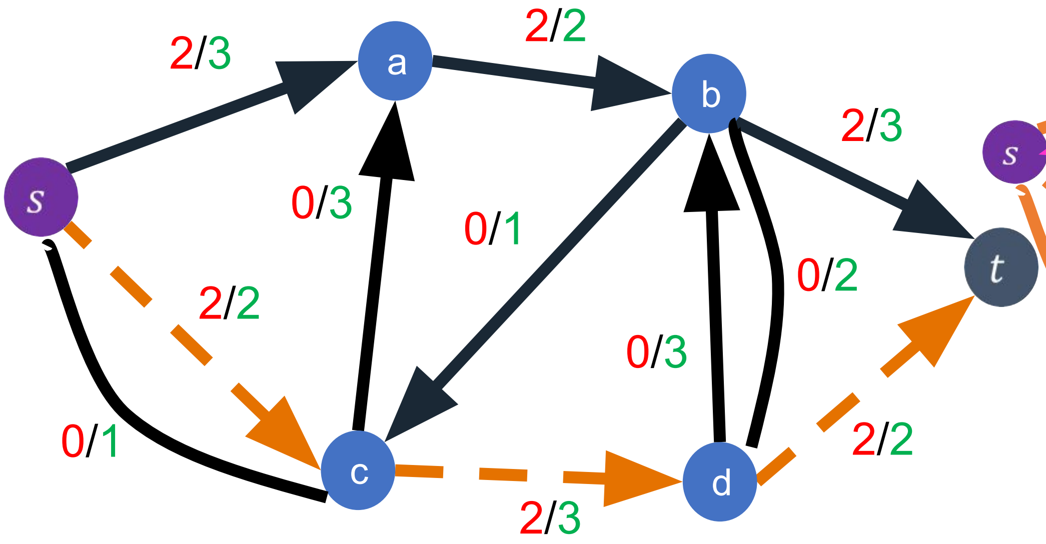
# Ford-Fulkerson Algorithm- Example 3 Sol.1

Increase flow by 2 unit

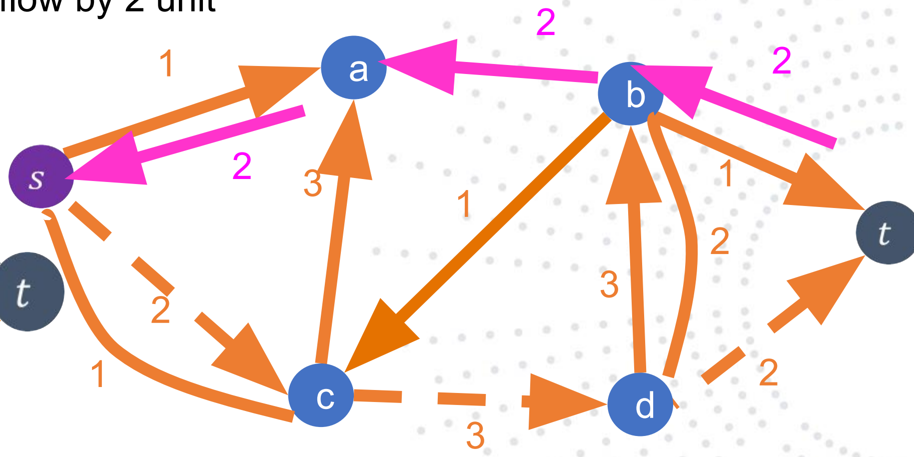



Residual graph $G_f$

# Ford-Fulkerson Algorithm- Example 3 Sol.1



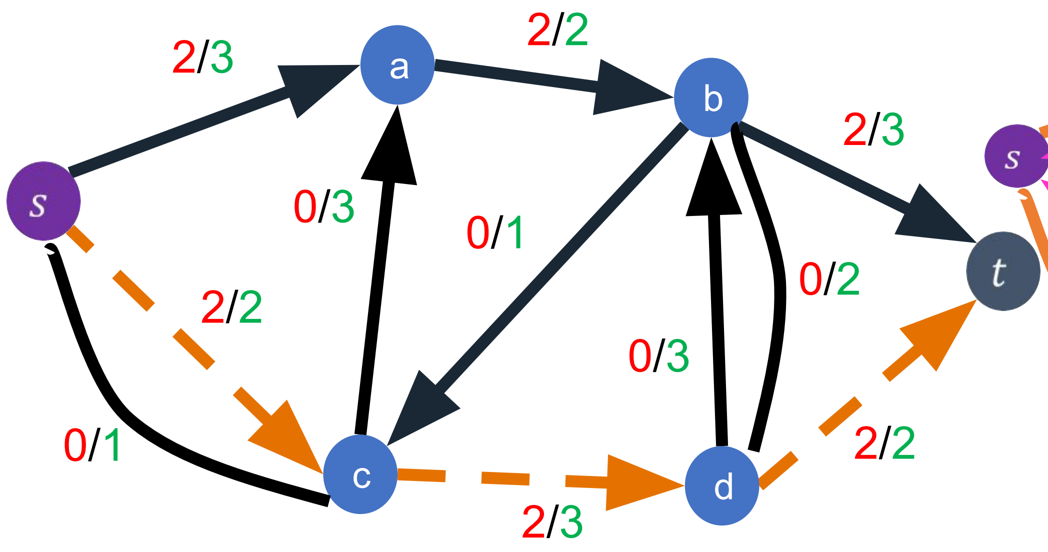

Increase flow by 2 unit

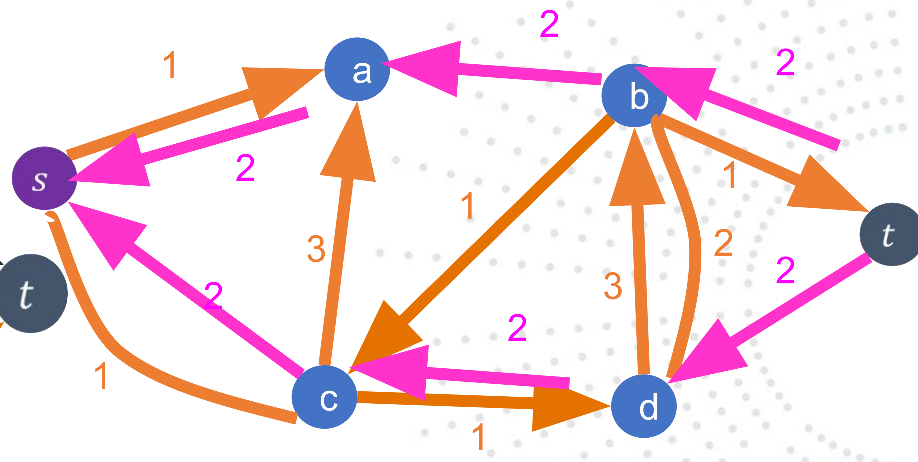



Residual graph $G_f$


46

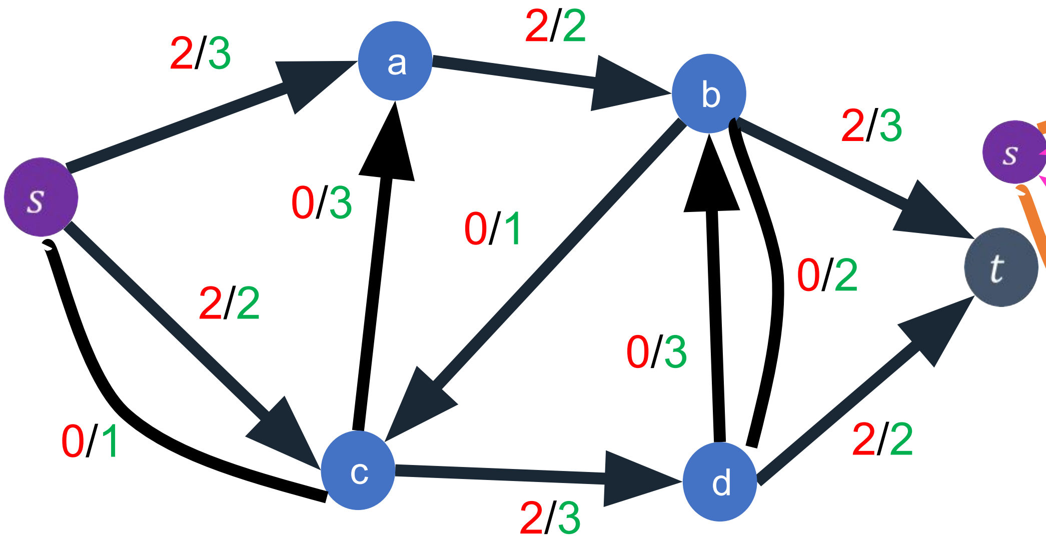
# Ford-Fulkerson Algorithm- Example 3 Sol.1



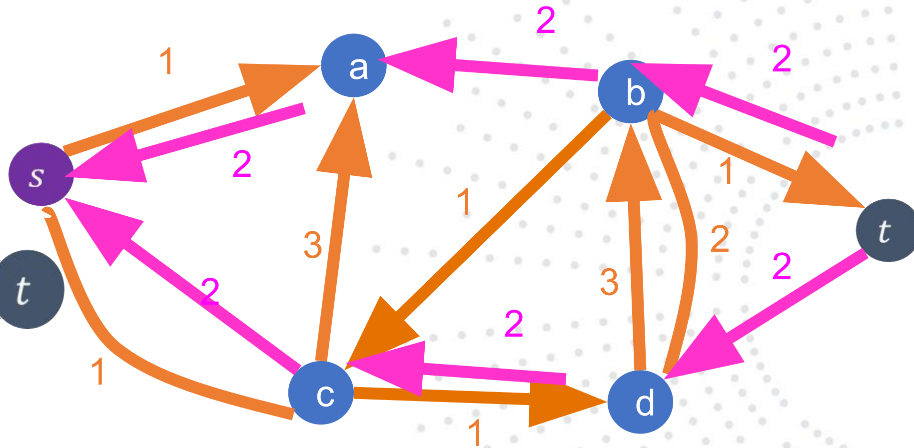




Residual graph $G_f$

# Ford-Fulkerson Algorithm- Example 3 Sol.1



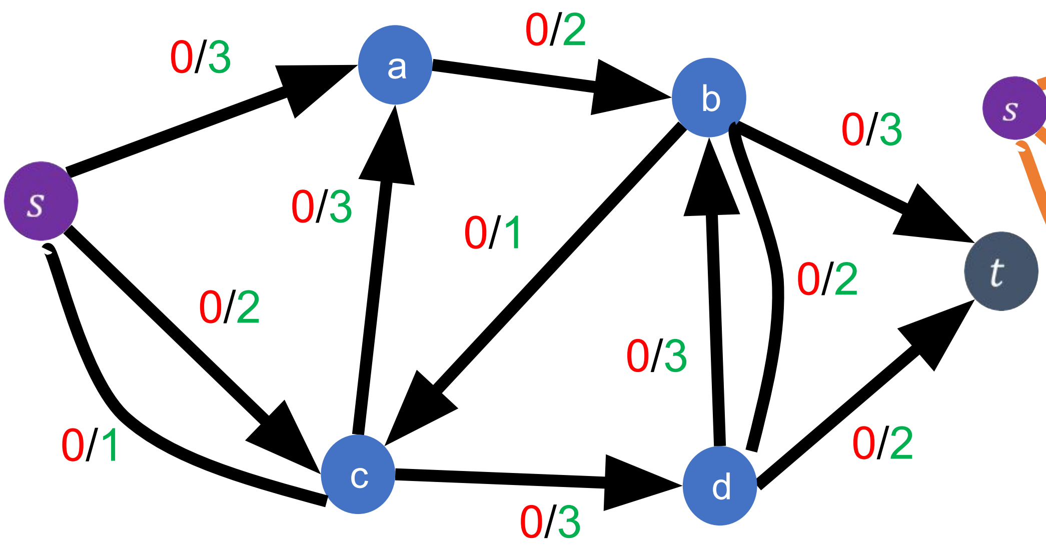

**No more augmented paths**

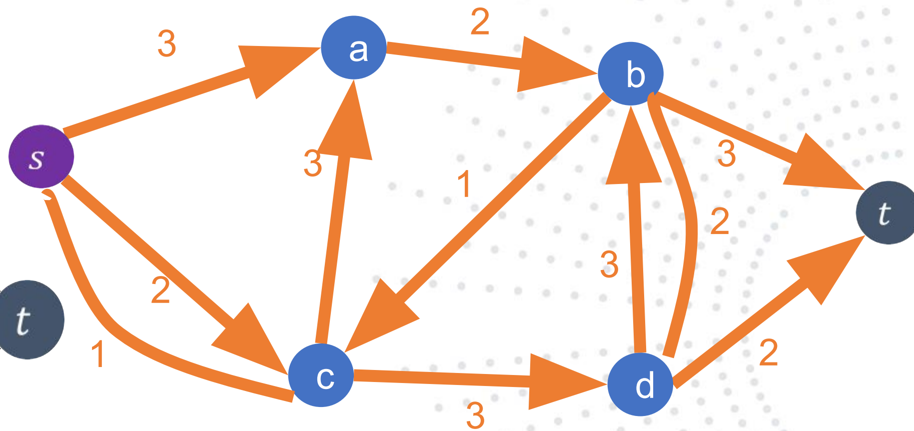



Residual graph $G_f$

**Maximum flow:** 4

# Ford-Fulkerson Algorithm- Example 3 Sol.2



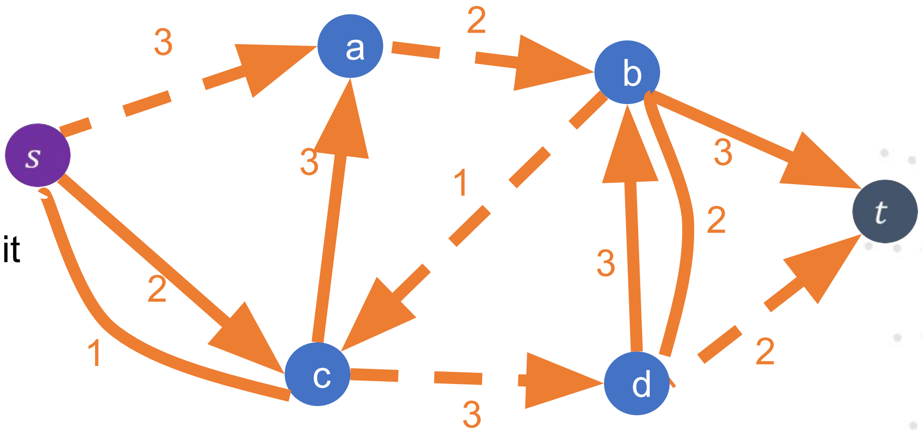




Residual graph $G_f$

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

# Ford-Fulkerson Algorithm- Example 3 Sol.2

Increase flow by 1 unit



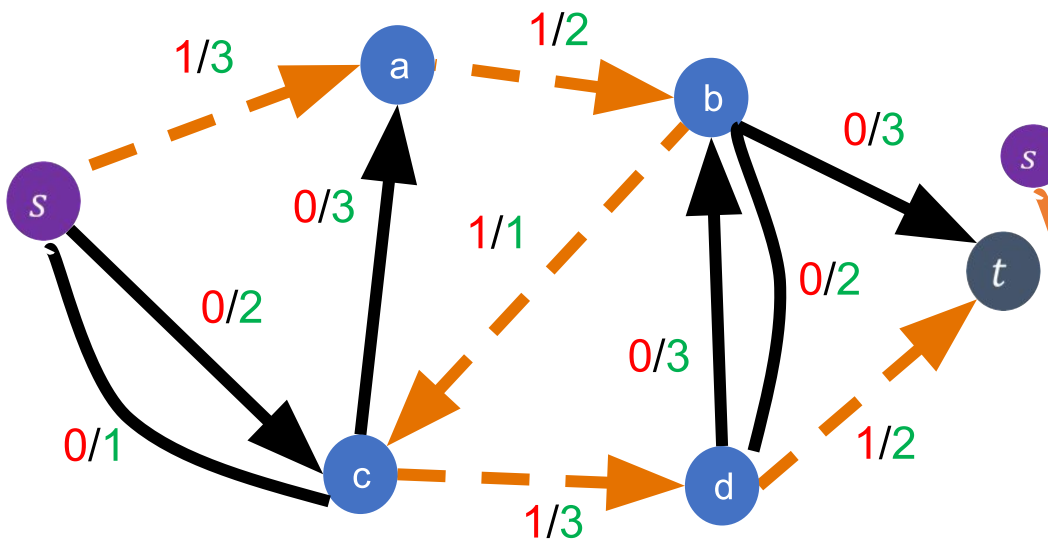

Residual graph $G_f$

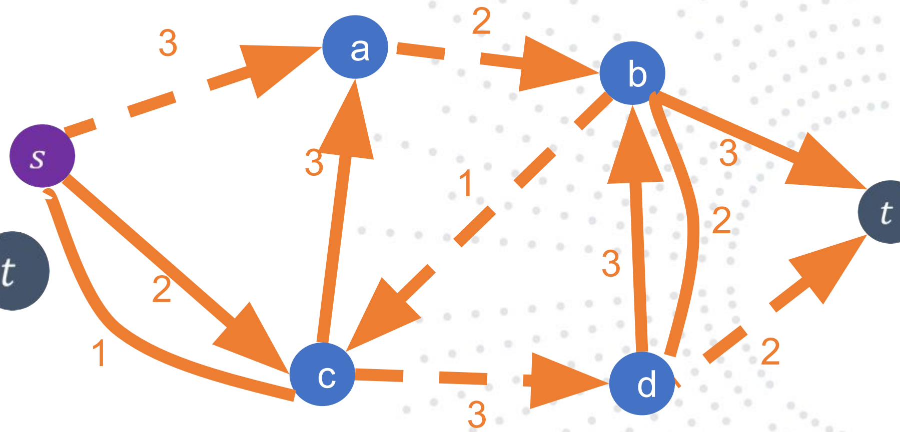


50

# Ford-Fulkerson Algorithm- Example 3 Sol.2

Increase flow by 1 unit



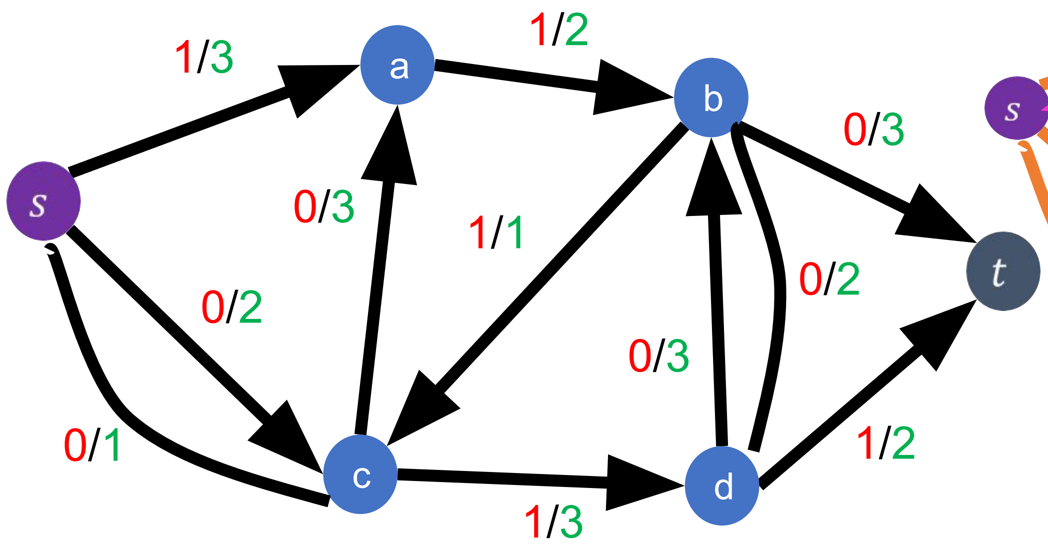



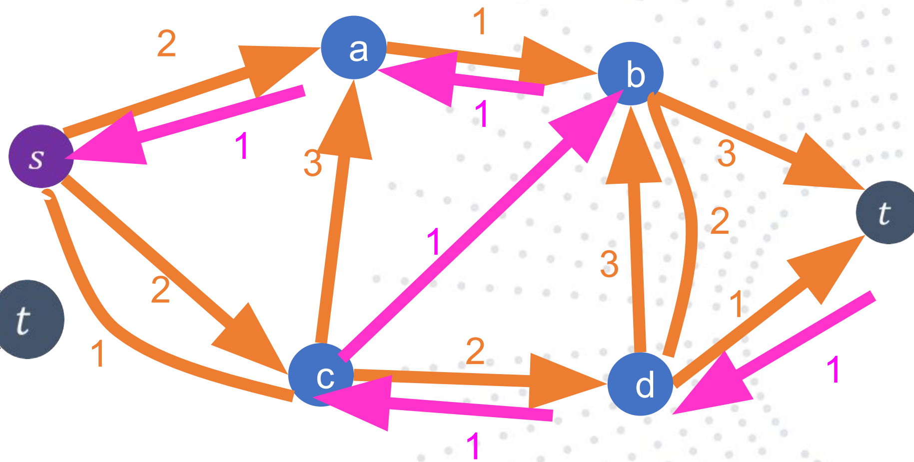

Residual graph $G_f$



51

# Ford-Fulkerson Algorithm- Example 3 Sol.2

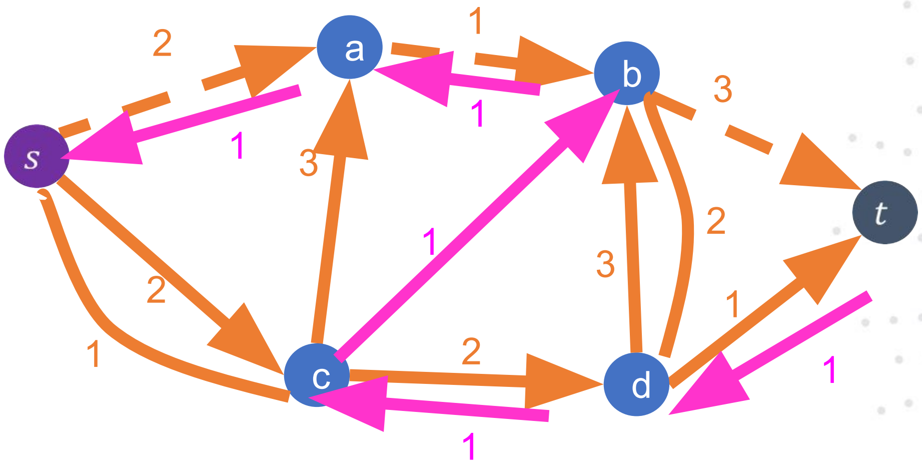



Residual graph $G_f$

# Ford-Fulkerson Algorithm- Example 3 Sol.2

Increase flow by 1 unit



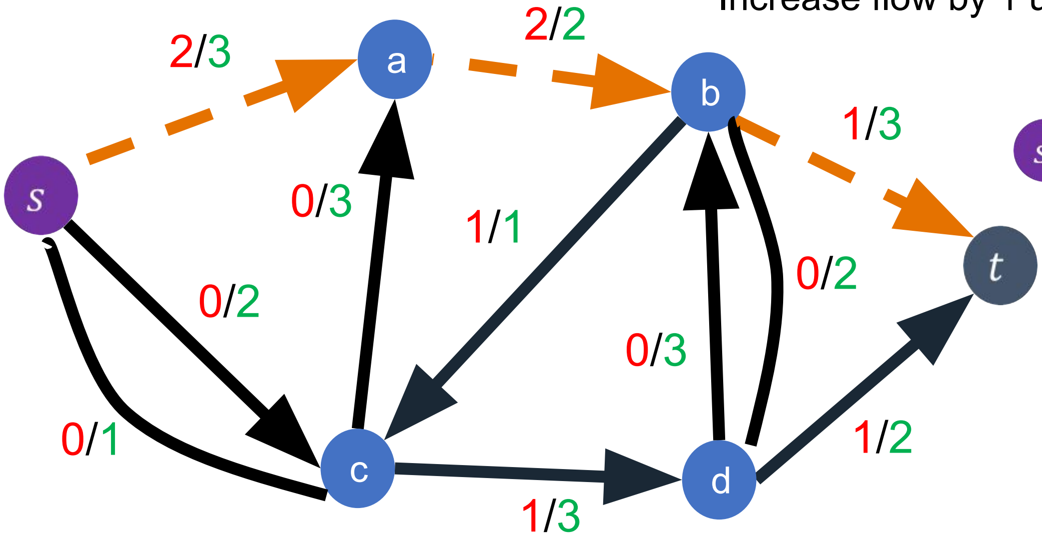

Residual graph $G_f$

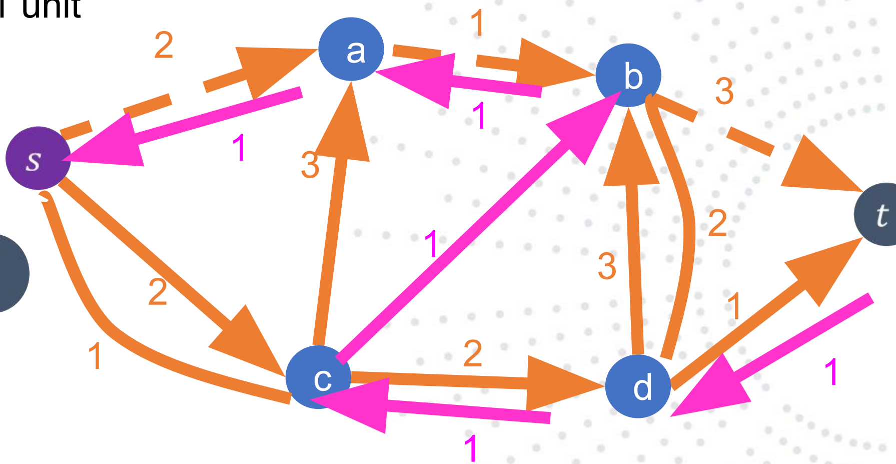


53

# Ford-Fulkerson Algorithm- Example 3 Sol.2

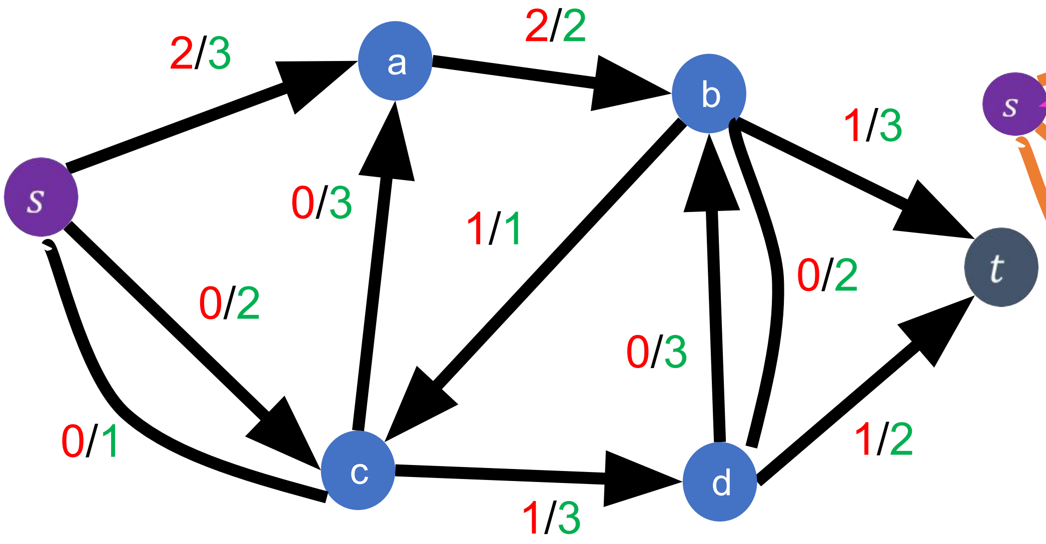
Increase flow by 1 unit

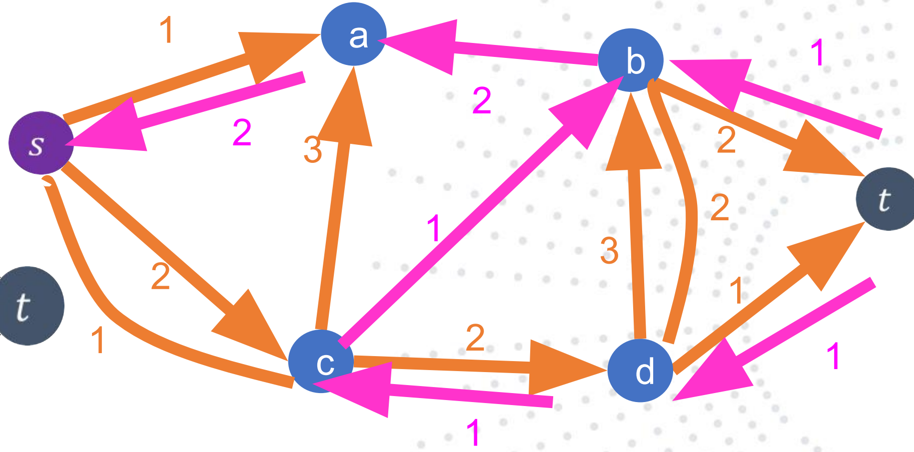



Residual graph $G_f$

# Ford-Fulkerson Algorithm- Example 3 Sol.2



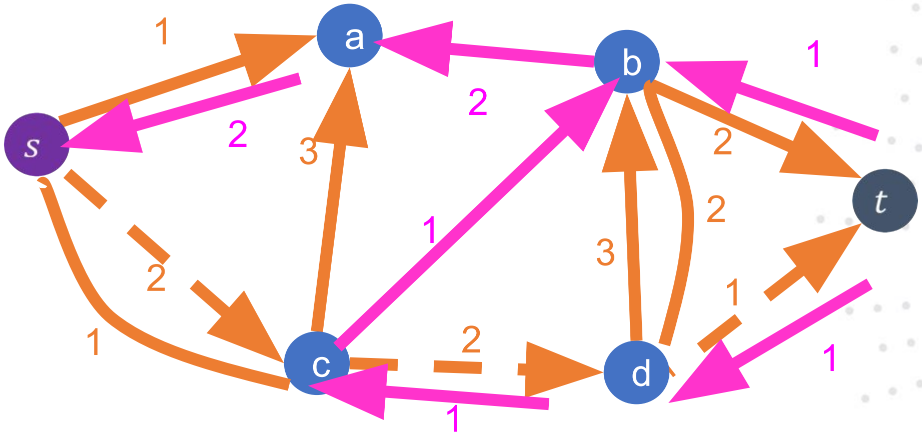




Residual graph $G_f$


55

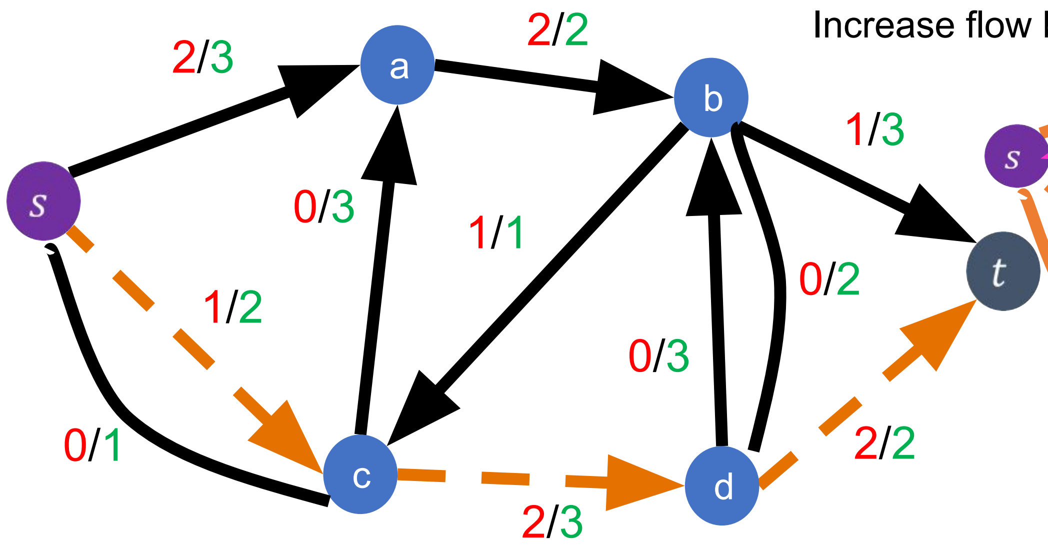
# Ford-Fulkerson Algorithm- Example 3 Sol.2

Increase flow by 1 unit

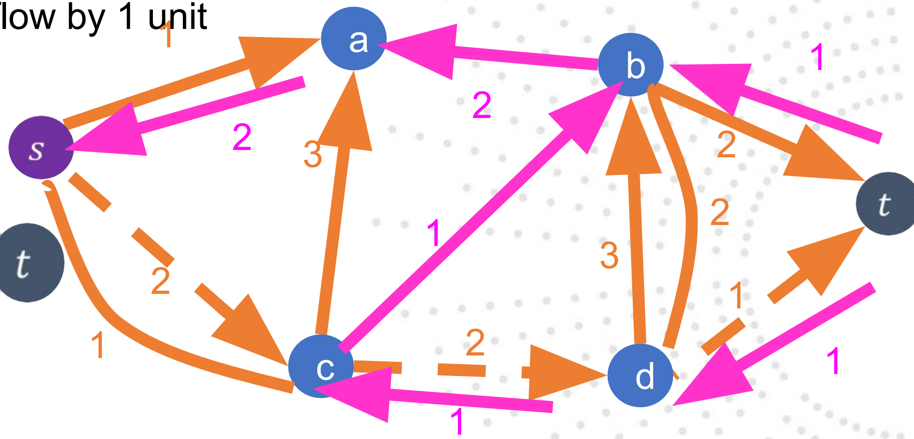



Residual graph $G_f$

# Ford-Fulkerson Algorithm- Example 3 Sol.2

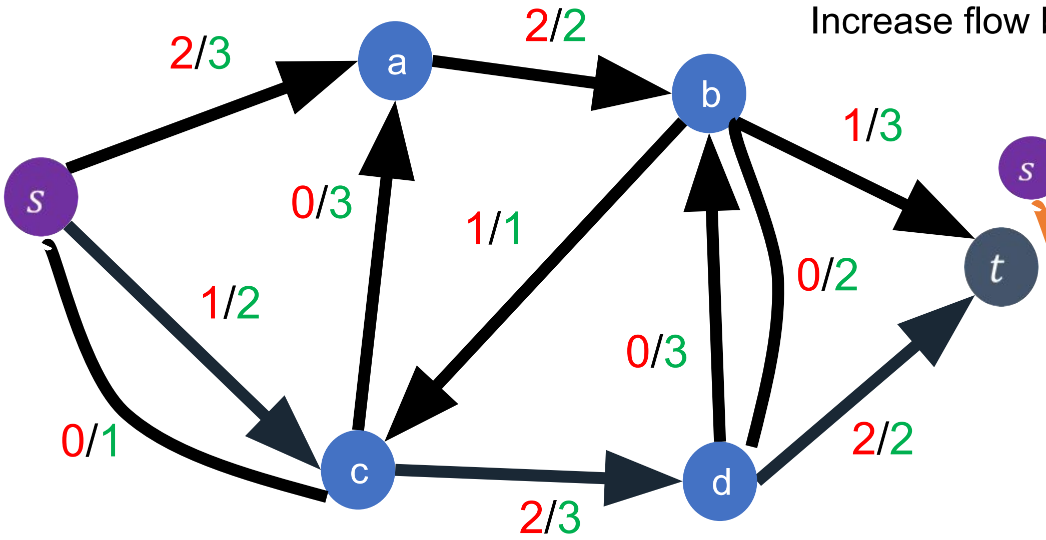



Increase flow by 1 unit

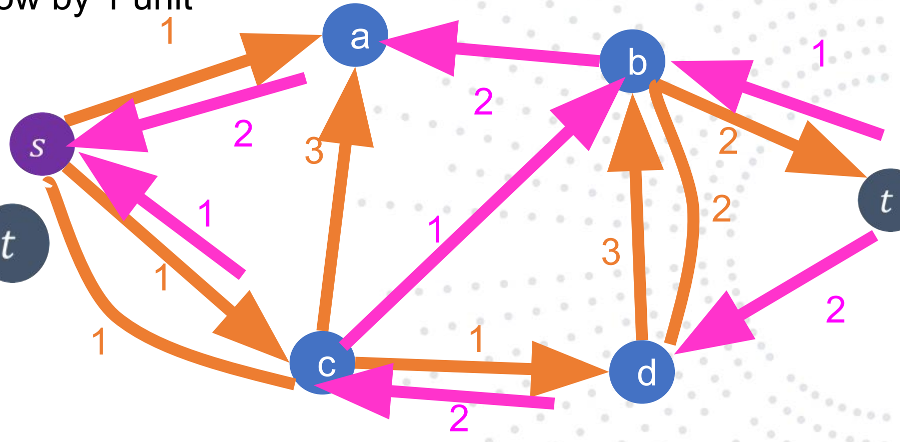



Residual graph $G_f$

# Ford-Fulkerson Algorithm- Example 3 Sol.2



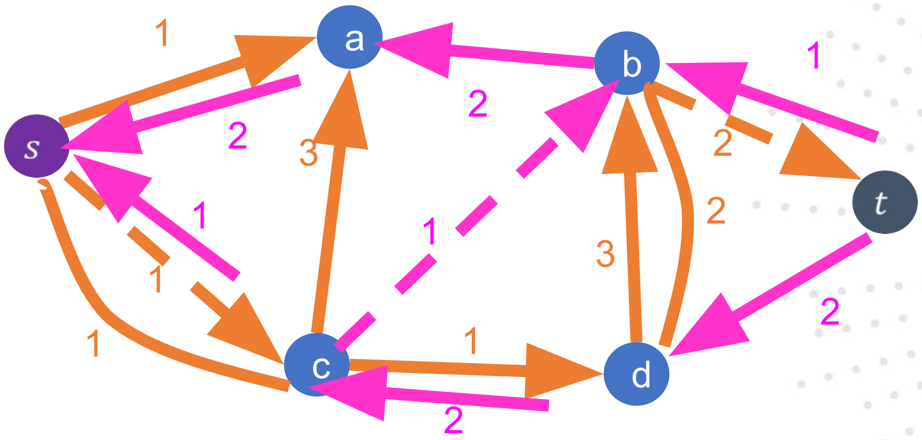

Increase flow by 1 unit




Residual graph $G_f$

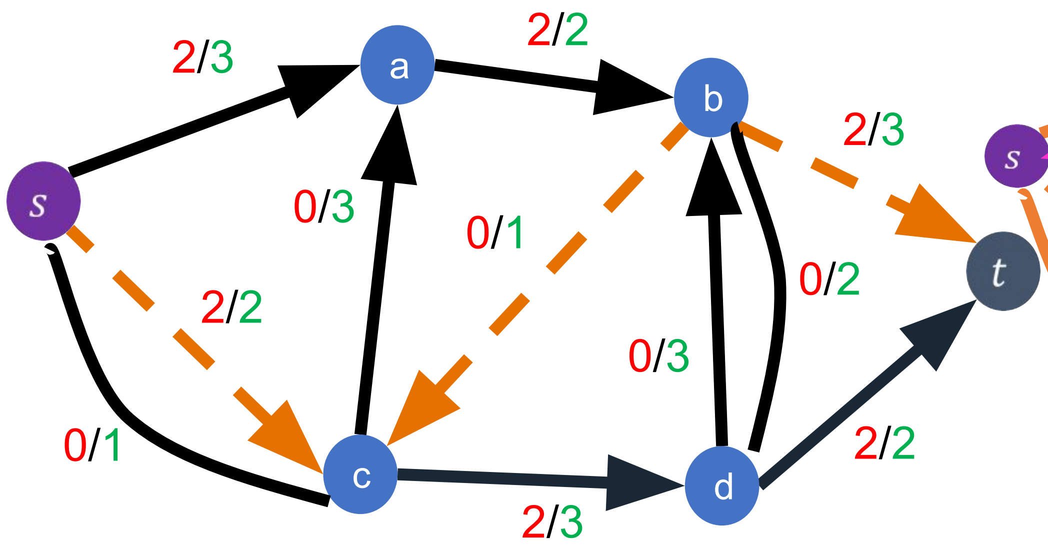
# Ford-Fulkerson Algorithm- Example 3 Sol.2

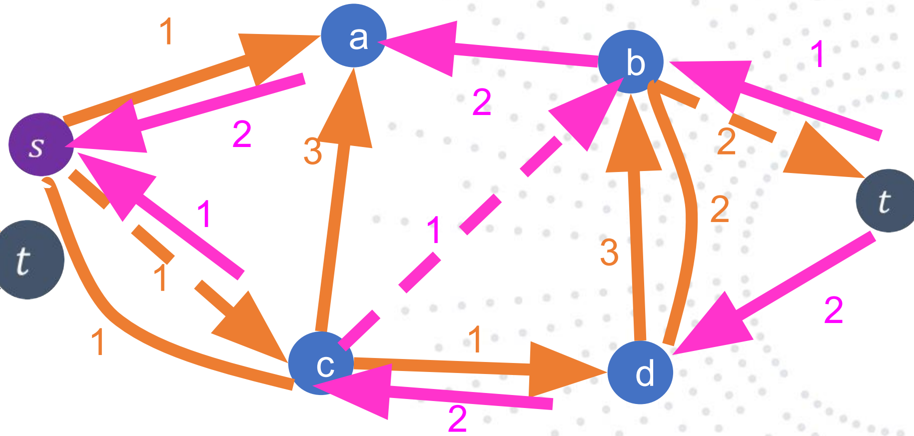
Increase flow by 1 unit




Residual graph $G_f$

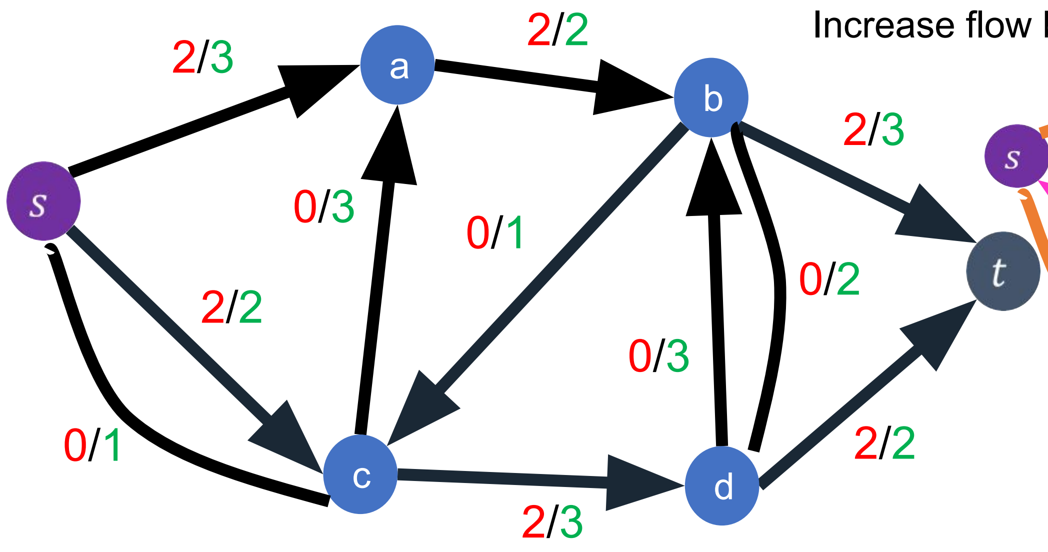
# Ford-Fulkerson Algorithm- Example 3 Sol.2



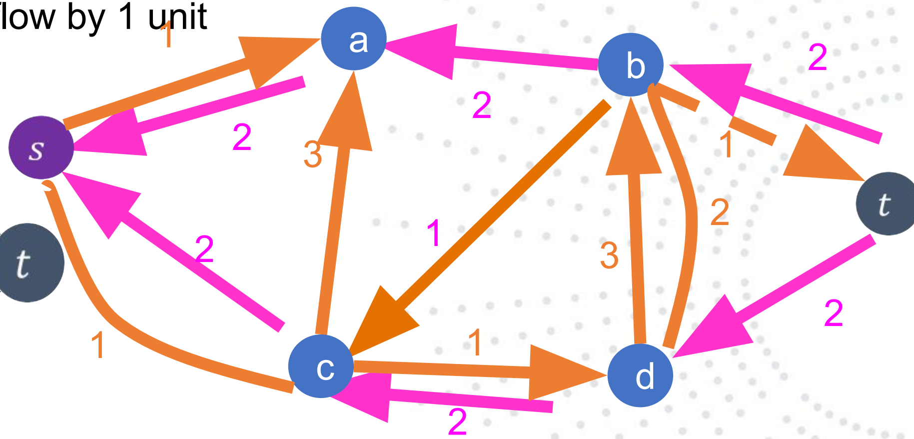

Residual graph $G_f$

# Ford-Fulkerson Algorithm- Example 3 Sol.2




Increase flow by 1 unit

**No more augmented paths**



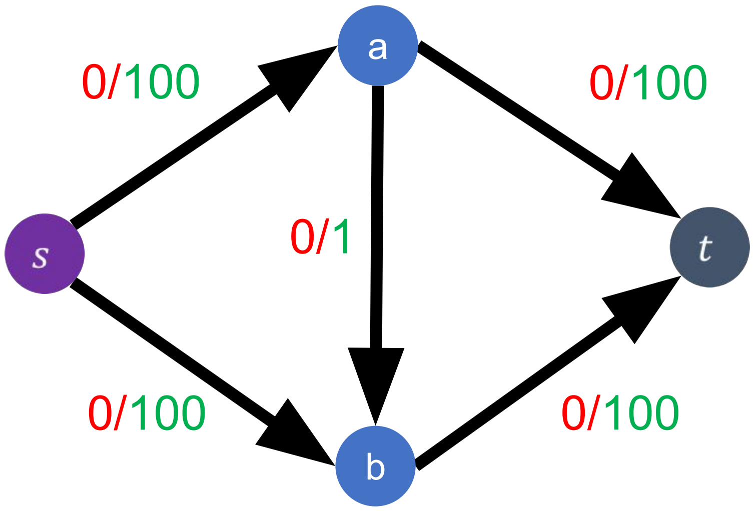

Residual graph $G_f$

**Maximum flow:** 4

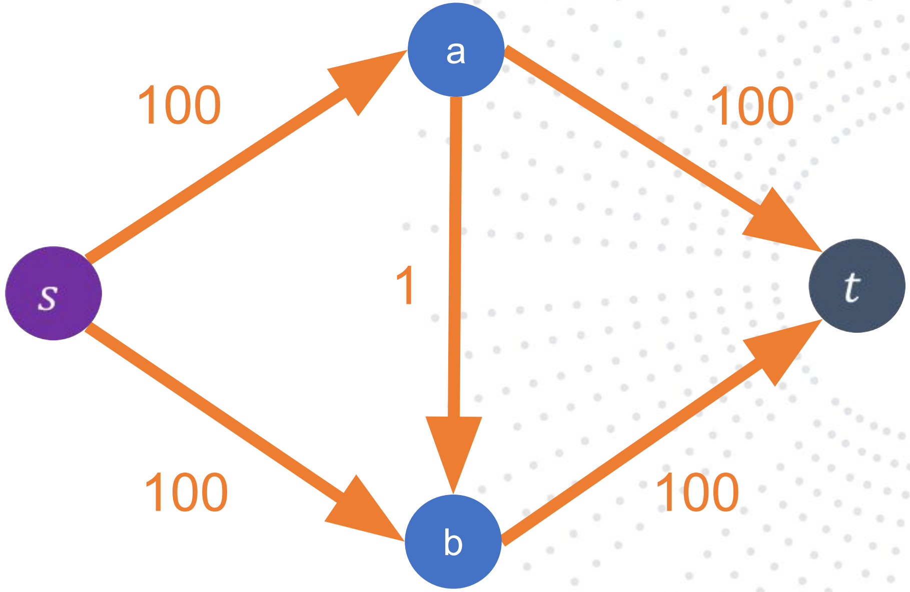
# Worst-Case Ford-Fulkerson Algorithm

- 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

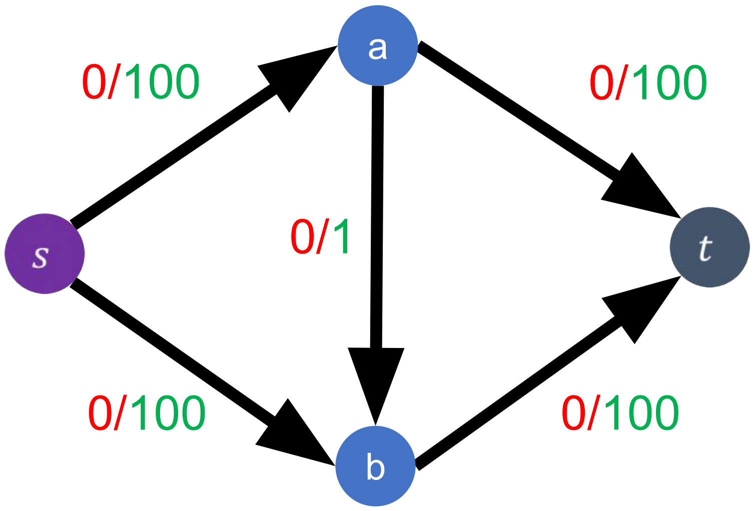



Flow graph



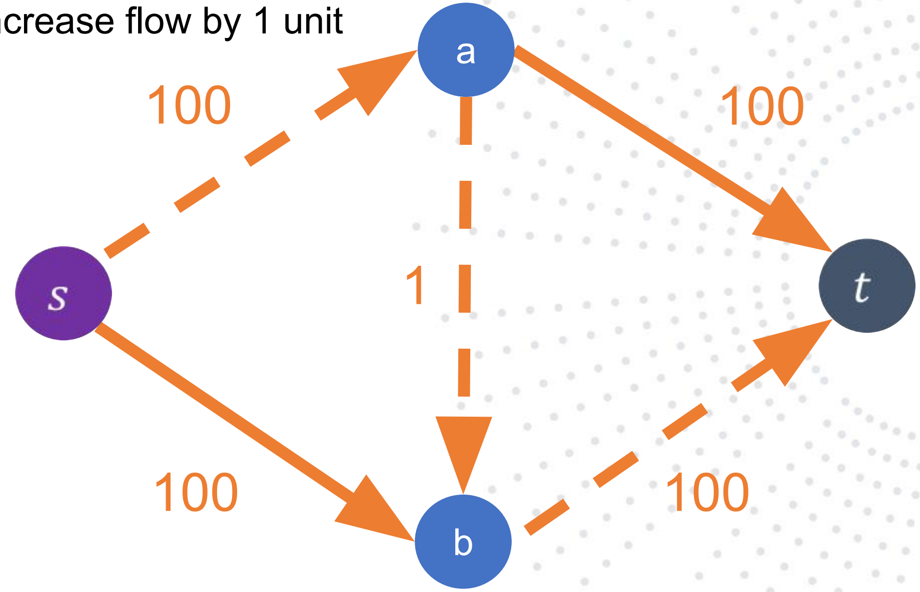

Residual graph

# Worst-Case Ford-Fulkerson Algorithm



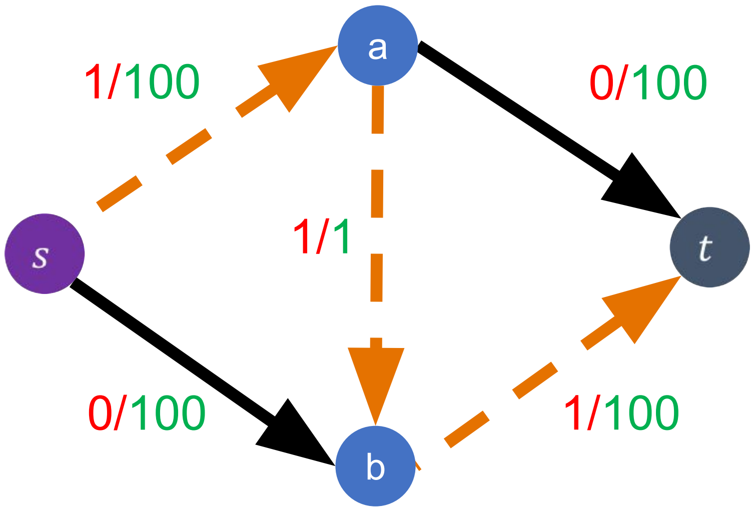

Flow graph

Increase flow by 1 unit



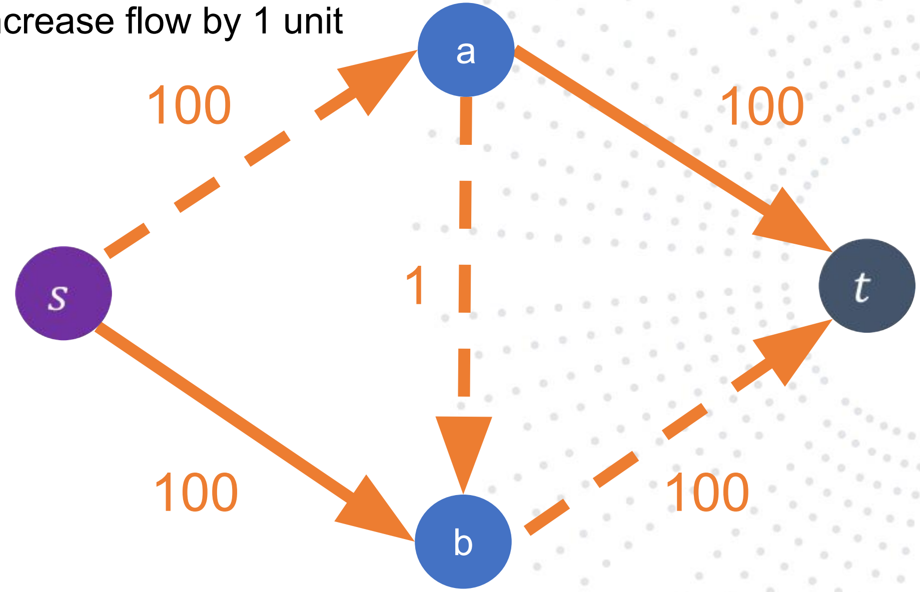

Residual graph

# Worst-Case Ford-Fulkerson Algorithm



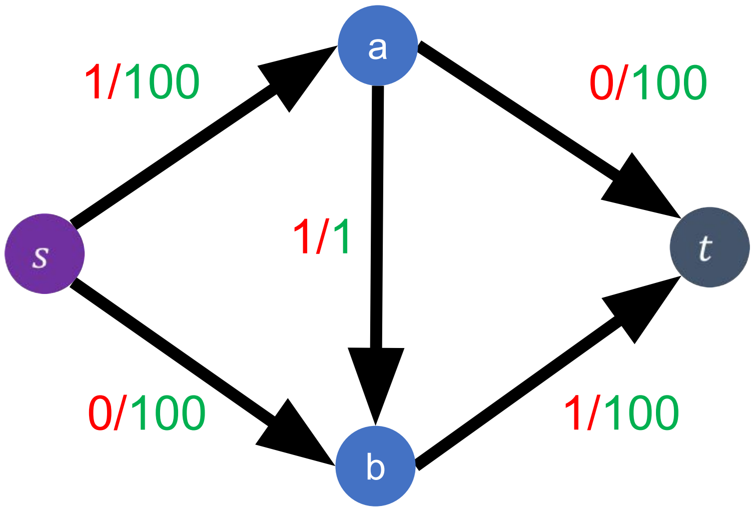
Flow graph

Increase flow by 1 unit

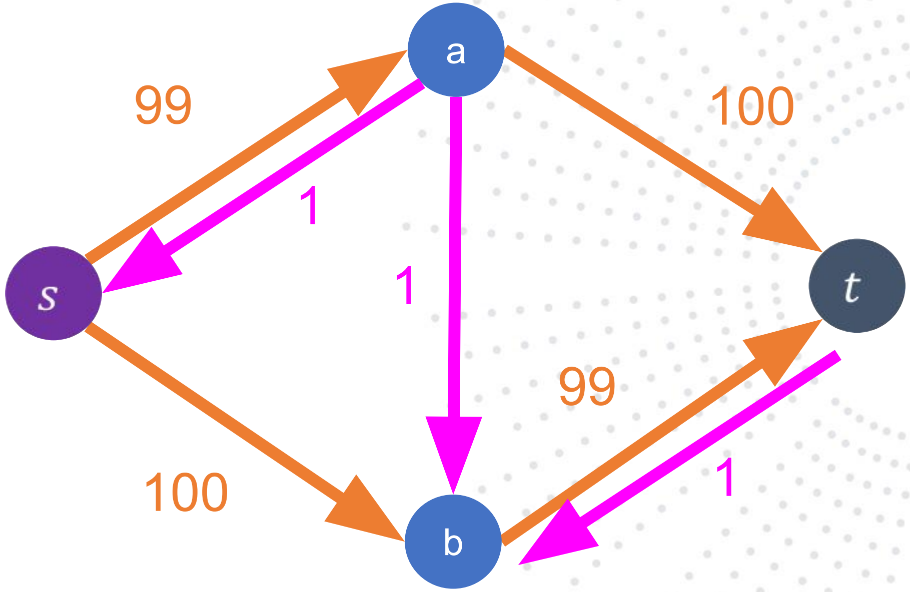


Residual graph

# Worst-Case Ford-Fulkerson Algorithm

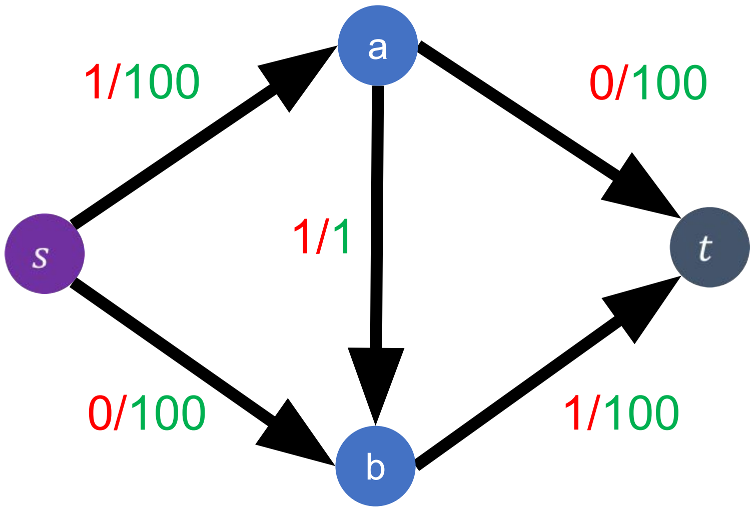



Flow graph



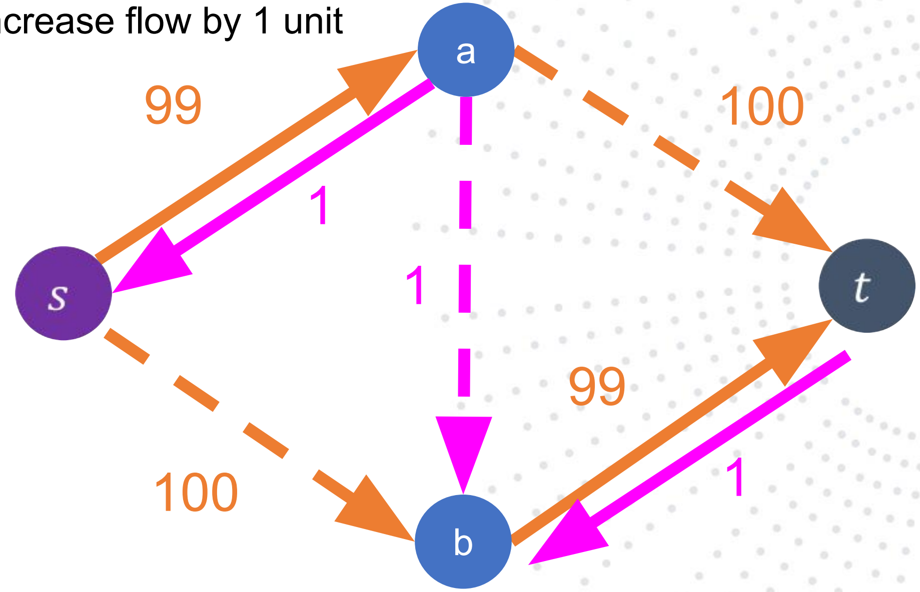

Residual graph

# Worst-Case Ford-Fulkerson Algorithm



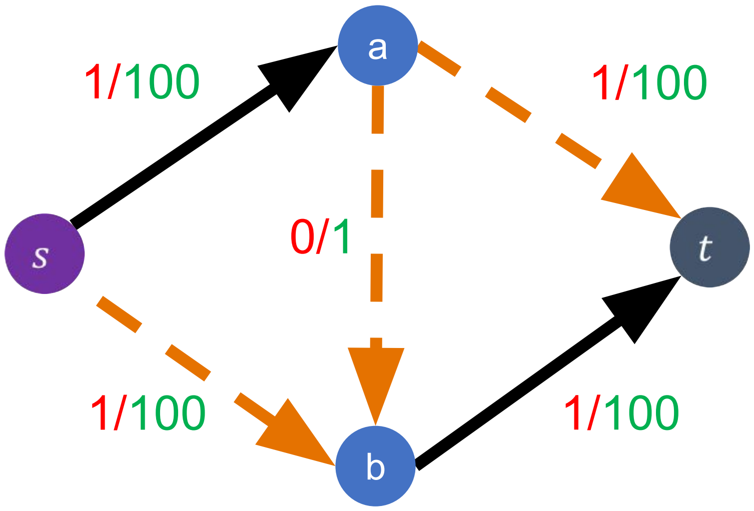

Flow graph

Increase flow by 1 unit



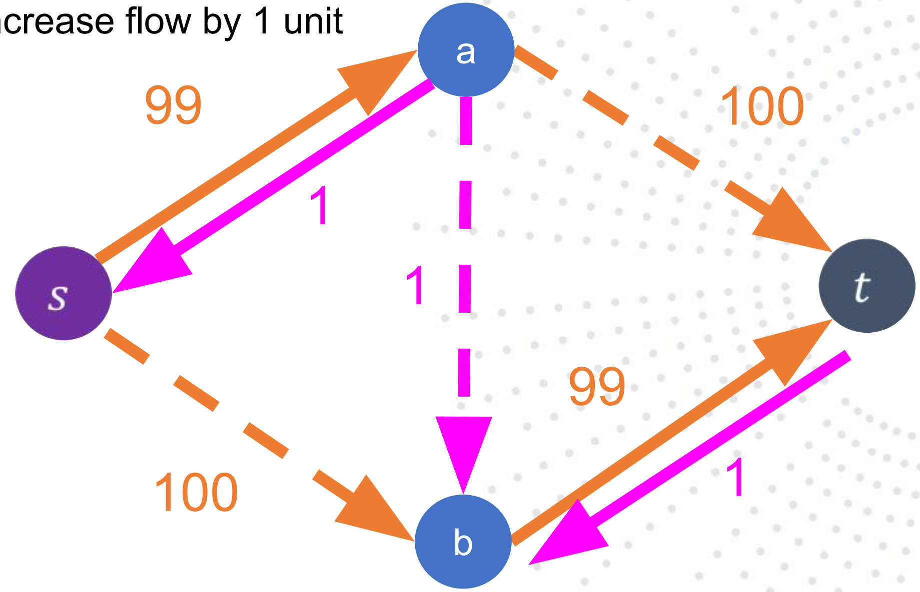

Residual graph

# Worst-Case Ford-Fulkerson Algorithm



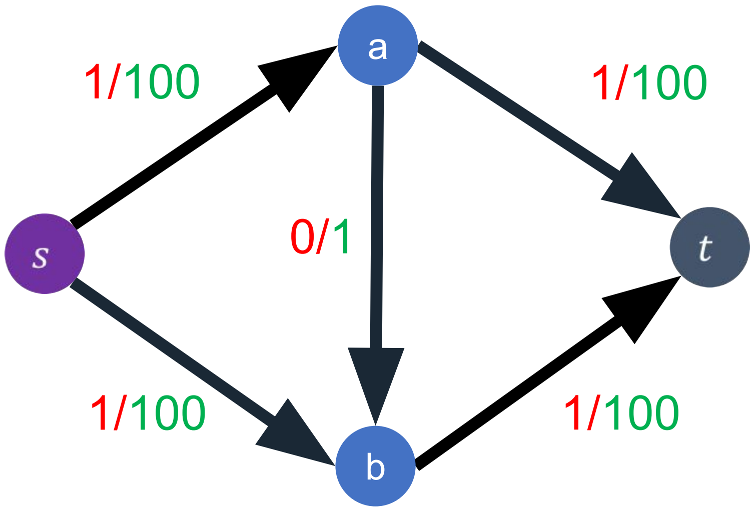

Flow graph

Increase flow by 1 unit

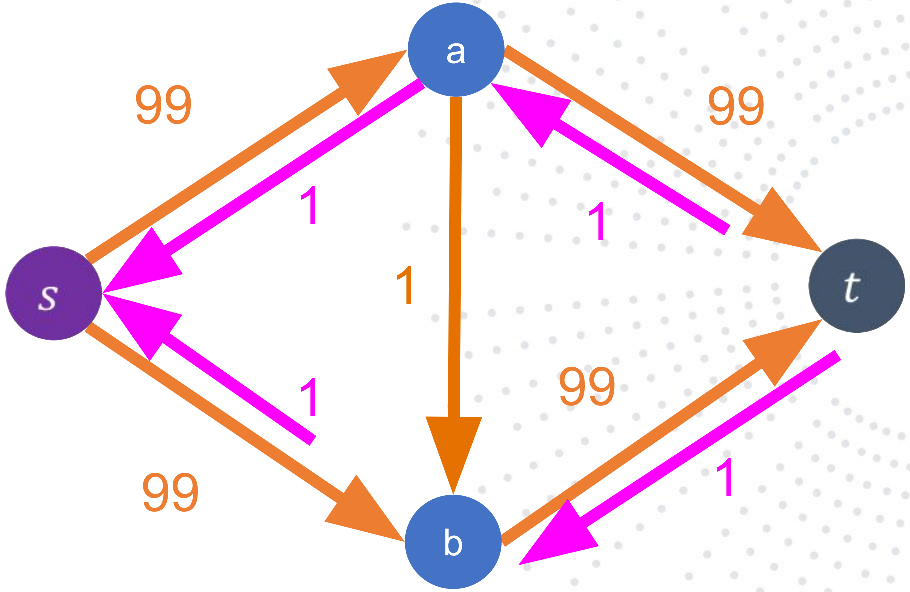



Residual graph

# Worst-Case Ford-Fulkerson Algorithm

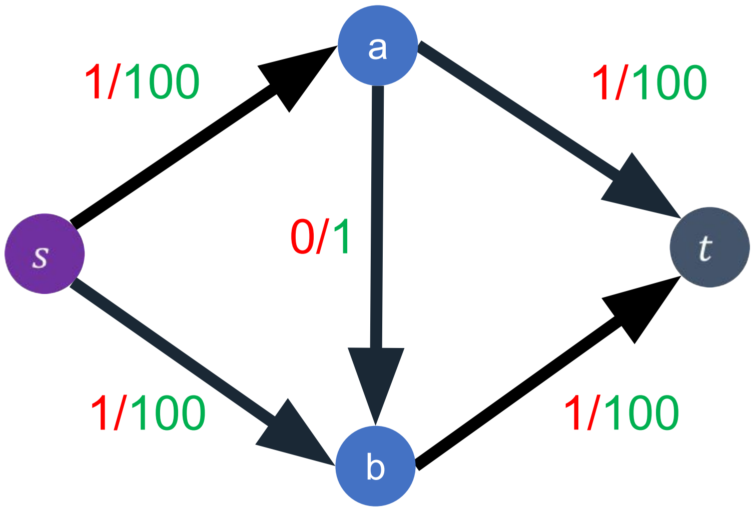



Flow graph



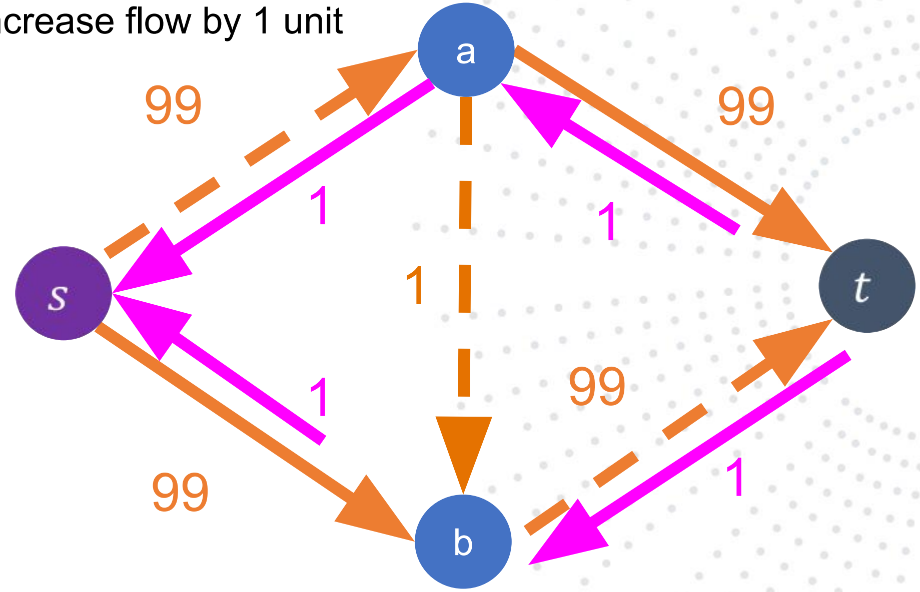

Residual graph

# Worst-Case Ford-Fulkerson Algorithm




Flow graph

Increase flow by 1 unit




Residual graph

# Ford-Fulkerson algorithm

- **Worst Case (Exponential Time Complexity)**:
  - O(E · 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 · 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

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 $G_f$)

# Edmonds-Karp Algorithm

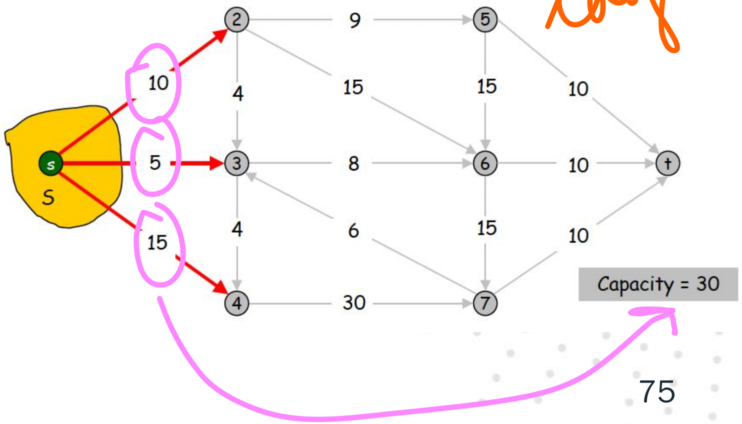
- **Edmonds-Karp algorithm** has a time complexity of $O(V \cdot E^2)$
    - BFS takes $O(V+E)$ time per iteration
    - Each augmenting path can increase the flow by a finite amount.
    - The maximum number of BFS iterations is $O(V \cdot E)$, because each edge is involved at most V times.
- Since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

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

# Minimum Cut Problem

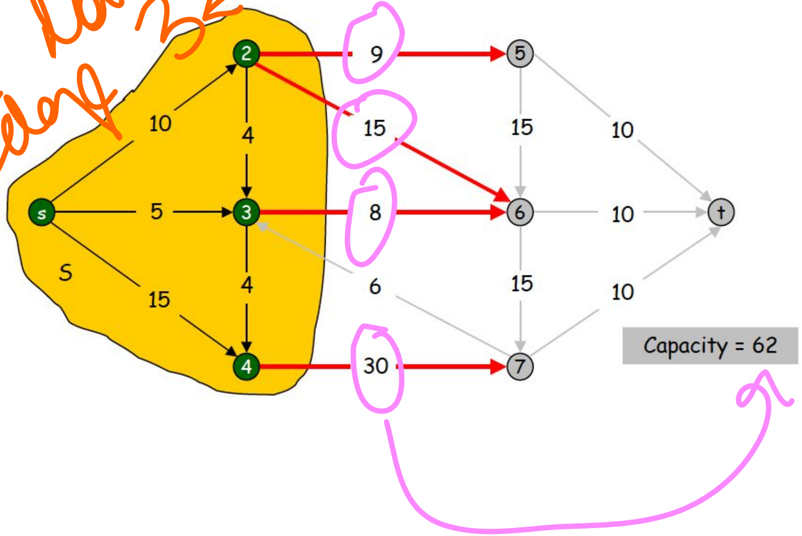
- Given a graph G=(V,E) with vertices V and edges E, and two designated vertices s (source) and t (sink)
- The **min-cut** is the smallest set of edges whose removal disconnects the graph or separates s from t.
- A cut in a graph is a partition of the vertices V into two disjoint subsets S and T
    - *s is in S and t is in T*
    - S ∪ T=V and S ∩ T=∅
- The **capacity of a cut** is the sum of the weights crossing from S to T
    - *capacity(S, T) = sum of weights of edges leaving S*

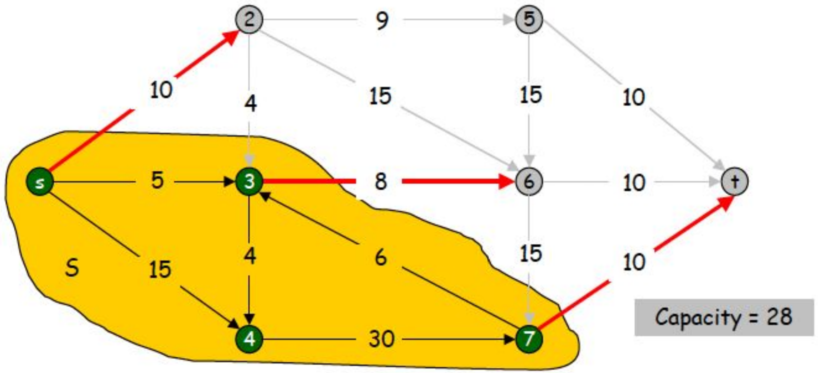
We count the leaving edges



# Minimum Cut Problem

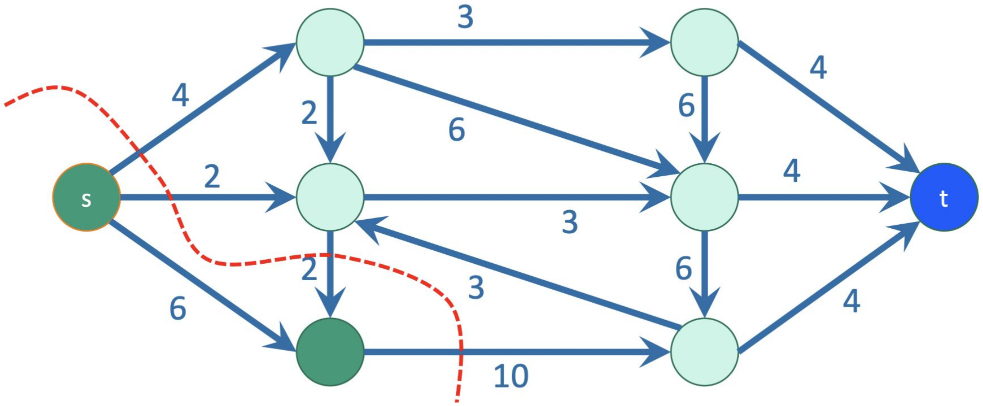
we don't include edge 3←7 b/c it leaves 7, not 3



# Minimum Cut Problem

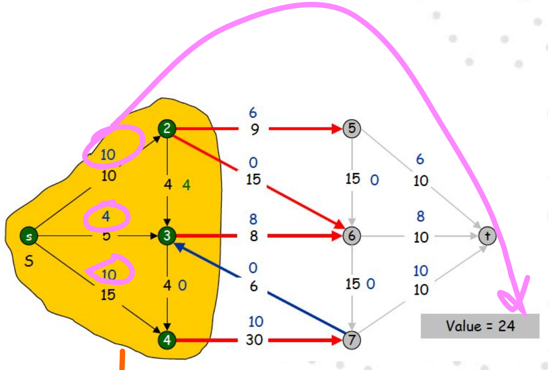
- An edge crosses the cut if it goes from s's side to t's side




This cut has capacity =>
4 + 2 + 10 = 16

# Flows and Cuts

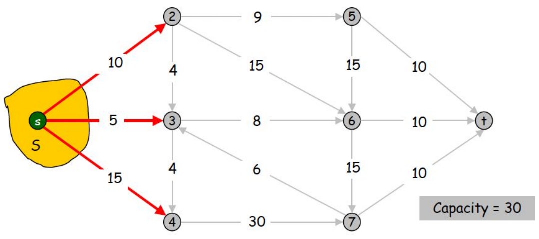
- Let f be a flow, and let (S, T) be any s-t cut. Then, the net flow sent across the cut is equal to the amount reaching t




net flow is different from the cut

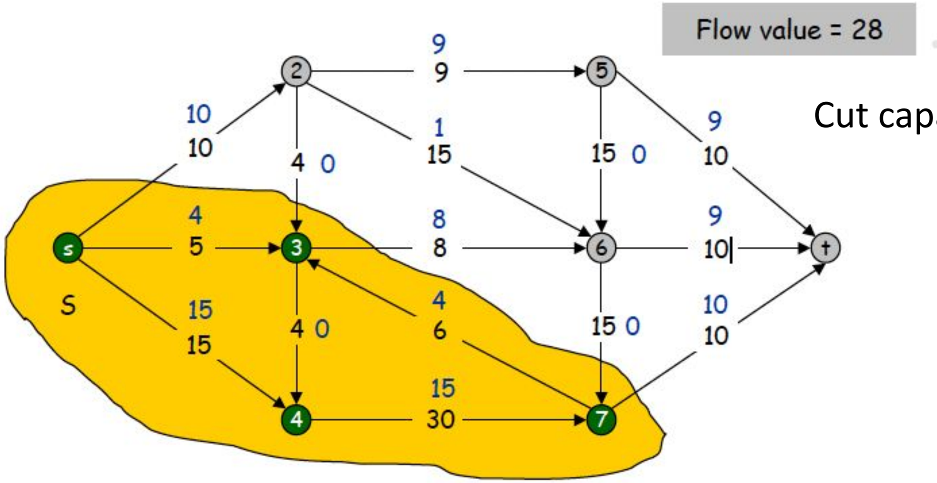
Cut capacity = 30    Flow value <= 30

- Let f be a flow, and let (S, T) be any s-t cut. Then the value of the flow is at most the capacity of the cut



# Max Flow and Min Cut

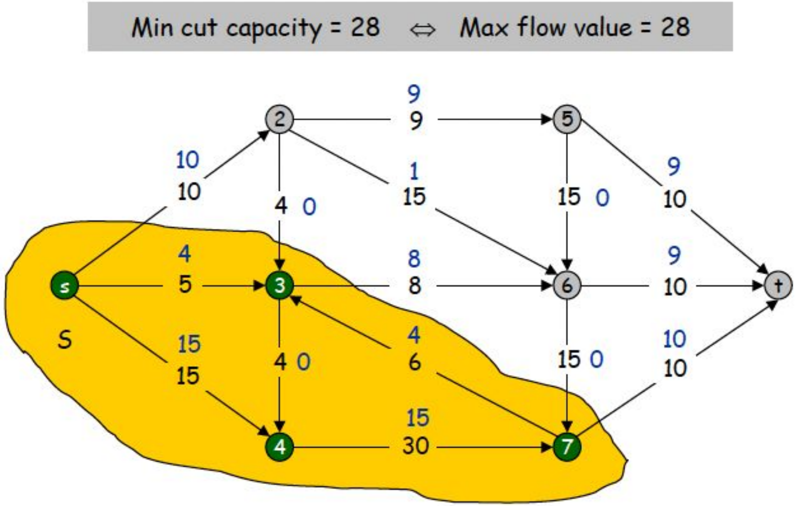
- Let f be a flow, and let (S, T) be an s-t cut whose capacity equals the value of f. Then f is a max flow and (S, T) is a min cut




Cut capacity = 28    Flow value <= 28

# Max-Flow Min-Cut Theorem

Max-flow min-cut theorem. (Ford-Fulkerson, 1956): In any network, the value of max flow equals capacity of min cut



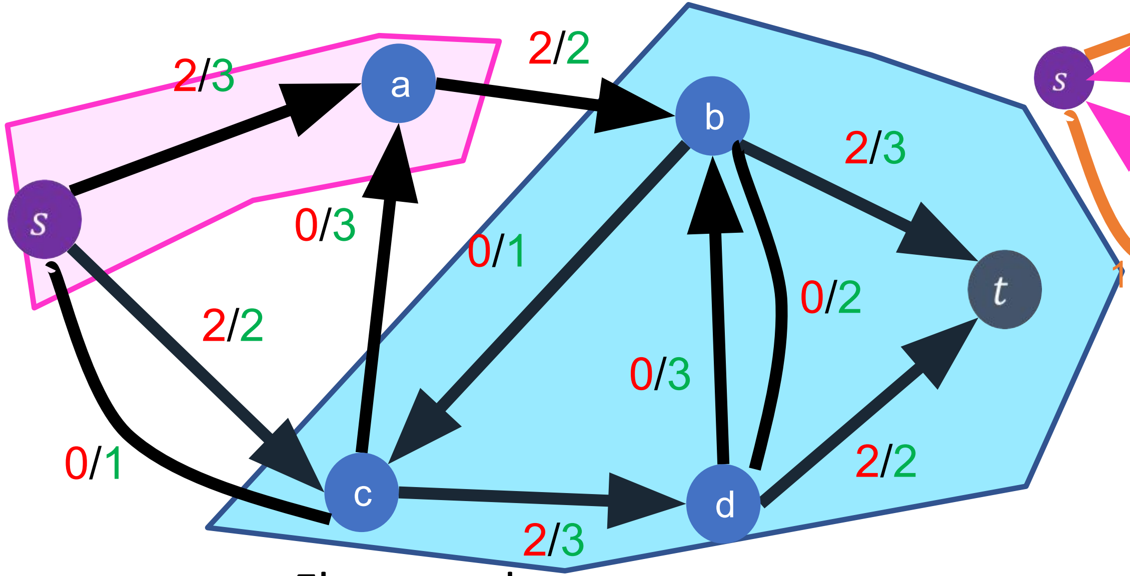
# Max-Flow Min-Cut Theorem

Let $f$ be a flow in a graph $G$

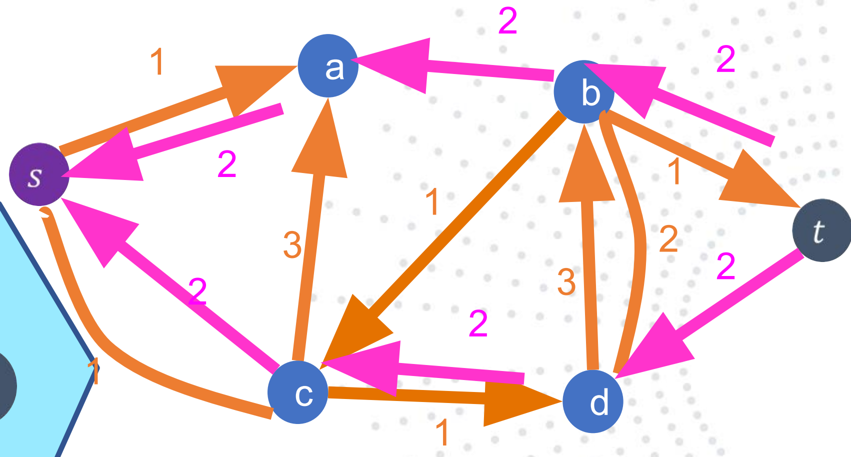
$f$ is a maximum flow in $G$ → there are no augmenting paths in the residual graph $G_f$ → there exists a cut $(S, T)$ of $G$ where $|f| = ||S, T||$

Augmenting path theorem: A flow f is a max flow if and only if there are no augmenting paths

# Max-Flow Min-Cut Theorem




Flow graph




Residual graph

When there are no more augmenting paths in the graph, there is a cut whose cost/ capacity matches the flow

**Max flow:4**

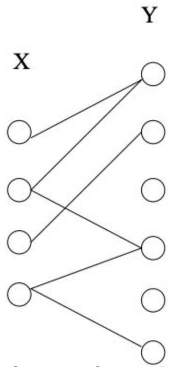
**Min cut: 4**

# Why Study Flow Networks?

- Unlike divide-and-conquer, greedy, or DP, flow network doesn't seem like an algorithmic framework
  - It seems more like a single problem
- How this single problem be useful?

# Bipartite Matching Problem

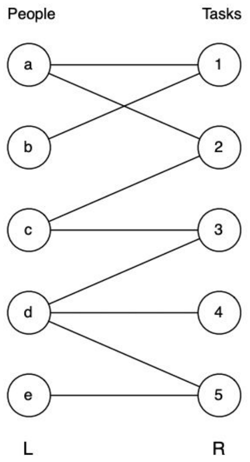
- A Bipartite Graph G = (V, E) is a graph in which the vertex set V can be divided into two disjoint subsets X and Y such that every edge e ∈ E has one endpoint in X and the other end point in Y
- A matching M is a subset of edges such that each node in V appears in at most one edge in M




Bipartite Graph

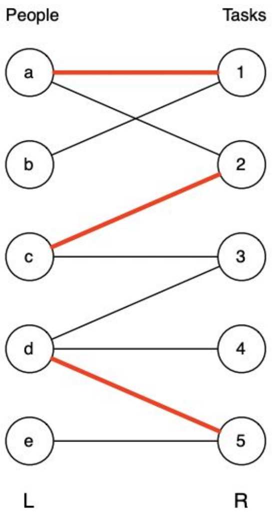
# Bipartite Matching Problem

- Given a set of people L and set of jobs R
- Each person can do only some of the jobs
- How to model this relation as bipartite graph?



# Bipartite Matching Problem

- A matching gives an assignment of people to tasks
- We want to get as many tasks done as possible
- So, want a maximum matching: one that contains as many edges as possible



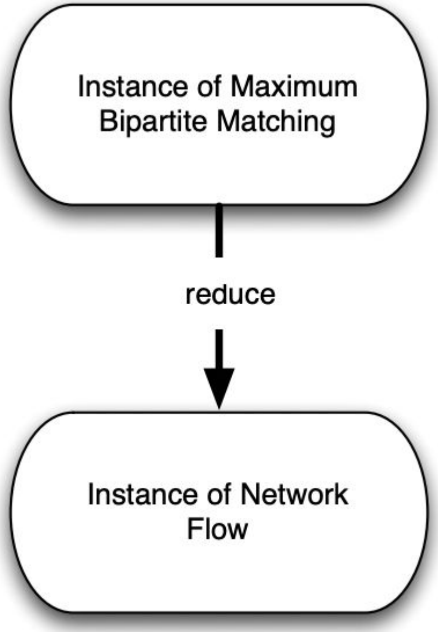

This one is not maximum

# Bipartite Matching Problem

- Our goal is to get the **maximum matching**
- **Maximum matching** is a matching that contains the **maximum number of edges** possible. This is the most "complete" matching in terms of the number of edges that can be added without violating the matching condition

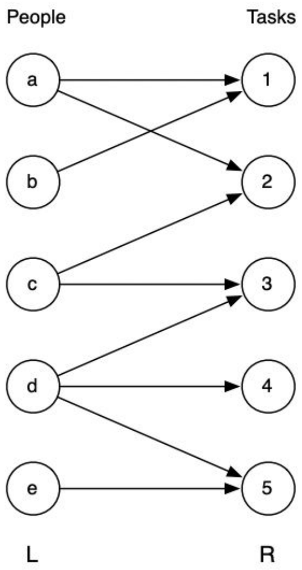
# Reduction

- Given an instance of bipartite matching
- Create an instance of network flow
- The solution to the network flow problem can easily be used to find the solution to the bipartite matching



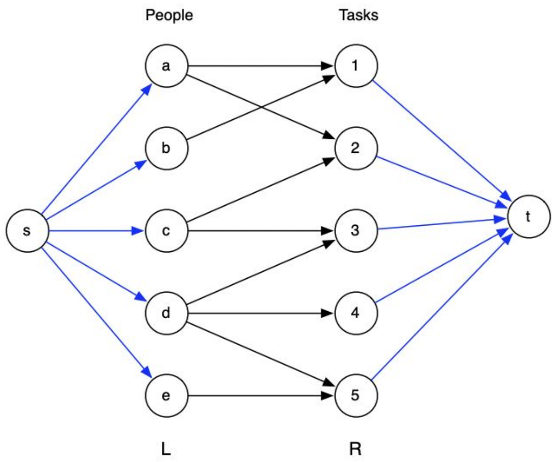
# Using Flow Network to Solve Bipartite Matching

1. Given bipartite graph G = (A ∪ B, E), direct the edges from A to B



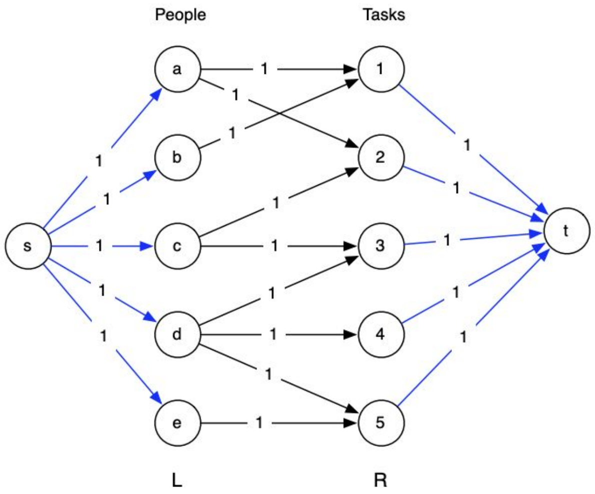
# Using Flow Network to Solve Bipartite Matching

1. Given bipartite graph G = (A ∪ B, E), direct the edges from A to B
2. Add new vertices s and t
3. Add an edge from s to every vertex in A
4. Add an edge from every vertex in B to t



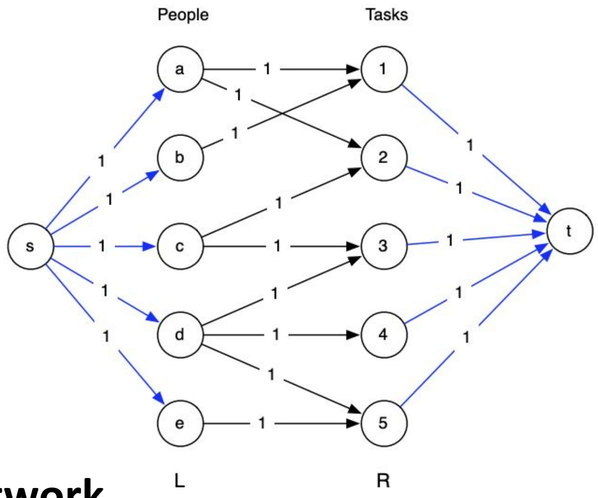
# Using Flow Network to Solve Bipartite Matching

1. Given bipartite graph G = (A ∪ B, E), direct the edges from A to B
2. Add new vertices s and t
3. Add an edge from s to every vertex in A
4. Add an edge from every vertex in B to t
5. Make all the capacities 1
6. Solve maximum network flow problem on this new graph G



# Using Flow Network to Solve Bipartite Matching

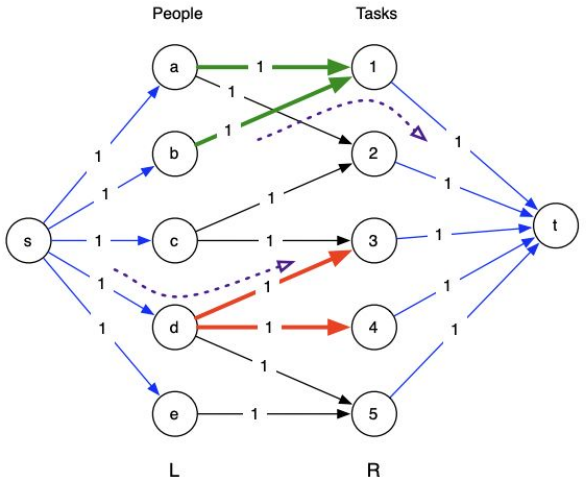
1. Given bipartite graph G = (A ∪ B, E), direct the edges from A to B
2. Add new vertices s and t
3. Add an edge from s to every vertex in A
4. Add an edge from every vertex in B to t
5. Make all the capacities 1
6. Solve maximum network flow problem on this new graph G




**The edges used in the maximum network flow will correspond to the largest possible matching!**

# Using Flow Network to Solve Bipartite Matching

- We can choose at most one edge leaving any node in A
- We can choose at most one edge entering any node in B




If we chose more than 1, we will **violate the conservation rule** in flow network