上海科技大学
ShanghaiTech University

CS240 Algorithm Design and Analysis

Lecture 6

Dynamic Programming (Cont.)
Network Flow

Quan Li

Fall 2025

2025.10.09

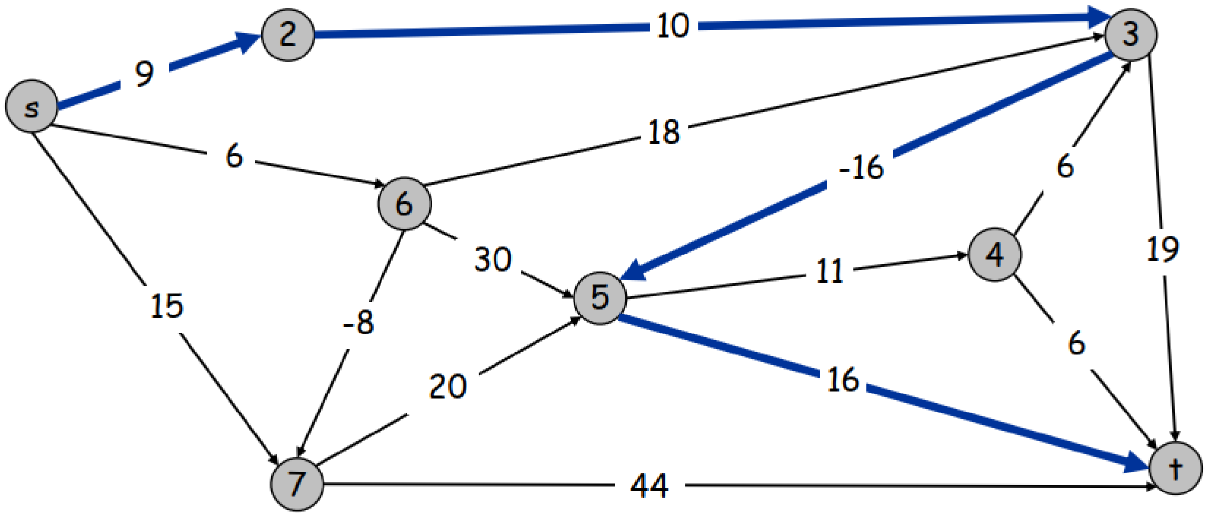
Shortest Paths

Shortest Paths

- Shortest path problem. Given a directed graph G = (V, E), with edge weights c_{vw} , find shortest path from node s to node t

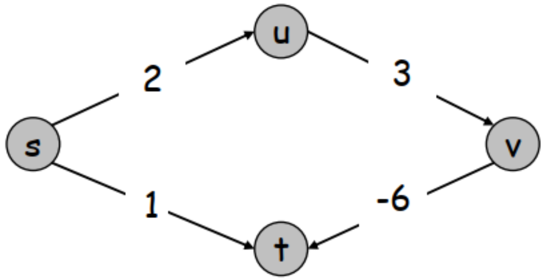
allow negative weights

- Ex. Nodes represent agents in a financial setting and c_{vw} is cost of transaction in which we buy from agent v and sell immediately to w

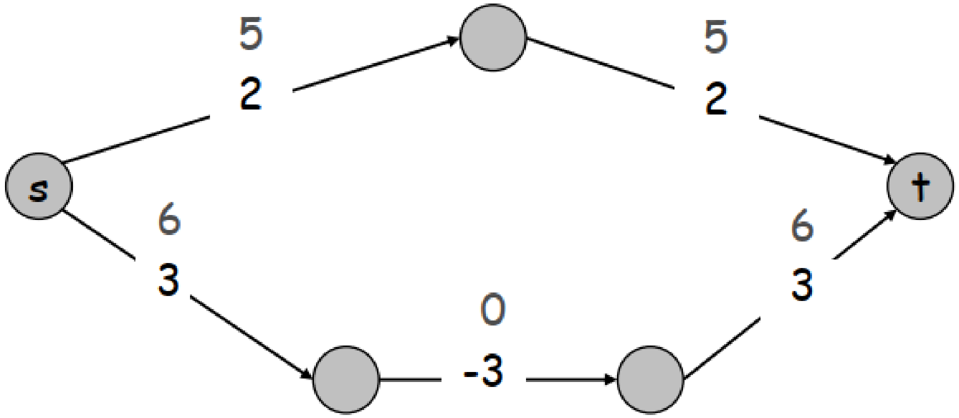


Shortest Paths: Failed Attempts

- **Dijkstra.** Can fail if negative edge costs

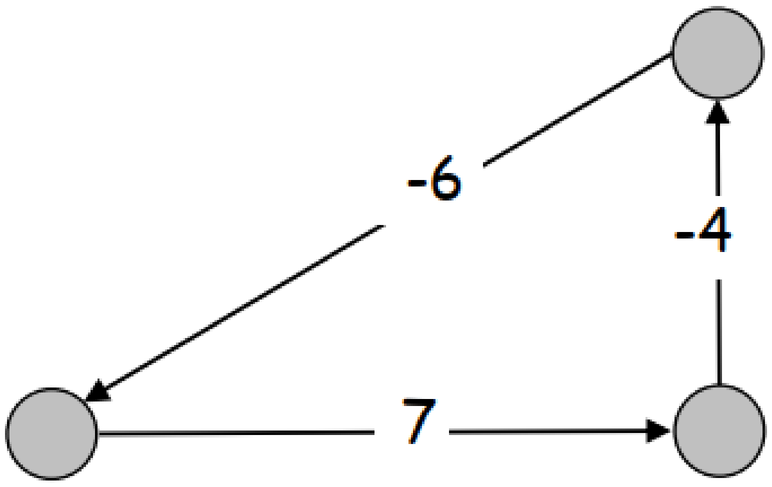


- **Re-weighting.** Adding a constant to every edge weight can fail

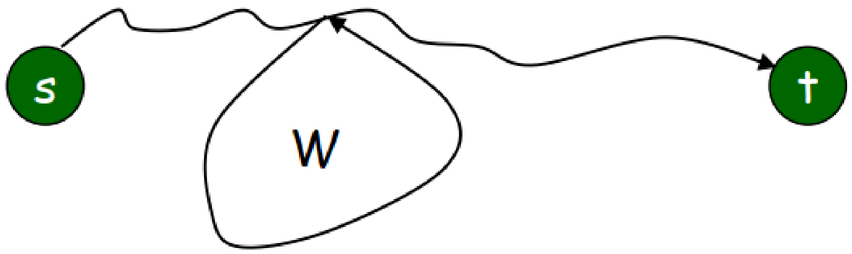


Shortest Paths: Negative Cost Cycles

- **Negative cost cycle**



- **Observation.** If some path from s to t contains a negative cost cycle, there does not exist a shortest s-t path; otherwise, there exists one that is simple



Shortest Paths: Dynamic Programming

- **Def.** OPT(i, v) = length of shortest v-t path P using at most i edges
- **Case 1:** P uses at most i-1 edges
  - OPT(i, v) = OPT(i-1, v)
- **Case 2:** P may use i edges
  - If (v, w) is first edge, then OPT uses (v, w), and then selects best w-t path using at most i-1 edges

$$OPT(i,v)=\begin{cases} \infty & \text{if } i=0, v\neq t \\ 0 & \text{if } v=t \\ \min\left\{OPT(i-1,v),\ \min\limits_{(v,w)\in E}\left\{OPT(i-1,w)+c_{vw}\right\}\right\} & \text{otherwise} \end{cases}$$

- **Remark.** By previous observation, if no negative cycles, then OPT(n-1, v) = length of shortest v-t path

Shortest Paths: Implementation


```
Shortest-Path(G, s, t) {
   foreach node v ∈ V
      M[0, v] ← ∞
   M[0, t] ← 0

   for i = 1 to n-1
      foreach node v ∈ V
         M[i, v] ← M[i-1, v]
      foreach edge (v, w) ∈ E
         M[i, v] ← min { M[i, v], M[i-1, w] + c_vw }

   return M[n-1, s]
}
```

Analysis. $\Theta(mn)$ time, $\Theta(n^2)$ space

Finding the shortest paths. Maintain a "successor" for each table entry

Shortest Paths: Improvements

```
Shortest-Path(G, s, t) {
   foreach node v ∈ V
      M[0, v] ← ∞
   M[0, t] ← 0

   for i = 1 to n-1
      foreach node v ∈ V
         M[i, v] ← M[i-1, v]
      foreach edge (v, w) ∈ E
         M[i, v] ← min { M[i, v], M[i-1, w] + c_vw }

   return M[n-1, s]
}
```

Practical improvements.

- Maintain only one array M[v] = shortest v-t path that we have found so far

Shortest Paths: Improvements

```
Shortest-Path(G, s, t) {
   foreach node v ∈ V
      M[v] ← ∞
   M[t] ← 0

   for i = 1 to n-1
      ~~foreach node v ∈ V~~
         ~~M[v] ← M[v]~~
      foreach edge (v, w) ∈ E
         M[v] ← min { M[v], M[w] + c_vw }

   return M[s]
}
```

Practical improvements.

- Maintain only one array M[v] = shortest v-t path that we have found so far
- No need to check edges of the form (v, w) unless M[w] changed in previous iteration

Bellman-Ford: Efficient Implementation

```
Push-Based-Shortest-Path(G, s, t) {
   foreach node v ∈ V
      M[v] ← ∞
   M[t] ← 0

   for i = 1 to n-1 {
      foreach node w ∈ V
         if (M[w] has been updated in previous iteration)
            foreach node v such that (v, w) ∈ E
               if (M[v] > M[w] + c_vw)
                  M[v] ← M[w] + c_vw
      If no M[w] value changed in iteration i, stop.
   }

   return M[s]
}
```

Analysis.

- O(n) extra space
- Time: O(mn) worst case, but substantially faster in practice

Application 1: Distance Vector Protocol

Distance Vector Protocol

- **Communication network**
  - Nodes ≈ routers
  - Edges ≈ direct communication link
  - Cost of edge ≈ delay on link ⟵ naturally nonnegative

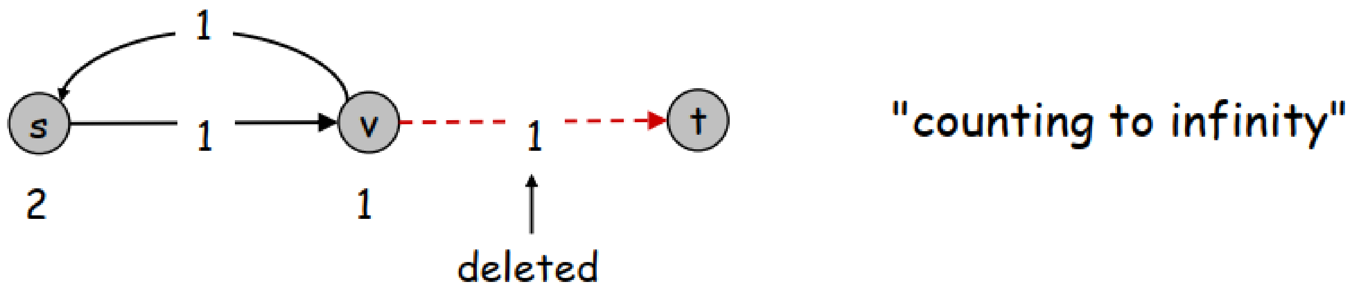
- **Dijkstra's algorithm.** Requires global information of network
- **Bellman-Ford.** Uses only local knowledge of neighboring nodes
- **Synchronization.** We don't expect routers to run in lockstep. The order in which each foreach loop executes is not important. Moreover, algorithm still converges even if updates are asynchronous

Distance Vector Protocol

- **Distance vector protocol**
  - Each router maintains a vector of shortest path lengths to every other node (distances) and the first hop on each path (directions)
  - Algorithm: each router performs n separate computations, one for each potential destination node
  - "Routing by rumor"

- **Ex.** RIP, Xerox XNS RIP, Novell's IPX RIP, Cisco's IGRP, DEC's DNA Phase IV, AppleTalk's RTMP

- **Caveat.** Edge costs may change during algorithm (or fail completely)



Path Vector Protocols

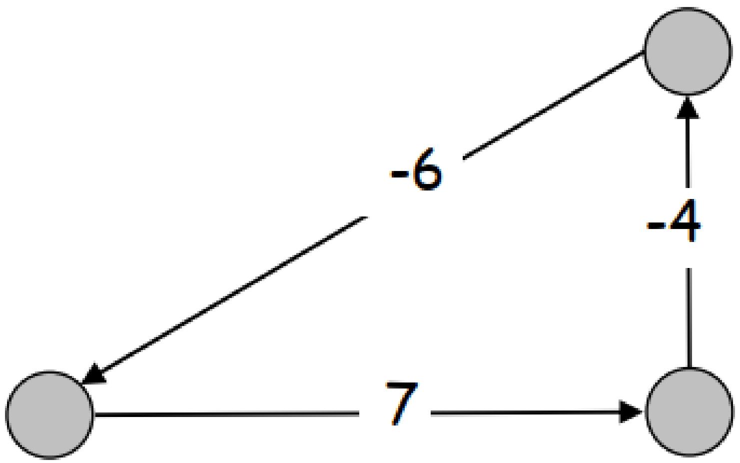
- **Link state routing**   not just the distance and first hop
    - Each router also stores the entire path
    - Avoids "counting-to-infinity" problem and related difficulties
    - Requires significantly more storage

- **Ex.** Border Gateway Protocol (BGP), Open Shortest Path First (OSPF)

Application 2: Negative Cycles in a Graph

Detecting Negative Cycles

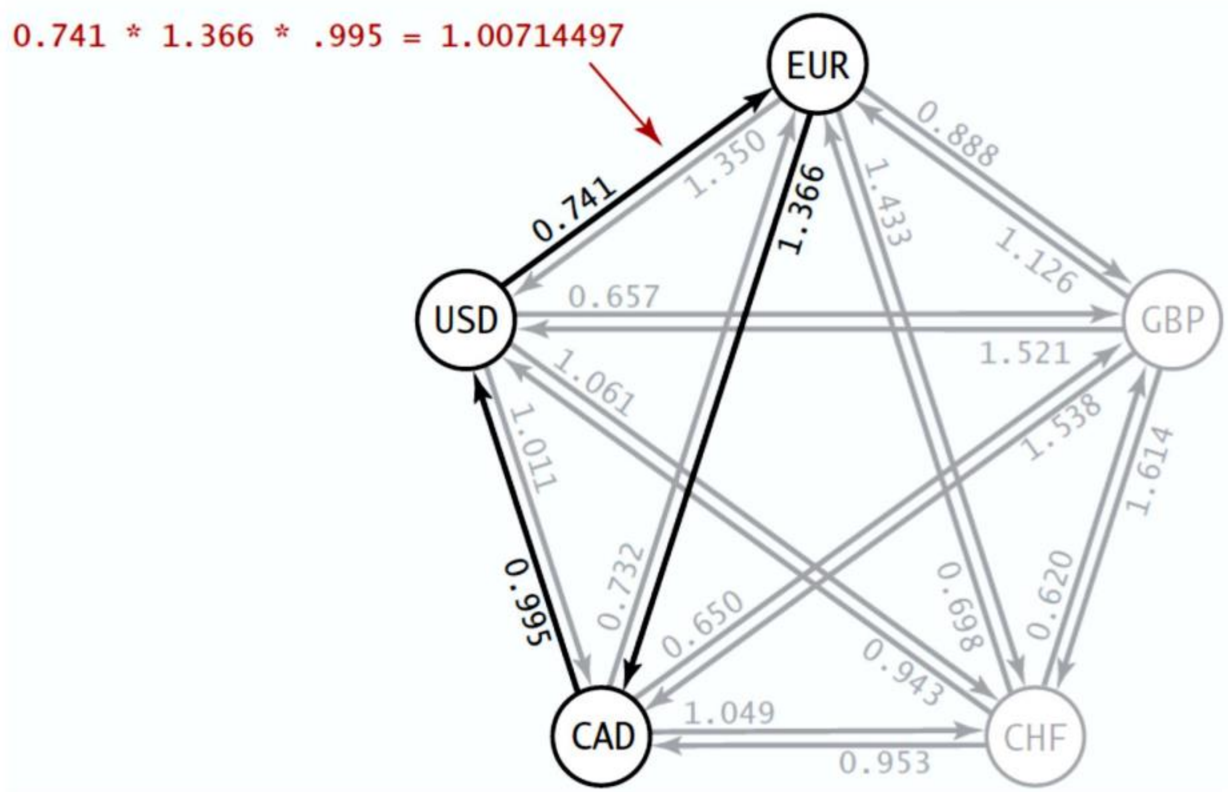
- **Negative cycle detection problem.** Given a digraph G = (V, E), with edge weights c_{vw}, find a negative cycle (if one exists)



Detecting Negative Cycles: Application

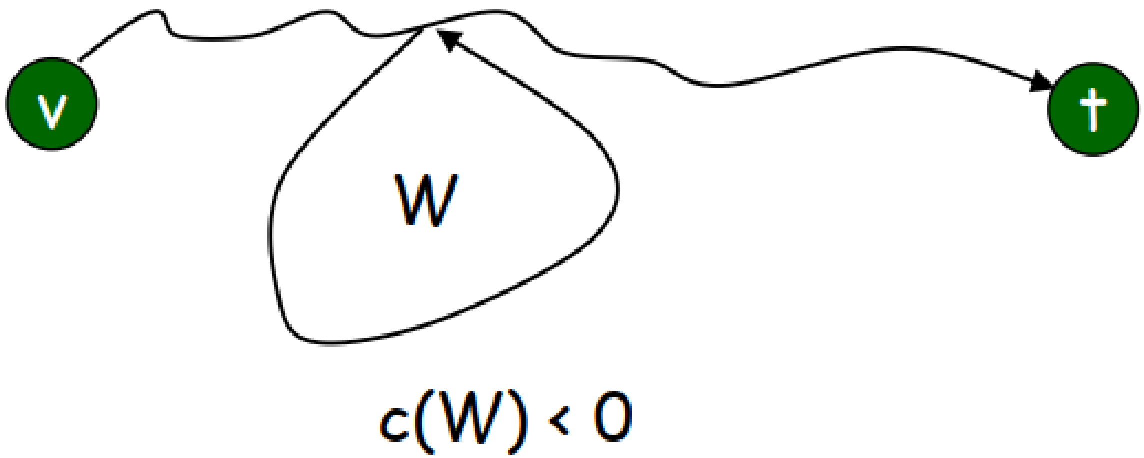
- **Currency conversion.** Given n currencies and exchange rates between pairs of currencies, is there an arbitrage opportunity?

- **Remark.** Fastest algorithm very valuable!



Detecting Negative Cycles

- **Lemma.** If OPT(n, v) = OPT(n-1, v) for all v, then there is no negative cycle with a path to t
- **Pf.** (by contradiction)
  - OPT(n, v) = OPT(n-1, v) → OPT(i, v) = OPT(n-1, v) for i >= n
  - But negative cycle in a path implies that OPT(i, v) always decreases as i increases

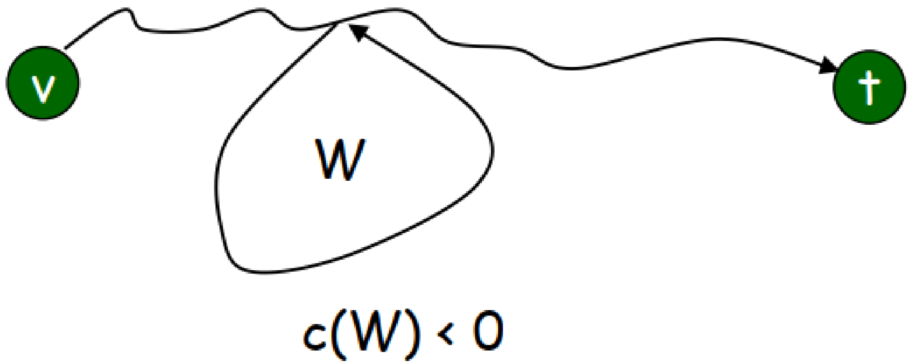


立志 成才 报国 裕民

Detecting Negative Cycles

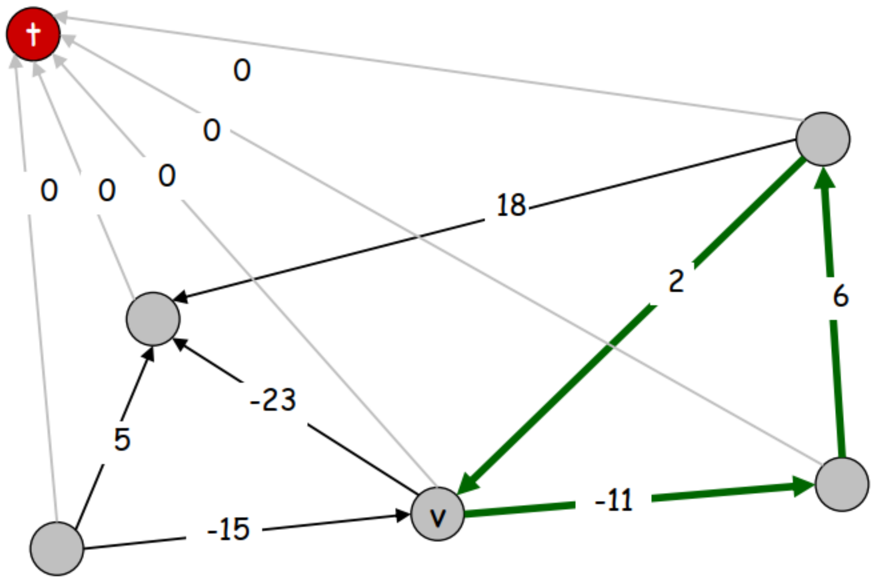
- **Lemma.** If OPT(n, v) < OPT(n-1, v) for some node v, then (any) shortest path from v to t contains a cycle W. Moreover, W has negative cost

- **Pf.** (by contradiction)
  - Since OPT(n, v) < OPT(n-1, v), we know the shortest v-t path P has exactly n edges
  - By pigeonhole principle, P must contain a directed cycle W
  - Deleting W yields a v-t path with < n edges → W has negative cost



Detecting Negative Cycles

- **Theorem.** Can detect negative cost cycle in O(mn) time
  - Add new node t and connect all nodes to t with 0-cost edge
  - Check if OPT(n, v) = OPT(n-1, v) for all nodes v
    - If yes, then no negative cycles
    - If no, then extract cycle from shortest path from v to t



Dynamic Programming: Summary

Dynamic Programming

- **Basic idea**
  - Polynomial number of sub-problems with a natural ordering from smallest to largest
  - Optimal solution to a sub-problem can be constructed from optimal solutions of smaller sub-problems
  - Sub-problems are overlapping!

- **Guideline**
  - Define the sub-problems
    - OPT(…)
  - Write down the recursive formulas
    - Ex: OPT(i) = max(f(OPT(j)), g(OPT(k)), …), j, k < i
  - Compute the formulas either bottom-up or top-down

Dynamic Programming

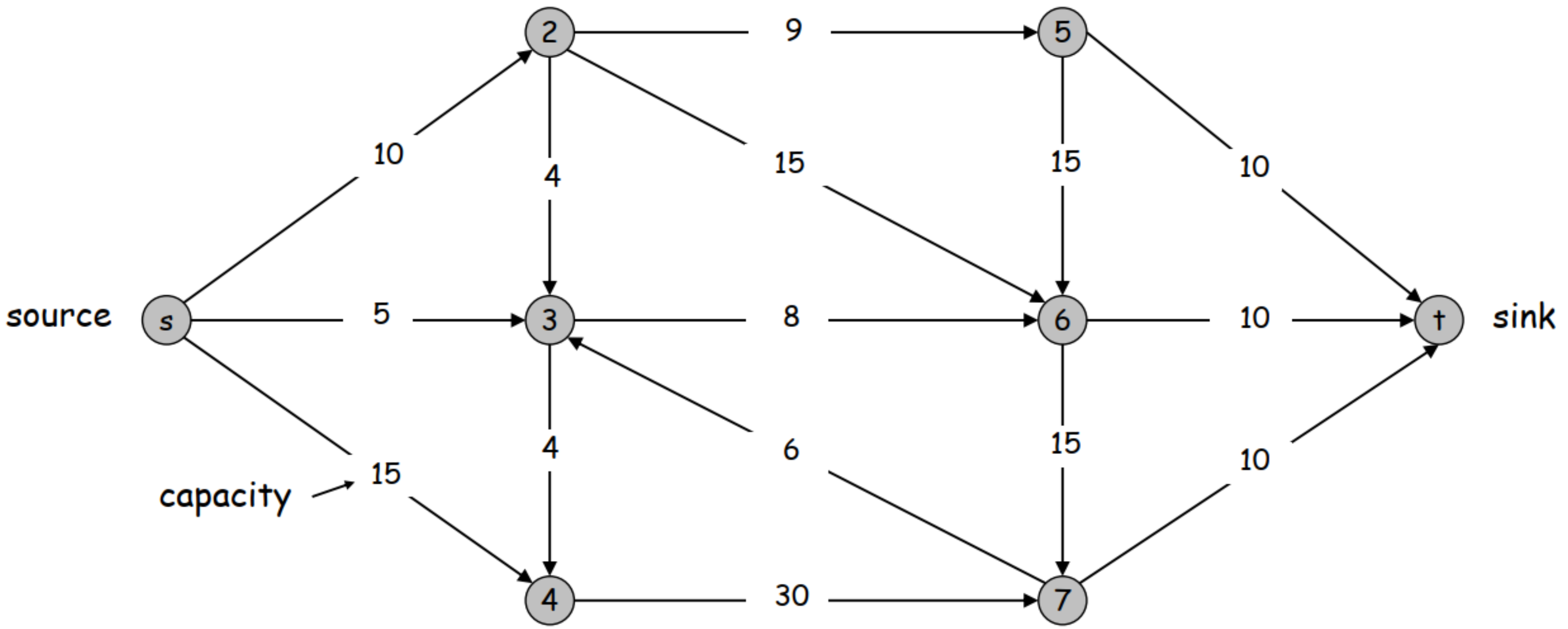
- **Algorithms**
  - Weighted interval scheduling
    - 1D array; binary choice
  - Knapsack
    - 2D array; adding a new variable (weight limit)
  - RNA secondary structure
    - 2D array: intervals
  - Sequence Alignment
    - 2D array: prefix alignment
  - Sequence Alignment in Linear Space
    - Combination of divide-and-conquer and dynamic programming
  - Shortest path with negative edges
    - (Bellman-Ford) 2D array: shortest path with edge number <= i
  - Distance Vector Protocol
  - Negative Cycle Detection

Network Flow

Max-flow and Ford-Fulkerson Algorithm

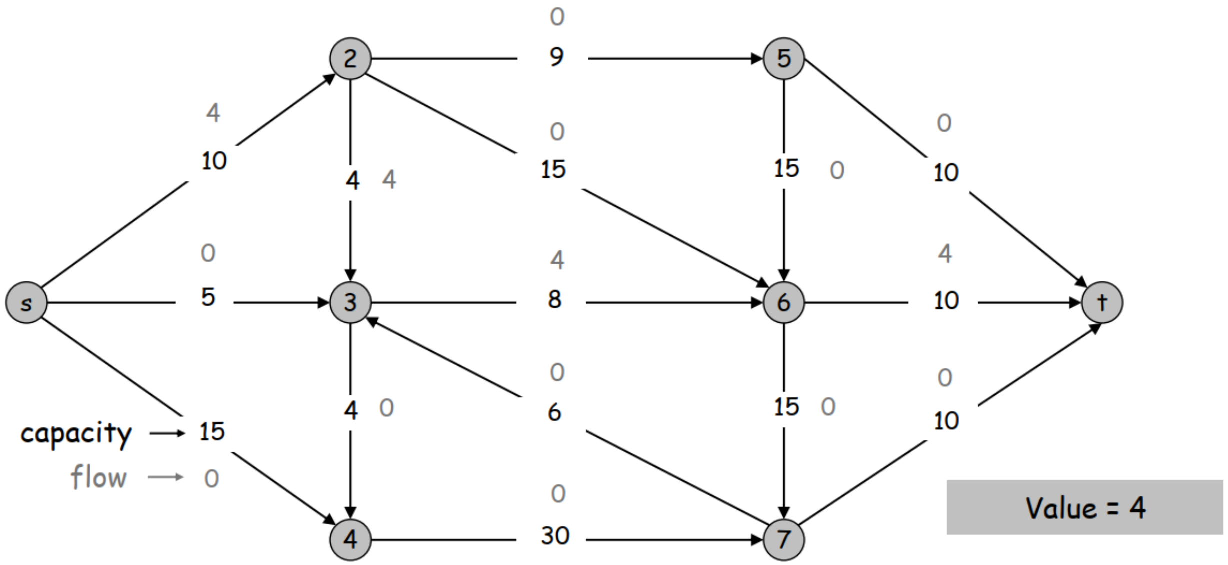
Flows

- Flow network
  - Abstraction for material **flowing** through the edges
  - G = (V, E) = directed graph
  - Two distinguished nodes: s = source, t = sink
  - c(e) = nonnegative capacity of edge e



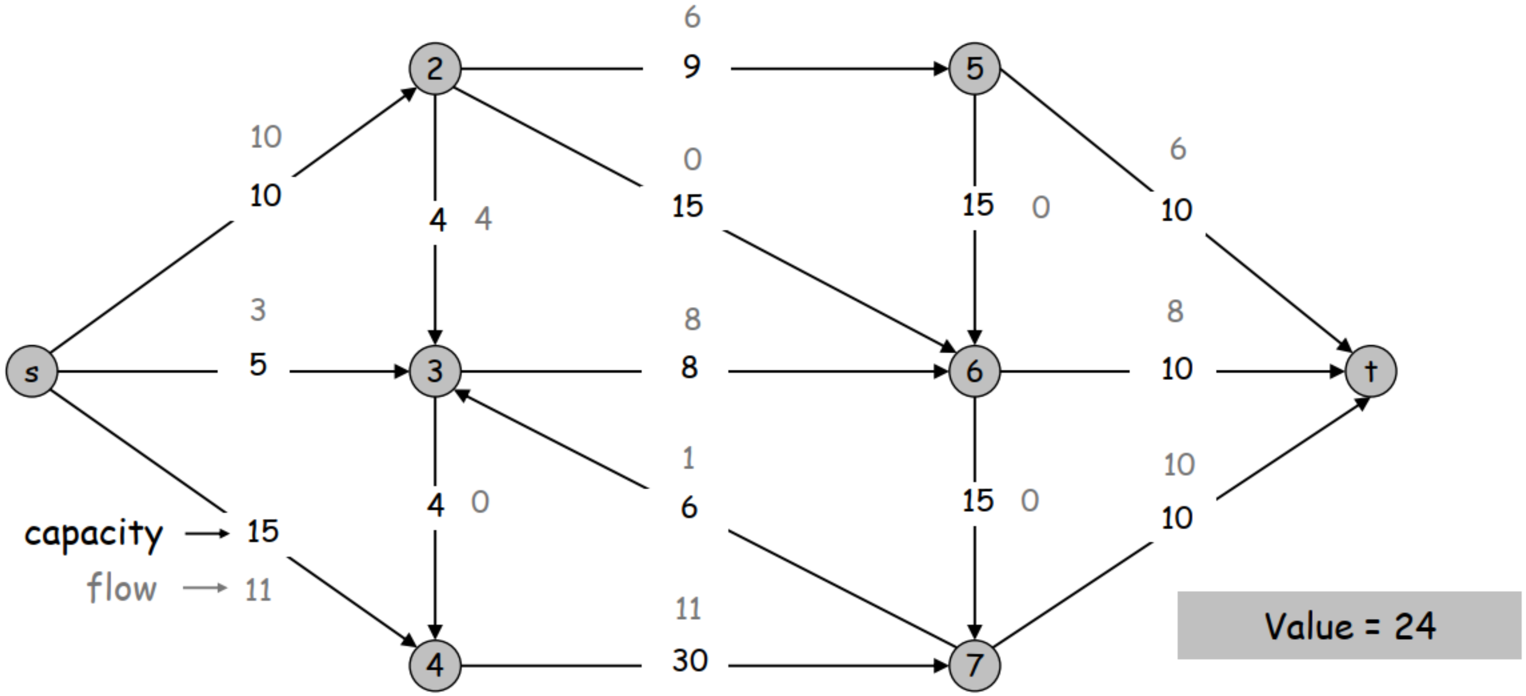
Flows

- **Def.** An **s-t flow** is a function that satisfies:
  - For each e ∈ E: $0 <= f(e) <= c(e)$ (capacity)
  - For each v ∈ V – {s, t}: $\sum_{e \text{ in to } v} f(e) = \sum_{e \text{ out of } v} f(e)$ ion)
- **Def.** The value of a flow f is : $v(f) = \sum_{e \text{ out of } s} f(e)$



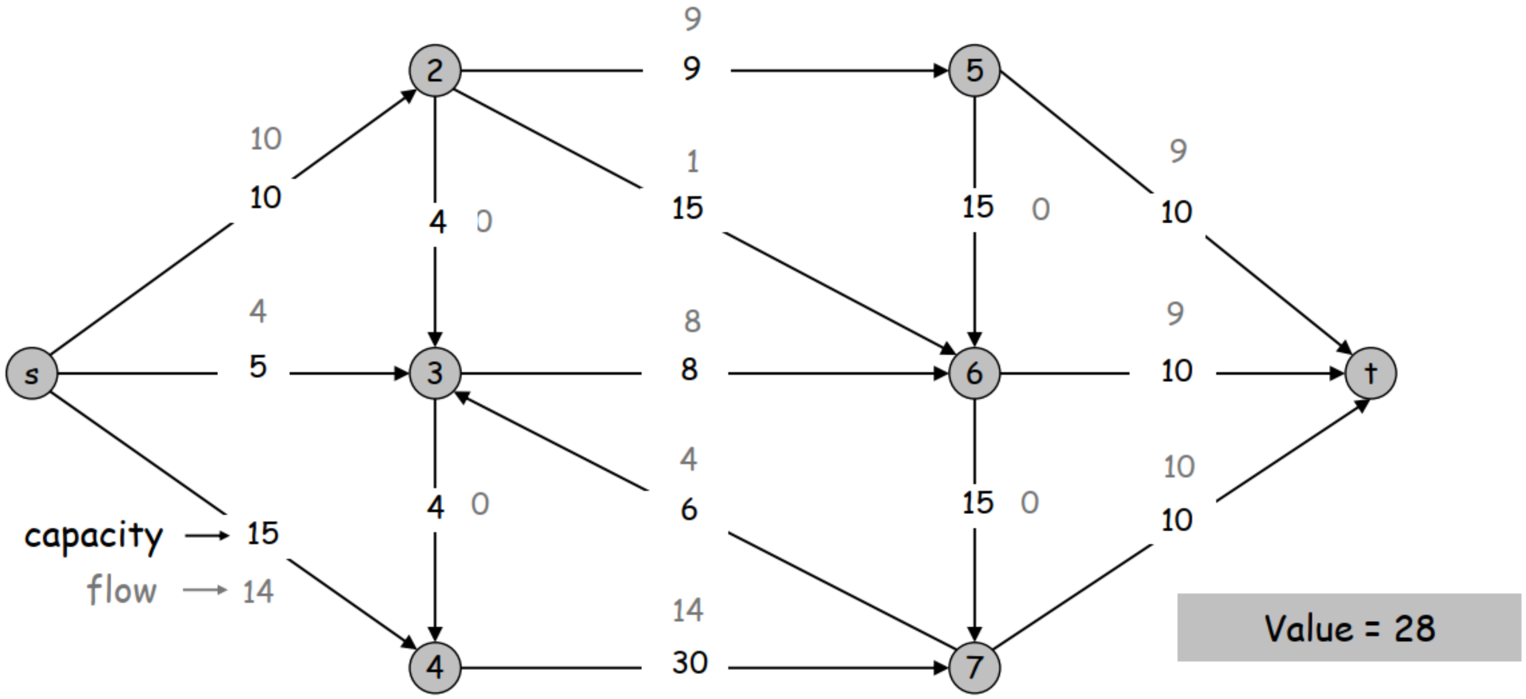
Flows

- **Def.** An **s-t flow** is a function that satisfies:
  - For each e ∈ E: $0 <= f(e) <= c(e)$ (capacity)
  - For each v ∈ V – {s, t}: $\sum_{e \text{ in to } v} f(e) = \sum_{e \text{ out of } v} f(e)$ tion)
- **Def.** The value of a flow f is : $v(f) = \sum_{e \text{ out of } s} f(e)$



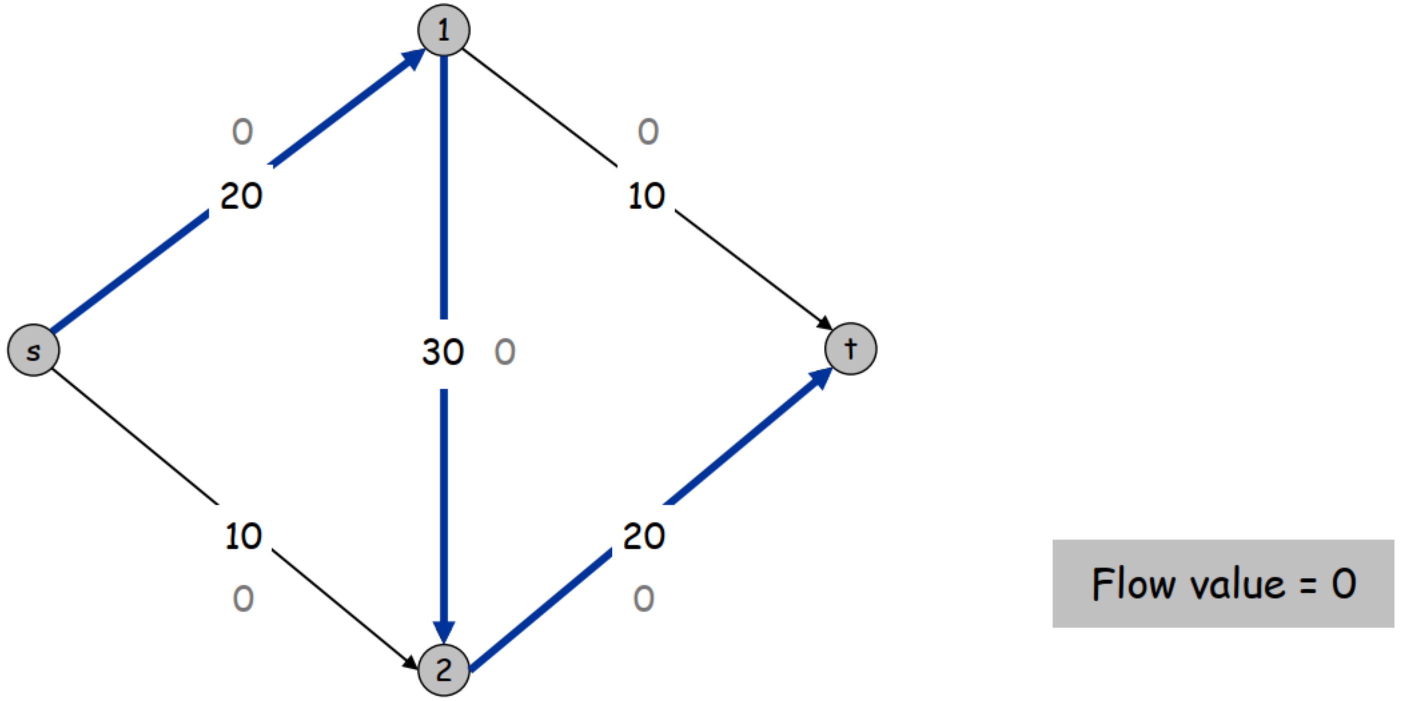
Maximum Flow Problem

- **Max flow problem.** Find s-t flow of maximum value



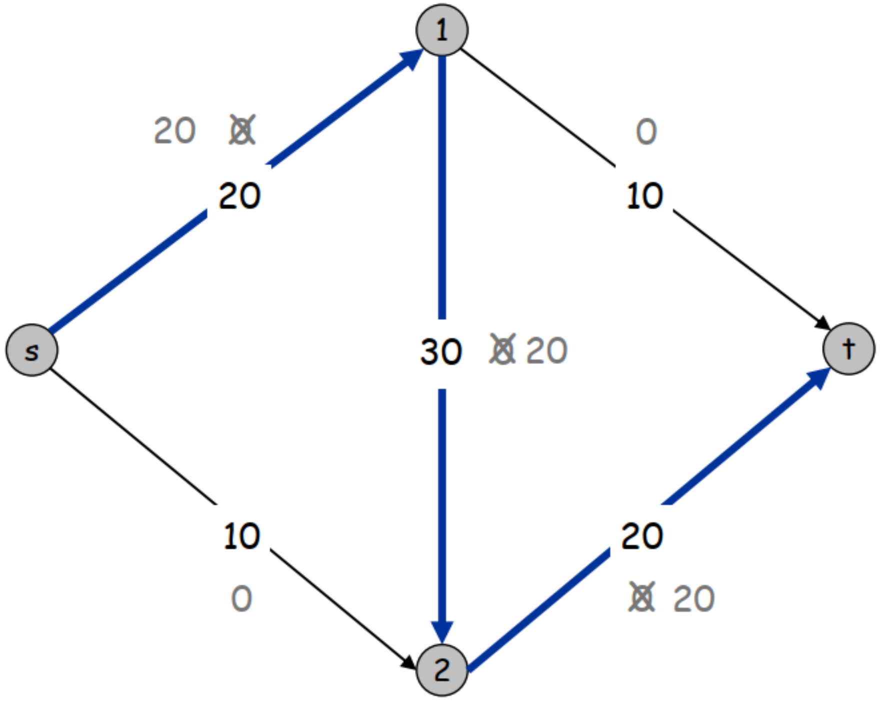
Towards a Max Flow Algorithm

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



Towards a Max Flow Algorithm

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

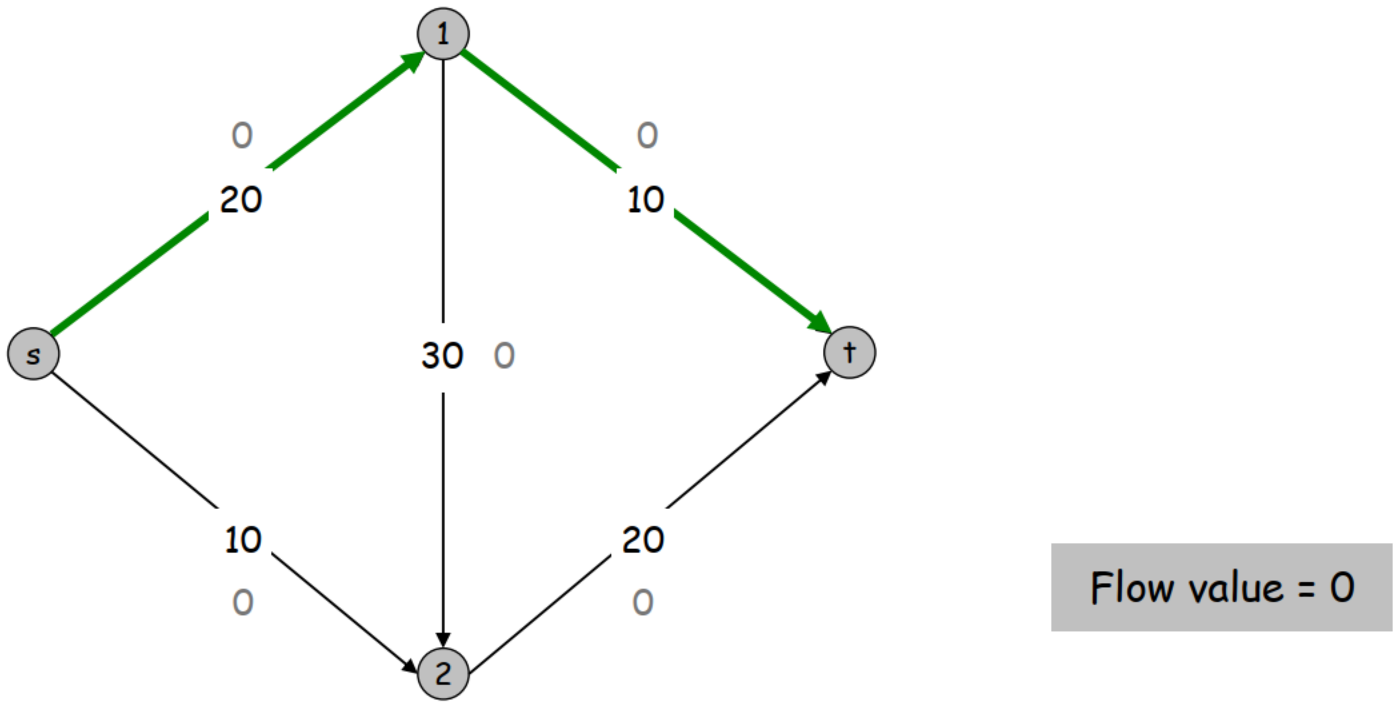


Flow value = 20

Towards a Max Flow Algorithm

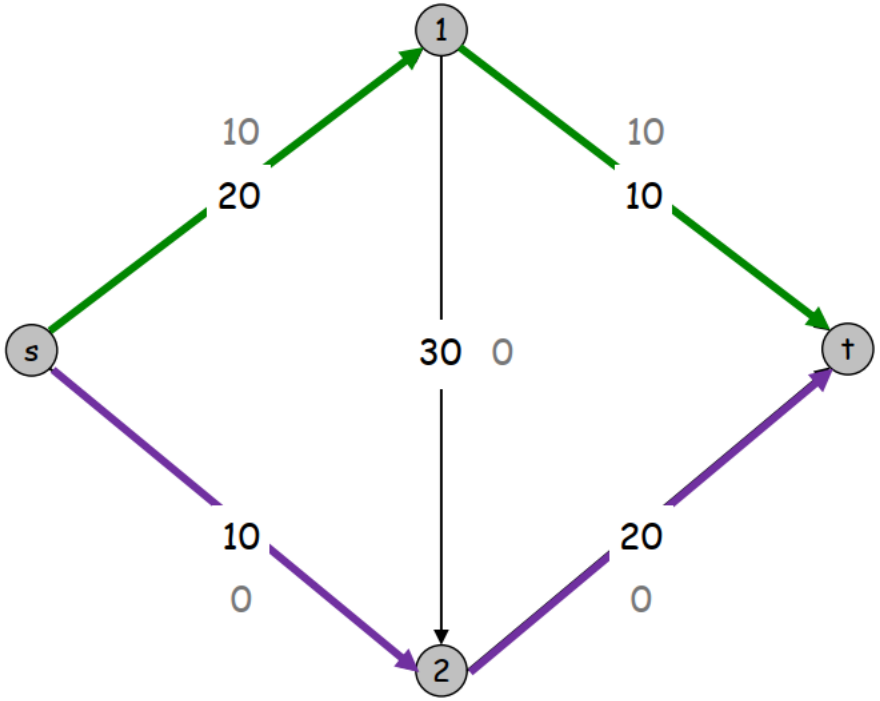
上海科技大学 ShanghaiTech University

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



Towards a Max Flow Algorithm

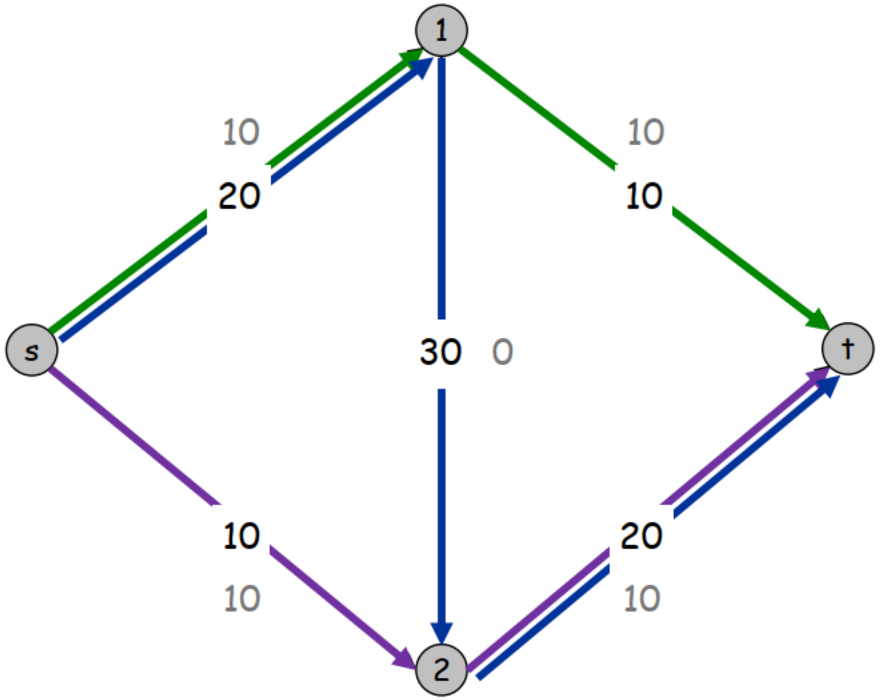
- **Greedy algorithm**
  - Start with f(e) = 0 for all edge e ∈ E
  - Find an s-t path P where each edge has f(e) < c(e)
  - Augment flow along path P
  - Repeat until you get stuck



Flow value = 10

Towards a Max Flow Algorithm

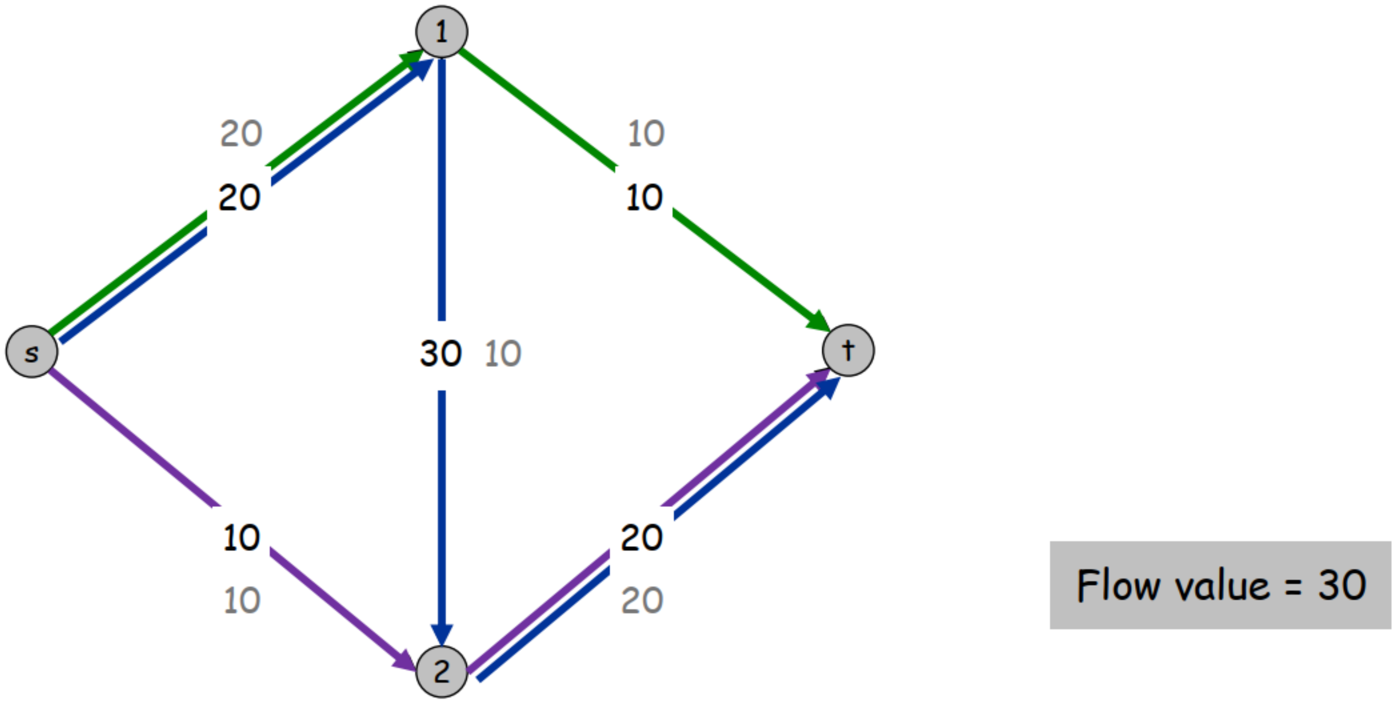
- **Greedy algorithm**
  - Start with f(e) = 0 for all edge e ∈ E
  - Find an s-t path P where each edge has f(e) < c(e)
  - Augment flow along path P
  - Repeat until you get stuck



Flow value = 20

Towards a Max Flow Algorithm

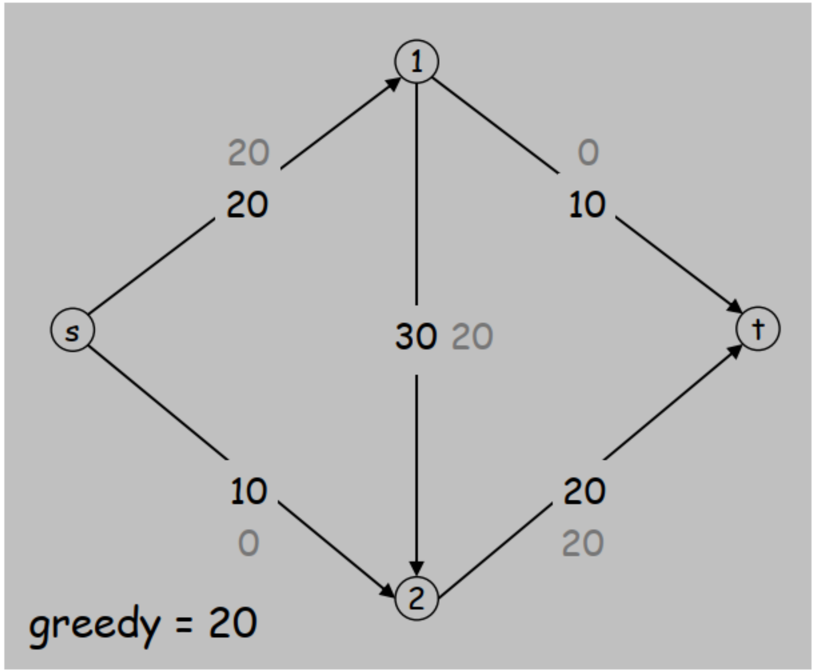
- **Greedy algorithm**
  - Start with f(e) = 0 for all edge e ∈ E
  - Find an s-t path P where each edge has f(e) < c(e)
  - Augment flow along path P
  - Repeat until you get stuck

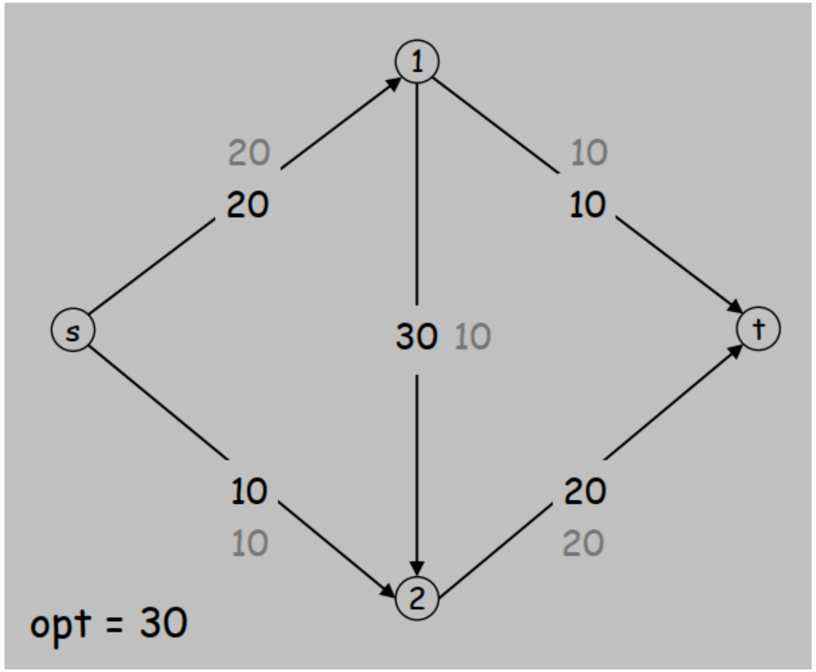


Towards a Max Flow Algorithm

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

locally optimality ≠ globally optimality



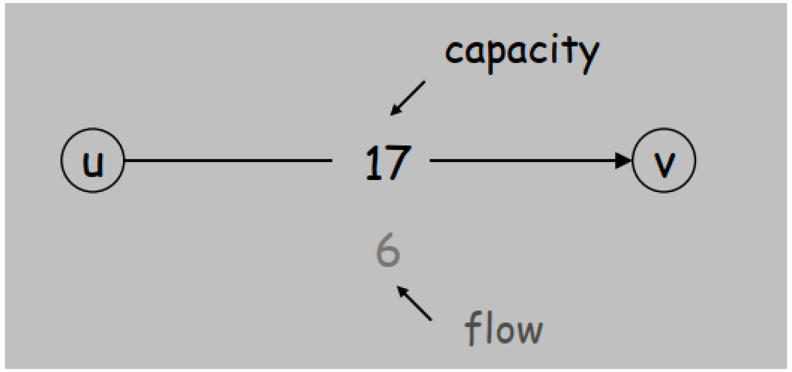


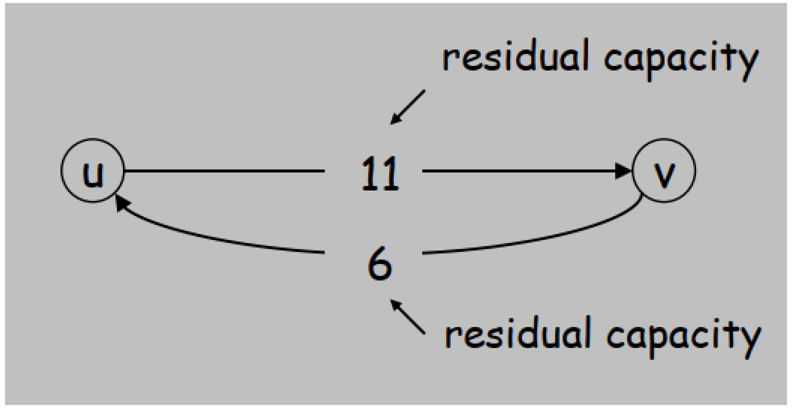
Residual Graph

- **Original edge:** $e = (u, v) \in E$
  - Flow $f(e)$, capacity $c(e)$
- **Residual edge**
  - "Undo" flow sent
  - $e = (u, v)$ and $e^R = (v, u)$
  - Residual capacity:

$$c_f(e) = \begin{cases} c(e) - f(e) & \text{if } e \in E \\ f(e) & \text{if } e^R \in E \end{cases}$$

- **Residual graph:** $G_f = (V, E_f)$
  - Residual edges with positive residual capacity
  - $E_f = \{e: f(e) < c(e)\} \cup \{e^R: f(e) > 0\}$





Augmenting Path

- **Augmenting path:** a simple s-t path P in the residual graph G_f
- **Bottleneck capacity** of an augmenting path P is the minimum residual capacity of any edge in P

```
Augment(f, c, P) {
   b ← bottleneck(P)
   foreach e ∈ P {
      if (e ∈ E) f(e) ← f(e) + b          forward edge
      else       f(eR) ← f(eR) - b        reverse edge
   }
   return f
}
```

- **Claim:** After augmentation, f is still a flow

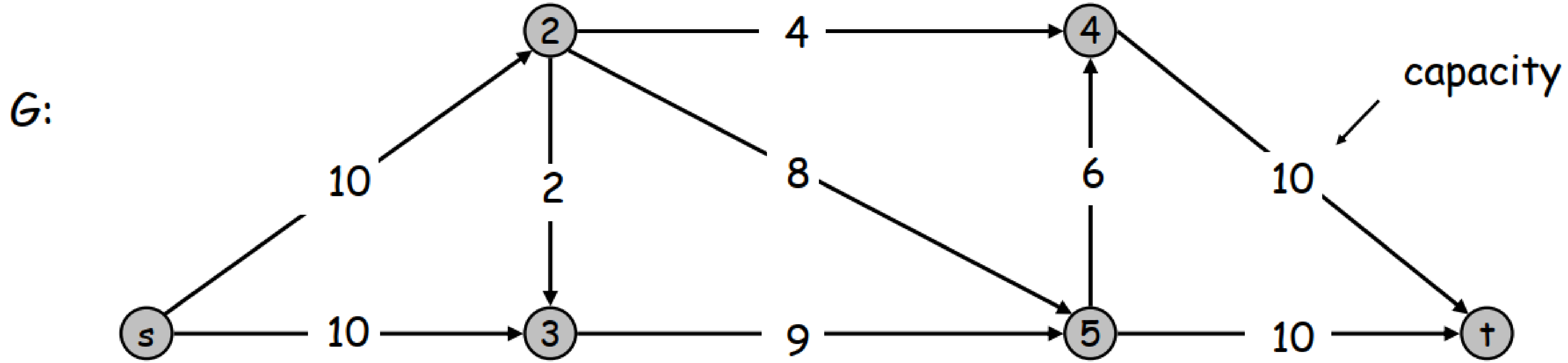
Ford-Fulkerson Algorithm

- **Ford-Fulkerson Algorithm**
  - Start with $f(e) = 0$ for all edge $e \in E$
  - Find an augmenting path P in the residual graph G_f
  - Augment flow along path P
  - Repeat until you get stuck

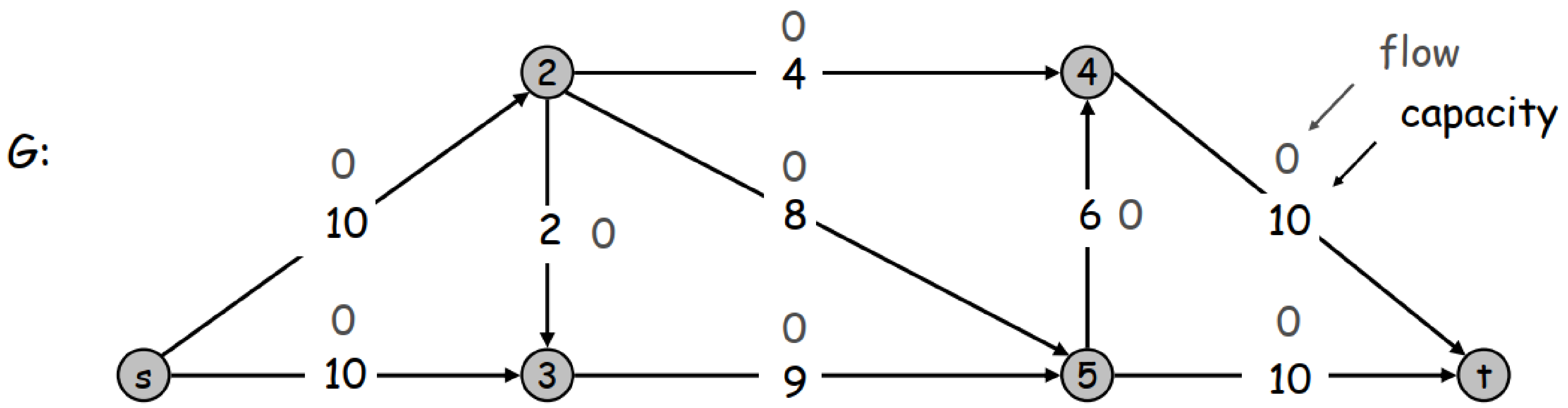
```
Ford-Fulkerson(G, s, t, c) {
   foreach e ∈ E  f(e) ← 0
   G_f ← residual graph

   while (there exists augmenting path P) {
      f ← Augment(f, c, P)
      update G_f
   }
   return f
}
```

立志成才报国裕民

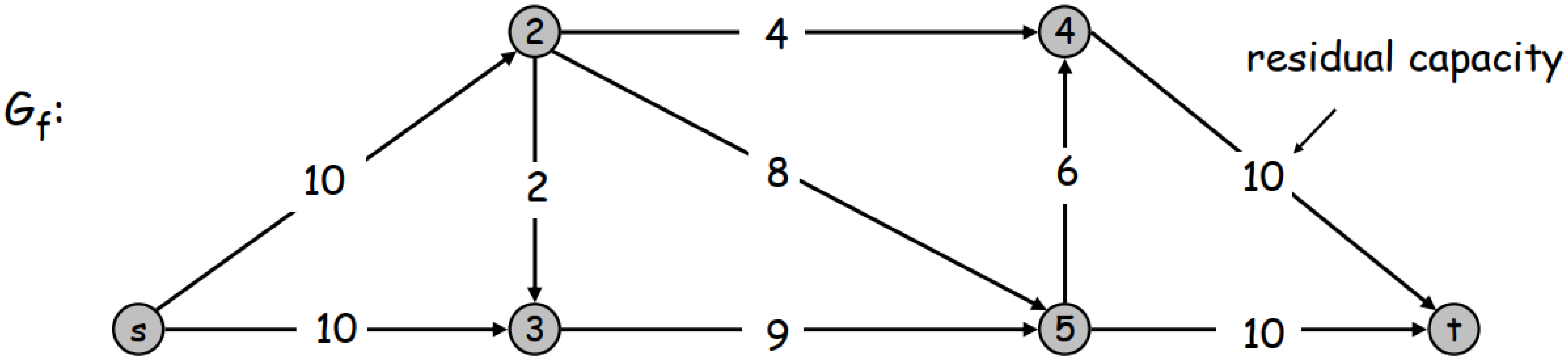
Ford-Fulkerson Algorithm



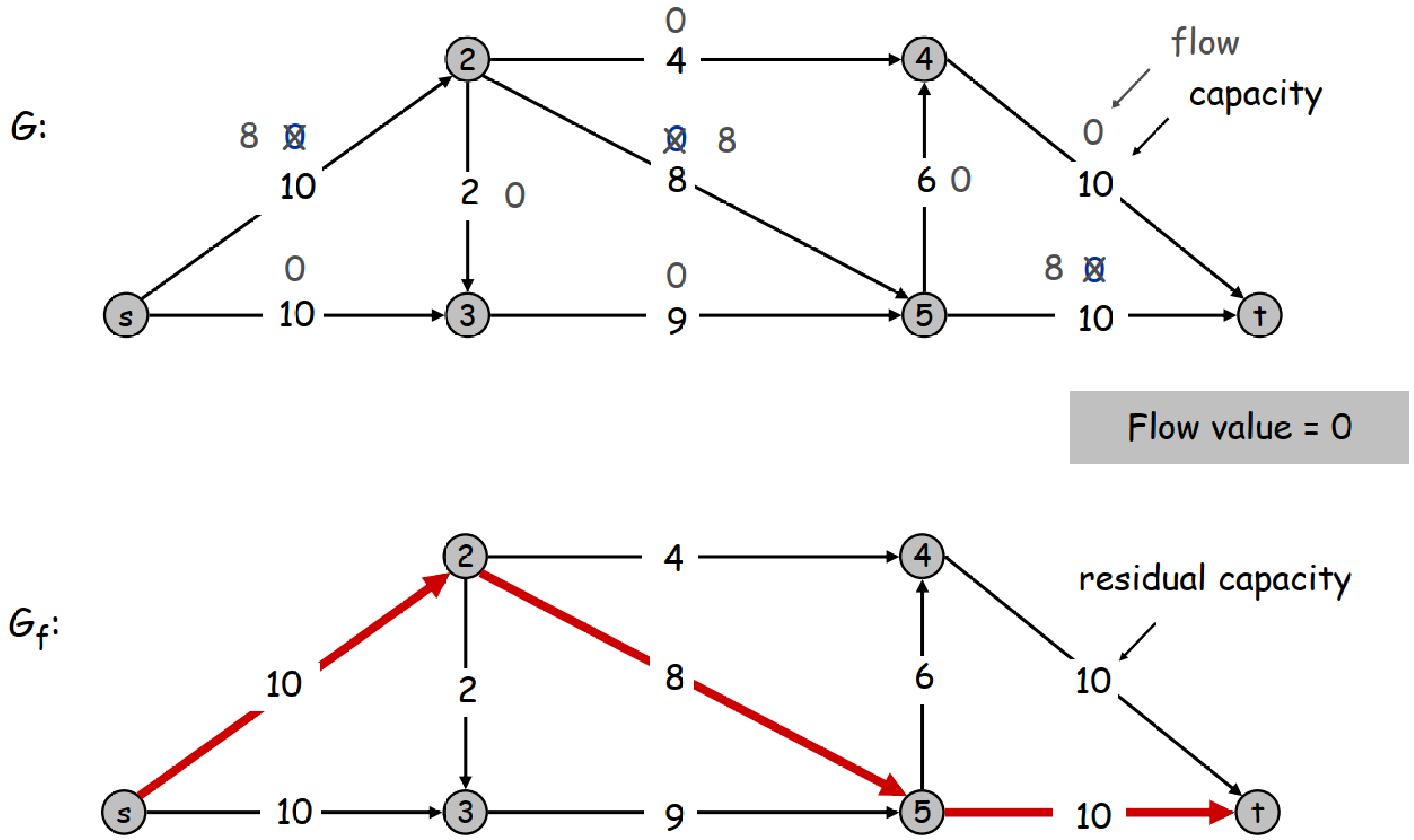
Ford-Fulkerson Algorithm



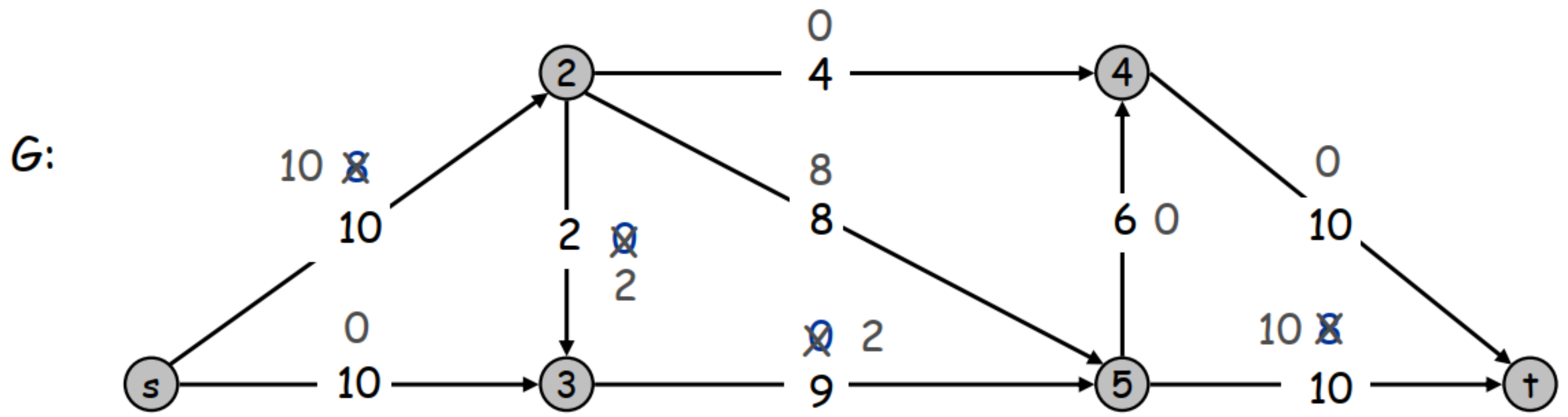
Flow value = 0



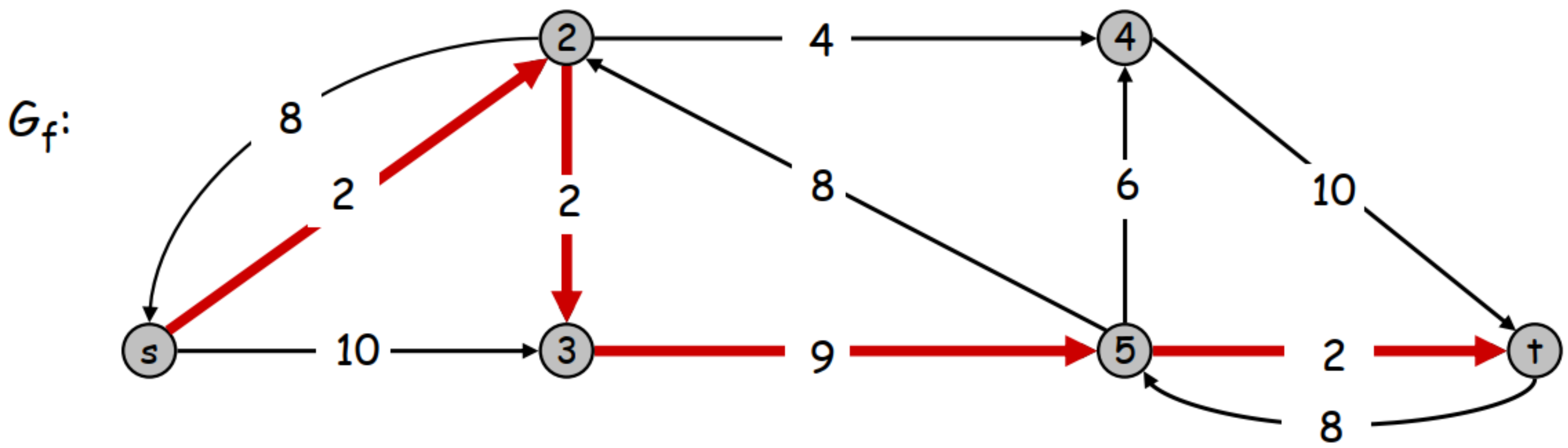
Ford-Fulkerson Algorithm



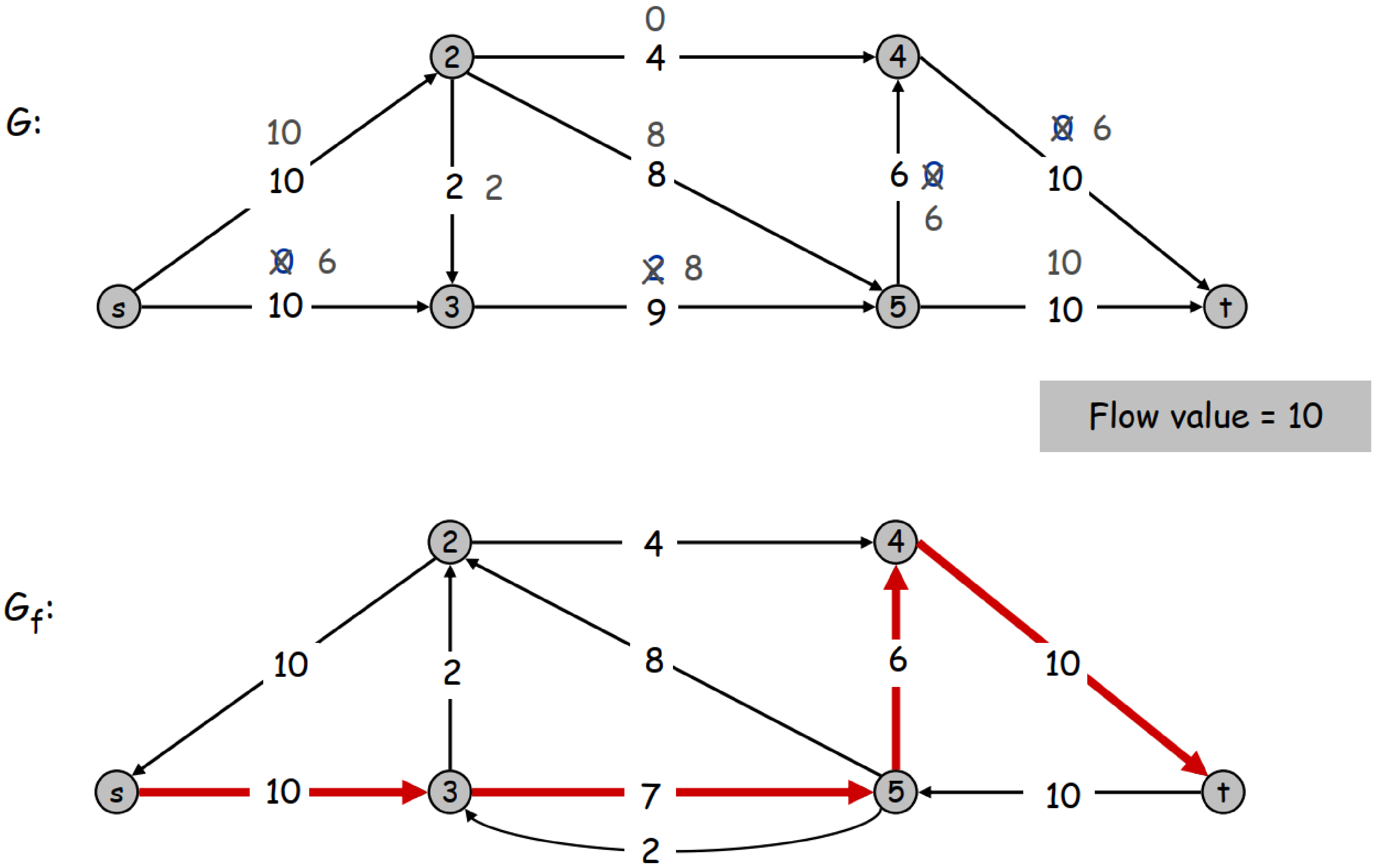
Ford-Fulkerson Algorithm



Flow value = 8

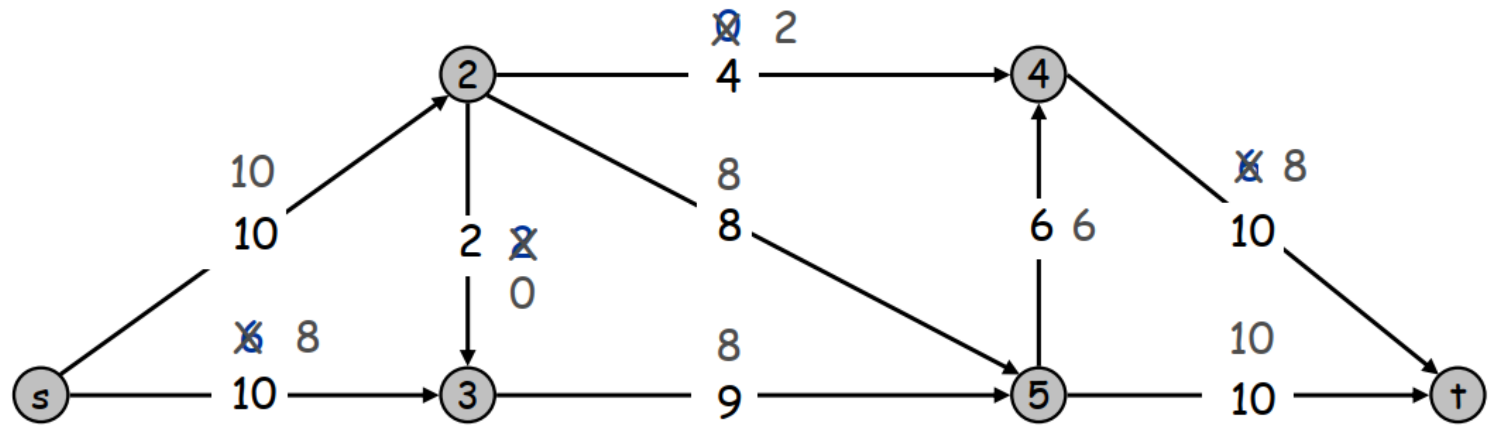


Ford-Fulkerson Algorithm



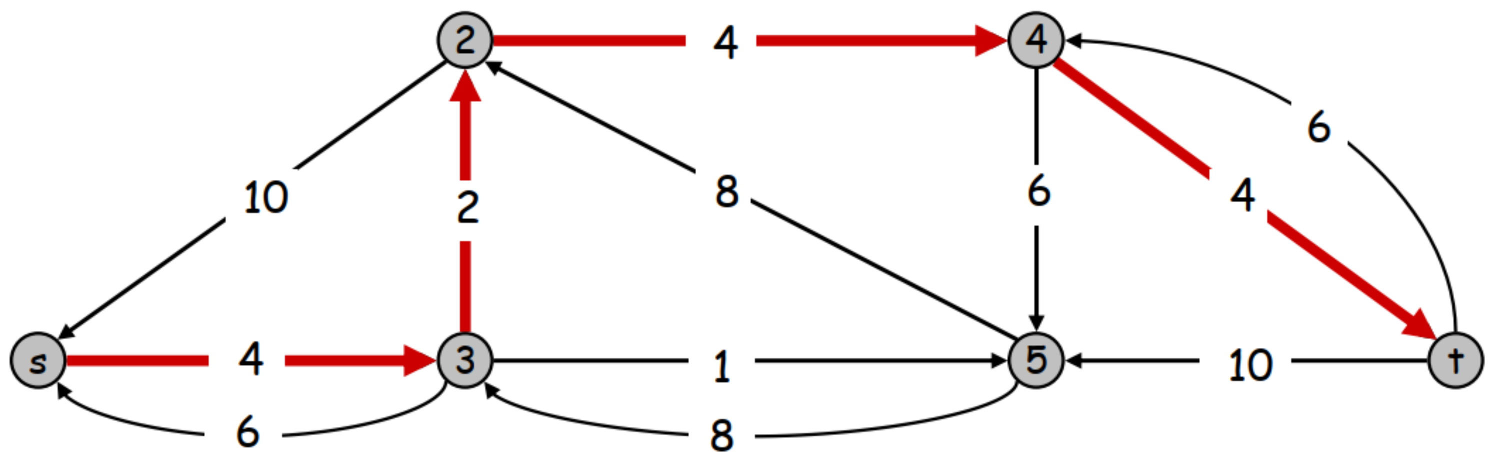
Ford-Fulkerson Algorithm

G:

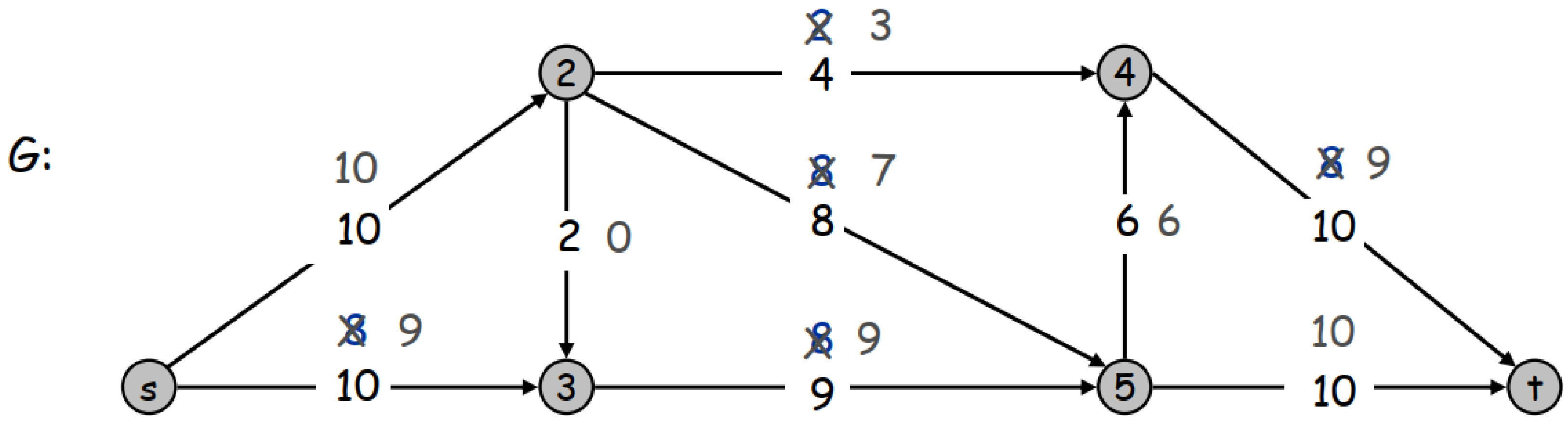


Flow value = 16

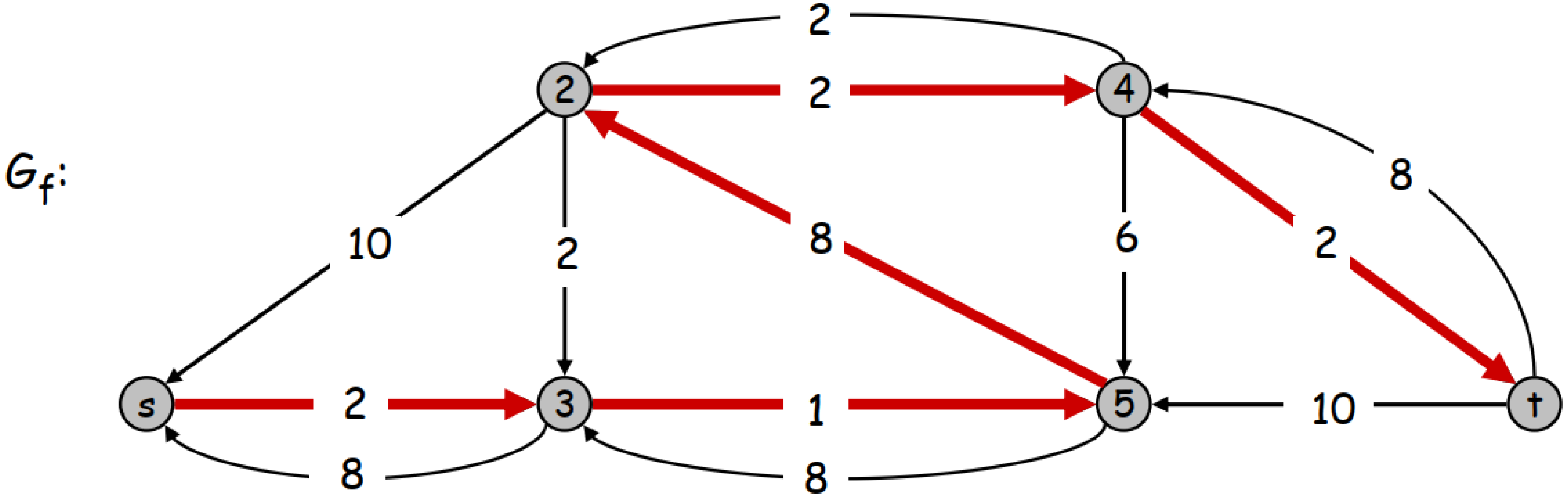
G_f:



Ford-Fulkerson Algorithm



Flow value = 18



Ford-Fulkerson Algorithm