CSE 421 Winter 2025
Lecture 15: Bellman-Ford
Max Flow

Nathan Brunelle

http://www.cs.uw.edu/421

Single-source shortest paths, with negative edge weights

Given: an (un)directed graph $\boldsymbol{G = (V, E)}$ with each edge $\boldsymbol{e}$ having a weight $\boldsymbol{w(e)}$ and a vertex $\boldsymbol{s}$

Find: (length of) shortest paths from $\boldsymbol{s}$ to each vertex in $\boldsymbol{G}$, **or else indicate that there is a negative-cost cycle**

Called the Bellman-Ford algorithm

(The original DP algorithm!)

(Also, the original shortest path algorithm!)

Bellman Ford– Four Steps

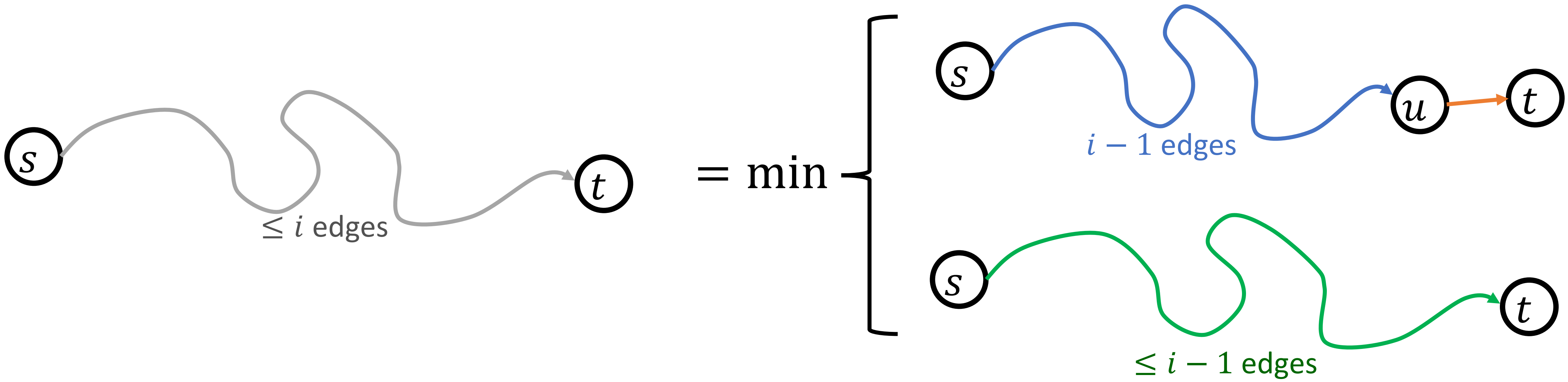
1. Formulate the answer with a recursive structure
   - What are the options for the last choice?
   - For each such option, what does the subproblem look like? How do we use it?
2. Choose a memory structure.
   - Figure out the possible values of all parameters in the recursive calls.
   - How many subproblems (options for last choice) are there?
   - What are the parameters needed to identify each?
   - How many different values could there be per parameter?
3. Specify an order of evaluation.
   - Want to guarantee that the necessary subproblem solutions are in memory when you need them.
   - With this step: a “Bottom-up” (iterative) algorithm
   - Without this step: a “Top-down” (recursive) algorithm
4. See if there’s a way to save space
   - Is it possible to reuse some memory locations?

Final Recursive Structure

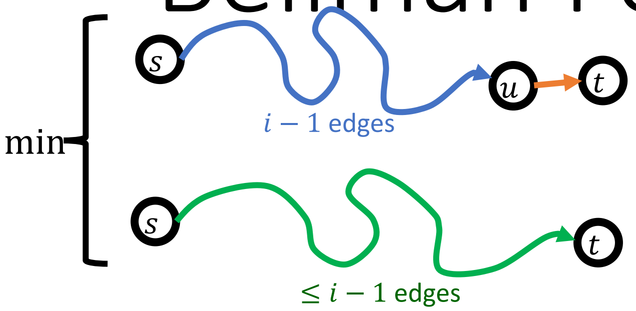
$OPT(i,t) =$ the weight of the shortest path from s to t with at most i edges

$$OPT(i,t) = \begin{cases} 0 \text{ if } i = 0 \text{ and } s = t \\ \infty \text{ if } i = 0 \text{ and } s \neq t \\ \min \begin{cases} \min_{u \in V} \{ OPT(i-1,u) + w(u,t) \} \\ OPT(i-1,t) \end{cases} \end{cases}$$

Where $w(u,t)$ is the weight of the edge from u to t if it exists and ∞ if not.



Bellman Ford– Four Steps



1. Formulate the answer with a recursive structure
   - What are the options for the last choice?
   - For each such option, what does the subproblem look like? How do we use it?
2. Choose a memory structure.
   - Figure out the possible values of all parameters in the recursive calls.
   - How many subproblems (options for last choice) are there?
   - What are the parameters needed to identify each?
   - How many different values could there be per parameter?
3. Specify an order of evaluation.
   - Want to guarantee that the necessary subproblem solutions are in memory when you need them.
   - With this step: a “Bottom-up” (iterative) algorithm
   - Without this step: a “Top-down” (recursive) algorithm
4. See if there’s a way to save space
   - Is it possible to reuse some memory locations?

Identifying the Memory Structure

$$OPT(i,t) = \begin{cases} 0 \text{ if } i = 0 \text{ and } s = t \\ \infty \text{ if } i = 0 \text{ and } s \neq t \\ \min \begin{cases} \min_{u \in V}\{{\color{blue}OPT(i-1,u)} + {\color{orange}w(u,t)}\} \\ {\color{green}OPT(i-1,t)} \end{cases} \end{cases}$$

- How many parameters?
  - 2
- What does each represent?
  - i: the length of the path
  - t: a node
- How many different values?
  - i: $|V|$ (from length 0 up to $|V| - 1$ if the path is simple)
  - t: $|V|$ (one value per node)

	1	2	3	4	5	6	7
0							
1							
2							
3							
4							
5							
6							

Top-Down Bellman-Ford

This algorithm correctly finds shortest paths when there are no negative-cost cycles
How can we check for negative cost cycles?

BF(i, t):

```
if OPT[i][t] not blank:    // Check if we've solved this already
        return OPT[i][j]
if i == 0:    // Check if this is a base case
        solution = 0 ? t == s : ∞
        OPT[i][t] = solution    // Always save your solution before returning
        return solution
solution = ∞
for each u ∈ V:
        solution = min(solution, BF(i − 1, u) +w(u, t)) // solve each subproblem, pick which to use
solution = min(solution, BF(i − 1,t)) // solve each subproblem, pick which to use
OPT[i][t] = solution    // Always save your solution before returning
return solution
```

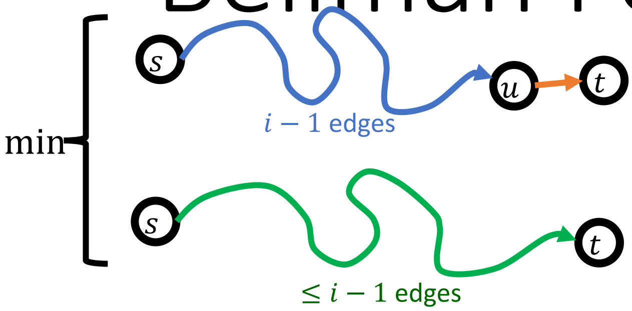
Checking for Negative Cycles

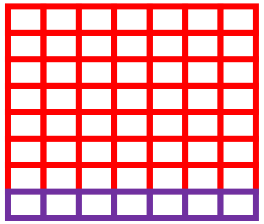
$$OPT(i,t) = \begin{cases} 0 \text{ if } i = 0 \text{ and } s = t \\ \infty \text{ if } i = 0 \text{ and } s \neq t \\ \min \begin{cases} \min_{u \in V}\{OPT(i-1,u) + w(u,t)\} \\ OPT(i-1,t) \end{cases} \end{cases}$$

- How many parameters?
  - 2
- What does each represent?
  - i: the length of the path
  - t: a node
- How many different values?
  - $\boldsymbol{i}$: $|\boldsymbol{V}|$**+1**
    - **a path of $|V|$ edges is not simple, so if any $|V|$-edge path is shorter than one with fewer edges, there must be a negative cycle!**
  - t: $|V|$ (one value per node)

	1	2	3	4	5	6	7
0							
1							
2							
3							
4							
5							
6							
7							

Bellman Ford– Four Steps





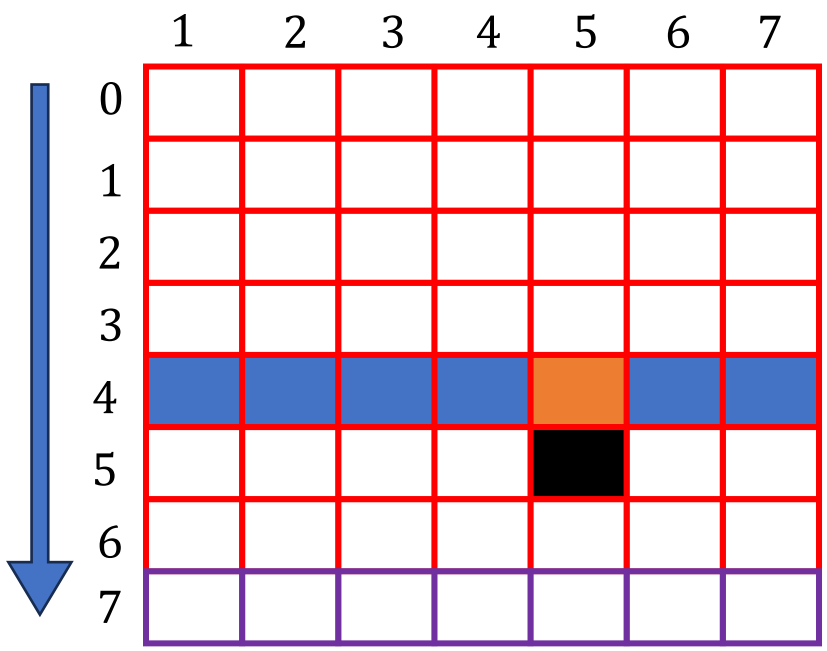
1. Formulate the answer with a recursive structure
   - What are the options for the last choice?
   - For each such option, what does the subproblem look like? How do we use it?
2. Choose a memory structure.
   - Figure out the possible values of all parameters in the recursive calls.
   - How many subproblems (options for last choice) are there?
   - What are the parameters needed to identify each?
   - How many different values could there be per parameter?
3. Specify an order of evaluation.
   - Want to guarantee that the necessary subproblem solutions are in memory when you need them.
   - With this step: a “Bottom-up” (iterative) algorithm
   - Without this step: a “Top-down” (recursive) algorithm
4. See if there’s a way to save space
   - Is it possible to reuse some memory locations?

Order of Evaluations

$$OPT(i,t) = \begin{cases} 0 \text{ if } i = 0 \text{ and } s = t \\ \infty \text{ if } i = 0 \text{ and } s \neq t \\ \min \begin{cases} \min_{u \in V} \{ OPT(i-1,u) + w(u,t) \} \\ OPT(i-1,t) \end{cases} \end{cases}$$

Each cell depends on every value in the previous row

Solve in order of i



Bottom-Up Bellman-Ford

```
BF(s, t):
    OPT[0][s] = 0   // Solve and save base cases
    for u ∈ V \ {s}:
        OPT[0][u] = ∞   // Solve and save base cases
    for i = 0 up to |V|:
        for u ∈ V:
            for v ∈ neighbors(u):
                OPT[i][u] = min(OPT[i][u], OPT[i − 1][ v]) // solve and pick
            OPT[i][u] = min(OPT[i][u], OPT[i − 1][u]) // solve and pick
    for u ∈ V:
        if OPT[|V|][u] < OPT[|V| − 1][u]: // check for negative cycles
            return "negative cycle"
    return OPT[s][t] // return the final answer
```

Bottom-Up Bellman-Ford

BF(s, t):

OPT[0][s] = 0 // Solve and save base cases $\Theta(1)$

for $u \in V \setminus \{s\}$:

OPT[0][u] = ∞ // Solve and save base cases $\Theta(|V|)$

for $i = 0$ up to $|V|$:

for $u \in V$:

for $v \in \text{neighbors}(u)$: $\Theta(|V||E|)$

$OPT[i][u] = \min(OPT[i][u], OPT[i-1][v])$ // solve and pick

$OPT[i][u] = \min(OPT[i][u], OPT[i-1][u])$ // solve and pick

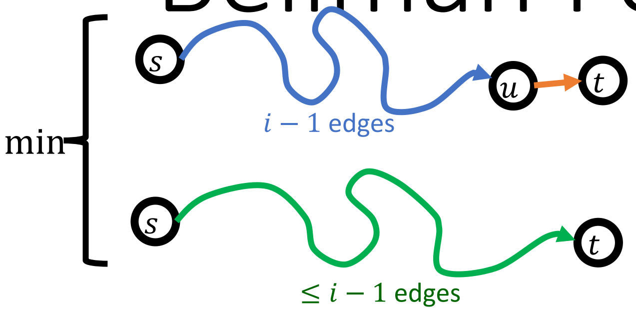
for $u \in V$:

if $OPT[|V|][u] < OPT[|V|-1][u]$: // check for negative cycles $\Theta(|V|)$

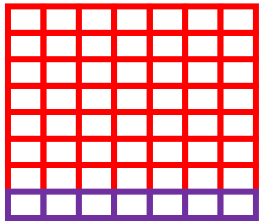
return "negative cycle"

return OPT[s][t] // return the final answer $\Theta(1)$

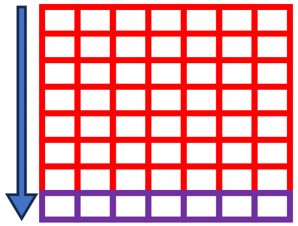
Bellman Ford– Four Steps



1. Formulate the answer with a recursive structure
   - What are the options for the last choice?
   - For each such option, what does the subproblem look like? How do we use it?



2. Choose a memory structure.
   - Figure out the possible values of all parameters in the recursive calls.
   - How many subproblems (options for last choice) are there?
   - What are the parameters needed to identify each?
   - How many different values could there be per parameter?



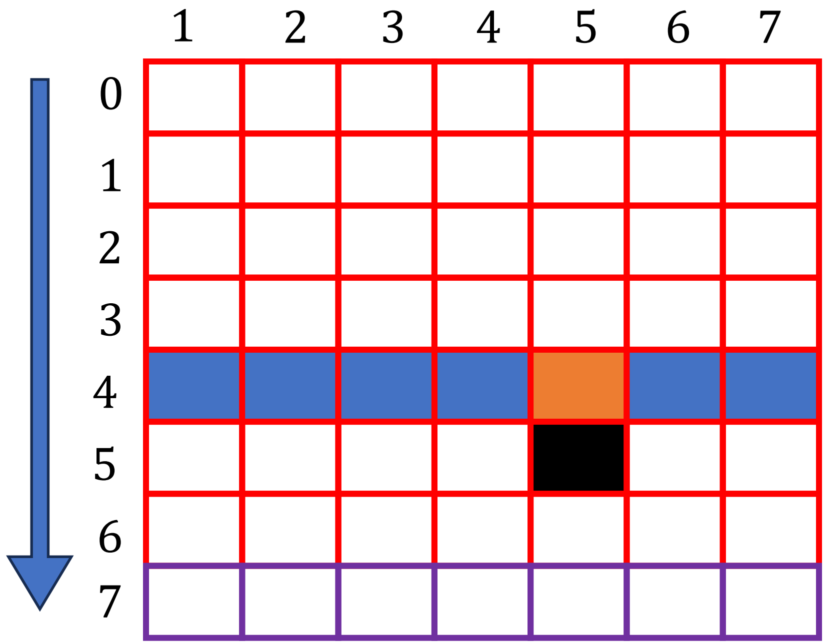
3. Specify an order of evaluation.
   - Want to guarantee that the necessary subproblem solutions are in memory when you need them.
   - With this step: a “Bottom-up” (iterative) algorithm
   - Without this step: a “Top-down” (recursive) algorithm
4. See if there’s a way to save space
   - Is it possible to reuse some memory locations?

Order of Evaluations

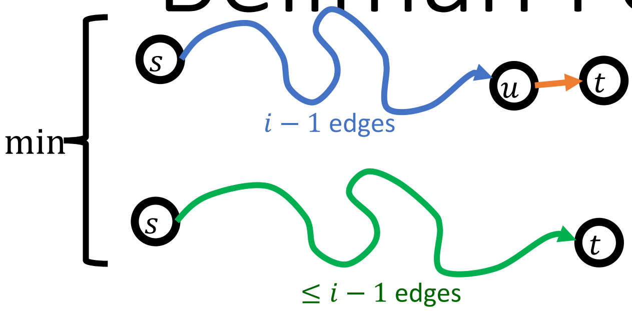
$$OPT(i,t) = \begin{cases} 0 \text{ if } i = 0 \text{ and } s = t \\ \infty \text{ if } i = 0 \text{ and } s \neq t \\ \min \begin{cases} \min_{u \in V}\{OPT(i-1,u) + w(u,t)\} \\ OPT(i-1,t) \end{cases} \end{cases}$$

Each cell depends *only* on values in the previous row

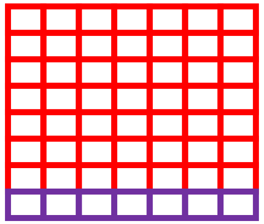
We only need two rows!



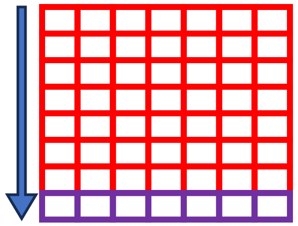
Bellman Ford– Four Steps



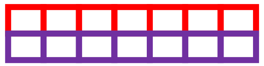
1. Formulate the answer with a recursive structure
   - What are the options for the last choice?
   - For each such option, what does the subproblem look like? How do we use it?



2. Choose a memory structure.
   - Figure out the possible values of all parameters in the recursive calls.
   - How many subproblems (options for last choice) are there?
   - What are the parameters needed to identify each?
   - How many different values could there be per parameter?



3. Specify an order of evaluation.
   - Want to guarantee that the necessary subproblem solutions are in memory when you need them.
   - With this step: a “Bottom-up” (iterative) algorithm
   - Without this step: a “Top-down” (recursive) algorithm

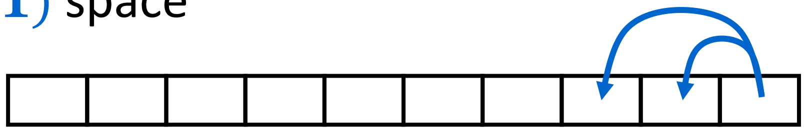


4. See if there’s a way to save space
   - Is it possible to reuse some memory locations?

Dynamic Programming Patterns

Fibonacci pattern:

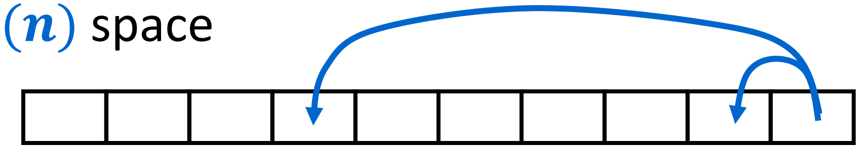
- 1-D, $O(\mathbf{1})$ immediately prior
- $O(\mathbf{1})$ space



$O(\boldsymbol{n})$

Weighted interval scheduling pattern:

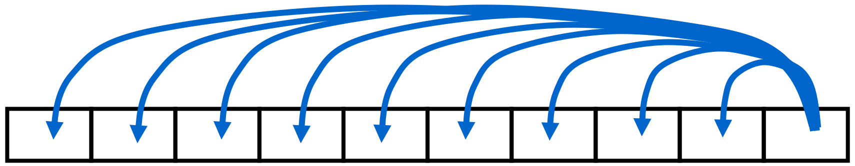
- 1-D, $O(\mathbf{1})$ arbitrary prior
- $O(\boldsymbol{n})$ space



$O(\boldsymbol{n})$

Longest increasing subsequence pattern:

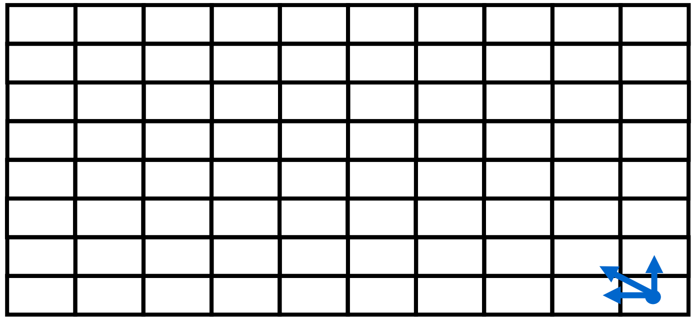
- 1-D, all $\boldsymbol{n}-\mathbf{1}$ prior
- $O(\boldsymbol{n})$ space



$O(\boldsymbol{n}^{\mathbf{2}})$

Alignment pattern:

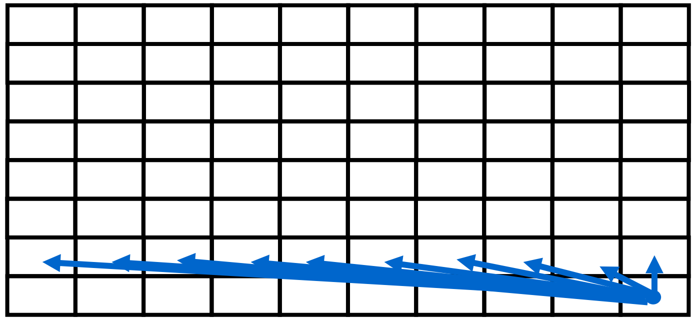
- 2-D, O(**1**) in previous row, above, left, diagonal
- $O(\boldsymbol{n}\cdot\boldsymbol{m})$ space



$O(\boldsymbol{nm})$

Bellman Ford pattern:

- 2-D, O(**|V|**) in previous row,
- $O(|V|)$ space



$O(|V||E|)$

Origins of Max Flow and Min Cut Problems

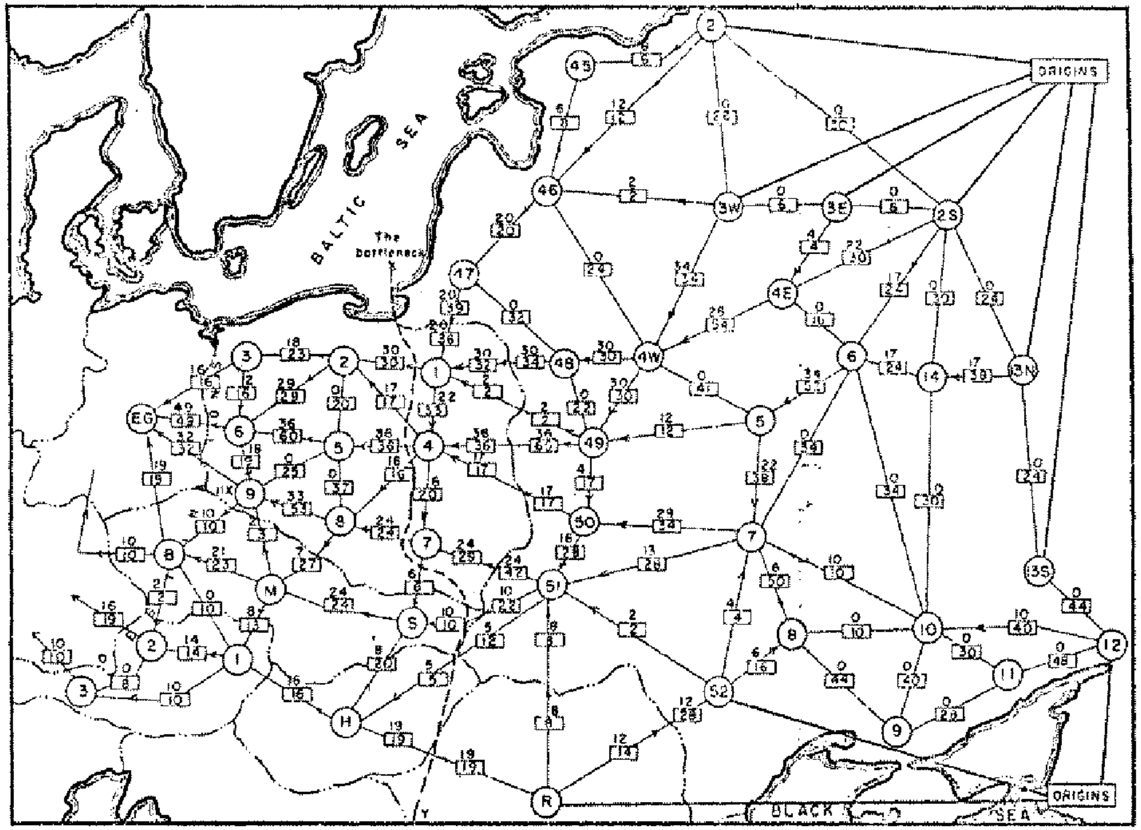
Max Flow problem formulation:

- [Tolstoy 1930] Rail transportation planning for the Soviet Union

Min Cut problem formulation:

- Cold War: US military planners want to find a way to cripple Soviet supply routes
- [Harris 1954] Secret RAND corp report for US Air Force

[Ford-Fulkerson 1955] Problems are equivalent

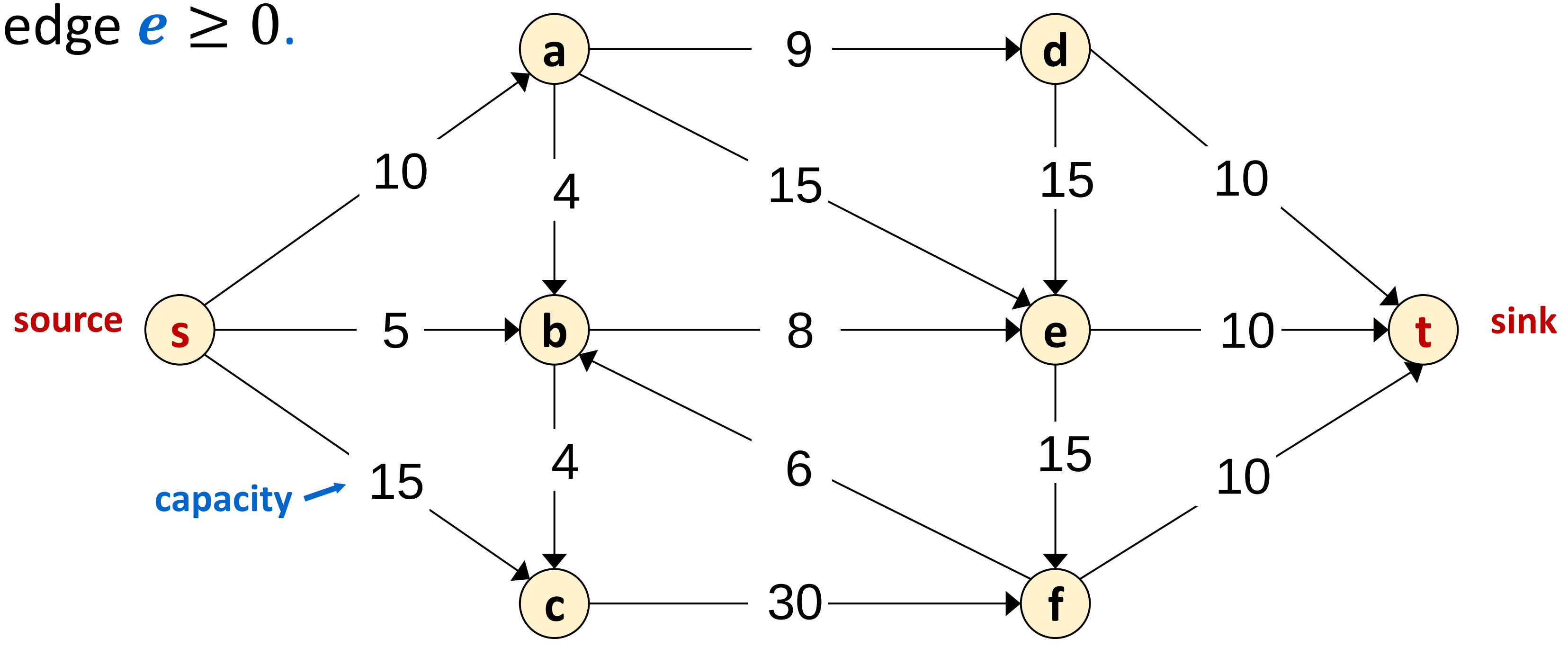


Reference: *On the history of the transportation and maximum flow problems*. Alexander Schrijver in Math Programming, 91: 3, 2002.

Flow Network

Flow network:

- Abstraction for material *flowing* through the edges.
- $G = (V, E)$ directed graph, no parallel edges.
- Two distinguished nodes: s = source, t = sink.
- $c(e)$ = capacity of edge $e \geq 0$.



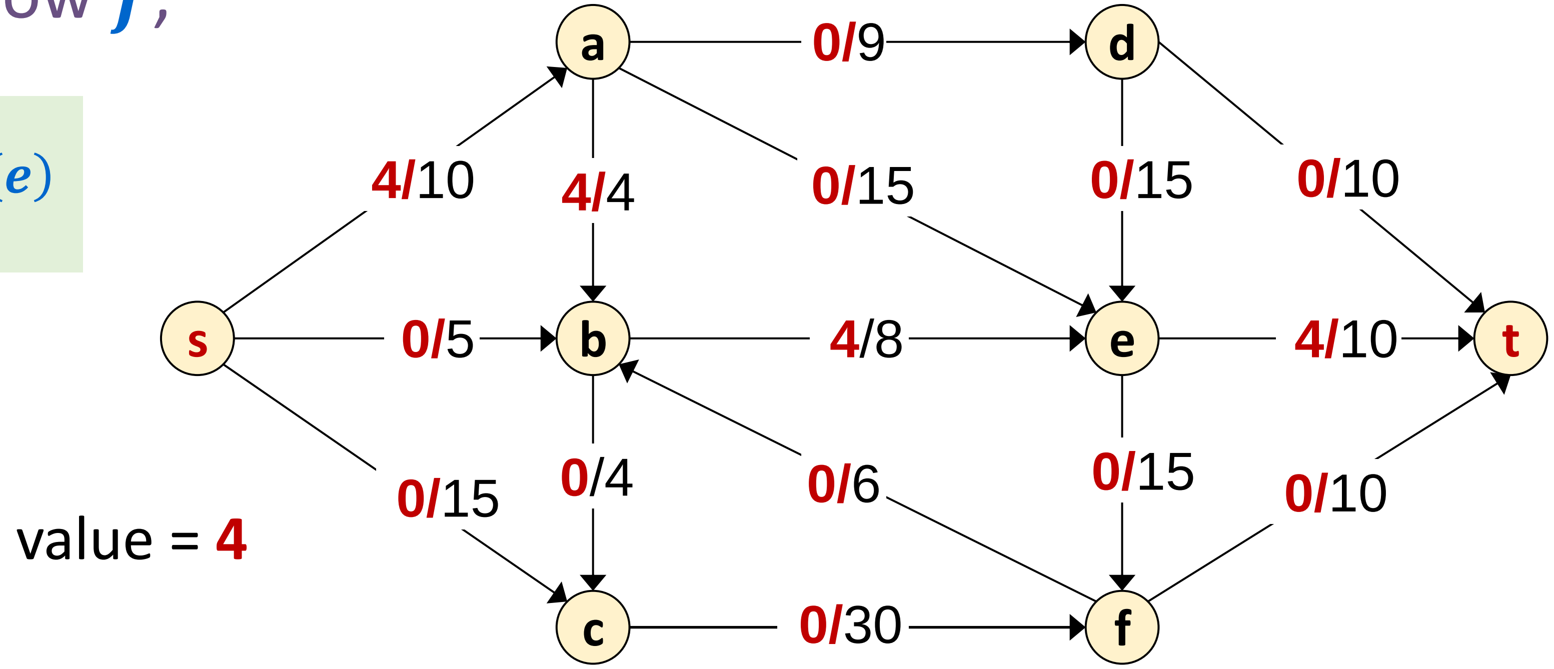
Flows

Defn: An s-t flow in a flow network is a function $f\colon E \to \mathbb{R}$ that satisfies:

- For each $e \in E$: $0 \le f(e) \le c(e)$ [capacity constraints]
- For each $v \in V - \{s, t\}$: $\displaystyle\sum_{e \text{ into } v} f(e) = \sum_{e \text{ out of } v} f(e)$ [flow conservation]

Defn: The value of flow f,

$$v(f) = \sum_{e \text{ out of } s} f(e)$$



Flows

Defn: An s-t flow in a flow network is a function $f\colon E \to \mathbb{R}$ that satisfies:

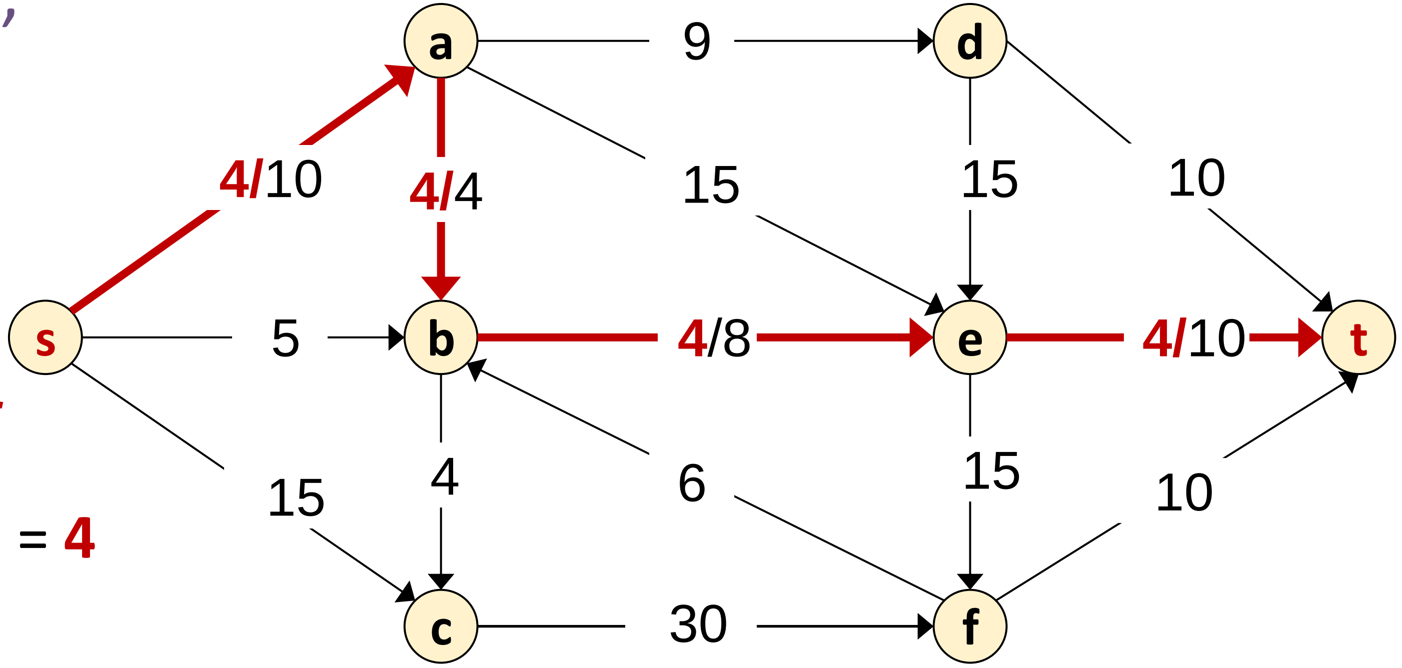
- For each $e \in E$: $0 \leq f(e) \leq c(e)$ [capacity constraints]
- For each $v \in V - \{s,t\}$: $\displaystyle\sum_{e \text{ into } v} f(e) = \sum_{e \text{ out of } v} f(e)$ [flow conservation]

Defn: The value of flow f,

$$v(f) = \sum_{e \text{ out of } s} f(e)$$

Only show non-zero values of f

value = 4



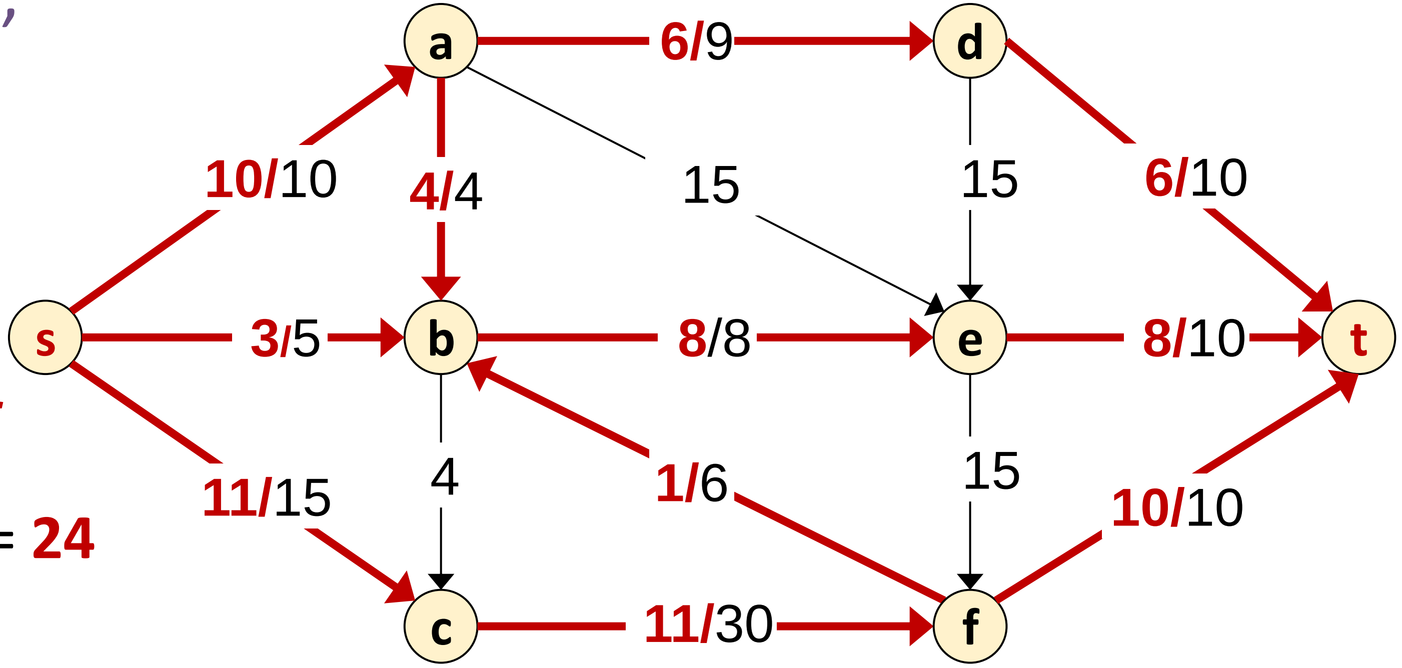
Flows

Defn: An s-t flow in a flow network is a function $f\colon E \to \mathbb{R}$ that satisfies:

- For each $e \in E$: $0 \le f(e) \le c(e)$ [capacity constraints]
- For each $v \in V - \{s, t\}$: $\displaystyle\sum_{e \text{ into } v} f(e) = \sum_{e \text{ out of } v} f(e)$ [flow conservation]

Defn: The value of flow f,

$$v(f) = \sum_{e \text{ out of } s} f(e)$$



Only show non-zero values of f

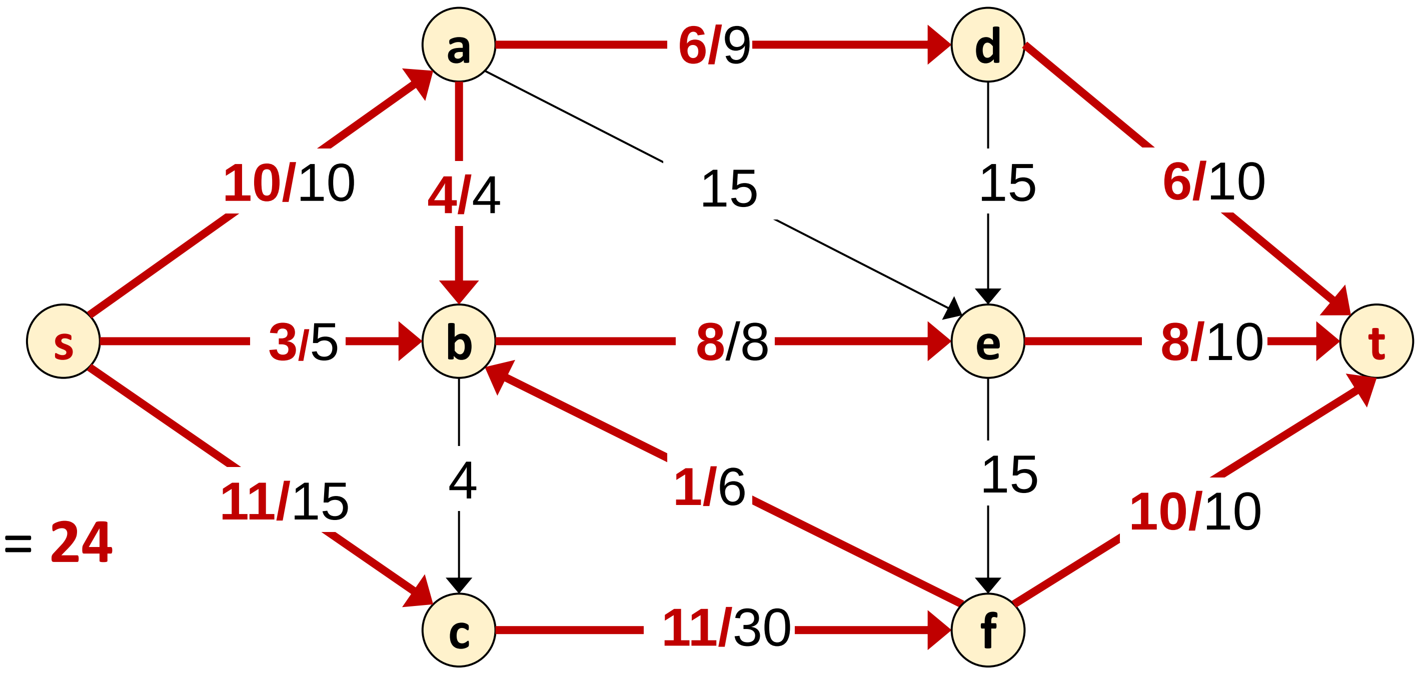
value = **24**

Maximum Flow Problem

Given: a flow network

Find: an s-t flow of maximum value

value = **24**

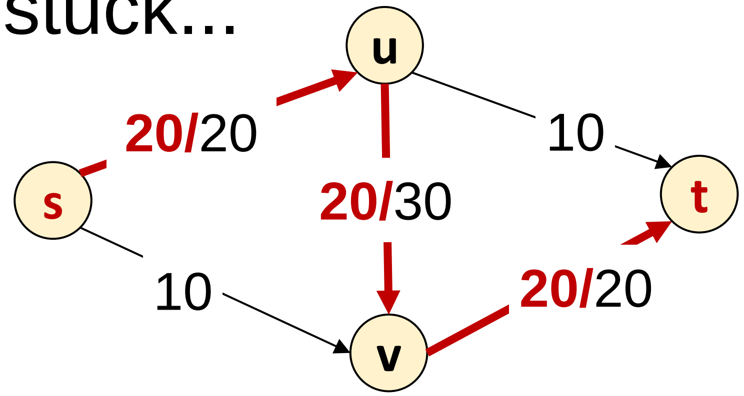


Towards a Max Flow Algorithm

What about the following greedy algorithm?

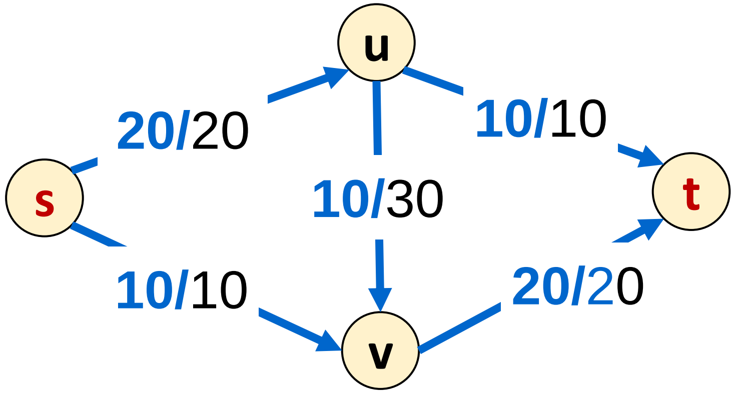
- Start with $f(e) = 0$ for all edges $e \in E$.
- While there is an s-t path P where each edge has $f(e) < c(e)$.
  - "Augment" flow along P; that is:
    - Let $\alpha = \min_{e \in P}(c(e) - f(e))$
    - Add α to flow on every edge e along path P. (Adds α to $v(f)$.)

Can get stuck...



Has flow value **20** and no path P

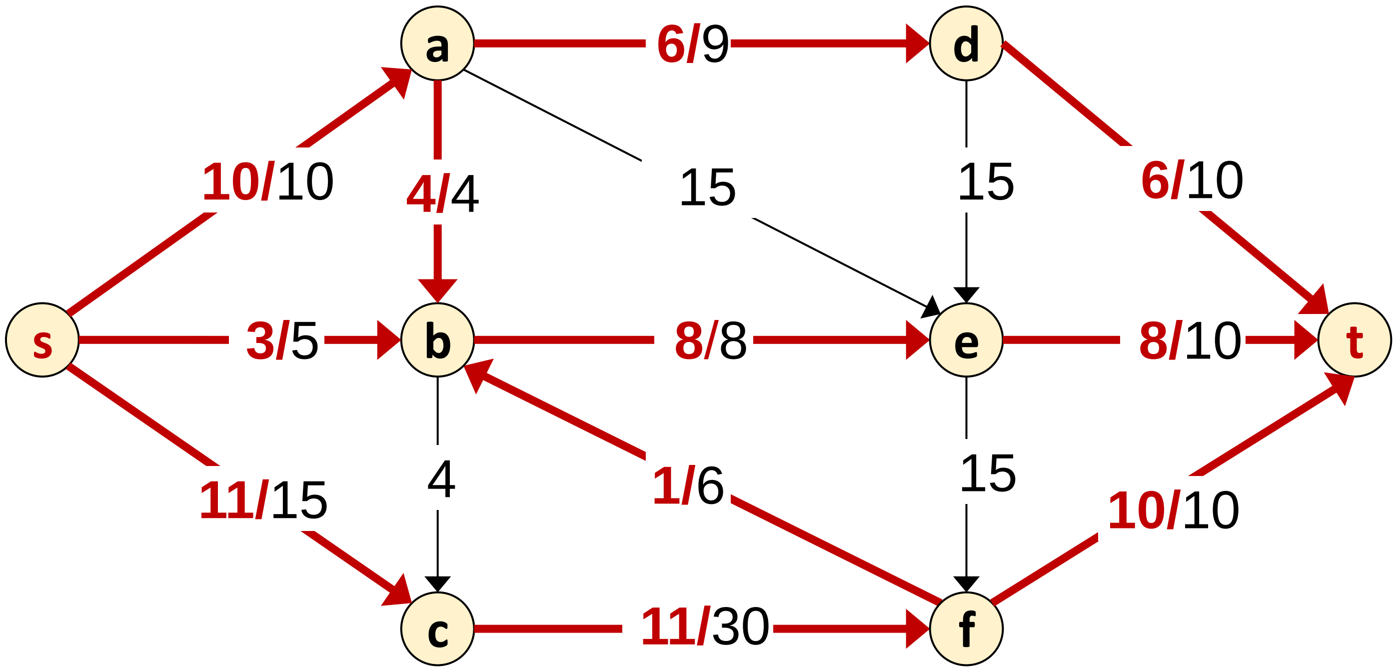
but **30** is possible



Another "Stuck" Example

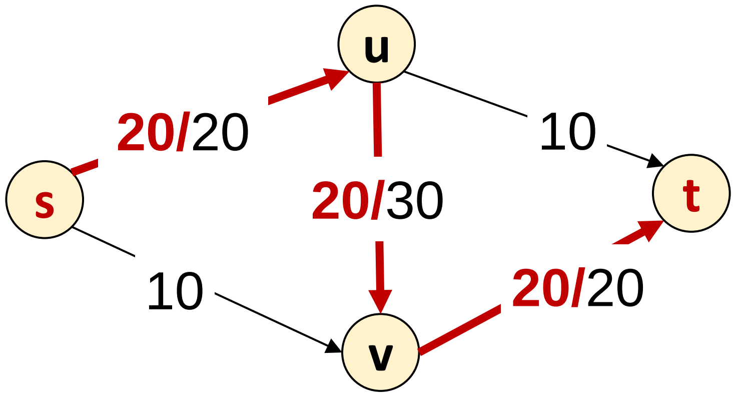
On every $\boldsymbol{s}$-$\boldsymbol{t}$ path there is some edge with $\boldsymbol{f(e) = c(e)}$:

Value of flow = **24**



Next idea: Ford-Fulkerson Algorithm, which applies greedy ideas to a "residual graph" that lets us reverse prior flow decisions from the basic greedy approach to get optimal results!

Greed Revisited: Residual Graph & Augmenting Paths

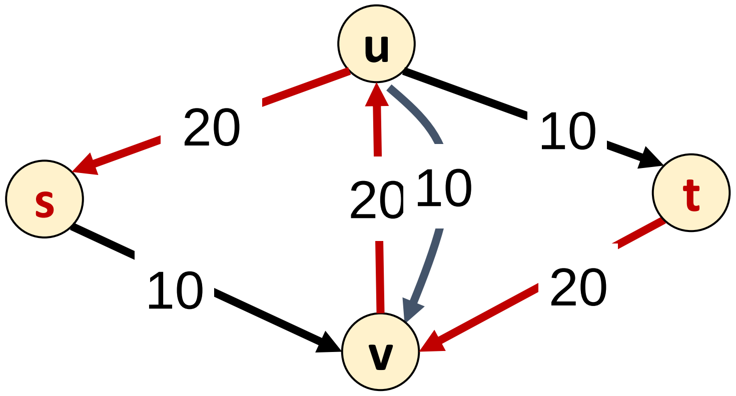


The only way we could route more flow from **s** to **t** would be to reduce the flow from **u** to **v** to make room for that amount of extra flow from **s** to **v**.
But to conserve flow we also would need to increase the flow from **u** to **t** by that same amount.

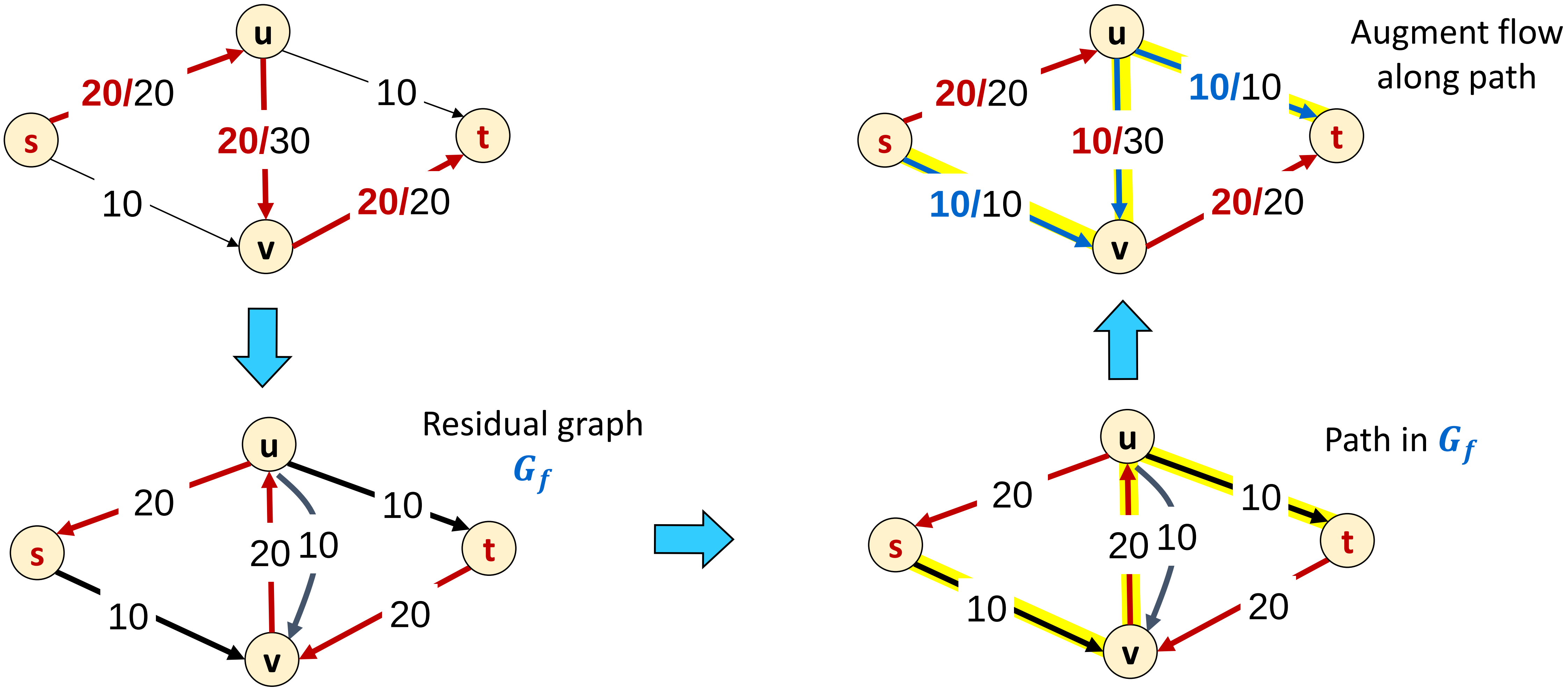
Suppose that we took this flow f as a baseline, what changes could each edge handle?

- We could add up to 10 units along **sv** or **ut** or **uv**
- We could reduce by up to 20 units from **su** or **uv** or **vt**

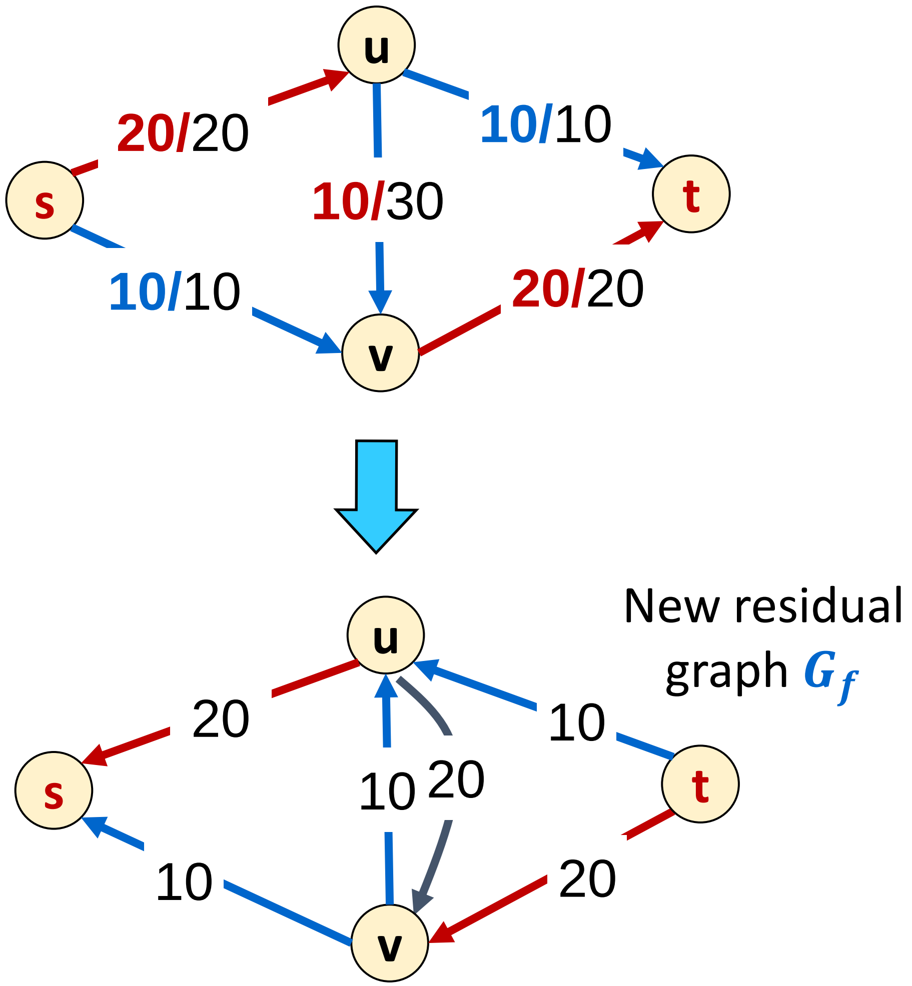
This gives us a residual graph G_f of possible changes where we draw reducing as "sending back".



Greed Revisited: Residual Graph & Augmenting Paths



Greed Revisited: Residual Graph & Augmenting Paths



No path can even leave s!

Residual Graphs

An alternative way to represent a flow network

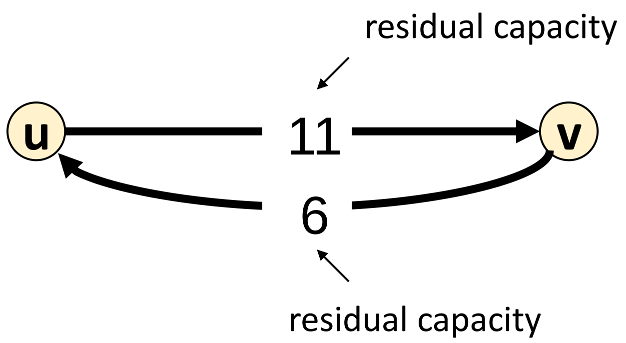
- Represents the net available flow between two nodes

Original edge: $\boldsymbol{e = (u, v) \in E}$.

- Flow $\boldsymbol{f(e)}$, capacity $\boldsymbol{c(e)}$.

Residual edges of two kinds:

- Forward: $\boldsymbol{e = (u, v)}$ with capacity $\boldsymbol{c_f(e) = c(e) - f(e)}$
  - Amount of extra flow we can add along $\boldsymbol{e}$
- Backward: $\boldsymbol{e^{\mathrm{R}} = (v, u)}$ with capacity $\boldsymbol{c_f(e) = f(e)}$
  - Amount we can reduce/undo flow along $\boldsymbol{e}$



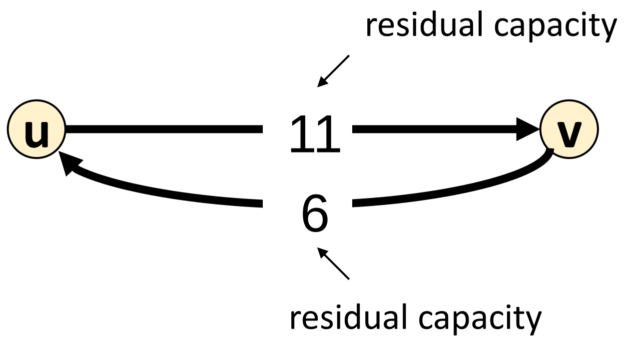
Residual graph: $\boldsymbol{G_f = (V, E_f)}$.

- Residual edges with residual capacity $\boldsymbol{c_f(e) > 0}$.
- $\boldsymbol{E_f = \{e : f(e) < c(e)\} \cup \{e^{\mathrm{R}} : f(e) > 0\}}$.

Residual Graphs and Augmenting Paths

Residual edges of two kinds:

- Forward: $\boldsymbol{e = (u, v)}$ with capacity $\boldsymbol{c_f(e) = c(e) - f(e)}$
  - Amount of extra flow we can add along $\boldsymbol{e}$
- Backward: $\boldsymbol{e^{\mathrm{R}} = (v, u)}$ with capacity $\boldsymbol{c_f(e) = f(e)}$
  - Amount we can reduce/undo flow along $\boldsymbol{e}$



Residual graph: $\boldsymbol{G_f = (V, E_f)}$.

- Residual edges with residual capacity $\boldsymbol{c_f(e) > 0}$.
- $\boldsymbol{E_f = \{e : f(e) < c(e)\} \cup \{e^{\mathrm{R}} : f(e) > 0\}}$.

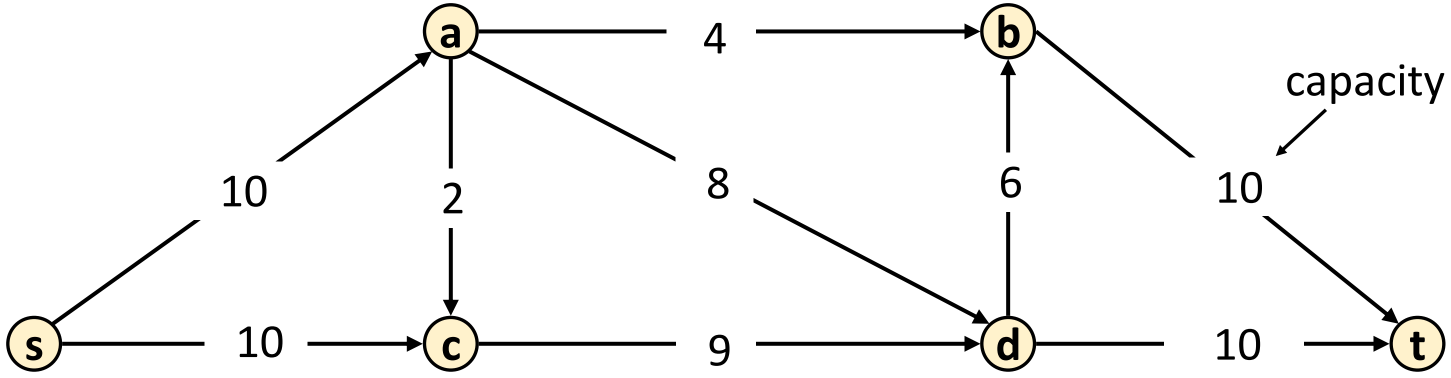
Augmenting Path: Any $\boldsymbol{s}$-$\boldsymbol{t}$ path $\boldsymbol{P}$ in $\boldsymbol{G_f}$. Let **bottleneck**($\boldsymbol{P}$)= $\min_{e \in P} \boldsymbol{c_f(e)}$.

Ford-Fulkerson idea: Repeat "find an augmenting path $\boldsymbol{P}$ and increase flow by **bottleneck**($\boldsymbol{P}$)" until none left.

38

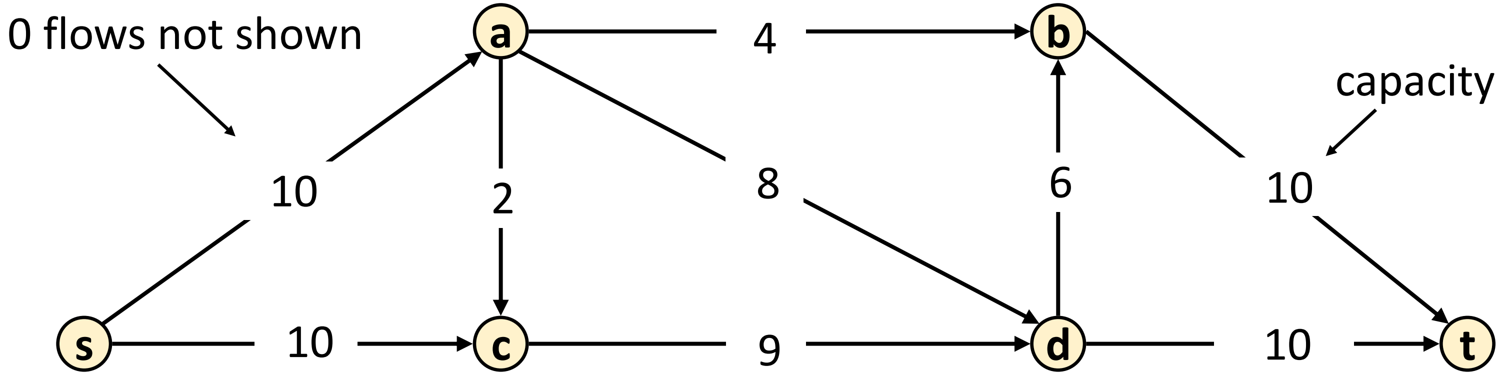
Ford-Fulkerson Algorithm

$\boldsymbol{G}$:



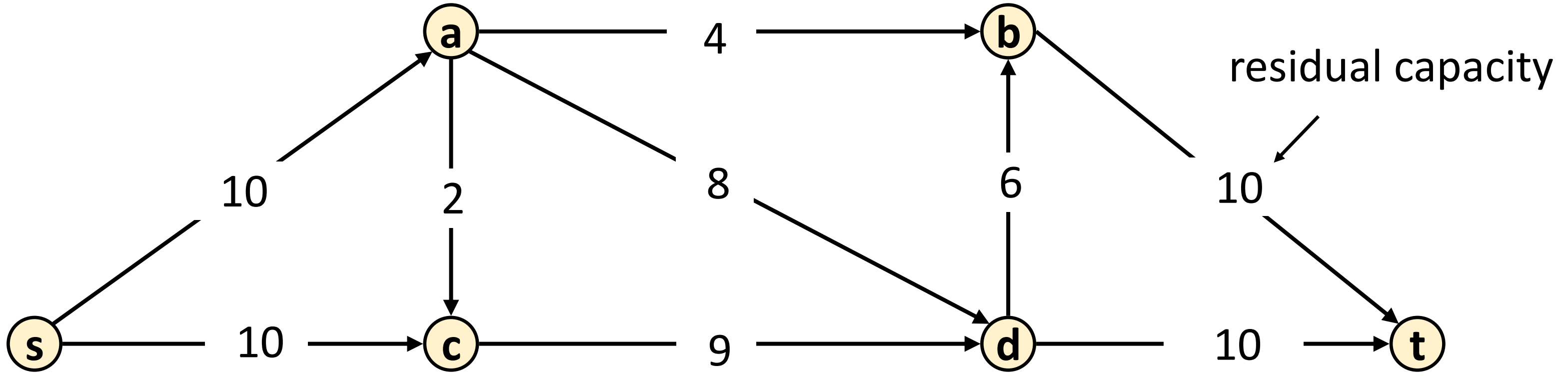
Ford-Fulkerson Algorithm

G:

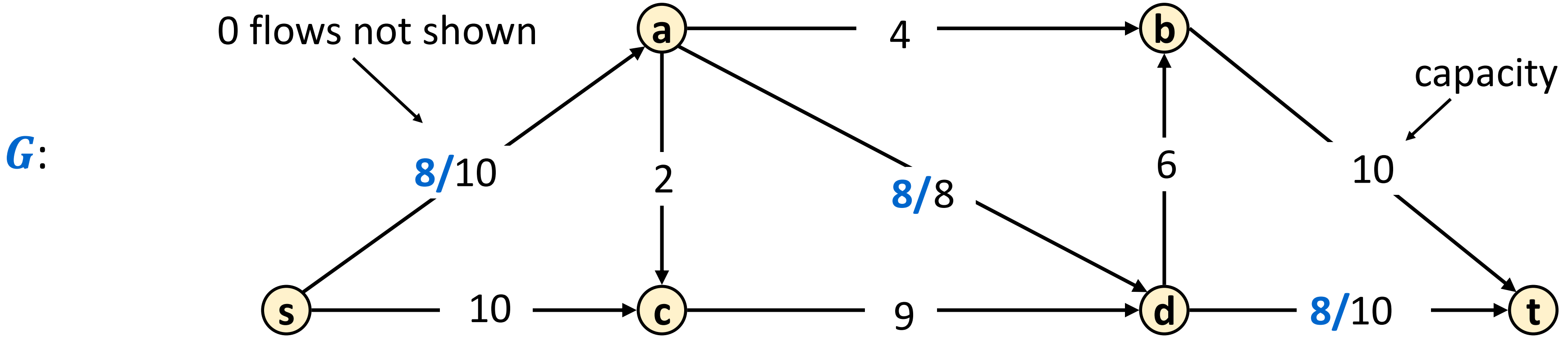


Flow value = **0**

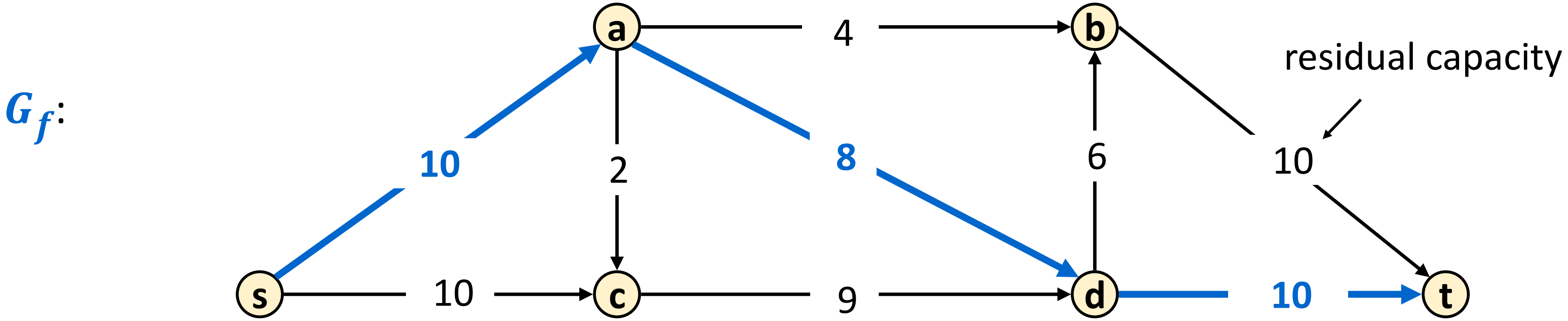
G_f:



Ford-Fulkerson Algorithm

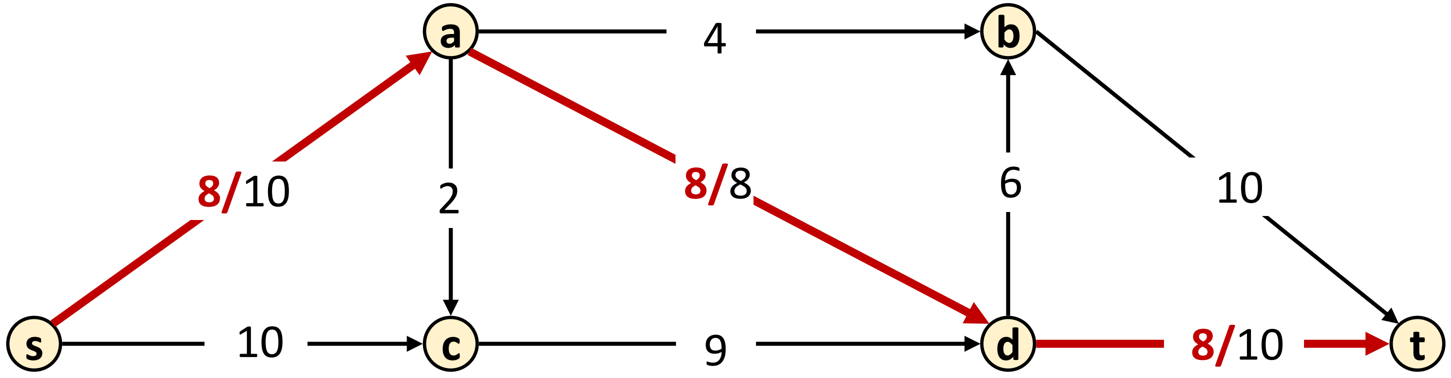


Flow value = **0**
+**8**=**8**



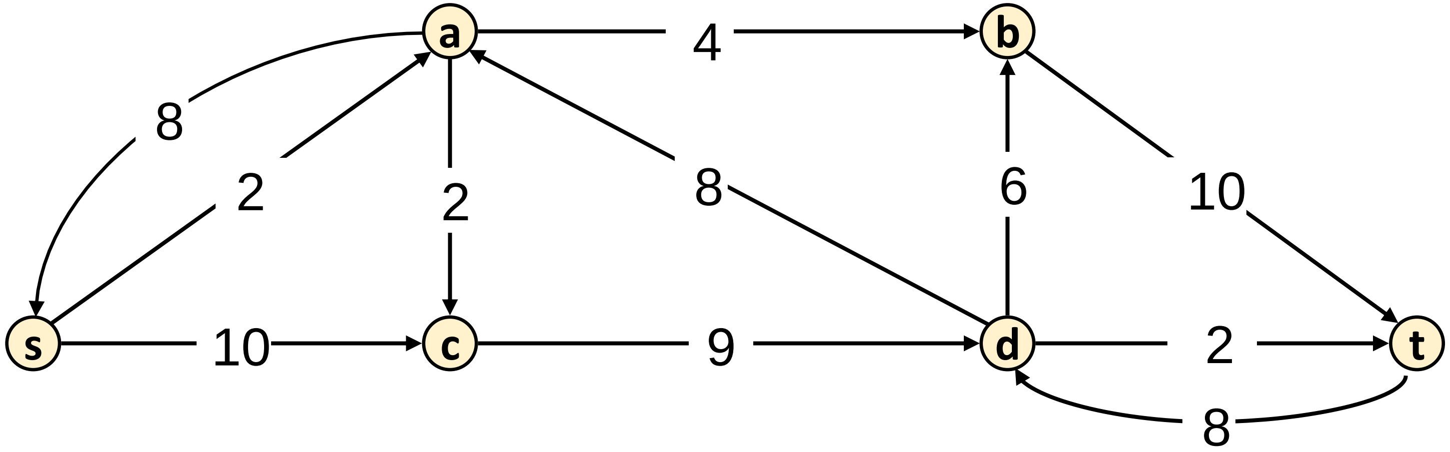
Ford-Fulkerson Algorithm

G:



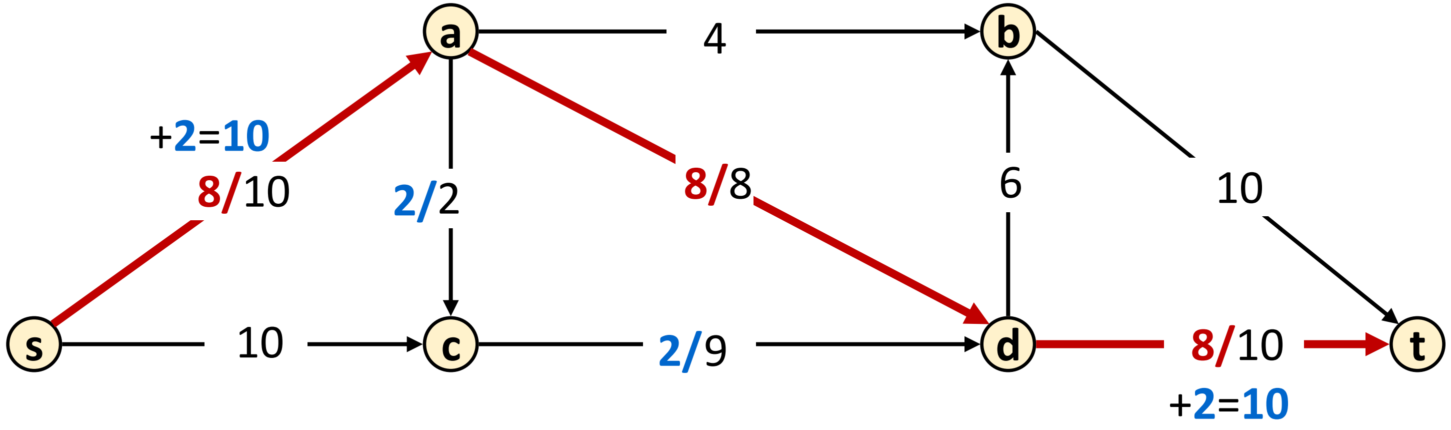
Flow value = **8**

G_f:



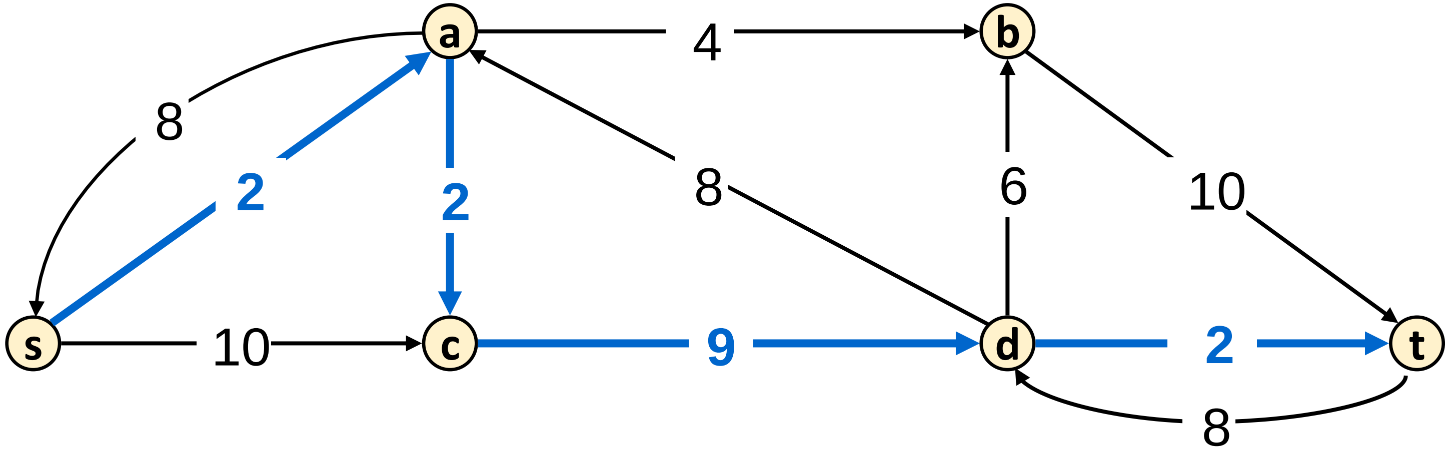
Ford-Fulkerson Algorithm

G:



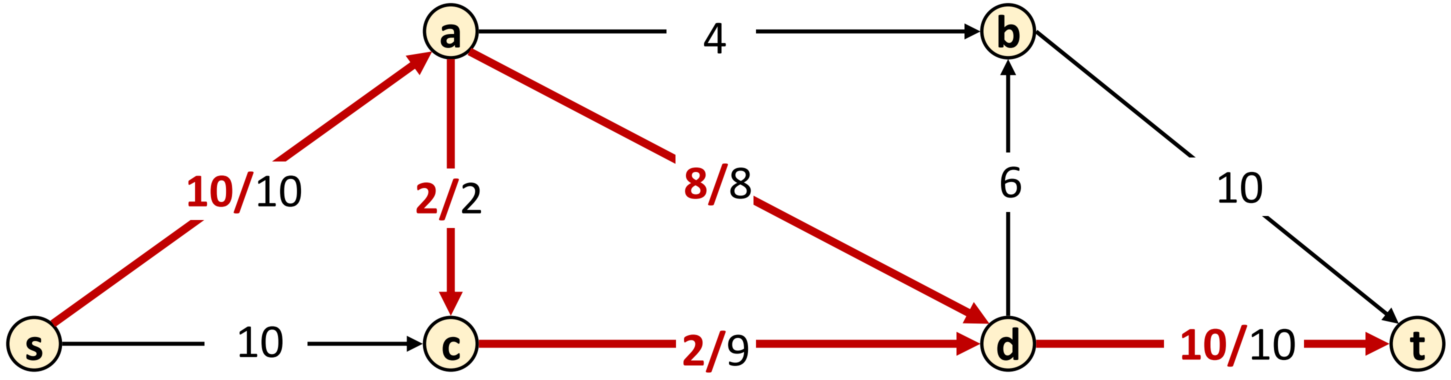
Flow value = 8
+2=10

G_f:



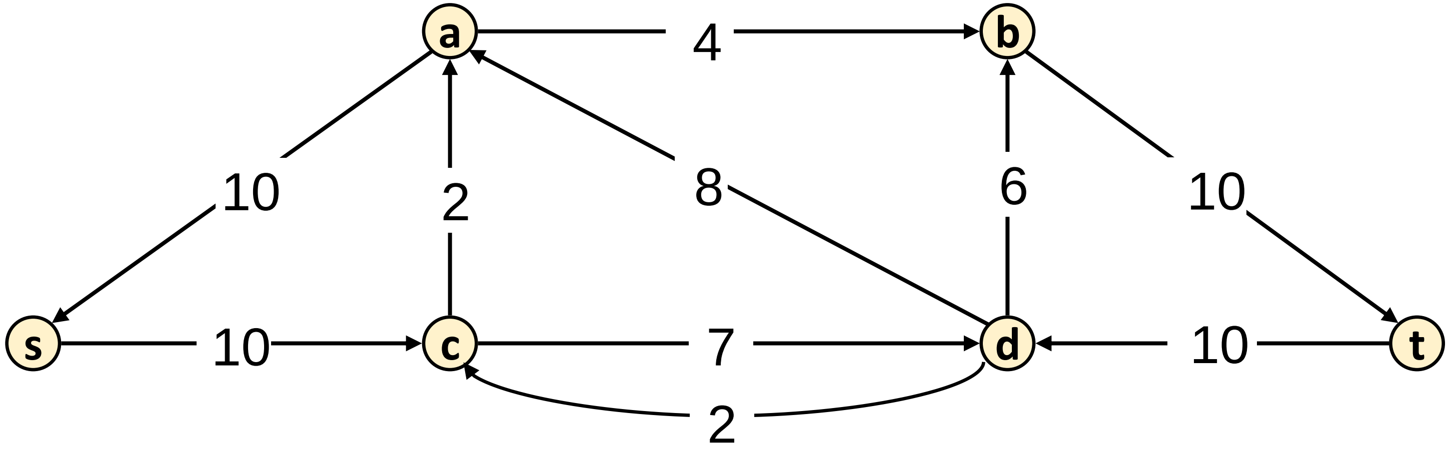
Ford-Fulkerson Algorithm

G:

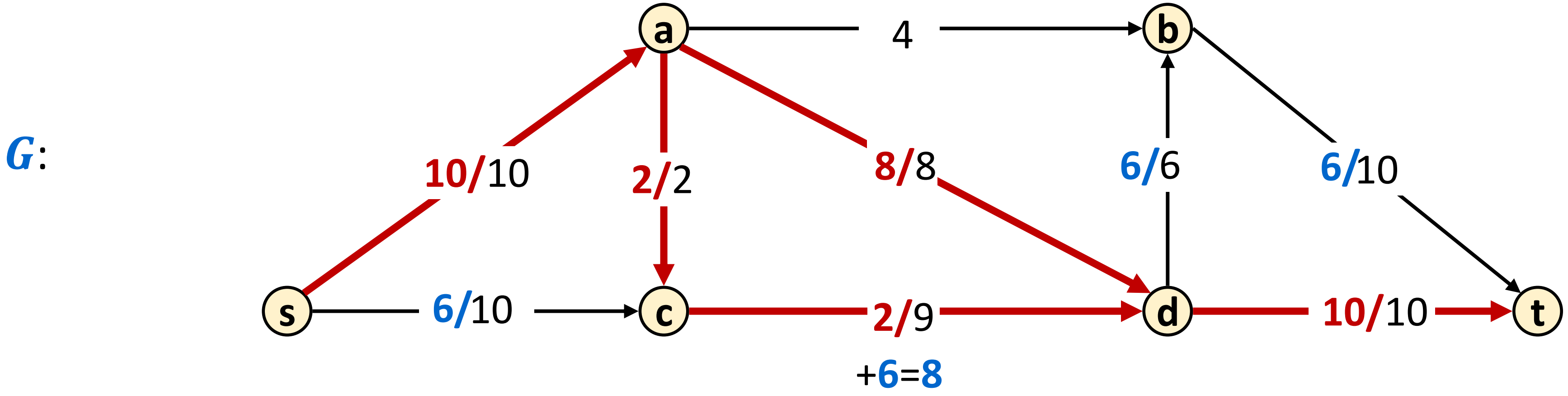


Flow value = **10**

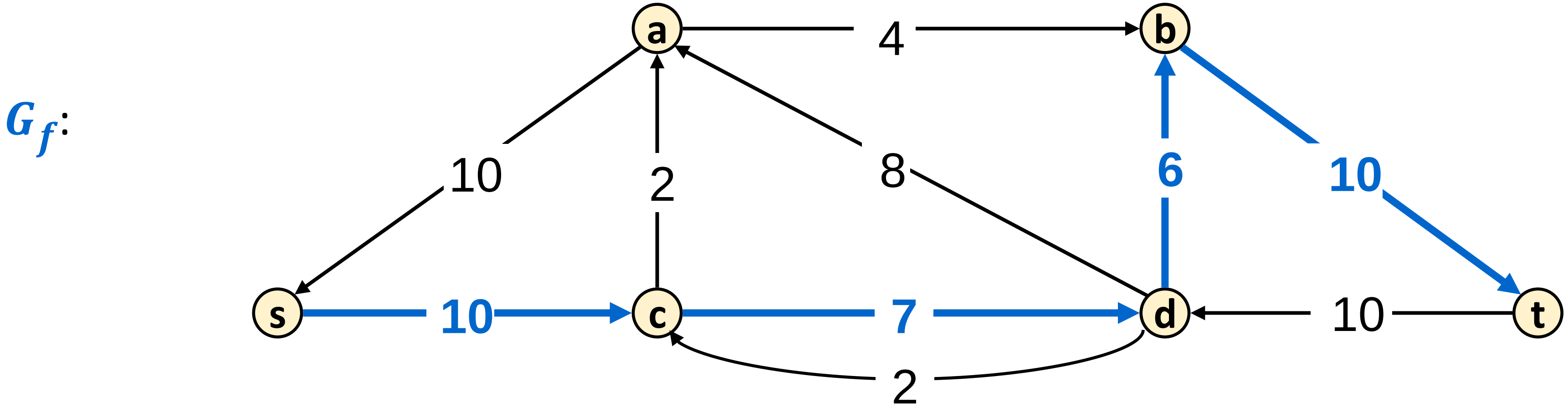
G_f:



Ford-Fulkerson Algorithm

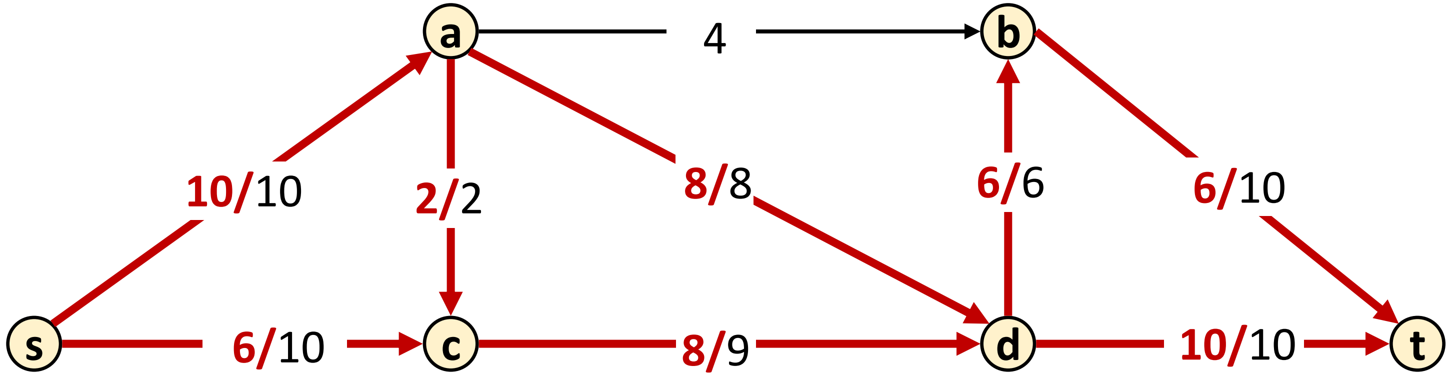


Flow value = **10**
+**6**=**16**



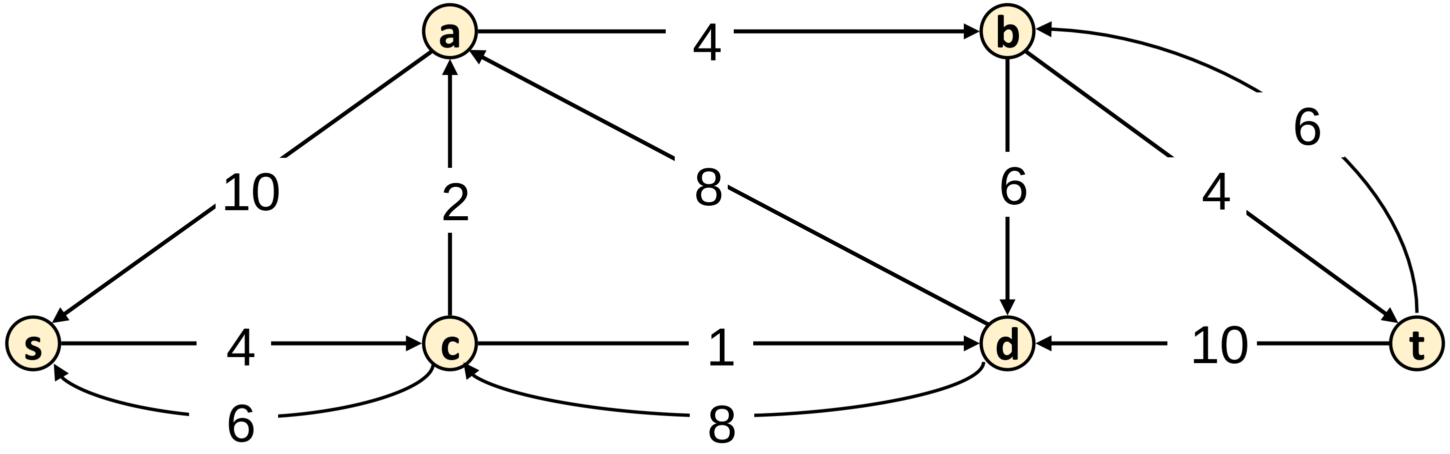
Ford-Fulkerson Algorithm

G:



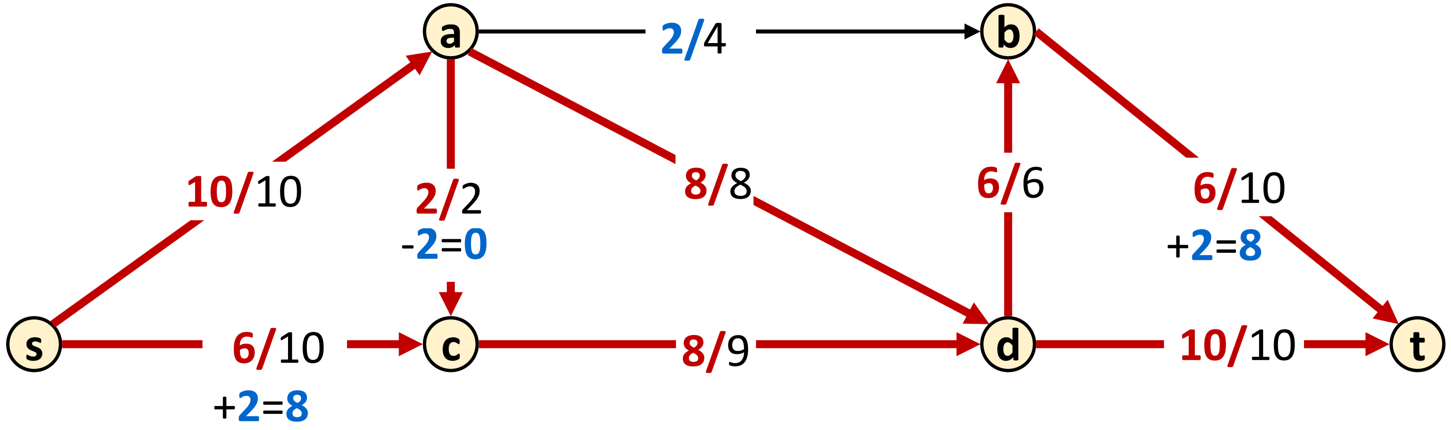
Flow value = **16**

G_f:



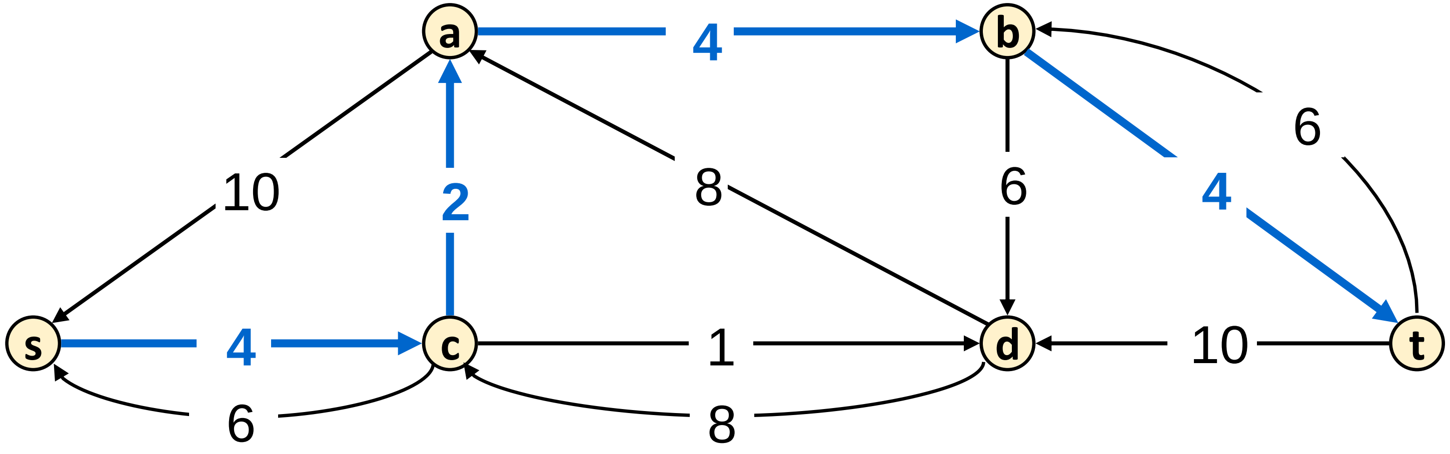
Ford-Fulkerson Algorithm

G:



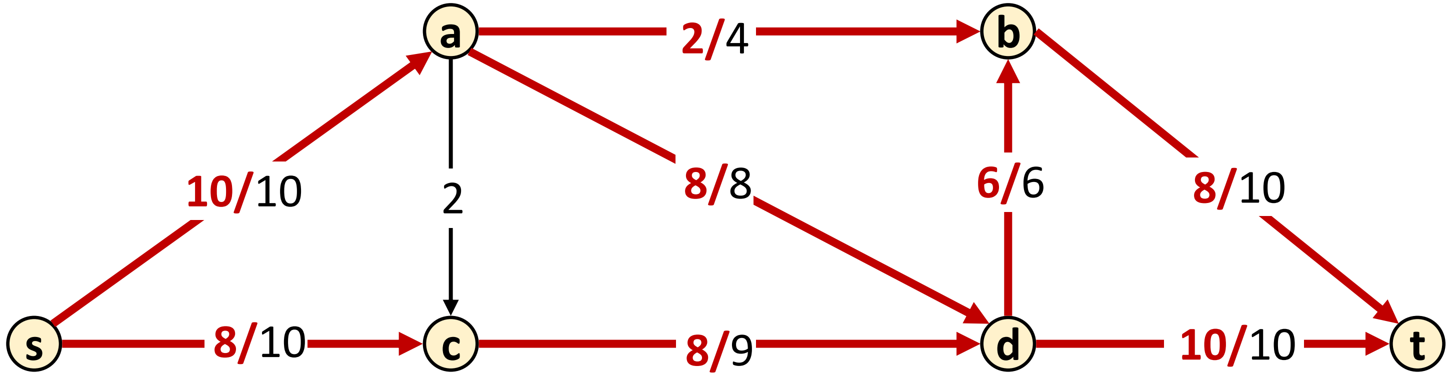
Flow value = **16**
+2=**18**

G_f:



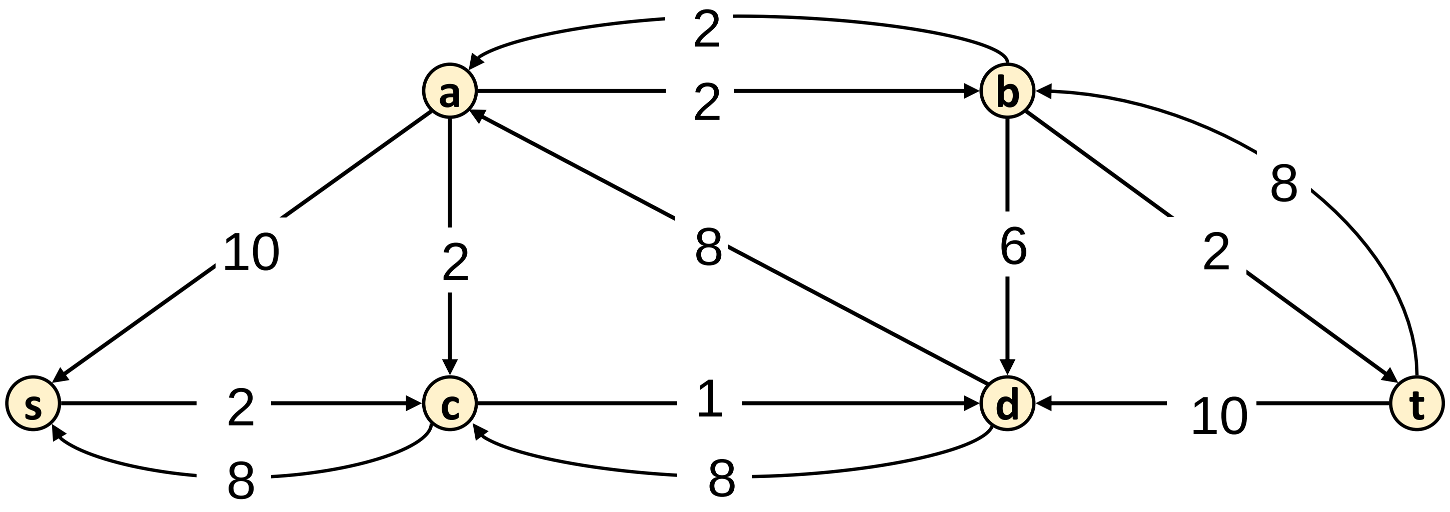
Ford-Fulkerson Algorithm

G:

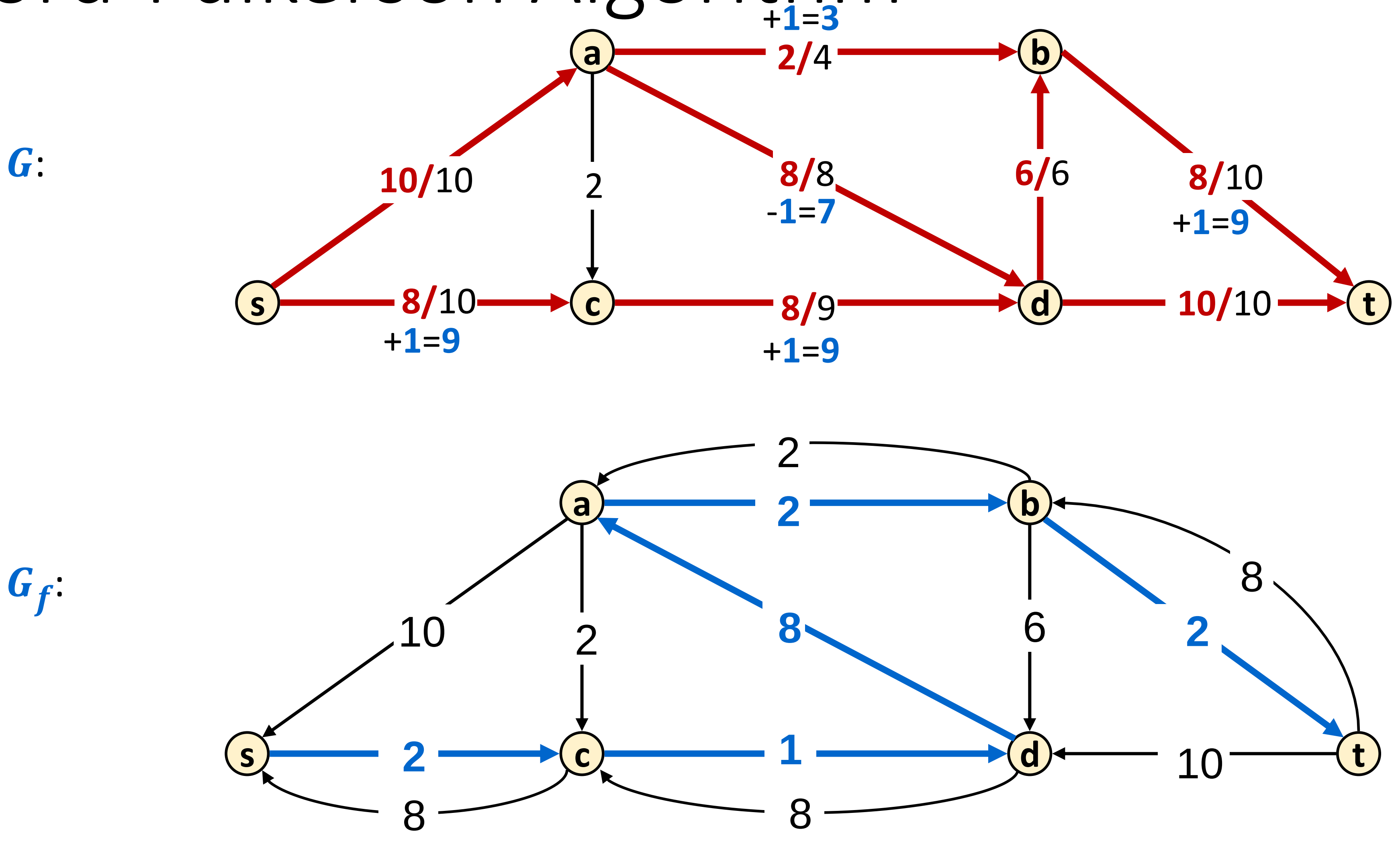


Flow value = **18**

G_f:



Ford-Fulkerson Algorithm

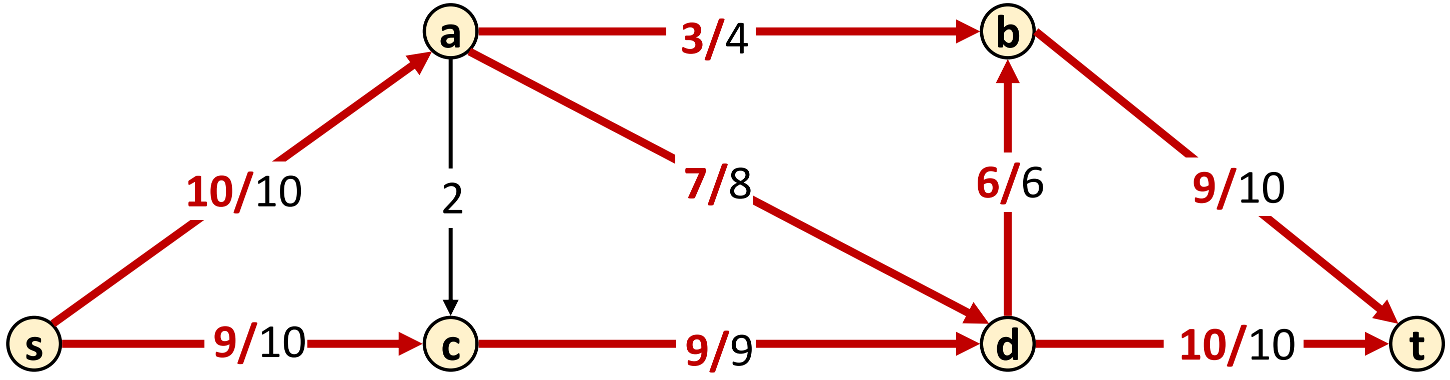


Flow value = 18

+1=19

Ford-Fulkerson Algorithm

G:



Flow value = **19**

G_f: