南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Algorithm Design and Analysis (H)

CS216

Instructor: Shan CHEN (陈杉)

chens3@sustech.edu.cn

(slides edited from Prof. Shiqi Yu)

SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

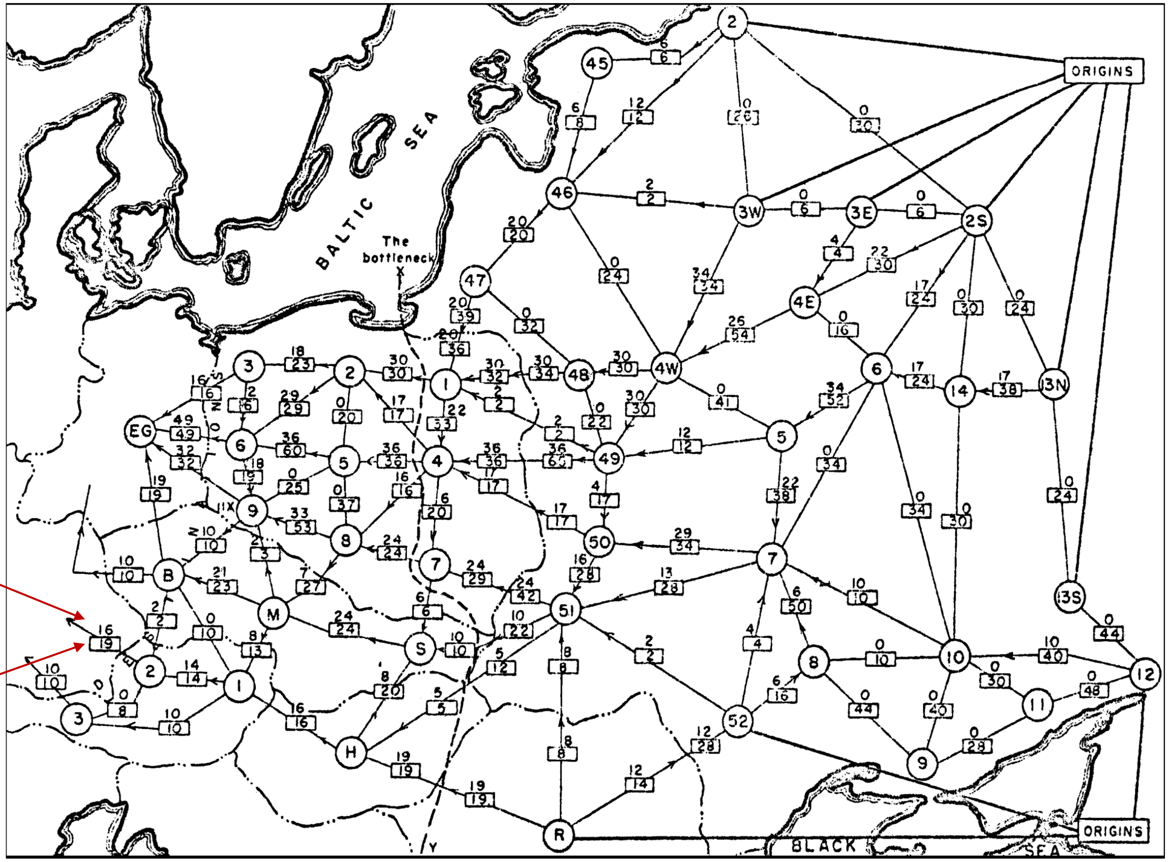
Network Flow

Maximum Flow Application (Tolstoǐ 1930s)

- **Soviet Union goal.** Maximize flow of supplies to Eastern Europe.

flow

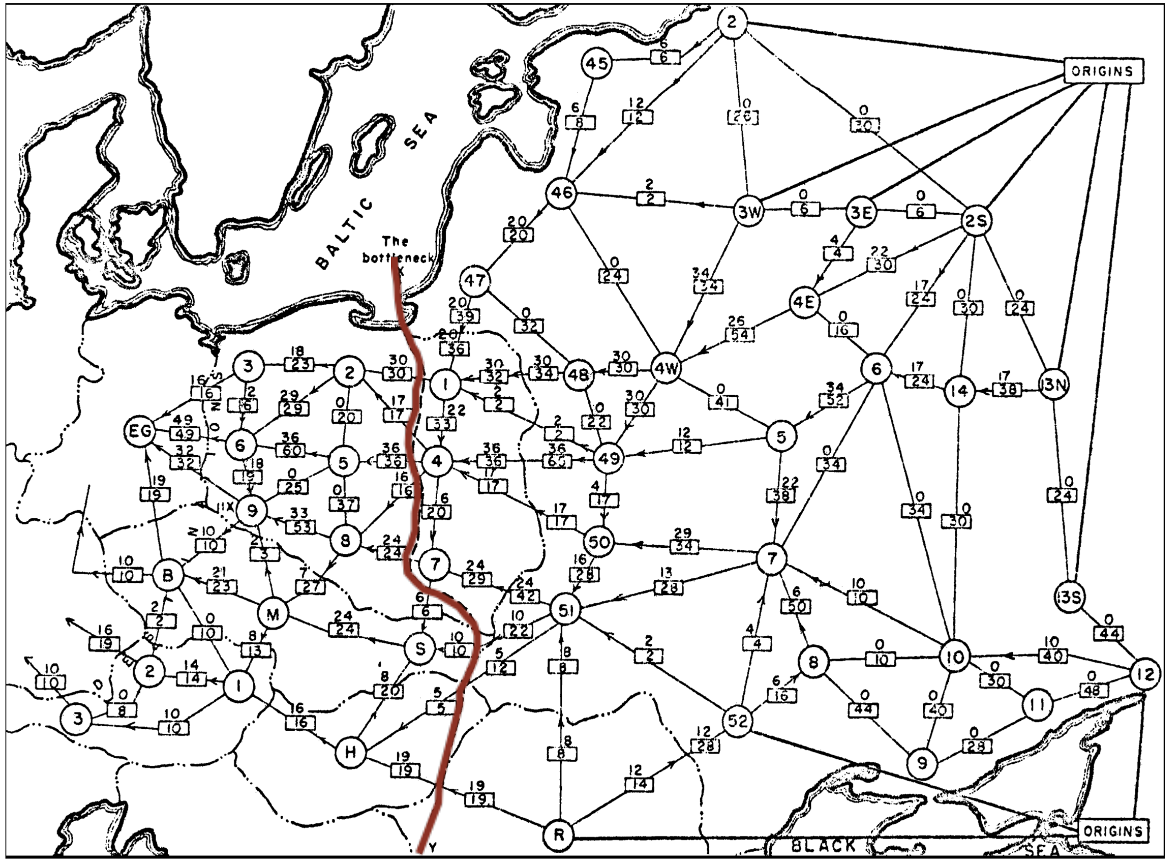
capacity



rail network connecting Soviet Union with Eastern European countries
(map declassified by Pentagon in 1999)

Minimum Cut Application (RAND 1950s)

- **"Free world" goal.** Cut supplies (if Cold War turns into real war).



rail network connecting Soviet Union with Eastern European countries
(map declassified by Pentagon in 1999)

Maximum Flow and Minimum Cut

- **Max-flow and min-cut problems.**
  - Beautiful mathematical duality.
  - Cornerstone problems in combinatorial optimization.
- **They are widely applicable models.**
  - Data mining, open-pit mining, bipartite matching, network reliability, baseball elimination, image segmentation, network connectivity, Markov random fields, distributed computing, security of statistical data, egalitarian stable matching, network intrusion detection, multi-camera scene reconstruction, sensor placement for homeland security, etc.

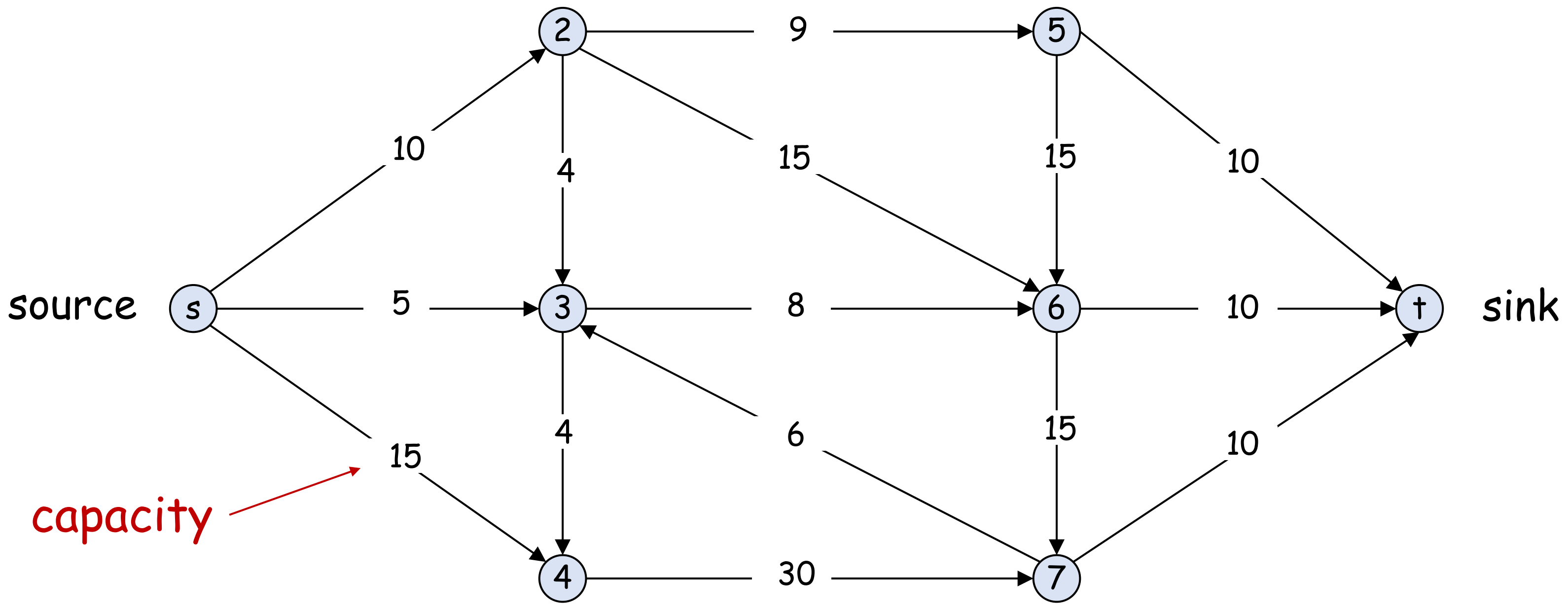
we will learn some of the applications in next section

1. Max Flow and Min Cut

Flow Network

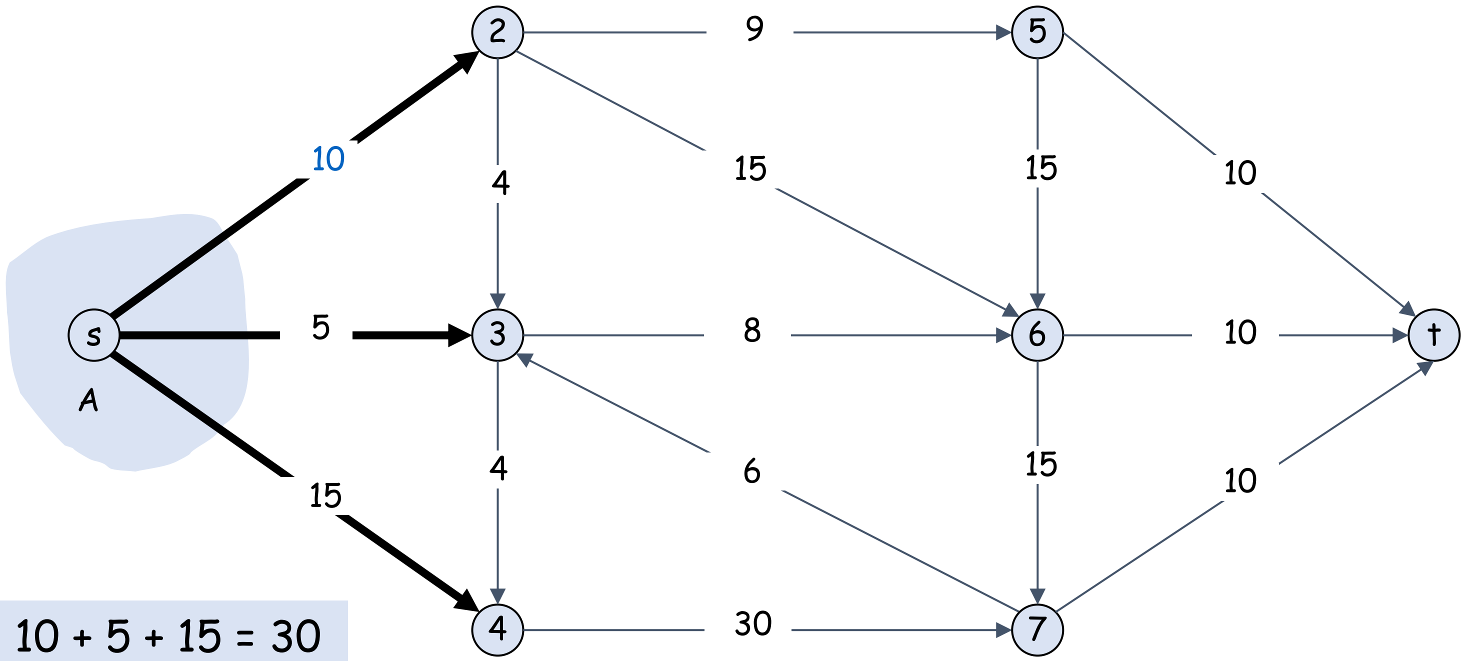
- A flow network is a tuple G = (V, E, s, t, c).
  - **Intuition:** material flowing through a transportation network, originating from source and sent to sink.
  - Digraph G = (V, E) with source s and sink t, no parallel edges.
  - Capacity c(e) ≥ 0 for each edge e.

assume all nodes are reachable from s



Minimum-Cut Problem

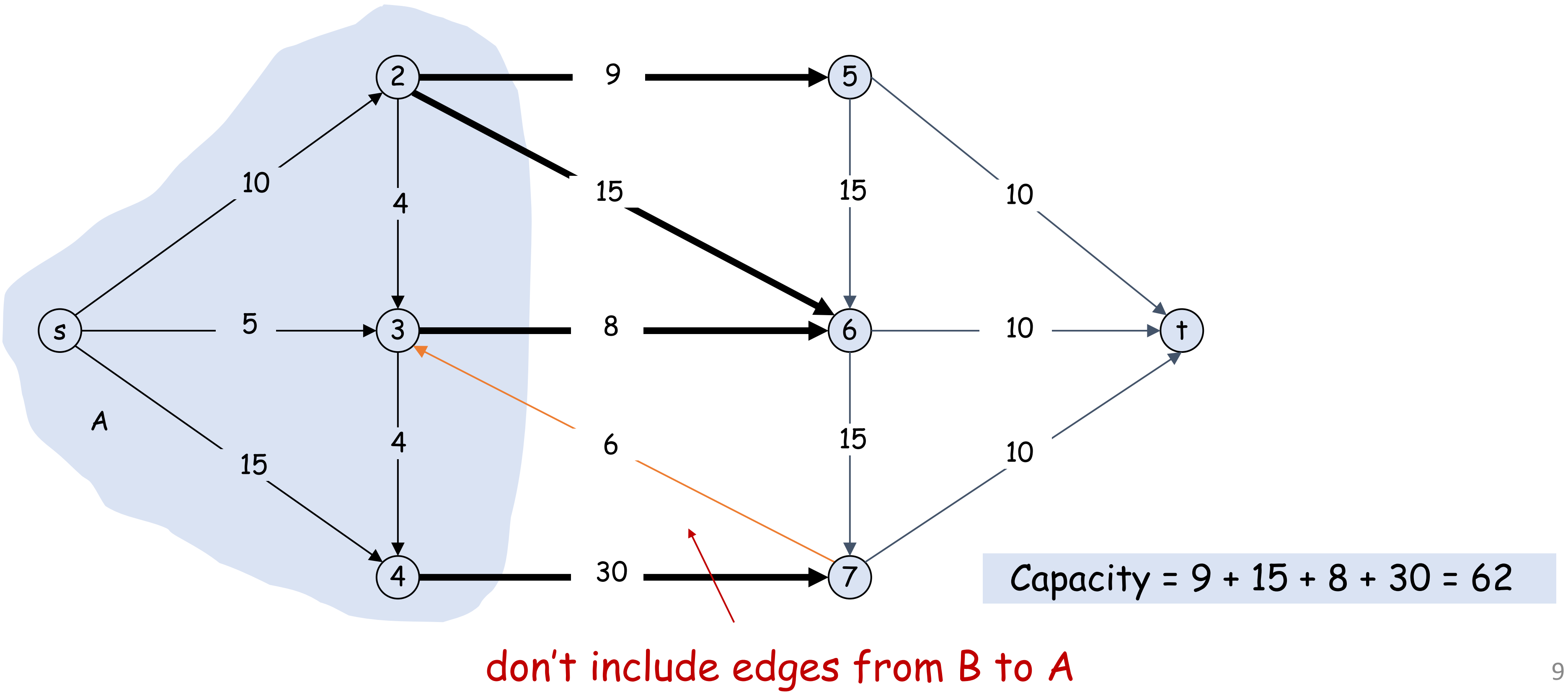
- **Def.** An *st*-cut (or cut) is a partition (A, B) of V with s ∈ A and t ∈ B.
- **Def.** The capacity of a cut (A, B) is $c(A, B) = \sum_{e \text{ out of } A} c_e$



Capacity = 10 + 5 + 15 = 30

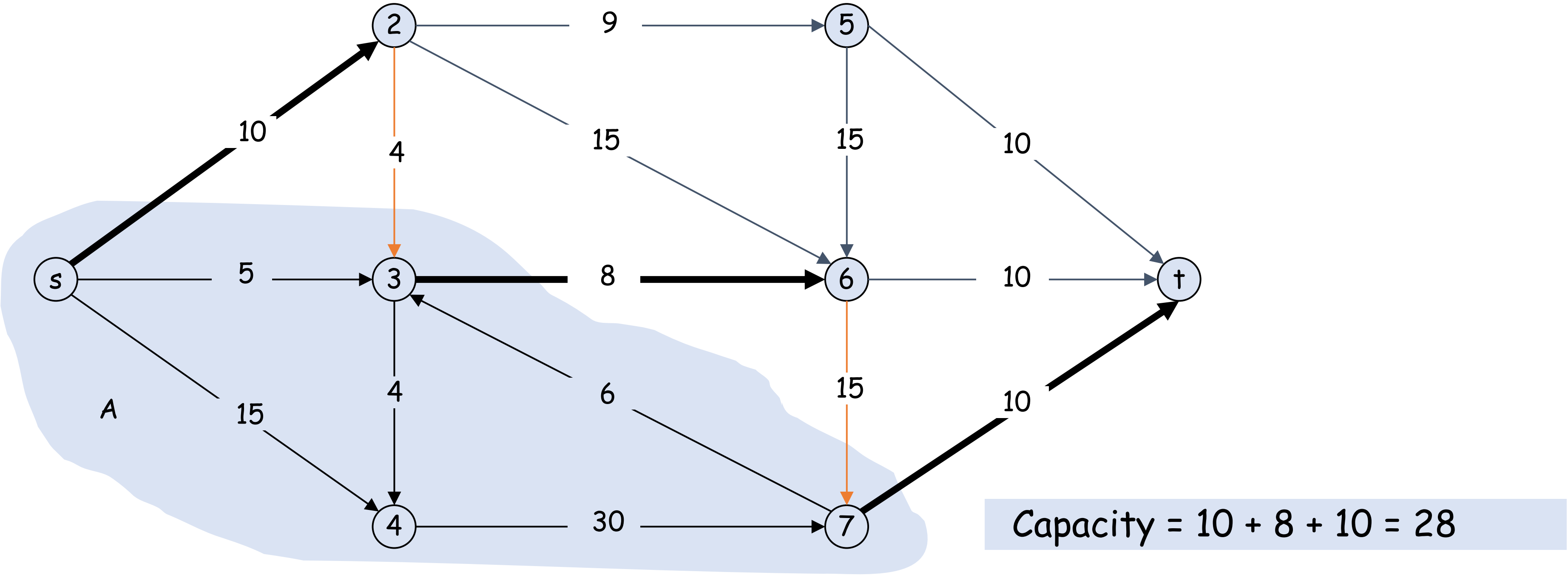
Minimum-Cut Problem

- **Def.** An *st*-cut (or cut) is a partition (A, B) of V with s ∈ A and t ∈ B.
- **Def.** The capacity of a cut (A, B) is $c(A,B) = \sum_{e \text{ out of } A} c_e$



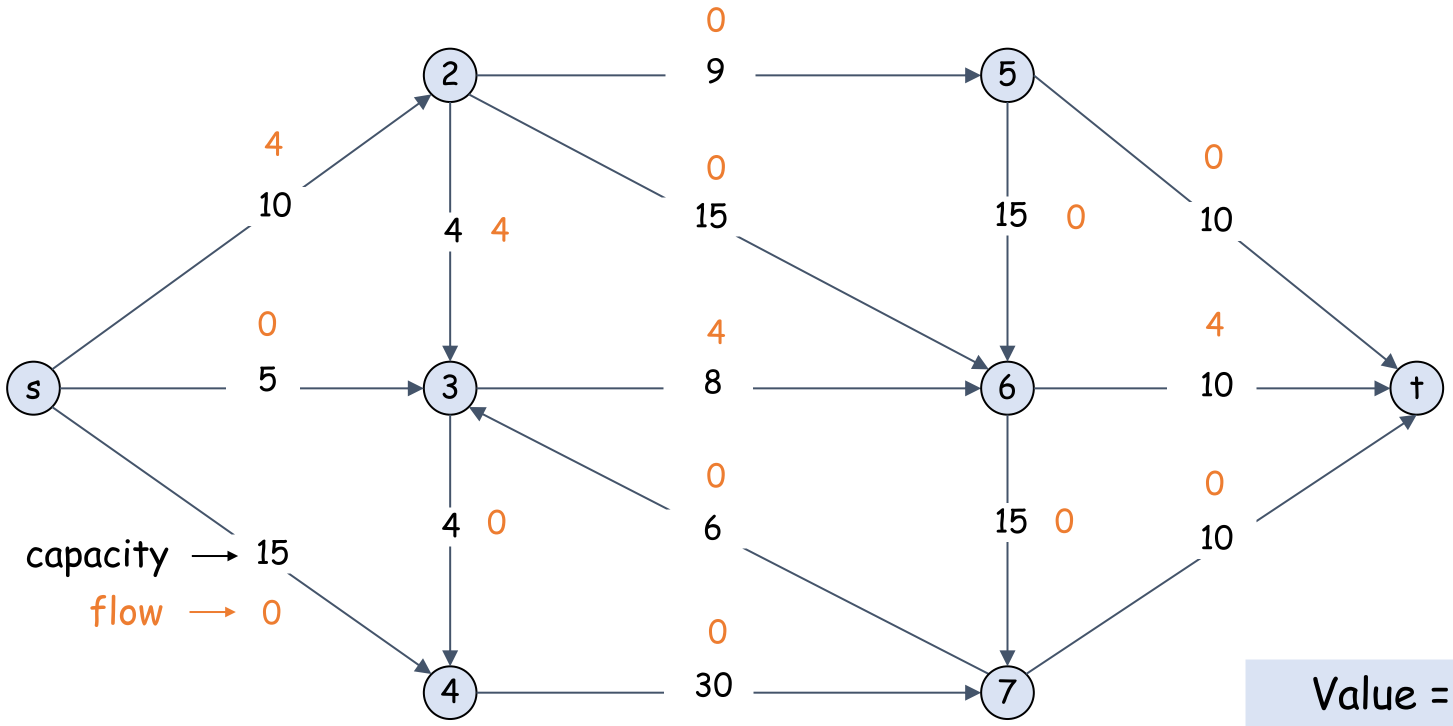
Minimum-Cut Problem

- **Def.** An *st*-cut (or cut) is a partition (A, B) of V with s ∈ A and t ∈ B.
- **Def.** The capacity of a cut (A, B) is $c(A,B) = \sum_{e \text{ out of } A} c_e$
- **Min-cut problem.** Find a cut of minimum capacity.



Maximum-Flow Problem

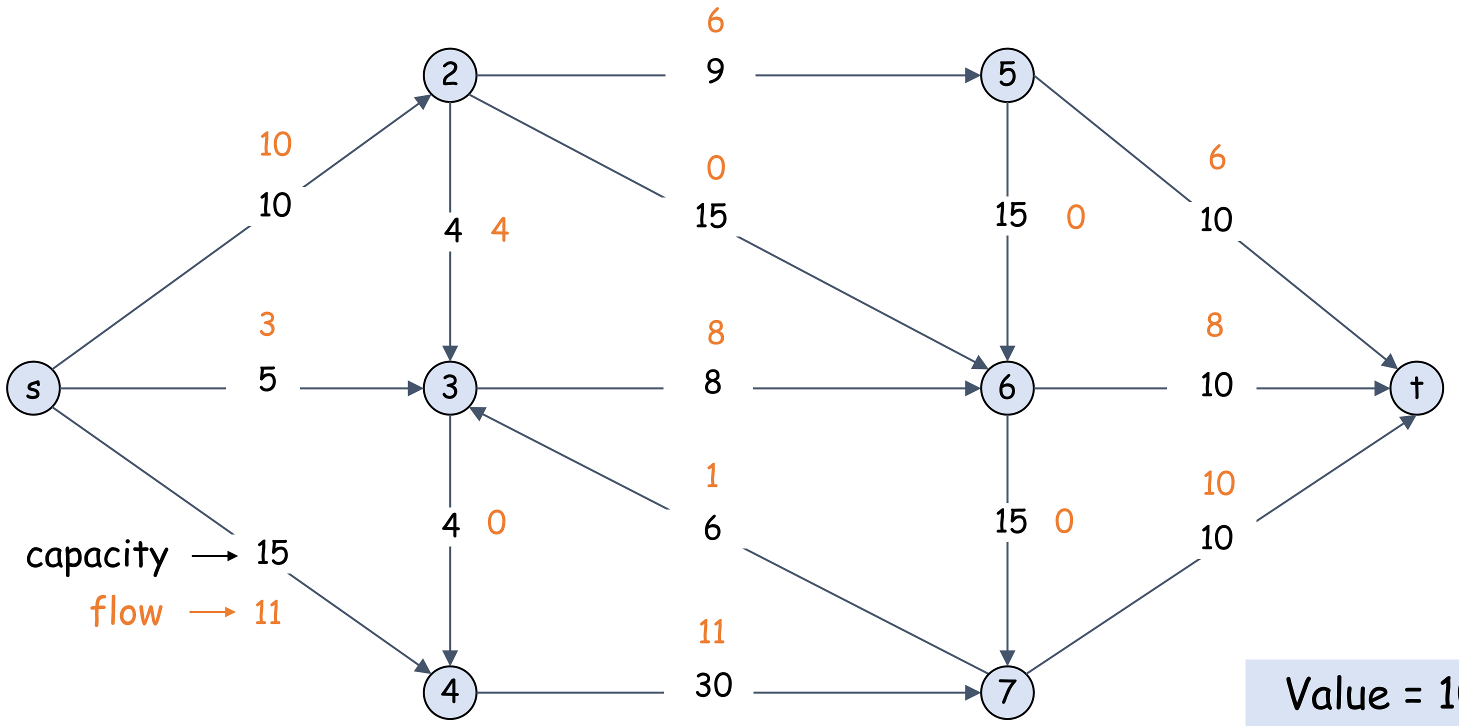
- **Def.** An *st*-flow (or flow) f is a function that satisfies
  - For each $e \in E: 0 \leq f(e) \leq c_e$ [capacity]
  - For each $v \in V - \{s, t\}$: $\sum_{e \text{ into } v} f(e) = \sum_{e \text{ out of } v} f(e)$ [flow conservation]
- **Def.** The value of a flow f is $v(f) = \sum_{e \text{ out of } s} f(e)$



Value = 4 + 0 + 0 = 4

Maximum-Flow Problem

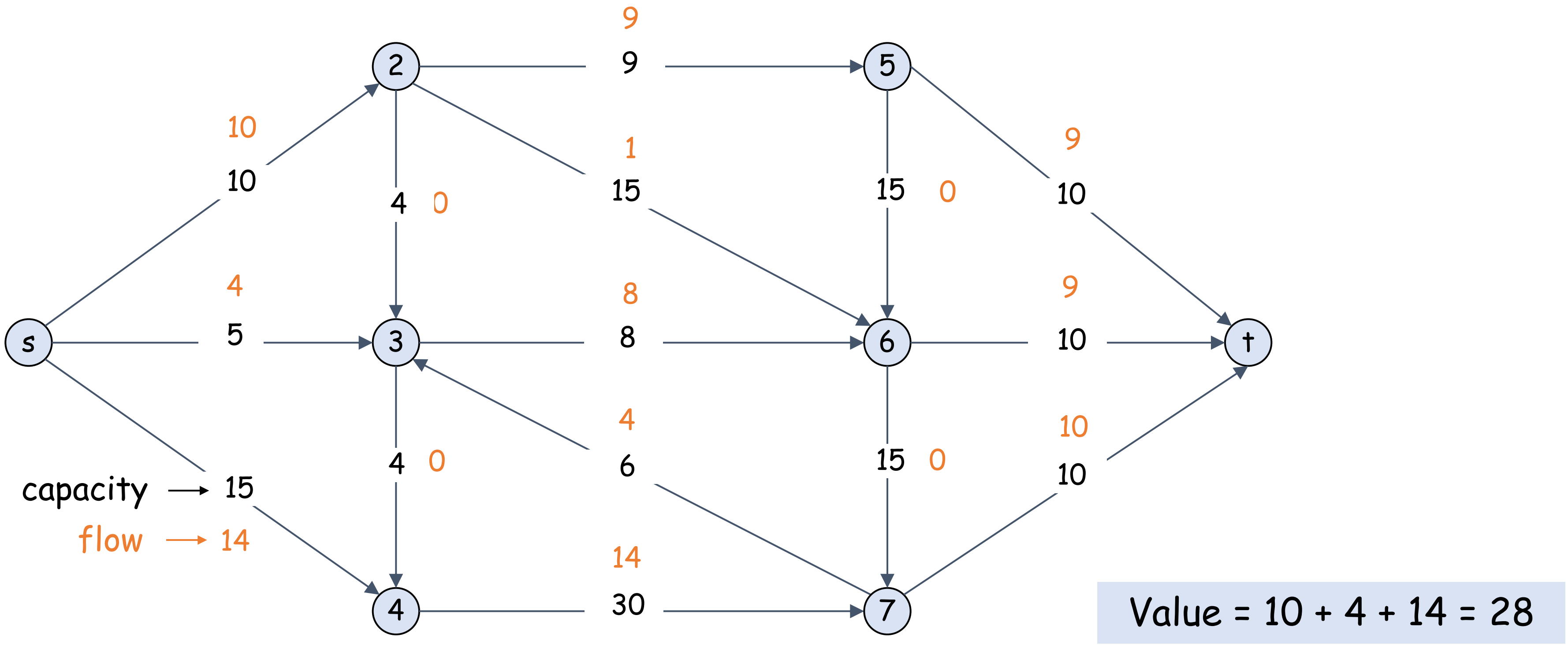
- **Def.** An *st*-flow (or flow) f is a function that satisfies
  - For each $e \in E: 0 \le f(e) \le c_e$ [capacity]
  - For each $v \in V - \{s, t\}$: $\sum_{e\ into\ v} f(e) = \sum_{e\ out\ of\ v} f(e)$ [flow conservation]
- **Def.** The value of a flow f is $v(f) = \sum_{e\ out\ of\ s} f(e)$



Value = 10 + 3 + 11 = 24

Maximum-Flow Problem

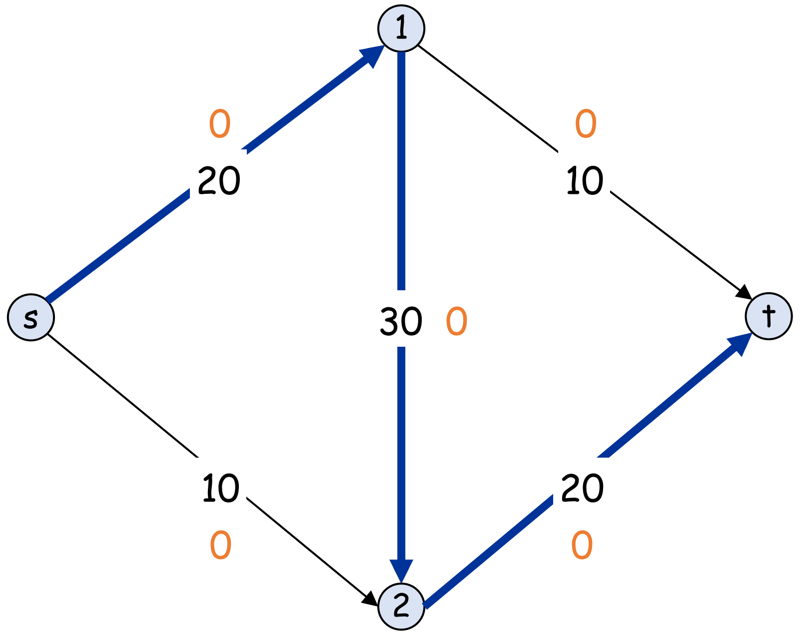
- **Def.** An *st*-flow (or flow) f is a function that satisfies
- **Def.** The value of a flow f is $v(f) = \sum_{e \text{ out of } s} f(e)$
- **Max-flow problem.** Find a flow of maximum value.



2. Ford-Fulkerson Algorithm

Toward a Max-Flow Algorithm

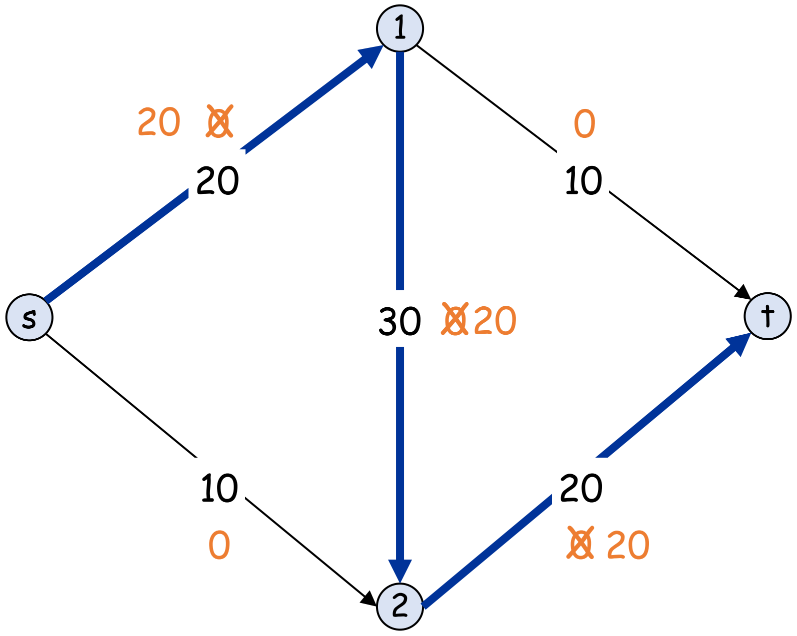
- **Greedy algorithm.**
  - Start with $f(e) = 0$ for all edges $e \in E$.
  - Find an s-t path P where each edge has $f(e) < c(e)$.
  - Augment flow along path P.
  - Repeat until you get stuck.



Flow value = 0

Toward a Max-Flow Algorithm

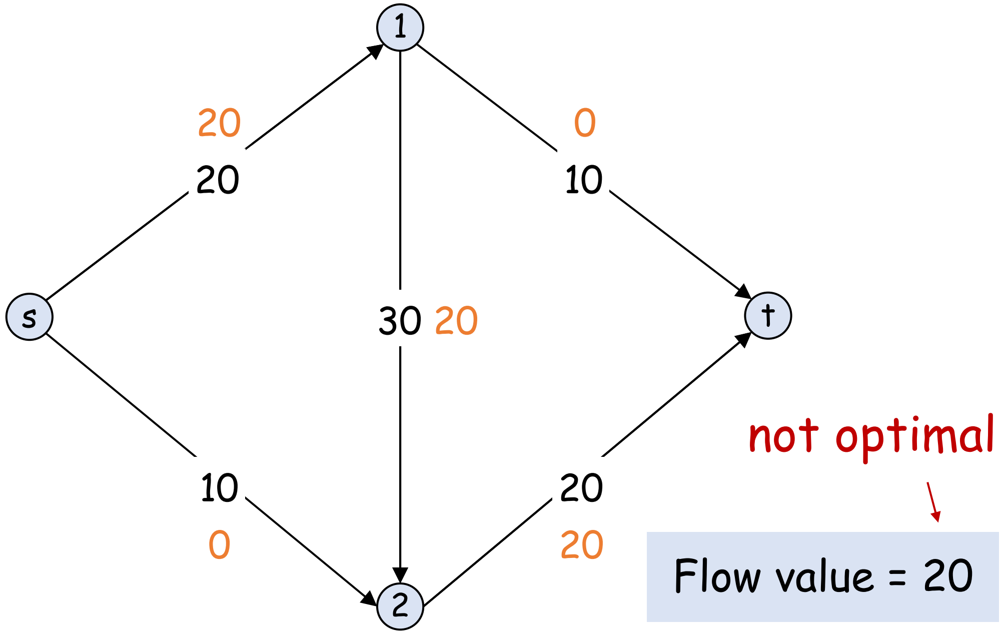
- **Greedy algorithm.**
  - Start with f(e) = 0 for all edges e ∈ E.
  - Find an s-t path P where each edge has f(e) < c(e).
  - Augment flow along path P.
  - Repeat until you get stuck.

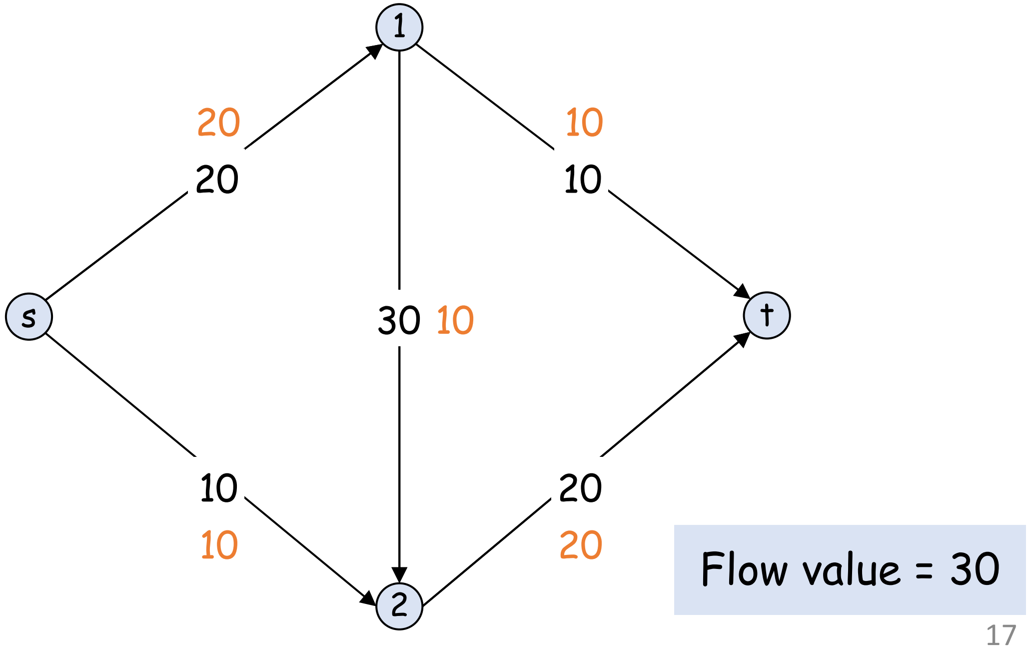


Flow value = 20

Toward a Max-Flow Algorithm

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

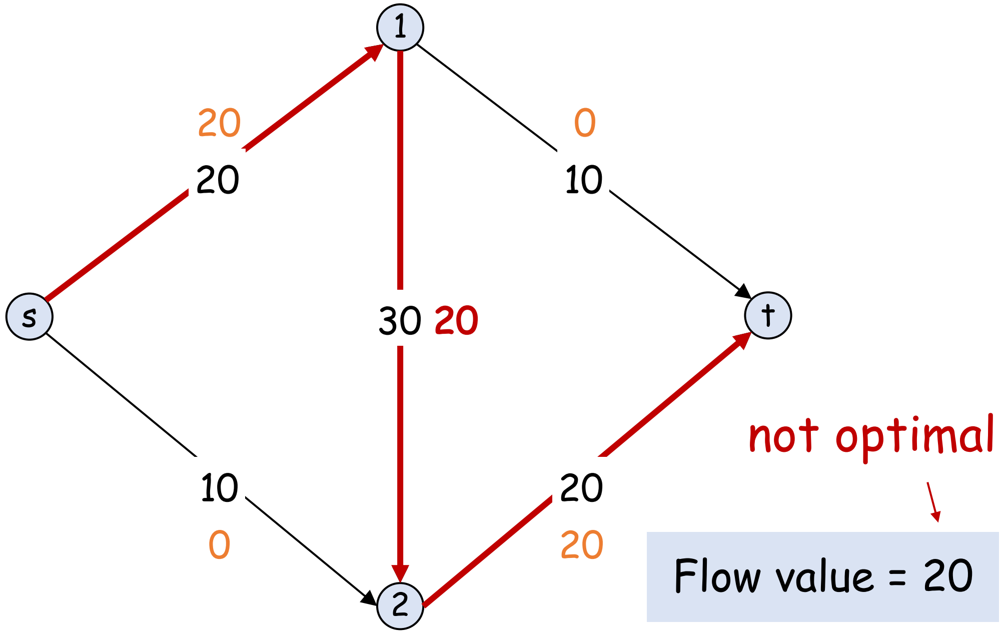


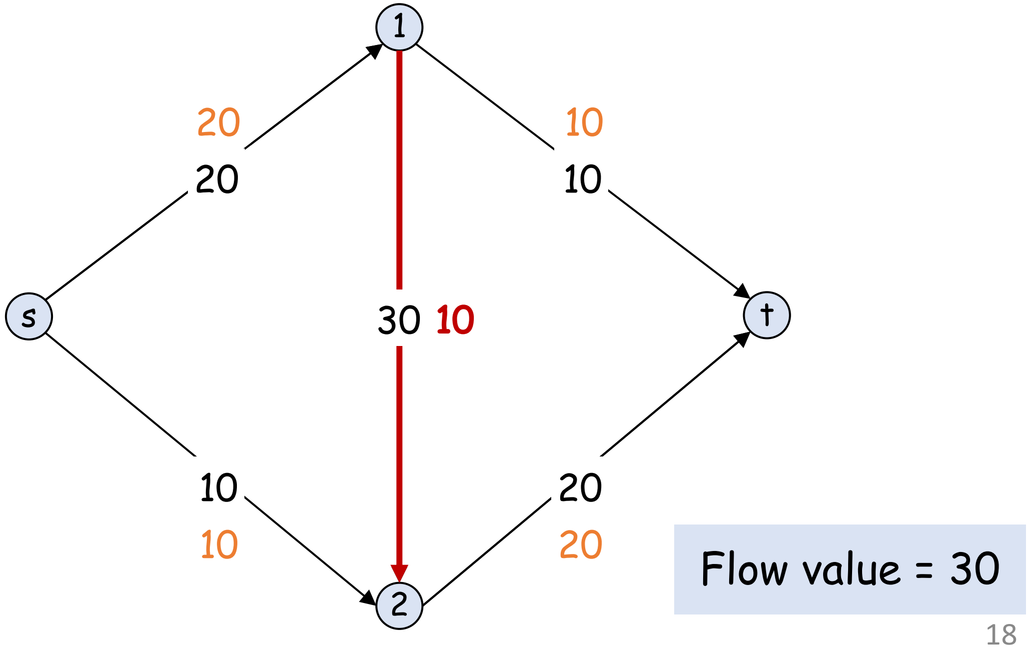


Toward a Max-Flow Algorithm

- **Q.** Why does the greedy algorithm fail?
- **A.** Once flow on an edge is increased, it never decreases.

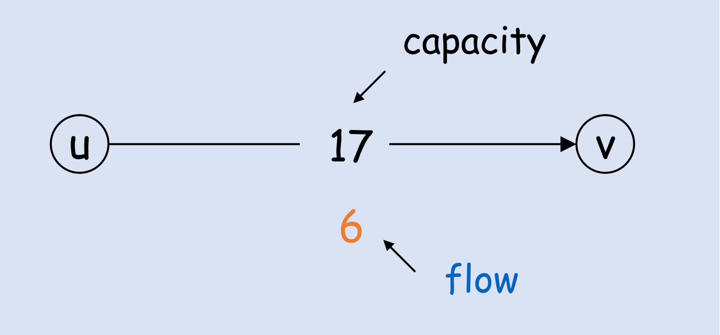
- **Bottom line.** Need some mechanism to "undo" a bad decision.

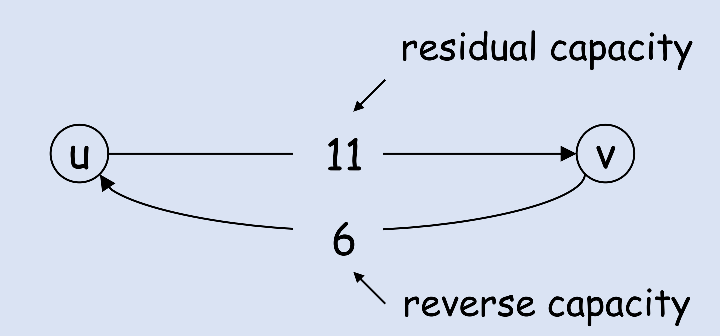




Residual Network

- **Original edge:** $e = (u, v) \in E$.
  - Flow $f(e)$, capacity $c(e)$
- **Reverse edge:** $e^R = (v, u)$.
  - "Undo" flow sent
- **Residual capacity:** c_f
  - Original edge: $c_f(e) = c(e) - f(e)$
  - Reverse edge: $c_f(e^R) = f(e)$
- **Residual network:** $G_f = (V, E_f, s, t, c_f)$.
  - $E_f = \{e: f(e) < c(e)\} \cup \{e^R: f(e) > 0\}$: residual edges with positive residual capacity





flow on a reverse edge negates flow on corresponding original forward edge

- **Key property.** f' is a flow in G_f iff $f + f'$ is a flow in G.

Augmenting Path

- **Def.** An augmenting path is a simple s-t path in the residual network G_f.
- **Def.** The bottleneck capacity of an augmenting path P is the minimum residual capacity of any edge in P.
- **Key property.** Let f be a flow and let P be an augmenting path in G_f. Then, after calling $f' \leftarrow$ Augment(f, c, P), the resulting f' is a flow and $val(f') = val(f) + bottleneck(G_f, P)$.

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

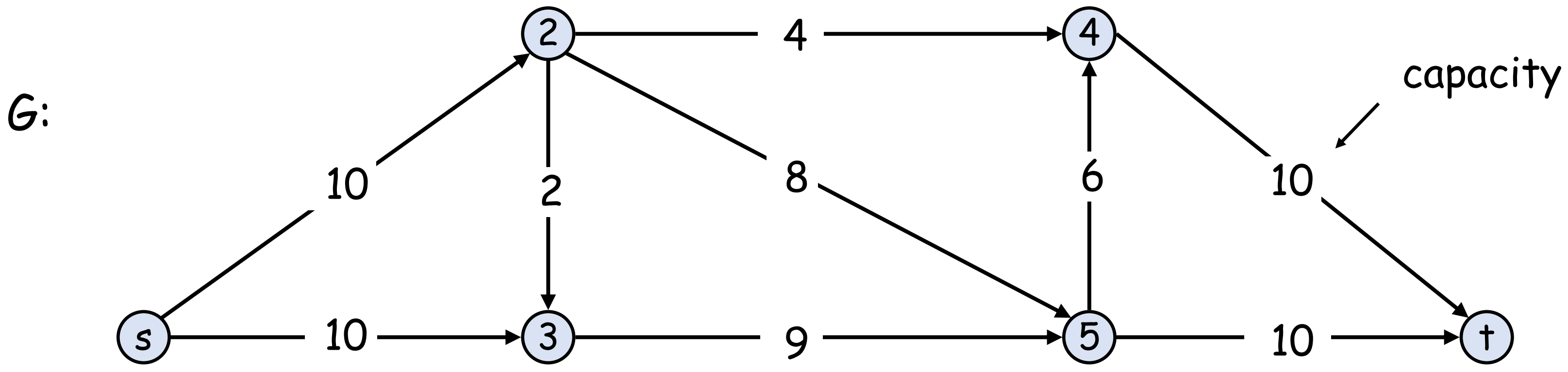
Ford-Fulkerson Algorithm

- **Ford-Fulkerson (FF) algorithm.**
  - Start with $f(e) = 0$ for each edge $e \in E$.
  - Find an s-t path P in the residual network G_f .
  - Augment flow along path P.
  - Repeat until you get stuck.

```
Ford-Fulkerson(G, s, t, c) {
   foreach e ∈ E: f(e) = 0
   Gf = residual network of G with respect to flow f

   while (there exists an augmenting path P) {
      f = Augment(f, c, P)
      update Gf
   }
   return f
}
```

Ford-Fulkerson Algorithm Demo

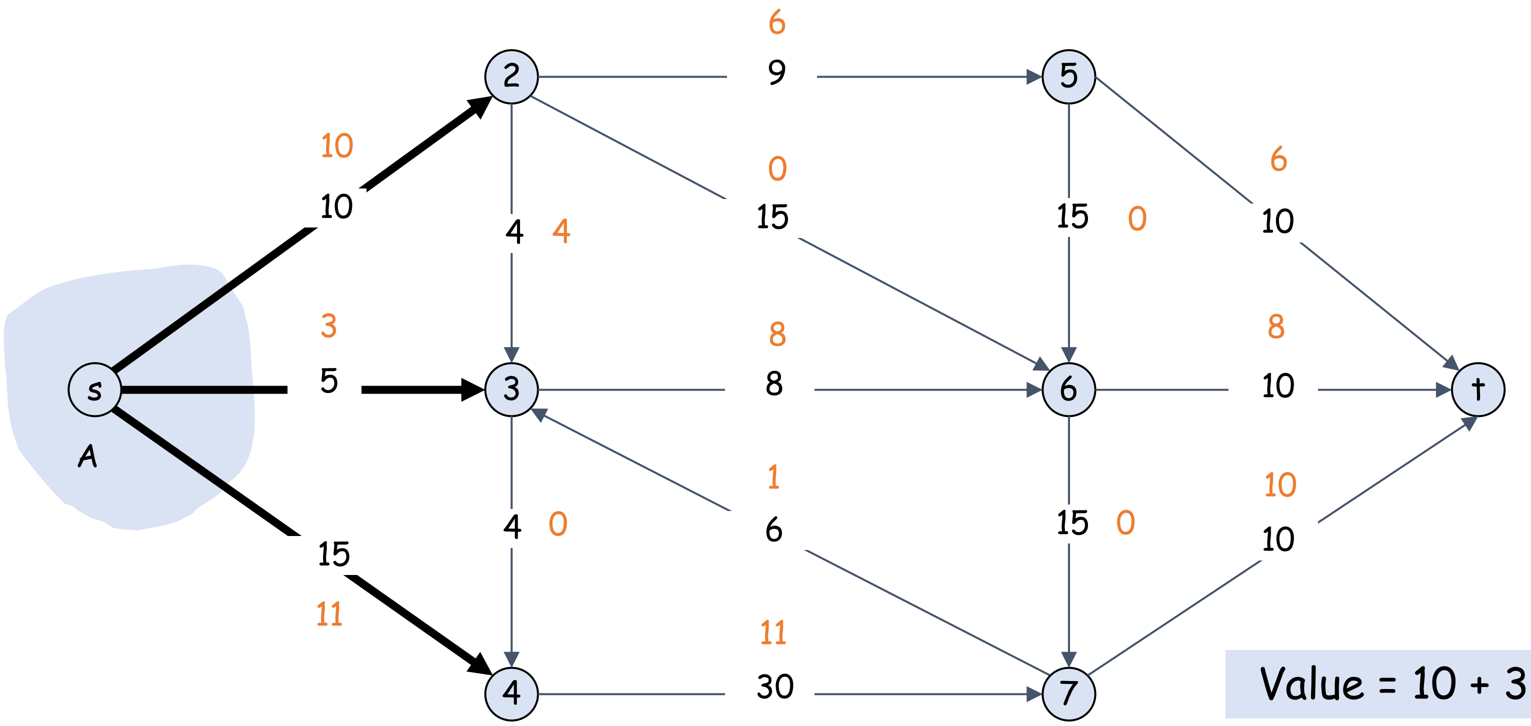


3. Max-Flow Min-Cut Theorem

Relationship between Flows and Cuts

- **Flow value lemma.** Let *f* be any flow, and let (A, B) be any cut. Then, the value of the flow *f* equals the net flow across the cut (A, B).

$$v(f) = f^{out}(A) - f^{in}(A)$$

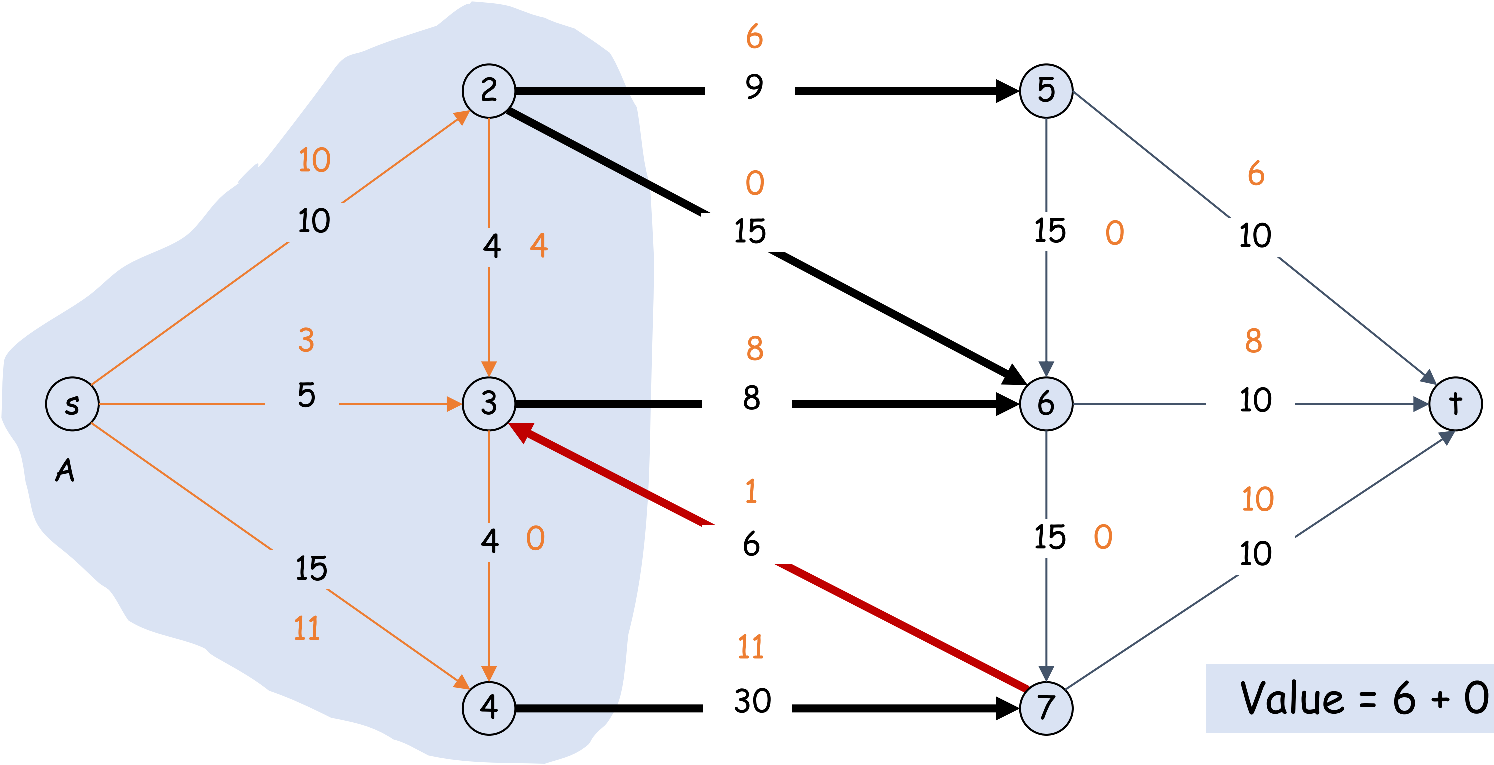


Value = 10 + 3 + 11 = 24

Relationship between Flows and Cuts

- **Flow value lemma.** Let *f* be any flow, and let (A, B) be any cut. Then, the value of the flow *f* equals the net flow across the cut (A, B).

$$v(f) = f^{out}(A) - f^{in}(A)$$

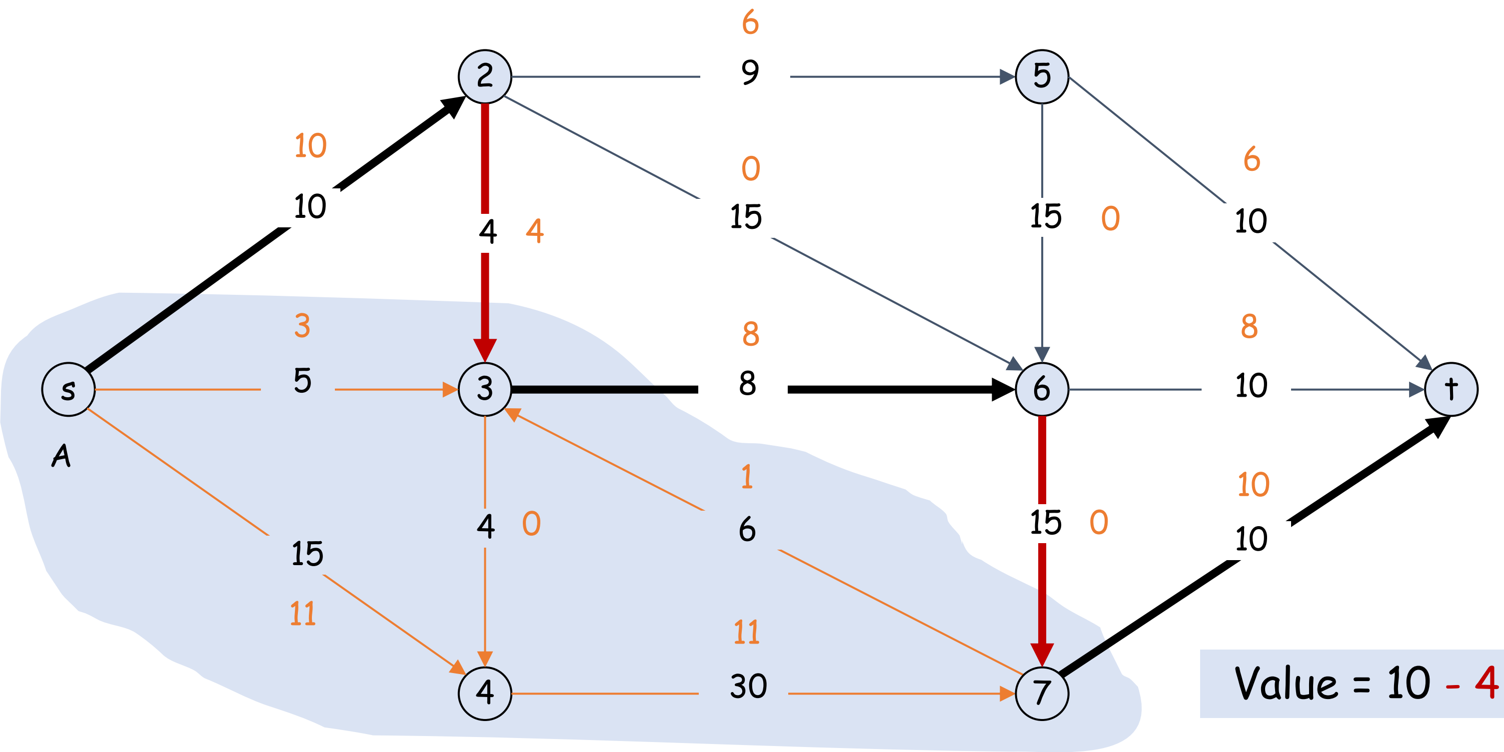


Value = 6 + 0 + 8 - 1 + 11 = 24

Relationship between Flows and Cuts

- **Flow value lemma.** Let f be any flow, and let (A, B) be any cut. Then, the value of the flow f equals the net flow across the cut (A, B).

$$v(f) = f^{out}(A) - f^{in}(A)$$



Value = 10 - 4 + 8 - 0 + 10 = 24

Relationship between Flows and Cuts

- **Flow value lemma.** Let *f* be any flow, and let (A, B) be any cut. Then, the value of the flow *f* equals the net flow across the cut (A, B).

$$v(f) = f^{out}(A) - f^{in}(A) \qquad v(f) = f^{in}(B) - f^{out}(B)$$

- **Pf.**

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

$f^{in}(s) = 0$

$$= \sum_{v \in A} (f^{out}(v) - f^{in}(v))$$

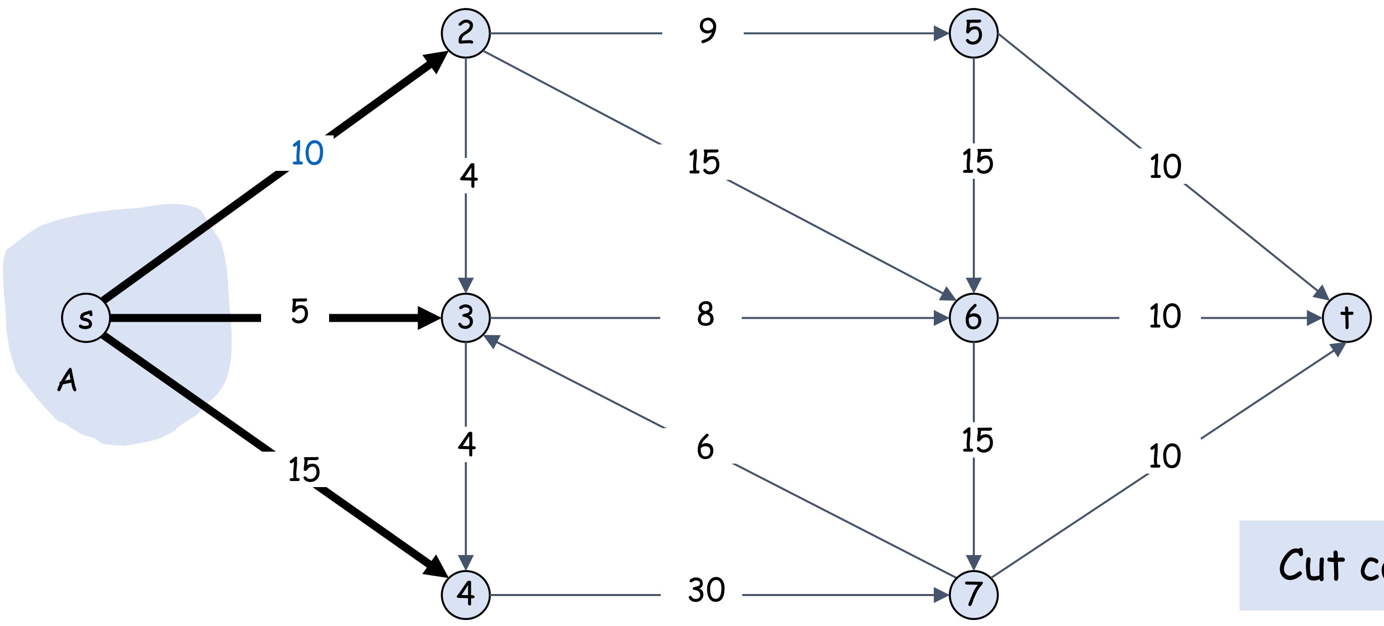
$f^{out}(v) - f^{in}(v) = 0$

$$= \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ into } A} f(e) = f^{out}(A) - f^{in}(A)$$

Relationship between Flows and Cuts

- **Weak duality.** Let *f* be any flow, and let (A, B) be any cut. Then the value of the flow *f* is at most the capacity of the cut: $v(f) \le c(A, B)$.

$$c(A,B) = \sum_{e \text{ out of } A} c_e$$



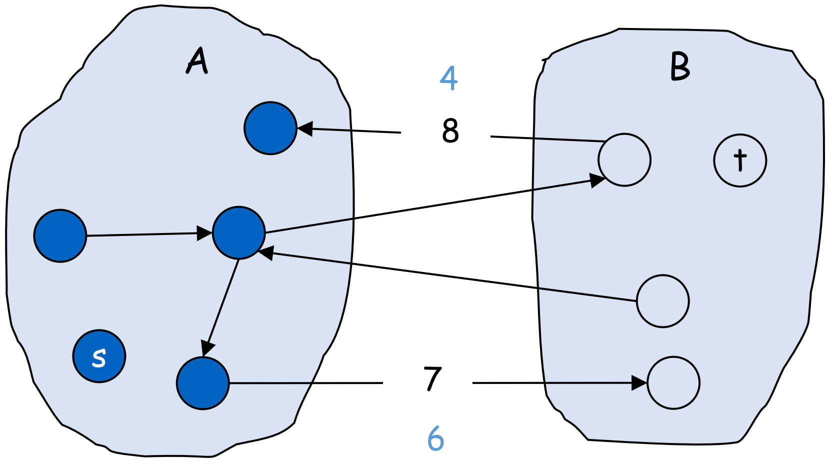
Cut capacity = 30 $\Rightarrow$ Flow value $\le$ 30

Relationship between Flows and Cuts

- **Weak duality.** Let *f* be any flow, and let (A, B) be any cut. Then the value of the flow *f* is at most the capacity of the cut: $v(f) \leq c(A, B)$.
- **Pf.**

flow value lemma

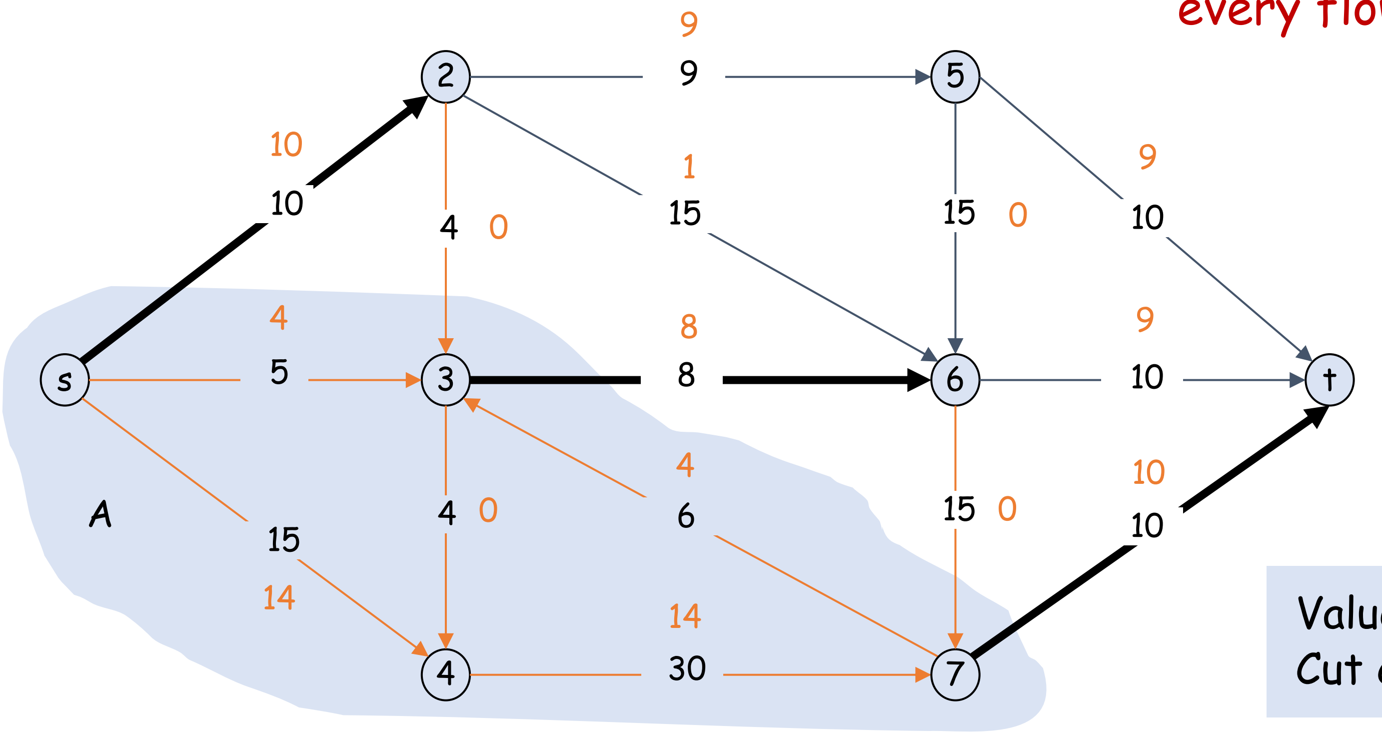
$$
\begin{aligned}
v(f) &= f^{\text{out}}(A) - f^{\text{in}}(A) \\
&\leq f^{\text{out}}(A) \\
&= \sum_{e \text{ out of } A} f(e) \\
&\leq \sum_{e \text{ out of } A} c_e \\
&= c(A, B).
\end{aligned}
$$



Certificate of Optimality

- **Corollary.** Let f be any flow, and let (A, B) be any cut. If v(*f*) = c(A, B), then *f* is a max flow and (A, B) is a min cut.

every flow is upper bounded by every cut!



Value of flow = 28
Cut capacity = 28 $\Rightarrow$ Flow value $\leq$ 28

Max-Flow Min-Cut Theorem

- **Max-flow min-cut theorem.** [Ford-Fulkerson 1956] Value of a max flow is equal to capacity of a min cut.
- **Augmenting path theorem.** Flow f is a max flow iff no augmenting paths.

- **Pf.** We prove both by showing the following are equivalent:
  - (i) There exists a cut (A, B) such that $v(f) = c(A, B)$.
  - (ii) f is a max flow.
  - (iii) There is no augmenting path with respect to f.

  (i) $\Rightarrow$ (ii): This is the weak duality corollary.

  (ii) $\Rightarrow$ (iii): We prove the contrapositive.

  - ➢ Let f be a flow. If there exists an augmenting path, then we can improve flow f by sending flow along this path. Then, f is not a max flow. Contradiction!

Max-Flow Min-Cut Theorem

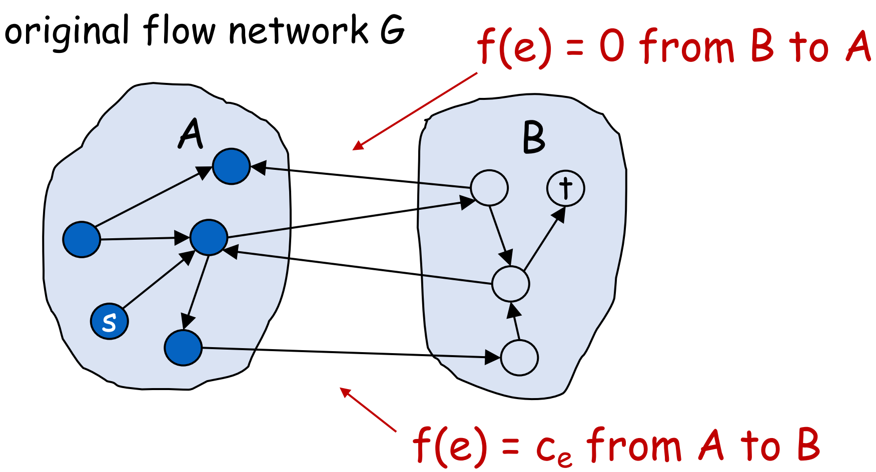
- **Pf continued.**
  - (i) There exists a cut (A, B) such that v(*f*) = c(A, B).
  - (iii) There is no augmenting path with respect to *f*.

(iii) $\Rightarrow$ (i):

- Let *f* be a flow with no augmenting paths.
- Let *A* = set of nodes reachable from *s* in residual network G_f.
- By definition of *A*: $s \in A$. By definition of flow *f*: $t \notin A$.

$$
\begin{aligned}
val(f) &= \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e) \\
&= \sum_{e \text{ out of } A} c(e) - 0 \\
&= cap(A, B) \quad \blacksquare
\end{aligned}
$$

flow value lemma



Max-Flow Min-Cut Theorem

- **Pf continued.**
  - (i) There exists a cut (A, B) such that v(*f*) = c(A, B).
  - (iii) There is no augmenting path with respect to *f*.

(iii) $\Rightarrow$ (i):

given any max flow f (then no augmenting path)
can find a min cut in O(m) time

- Let *f* be a flow with no augmenting paths.
- Let *A* = set of nodes reachable from *s* in residual network G_f.
- By definition of *A*: *s* ∈ *A*. By definition of flow *f*: *t* ∉ *A*.

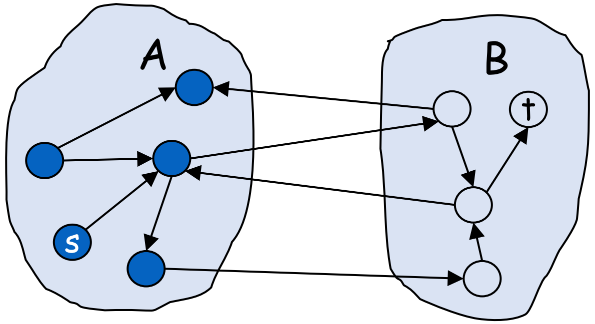
$$val(f) = \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e)$$

flow value lemma

$$= \sum_{e \text{ out of } A} c(e) - 0$$

$$= cap(A, B) \quad \blacksquare$$

original flow network G



4. Capacity-Scaling Algorithm

Ford-Fulkerson Algorithm: Analysis

- **Assumption.** Every edge capacity c_e is an integer between 1 and C.
- **Integrality invariant.** Throughout FF, every edge flow $f(e)$ and residual capacity $c_f(e)$ are integers.

- **Theorem.** FF terminates after at most $val(f^*) \leq nC$ augmenting paths, where f^* is a max flow.
- **Pf.** Each augmentation increases the value of the flow by at least 1. ▪

Ford-Fulkerson Algorithm: Analysis

- **Assumption.** Every edge capacity c_e is an integer between 1 and C.
- **Integrality invariant.** Throughout FF, every edge flow $f(e)$ and residual capacity $c_f(e)$ are integers.
- **Theorem.** FF terminates after at most $val(f^*) \leq nC$ augmenting paths, where f^* is a max flow.

- **Corollary.** The running time of Ford-Fulkerson is $O(mnC)$.
- **Pf.** Can use either BFS or DFS to find an augmenting path in $O(m)$ time. ▪

Ford-Fulkerson Algorithm: Analysis

- **Assumption.** Every edge capacity c_e is an integer between 1 and C.
- **Integrality invariant.** Throughout FF, every edge flow $f(e)$ and residual capacity $c_f(e)$ are integers.
- **Theorem.** FF terminates after at most $val(f^*) \le nC$ augmenting paths, where f^* is a max flow.
- **Corollary.** The running time of Ford-Fulkerson is $O(mnC)$.

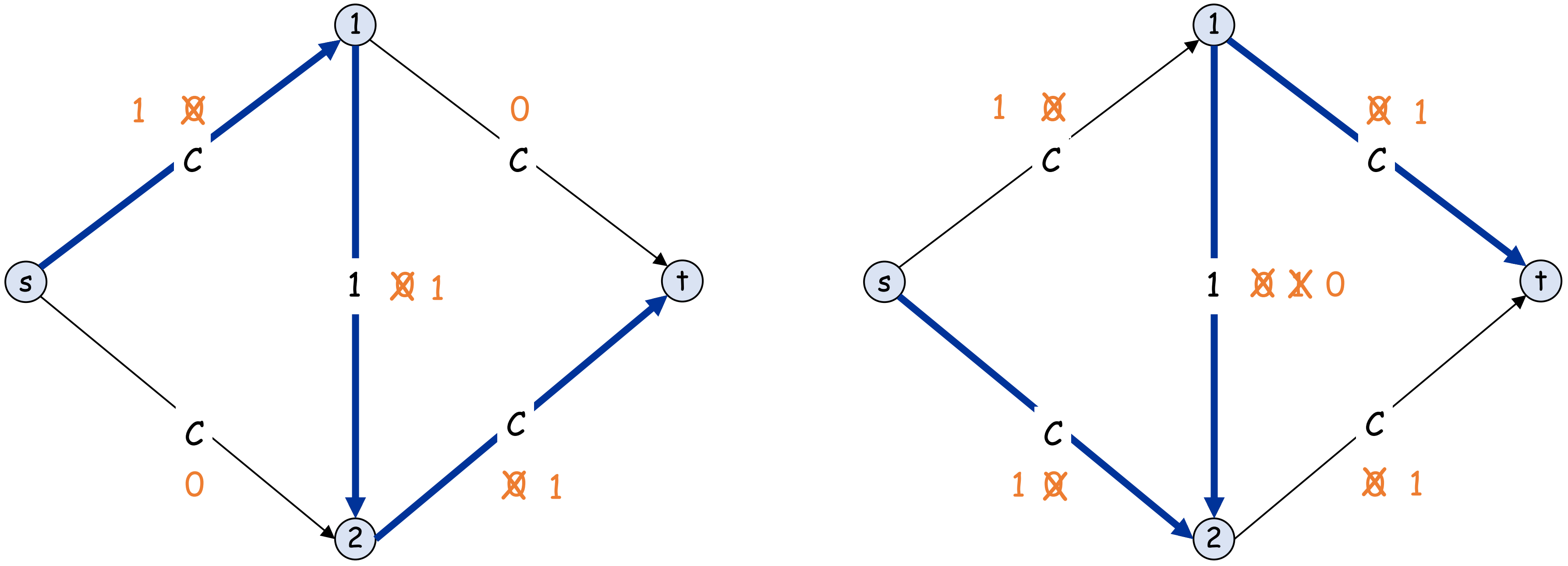
- **Integrality theorem.** There exists an integral max flow f^*.
- **Pf.** Since FF always terminates when capacities are integral, theorem follows from integrality invariant (and augmenting path theorem). ▪

Ford-Fulkerson: Exponential Example

- **Q.** Is generic Ford-Fulkerson algorithm polynomial in input size?

m, n, log C

- **A.** No. If max capacity is C, then algorithm can take $2C$ iterations.



Choosing Good Augmenting Paths

- **Note.** If capacities can be irrational, FF may not terminate or converge!

capacities are rational in practice
but FF could run in exponential time

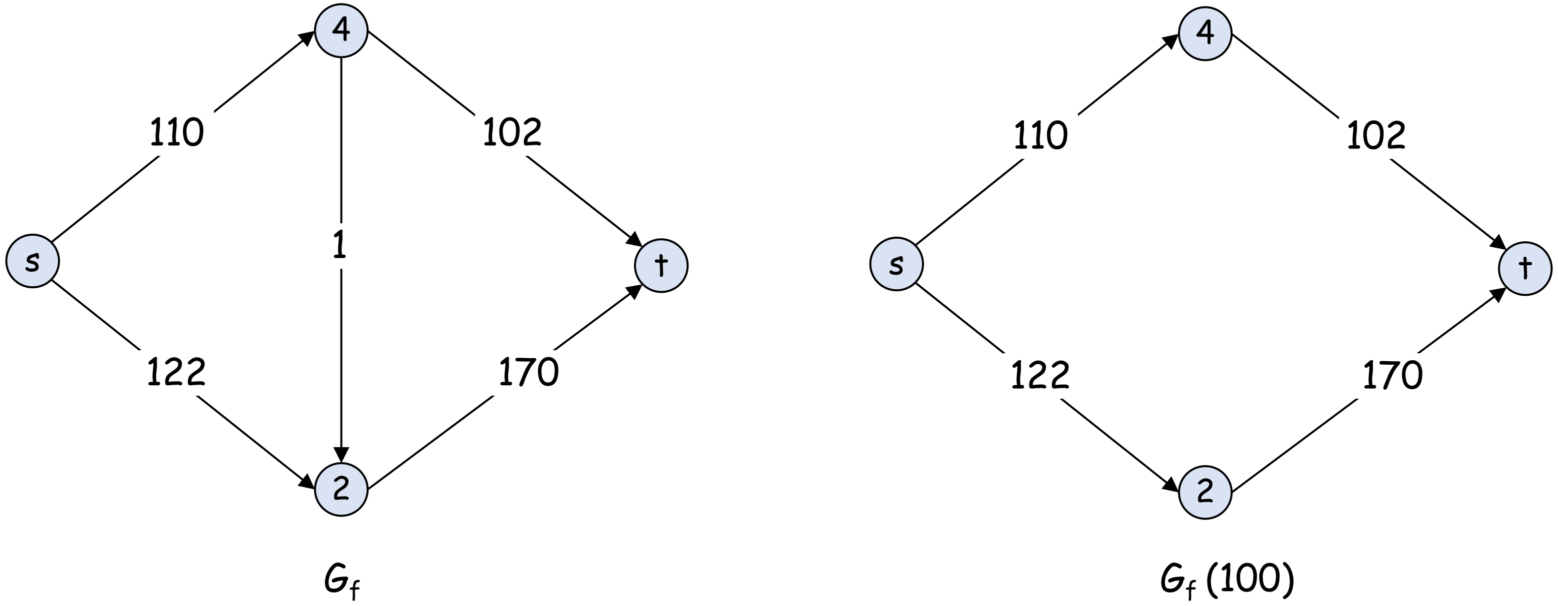
- **Use care when selecting augmenting paths.**
  - Some choices lead to exponential algorithms.
  - Clever choices lead to polynomial algorithms.
- **Goal.** Choose augmenting paths so that:
  - Can find augmenting paths efficiently.
  - Few iterations.
- **Choose augmenting paths with:**
  - Max bottleneck capacity ("fattest"). ← how to find?
  - Sufficiently large bottleneck capacity. ← coming next
  - Fewest number of edges. [Edmonds-Karp 1972, Dinitz 1970]

next section

Capacity-Scaling Algorithm

- **Overview.** Choosing augmenting paths with "large" bottleneck capacity.
  - Maintain scaling parameter Δ.
  - Let $G_f(\Delta)$ be the subnetwork of the residual network containing only those edges with capacity ≥ Δ.
  - Any augmenting path in $G_f(\Delta)$ has bottleneck capacity ≥ Δ.

not necessarily largest



Capacity-Scaling Algorithm

```
Capacity-Scaling(G, s, t, c) {
   foreach e ∈ E: f(e) = 0
   Δ = largest power of 2 ≤ C
   Gf = residual network with respect to flow f


   while (Δ ≥ 1) {
      Gf(Δ) = Δ-residual network of G with respect to flow f
      while (there exists an augmenting path P in Gf(Δ)) {
         f = Augment(f, c, P)
         update Gf(Δ)
      }
      Δ = Δ / 2
   }
   return f
}
```

Capacity-Scaling Algorithm: Correctness

- **Assumption.** All edge capacities are integers between 1 and C.
- **Integrality invariant.** Throughout the algorithm, every edge flow $f(e)$ and residual capacity $c_f\,(e)$ are integers.

- **Theorem.** If capacity-scaling algorithm terminates, then f is a max flow.
- **Pf.**
  - By integrality invariant, when $\Delta = 1 \Rightarrow G_f(\Delta) = G_f$.
  - Upon termination of $\Delta = 1$ phase, there are no augmenting paths. ▪

Capacity-Scaling Algorithm: Running Time

- **Lemma 1.** The outer while loop repeats $1 + \lfloor \log_2 C \rfloor$ times.
- **Pf.** Initially $C/2 < \Delta \le C$; Δ decreases by a factor of 2 each iteration. ▪

- **Lemma 2.** Let f be the flow at the end of a Δ-scaling phase. Then the value of the maximum flow $\le v(f) + m\,\Delta$. (Proof later.)

- **Lemma 3.** There are $\le 2m$ augmentations per scaling phase.
- **Pf.** Let f be the flow at the end of the previous scaling phase $\Delta' = 2\Delta$.
  - Lemma 2 $\Rightarrow$ maximum flow value $\le v(f) + m\,\Delta' = v(f) + m(2\Delta)$.
  - Each augmentation in a Δ-phase increases $v(f)$ by at least Δ. ▪

Capacity-Scaling Algorithm: Running Time

- **Lemma 1.** The outer while loop repeats $1 + \lfloor \log_2 C \rfloor$ times.
- **Lemma 2.** Let f be the flow at the end of a Δ-scaling phase. Then the value of the maximum flow $\leq v(f) + m\,\Delta$. (Proof on next slide.)
- **Lemma 3.** There are $\leq 2m$ augmentations per scaling phase.

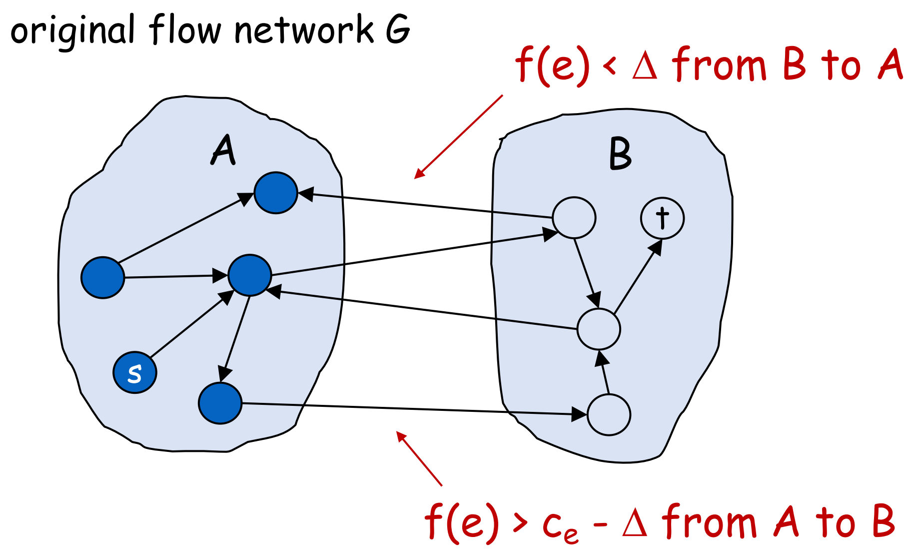
- **Theorem.** The capacity-scaling algorithm takes $O(m^2 \log C)$ time.
- **Pf.** Lemma 1 + Lemma 3 $\Rightarrow$ $O(m \log C)$ augmentations. Finding an augmenting path takes $O(m)$ time. ▪

Capacity-Scaling Algorithm: Running Time

- **Lemma 2.** Let f be the flow at the end of a Δ-scaling phase. Then the value of the maximum flow $\leq v(f) + m\,\Delta$.
- **Pf.** (similar to the proof of max-flow min-cut theorem)
  - We show that there exists a cut (A, B) such that $c(A, B) \leq v(f) + m\,\Delta$.
  - Choose A to be the set of nodes reachable from s in $G_f(\Delta)$.
  - By definition of A: $s \in A$. By definition of f : $t \notin A$.

$$
\begin{aligned}
val(f) &= \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e) \\
&\geq \sum_{e \text{ out of } A} (c(e) - \Delta) - \sum_{e \text{ in to } A} \Delta \\
&\geq \sum_{e \text{ out of } A} c(e) - \sum_{e \text{ out of } A} \Delta - \sum_{e \text{ in to } A} \Delta \\
&\geq cap(A, B) - m\Delta \quad \blacksquare
\end{aligned}
$$

flow value lemma



5. Edmonds-Karp Algorithm

Shortest Augmenting Path (Edmonds-Karp)

- **Q.** How to choose next augmenting path in Ford-Fulkerson?
- **A.** Pick one that uses the fewest edges.

can find via BFS

- **Edmonds-Karp algorithm:**

```
Edmonds-Karp(G, s, t, c) {
   foreach e ∈ E: f(e) = 0
   Gf = residual network of G with respect to flow f

   while (there exists an augmenting path P in Gf) {
      P = Breath-First-Search(Gf)
      f = Augment(f, c, P)
      update Gf
   }
   return f
}
```

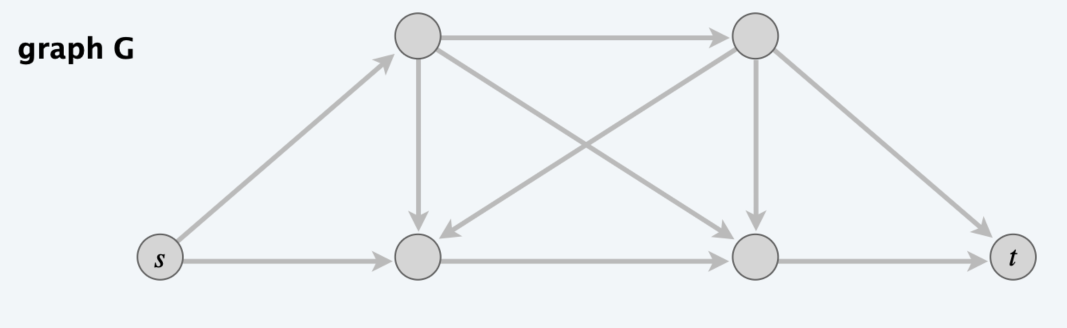
Edmonds-Karp Algorithm: Analysis Overview

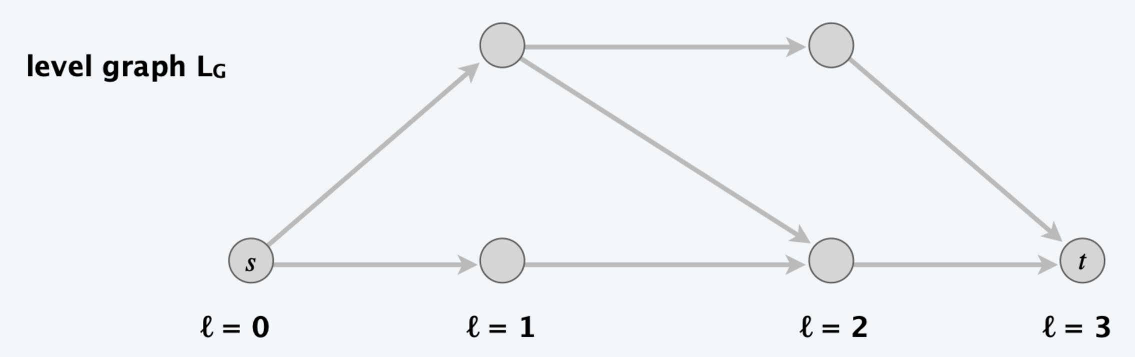
- **Lemma 1.** Length of a shortest augmenting path never decreases.
- **Lemma 2.** After at most *m* shortest-path augmentations, the length of a shortest augmenting path strictly increases.

- **Theorem.** The Edmonds-Karp algorithm takes $O(m^2n)$ time.
- **Pf.**
  - $O(m)$ time to find a shortest augmenting path via BFS.
  - There are $\leq mn$ augmentations.
    - ✓Augmenting paths are simple $\Rightarrow$ at most $n - 1$ different lengths
    - ✓Lemma 1 + Lemma 2 $\Rightarrow$ at most *m* augmenting paths for each length ▪

Edmonds-Karp Algorithm: Analysis

- **Def.** Given a digraph $G = (V, E)$ with source s, its level graph is defined by:
  - ➢ $\ell(v)$ = number of edges in shortest s-v path.
  - ➢ $L_G = (V, E_G)$ is the subgraph of G that contains only those edges $(v, w) \in E$ such that $\ell(w) = \ell(v) + 1$.



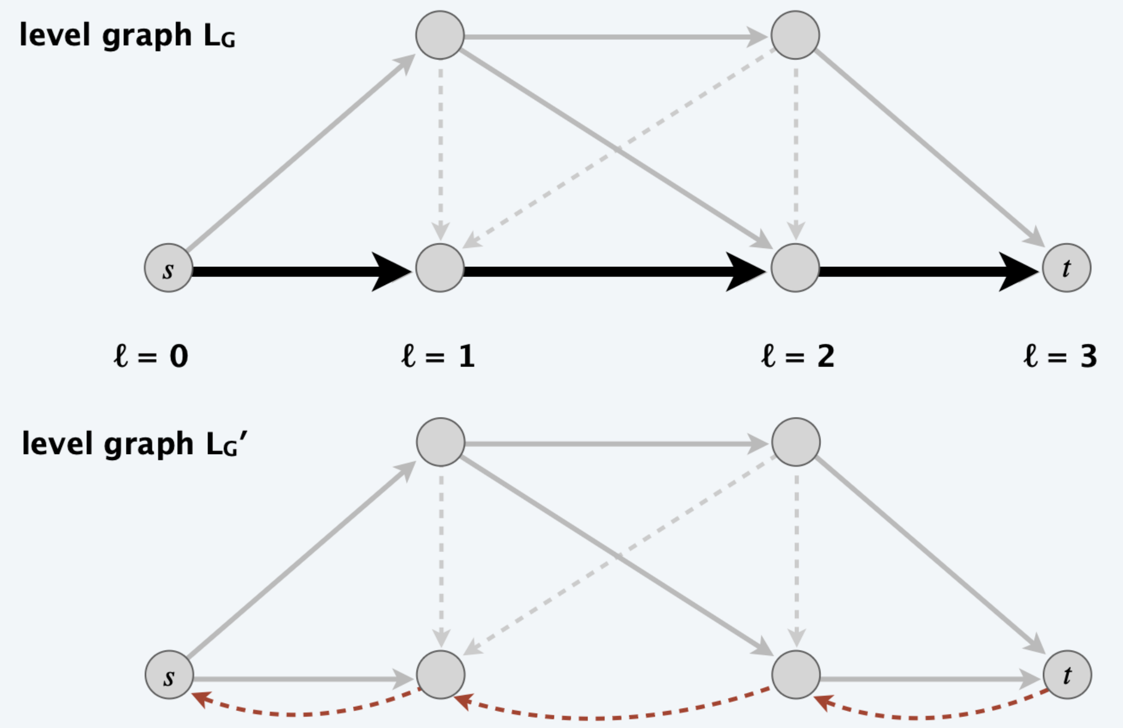


- **Key property.** P is a shortest s-v path in G iff P is an s-v path in L_G.

all possible shortest s-v paths are captured in L_G

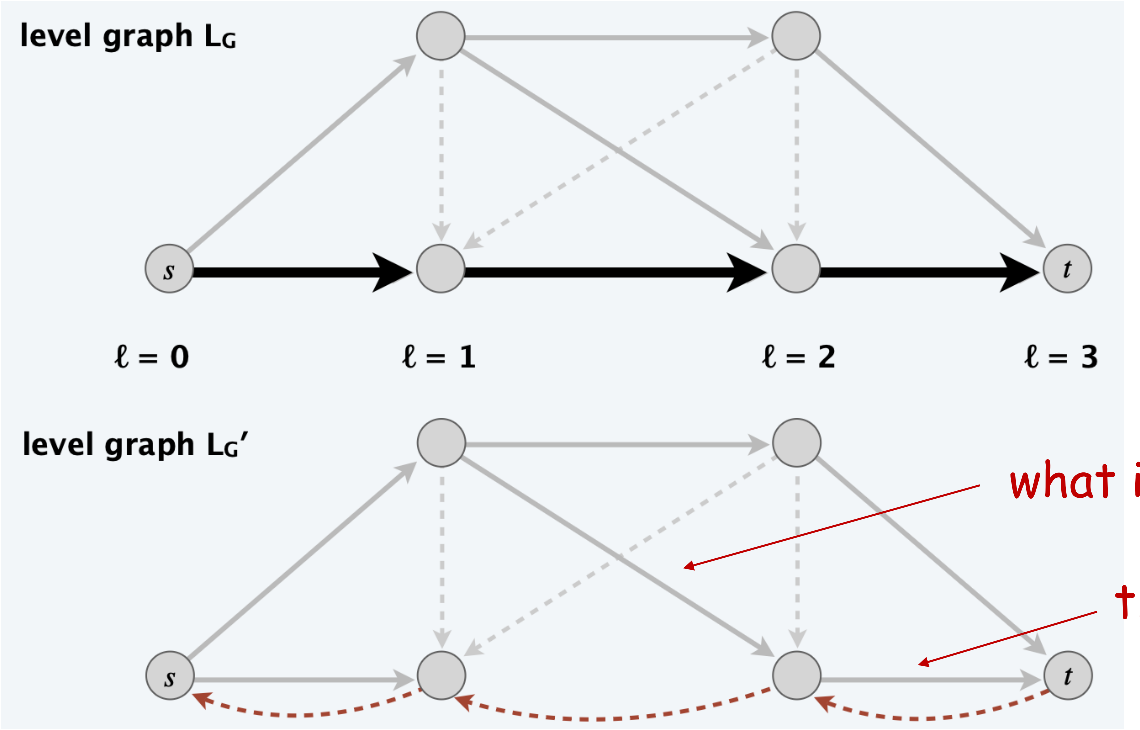
Edmonds-Karp Algorithm: Analysis

- **Pf of Lemma 1:** (Length of a shortest augmenting path never decreases.)
  - Let f and $f\,'$ be flow before and after a shortest-path augmentation.
  - Let L_G and $L_{G'}$ be level graphs of G_f and $G_{f\,'}$. Only reverse edges added to $G_{f'}$.
  - Any s-t path that uses a reverse edge is longer than previous length. ▪



Edmonds-Karp Algorithm: Analysis

- **Pf of Lemma 2:** (After at most *m* shortest-path augmentations, the length of a shortest augmenting path strictly increases.)
  - At least one (bottleneck) edge is deleted from L_G per augmentation.
  - No new edge added to L_G until no s-t path exists, i.e., shortest length strictly increases. ▪



what if this edge is deleted?

this edge can be safely deleted

Edmonds-Karp Algorithm: Summary

- **Lemma 1.** Length of a shortest augmenting path never decreases.
- **Lemma 2.** After at most m shortest-path augmentations, the length of a shortest augmenting path strictly increases.

- **Theorem.** The Edmonds-Karp algorithm takes $O(m^2 n)$ time.

- **Note.** $\Theta(mn)$ augmentations necessary for some flow networks.
  - Try to decrease time per augmentation instead.
  - Simple idea $\Rightarrow O(mn^2)$ [Dinitz 1970] ← next section
  - Dynamic trees $\Rightarrow O(mn \log n)$ [Sleator–Tarjan 1983]

invented in response to a class exercise by Adel'son-Vel'skiĭ

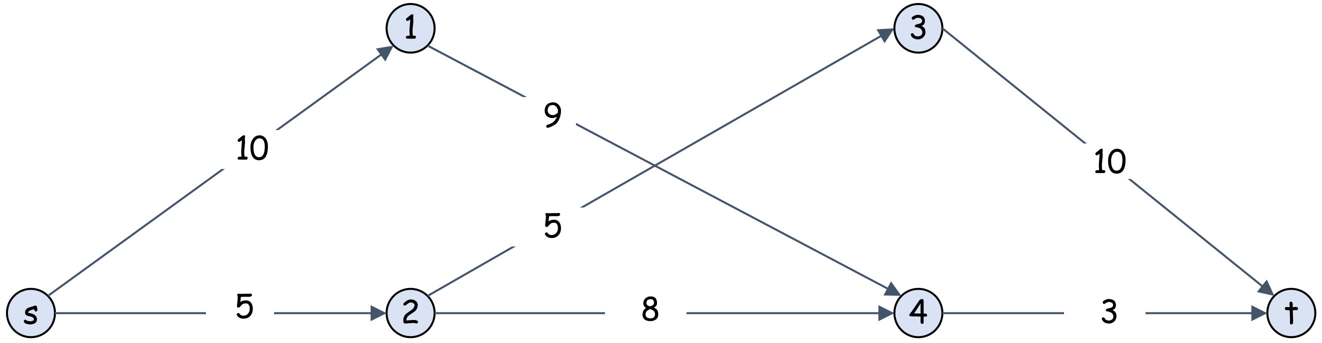
6. Dinitz's Algorithm

Dinitz's Algorithm

- **Two types of augmentations.**
  - Normal: length of shortest path does not change.
  - Special: length of shortest path strictly increases.
- **Phase of normal augmentations.**
  - Construct level graph L_G.
  - Start at s, advance along an edge in L_G until reach t or get stuck.
  - If reach t, augment flow; update L_G; and restart from s.
  - If get stuck, delete node from L_G and retreat to previous node.

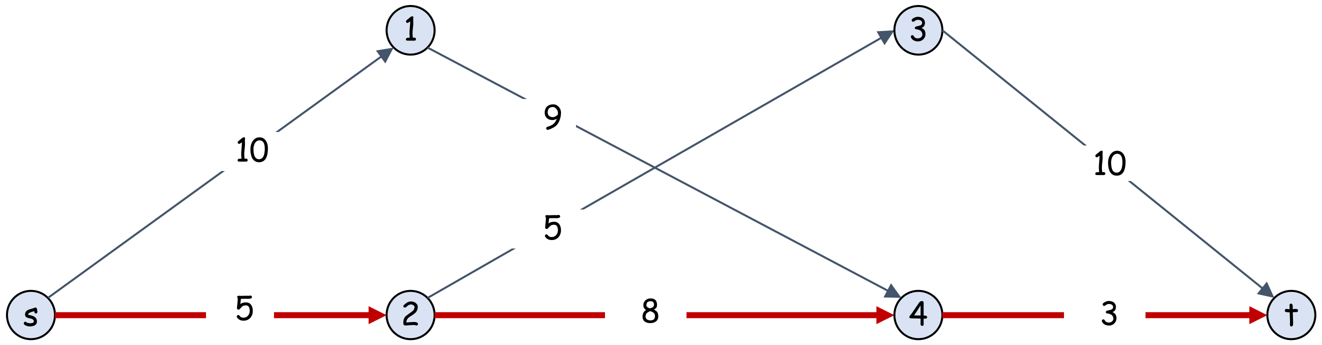
Dinitz’s Algorithm

- **Two types of augmentations.**
  - Normal: length of shortest path does not change.
  - Special: length of shortest path strictly increases.
- **Phase of normal augmentations.**
  - Construct level graph L_G.
  - Start at s, advance along an edge in L_G until reach t or get stuck.
  - If reach t, augment flow; update L_G ; and restart from s.
  - If get stuck, delete node from L_G and retreat to previous node.



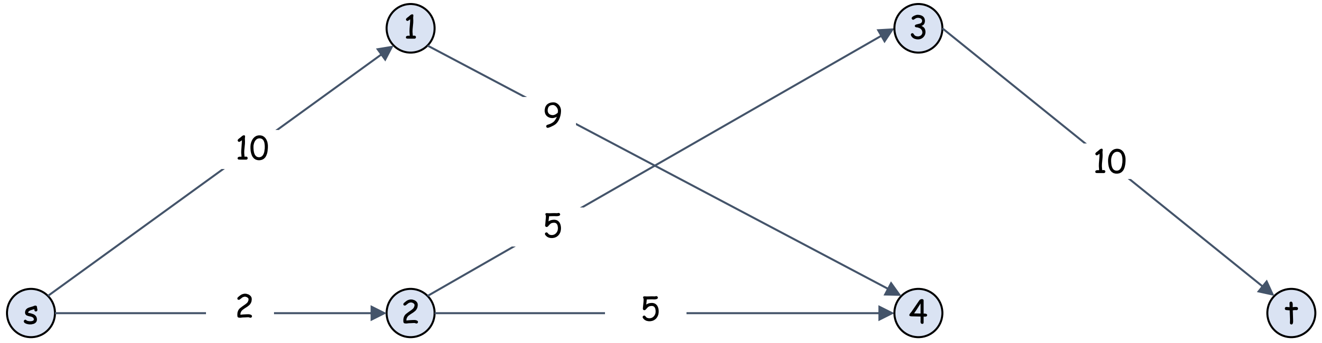
Dinitz's Algorithm

- **Two types of augmentations.**
  - Normal: length of shortest path does not change.
  - Special: length of shortest path strictly increases.
- **Phase of normal augmentations.**
  - Construct level graph L_G.
  - Start at s, advance along an edge in L_G until reach t or get stuck.
  - If reach t, augment flow; update L_G; and restart from s.
  - If get stuck, delete node from L_G and retreat to previous node.



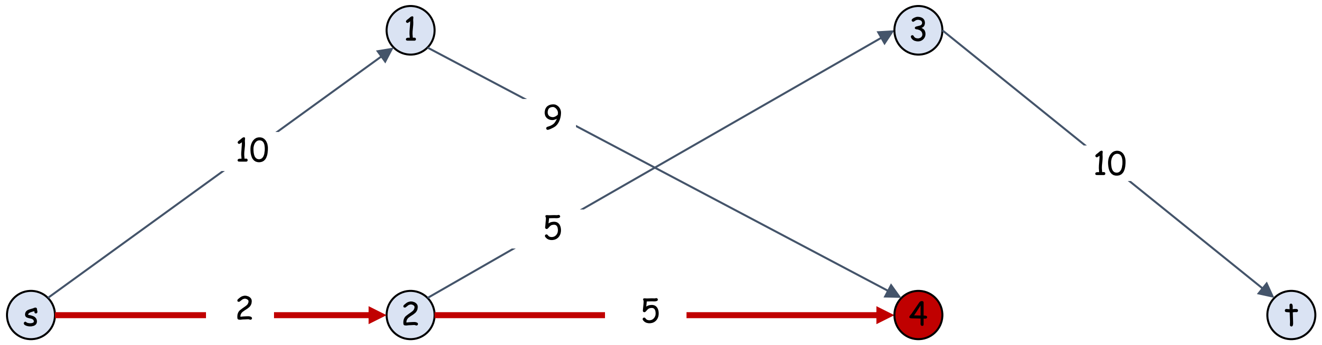
Dinitz's Algorithm

- **Two types of augmentations.**
  - Normal: length of shortest path does not change.
  - Special: length of shortest path strictly increases.
- **Phase of normal augmentations.**
  - Construct level graph L_G.
  - Start at s, advance along an edge in L_G until reach t or get stuck.
  - If reach t, augment flow; update L_G ; and restart from s.
  - If get stuck, delete node from L_G and retreat to previous node.



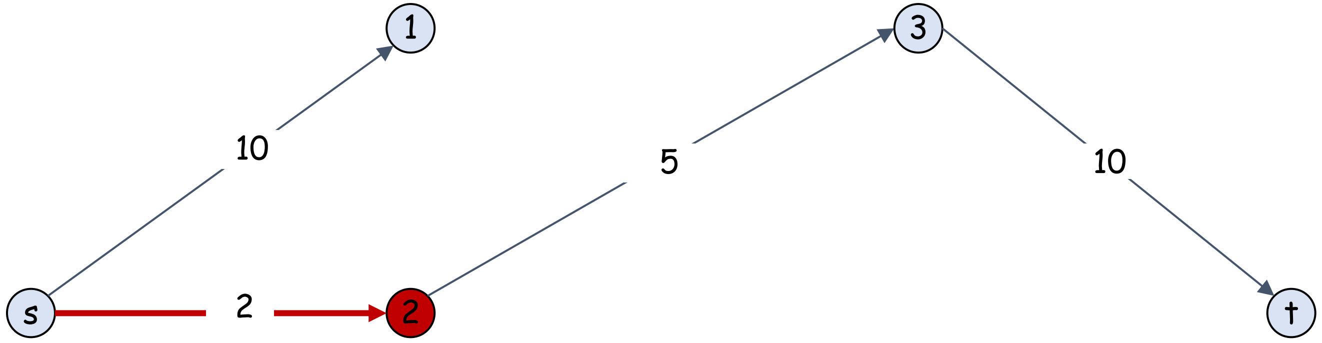
Dinitz's Algorithm

- **Two types of augmentations.**
  - Normal: length of shortest path does not change.
  - Special: length of shortest path strictly increases.
- **Phase of normal augmentations.**
  - Construct level graph L_G.
  - Start at s, advance along an edge in L_G until reach t or get stuck.
  - If reach t, augment flow; update L_G ; and restart from s.
  - If get stuck, delete node from L_G and retreat to previous node.



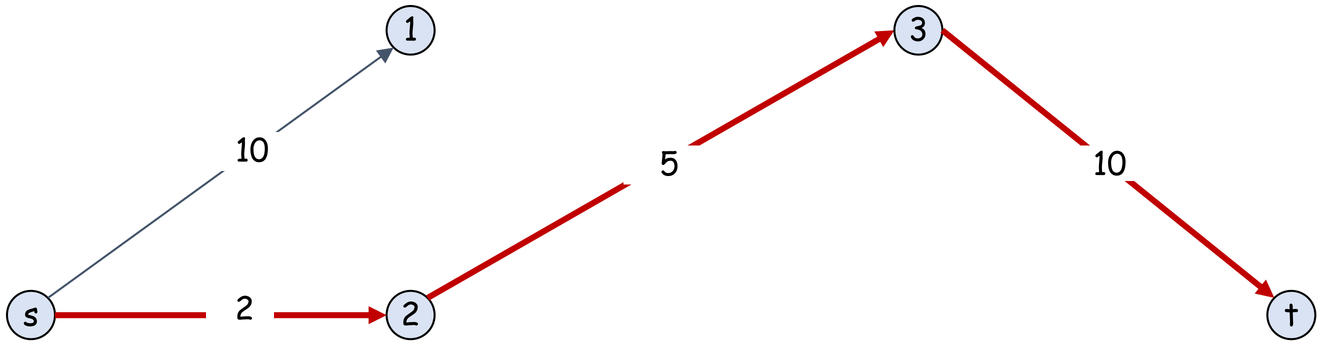
Dinitz's Algorithm

- **Two types of augmentations.**
  - Normal: length of shortest path does not change.
  - Special: length of shortest path strictly increases.
- **Phase of normal augmentations.**
  - Construct level graph L_G.
  - Start at s, advance along an edge in L_G until reach t or get stuck.
  - If reach t, augment flow; update L_G ; and restart from s.
  - If get stuck, delete node from L_G and retreat to previous node.



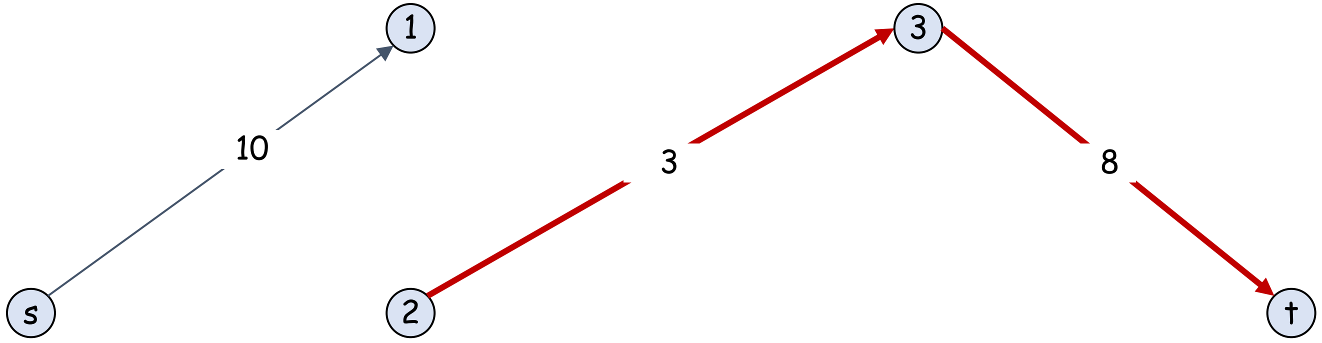
Dinitz's Algorithm

- **Two types of augmentations.**
  - Normal: length of shortest path does not change.
  - Special: length of shortest path strictly increases.
- **Phase of normal augmentations.**
  - Construct level graph L_G.
  - Start at s, advance along an edge in L_G until reach t or get stuck.
  - If reach t, augment flow; update L_G; and restart from s.
  - If get stuck, delete node from L_G and retreat to previous node.



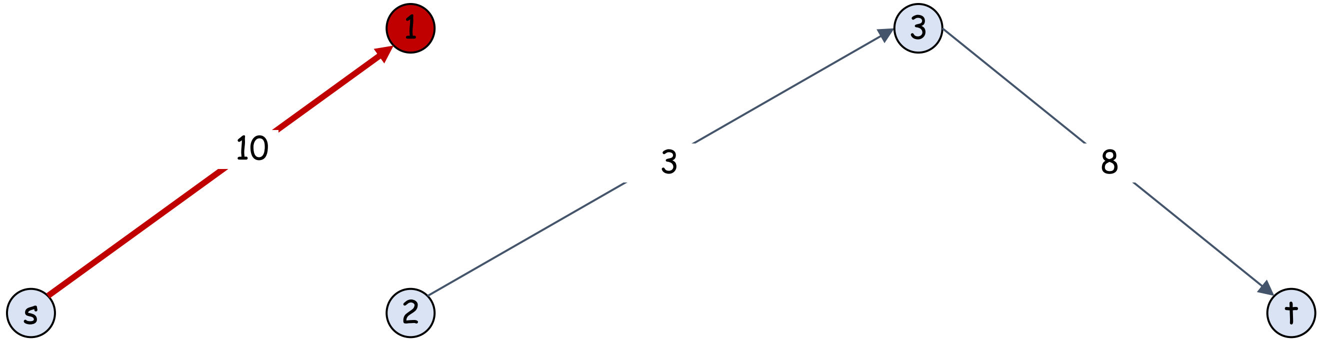
Dinitz's Algorithm

- **Two types of augmentations.**
  - Normal: length of shortest path does not change.
  - Special: length of shortest path strictly increases.
- **Phase of normal augmentations.**
  - Construct level graph L_G.
  - Start at s, advance along an edge in L_G until reach t or get stuck.
  - If reach t, augment flow; update L_G ; and restart from s.
  - If get stuck, delete node from L_G and retreat to previous node.



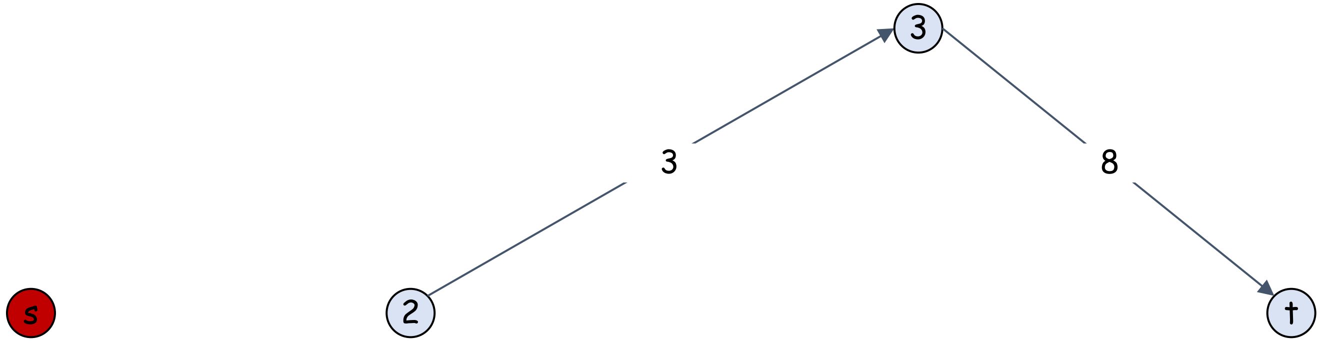
Dinitz's Algorithm

- **Two types of augmentations.**
  - Normal: length of shortest path does not change.
  - Special: length of shortest path strictly increases.
- **Phase of normal augmentations.**
  - Construct level graph L_G.
  - Start at s, advance along an edge in L_G until reach t or get stuck.
  - If reach t, augment flow; update L_G ; and restart from s.
  - If get stuck, delete node from L_G and retreat to previous node.



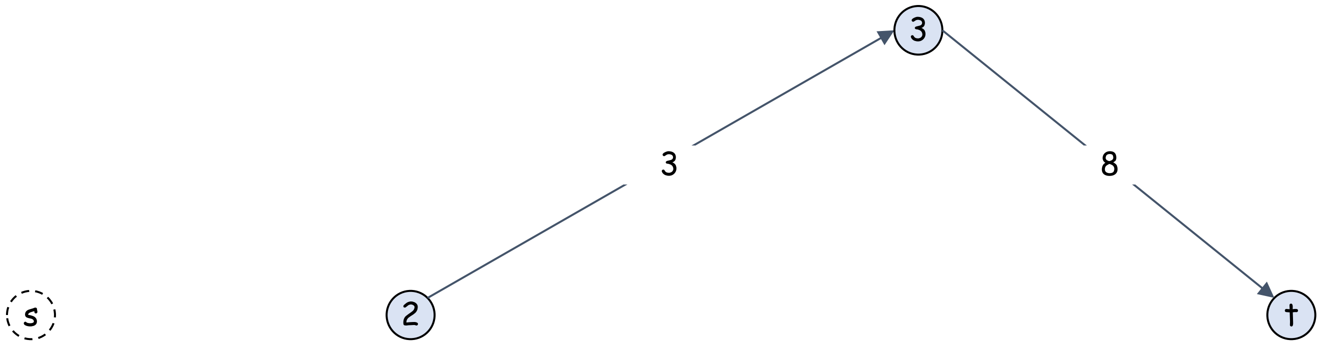
Dinitz's Algorithm

- **Two types of augmentations.**
  - Normal: length of shortest path does not change.
  - Special: length of shortest path strictly increases.
- **Phase of normal augmentations.**
  - Construct level graph L_G.
  - Start at s, advance along an edge in L_G until reach t or get stuck.
  - If reach t, augment flow; update L_G ; and restart from s.
  - If get stuck, delete node from L_G and retreat to previous node.



Dinitz's Algorithm

- **Two types of augmentations.**
  - Normal: length of shortest path does not change.
  - Special: length of shortest path strictly increases.
- **Phase of normal augmentations.**
  - Construct level graph L_G.
  - Start at s, advance along an edge in L_G until reach t or get stuck.
  - If reach t, augment flow; update L_G ; and restart from s.
  - If get stuck, delete node from L_G and retreat to previous node.



Dinitz's Algorithm

- **Two types of augmentations.**
  - Normal: length of shortest path does not change.
  - Special: length of shortest path strictly increases.
- **Dinitz's algorithm per normal phase: (as refined by Even and Itai)**

```
Dinitz-Normal-Phase(Gf, s, t) {
   LG = level graph of Gf
   P = empty path
   Advance(s)
}

Retreat(v) {
   if (v = s) return
   else
      delete v and incident edges from LG
      remove last edge (u, v) from P
      Advance(u)
}
```

```
Advance(v) {
   if (v = t)
      f = Augment(f, c, P)
      remove bottleneck edges from LG
      P = empty path
      Advance(s)
   if (there exists (v, w) ∈ LG)
      add edge (v, w) to P
      Advance(w)

   Retreat(v)
}
```

Dinitz's Algorithm

- **Two types of augmentations.**
  - Normal: length of shortest path does not change.
  - Special: length of shortest path strictly increases.
- **Dinitz's algorithm:**

```
Dinitz(G, s, t, c) {
   foreach e ∈ E: f(e) = 0
   Gf = residual network of G with respect to flow f

   while (there exists an augmenting path P in Gf) {
      Dinitz-Normal-Phase(Gf, s, t)
   }
   return f
}
```

Dinitz's Algorithm: Analysis

- **Lemma.** A phase can be implemented to run in $O(mn)$ time.
- **Pf.**
  - Initialization happens once per phase. ← O(m) per phase using BFS
  - At most m augmentations per phase. ← O(mn) per phase
    (because an augmentation deletes at least one edge from L_G)
  - At most n retreats per phase. ← O(m + n) per phase
    (because a retreat deletes one node and all incident edges from L_G)
  - At most mn advances per phase. ← O(mn) per phase
    (because at most n advances before retreat or augmentation) ▪
- **Theorem.** [Dinitz 1970] Dinitz' algorithm runs in $O(mn^2)$ time.
- **Pf.** There are at most $n - 1$ phases and each phase runs in $O(mn)$ time. ▪

Augmenting-Path Algorithms: Summary

year	method	# augmentations	running time
1955	**augmenting path**	$n\,C$	$O(m\,n\,C)$
1972	**fattest path**	$m \log(mC)$	$O(m^2 \log n \log(mC))$
1972	**capacity scaling**	$m \log C$	$O(m^2 \log C)$
1985	**improved capacity scaling**	$m \log C$	$O(m\,n \log C)$
1970	**shortest augmenting path**	$m\,n$	$O(m^2\,n)$
1970	**level graph**	$m\,n$	$O(m\,n^2)$
1983	**dynamic trees**	$m\,n$	$O(m\,n \log n)$

augmenting-path algorithms with m edges, n nodes, and integer capacities between 1 and C

fat paths

shortest paths

Max-Flow Algorithms: Theory Highlights

year	method	worst case	discovered by
1951	**simplex**	$O(m\, n^2\, C)$	Dantzig
1955	**augmenting paths**	$O(m\, n\, C)$	Ford–Fulkerson
1970	**shortest augmenting paths**	$O(m\, n^2)$	Edmonds–Karp, Dinitz
1974	**blocking flows**	$O(n^3)$	Karzanov
1983	**dynamic trees**	$O(m\, n \log n)$	Sleator–Tarjan
1985	**improved capacity scaling**	$O(m\, n \log C)$	Gabow
1988	**push-relabel**	$O(m\, n \log (n^2 / m))$	Goldberg–Tarjan
1998	**binary blocking flows**	$O(m^{3/2} \log (n^2 / m) \log C)$	Goldberg–Rao
2013	**compact networks**	$O(m\, n)$	Orlin
2014	**interior-point methods**	$\tilde{O}(m\, n^{1/2} \log C)$	Lee–Sidford
2016	**electrical flows**	$\tilde{O}(m^{10/7}\, C^{1/7})$	Mądry
20xx		???	

max-flow algorithms with m edges, n nodes, and integer capacities between 1 and C

Max-Flow Algorithms: Practice

- **Caveat.** Worst-case running time is generally not useful for predicting or comparing max-flow algorithm performance in practice.
- **Best in practice.** Push-relabel algorithm [Goldberg-Tarjan 1988] with gap relabeling: $O(m^{3/2})$ in practice. [Textbook, Section 7.4]
  - Increases flow one edge at a time instead of one augmenting path at a time.
- **Computer vision.** Different algorithms work better for some dense problems that arise in applications to computer vision.
- **Implementation.** MATLAB, Google OR-Tools, etc.