ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS

Special Topics on Algorithms

Algorithms for flows and matchings

Vangelis Markakis – George Zois

Contents

The maximum flow problem

Maximum Flow and Minimum Cut

Max flow and min cut.

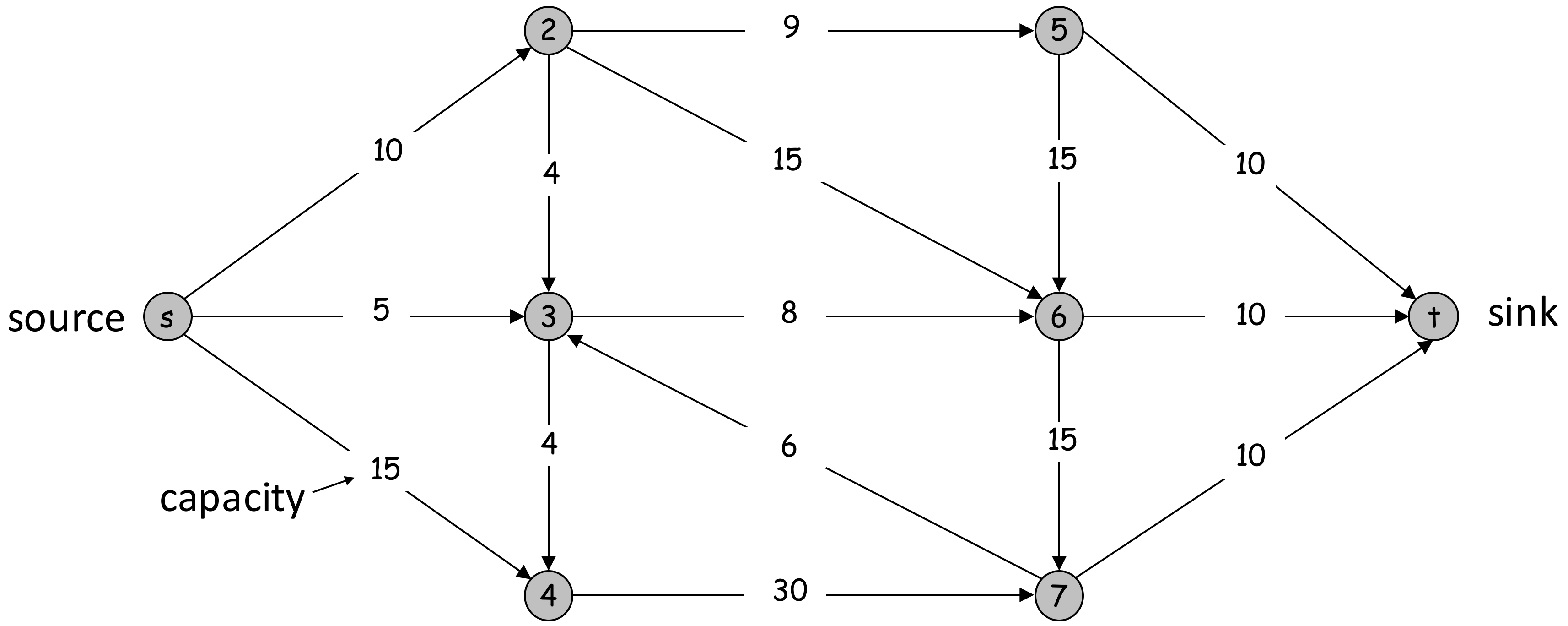
- Two very rich algorithmic problems.
- Cornerstone problems in combinatorial optimization.
- Beautiful mathematical duality.

Nontrivial applications / reductions.

- Data mining.
- Project selection.
- Airline scheduling.
- Bipartite matching.
- Image segmentation.
- Network connectivity.
- Network reliability.
- Distributed computing.
- Security of statistical data.
- Many many more . . .

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.



The max flow problem

A feasible flow is an assignment of a flow f(e) to every edge so that

1. f(e) ≤ c(e) (capacity constraints)
2. For every node other than source and sink:

    incoming flow = outgoing flow (preservation of flow)

Goal: find a feasible flow so as to maximize the total amount of flow coming out of s (or equivalently going into t)

Flow going out of s: v(f) = $\sum_{(s,u)\in E} f(s,u)$

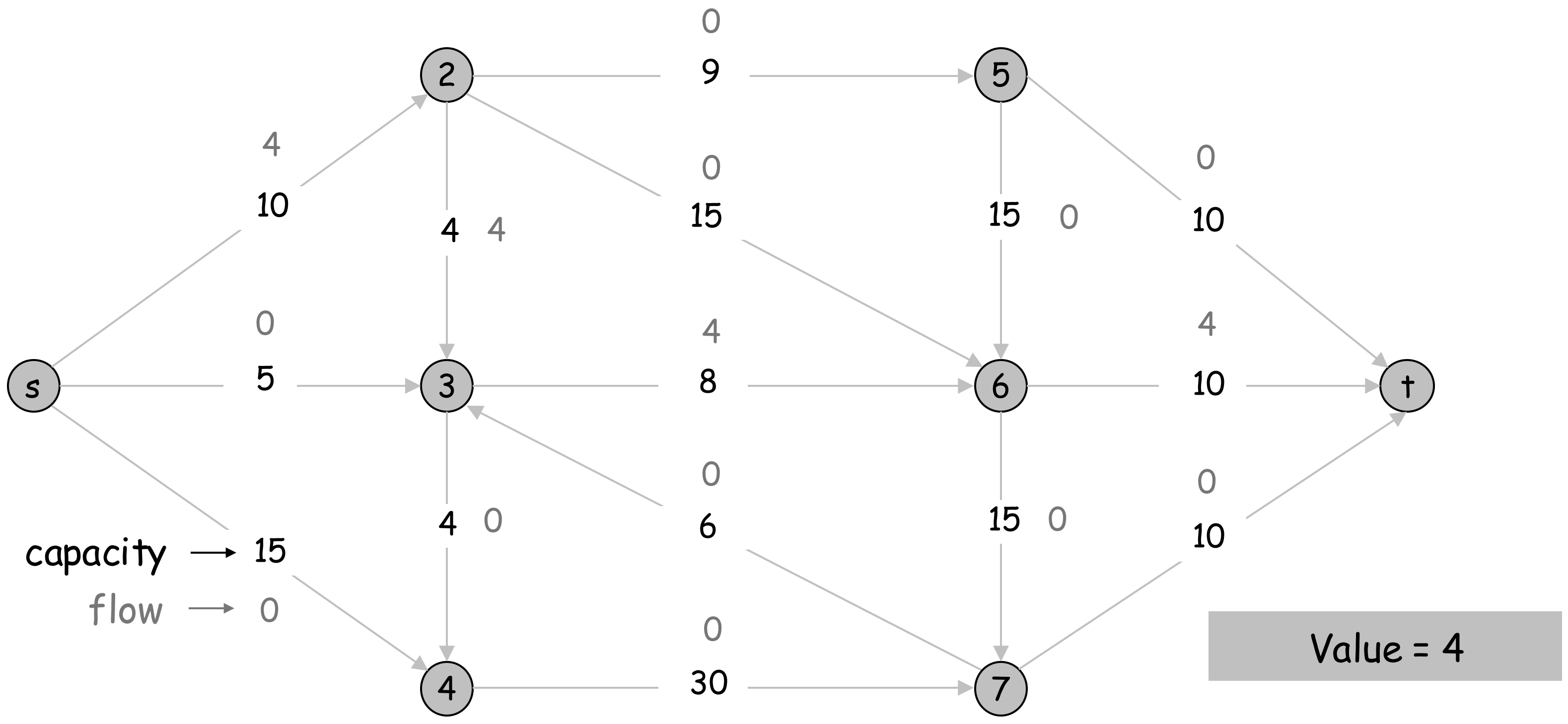
By preservation of flow this equals: $\sum_{(u,t)\in E} f(u,t)$

Flows

Constraints:

- For each e ∈ E: $0 \le f(e) \le c(e)$ (capacity)
- For each v ∈ V – {s, t}: $\sum_{e \text{ in to } v} f(e) = \sum_{e \text{ out of } v} f(e)$ (conservation)

The value of a flow f is: $v(f) = \sum_{e \text{ out of } s} f(e)$.

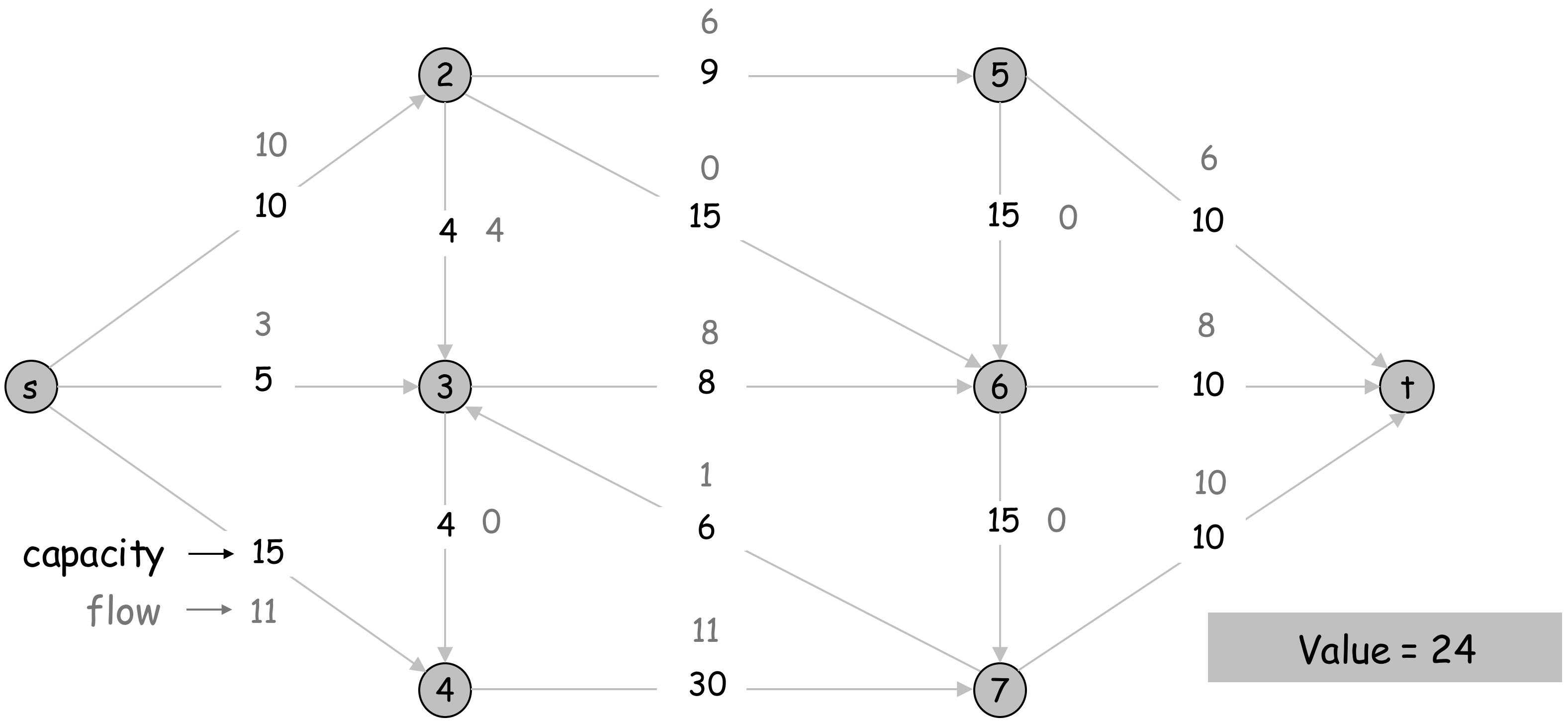


Flows

Constraints:

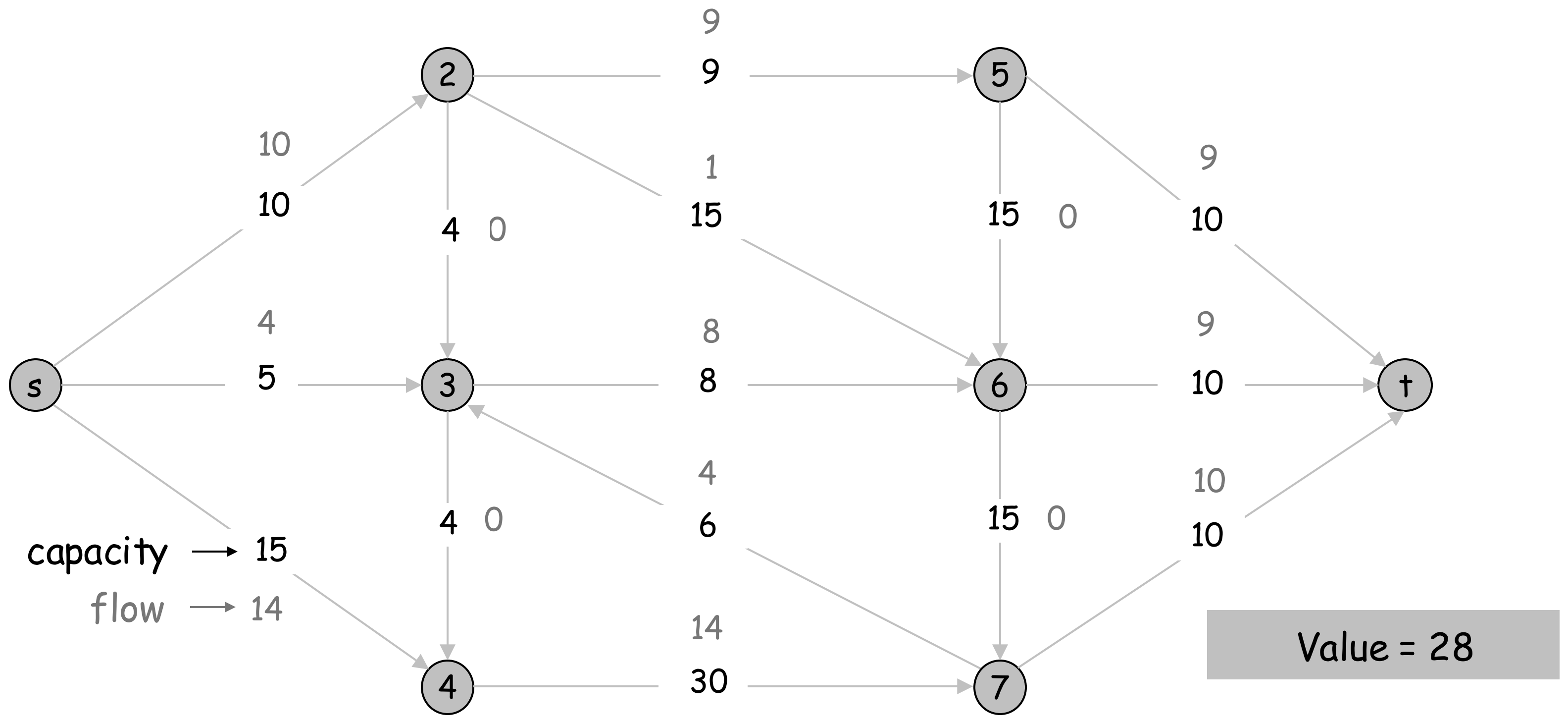
- For each e ∈ E: $0 \pounds f(e) \pounds c(e)$ (capacity)
- For each v ∈ V – {s, t}: $\underset{e \text{ in to } v}{\text{å}} f(e) = \underset{e \text{ out of } v}{\text{å}} f(e)$ (conservation)

The value of a flow f is: $v(f) = \underset{e \text{ out of } s}{\text{å}} f(e)$.



The Maximum Flow Problem

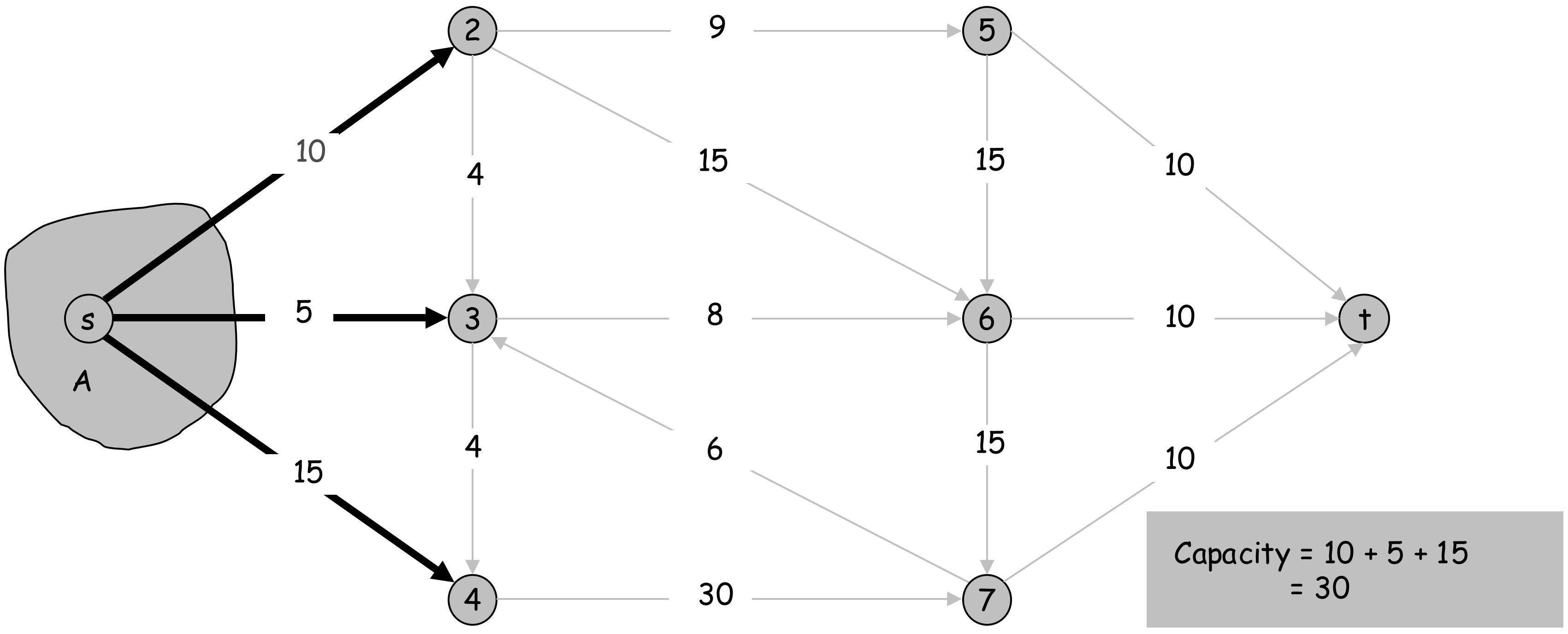
Optimal flow: 28 units of flow from s to t



Cuts

Def. An s-t cut is a partition (A, B) of V with $s \in A$ and $t \in B$.

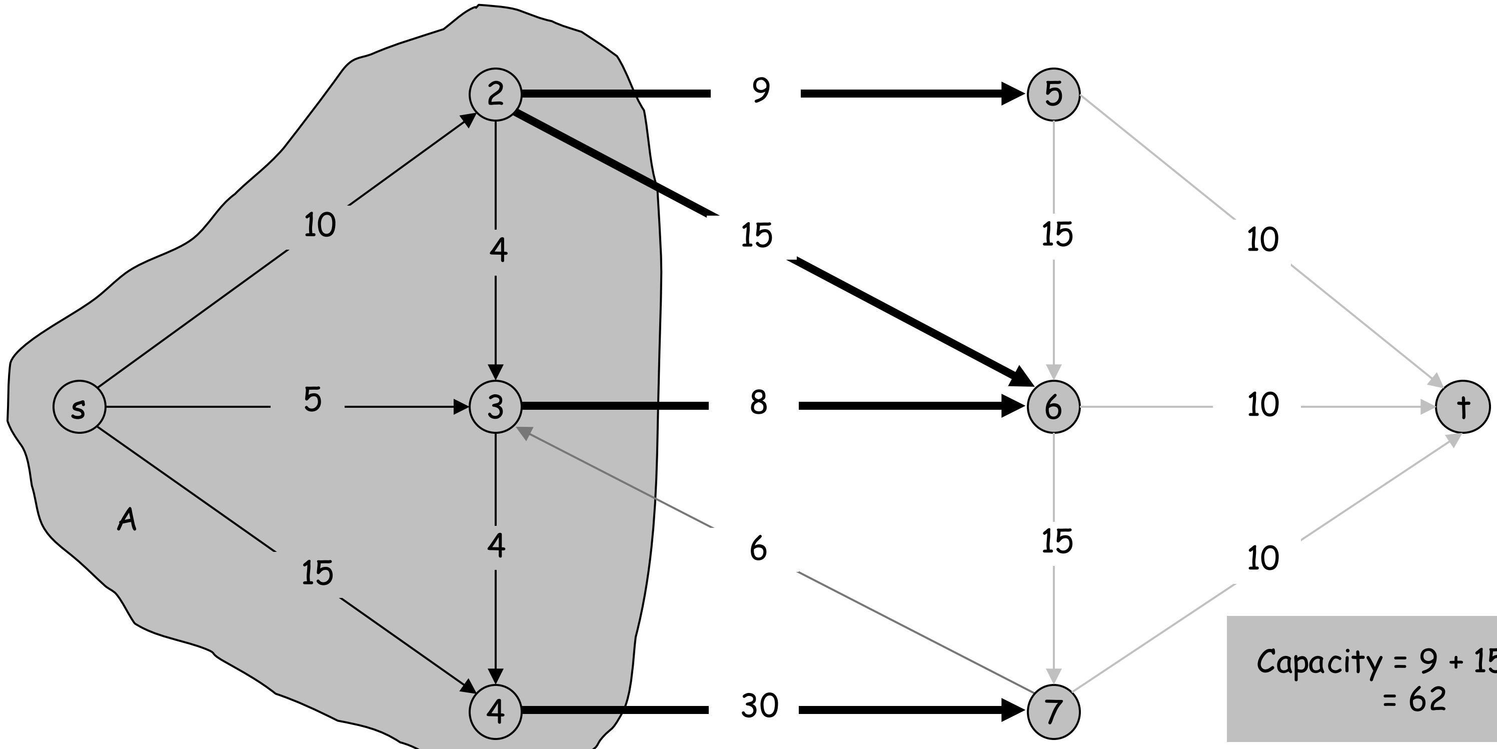
Def. The capacity of a cut (A, B) is: $cap(A, B) = \sum_{e \text{ out of } A} c(e)$



Cuts

Def. An s-t cut is a partition (A, B) of V with s $\in$ A and t $\in$ B.

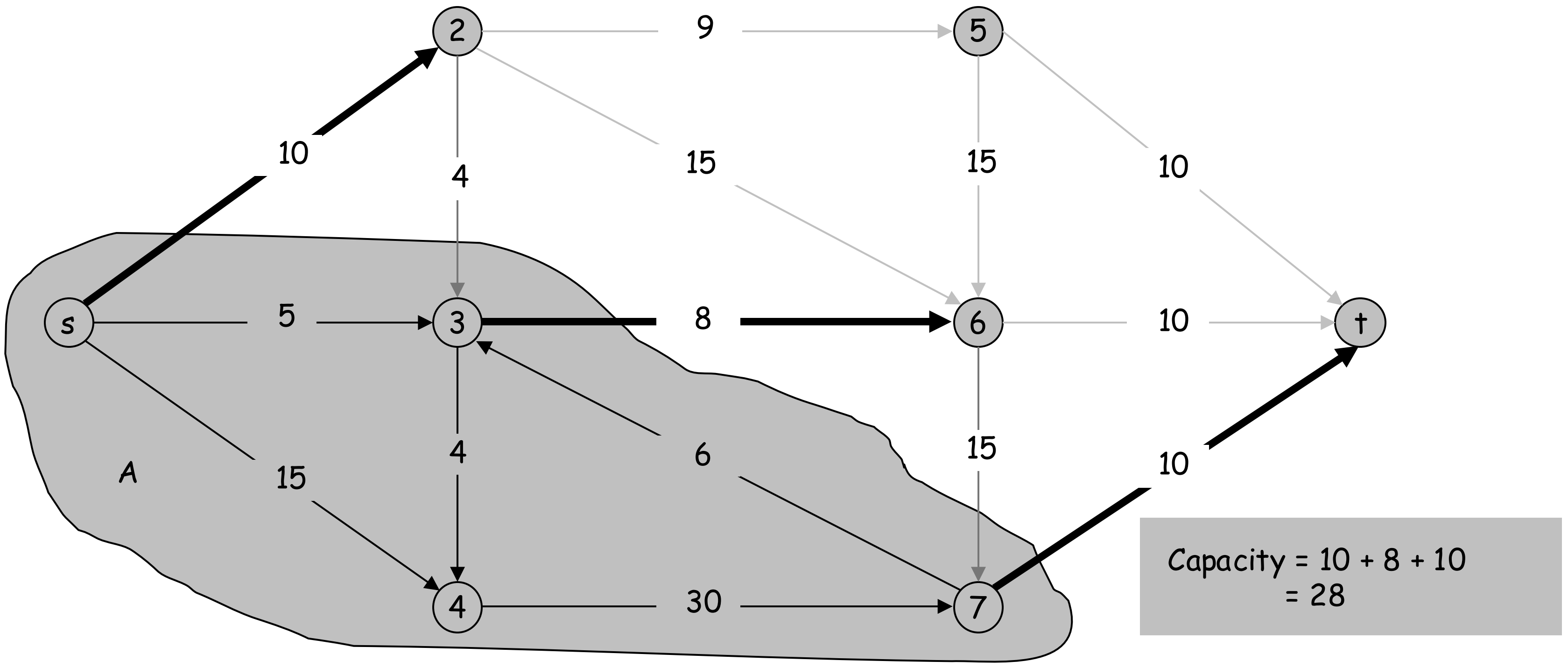
Def. The capacity of a cut (A, B) is: $cap(A, B) = \sum_{e \text{ out of } A} c(e)$



Capacity = 9 + 15 + 8 + 30
= 62

The Minimum Cut Problem

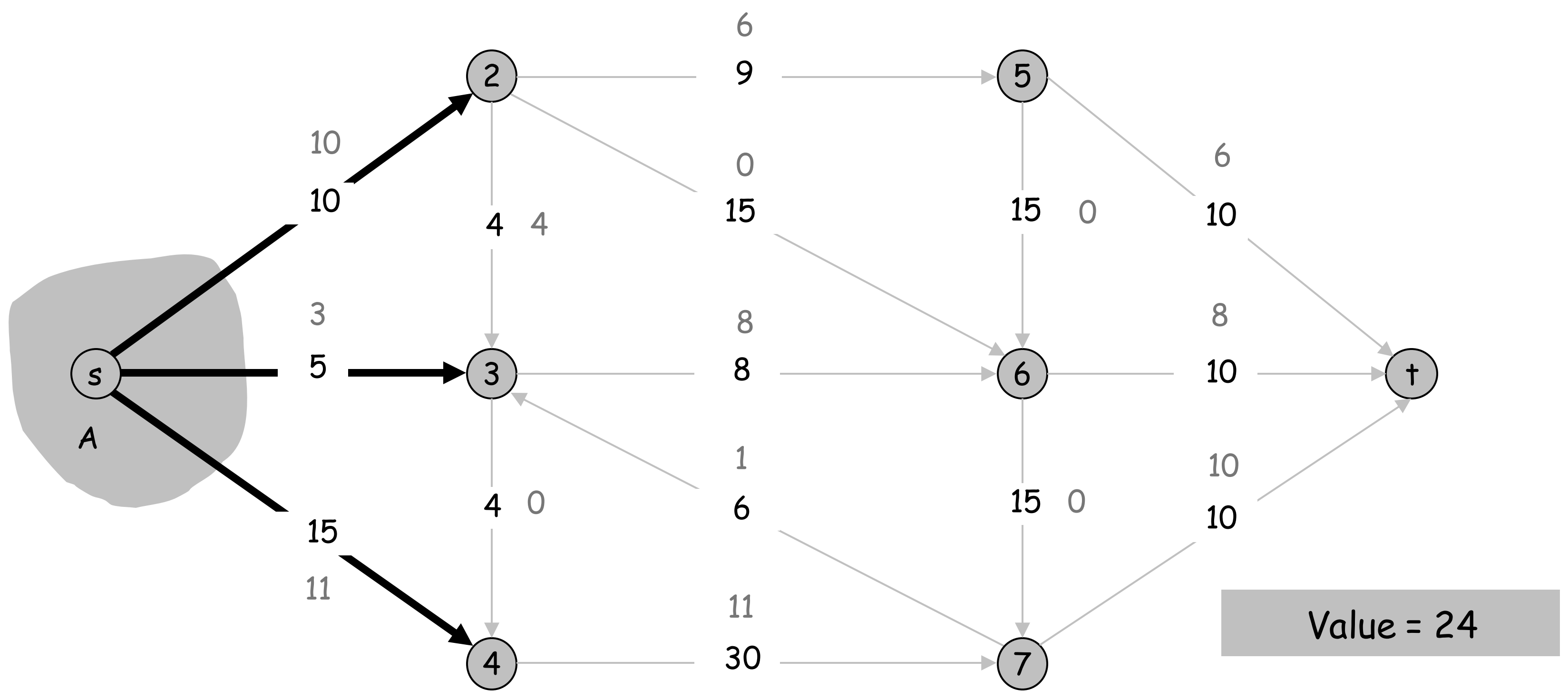
Min s-t cut problem. Find an s-t cut of minimum capacity.



Flow and Cut Properties

Lemma 1. Let f be any flow, and let (A, B) be any s-t cut. Then, the net flow sent across the cut is equal to the amount leaving s.

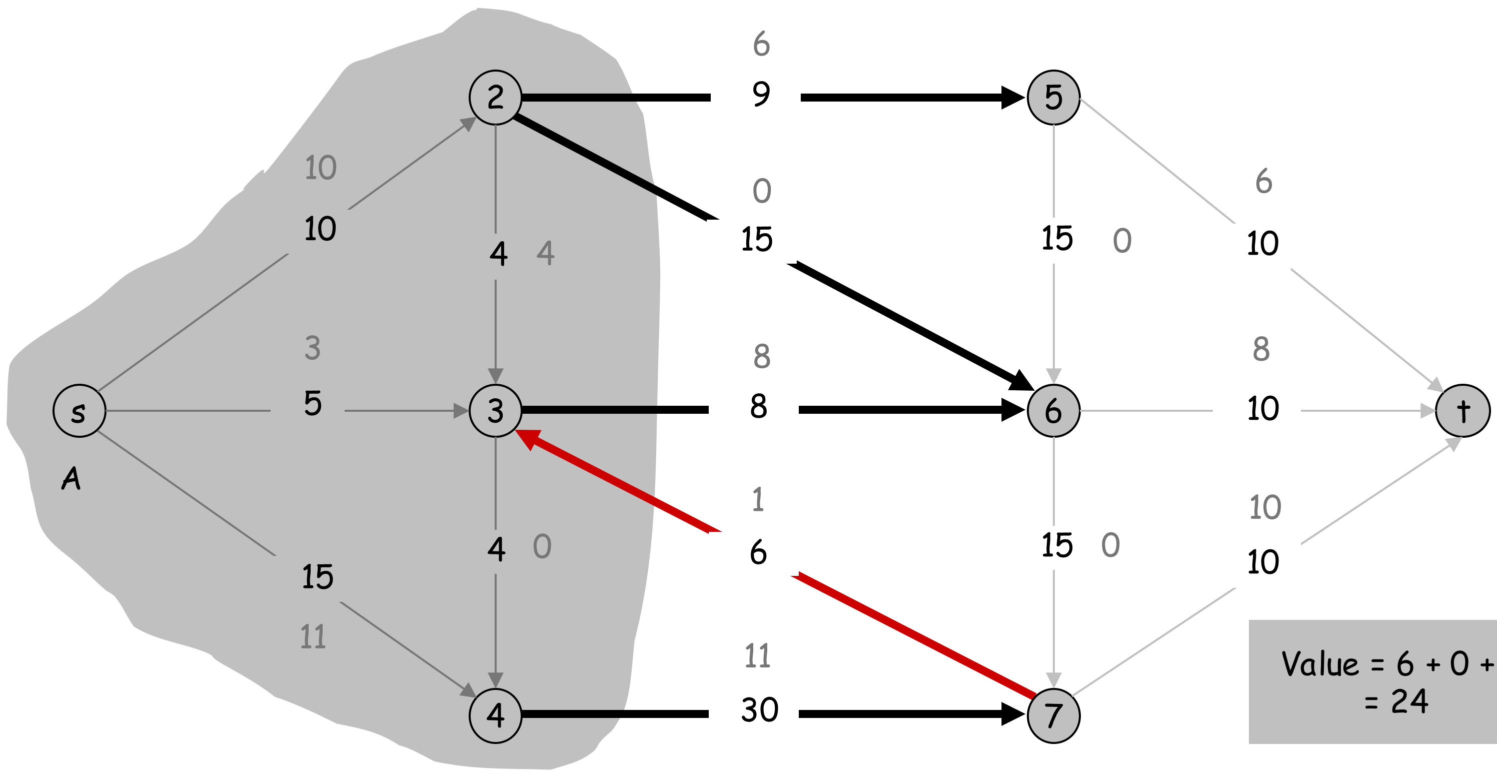
$$\sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to A}} f(e) = v(f)$$



Flow and Cut Properties

Lemma 1. Let f be any flow, and let (A, B) be any s-t cut. Then, the net flow sent across the cut is equal to the amount leaving s.

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

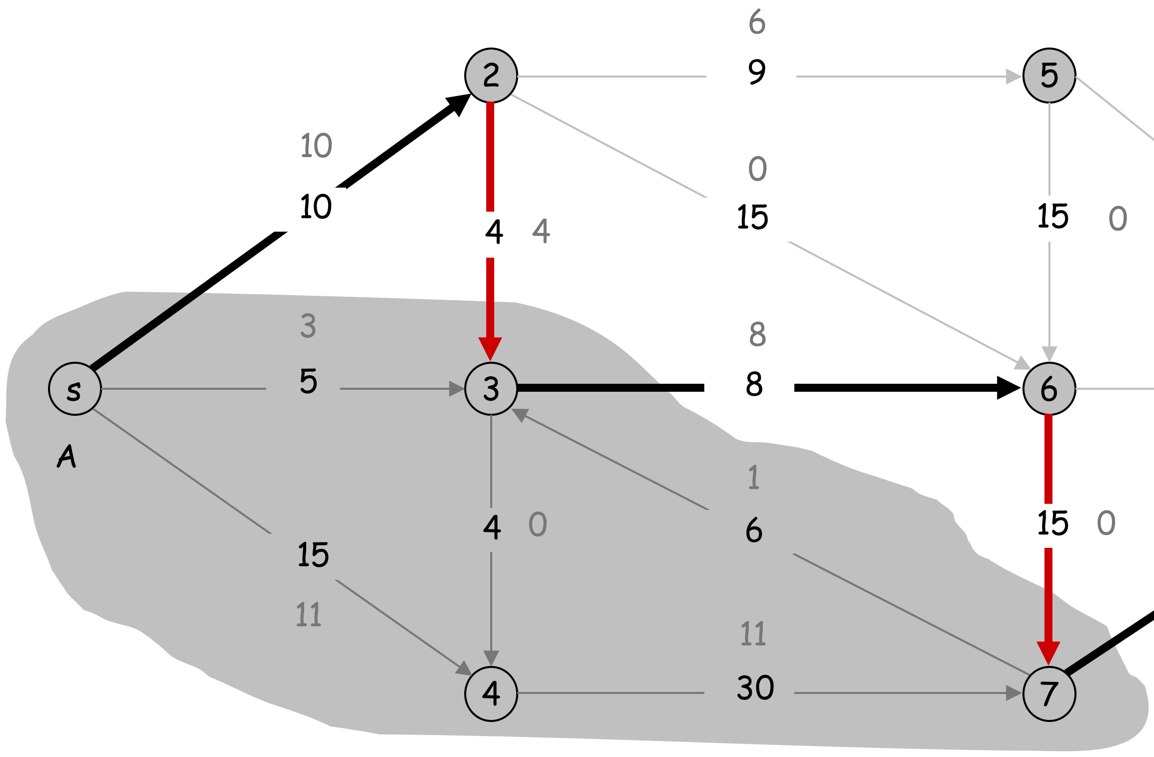


Value = 6 + 0 + 8 - 1 + 11
= 24

Flow and Cut Properties

Lemma 1. Let f be any flow, and let (A, B) be any s-t cut. Then, the net flow sent across the cut is equal to the amount leaving s.

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



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

Flow and Cut Properties

Lemma 1. Let f be any flow, and let (A, B) be any s-t cut. Then, the net flow sent across the cut is equal to the amount leaving s.

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

Pf.

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

by flow conservation, all terms except v = s are 0 $\longrightarrow$

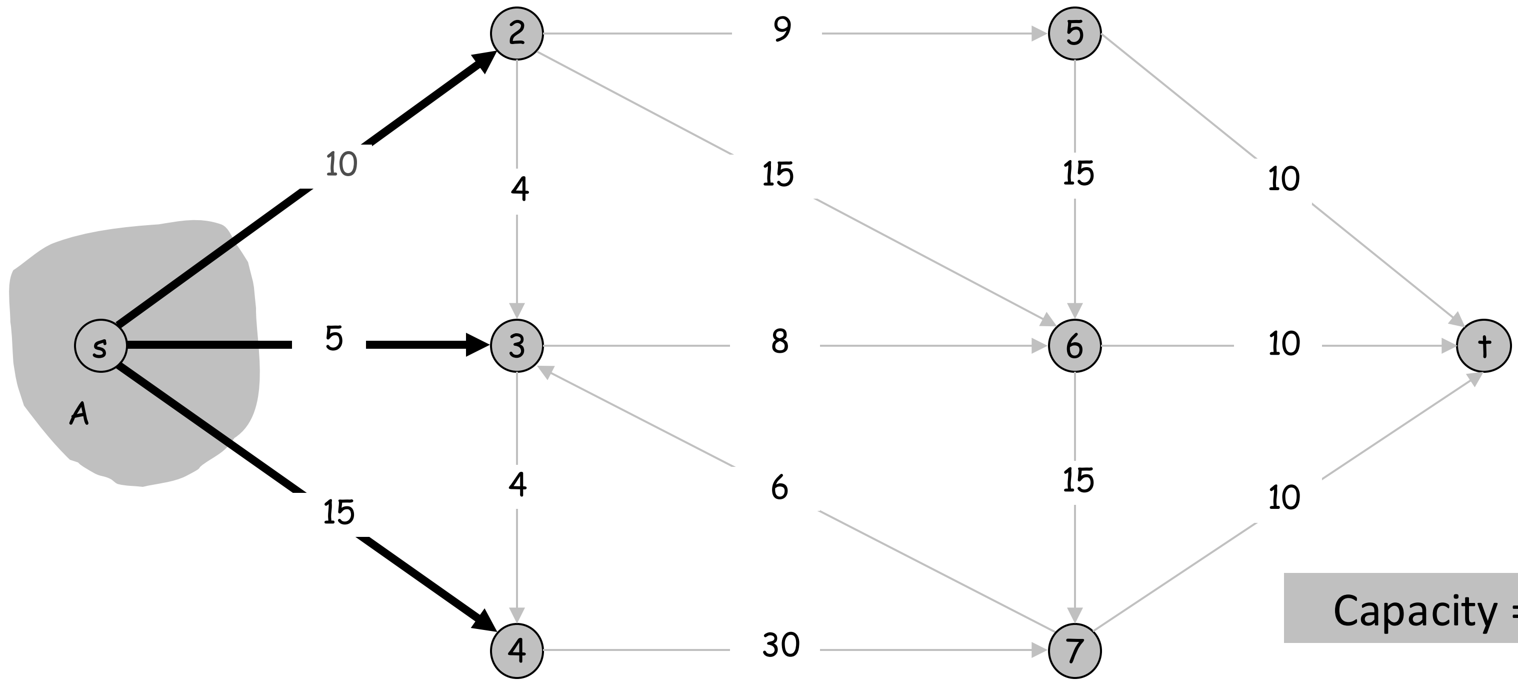
$$= \sum_{v \in A} \left(\sum_{e \text{ out of } v} f(e) - \sum_{e \text{ in to v}} f(e) \right)$$

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

Flow and Cut Properties

Lemma 2. Let f be any flow, and let (A, B) be any s-t cut. Then the value of the flow is at most the capacity of the cut.

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



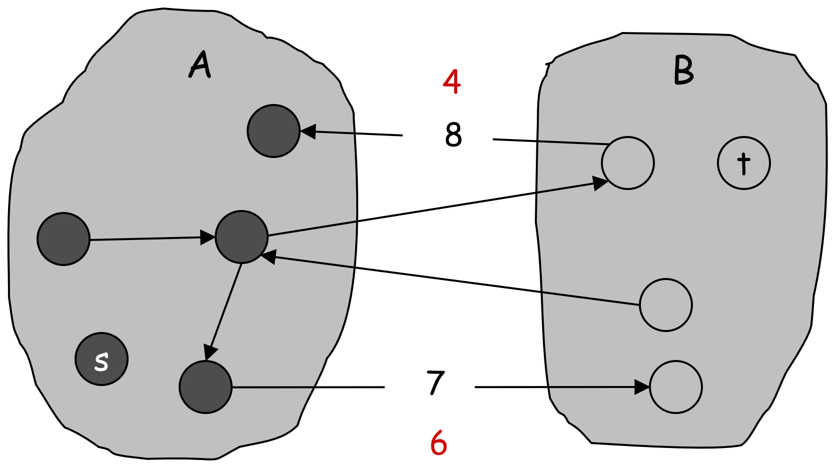
Capacity = 30

Flow and Cut Properties

Lemma 2. Let f be any flow. Then, for any s-t cut (A, B) we have
v(f) ≤ cap(A, B).

Pf.

$$
\begin{aligned}
v(f) &= \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e) \\
&\le \sum_{e \text{ out of } A} f(e) \\
&\le \sum_{e \text{ out of } A} c(e) \\
&= \text{cap}(A, B) \qquad ▪
\end{aligned}
$$

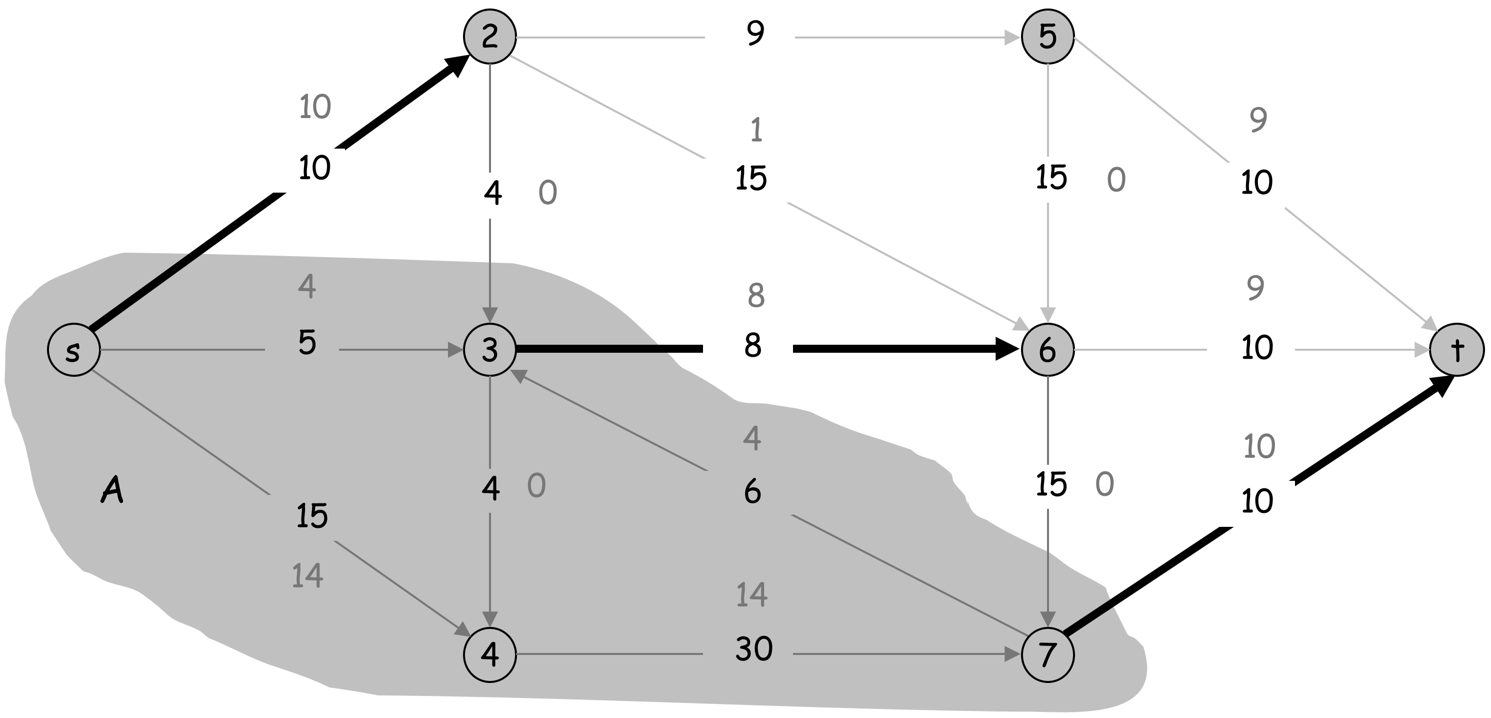


Certificate of Optimality

Corollary 1. Max flow is at most equal to the capacity of the min cut (i.e., max flow is a lower bound to min cut)

Corollary 2. Let f be any flow, and let (A, B) be any cut.
If v(f) = cap(A, B), then f is a max flow and (A, B) is a min cut.

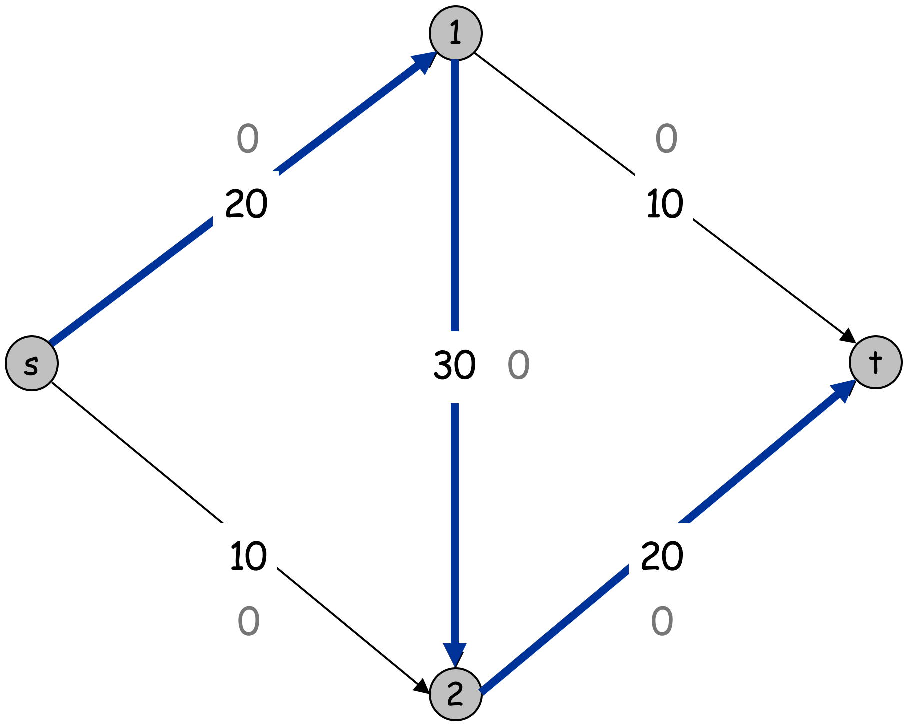
Value of flow = Cut capacity = 28 $\Rightarrow$ Flow value $\leq$ 28



Towards a Max Flow Algorithm

Greedy algorithm.

- Start with f(e) = 0 for every 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.

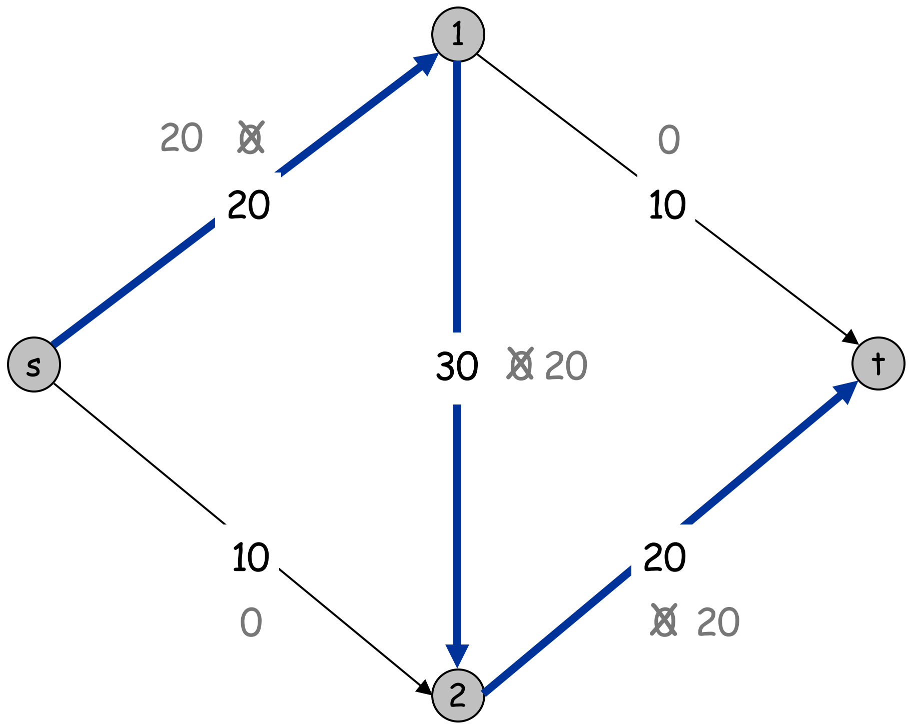


Flow value = 0

Towards a Max Flow Algorithm

Greedy algorithm.

- Start with f(e) = 0 for every 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.



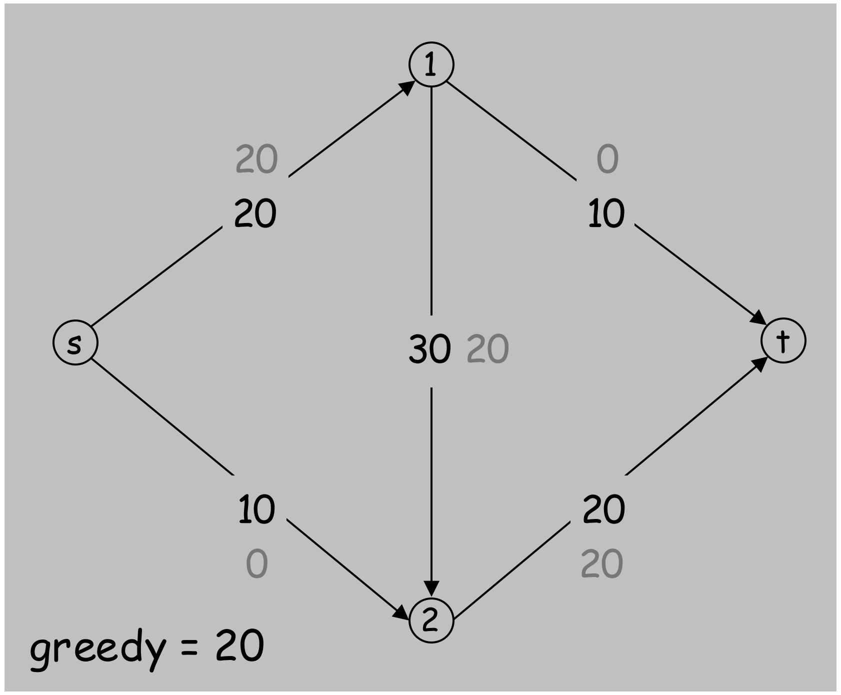
Flow value = 20

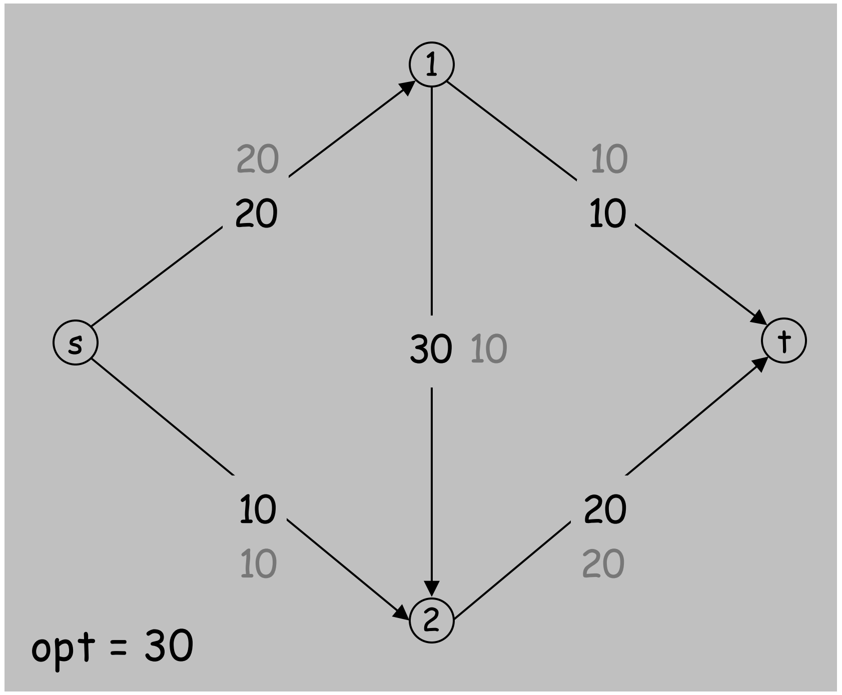
Towards a Max Flow Algorithm

Greedy algorithm.

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

↖ locally optimality $\not\Rightarrow$ global optimality





Towards a Max Flow Algorithm

We need an algorithm with more flexibility
Desired operations:

- Push flow forward along a non-saturated path
- Push flow backwards (i.e., undo some units of flow when necessary)
  - in order to to divert flow to a different direction

The residual graph:

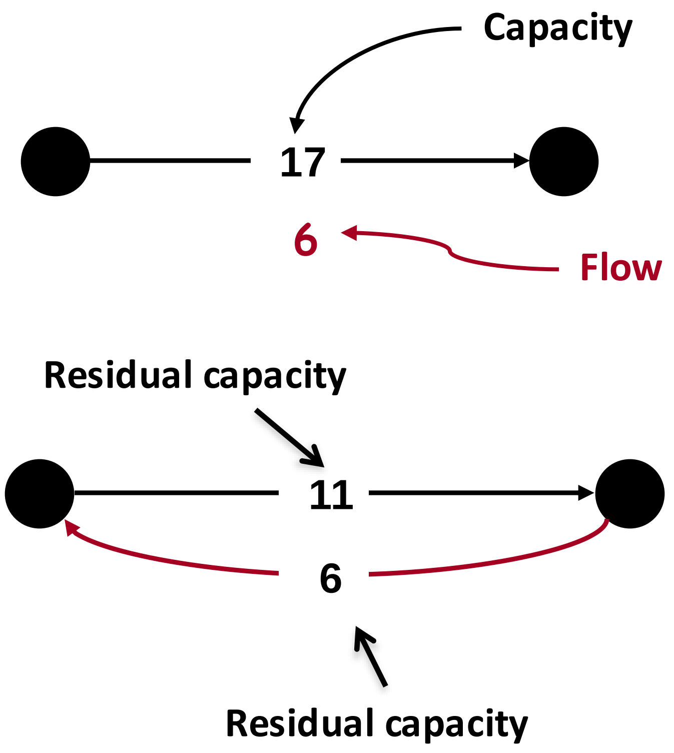
Given the initial graph G, and a fesible flow f, the residual graph G_f has

- the same set of nodes as G
- forward edges: for every edge e = (u, v) of G with f(e) < c(e), we include the same edge in G_f with residual capacity c(e) – f(e)
- backward edges: for every edge e = (u, v) of G with f(e) > 0, we include the edge (v, u) in G_f with residual capacity f(e)

Towards a Max Flow Algorithm

Simple Facts:

- Given G and f, the graph G_f can be constructed efficiently
- G_f has at most twice as many edges as G
- Capacities in G_f are strictly positive



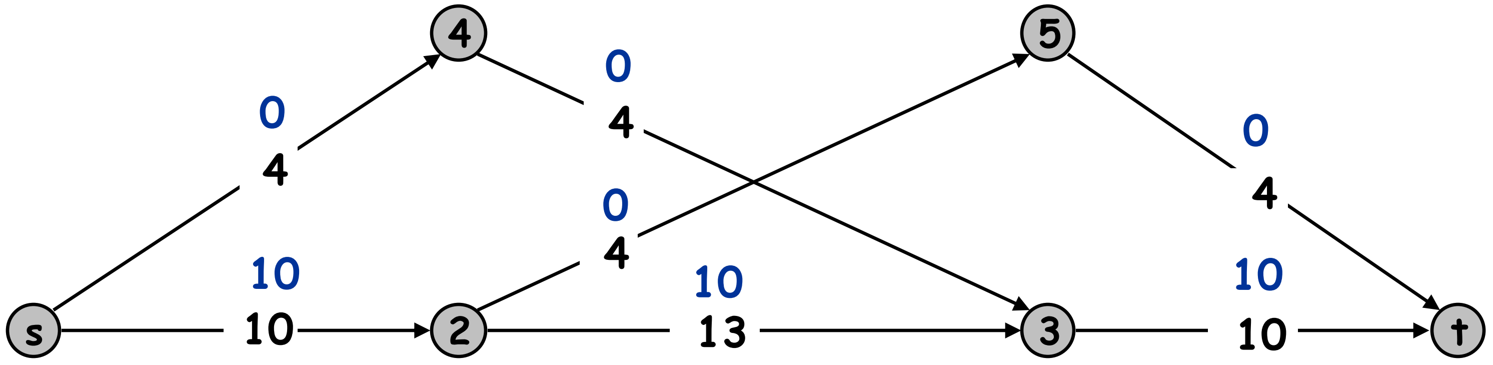
Residual Graph and Augmenting Paths

Residual graph: $G_f = (V, E_f)$.

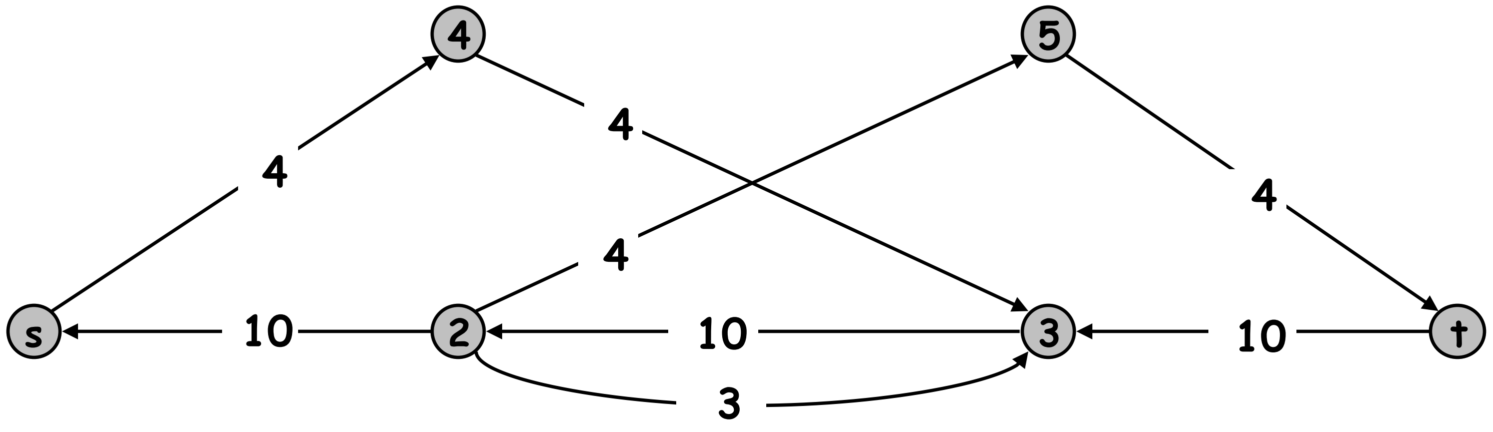
- $E_f = \{e : f(e) < c(e)\} \cup \{e^R : f(e) > 0\}$.

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

G



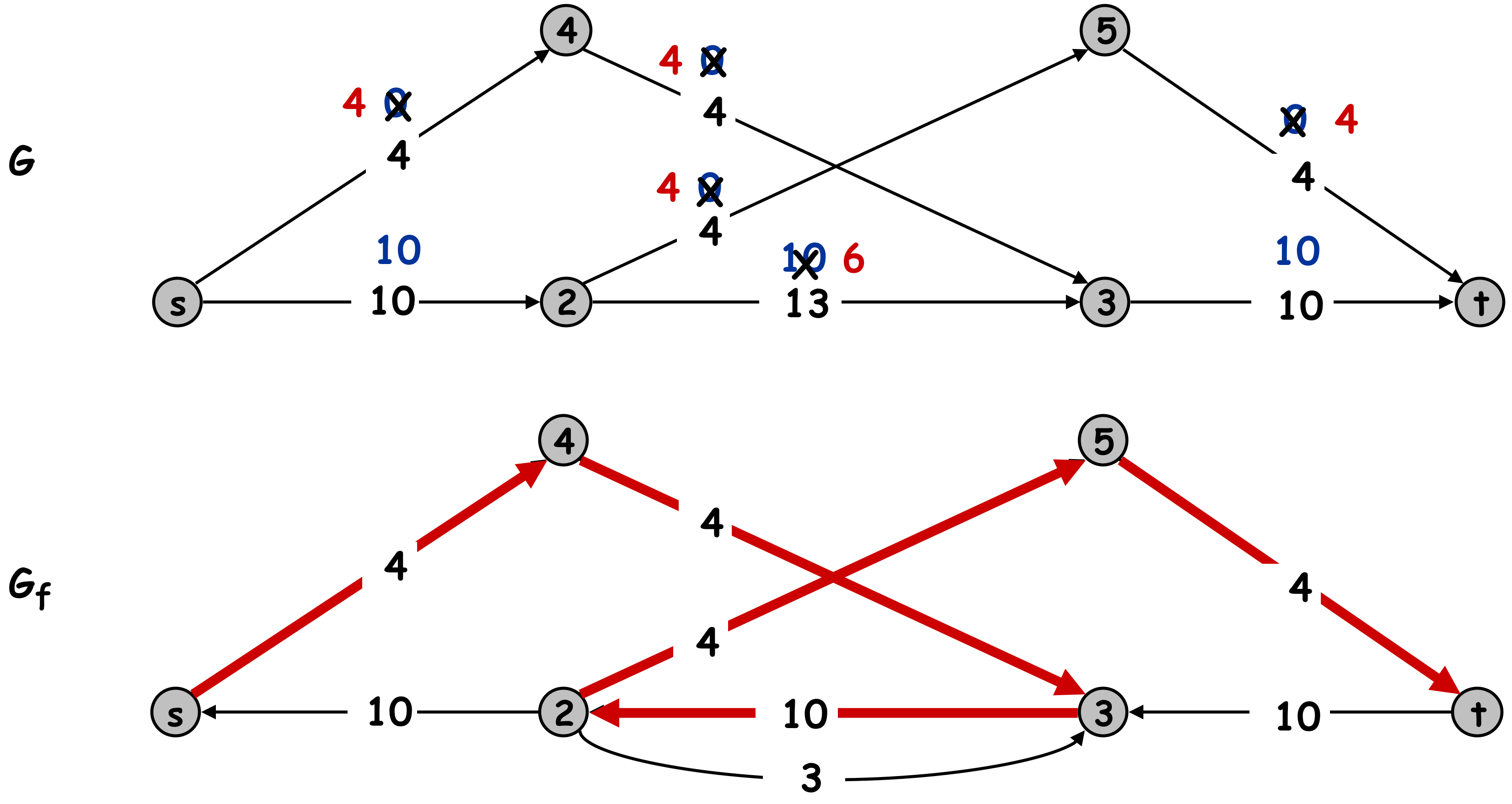
G



Augmenting Path

Augmenting path = path in residual graph

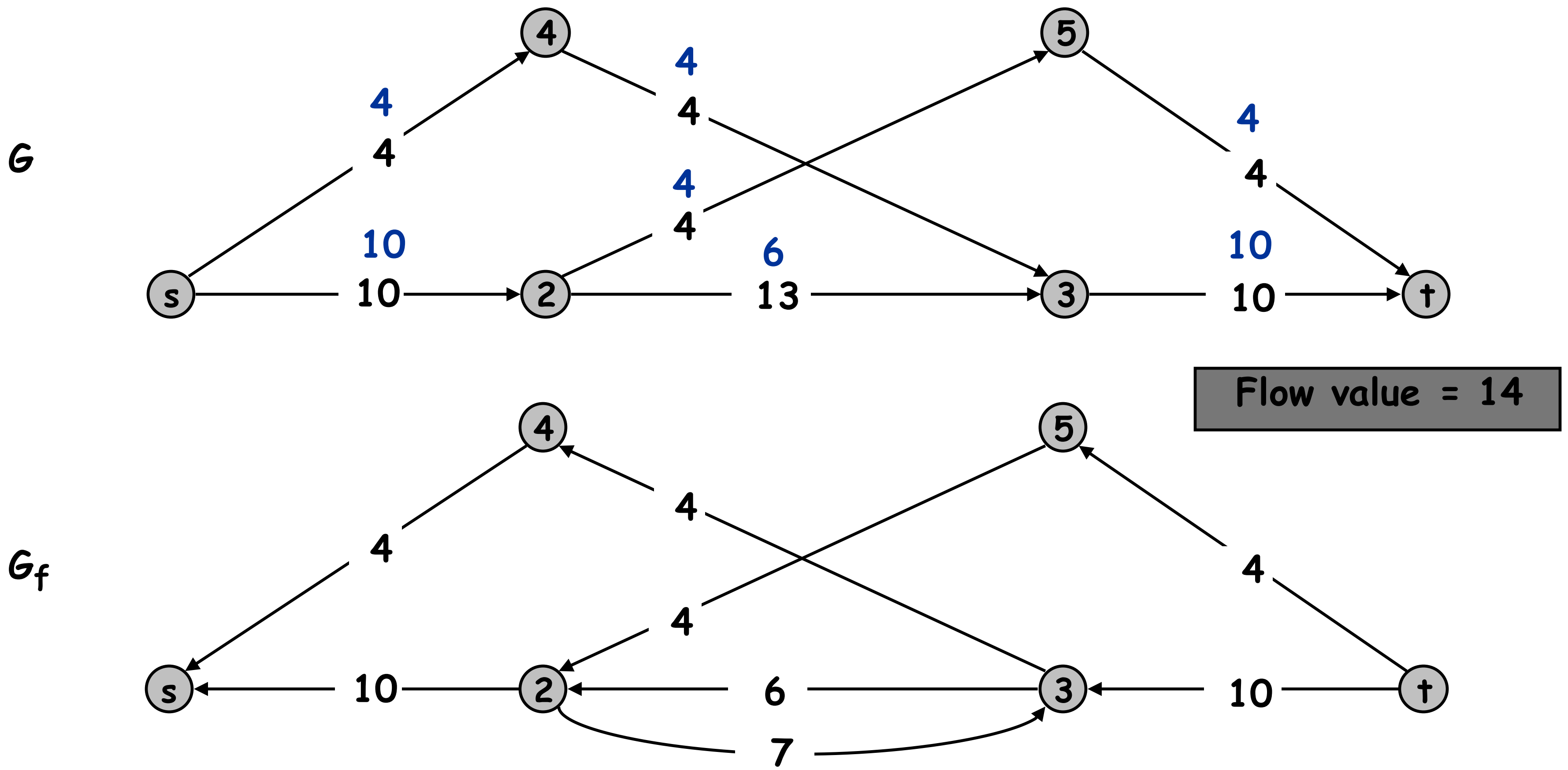
- Allows to undo some flow units from current solution
- And produce a flow of higher value



Augmenting Path

Augmenting path = path in residual graph.

- Max flow $\Leftrightarrow$ no augmenting paths ???



Augmenting Path Algorithm

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

Bottleneck is the minimum residual capacity of any edge in P

forward edge

reverse edge

```
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
}
```

Max flow - Min cut

[Ford, Fulkerson ’56]:
Theorem 1 (algorithm correctness): A feasible flow is optimal if and only if there is no augmenting path (i.e., no s-t path in the residual graph)

Theorem 2 (the max-flow min-cut theorem): For any flow graph G = (V, E) with capacities on its edges,
value of max flow = capacity of min s-t cut

We will prove both theorems together

Max flow - Min cut

Proof sketch:

Let f be a feasible flow computed by the algorithm. We prove that the following are equivalent:

(i) The flow f is optimal

(ii) There is no augmenting path with respect to f (i.e., no s-t path in the residual graph)

(iii) There exists a cut (A, B) such that v(f) = cap(A, B)

Max flow - Min cut

Proof sketch:

(i) $\Rightarrow$ (ii)

trivial, if there was an augmenting path, we would increase the flow and f would not be optimal

(ii) $\Rightarrow$ (iii)

- Let f be a flow with no augmenting paths
- Let A be the set of vertices reachable from s in the residual graph G_f
- Let B := V \ A
- By definition of A, $s \in A$
- By our assumption on f (no augmenting paths), $t \notin A$
- Hence (A, B) is a valid s-t cut

Max flow - Min cut

Proof sketch:

(ii) $\Rightarrow$ (iii) cont'd

- Claim 1: for an edge e = (u, v) with u $\in$ A and v $\in$ B, f(e) = c(e)
  - Otherwise, v is reachable in G_f from s (since u $\in$ A)
- Claim 2: for an edge e = (u, v) with u $\in$ B and v $\in$ A, f(e) = 0
  - Otherwise, there is a backward edge (v, u) in G_f, and hence u is reachable from s

$$\begin{aligned} v(f) &= \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to A}} f(e) \quad \text{(From Lemma 1)} \\ &= \sum_{e \text{ out of } A} c(e) \\ &= cap(A, B) \end{aligned}$$

(iii) $\Rightarrow$ (i)

- follows by the Corollary 2 on certificates of optimality

Running time

Assumption: Assume all capacities are integers

Claim 1: All flow values and residual capacities are integers throughout the execution of the algorithm

Claim 2: In every iteration of the while loop, the flow increases by at least 1 unit

Claim 3: Let $C = \sum_{(s,u) \in E} c(s,u)$. Then max flow ≤ C

Total running time: O((m+n) C) pseudopolynomial algorithm

Corollary: If all capacities are 0 or 1, then running time is O(mn)

- important special case in some applications

Improving the running time

Worst case scenarios:

With integer capacities, the algorithm may need to do C augmentations

- If capacities are irrational, algorithm not even guaranteed to terminate!

Some improvements

[Edmonds-Karp 1972, Dinitz 1970]:

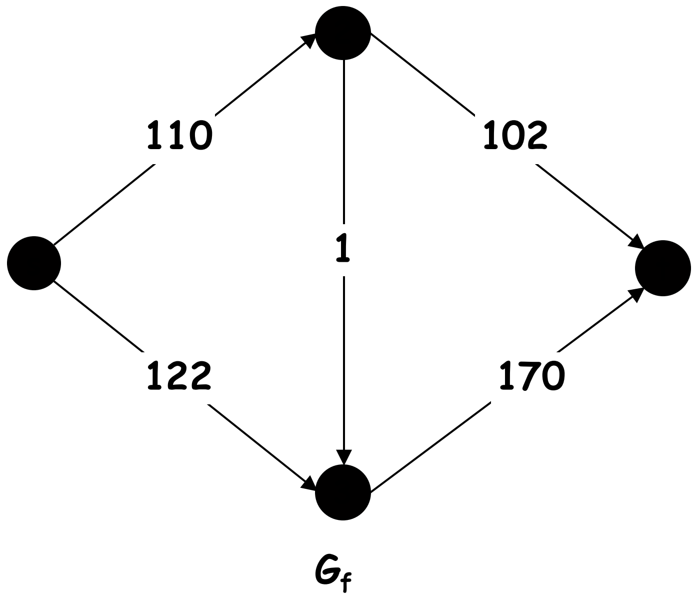
Choose augmenting paths with:

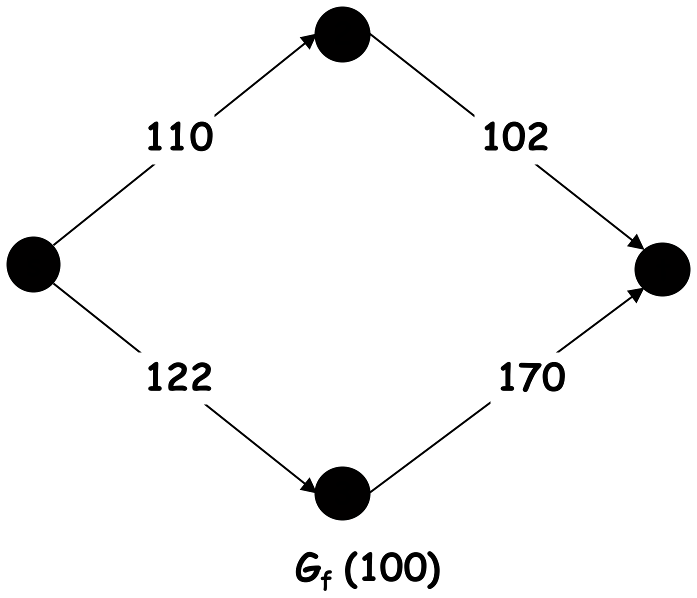
- Max bottleneck capacity
- Sufficiently large bottleneck capacity
- Fewest number of edges

Capacity Scaling

Intuition: Choosing a path with the highest bottleneck capacity increases flow by max possible amount.

- Actually, don't worry about finding the exact highest bottleneck path (this may slow down the algorithm)
- Maintain a scaling parameter Δ.
- Let $G_f(\Delta)$ be the subgraph of the residual graph consisting only of arcs with capacity at least Δ





Capacity Scaling

```
Scaling-Max-Flow(G, s, t, c) {
   foreach e $\in$ E  f(e) $\leftarrow$ 0
   $\Delta$ $\leftarrow$ smallest power of 2 less than or equal to C
   $G_f$ $\leftarrow$ residual graph

   while ($\Delta$ $\geq$ 1) {
      $G_f(\Delta)$ $\leftarrow$ $\Delta$-residual graph
      while (there exists an augmenting path P in $G_f(\Delta)$) {
         f $\leftarrow$ augment(f, c, P)
         update $G_f(\Delta)$
      }
      $\Delta$ $\leftarrow$ $\Delta$ / 2
   }
   return f
}
```

Correctness and running time

Assume integer capacities

Correctness:

- Eventually, when $\Delta = 1 \Rightarrow G_f(\Delta) = G_f$
- Hence the algorithm stops when there are no s-t paths in G_f
- The flow must be optimal by the correctness analysis of Ford-Fulkerson

Running time analysis

Lemma 1: The outer while loop runs for $1 + \lceil \log_2 C \rceil$ iterations

Proof: Initially $C \leq \Delta < 2C$. Δ decreases by a factor of 2 in each iteration of the outer while loop

Correctness and running time

Assume integer capacities

Running time analysis (cont'd)

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

Proof: do it as an exercise

Lemma 3: There are at most 2m augmentations per scaling phase

Proof: Consider the beginning of a scaling phase with parameter Δ

- Let f be the flow at the end of the previous scaling phase
- Lemma 2 $\Rightarrow$ $v(f^*) \leq v(f) + m\,(2\Delta)$ [previous is twice the current Δ]
- Each augmentation in a Δ-phase increases v(f) by at least Δ

Theorem: The capacity scaling max-flow algorithm finds a max flow in O(m log C) augmentations. It can be implemented to run in $O(m^2 \log C)$ time

Application to Matching problems

Matching Problems

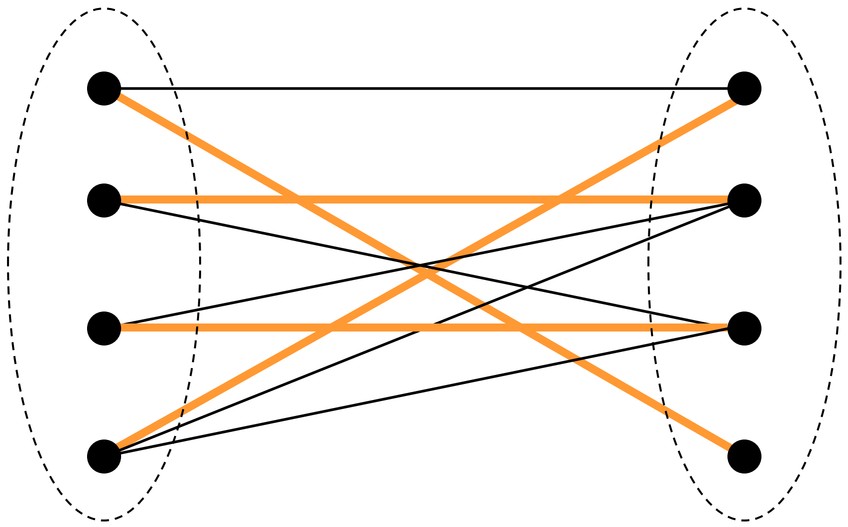
Consider an undirected graph G = (V, E)

Definition: A matching M is a collection of edges $M \subseteq E$, such that no 2 edges share a common vertex

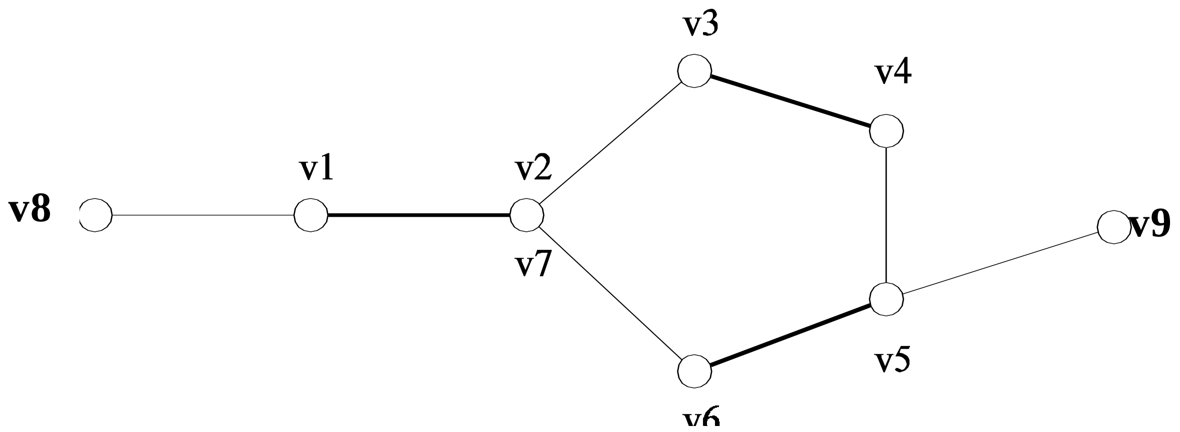
Given a matching M, a vertex u is called *matched* if there exists an edge $e \in M$ such that e has u as one of its endpoints

Matching Problems

Examples



a matching in a bipartite graph



A matching in general graphs (vertex v8 is unmatched)

Matching Problems

Types of matching problems that arise in optimization:

Maximal matching: find a matching where no more edges can be added

Maximum matching: find a matching with the maximum possible number of edges

Perfect matching: find a matching where every vertex is matched (if one exists)

Maximum weight matching: given a weighted graph, find a matching with maximum possible total weight

Minimum weight perfect matching: given a weighted graph, find a perfect matching with minimum cost

All the above problems can be solved in polynomial time (several algorithms and publications over the last decades)

Matching Problems

Trivial algorithm for maximal matching:

- Start from the empty set of edges
- Keep adding edges that do not have common endpoints to the current solution
- Stop when it is not possible to add an edge that does not have any common endpoint with the edges already picked
- The selected set of edges forms a maximal matching

More sophisticated algorithms are required for maximum matching and perfect matching

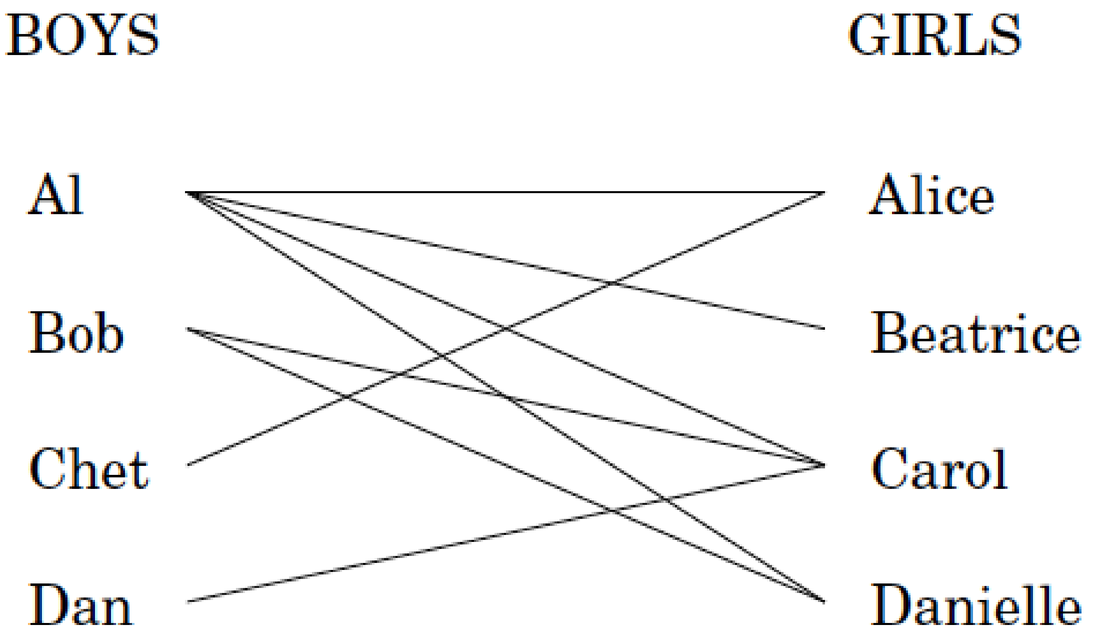
[Edmonds '65]: first algorithm for maximum matching in general graphs

- Also first mention of polynomial time solvability as a measure of efficiency

Matching in Bipartite Graphs

An interesting special case for matching problems:

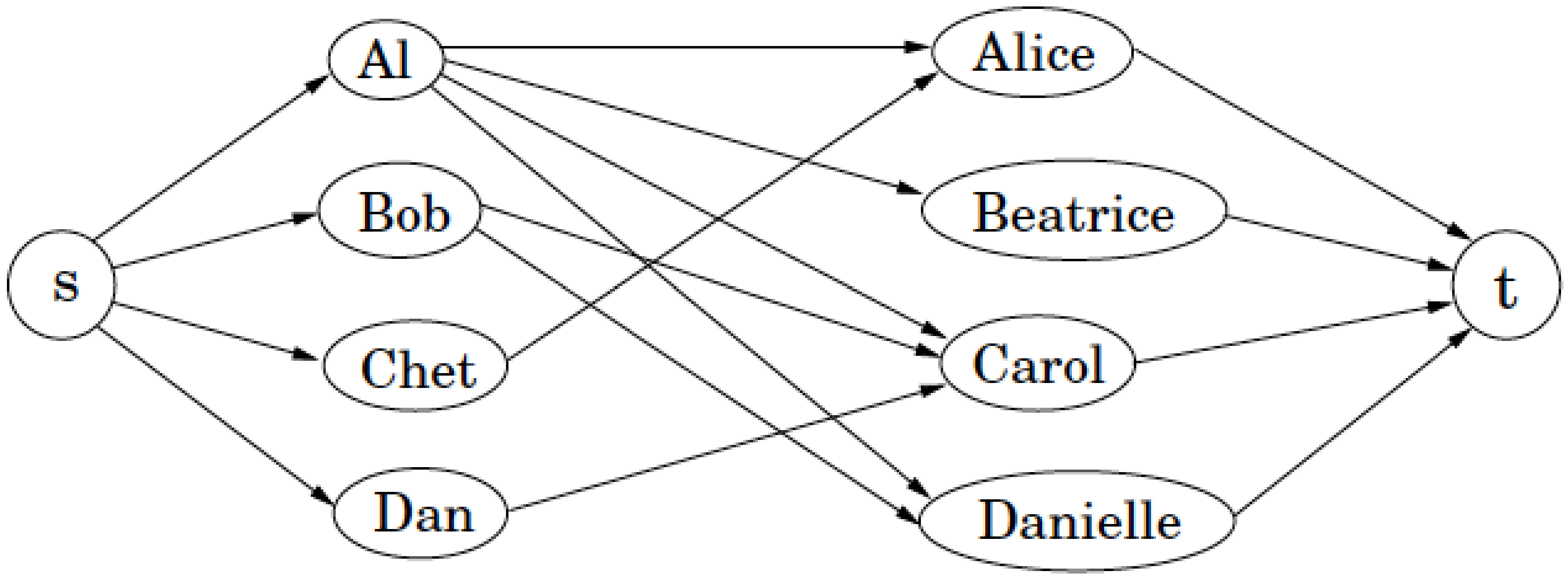
A graph G = (V, E) is called bipartite if V can be partitioned into 2 sets V_1, V_2 such that all edges connect a vertex from V_1 with a vertex from V_2



Q: How can we find a maximum matching in a bipartite graph?

Matching in Bipartite Graphs

We can reduce this to a max-flow problem



- Orient all edges from left to right
- Add a source node s, connect it to all of V_1
- Add a sink node t, connect all of V_2 to t
- Capacities: set them to 1 for all edges

Matching in Bipartite Graphs

Hence:
- a maximum matching for bipartite graphs can be computed in polynomial time
- The graph has a perfect matching if and only if the max flow in the modified graph equals n

But wait a minute…
- What if the max flow assigns a flow of 0.65 to an edge?

- Fortunately this can be avoided

Theorem: If all the capacities of a graph are integral, then there is an integral optimal flow and our algorithms compute such an integral optimal flow