

# Graduate Design and Analysis of Algorithms

Slides from the Video Lecture  
*Flipped Class Offering*

*Instructor Krishna V Palem*

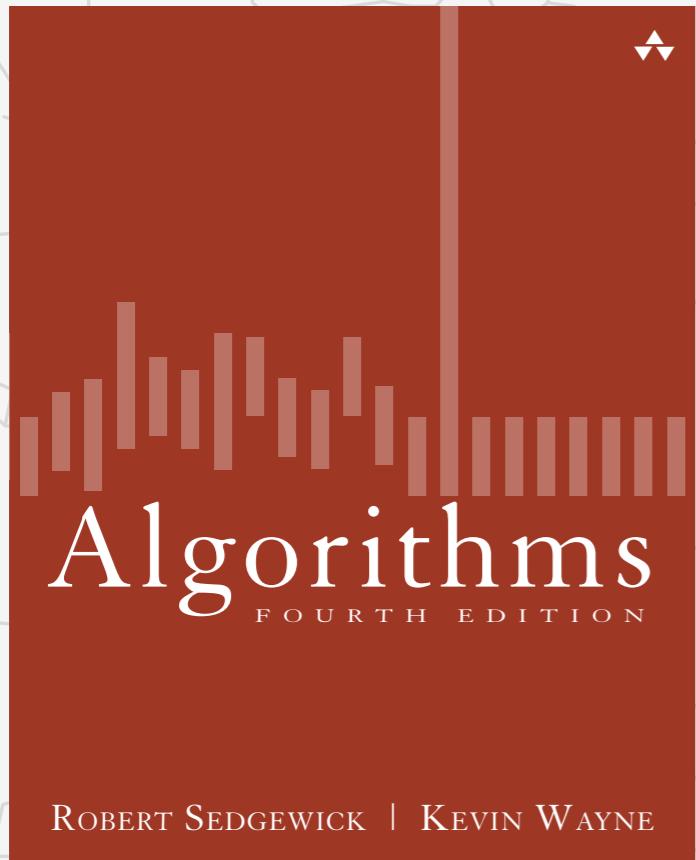
*Course CS 582/ ELEC 512*

*Fall 2016*

*Please do not copy or distribute without permission  
from the authors*

# Algorithms

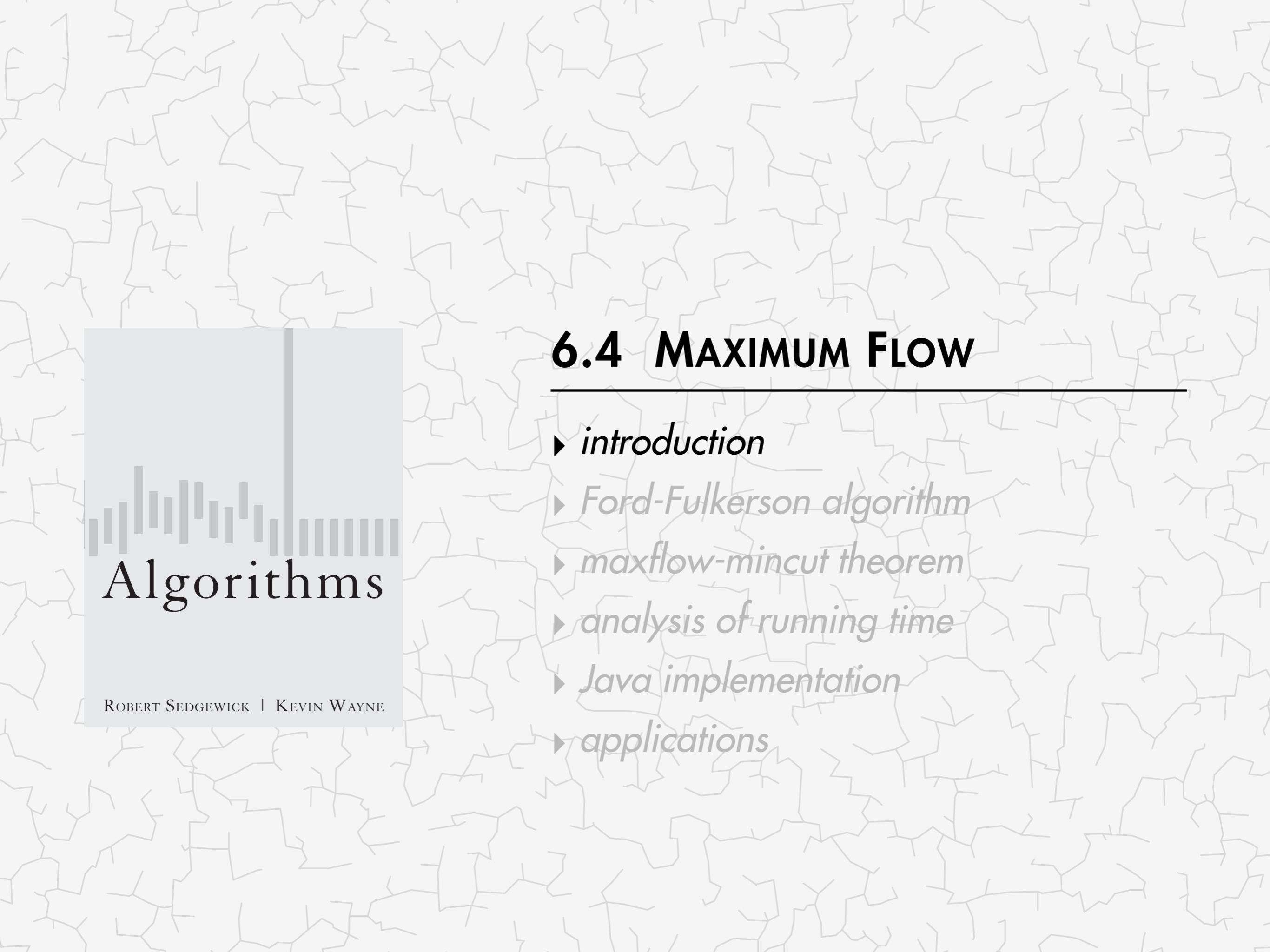
ROBERT SEDGEWICK | KEVIN WAYNE



## 6.4 MAXIMUM FLOW

---

- ▶ *introduction*
- ▶ *Ford-Fulkerson algorithm*
- ▶ *maxflow-mincut theorem*
- ▶ *analysis of running time*
- ▶ *Java implementation*
- ▶ *applications*



# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

## 6.4 MAXIMUM FLOW

---

- ▶ *introduction*
- ▶ *Ford-Fulkerson algorithm*
- ▶ *maxflow-mincut theorem*
- ▶ *analysis of running time*
- ▶ *Java implementation*
- ▶ *applications*

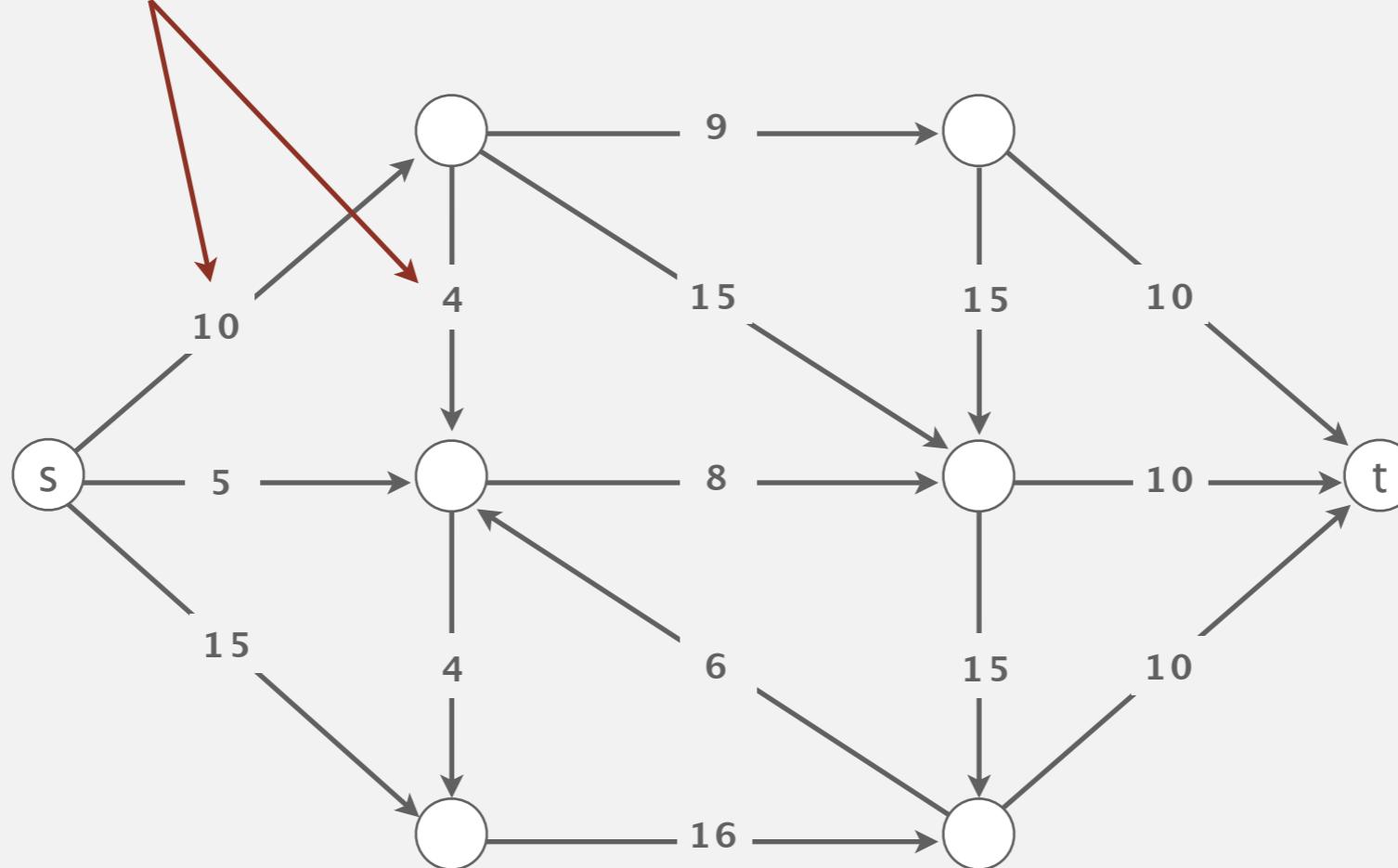
# Mincut problem

---

**Input.** An edge-weighted digraph, source vertex  $s$ , and target vertex  $t$ .

each edge has a  
positive capacity

capacity

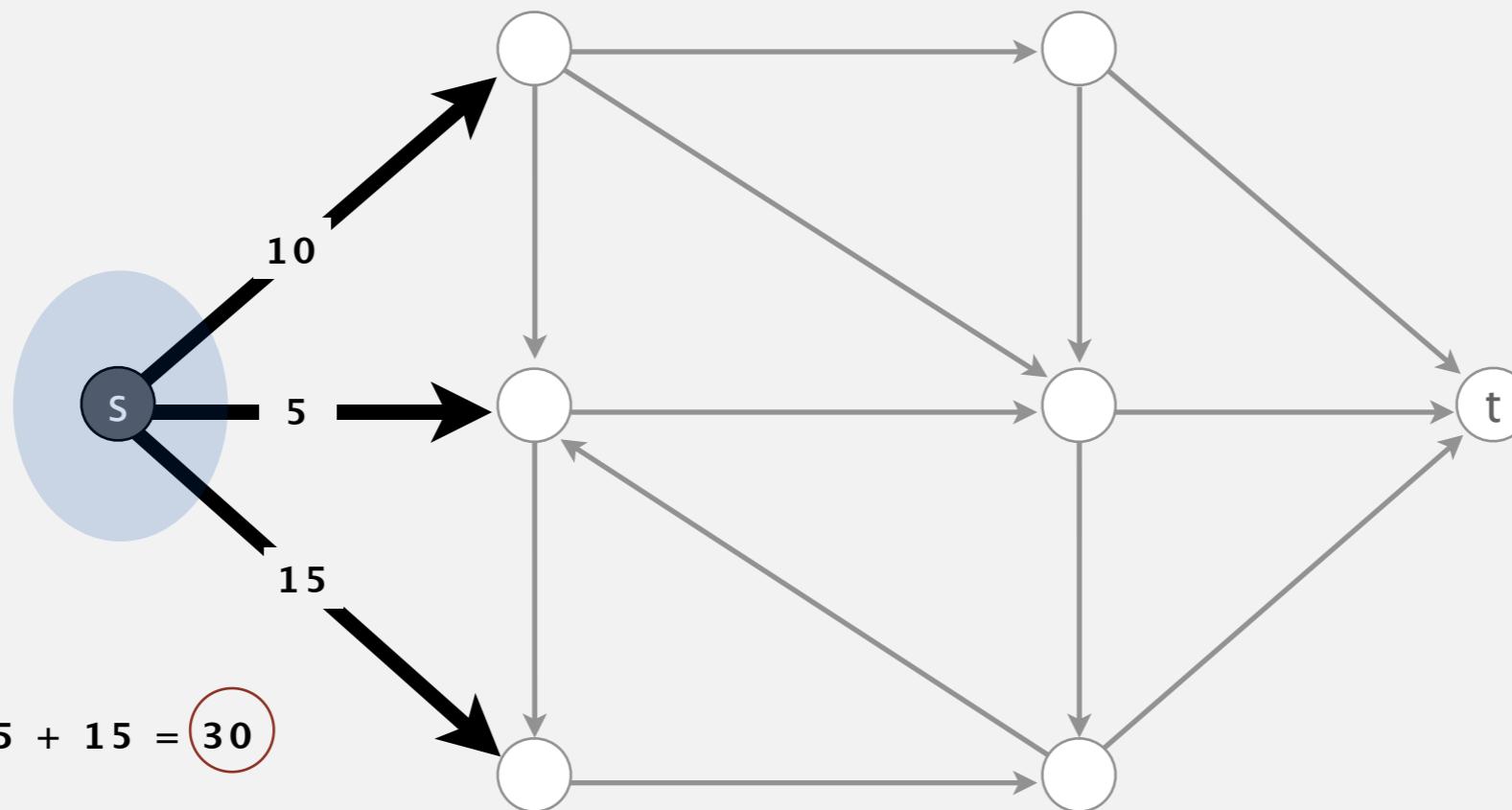


# Mincut problem

---

Def. A ***st-cut (cut)*** is a partition of the vertices into two disjoint sets, with  $s$  in one set  $A$  and  $t$  in the other set  $B$ .

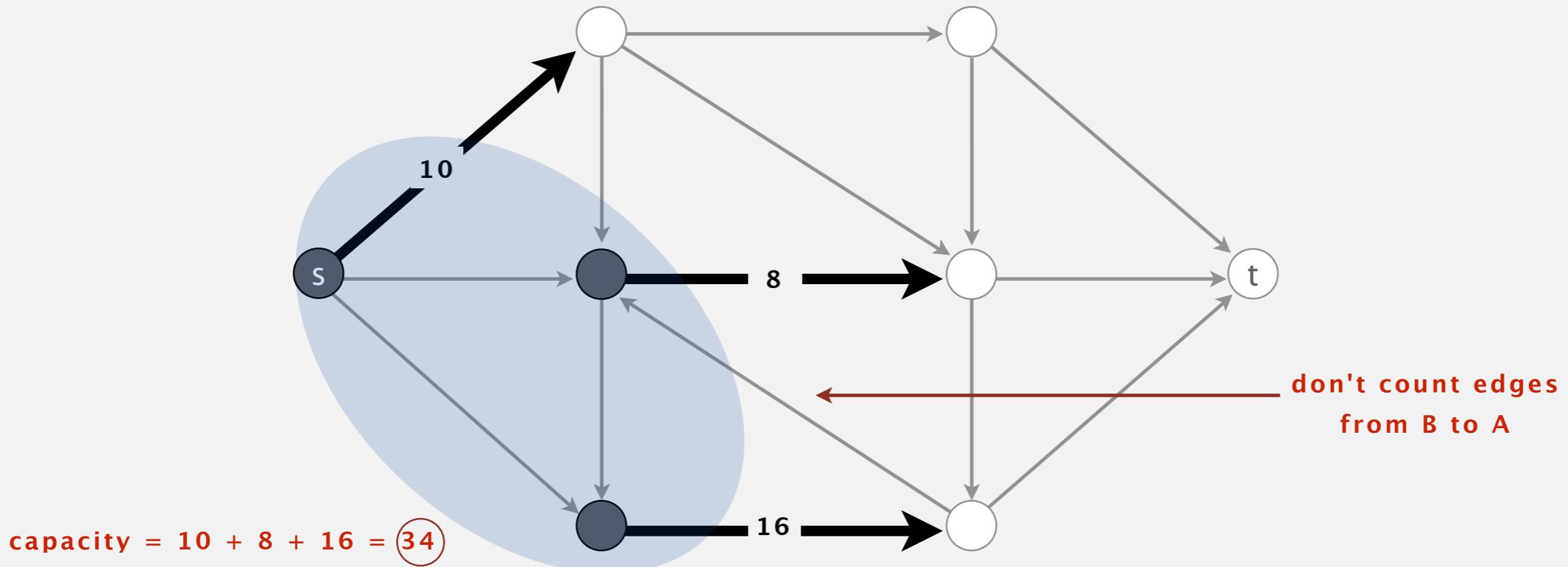
Def. Its **capacity** is the sum of the capacities of the edges from  $A$  to  $B$ .



# Mincut problem

Def. A ***st-cut (cut)*** is a partition of the vertices into two disjoint sets, with  $s$  in one set  $A$  and  $t$  in the other set  $B$ .

Def. Its **capacity** is the sum of the capacities of the edges from  $A$  to  $B$ .



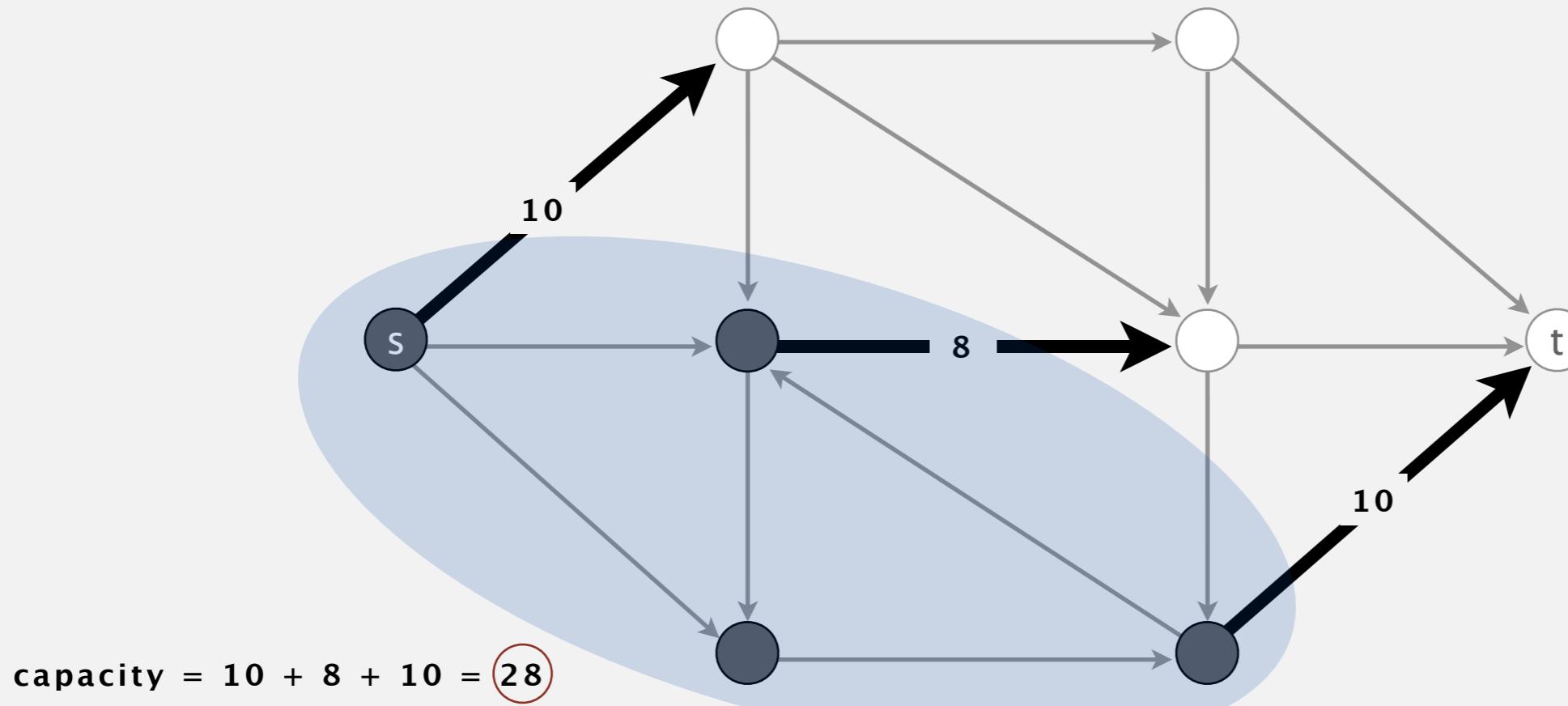
# Mincut problem

---

Def. A ***st-cut (cut)*** is a partition of the vertices into two disjoint sets, with  $s$  in one set  $A$  and  $t$  in the other set  $B$ .

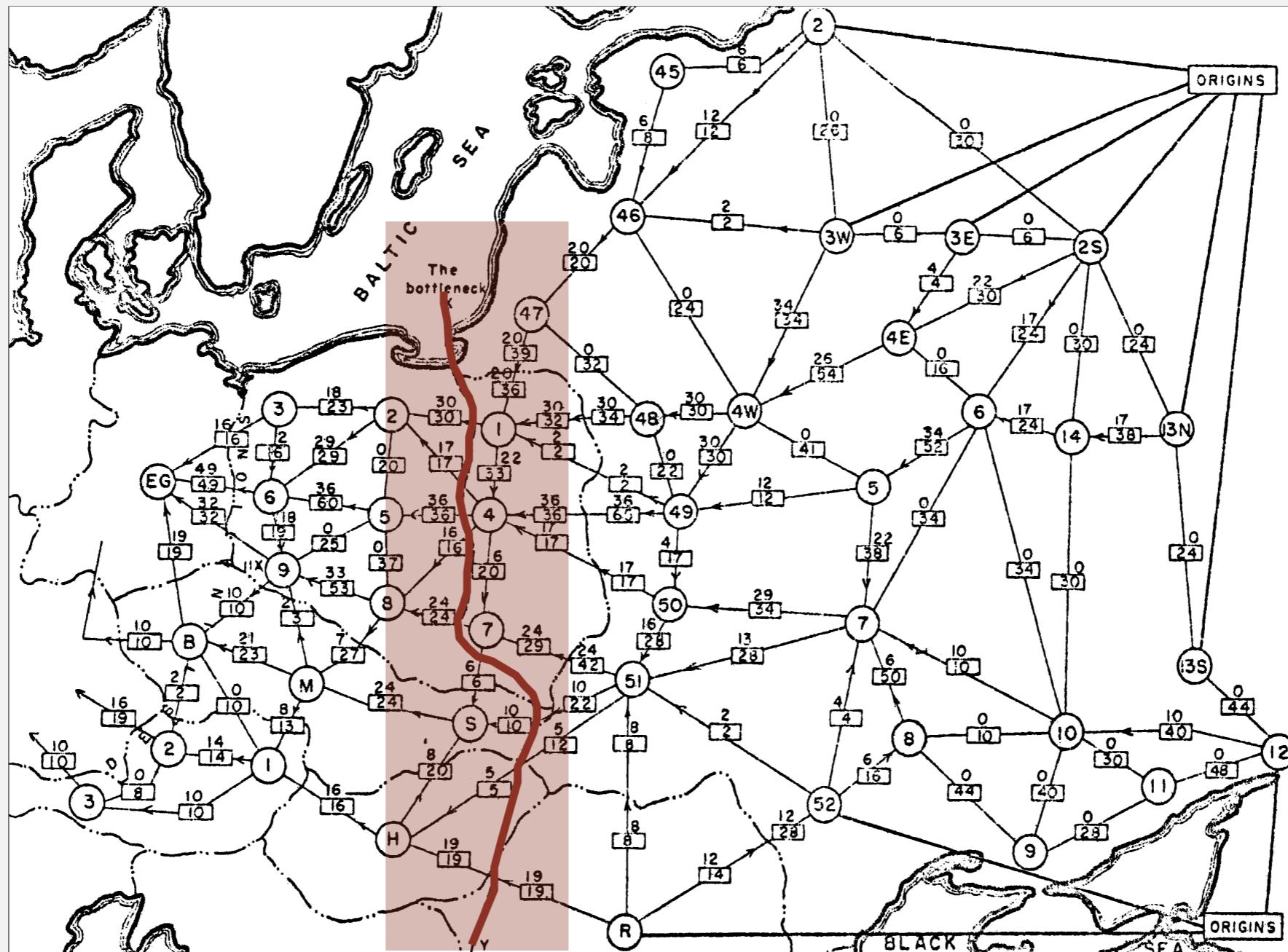
Def. Its **capacity** is the sum of the capacities of the edges from  $A$  to  $B$ .

Minimum st-cut (mincut) problem. Find a cut of minimum capacity.



# Mincut 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)

# Potential mincut application (2010s)

---

Government-in-power's goal. Cut off communication to set of people.

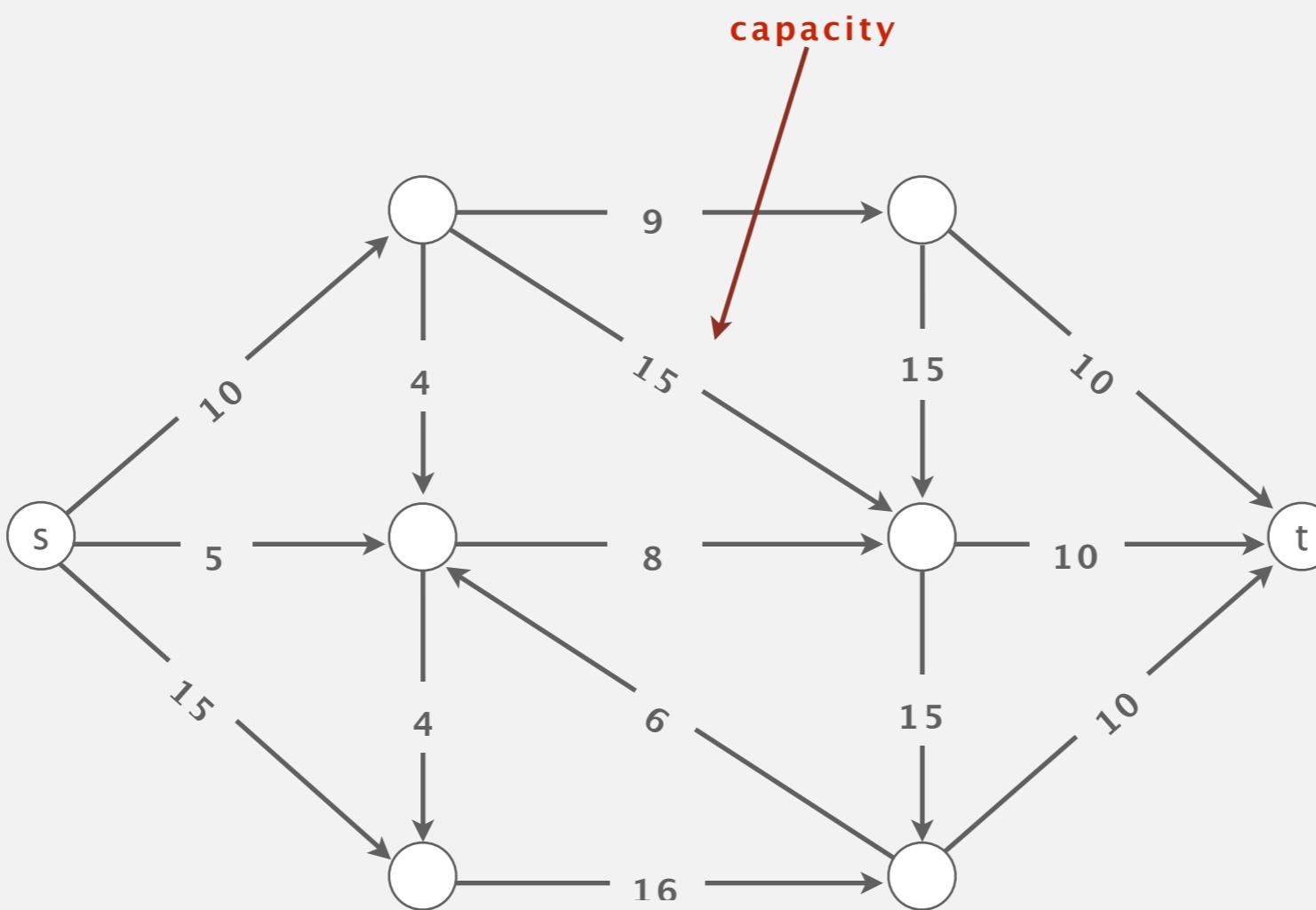


# Maxflow problem

---

**Input.** An edge-weighted digraph, source vertex  $s$ , and target vertex  $t$ .

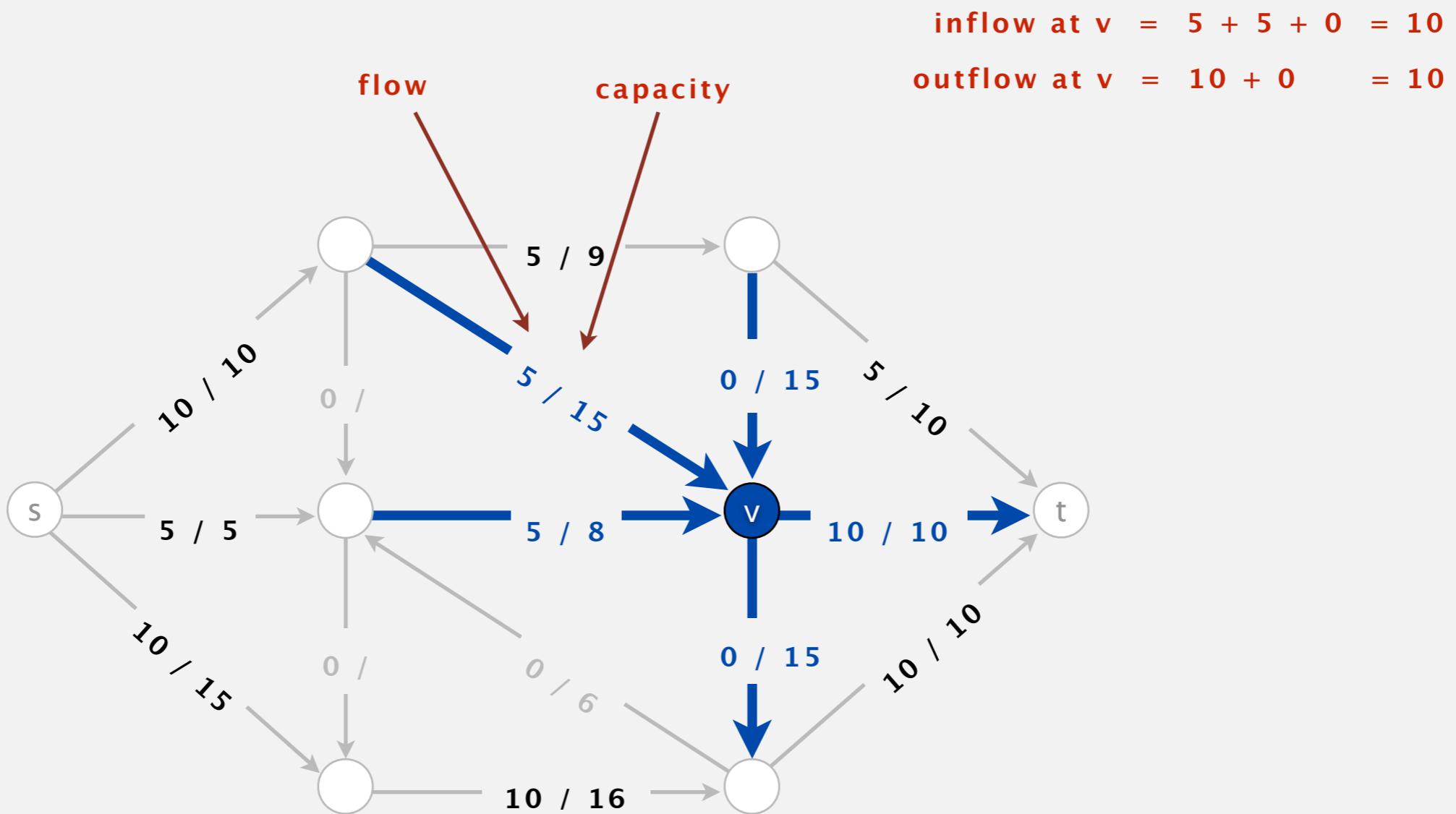
each edge has a  
positive capacity



# Maxflow problem

Def. An *st-flow (flow)* is an assignment of values to the edges such that:

- Capacity constraint:  $0 \leq$  edge's flow  $\leq$  edge's capacity.
- Local equilibrium: inflow = outflow at every vertex (except  $s$  and  $t$ ).



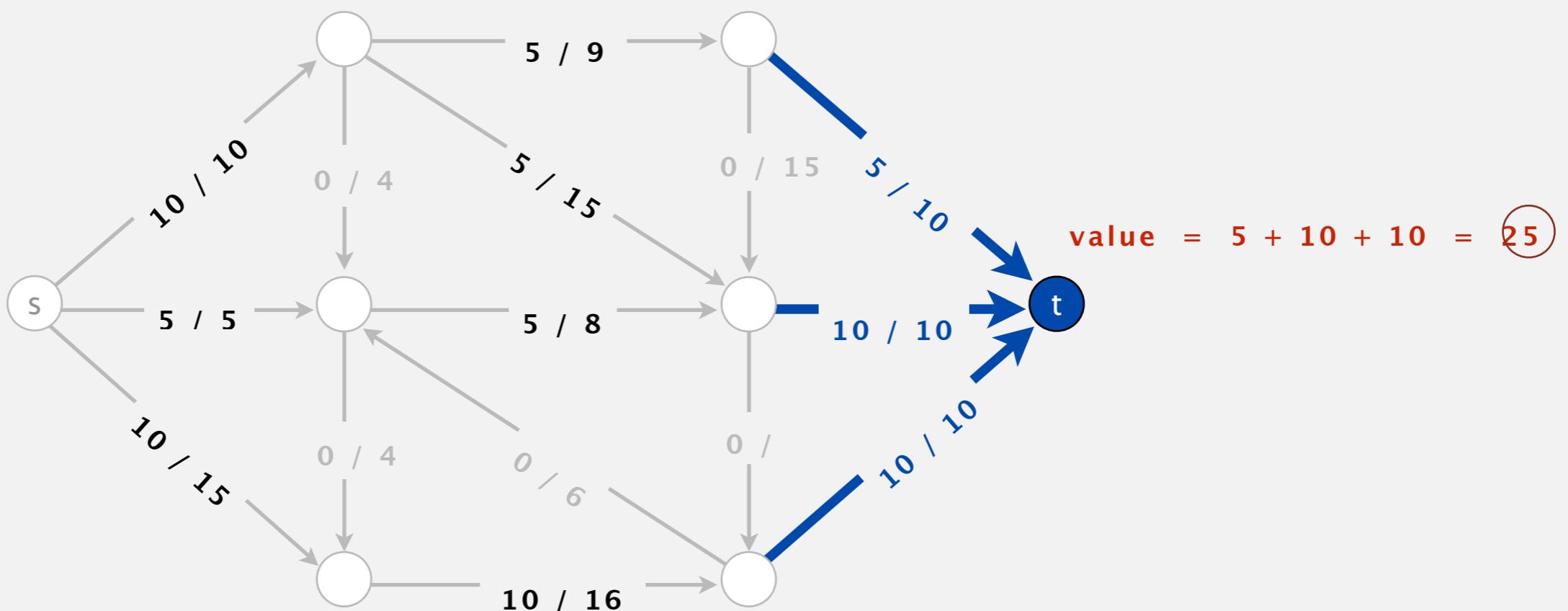
# Maxflow problem

Def. An *st-flow (flow)* is an assignment of values to the edges such that:

- Capacity constraint:  $0 \leq$  edge's flow  $\leq$  edge's capacity.
- Local equilibrium: inflow = outflow at every vertex (except  $s$  and  $t$ ).

Def. The **value** of a flow is the inflow at  $t$ .

we assume no edges point to  $s$  or from  $t$



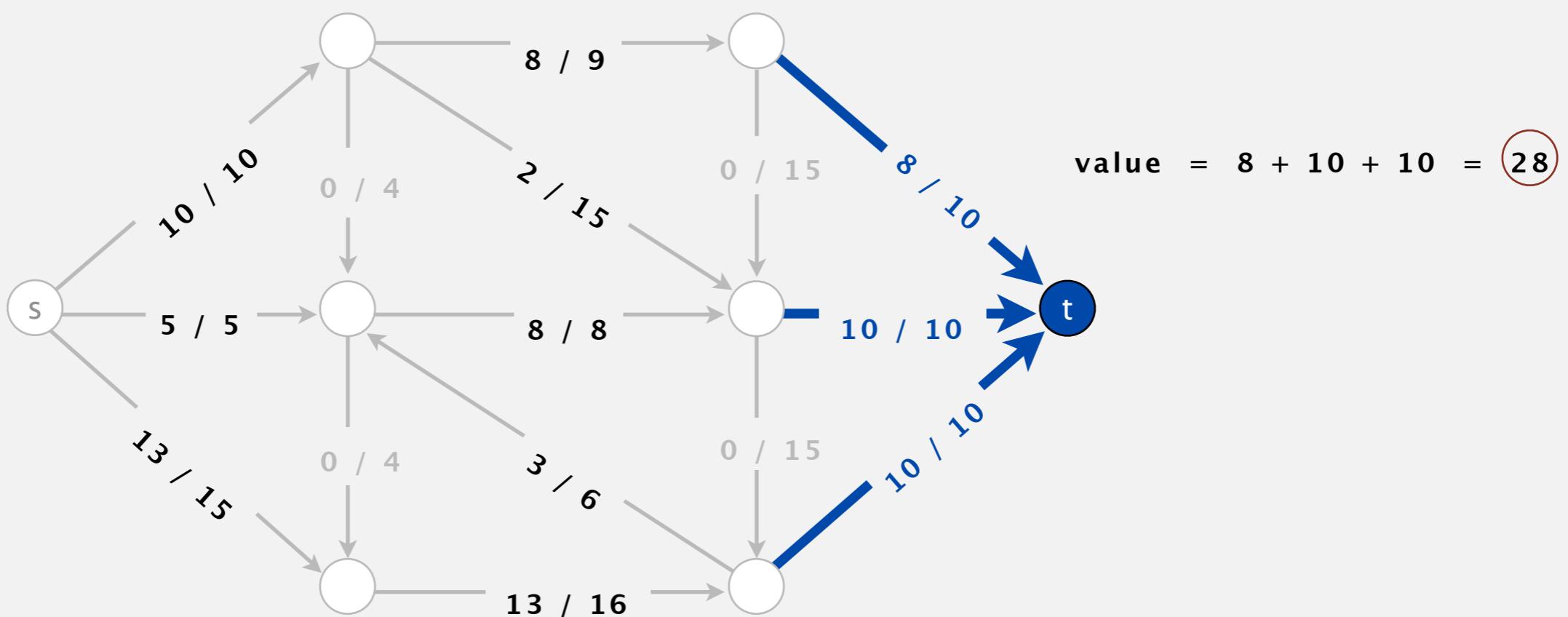
# Maxflow problem

Def. An *st-flow (flow)* is an assignment of values to the edges such that:

- Capacity constraint:  $0 \leq$  edge's flow  $\leq$  edge's capacity.
- Local equilibrium: inflow = outflow at every vertex (except  $s$  and  $t$ ).

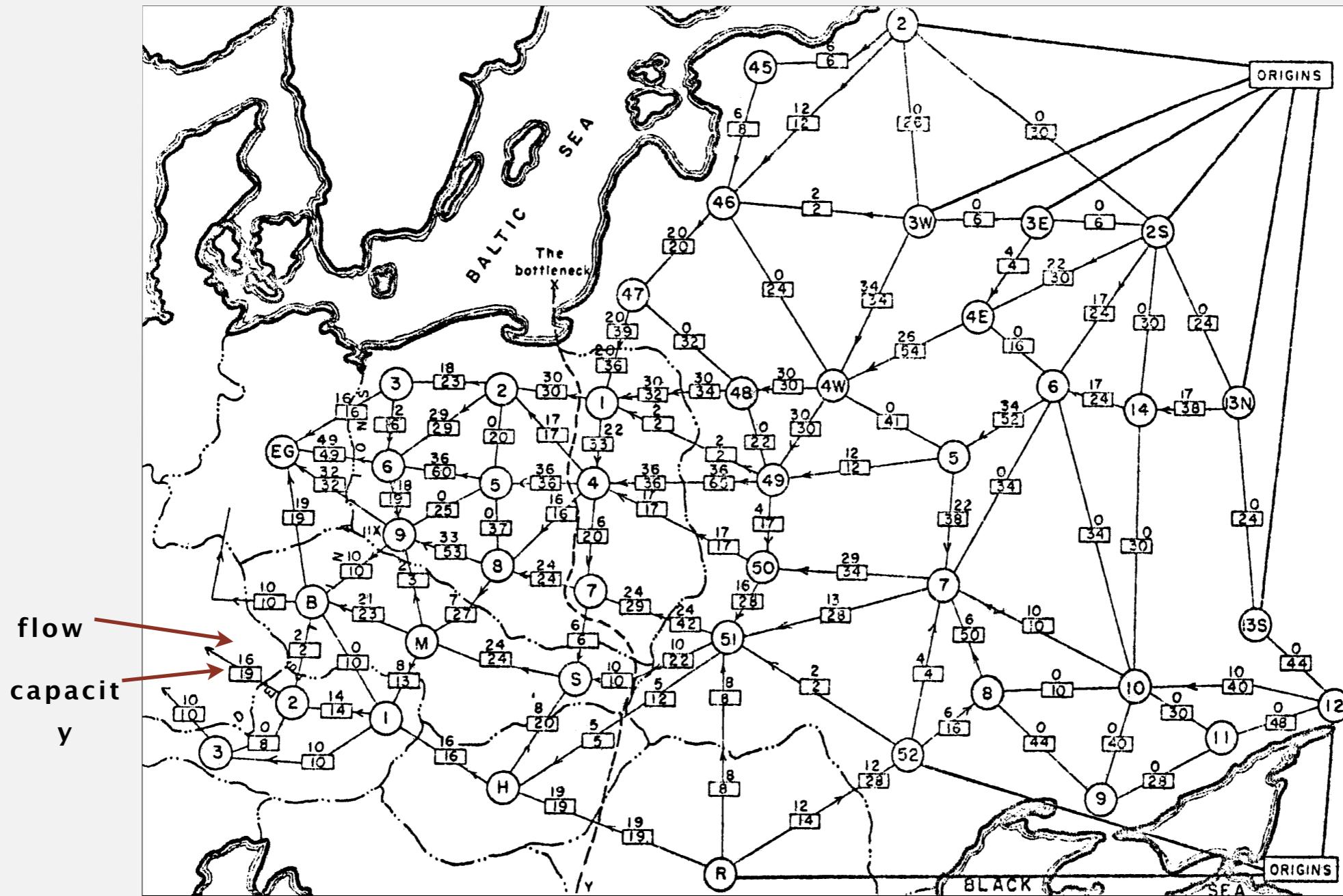
Def. The *value* of a flow is the inflow at  $t$ .

Maximum st-flow (maxflow) problem. Find a flow of maximum value.



# Maxflow application (Tolstoř 1930s)

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



rail network connecting Soviet Union with Eastern European countries

(map declassified by Pentagon in 1999)

# Potential maxflow application (2010s)

---

"Free world" goal. Maximize flow of information to specified set of people.



facebook graph

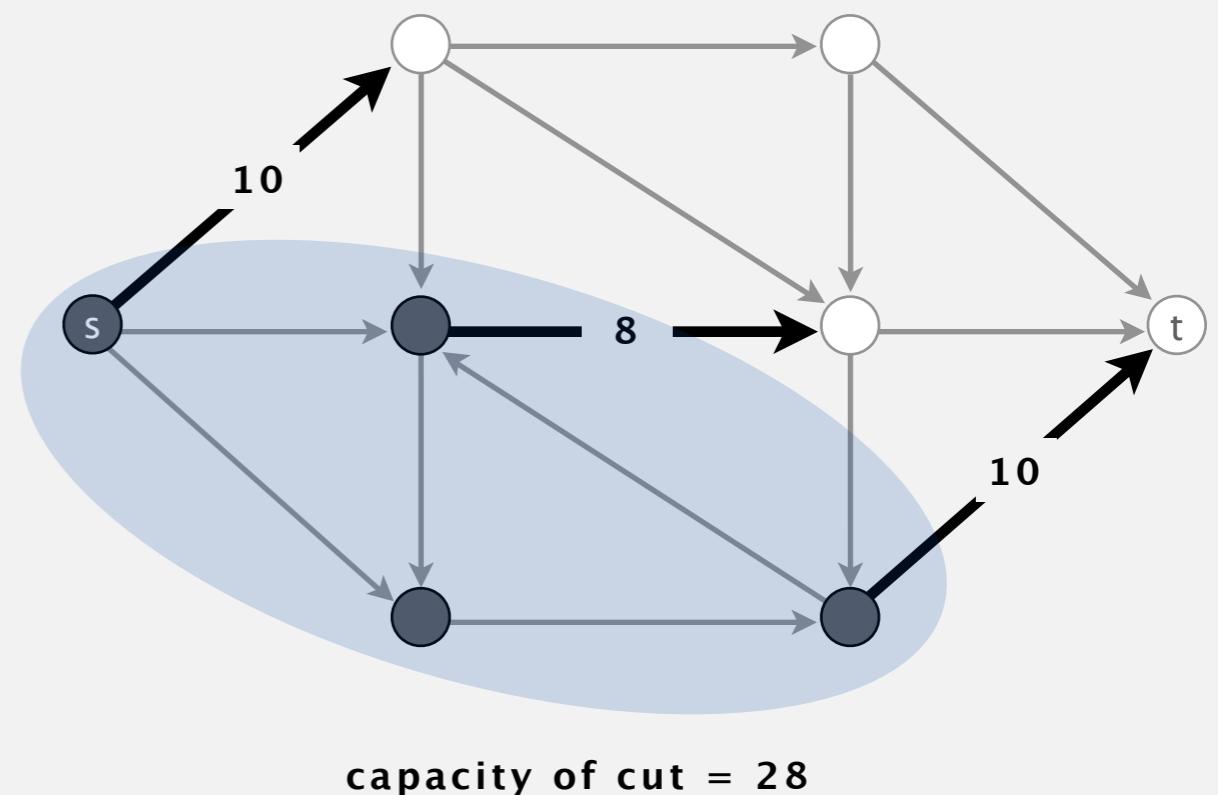
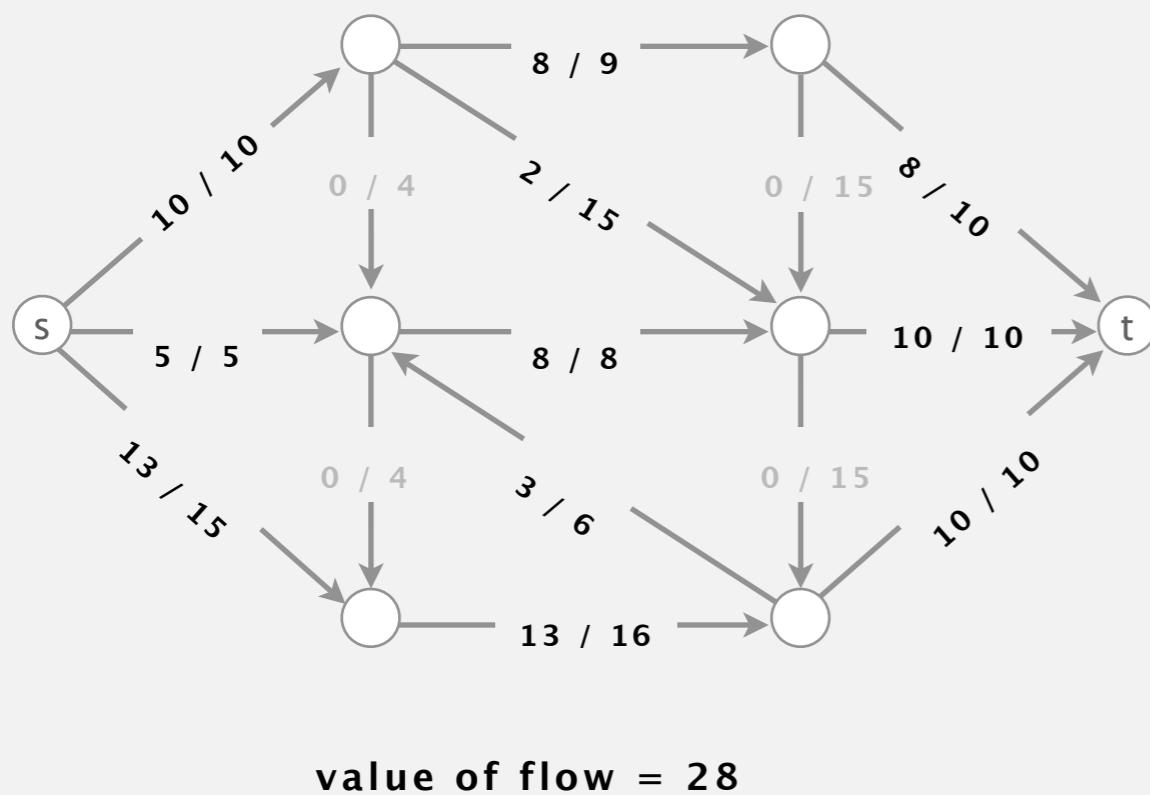
# Summary

---

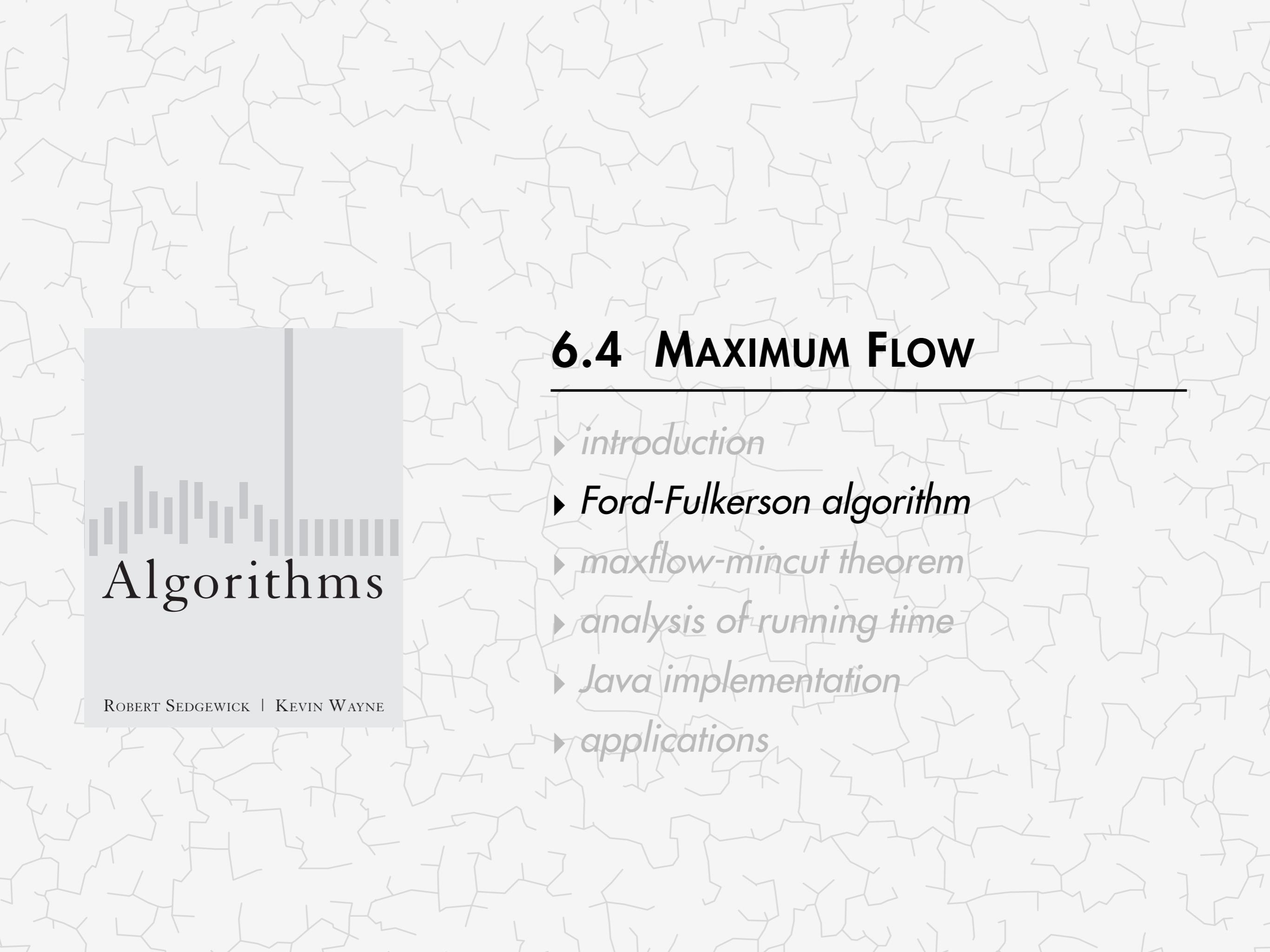
**Input.** A weighted digraph, source vertex  $s$ , and target vertex  $t$ .

**Mincut problem.** Find a cut of minimum capacity.

**Maxflow problem.** Find a flow of maximum value.



**Remarkable fact.** These two problems are dual!



# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

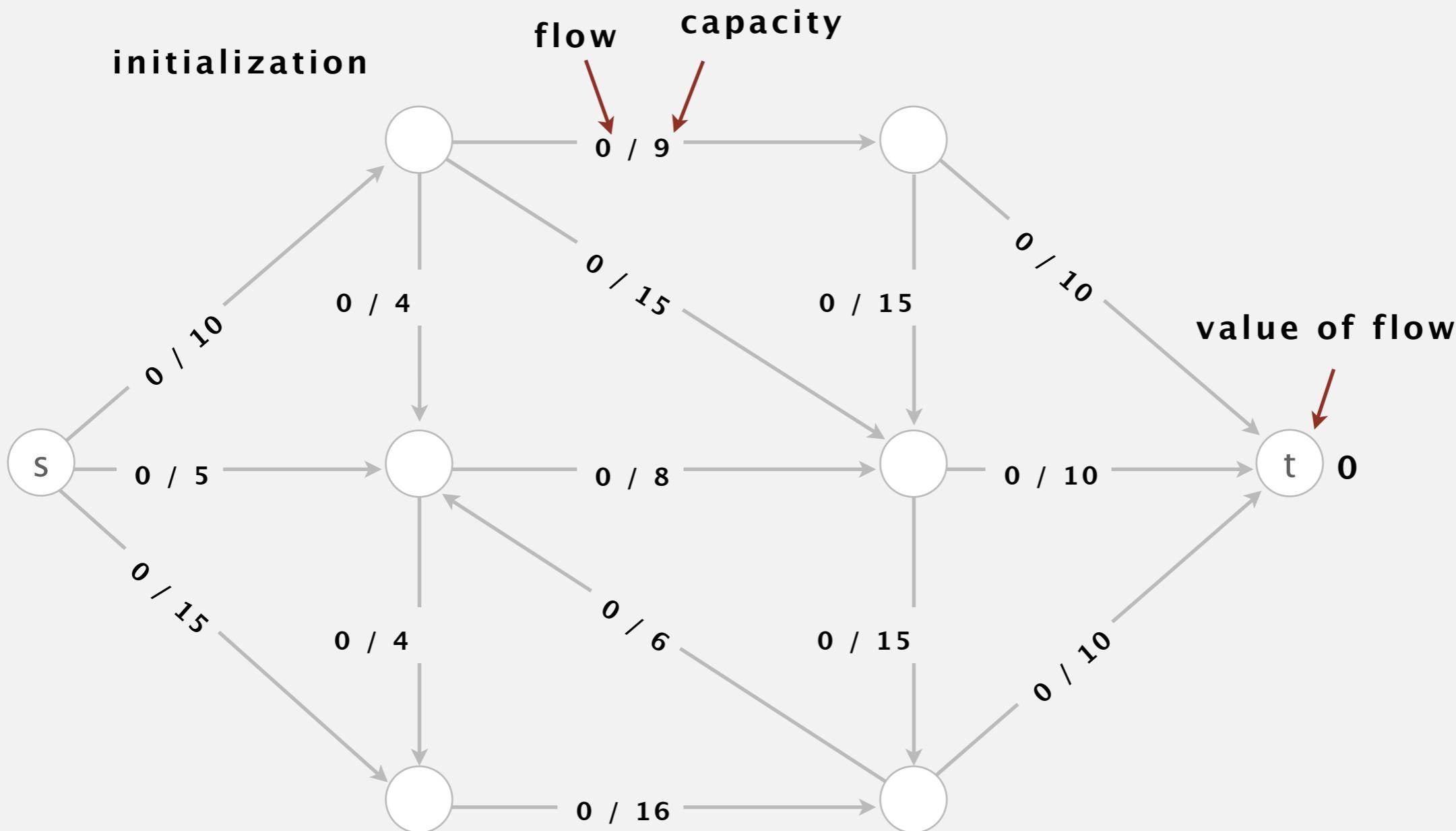
## 6.4 MAXIMUM FLOW

---

- ▶ *introduction*
- ▶ ***Ford-Fulkerson algorithm***
- ▶ *maxflow-mincut theorem*
- ▶ *analysis of running time*
- ▶ *Java implementation*
- ▶ *applications*

# Ford-Fulkerson algorithm

Initialization. Start with 0 flow.

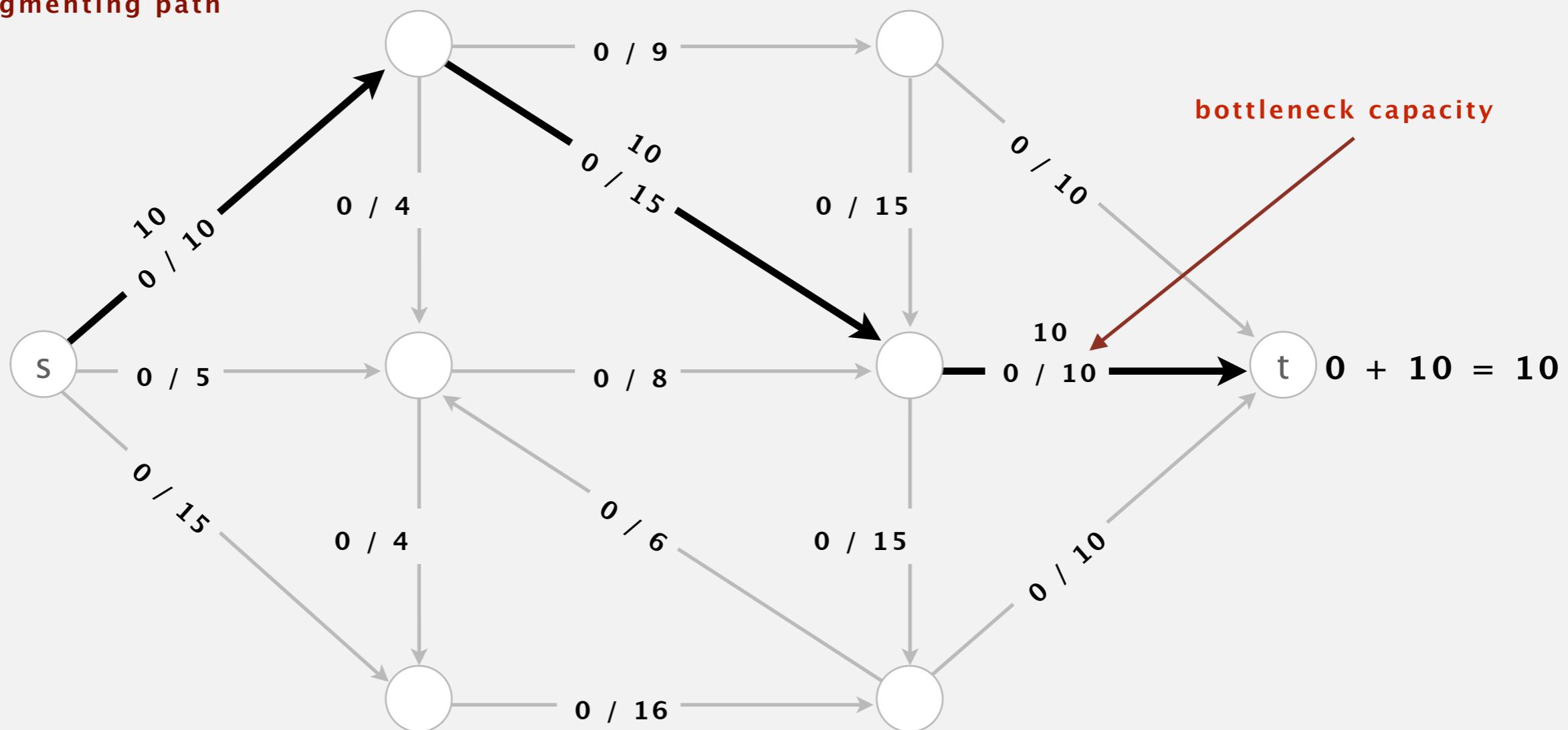


# Idea: increase flow along augmenting paths

## Augmenting path.

- Can increase flow on a forward edge (not full)
- Can decrease flow on backward edge (not empty)

1st Augmenting path

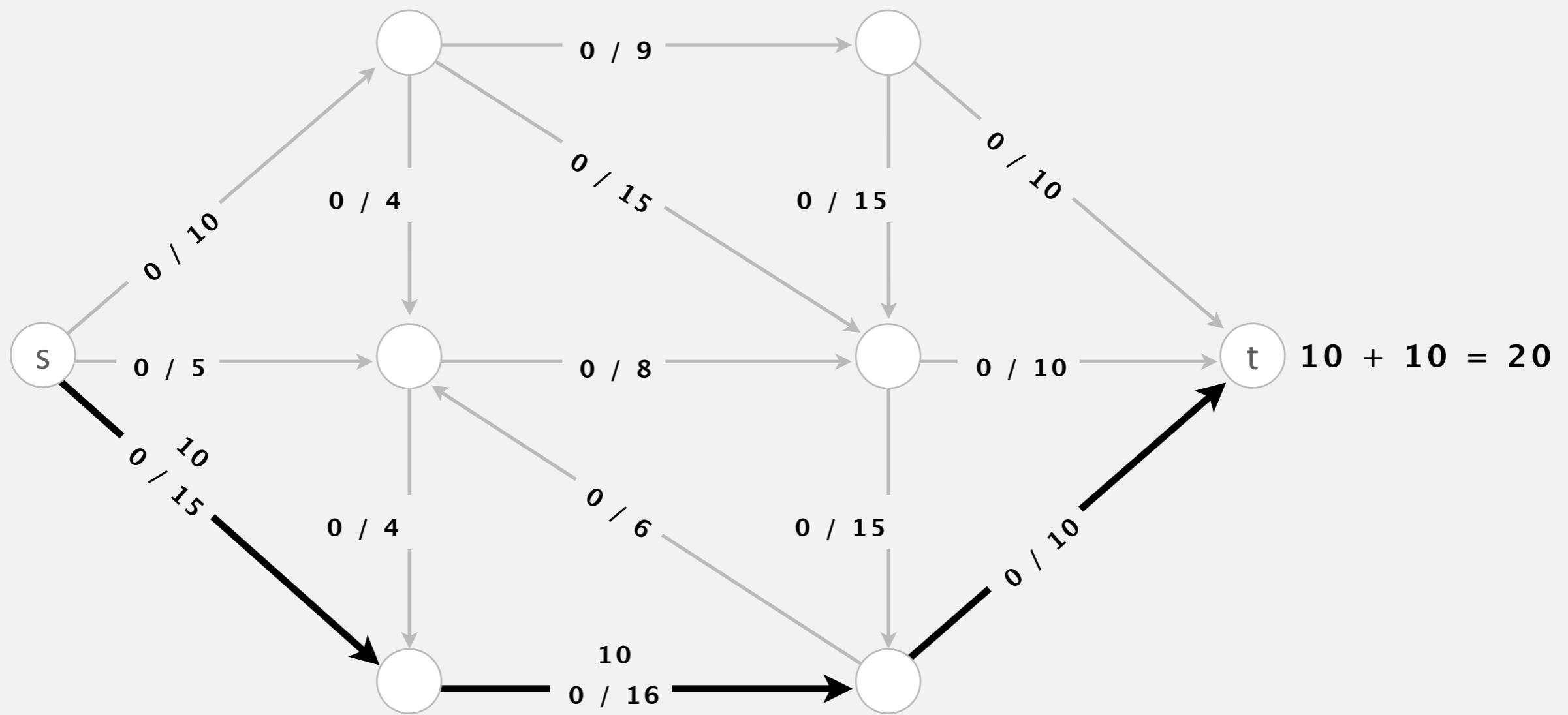


# Idea: increase flow along augmenting paths

## Augmenting path.

- Can increase flow on a forward edge (not full)
- Can decrease flow on backward edge (not empty)

2nd Augmenting path

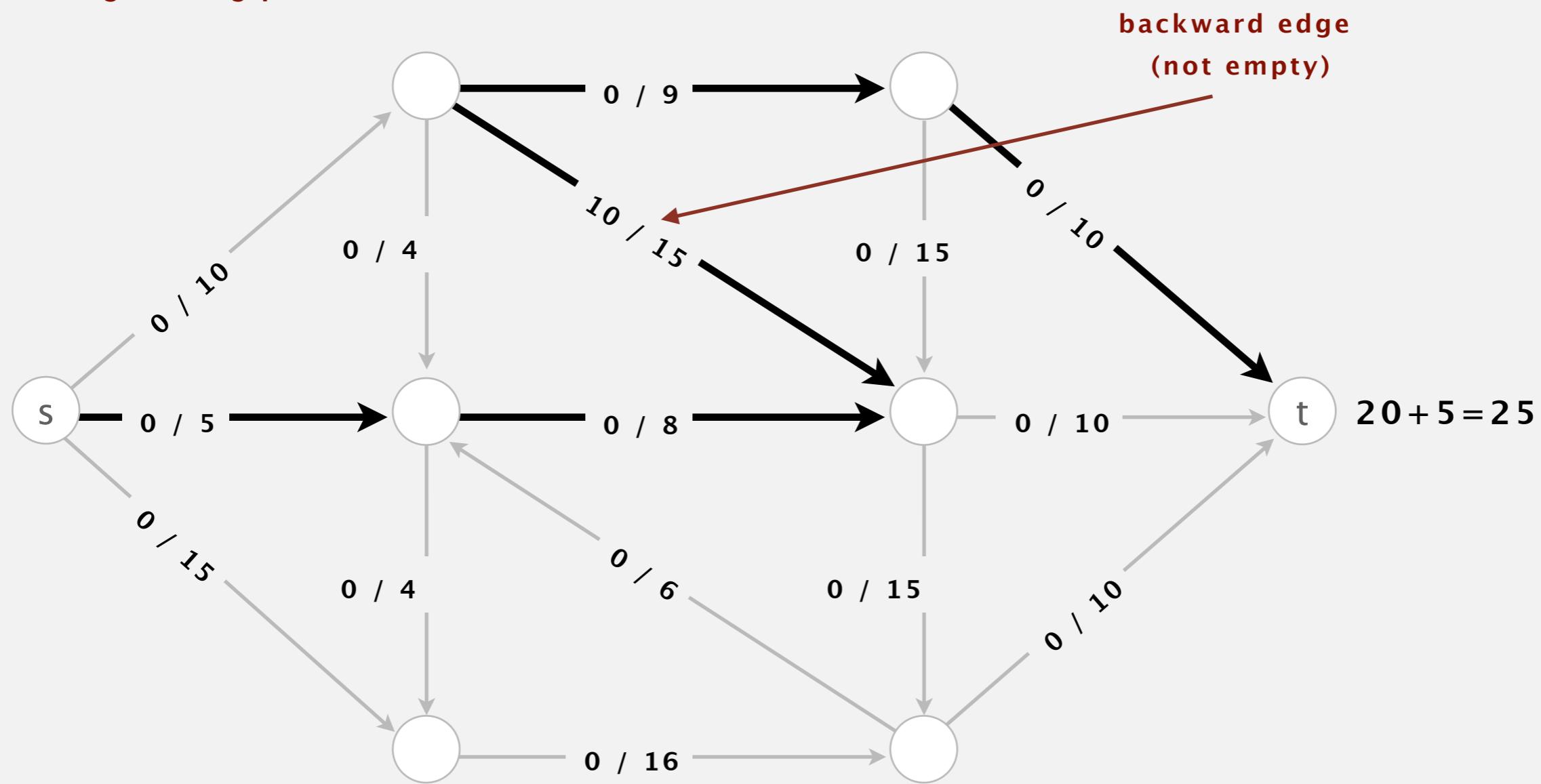


Idea: increase flow along augmenting paths

# Augmenting path.

- Can increase flow on a forward edge (not full)
  - Can decrease flow on backward edge (not empty)

## 3rd Augmenting path

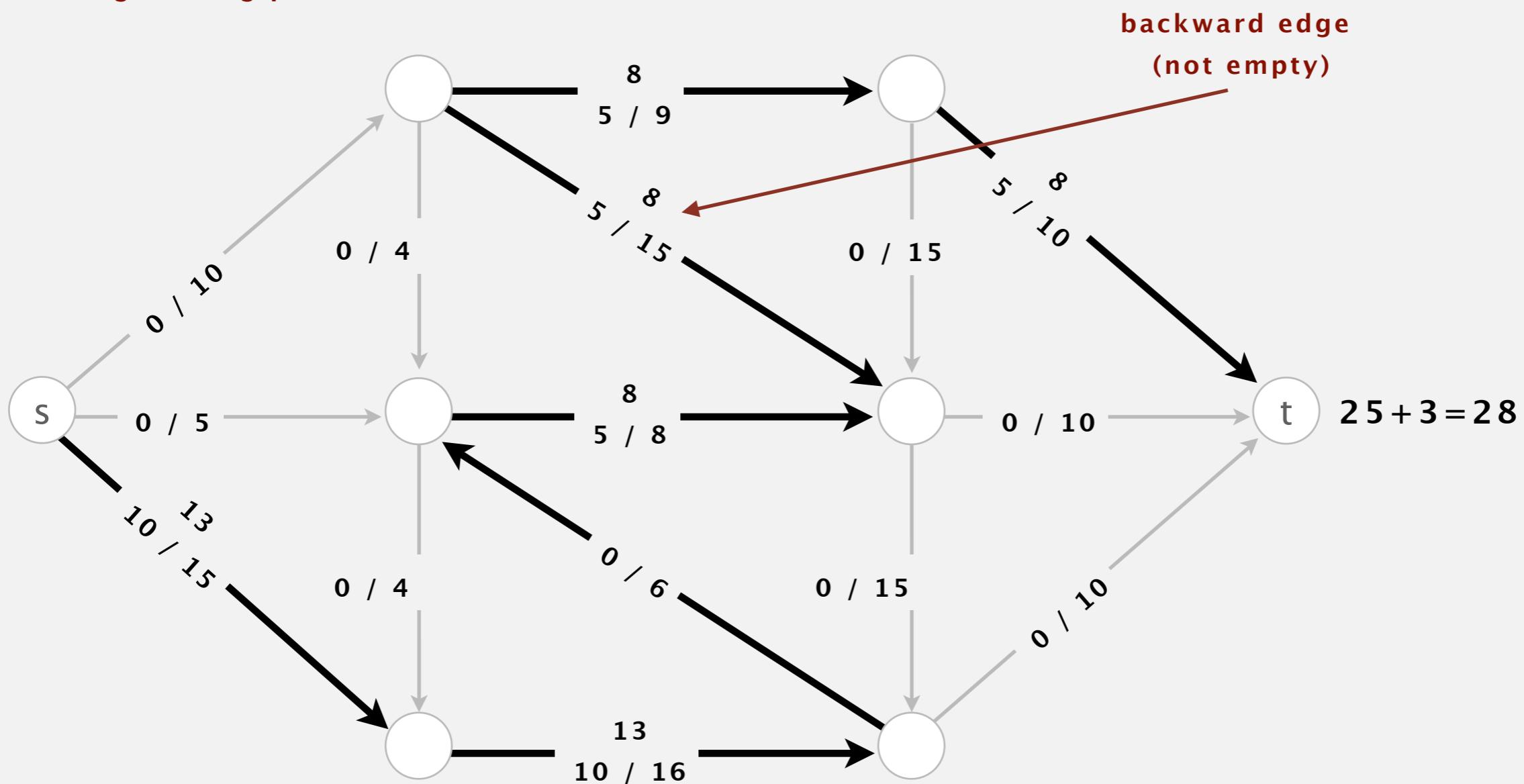


# Idea: increase flow along augmenting paths

## Augmenting path.

- Can increase flow on a forward edge (not full)
- Can decrease flow on backward edge (not empty)

4th Augmenting path

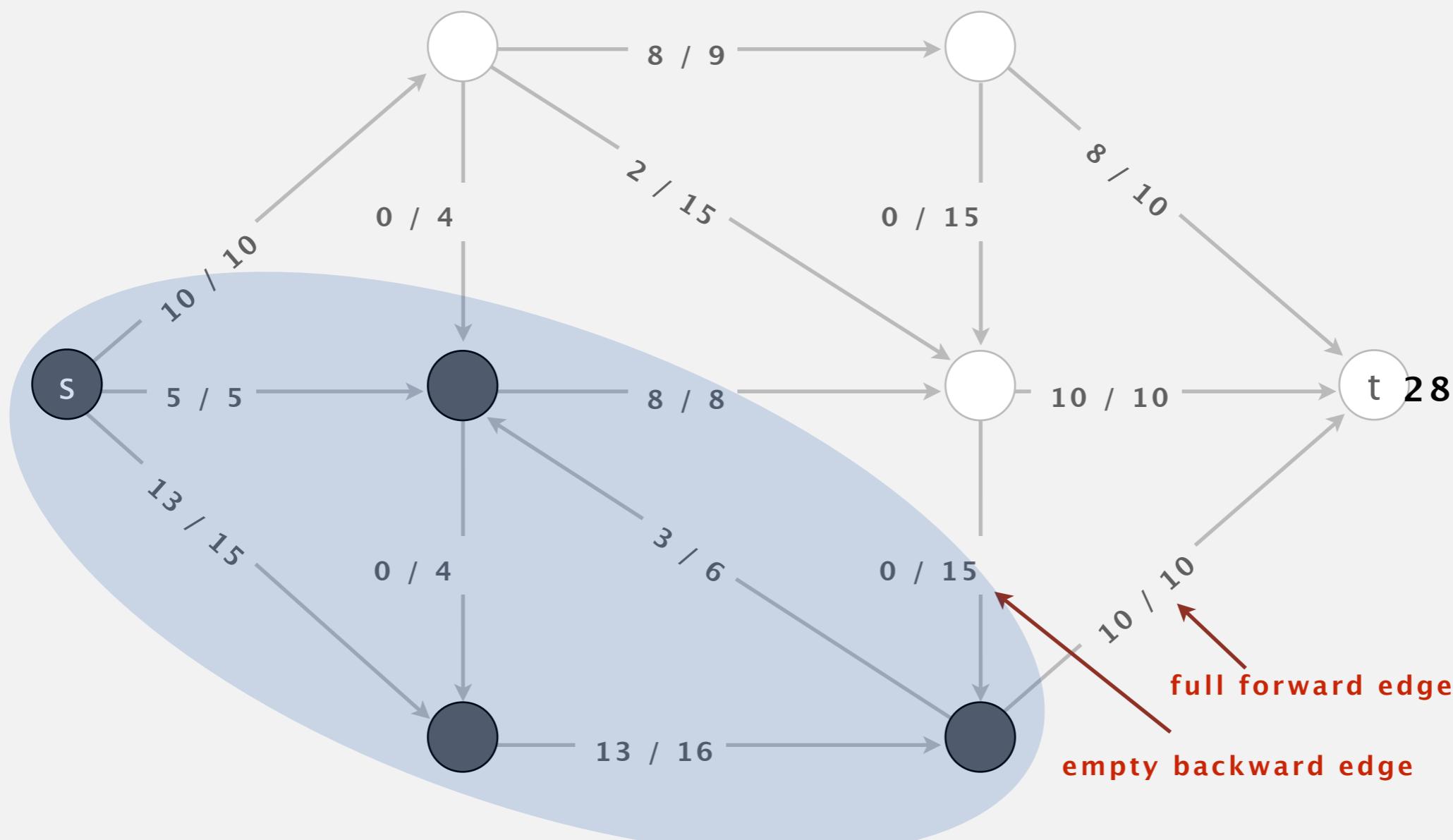


## Idea: increase flow along augmenting paths

Termination. All paths from  $s$  to  $t$  are blocked by either a

- Full forward edge.
- Empty backward edge.

no more augmenting paths



# Ford-Fulkerson algorithm

---

## Ford-Fulkerson algorithm

---

Start with 0 flow.

While there exists an augmenting path:

- find an augmenting path
  - compute bottleneck capacity
  - increase flow on that path by bottleneck capacity
- 

## Questions.

- How to compute a mincut?
- How to find an augmenting path?
- If FF terminates, does it always compute a maxflow?
- Does FF always terminate? If so, after how many augmentations?

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

## 6.4 MAXIMUM FLOW

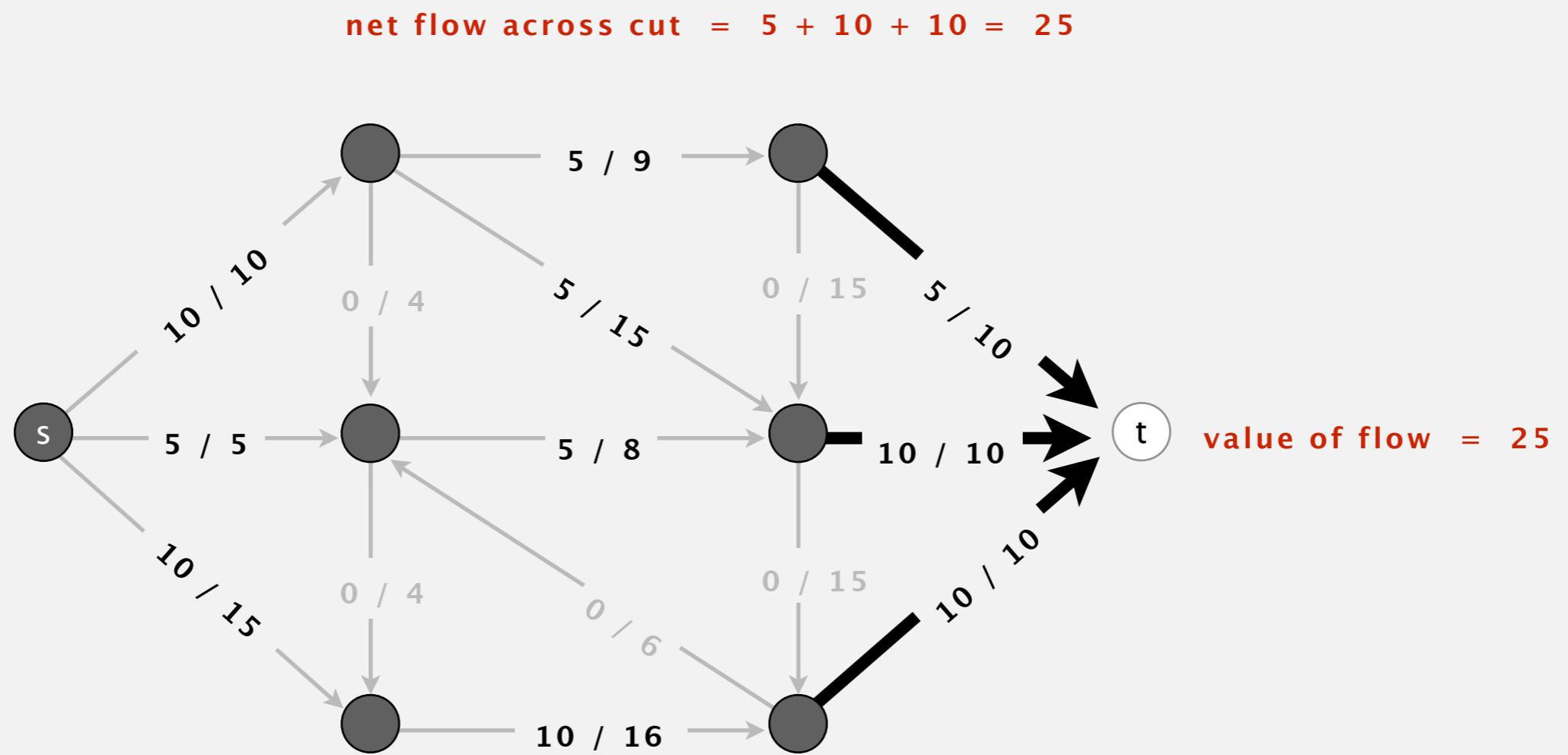
---

- ▶ *introduction*
- ▶ *Ford-Fulkerson algorithm*
- ▶ ***maxflow-mincut theorem***
- ▶ *analysis of running time*
- ▶ *Java implementation*
- ▶ *applications*

# Relationship between flows and cuts

**Def.** The **net flow across** a cut  $(A, B)$  is the sum of the flows on its edges from  $A$  to  $B$  minus the sum of the flows on its edges from  $B$  to  $A$ .

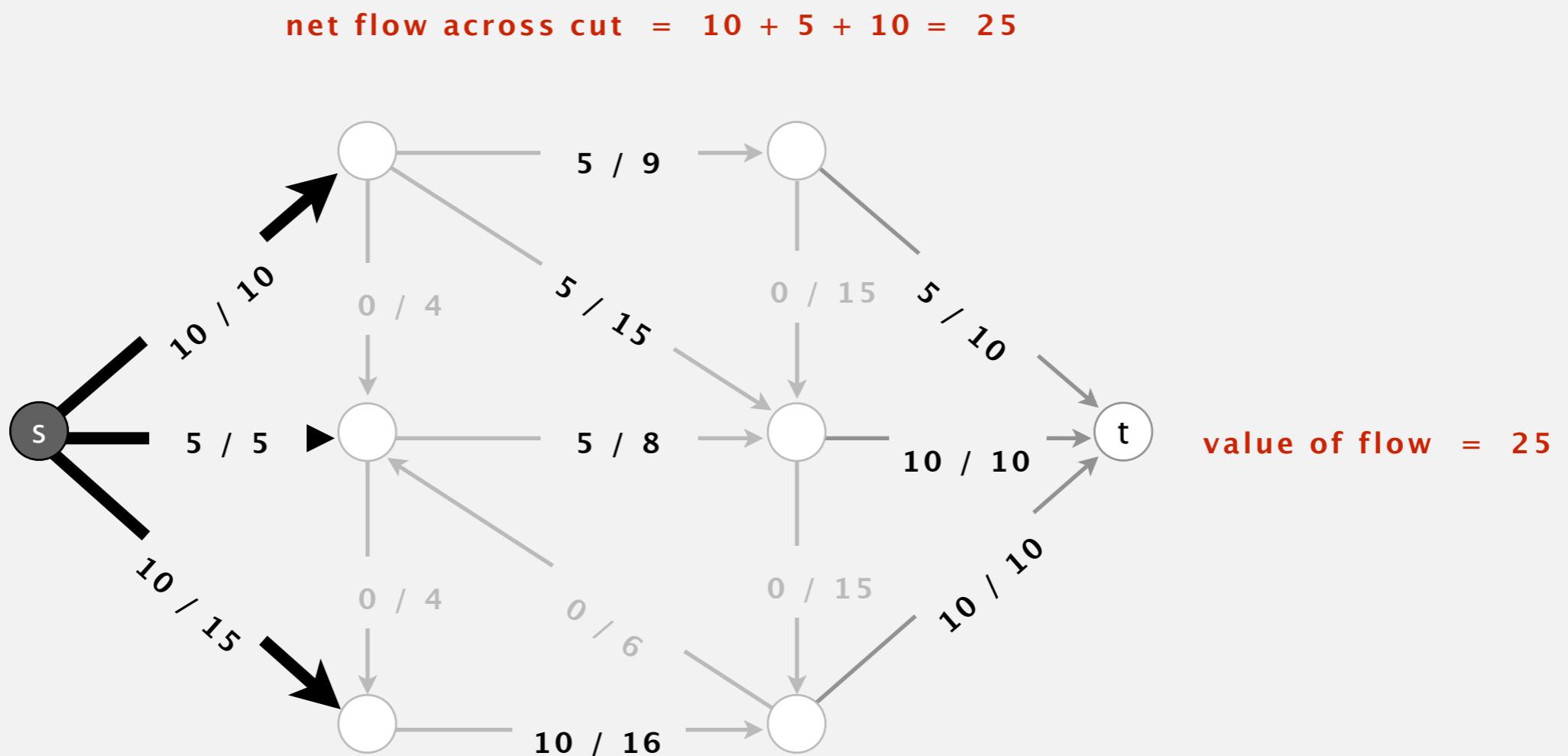
**Flow-value lemma.** Let  $f$  be any flow and let  $(A, B)$  be any cut. Then the net flow across  $(A, B)$  equals the value of  $f$



# Relationship between flows and cuts

**Def.** The **net flow across** a cut  $(A, B)$  is the sum of the flows on its edges from  $A$  to  $B$  minus the sum of the flows on its edges from  $B$  to  $A$ .

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

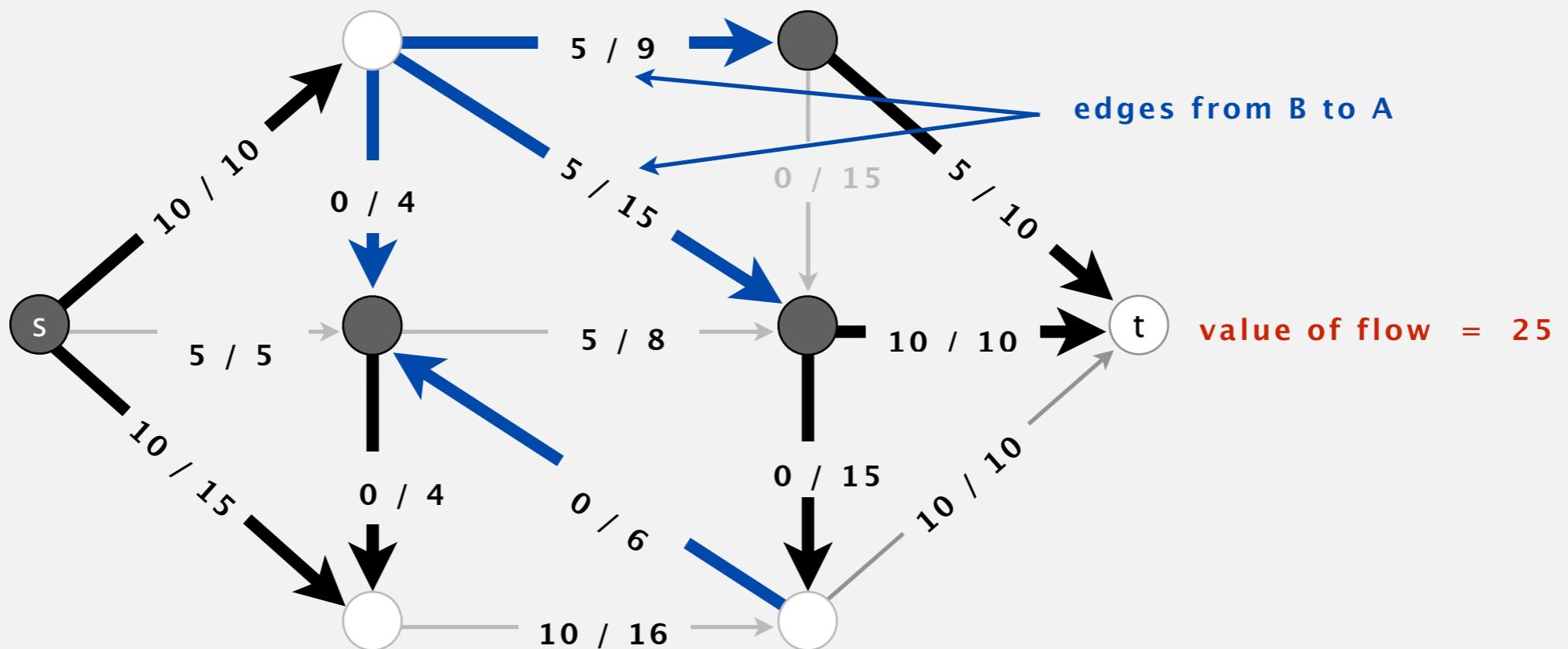


# Relationship between flows and cuts

**Def.** The **net flow across** a cut  $(A, B)$  is the sum of the flows on its edges from  $A$  to  $B$  minus the sum of the flows on its edges from  $B$  to  $A$ .

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

$$\text{net flow across cut} = (10 + 10 + 5 + 10 + 0 + 0) - (5 + 5 + 0 + 0) = 25$$



## Relationship between flows and cuts

---

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

**Intuition.** Conservation of flow.

**Pf.** By induction on the size of  $B$ .

- Base case:  $B = \{ t \}$ .
- Induction step: remains true by local equilibrium when moving any vertex from  $A$  to  $B$ .

**Corollary.** Outflow from  $s$  = inflow to  $t$  = value of flow.

# 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  $\leq$  the capacity of the cut.

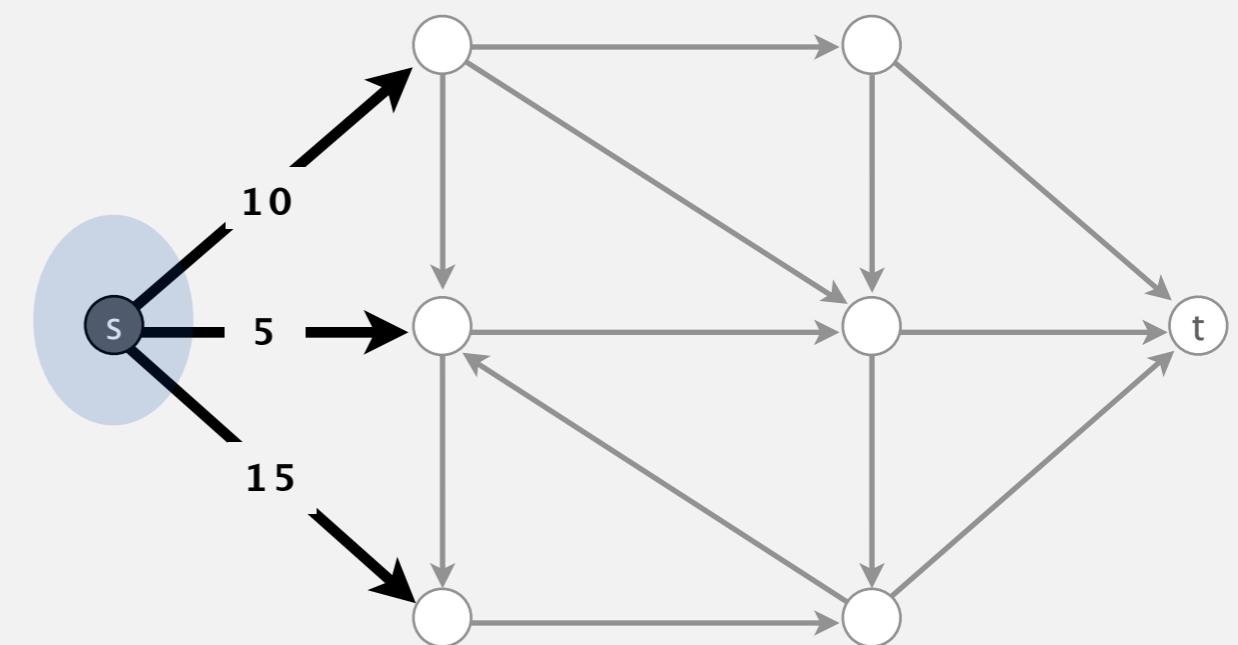
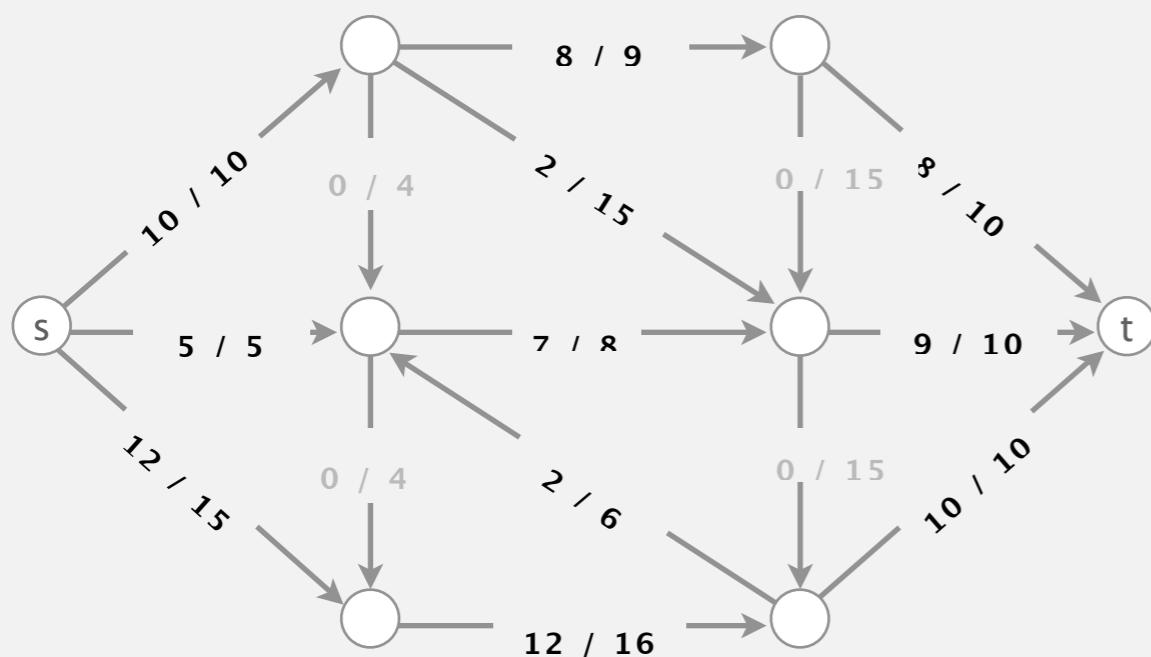
Pf. Value of flow  $f =$  net flow across cut  $(A, B) \leq$  capacity of cut  $(A, B).$



flow-value lemma



flow bounded by capacity



## Maxflow-mincut theorem

---

Augmenting path theorem. A flow  $f$  is a maxflow iff no augmenting paths.

Maxflow-mincut theorem. Value of the maxflow = capacity of mincut.

Pf. The following three conditions are equivalent for any flow  $f$ :

- i. There exists a cut whose capacity equals the value of the flow  $f$ .
- ii.  $f$  is a maxflow.
- iii. There is no augmenting path with respect to  $f$ .

[ i  $\Rightarrow$  ii ]

- Suppose that  $(A, B)$  is a cut with capacity equal to the value of  $f$ .
- Then, the value of any flow  $f' \leq$  capacity of  $(A, B) =$  value of  $f$ .
- Thus,  $f$  is a maxflow.

↑  
weak  
duality      ↑  
by  
assumption

## Maxflow-mincut theorem

---

**Augmenting path theorem.** A flow  $f$  is a maxflow iff no augmenting paths.

**Maxflow-mincut theorem.** Value of the maxflow = capacity of mincut.

**Pf.** The following three conditions are equivalent for any flow  $f$ :

- i. There exists a cut whose capacity equals the value of the flow  $f$ .
- ii.  $f$  is a maxflow.
- iii. There is no augmenting path with respect to  $f$ .

[ ii  $\Rightarrow$  iii ] We prove contrapositive:  $\sim$ iii  $\Rightarrow$   $\sim$ ii.

- Suppose that there is an augmenting path with respect to  $f$ .
- Can improve flow  $f$  by sending flow along this path.
- Thus,  $f$  is not a maxflow.

## Maxflow-mincut theorem

---

**Augmenting path theorem.** A flow  $f$  is a maxflow iff no augmenting paths.

**Maxflow-mincut theorem.** Value of the maxflow = capacity of mincut.

**Pf.** The following three conditions are equivalent for any flow  $f$ :

- i. There exists a cut whose capacity equals the value of the flow  $f$ .
- ii.  $f$  is a maxflow.
- iii. There is no augmenting path with respect to  $f$ .

[ iii  $\Rightarrow$  i ]

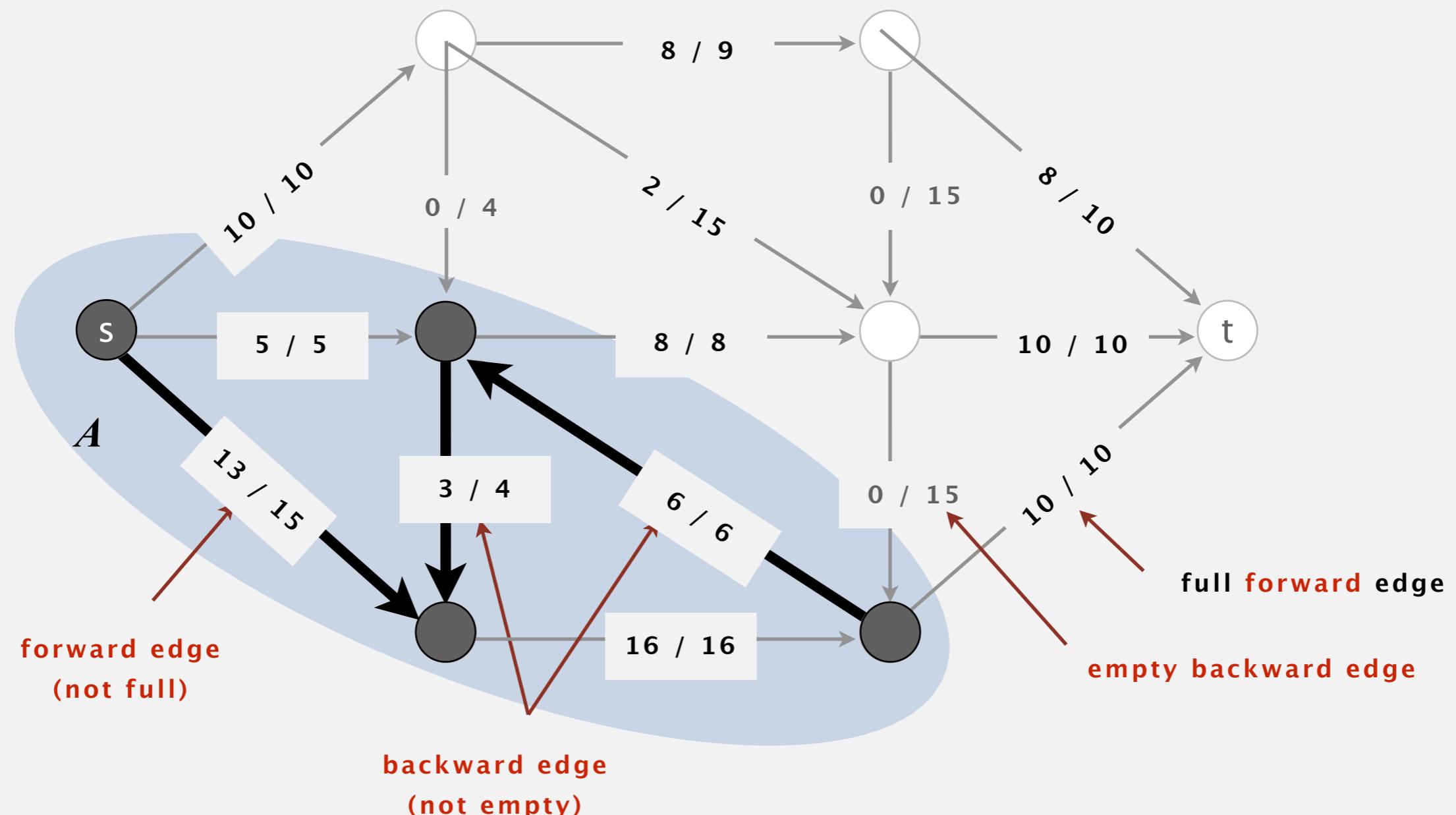
Suppose that there is no augmenting path with respect to  $f$ .

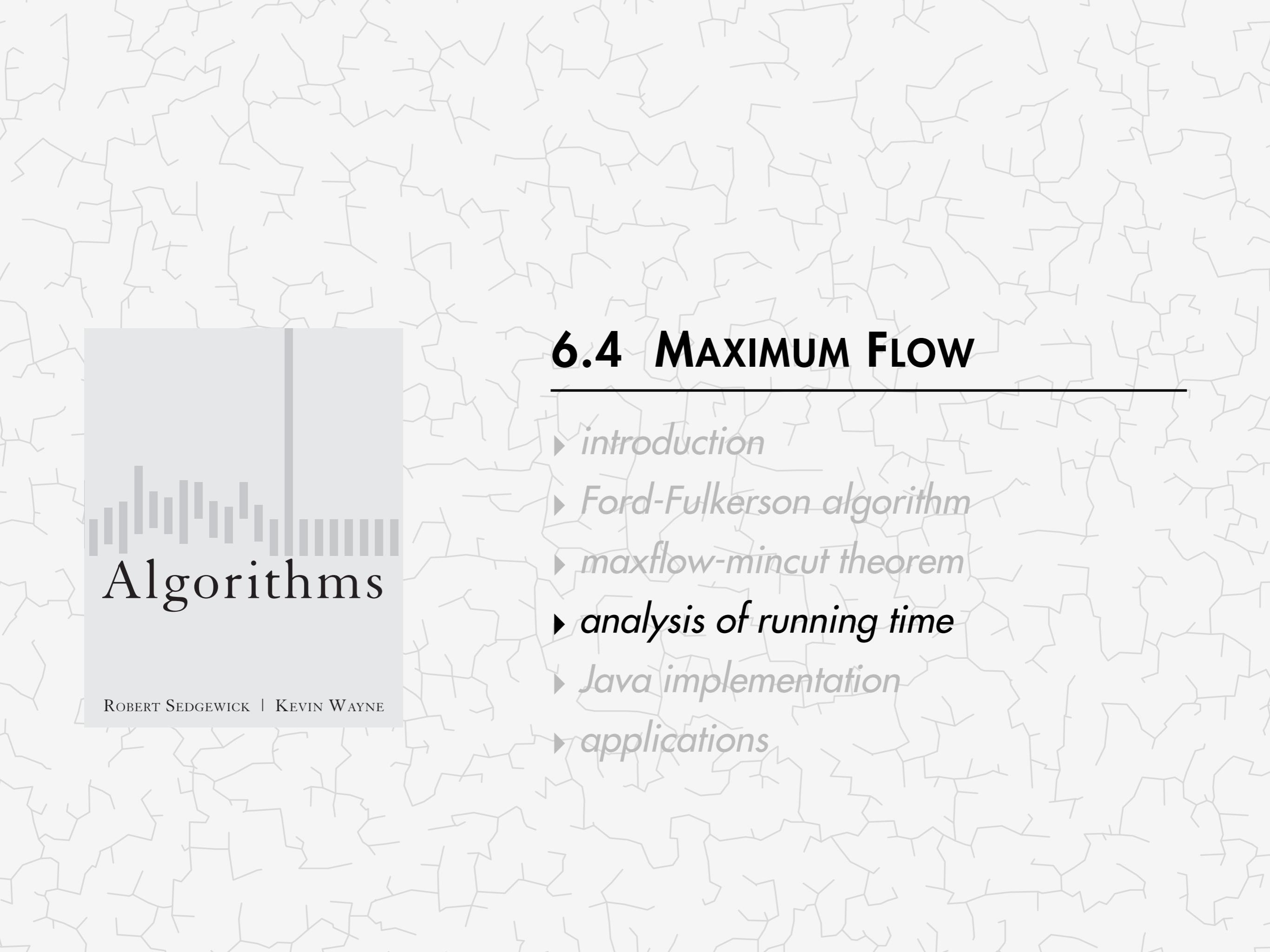
- Let  $(A, B)$  be a cut where  $A$  is the set of vertices connected to  $s$  by an undirected path with no full forward or empty backward edges.
- By definition of cut,  $s$  is in  $A$ .
- Since no augmenting path,  $t$  is in  $B$ .
- Capacity of cut = net flow across cut  $\leftarrow$  **forward edges full; backward edges empty**  
= value of flow  $f$ .  $\leftarrow$  **flow-value lemma**

# Computing a mincut from a maxflow

To compute mincut  $(A, B)$  from maxflow  $f$  :

- By augmenting path theorem, no augmenting paths with respect to  $f$ .
- Compute  $A = \text{set of vertices connected to } s \text{ by an undirected path}$  with no full forward or empty backward edges.





# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

## 6.4 MAXIMUM FLOW

---

- ▶ *introduction*
- ▶ *Ford-Fulkerson algorithm*
- ▶ *maxflow-mincut theorem*
- ▶ ***analysis of running time***
- ▶ *Java implementation*
- ▶ *applications*

# Ford-Fulkerson algorithm

## Ford-Fulkerson algorithm

Start with 0 flow.

While there exists an augmenting path:

- find an augmenting path
- compute bottleneck capacity
- increase flow on that path by bottleneck capacity

## Fundamental questions.

- How to compute a mincut? Easy. ✓
- How to find an augmenting path? BFS works well.
- If FF terminates, does it always compute a maxflow? Yes. ✓
- Does FF always terminate? If so, after how many augmentations?

yes, provided edge capacities are integers      requires clever analysis  
(or augmenting paths are chosen carefully)

# Ford-Fulkerson algorithm with integer capacities

---

Important special case. Edge capacities are integers between 1 and  $U$ .

flow on each edge is an integer



Invariant. The flow is integer-valued throughout Ford-Fulkerson.

Pf. [by induction]

- Bottleneck capacity is an integer.
- Flow on an edge increases/decreases by bottleneck capacity.

Proposition. Number of augmentations  $\leq$  the value of the maxflow.

Pf. Each augmentation increases the value by at least 1.

critical for some applications (stay tuned)



and FF finds one!

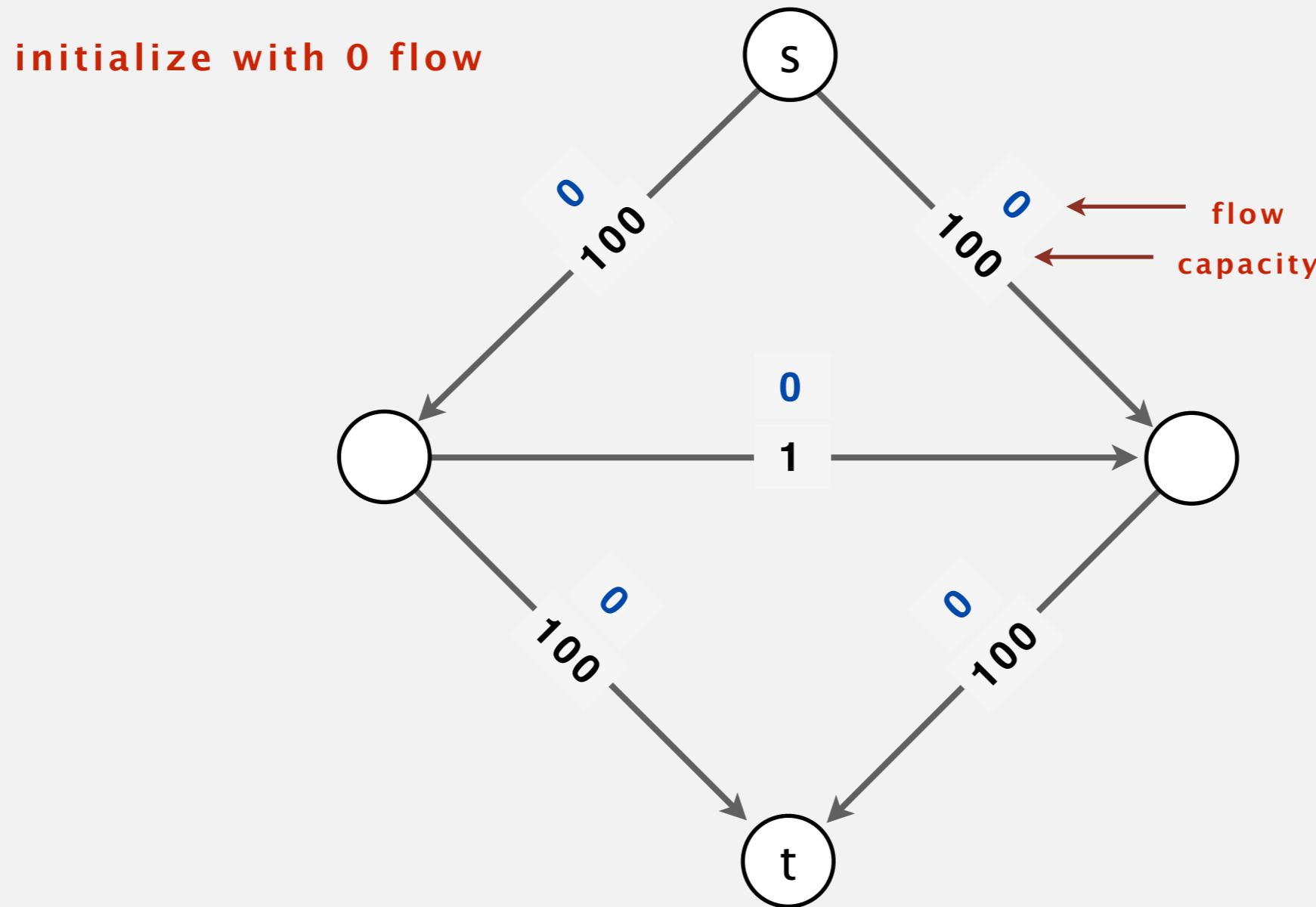
Integrality theorem. There exists an integral maxflow.

Pf. Ford-Fulkerson terminates and maxflow that it finds is integer-valued.

## Bad case for Ford-Fulkerson

---

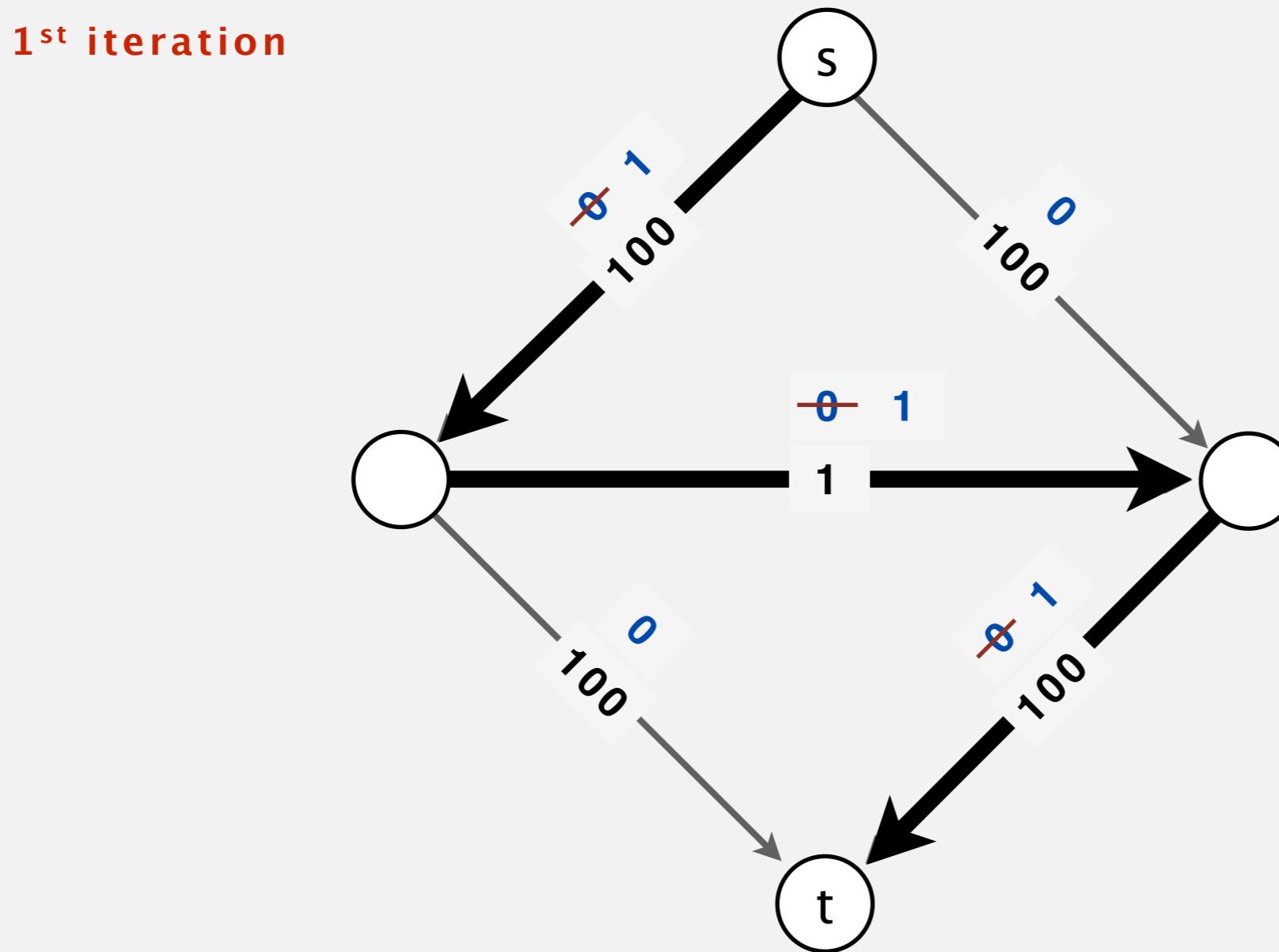
**Bad news.** Even when edge capacities are integers, number of augmenting paths could be equal to the value of the maxflow.



## Bad case for Ford-Fulkerson

---

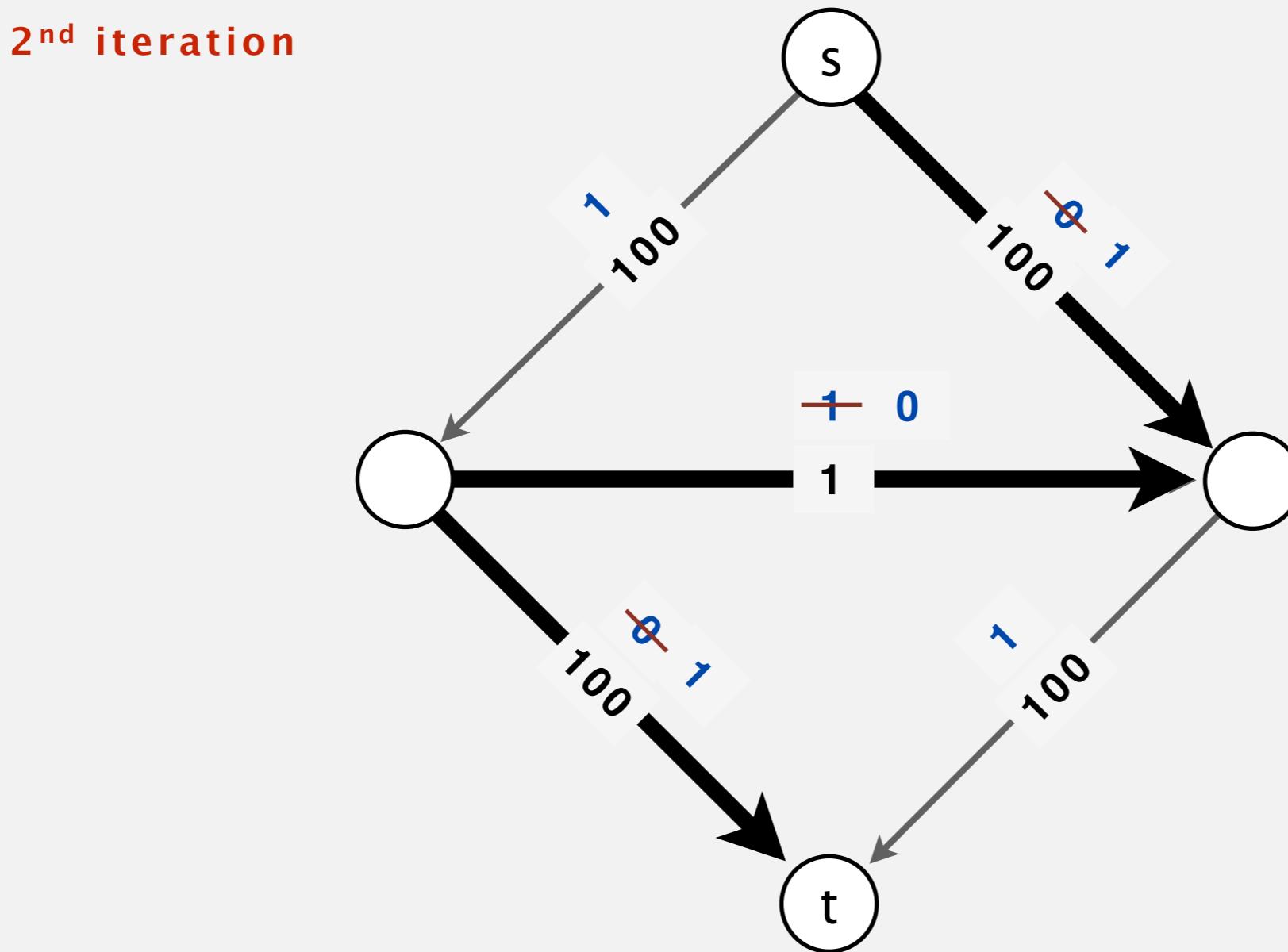
**Bad news.** Even when edge capacities are integers, number of augmenting paths could be equal to the value of the maxflow.



## Bad case for Ford-Fulkerson

---

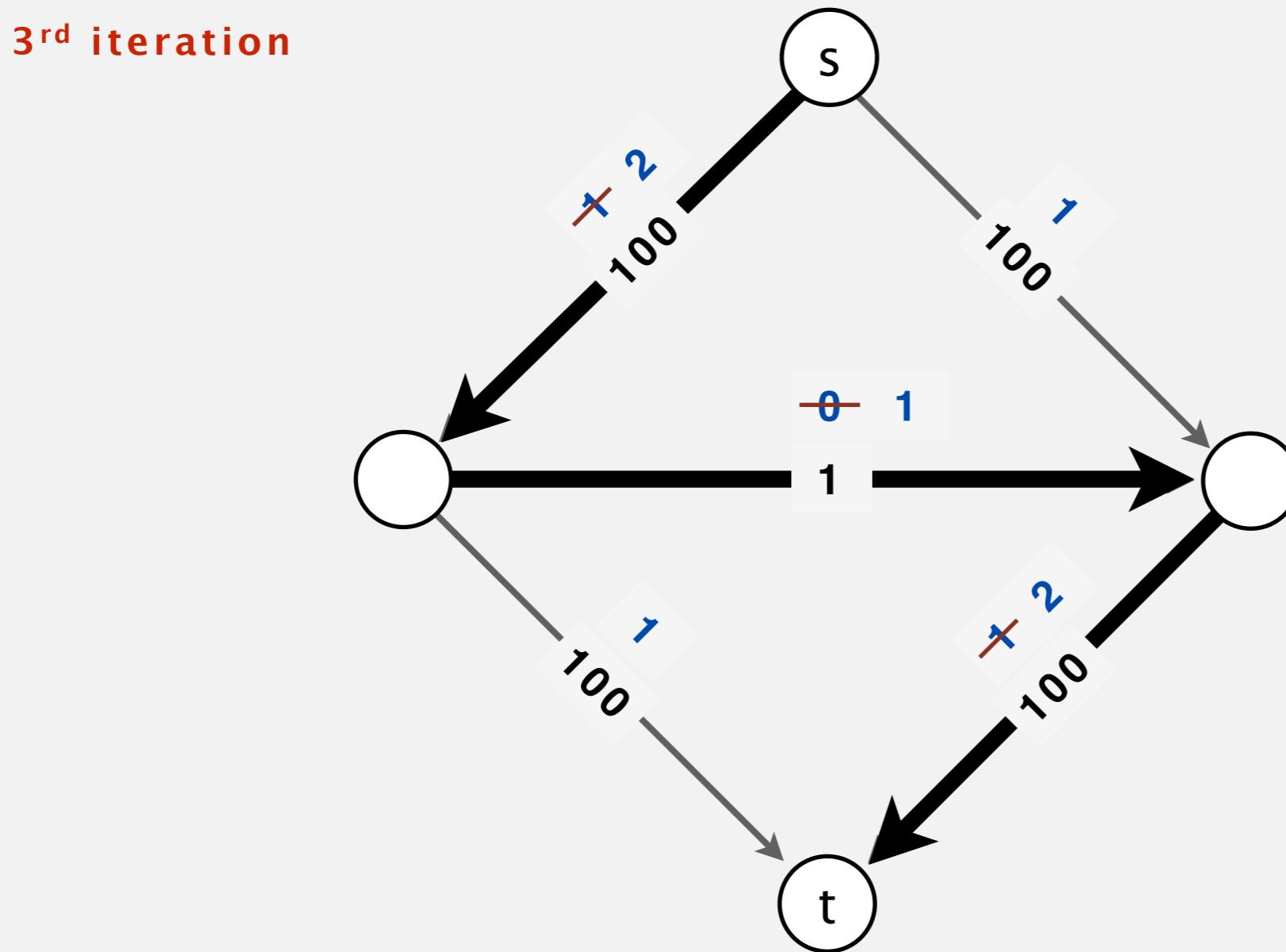
**Bad news.** Even when edge capacities are integers, number of augmenting paths could be equal to the value of the maxflow.



## Bad case for Ford-Fulkerson

---

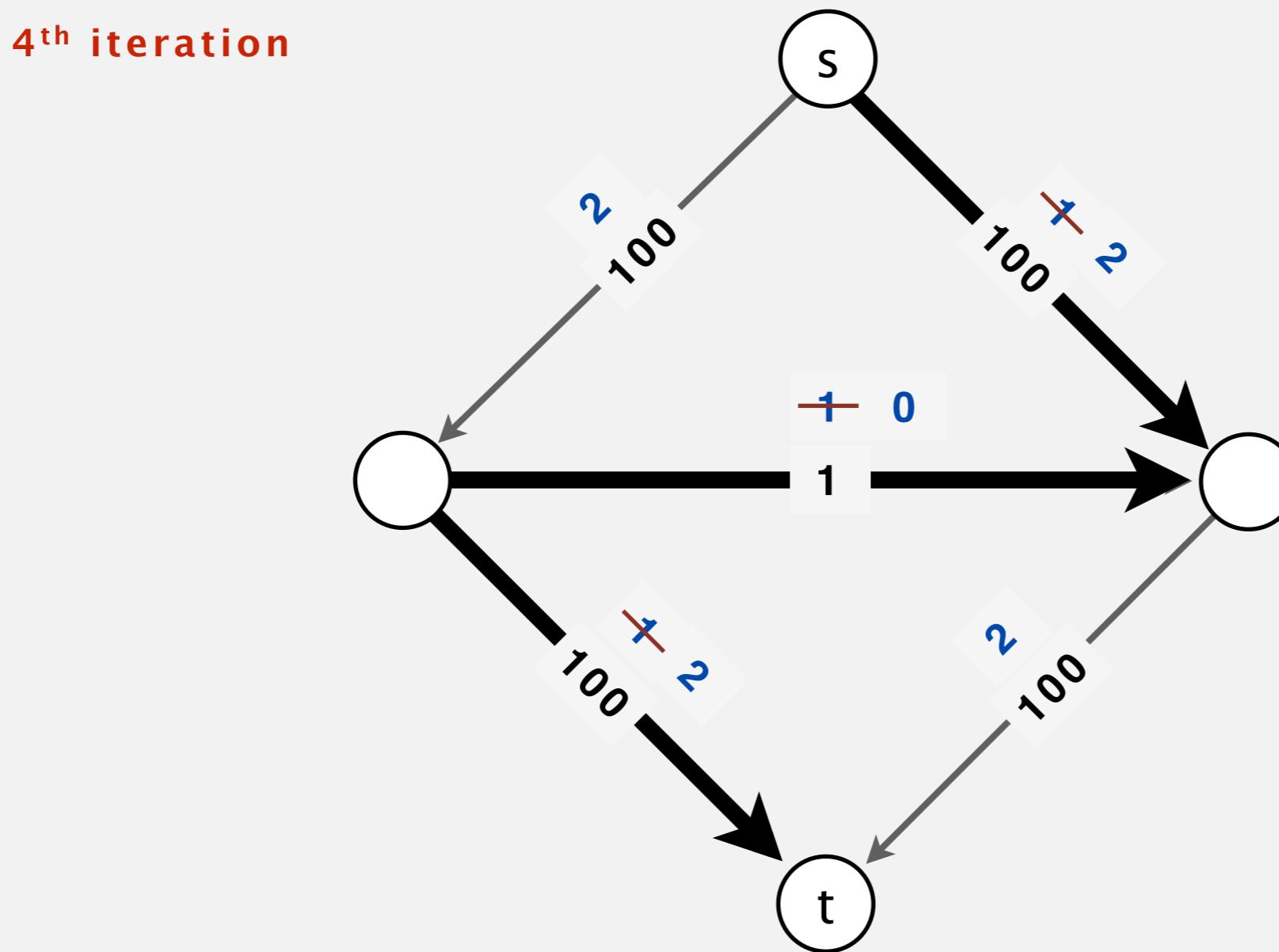
**Bad news.** Even when edge capacities are integers, number of augmenting paths could be equal to the value of the maxflow.



## Bad case for Ford-Fulkerson

---

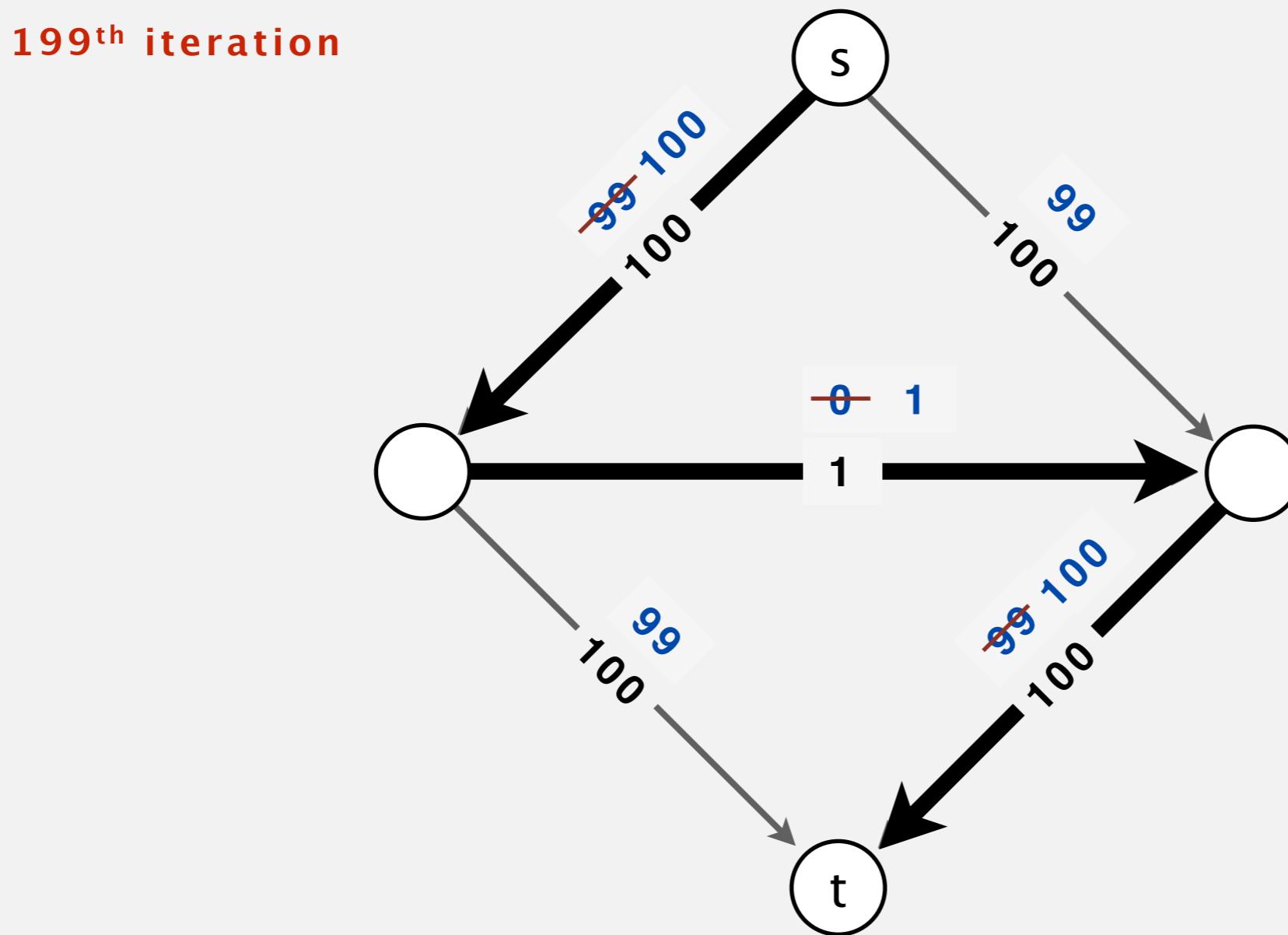
**Bad news.** Even when edge capacities are integers, number of augmenting paths could be equal to the value of the maxflow.



## Bad case for Ford-Fulkerson

---

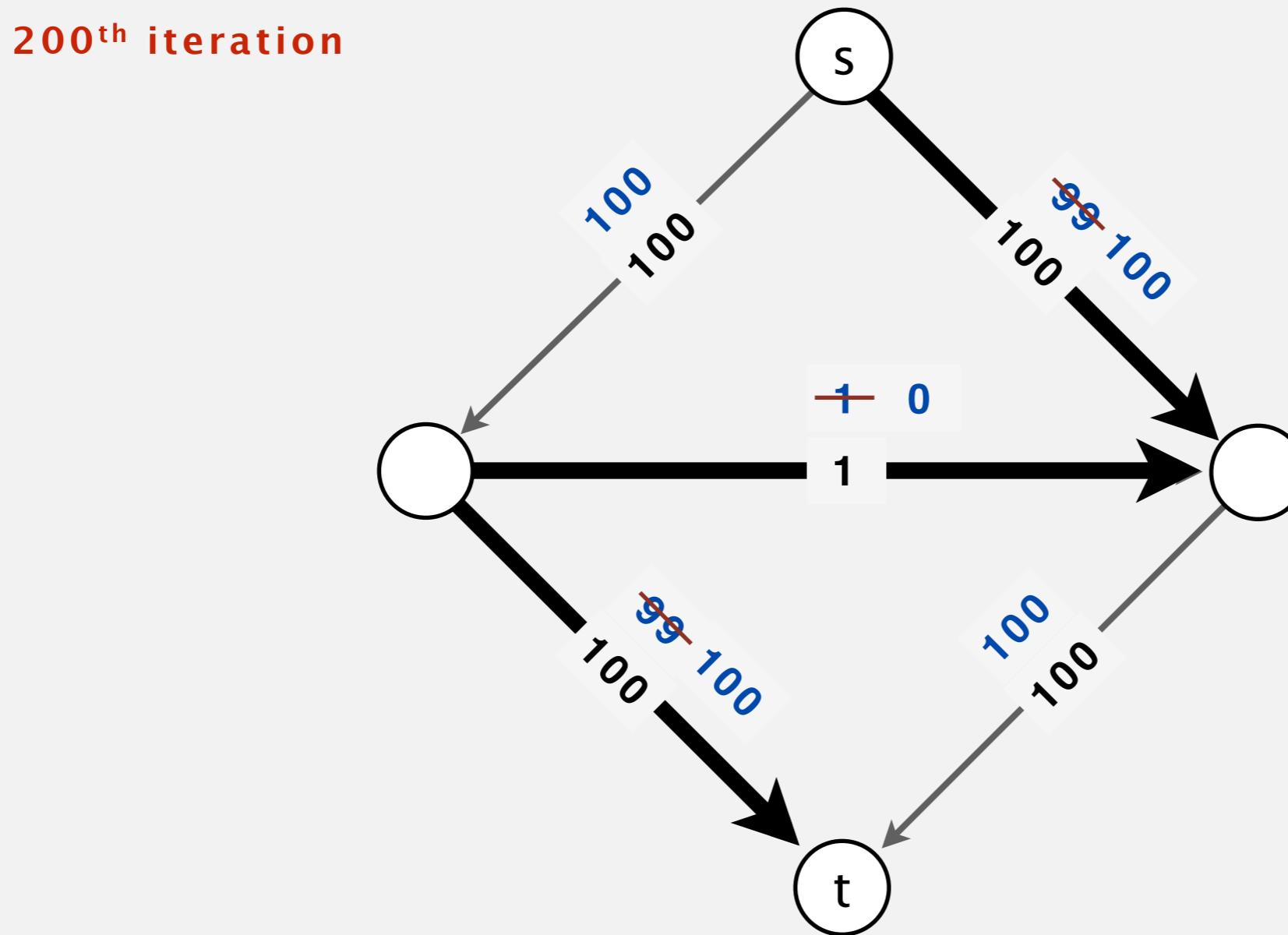
**Bad news.** Even when edge capacities are integers, number of augmenting paths could be equal to the value of the maxflow.



## Bad case for Ford-Fulkerson

---

**Bad news.** Even when edge capacities are integers, number of augmenting paths could be equal to the value of the maxflow.



## Bad case for Ford-Fulkerson

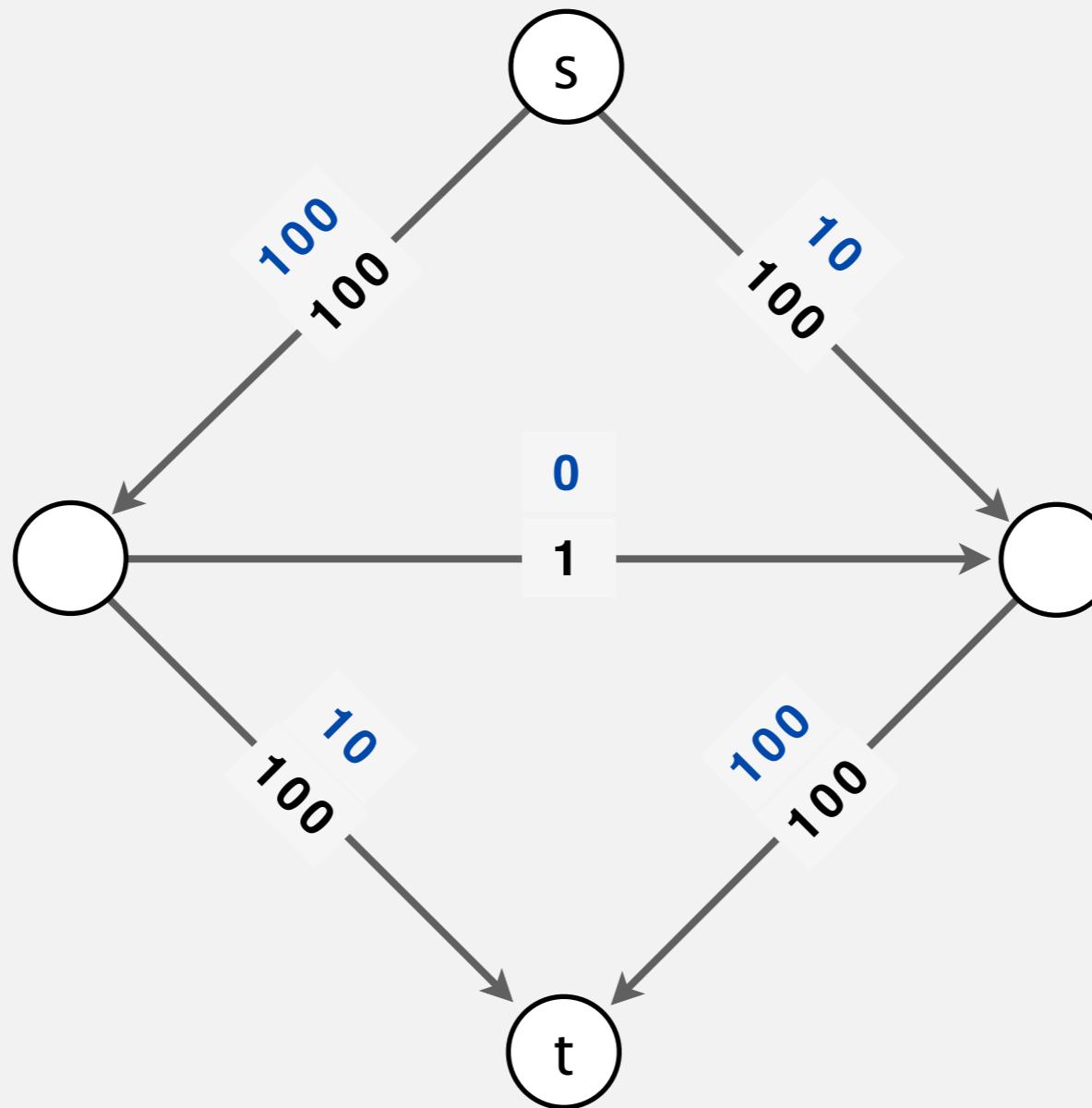
---

**Bad news.** Even when edge capacities are integers, number of augmenting paths could be equal to the value of the maxflow.

can be exponential in input

size

**Good news.** This case is easily avoided. [ use shortest/fattest path ]



# How to choose augmenting paths?

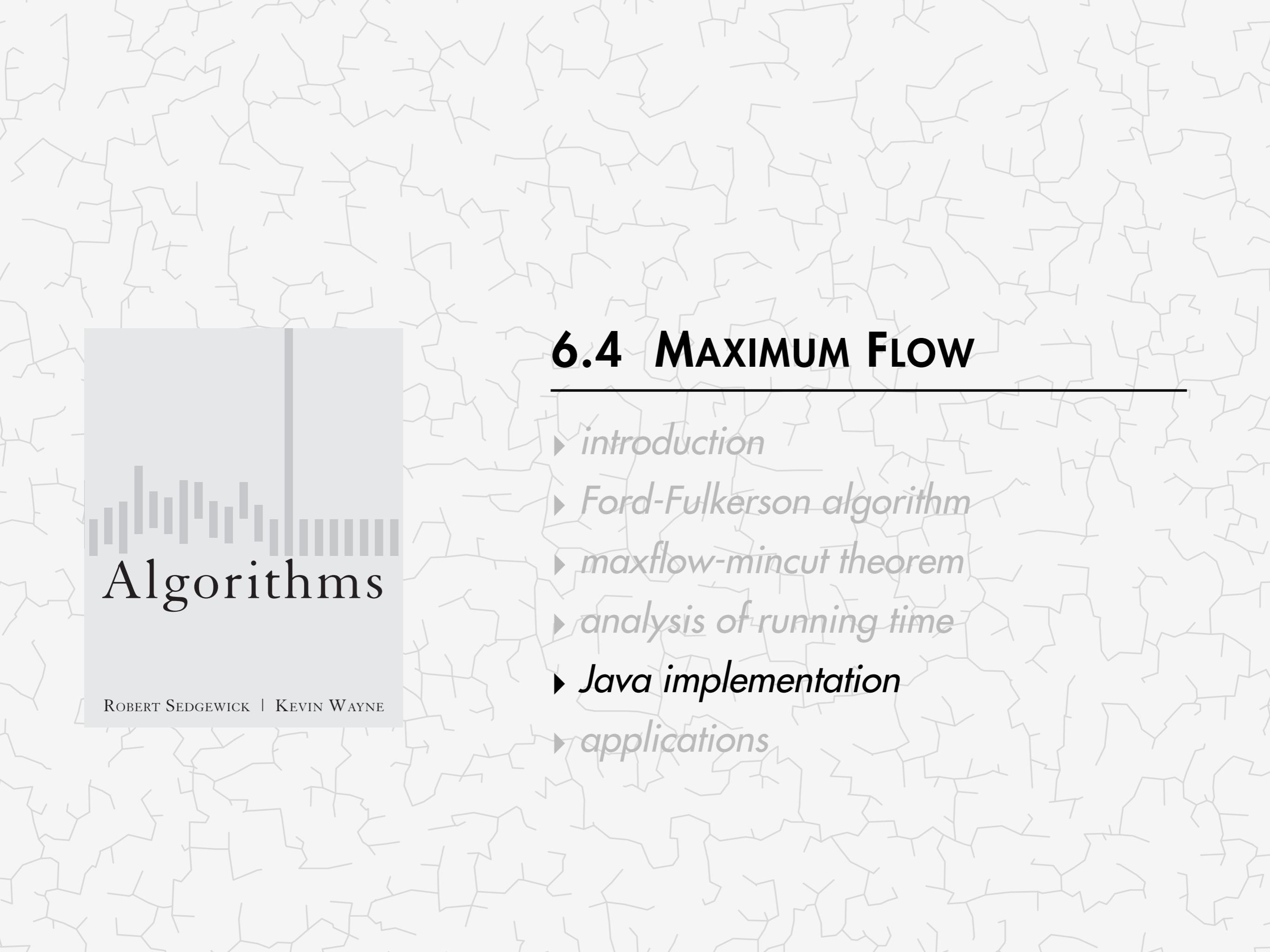
Warning: This slide is slightly different from video

Use care when selecting augmenting paths.

- Some choices lead to exponential algorithms.
- Clever choices lead to polynomial algorithms.

| augmenting path      | number of paths        | implementation   |
|----------------------|------------------------|------------------|
| <b>random path</b>   | $\leq E^U$             | randomized queue |
| <b>DFS path</b>      | $\leq E^U$             | stack (DFS)      |
| <b>shortest path</b> | $\leq \frac{1}{2} E V$ | queue (BFS)      |
| <b>fattest path</b>  | $\leq E \ln(E^U)$      | priority queue   |

digraph with  $V$  vertices,  $E$  edges, and integer capacities between 1 and  $U$



# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

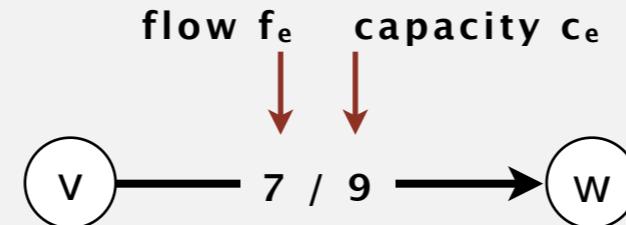
## 6.4 MAXIMUM FLOW

---

- ▶ *introduction*
- ▶ *Ford-Fulkerson algorithm*
- ▶ *maxflow-mincut theorem*
- ▶ *analysis of running time*
- ▶ ***Java implementation***
- ▶ *applications*

# Flow network representation

Flow edge data type. Associate flow  $f_e$  and capacity  $c_e$  with edge  $e = v \rightarrow w$ .



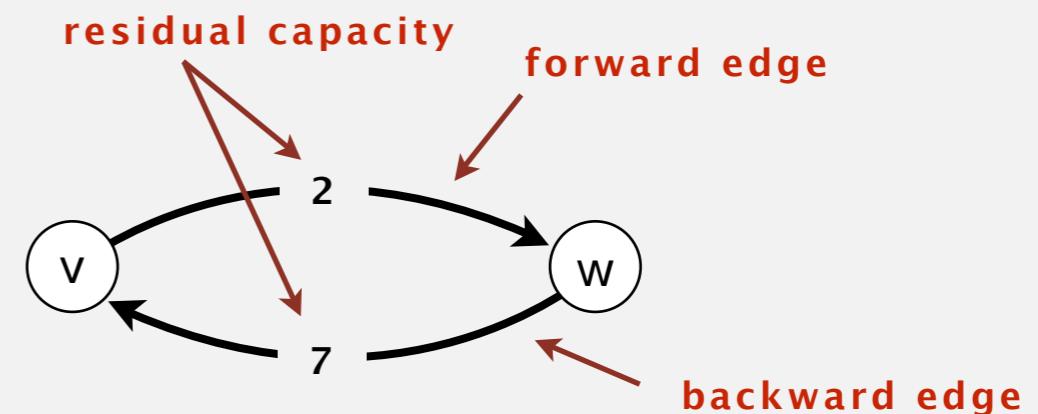
Flow network data type. Must be able to process edge  $e = v \rightarrow w$  in either direction: include  $e$  in adjacency lists of both  $v$  and  $w$ .

## Residual (spare) capacity.

- Forward edge: residual capacity  $= c_e - f_e$ .
- Backward edge: residual capacity  $= f_e$ .

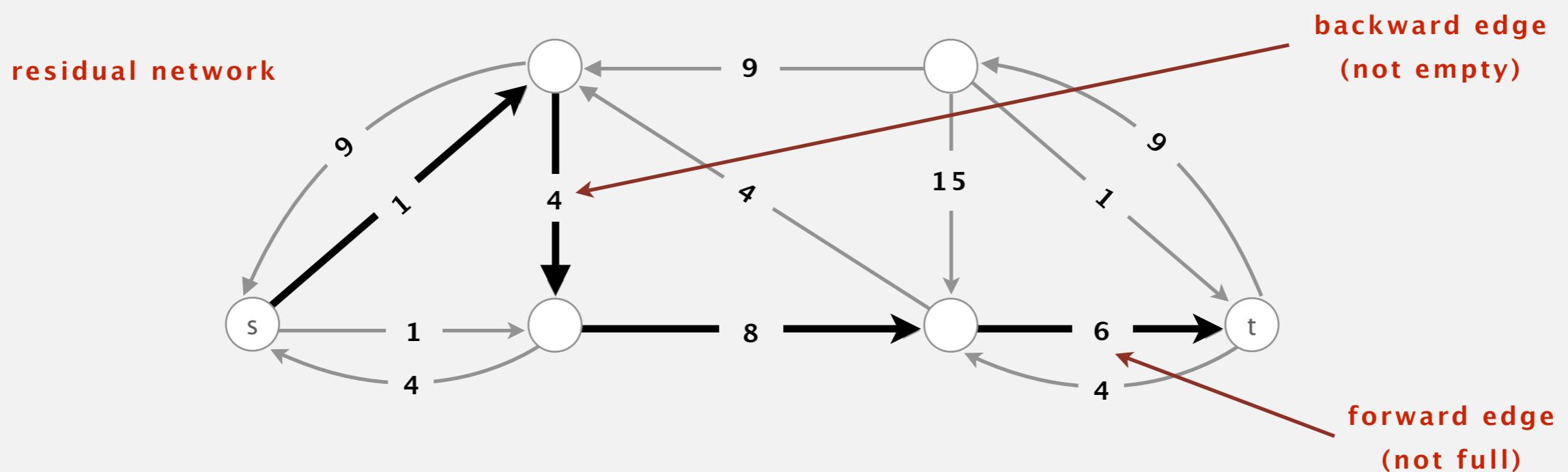
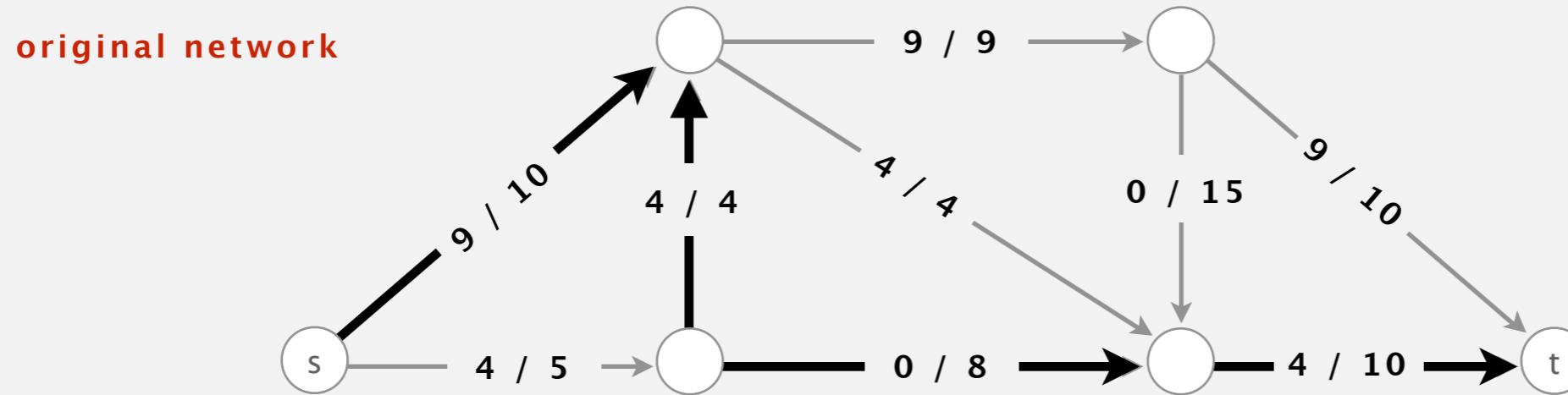
## Augment flow.

- Forward edge: add  $\Delta$ .
- Backward edge: subtract  $\Delta$ .



# Flow network representation

Residual network. A useful view of a flow network.



Key point. Augmenting paths in original network are in 1-1 correspondence with directed paths in residual network.

# Flow edge API

```
public class FlowEdge
```

```
    FlowEdge(int v, int w, double capacity)      create a flow edge  $v \rightarrow w$ 
```

```
    int from()                                vertex this edge points from
```

```
    int to()                                  vertex this edge points to
```

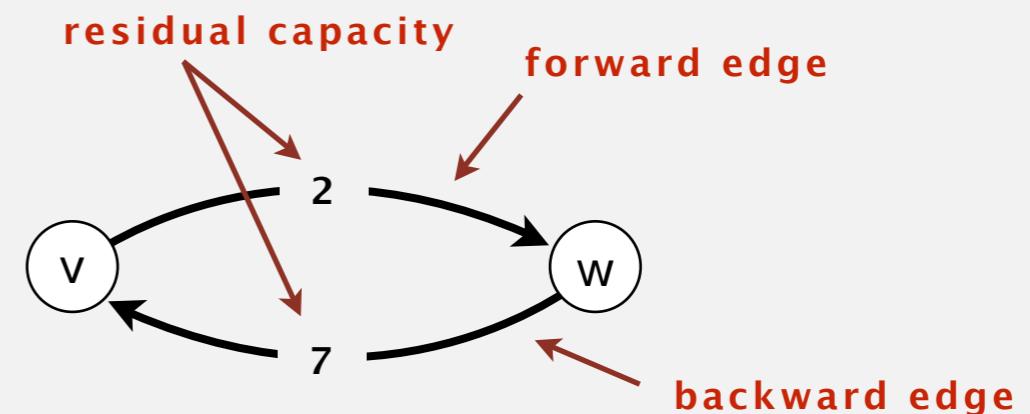
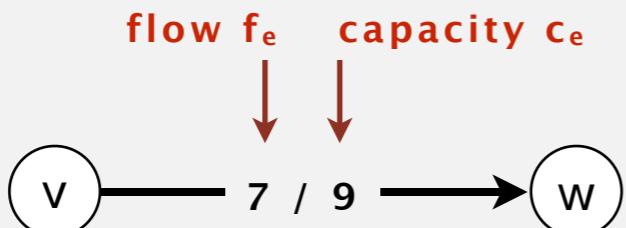
```
    int other(int v)                          other endpoint
```

```
    double capacity()                         capacity of this edge
```

```
    double flow()                            flow in this edge
```

```
    double residualCapacityTo(int v)        residual capacity toward v
```

```
    void addResidualFlowTo(int v, double delta)    add delta flow toward v
```



# Flow edge: Java implementation

```
public class FlowEdge
{
    private final int v, w;          // from and to
    private final double capacity;   // capacity
    private double flow;            // flow

    public FlowEdge(int v, int w, double capacity)
    {
        this.v      = v;
        this.w      = w;
        this.capacity = capacity;
    }

    public int from()      { return v; }
    public int to()        { return w; }
    public double capacity() { return capacity; }
    public double flow()    { return flow; }

    public int other(int vertex)
    {
        if      (vertex == v) return w;
        else if (vertex == w) return v;
        else throw new IllegalArgumentException();
    }

    public double residualCapacityTo(int vertex)      {...}
    public void addResidualFlowTo(int vertex, double delta) {...}
}
```

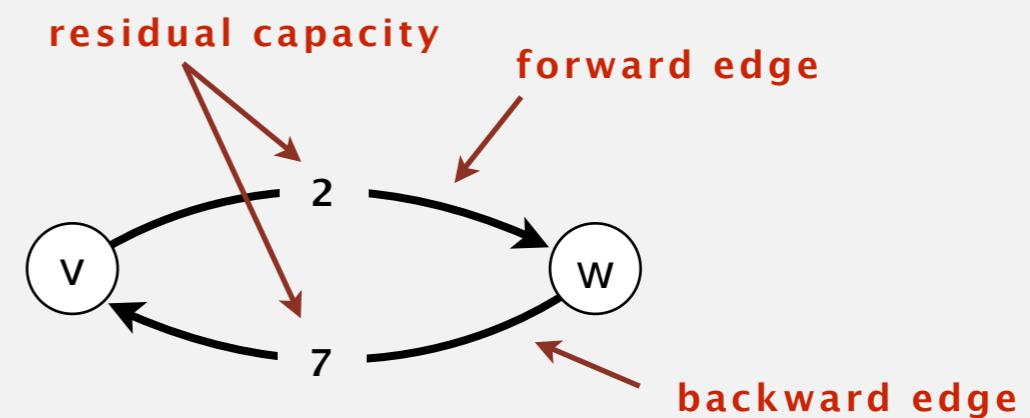
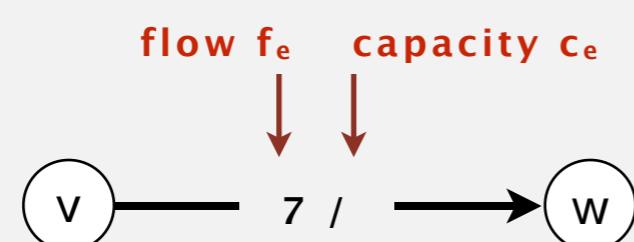
flow variable

next slide

# Flow edge: Java implementation

```
public double residualCapacityTo(int vertex)
{
    if      (vertex == v) return flow;
    else if (vertex == w) return capacity - flow;
    else throw new IllegalArgumentException();
}

public void addResidualFlowTo(int vertex, double delta)
{
    if      (vertex == v) flow -= delta;
    else if (vertex == w) flow += delta;
    else throw new IllegalArgumentException();
}
```



# Flow network API

---

```
public class FlowNetwork
```

|                    |                     |   |
|--------------------|---------------------|---|
|                    | FlowNetwork(int V)  | <i>create an empty flow network with V vertices</i> |
|                    | FlowNetwork(In in)  | <i>construct flow network input stream</i>          |
| void               | addEdge(FlowEdge e) | <i>add flow edge e to this flow network</i>         |
| Iterable<FlowEdge> | adj(int v)          | <i>forward and backward edges incident to v</i>     |
| Iterable<FlowEdge> | edges()             | <i>all edges in this flow network</i>               |
| int                | V()                 | <i>number of vertices</i>                           |
| int                | E()                 | <i>number of edges</i>                              |
| String             | toString()          | <i>string representation</i>                        |

**Conventions.** Allow self-loops and parallel edges.

# Flow network: Java implementation

```
public class FlowNetwork
{
    private final int V;
    private Bag<FlowEdge>[] adj;

    public FlowNetwork(int V)
    {
        this.V = V;
        adj = (Bag<FlowEdge>[]) new Bag[V];
        for (int v = 0; v < V; v++)
            adj[v] = new Bag<FlowEdge>();
    }

    public void addEdge(FlowEdge e)
    {
        int v = e.from();
        int w = e.to();
        adj[v].add(e);
        adj[w].add(e);
    }

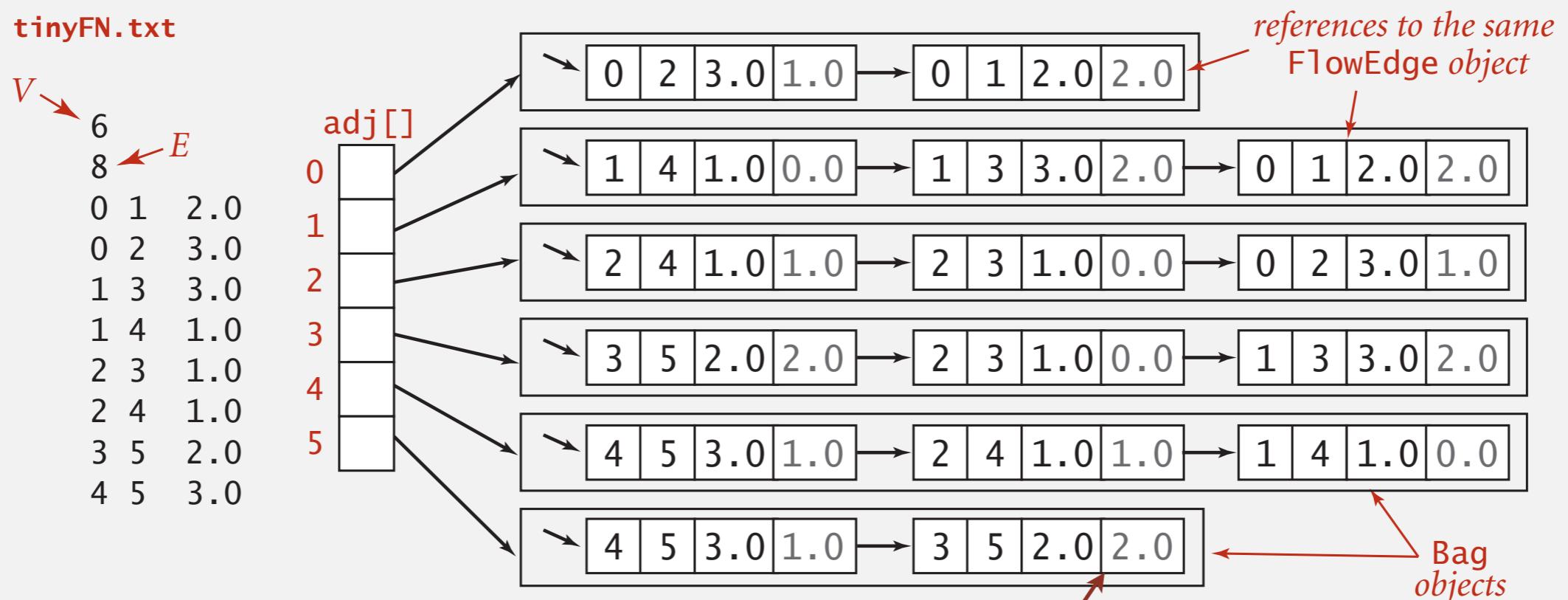
    public Iterable<FlowEdge> adj(int v)
    { return adj[v]; }
}
```

← same as EdgeWeightedGraph,  
but adjacency lists of  
FlowEdges instead of Edges

← add forward edge  
← add backward edge

# Flow network: adjacency-lists representation

Maintain vertex-indexed array of FlowEdge lists (use Bag abstraction).



**Note.** Adjacency list includes edges with 0 residual capacity.  
(residual network is represented implicitly)

# Ford-Fulkerson: Java implementation

```
public class FordFulkerson
{
    private boolean[] marked;      // true if s->v path in residual network
    private FlowEdge[] edgeTo;     // last edge on s->v path
    private double value;          // value of flow

    public FordFulkerson(FlowNetwork G, int s, int t)
    {
        value = 0.0;
        while (hasAugmentingPath(G, s, t))
        {
            double bottle = Double.POSITIVE_INFINITY;
            for (int v = t; v != s; v = edgeTo[v].other(v)) ←
                bottle = Math.min(bottle, edgeTo[v].residualCapacityTo(v)); ←
                    compute bottleneck capacity

            for (int v = t; v != s; v = edgeTo[v].other(v)) ←
                edgeTo[v].addResidualFlowTo(v, bottle); ←
                    augment flow

            value += bottle;
        }
    }

    private boolean hasAugmentingPath(FlowNetwork G, int s, int t)
    { /* See previous slide. */ }

    public double value()
    { return value; }

    public boolean inCut(int v) ←
    { return marked[v]; } ←
        is v reachable from s in residual network?
}
```

# Finding a shortest augmenting path (cf. breadth-first search)

```
private boolean hasAugmentingPath(FlowNetwork G, int s, int t)
{
    edgeTo = new FlowEdge[G.V()];
    marked = new boolean[G.V()];

    Queue<Integer> queue = new Queue<Integer>();
    queue.enqueue(s);
    marked[s] = true;
    while (!queue.isEmpty())
    {
        int v = queue.dequeue();

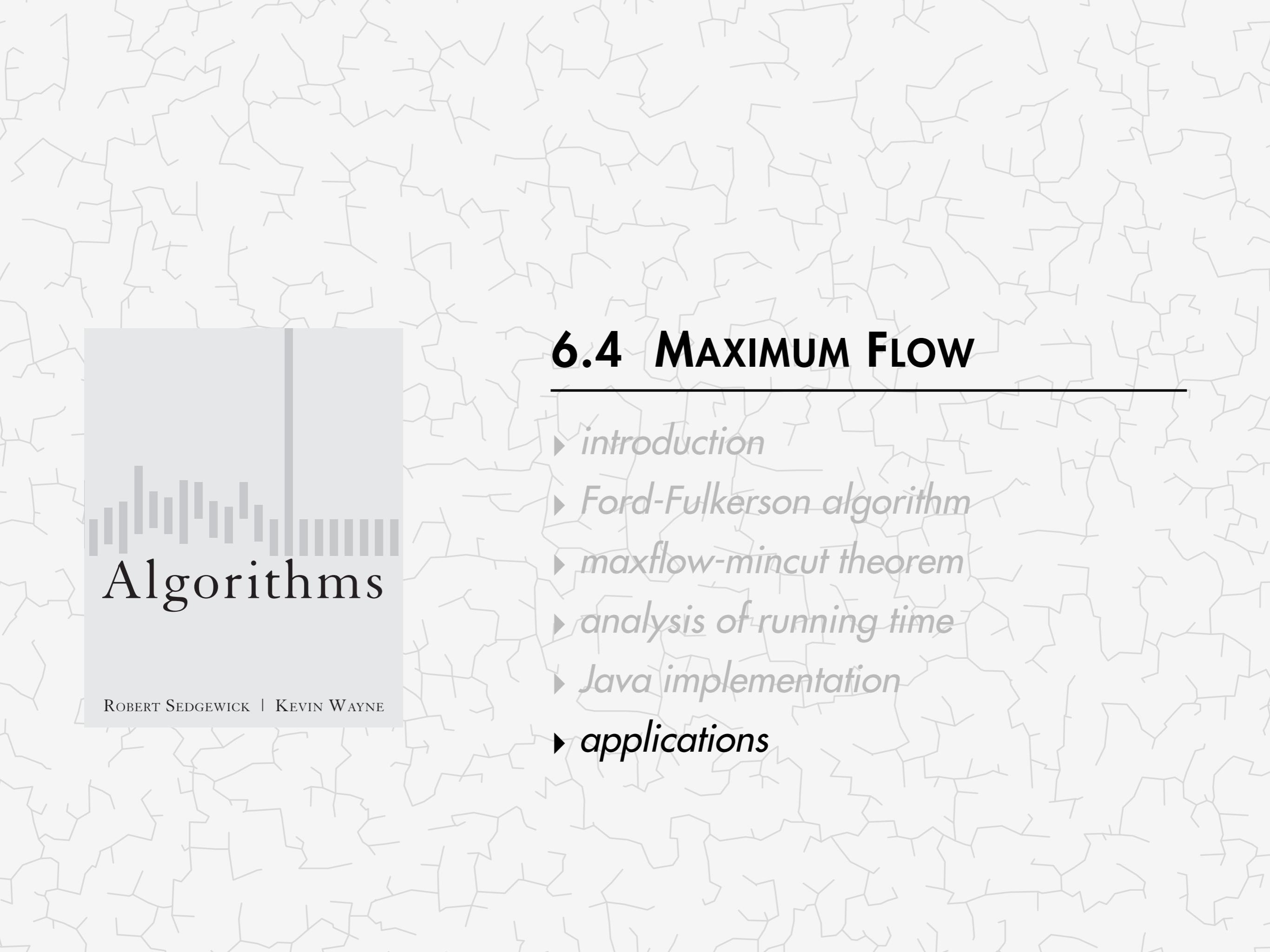
        for (FlowEdge e : G.adj(v))
        {
            int w = e.other(v);
            if (!marked[w] && (e.residualCapacityTo(w) > 0) )
            {
                edgeTo[w] = e;
                marked[w] = true; ←
                queue.enqueue(w); ←
            }
        }
    }

    return marked[t]; ← is t reachable from s in residual network?
}
```

found path from s to w in the residual network?

save last edge on path to w;  
mark w;  
add w to the queue

is t reachable from s in residual network?



# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

## 6.4 MAXIMUM FLOW

---

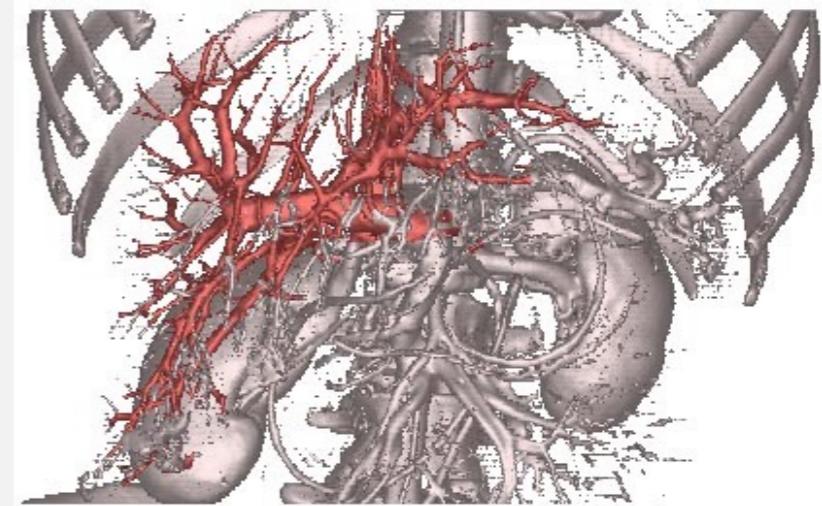
- ▶ *introduction*
- ▶ *Ford-Fulkerson algorithm*
- ▶ *maxflow-mincut theorem*
- ▶ *analysis of running time*
- ▶ *Java implementation*
- ▶ ***applications***

# Maxflow and mincut applications

---

Maxflow/mincut is a widely applicable problem-solving model.

- Data mining.
- Open-pit mining.
- Bipartite matching.
- Network reliability.
- Baseball elimination.
- Image segmentation.
- Network connectivity.
- Distributed computing.
- Security of statistical data.
- Egalitarian stable matching.
- Multi-camera scene reconstruction.
- Sensor placement for homeland security.
- Many, many, more.



liver and hepatic vascularization  
segmentation

# Bipartite matching problem

---

N students apply for N jobs.



bipartite matching problem

Each gets several offers.



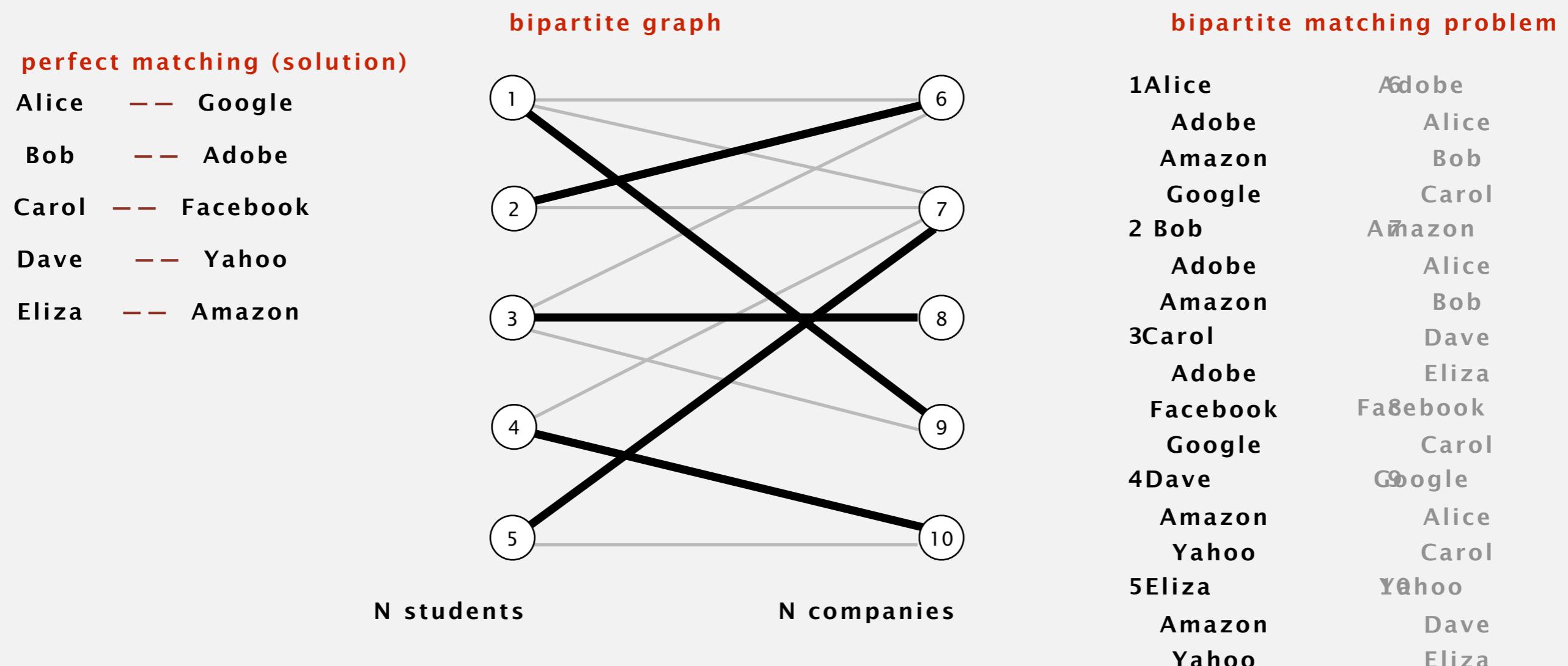
Is there a way to match all students to jobs?



|         |          |
|---------|----------|
| 1 Alice | Adobe    |
| 2 Bob   | Alice    |
| 3 Carol | Bob      |
| 4 Dave  | Carol    |
| 5 Eliza | Dave     |
| 1 Alice | Eliza    |
| 2 Bob   | Facebook |
| 3 Carol | Google   |
| 4 Dave  | Google   |
| 5 Eliza | Yahoo    |
| 1 Alice | Yahoo    |
| 2 Bob   | Adobe    |
| 3 Carol | Amazon   |
| 4 Dave  | Amazon   |
| 5 Eliza | Yahoo    |

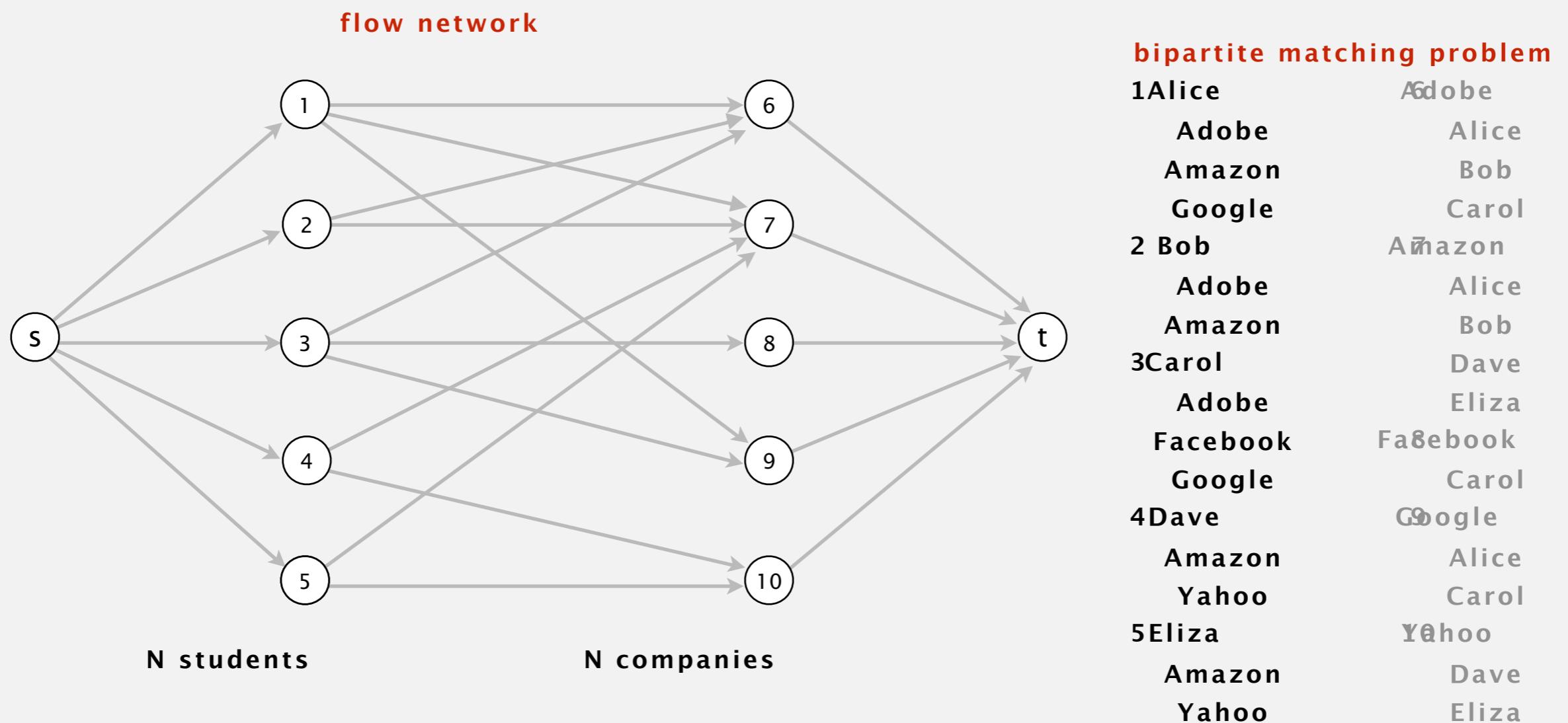
# Bipartite matching problem

Given a bipartite graph, find a perfect matching.



# Network flow formulation of bipartite matching

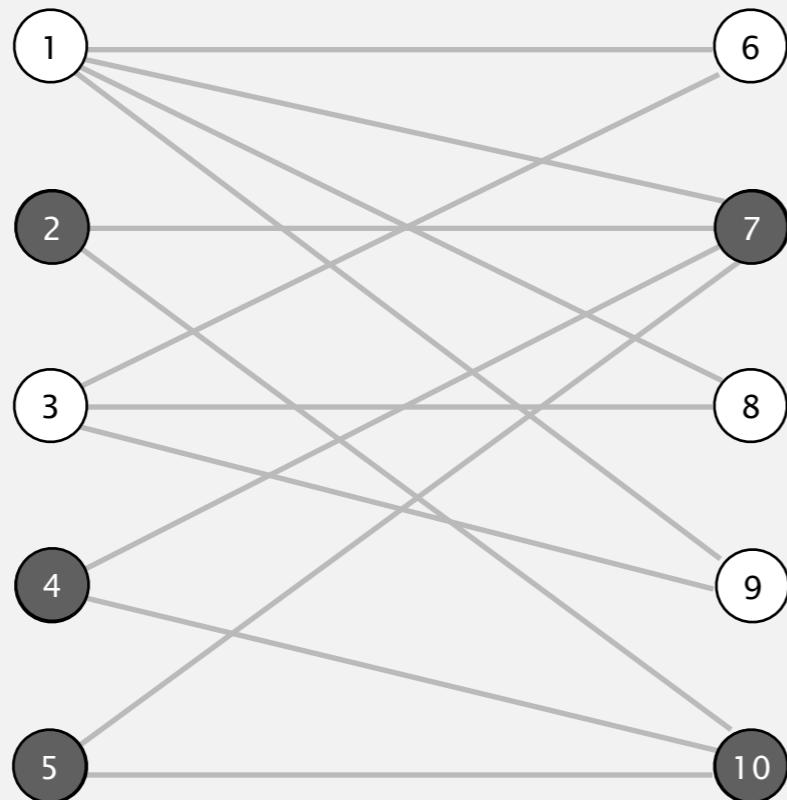
- Create  $s, t$ , one vertex for each student, and one vertex for each job.
- Add edge from  $s$  to each student (capacity 1).
- Add edge from each job to  $t$  (capacity 1).
- Add edge from student to each job offered (infinite capacity).



# What the mincut tells us

---

Goal. When no perfect matching, explain why.



$$S = \{ 2, 4, 5 \}$$
$$T = \{ 7, 10 \}$$

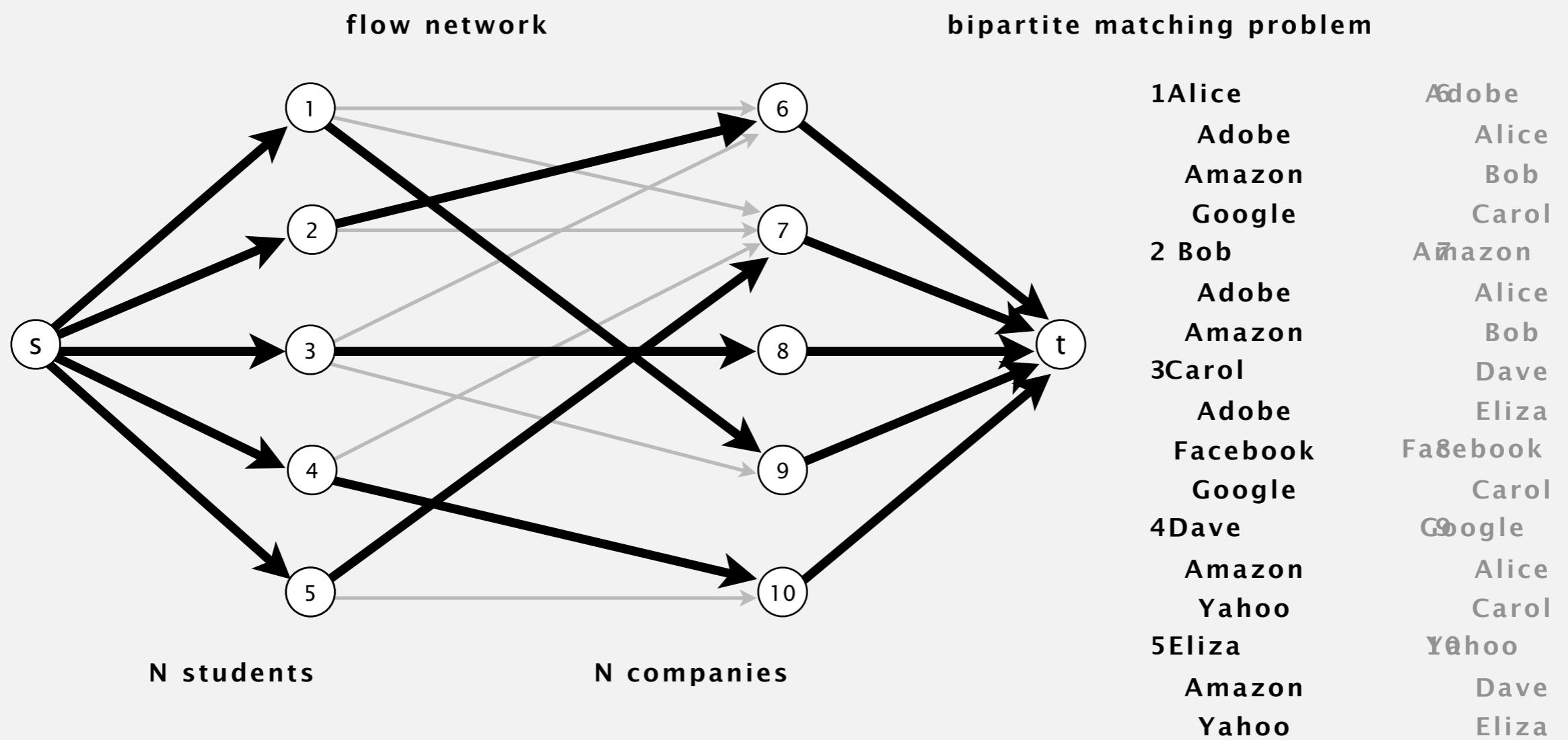
student in S  
can be matched  
only to  
companies in T

$$|S| > |T|$$

no perfect matching exists

# Network flow formulation of bipartite matching

1-1 correspondence between perfect matchings in bipartite graph and **integer-valued** maxflows of value  $N$ .

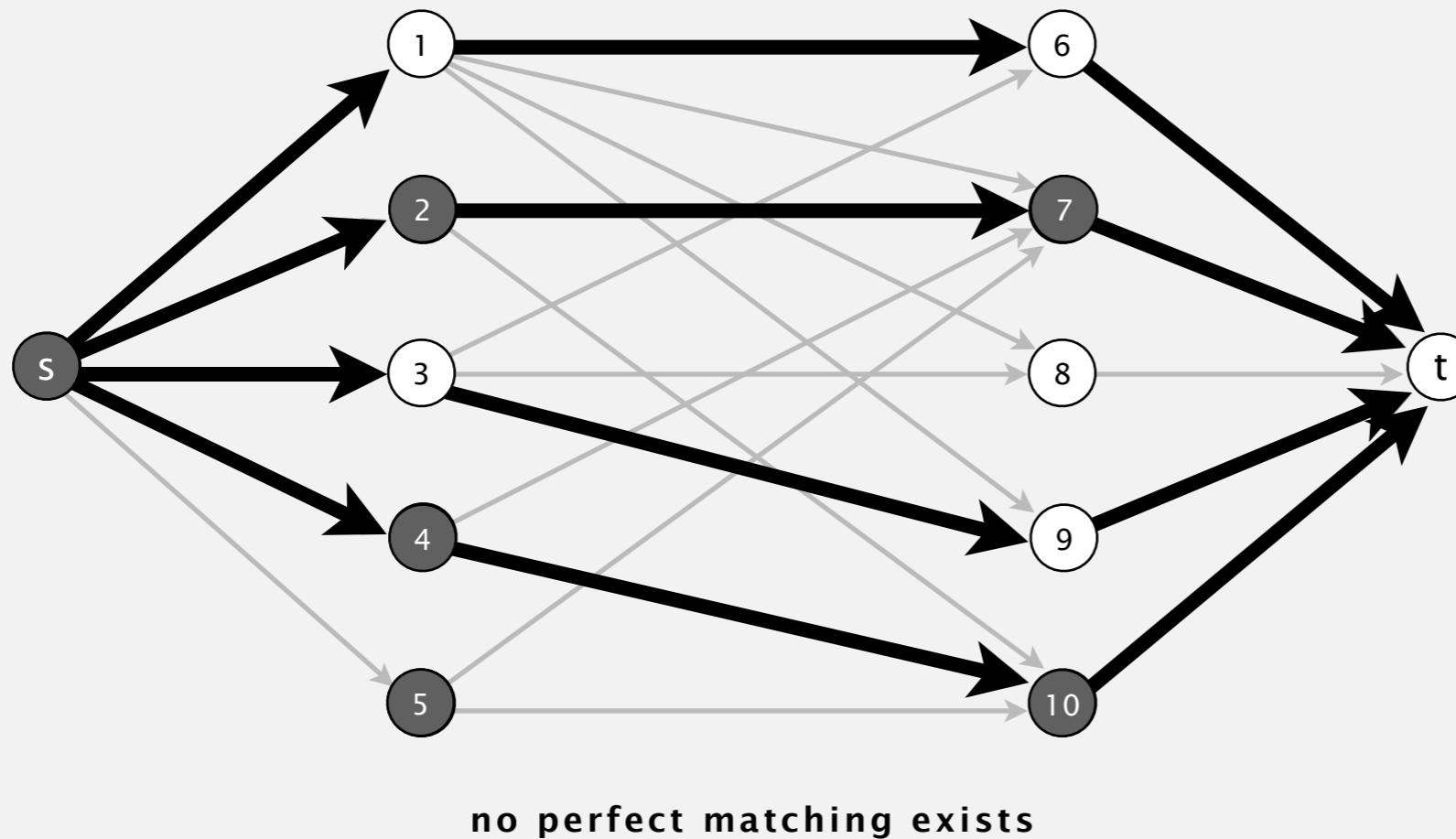


# What the mincut tells us

---

**Mincut.** Consider mincut  $(A, B)$ .

- Let  $S$  = students on  $s$  side of cut.
- Let  $T$  = companies on  $s$  side of cut.
- Fact:  $|S| > |T|$ ; students in  $S$  can be matched only to companies in  $T$ .



**Bottom line.** When no perfect matching, mincut explains why.

# Baseball elimination problem

---

Q. Which teams have a chance of finishing the season with the most wins?

| i | team  | wins | losses | to play | ATL | PHI | NYM | MON |
|---|---|------|--------|---------|-----|-----|-----|-----|
| 0 |  Atlanta   | 83   | 71     | 8       | -   | 1   | 6   | 1   |
| 1 |  Philly    | 80   | 79     | 3       | 1   | -   | 0   | 2   |
| 2 |  New York  | 78   | 78     | 6       | 6   | 0   | -   | 0   |
| 3 |  Montreal | 77   | 82     | 3       | 1   | 2   | 0   | -   |

Montreal is mathematically eliminated.

- Montreal finishes with  $\leq 80$  wins.
- Atlanta already has 83 wins.

# Baseball elimination problem

---

Q. Which teams have a chance of finishing the season with the most wins?

| i | team  | wins | losses | to play | ATL | PHI | NYM | MON |
|---|---|------|--------|---------|-----|-----|-----|-----|
| 0 |  Atlanta   | 83   | 71     | 8       | -   | 1   | 6   | 1   |
| 1 |  Philly    | 80   | 79     | 3       | 1   | -   | 0   | 2   |
| 2 |  New York  | 78   | 78     | 6       | 6   | 0   | -   | 0   |
| 3 |  Montreal | 77   | 82     | 3       | 1   | 2   | 0   | -   |

Philadelphia is mathematically eliminated.

- Philadelphia finishes with  $\leq 83$  wins.
- Either New York or Atlanta will finish with  $\geq 84$  wins.

**Observation.** Answer depends not only on how many games already won and left to play, but on whom they're against.

# Baseball elimination problem

Q. Which teams have a chance of finishing the season with the most wins?

| i | team  | wins | losses | to play | NYY | BAL | BOS | TOR | DET |
|---|---|------|--------|---------|-----|-----|-----|-----|-----|
| 0 |  New York  | 75   | 59     | 28      | -   | 3   | 8   | 7   | 3   |
| 1 |  Baltimore | 71   | 63     | 28      | 3   | -   | 2   | 7   | 4   |
| 2 |  Boston    | 69   | 66     | 27      | 8   | 2   | -   | 0   | 0   |
| 3 |  Toronto  | 63   | 72     | 27      | 7   | 7   | 0   | -   | 0   |
| 4 |  Detroit | 49   | 86     | 27      | 3   | 4   | 0   | 0   | -   |

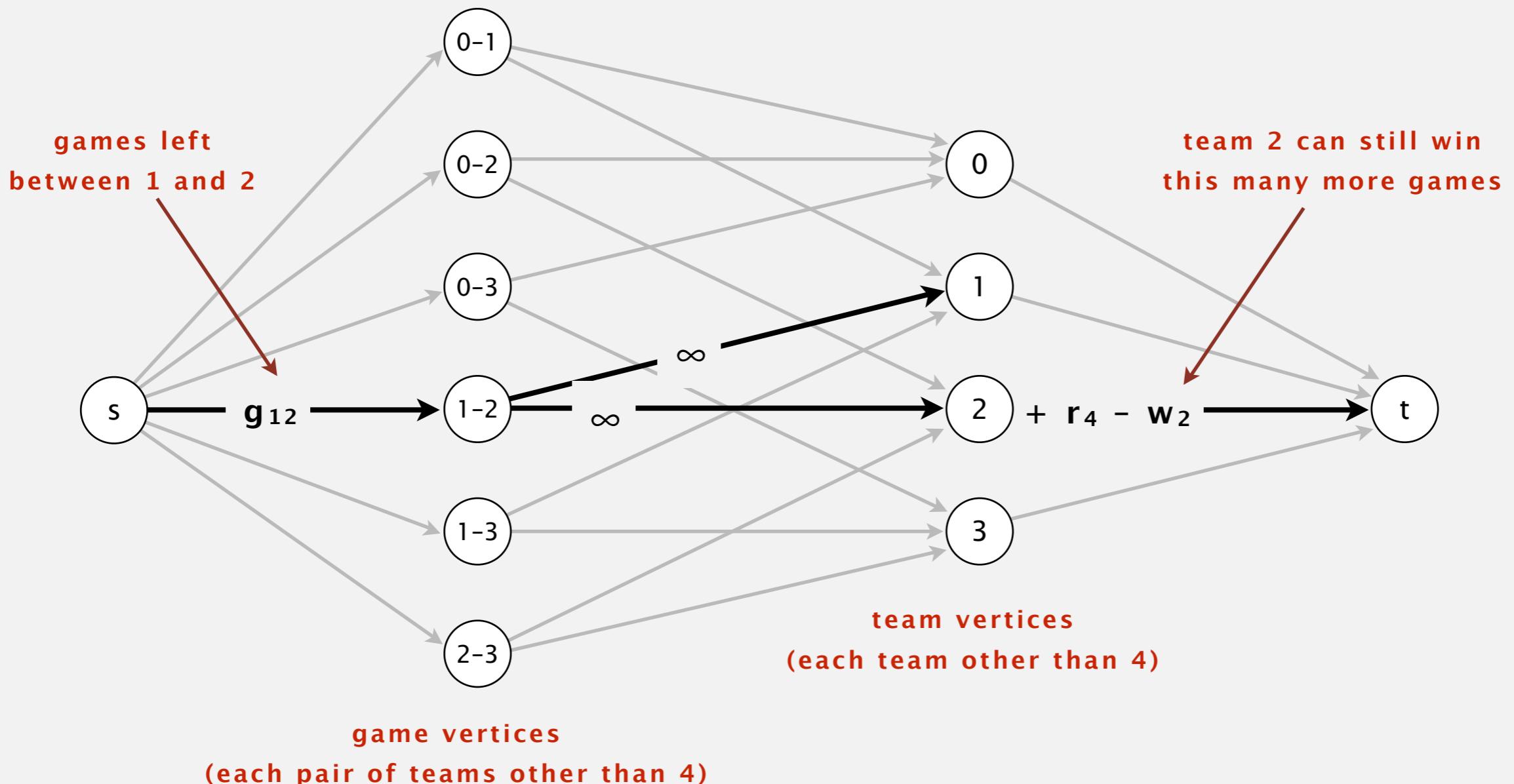
AL East (August 30, 1996)

Detroit is mathematically eliminated.

- Detroit finishes with  $\leq 76$  wins.
- Wins for  $R = \{ \text{NYY}, \text{BAL}, \text{BOS}, \text{TOR} \} = 278$ .
- Remaining games among  $\{ \text{NYY}, \text{BAL}, \text{BOS}, \text{TOR} \} = 3 + 8 + 7 + 2 + 7 = 27$ .
- Average team in  $R$  wins  $305/4 = 76.25$  games.

# Baseball elimination problem: maxflow formulation

**Intuition.** Remaining games flow from  $s$  to  $t$ .



**Fact.** Team 4 not eliminated iff all edges pointing from  $s$  are full in maxflow.

# Maximum flow algorithms: theory

(Yet another) holy grail for theoretical computer scientists.

| year | method                          | worst case              | discovered by        |
|------|---------------------------------|-------------------------|----------------------|
| 1951 | <b>simplex</b>                  | $E^3 U$                 | Dantzig              |
| 1955 | <b>augmenting path</b>          | $E^2 U$                 | Ford-Fulkerson       |
| 1970 | <b>shortest augmenting path</b> | $E^3$                   | Dinitz, Edmonds-Karp |
| 1970 | <b>fattest augmenting path</b>  | $E^2 \log E \log(EU)$   | Dinitz, Edmonds-Karp |
| 1977 | <b>blocking flow</b>            | $E^{5/2}$               | Cherkasky            |
| 1978 | <b>blocking flow</b>            | $E^{7/3}$               | Galil                |
| 1983 | <b>dynamic trees</b>            | $E^2 \log E$            | Sleator-Tarjan       |
| 1985 | <b>capacity scaling</b>         | $E^2 \log U$            | Gabow                |
| 1997 | <b>length function</b>          | $E^{3/2} \log E \log U$ | Goldberg-Rao         |
| 2012 | <b>compact network</b>          | $E^2 / \log E$          | Orlin                |
| ?    | ?                               | $E$                     | ?                    |

maxflow algorithms for sparse digraphs with  $E$  edges, integer capacities between 1 and  $U$

# Maximum flow algorithms: practice

**Warning.** Worst-case order-of-growth is generally not useful for predicting or comparing maxflow algorithm performance in practice.

**Best in practice.** Push-relabel method with gap relabeling:  $E^{3/2}$ .

## On Implementing Push-Relabel Method for the Maximum Flow Problem

Boris V. Cherkassky<sup>1</sup> and Andrew V. Goldberg<sup>2</sup>

<sup>1</sup> Central Institute for Economics and Mathematics,  
Krasikova St. 32, 117418, Moscow, Russia  
*cher@cemi.msk.su*

<sup>2</sup> Computer Science Department, Stanford University  
Stanford, CA 94305, USA  
*goldberg@cs.stanford.edu*

**Abstract.** We study efficient implementations of the push-relabel method for the maximum flow problem. The resulting codes are faster than the previous codes, and much faster on some problem families. The speedup is due to the combination of heuristics used in our implementations. We also exhibit a family of problems for which the running time of all known methods seem to have a roughly quadratic growth rate.



European Journal of Operational Research 97 (1997) 509–542

EUROPEAN  
JOURNAL  
OF OPERATIONAL  
RESEARCH

### Theory and Methodology

## Computational investigations of maximum flow algorithms

Ravindra K. Ahuja <sup>a</sup>, Murali Kodialam <sup>b</sup>, Ajay K. Mishra <sup>c</sup>, James B. Orlin <sup>d,\*</sup>

<sup>a</sup> Department of Industrial and Management Engineering, Indian Institute of Technology, Kanpur, 208 016, India

<sup>b</sup> AT & T Bell Laboratories, Holmdel, NJ 07733, USA

<sup>c</sup> Katz Graduate School of Business, University of Pittsburgh, Pittsburgh, PA 15260, USA

<sup>d</sup> Sloan School of Management, Massachusetts Institute of Technology, Cambridge, MA 02139, USA

Received 30 August 1995; accepted 27 June 1996

---

**Mincut problem.** Find an  $st$ -cut of minimum capacity.

**Maxflow problem.** Find an  $st$ -flow of maximum value.

**Duality.** Value of the maxflow = capacity of mincut.

**Proven successful approaches.**

- Ford-Fulkerson (various augmenting-path strategies).
- Preflow-push (various versions).

**Open research challenges.**

- Practice: solve real-world maxflow/mincut problems in linear time.
- Theory: prove it for worst-case inputs.
- Still much to be learned!