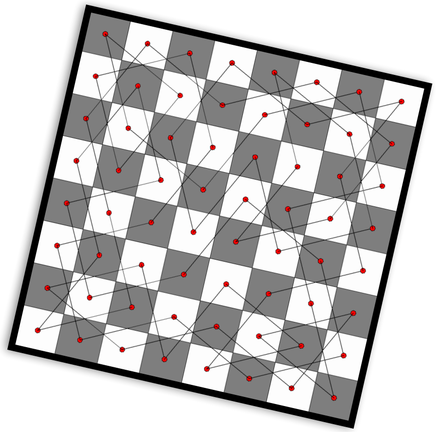
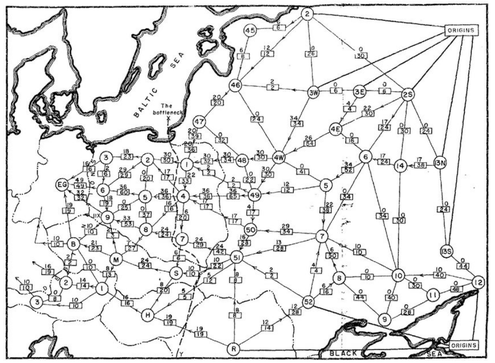
# LINFO2266 : Network Flows



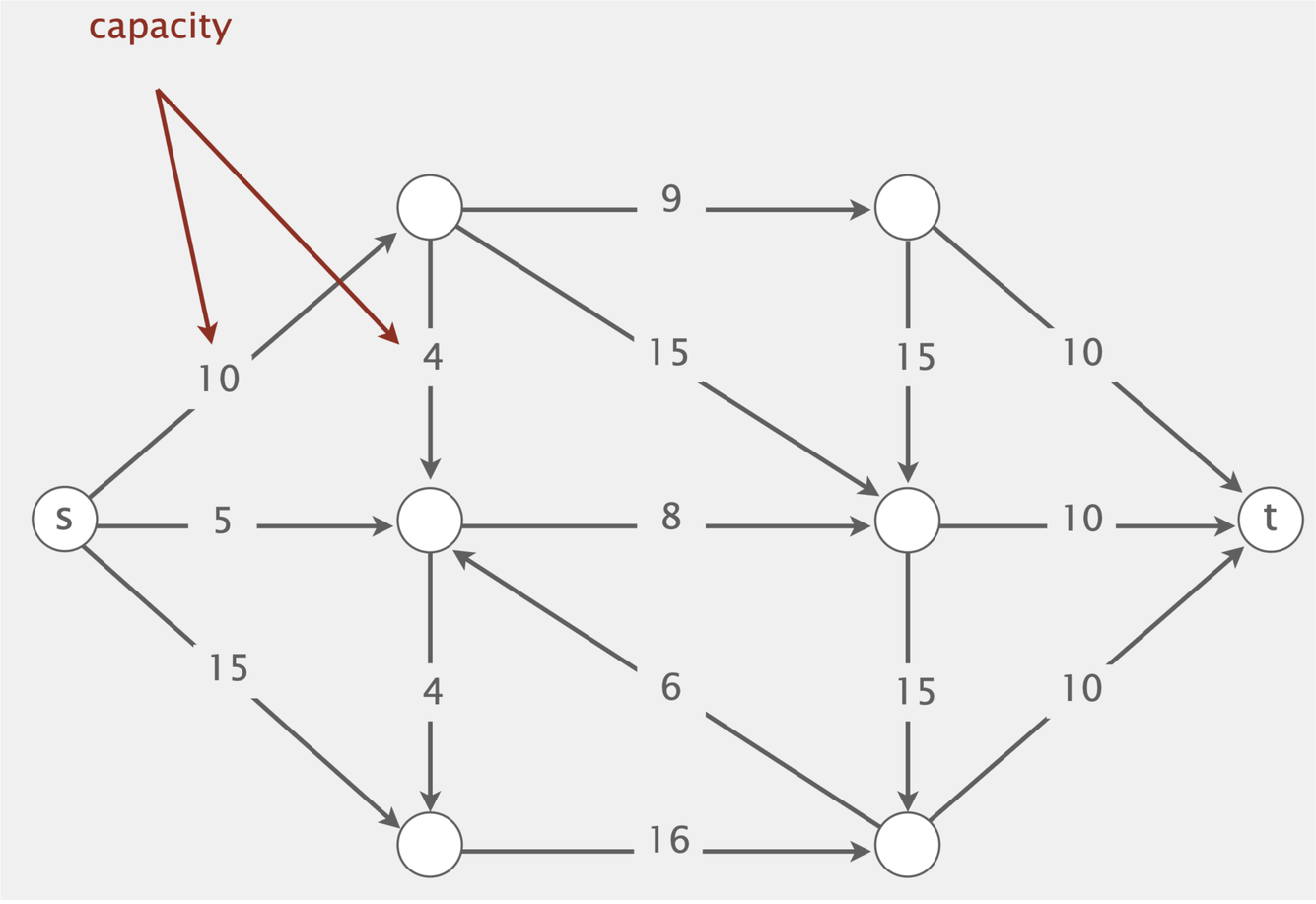



Many figures and material from: https://algs4.cs.princeton.edu/lectures/

# MinCut Problem

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

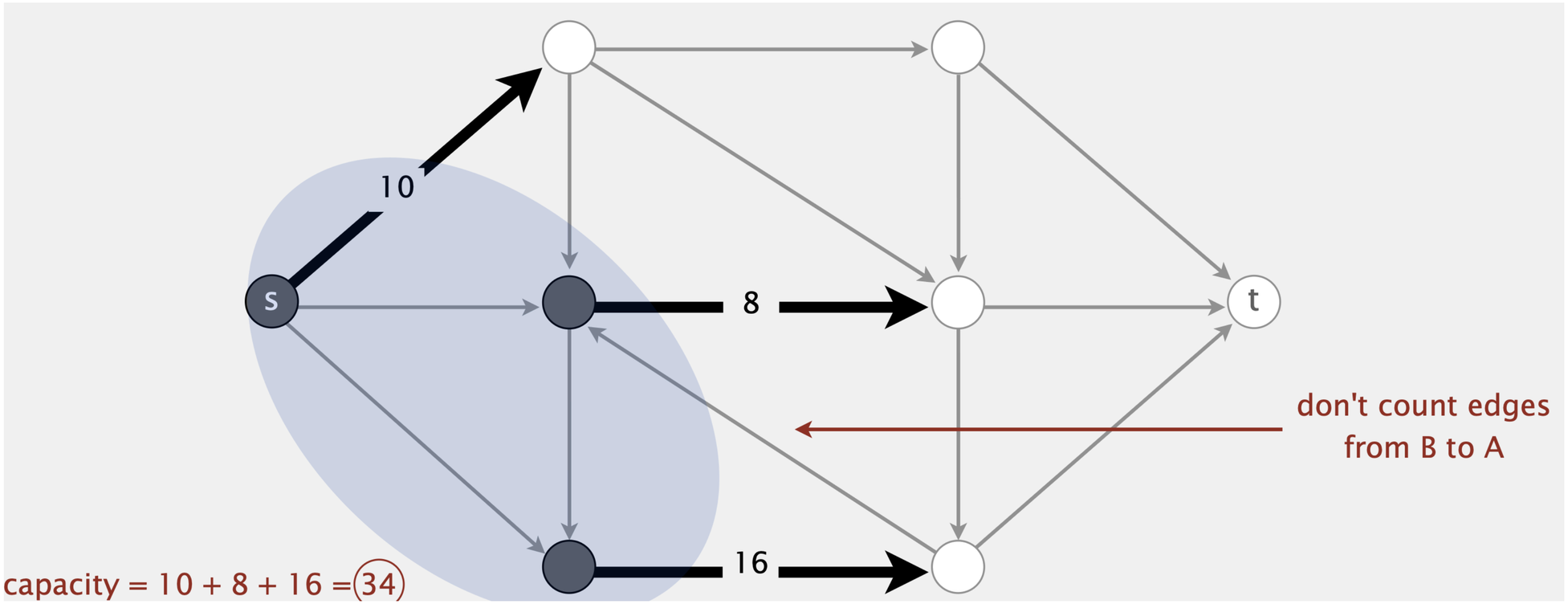
Each edge has a positive integer capacity



# 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 $B$.

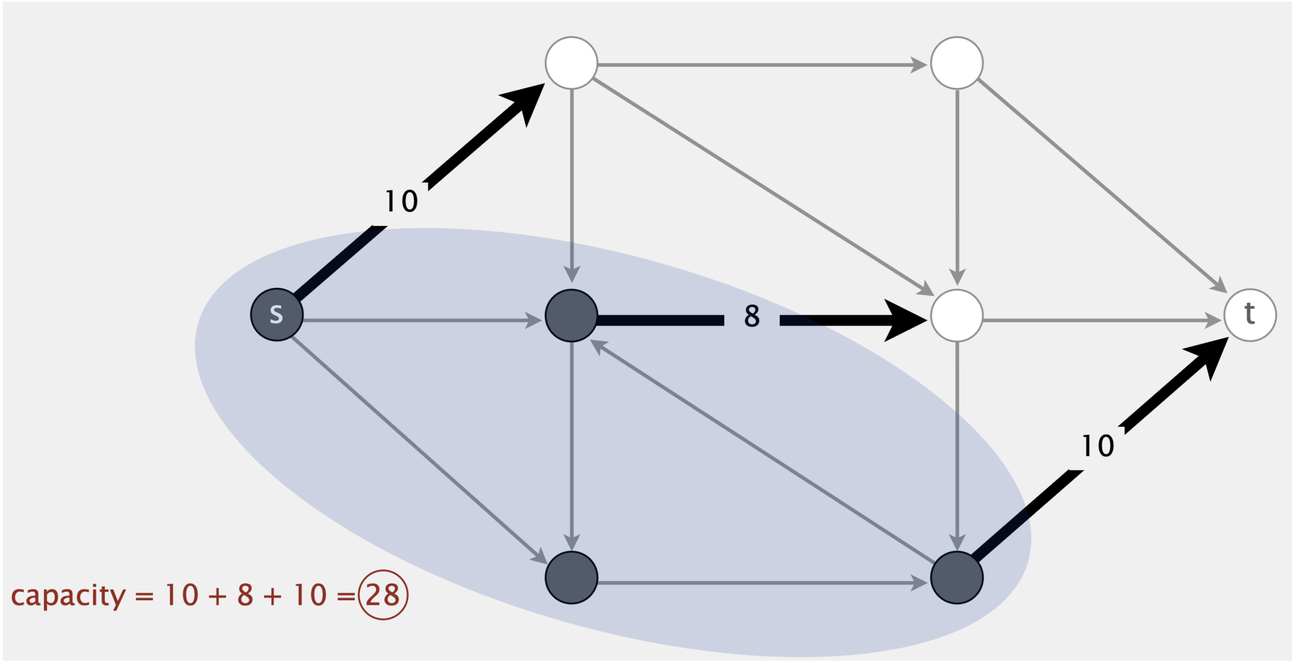


# MinCut Problem

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 $B$.

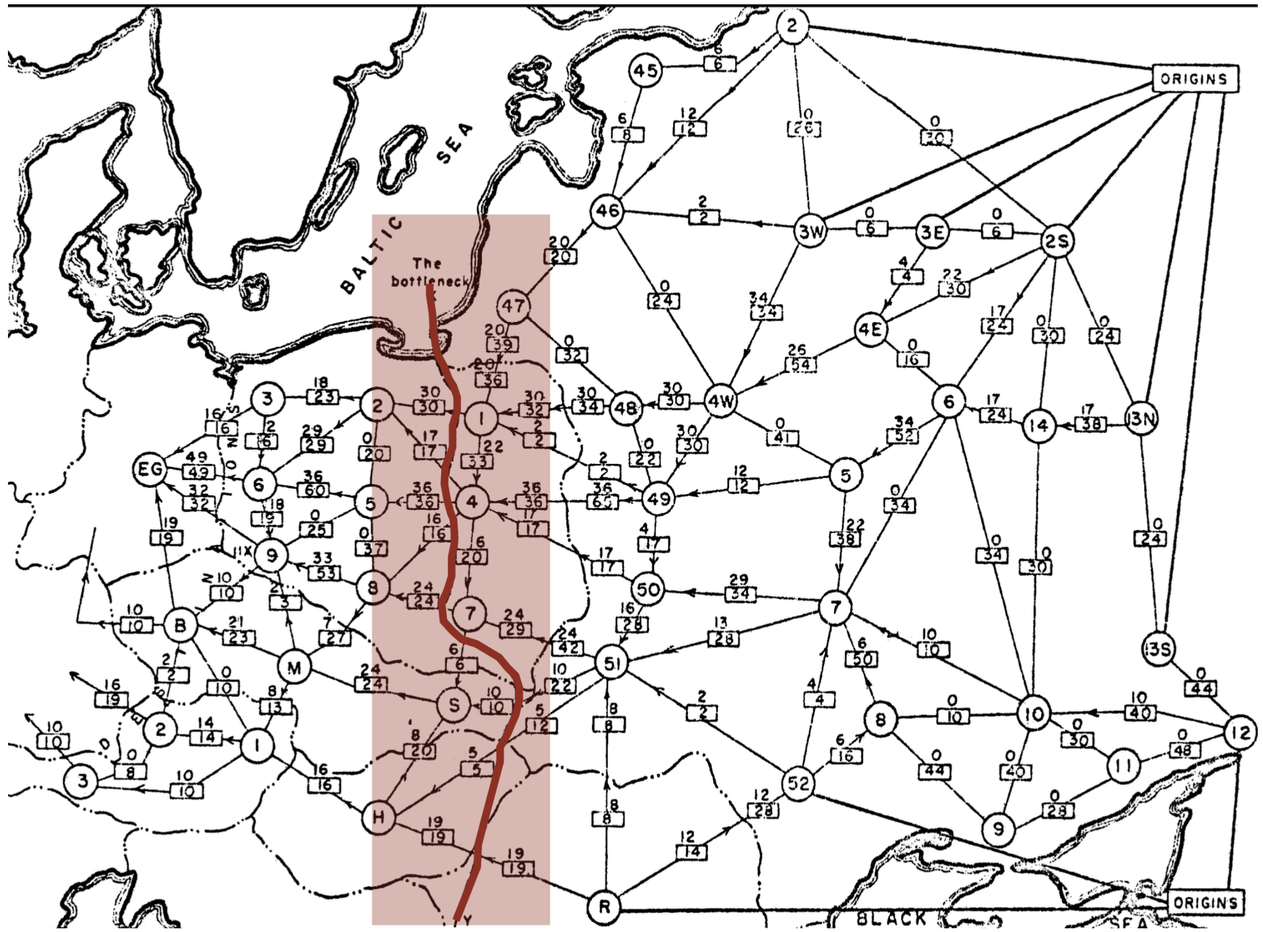
Def. Its capacity is the sum of the capacities of the edges from $A$ to $B$.

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



# MinCut Application (RAND 1950s)

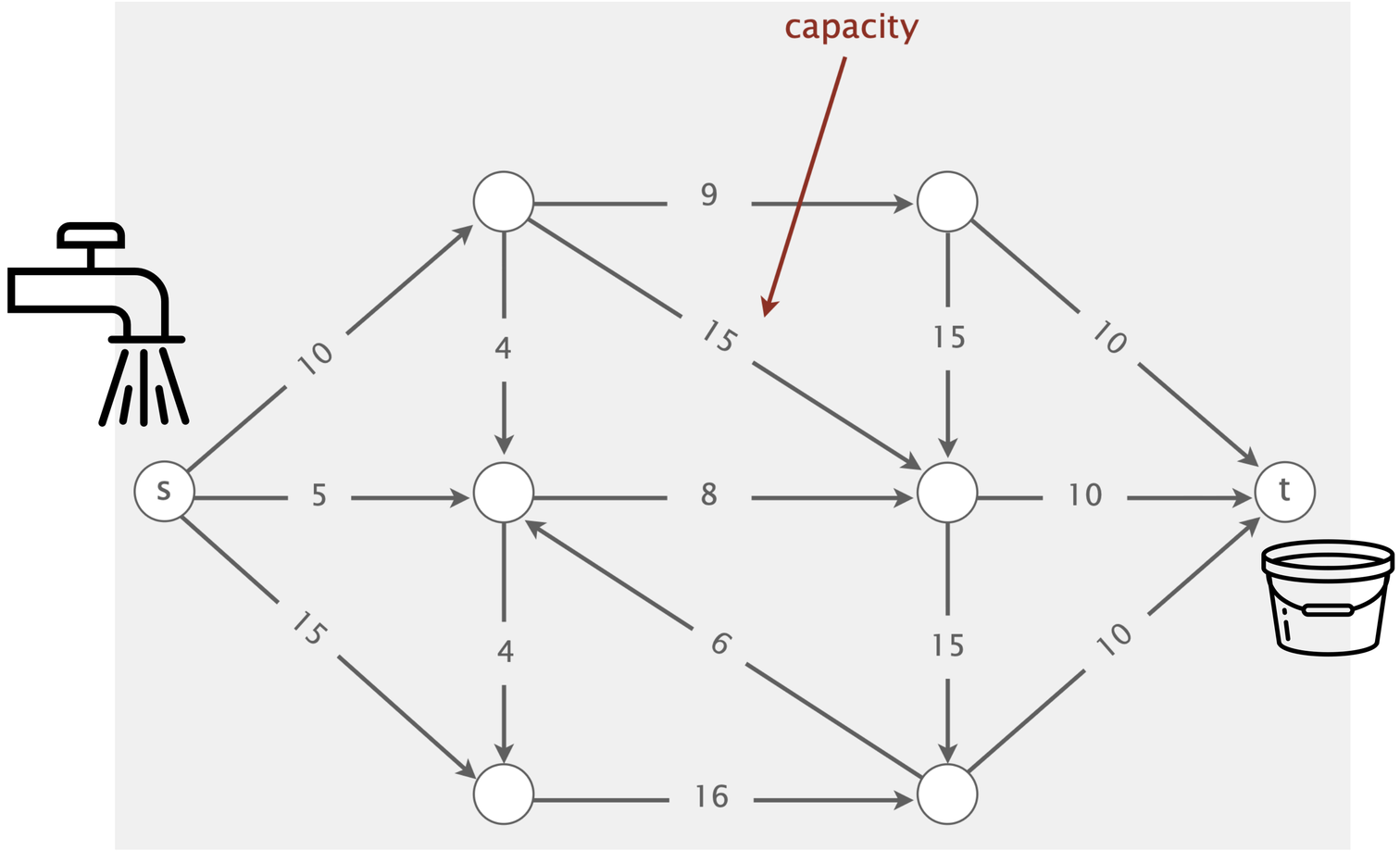
"Free world" goal. Cut supplies (if cold war turns into real war).




**rail network connecting Soviet Union with Eastern European countries (map declassified by Pentagon in 1999)**
From Harris and Ross [1955]: Schematic diagram of the railway network

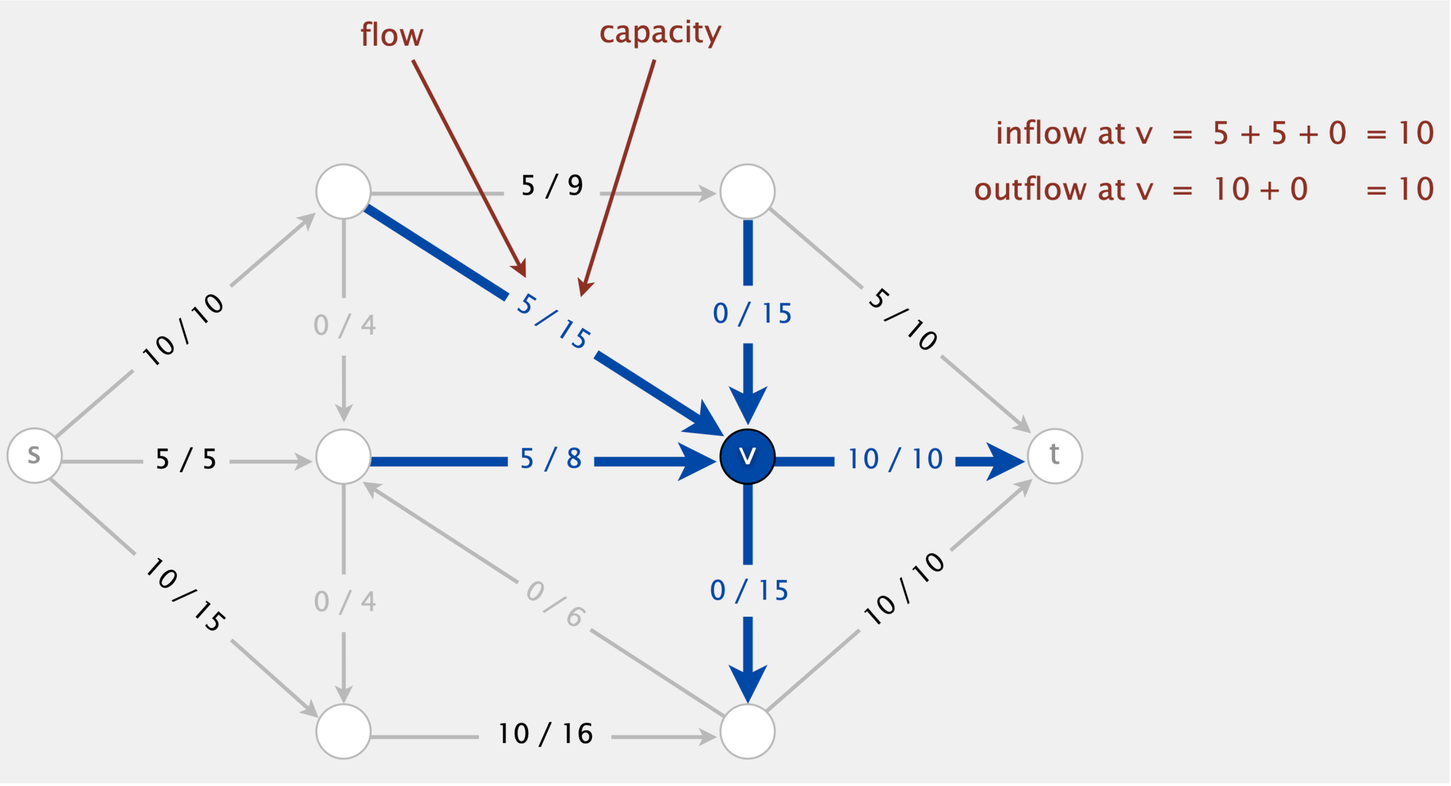
# MaxFlow Problem

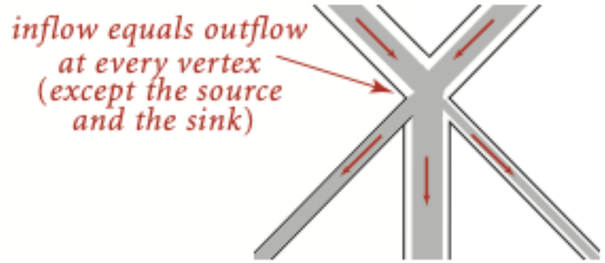
Input. An edge-weighted digraph, source vertex $s$, and target vertex $t$.



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

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



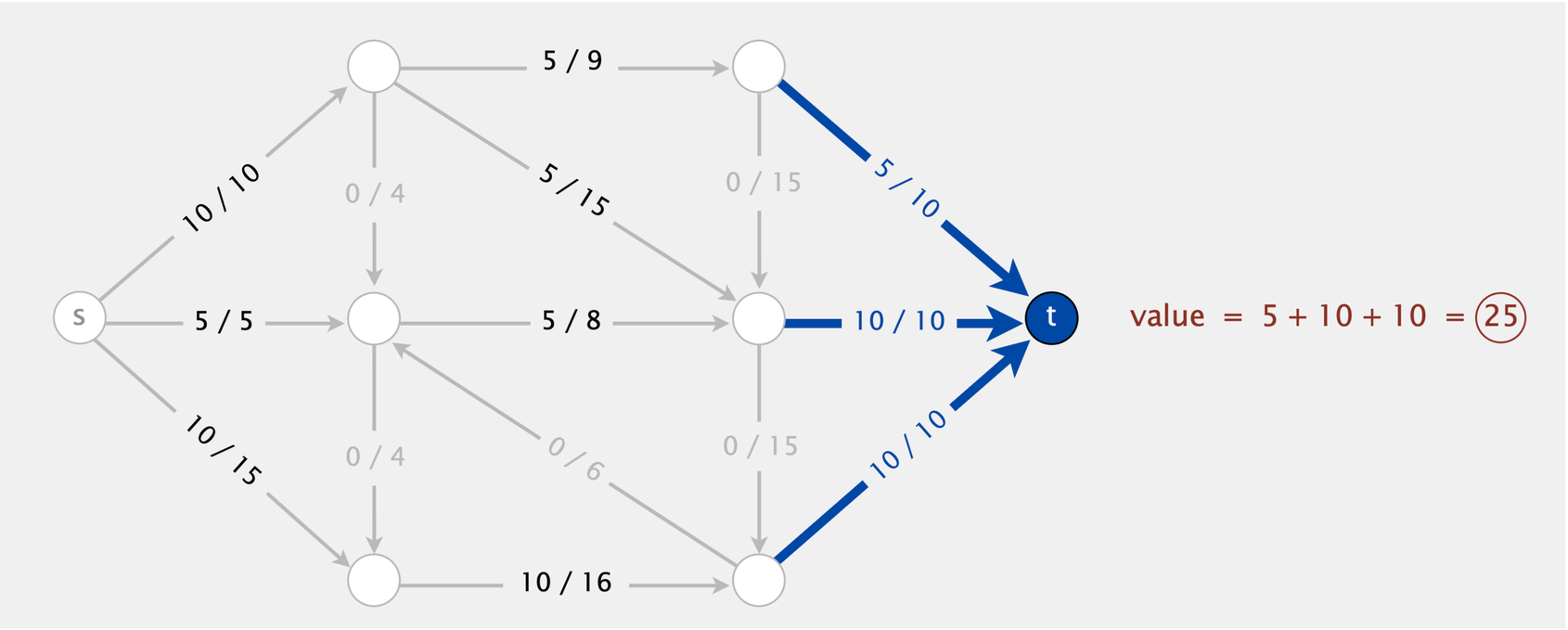

Local equilibrium in a flow network

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

- Capacity constraint: $0 \le$ edge's flow $\le$ 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 edge point from t or to s

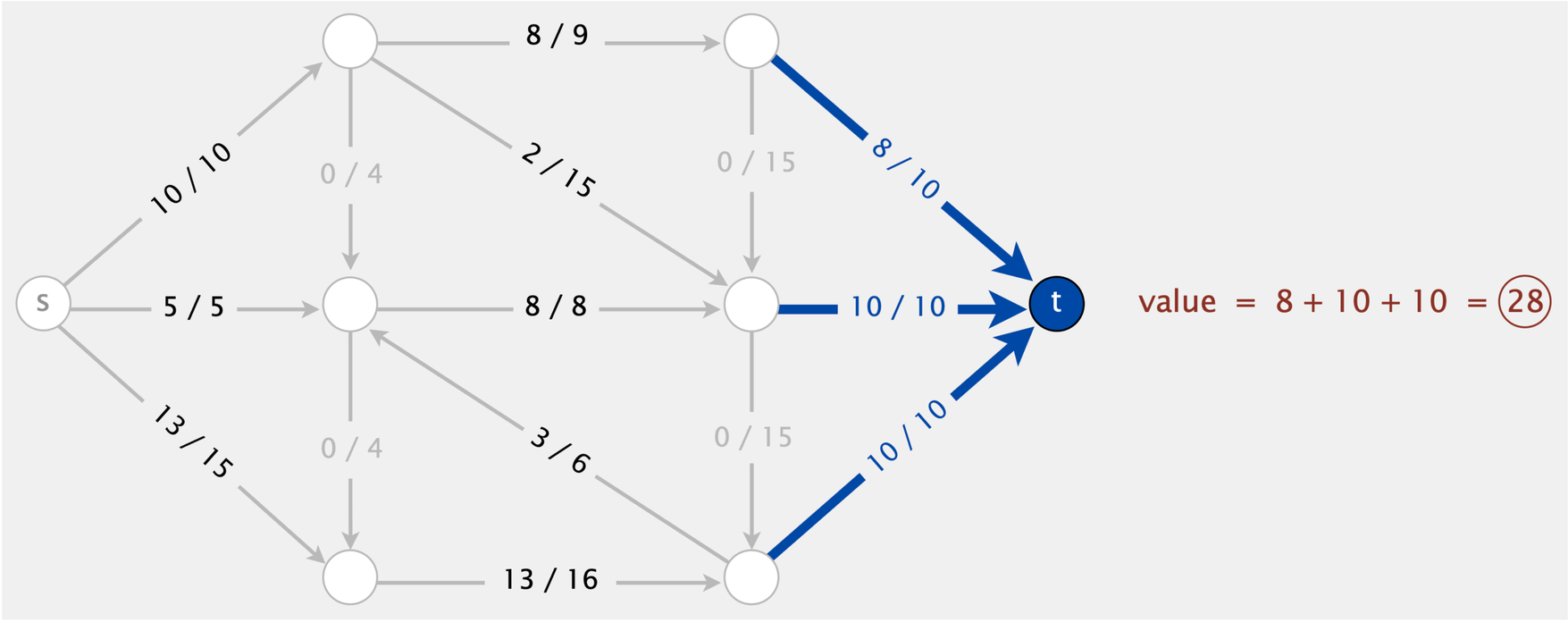


Def. An $st$-flow (flow) is an assignment of values to the edges such that:

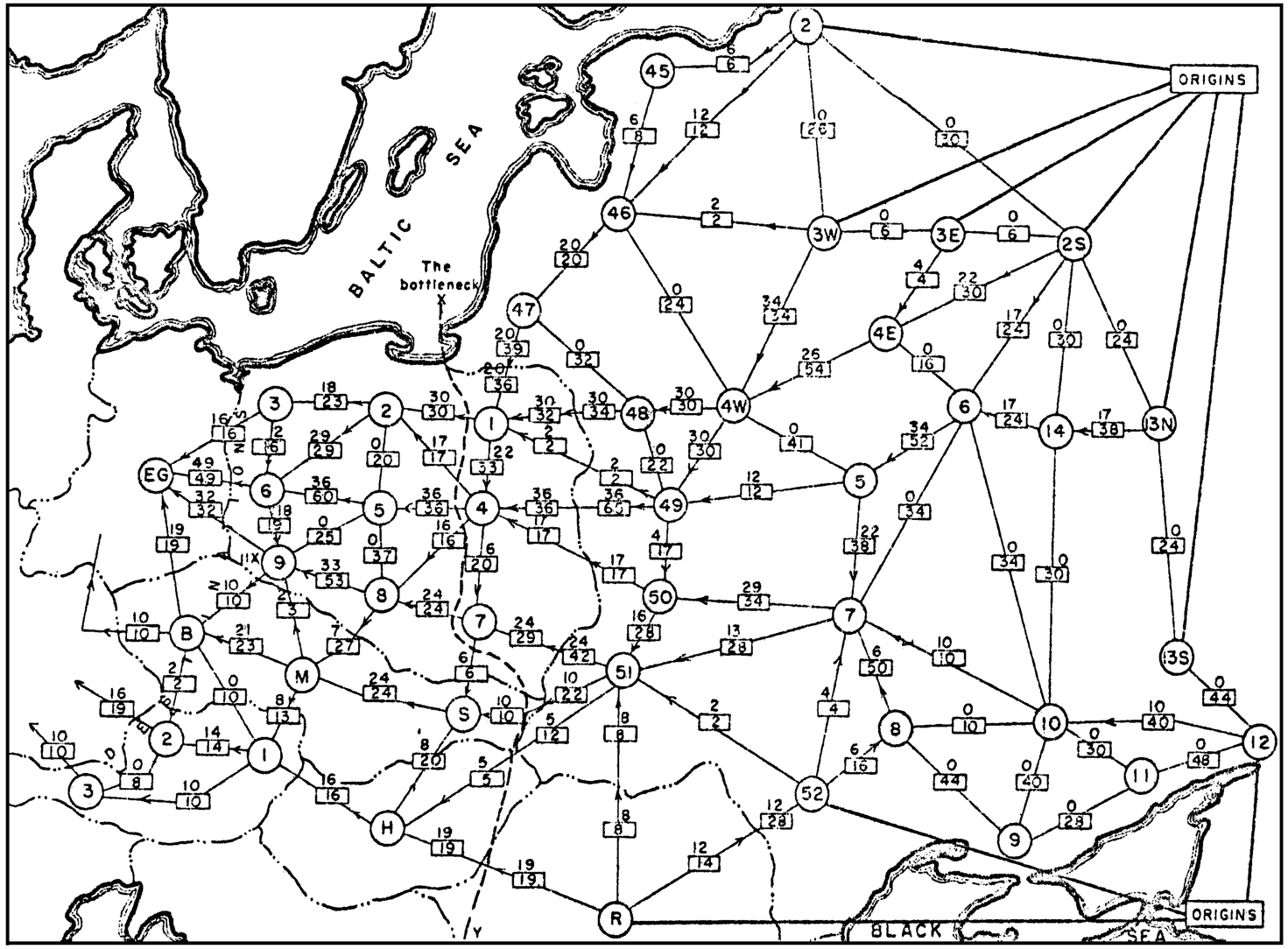
- Capacity constraint: $0 \le$ edge's flow $\le$ edge's capacity.
- Local equilibrium: inflow = outflow at every vertex (except $s$ and $t$).

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

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



# Max-Flow Application: compute the maximum supply in the train network



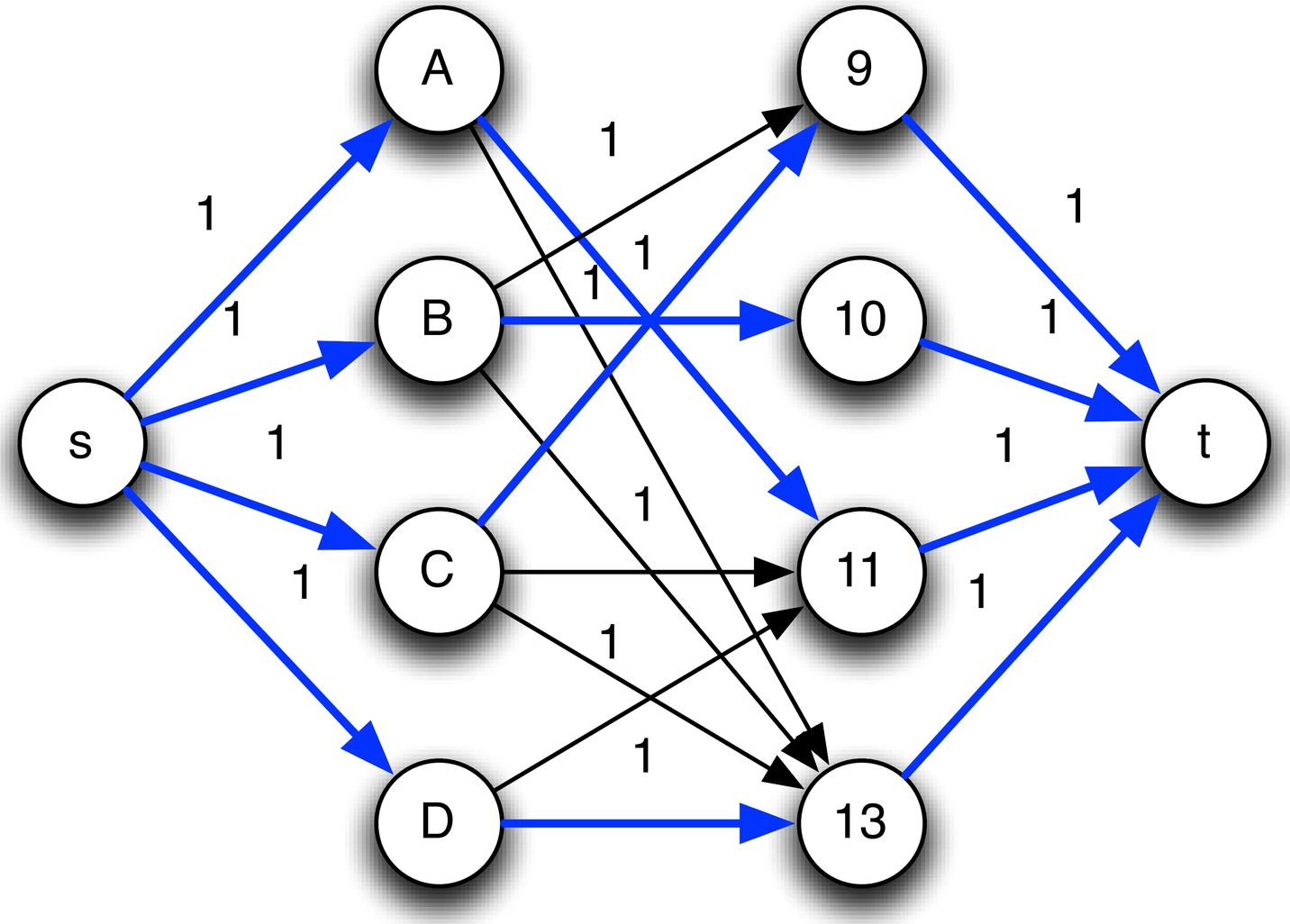
# Max-Flow Application

| Person | Available |
|---|---|
| Alice | 11h00<br>13h00 |
| Benoît | 9h00<br>10h00<br>13h00 |
| Clotilde | 9h00<br>11h00<br>13h00 |
| Dany | 11h00<br>13h00 |

- 4 persons must give a seminar in a single room
- 4 slots have been suggested
  - 9h00, 10h00, 11h00 and 13h00.
- Each speaker was asked to give possible slots.
- **Problem:** Create a schedule

Max flow problem can help us!

# Max-Flow Schedule Solution

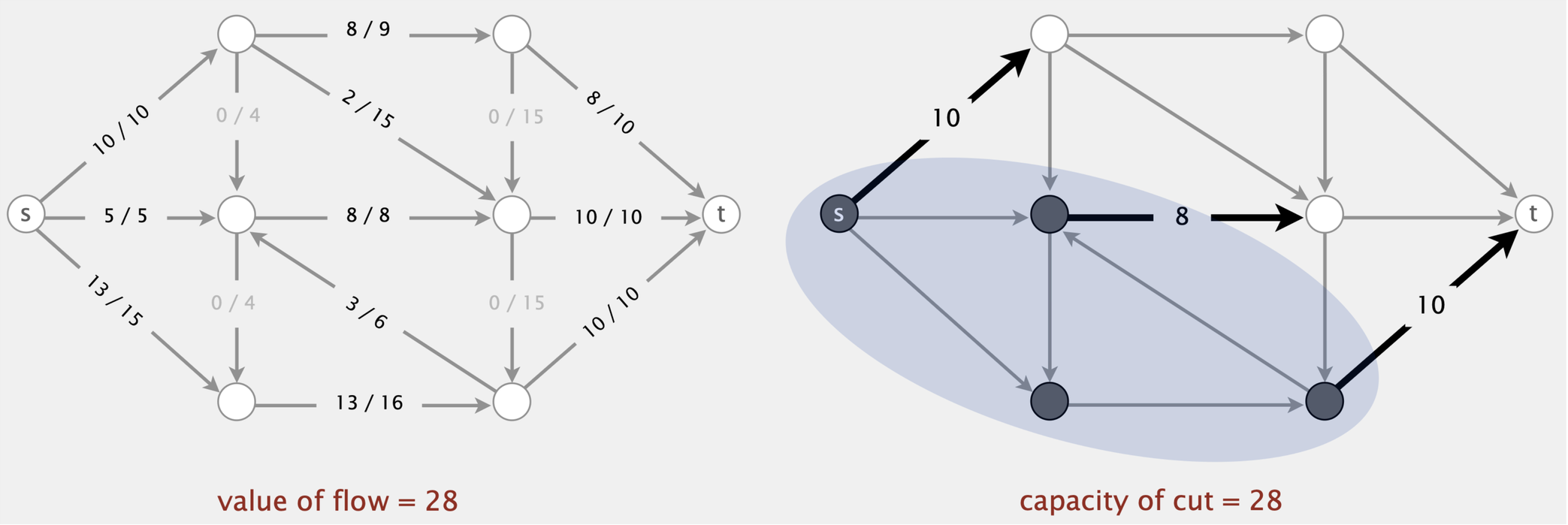


# Summary

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.



# Linear Programming Formulation

- MaxFlow

$$\text{maximize} \sum_{(s,v)\in E} f_{s,v}$$

$$\text{subject to} \sum_{(u,v)\in E} f_{uv} = \sum_{(v,w)\in E} f_{vw} \qquad \forall v \in V\backslash\{s,t\}$$

Local Equilibrium Constraints

$$0 \le f_e \le c_e \qquad \forall e \in E$$

MinCut is the Dual of MaxFlow

- MinCut

$$\text{minimize} \sum_{(s,v)\in E} c_{uv} \cdot z_{uv}$$

$$\text{subject to } z_{uv} \ge x_u - x_v \qquad \forall v \in V\backslash\{s,t\}$$

$$x_s = 1$$

$$x_u = 0$$

$$z_e \in \{0,1\} \qquad \forall e \in E$$

$$x_v \in \{0,1\} \qquad \forall v \in V$$

# Quick intro do duality in Linear Programming

*Primal*

Maximize $\mathbf{c}^T\mathbf{x}$
subject to $A\mathbf{x} \leq \mathbf{b}$,
$\mathbf{x} \geq 0$

*Dual*

Minimize $\mathbf{b}^T\mathbf{y}$
subject to $A^T\mathbf{y} \geq \mathbf{c}$,
$\mathbf{y} \geq 0$

Weak Duality Theorem:

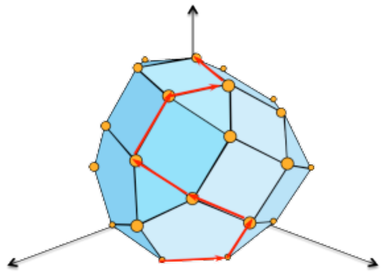
For each feasible solution $\mathbf{x}$ of the primal and each feasible solution $\mathbf{y}$ of the dual: $\mathbf{c}^T\mathbf{x} \leq \mathbf{b}^T\mathbf{y}$.

Strong Duality Theorem:

If one of the two problems has an optimal solution, so does the other one and the bounds given by the weak duality theorem are tight, i.e.: $\max_{\mathbf{x}} \mathbf{c}^T\mathbf{x} = \min_{\mathbf{y}} \mathbf{b}^T\mathbf{y}$

# Simplex can solve Max-Flow Min-cut

- Why? because the vertices of the feasible region correspond to integer-valued solutions



- How do we know? Because *A* (the matrix of constraints) is totally unimodular (all sub-determinants of the matrix are either 0, 1 or -1)

$$\begin{aligned} &\text{Maximize } \mathbf{c}^{\mathrm{T}}\mathbf{x} \\ &\text{subject to } A\mathbf{x} \leq \mathbf{b}, \\ &\qquad\qquad \mathbf{x} \geq 0 \end{aligned}$$

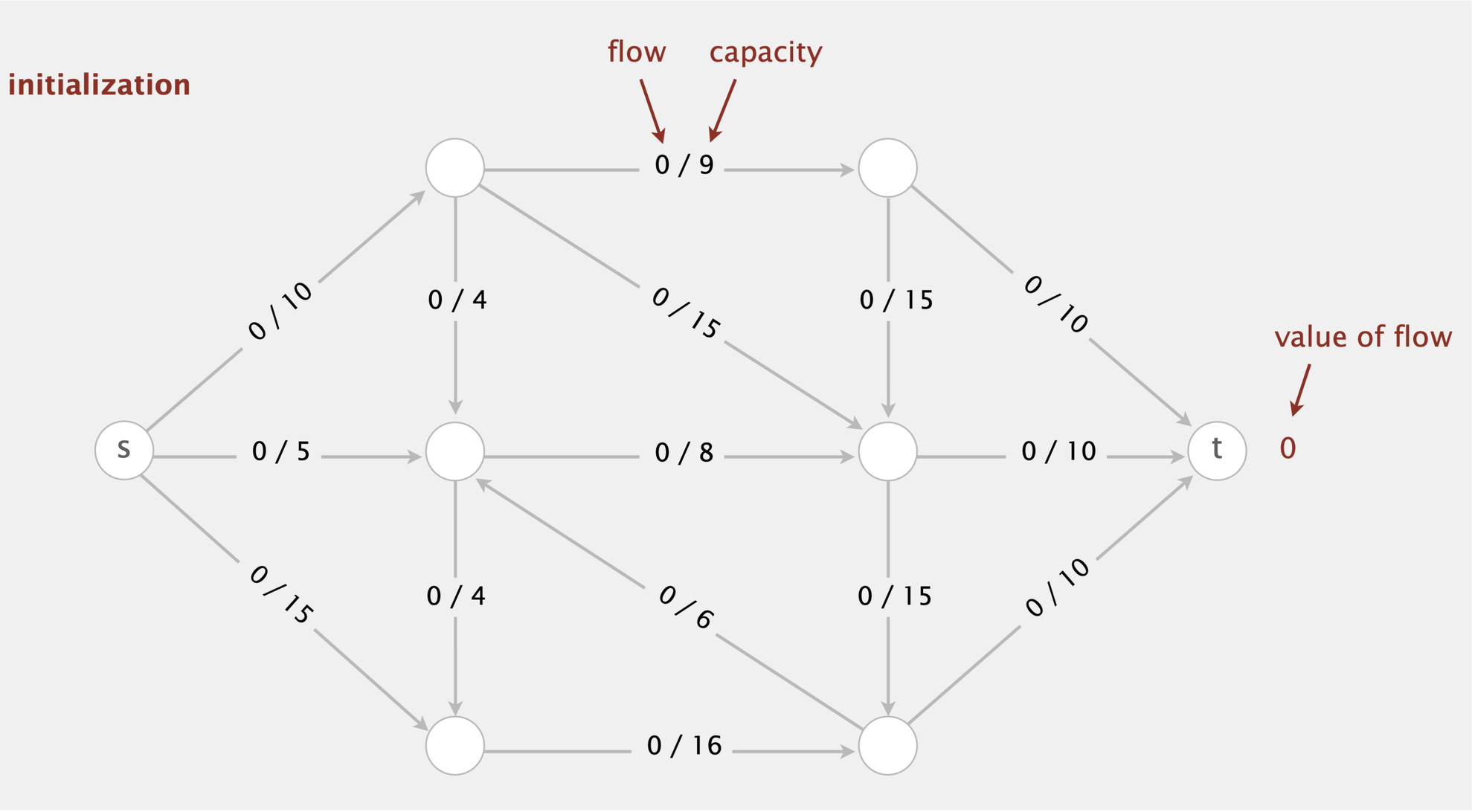
- Simplex is not the most efficient approach for solving these (graph) problems, we will see a more efficient approach.

# Ford-Fulkerson Algorithm

- Start from a zero flow
- Keep improving the flow while keeping it feasible (equilibrium constraints)
- Until not possible to improve it

# Ford-Fulkerson (augmenting path) Algorithm

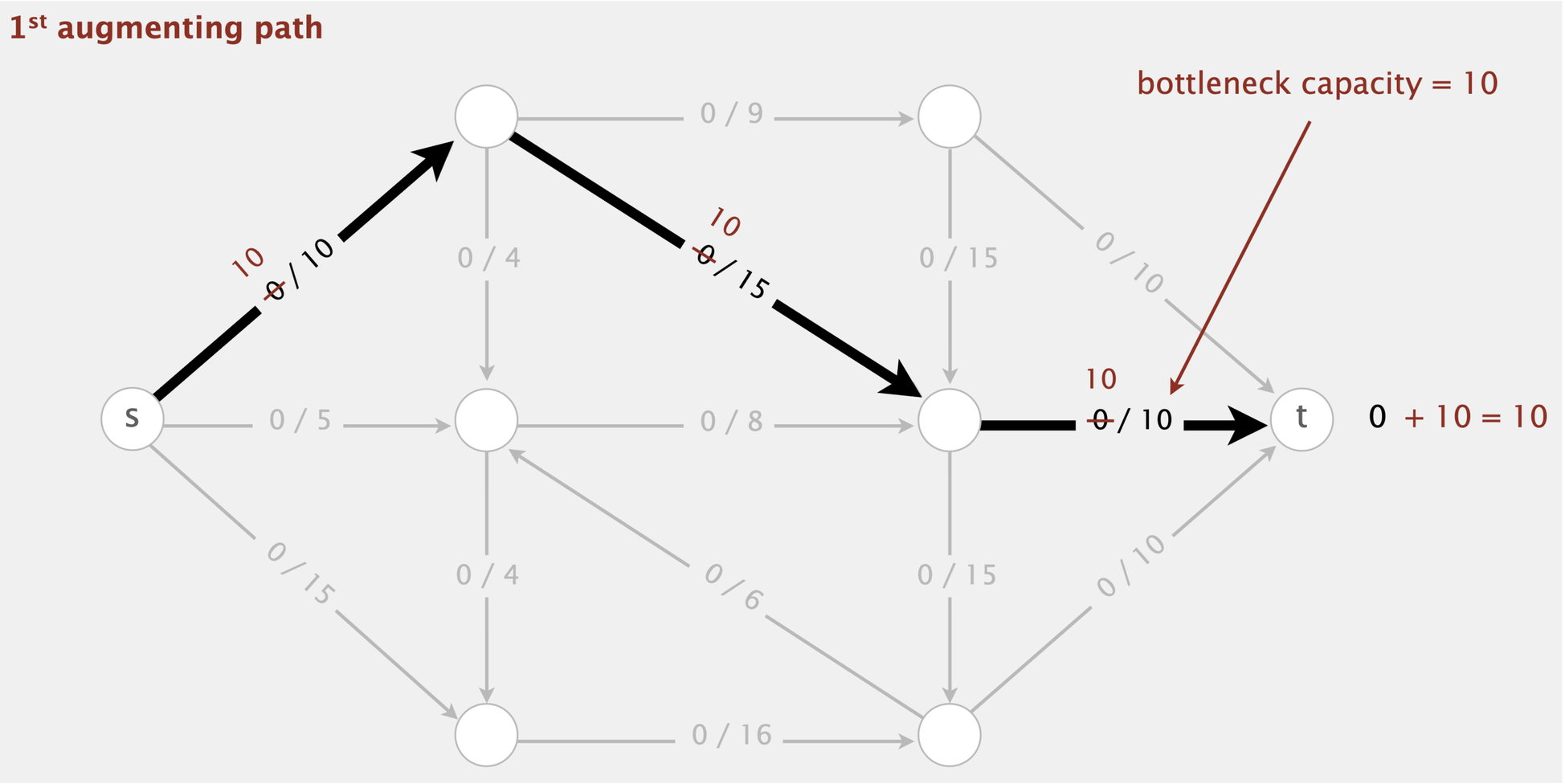
Initialization. Start with 0 flow.



# Ford-Fulkerson (augmenting path) Algorithm

Augmenting path. Find an undirected path from $s$ to $t$ such that:

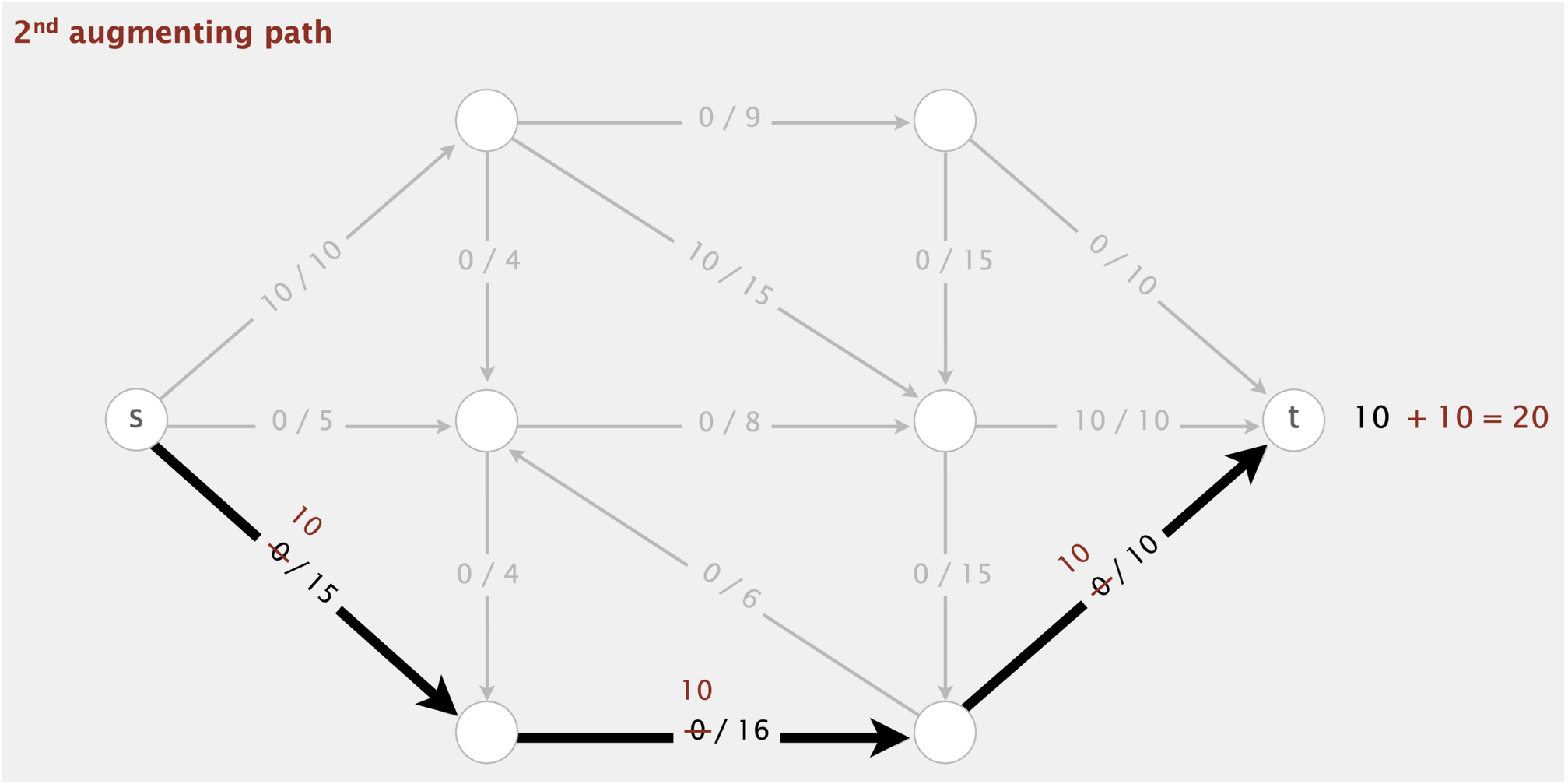
- Can increase flow on forward edges (not full).
- Can decrease flow on backward edge (not empty).



# Ford-Fulkerson (augmenting path) Algorithm

Augmenting path. Find an undirected path from $s$ to $t$ such that:

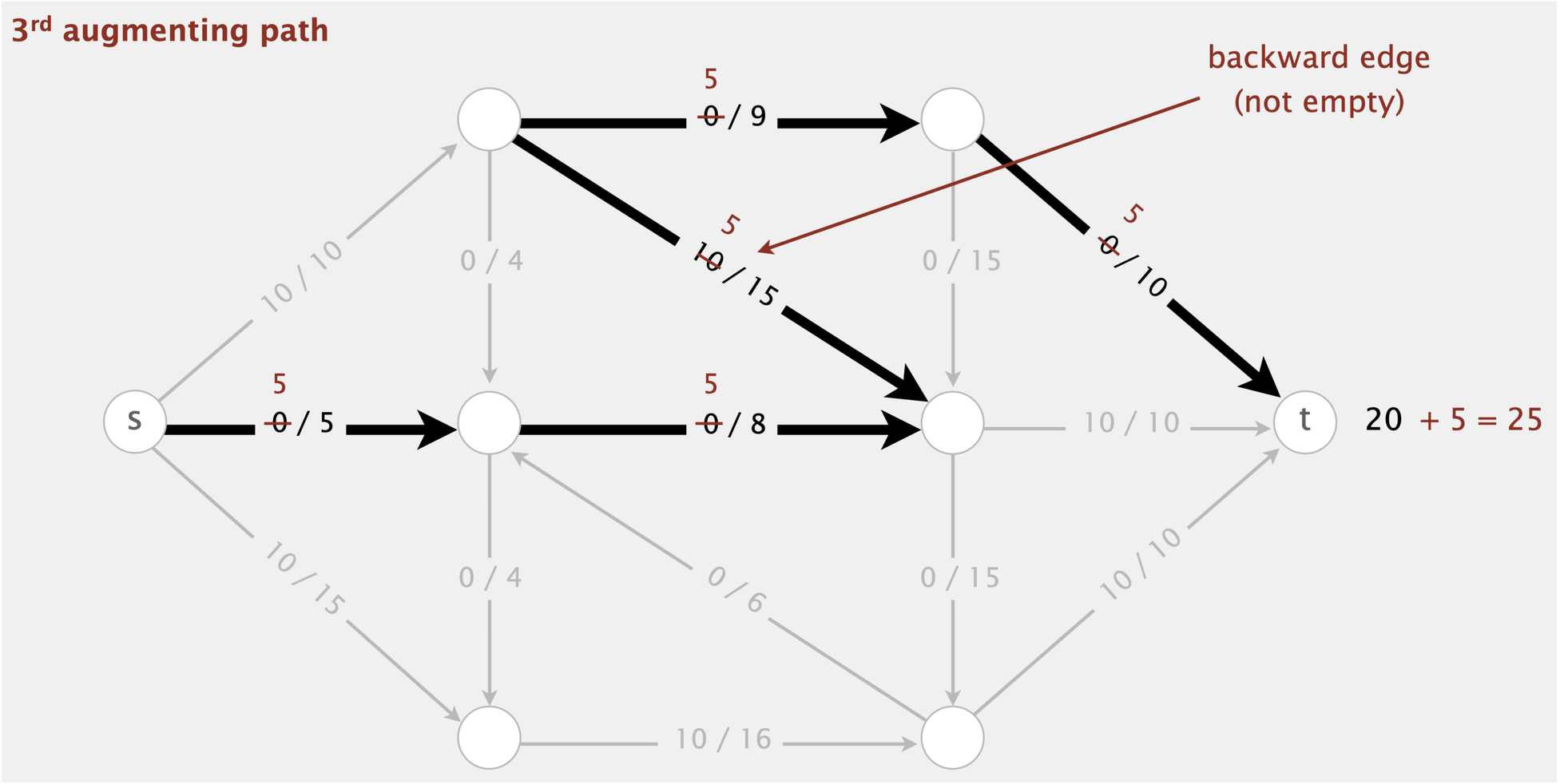
- Can increase flow on forward edges (not full).
- Can decrease flow on backward edge (not empty).



# Ford-Fulkerson (augmenting path) Algorithm

Augmenting path. Find an undirected path from $s$ to $t$ such that:

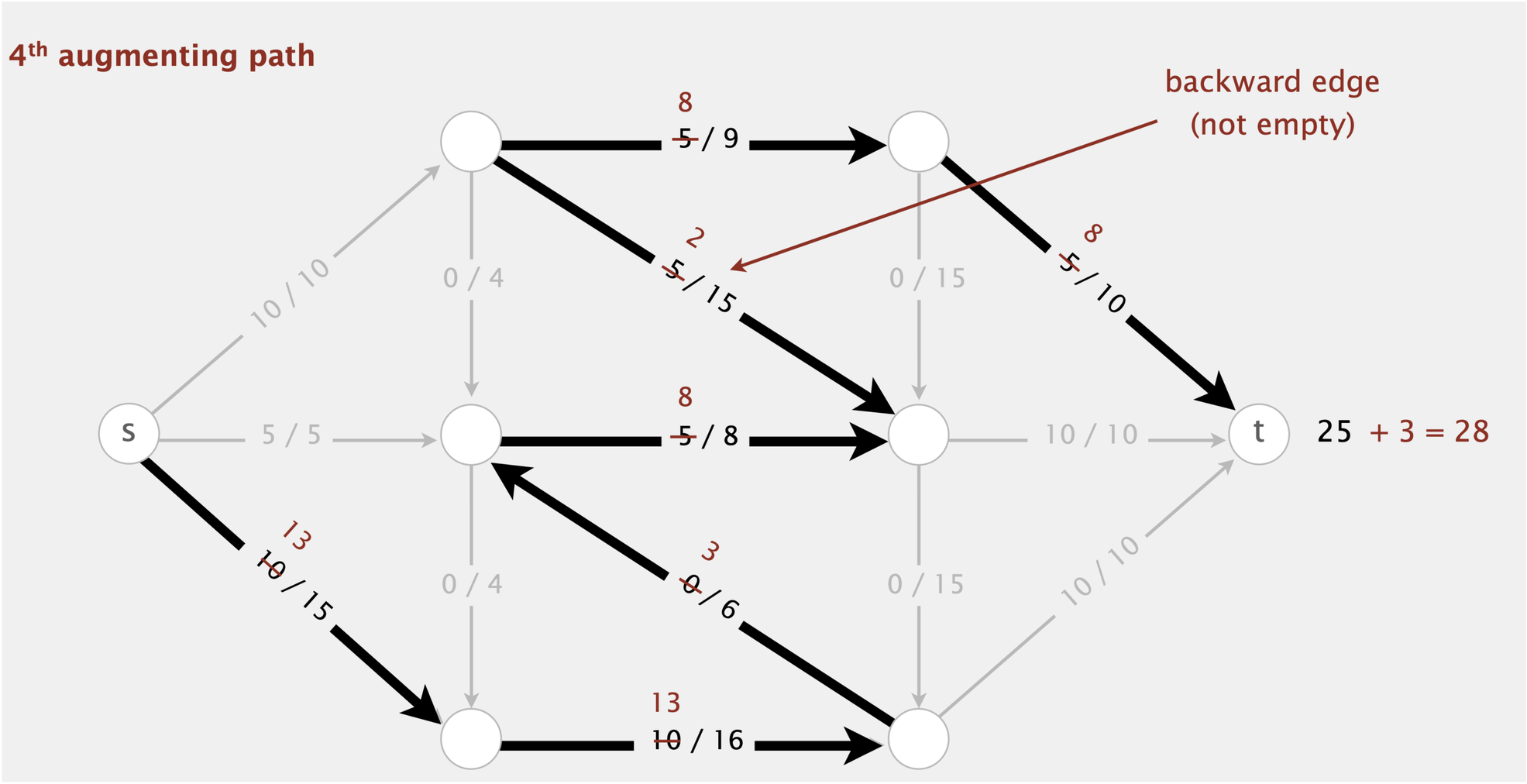
- Can increase flow on forward edges (not full).
- Can decrease flow on backward edge (not empty).



# Ford-Fulkerson (augmenting path) Algorithm

Augmenting path. Find an undirected path from $s$ to $t$ such that:

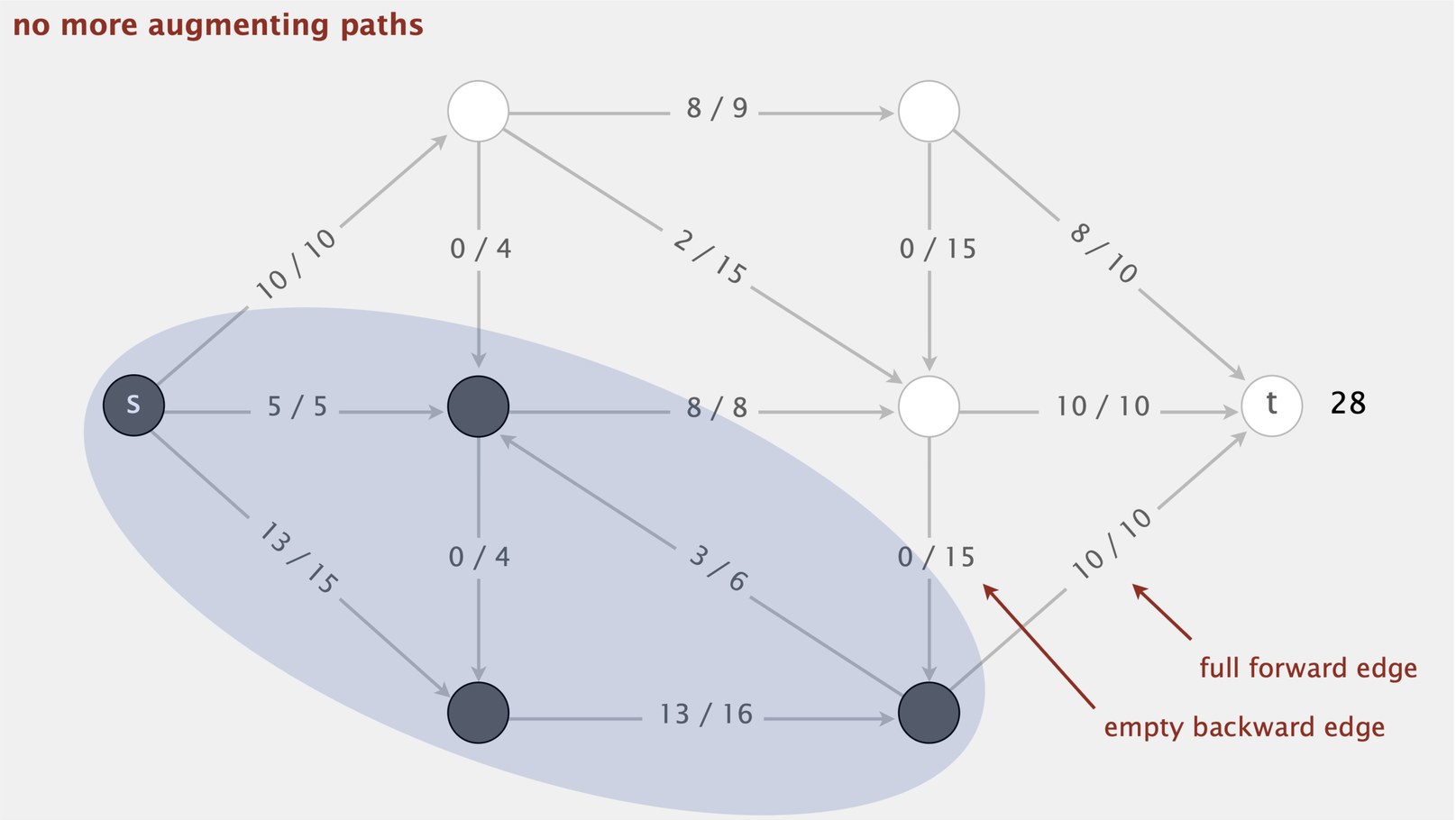
- Can increase flow on forward edges (not full).
- Can decrease flow on backward edge (not empty).



# Ford-Fulkerson (augmenting path) Algorithm

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

- Full forward edge.
- Empty backward edge.



# Ford-Fulkerson (augmenting path) 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**

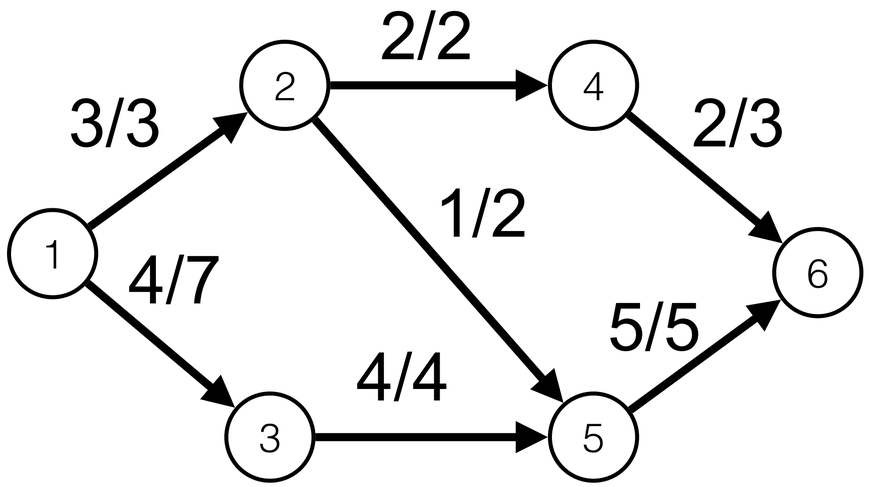
Questions.

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

# Residual Graph

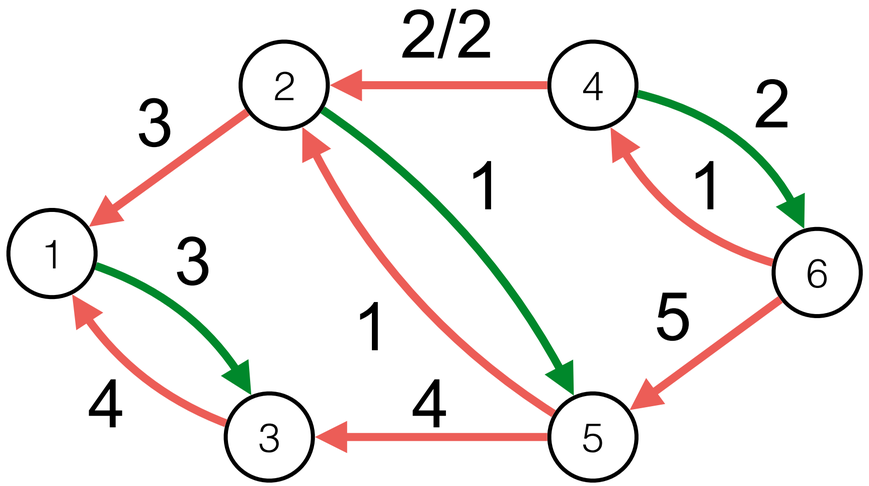
- The Residual Graph $G_f = (V, E_f)$ is used to discover s-t-paths on which it is possible to « push » an additional quantity of flow and so increase the value of the flow.
- This graph is composed of exactly the same nodes as the original graph $G = (V, E)$ but the edges may have a different direction and a different (residual) capacity.

# Residual Graph




Graph G with flow f

Forward Backward




Residual Graph $G_f$

# Residual Graph: Formally

- We have an edge (u,v) in the residual graph if and only if the edge (u, v) is not saturated or if some positive flow traverse the edge in the opposite direction :

$$(u, v) \in E_f \iff f_{u,v} < c_{u,v} \vee f_{v,u} > 0$$

- The residual capacity is given by

$$c^f(u, v) = \quad c_{u,v} - f_{u,v} \text{ if } f_{u,v} < c_{u,v}$$

forward edge

$$c^f(v, u) = \quad f_{u,v} \text{ if } f_{u,v} > 0$$

backward edge

# Augmenting Path = An *s*-*t* path in the residual graph

- A path from *s* to *t* in the residual graph $G_f$ is called **augmenting path**.
- Possible to « push » at least one more unit of flow along this path, i.e. increase by one unit the flow on each edge without increasing the capacity of the edges.
- **Theorem**: Consider an augmenting path and let *q* be the smallest residual capacity associated to an edge on this path. The following flow *f'* is a valid flow with value *v(f)* + *q*.

$$f'_{u,v} = \begin{cases} f_{u,v} + q & \text{if } (u,v) \text{ is forward edge in the augmenting path} \\ f_{u,v} - q & \text{if } (v,u) \text{ is a backward edge in the augmenting path} \\ f_{u,v} & \text{otherwise} \end{cases}$$

# Ford-Fulkerson Algoritm Summary

- Requires the capacities to be integers
- Is an iterative algorithm
- At each iteration, the algo takes a valid flow and transform it into another valid flow. This new flow has a strictly larger value, or the algo proves that there is no flow with a larger value.
- The flow increase is done along a path called « **augmenting path** ».
- The augmenting path are discovered in transformed graph called « **residual graph** ».

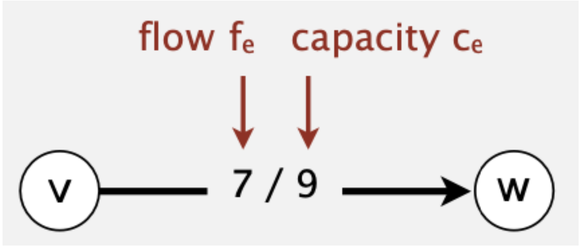
# Java Implementation: FlowEdge.java

Flow edge data type. Associate flow $fe$ and capacity $ce$ 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 $w$.

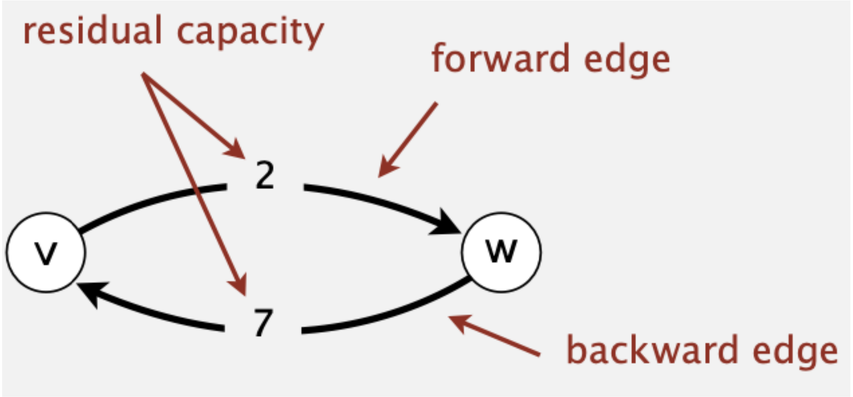
Residual (spare) capacity.

- Forward edge: residual capacity $= ce - fe$.
- Backward edge: residual capacity $= fe$.




Augment flow.

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



# Java Implementation: FlowEdge.java

| public class FlowEdge | |
|---|---|
| FlowEdge(int v, int w, double capacity) | *create a flow edge v→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* |

# Java Implementation: FlowEdge.java

```
public class FlowEdge {
    private final int v;              // from
    private final int w;              // 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;
        this.flow      = 0.0;
    }
    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("invalid endpoint");
    }
    public double residualCapacityTo(int vertex) {
        if      (vertex == v) return flow;              // backward edge
        else if (vertex == w) return capacity - flow;   // forward edge
        else throw new IllegalArgumentException("invalid endpoint");
    }
    public void addResidualFlowTo(int vertex, double delta) {
        if (!(delta >= 0.0)) throw new IllegalArgumentException("Delta must be non-negative");

        if      (vertex == v) flow -= delta;           // backward edge
        else if (vertex == w) flow += delta;           // forward edge
        else throw new IllegalArgumentException("invalid endpoint");
    }
}
```

# Java Implementation: FlowNetwork.java

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

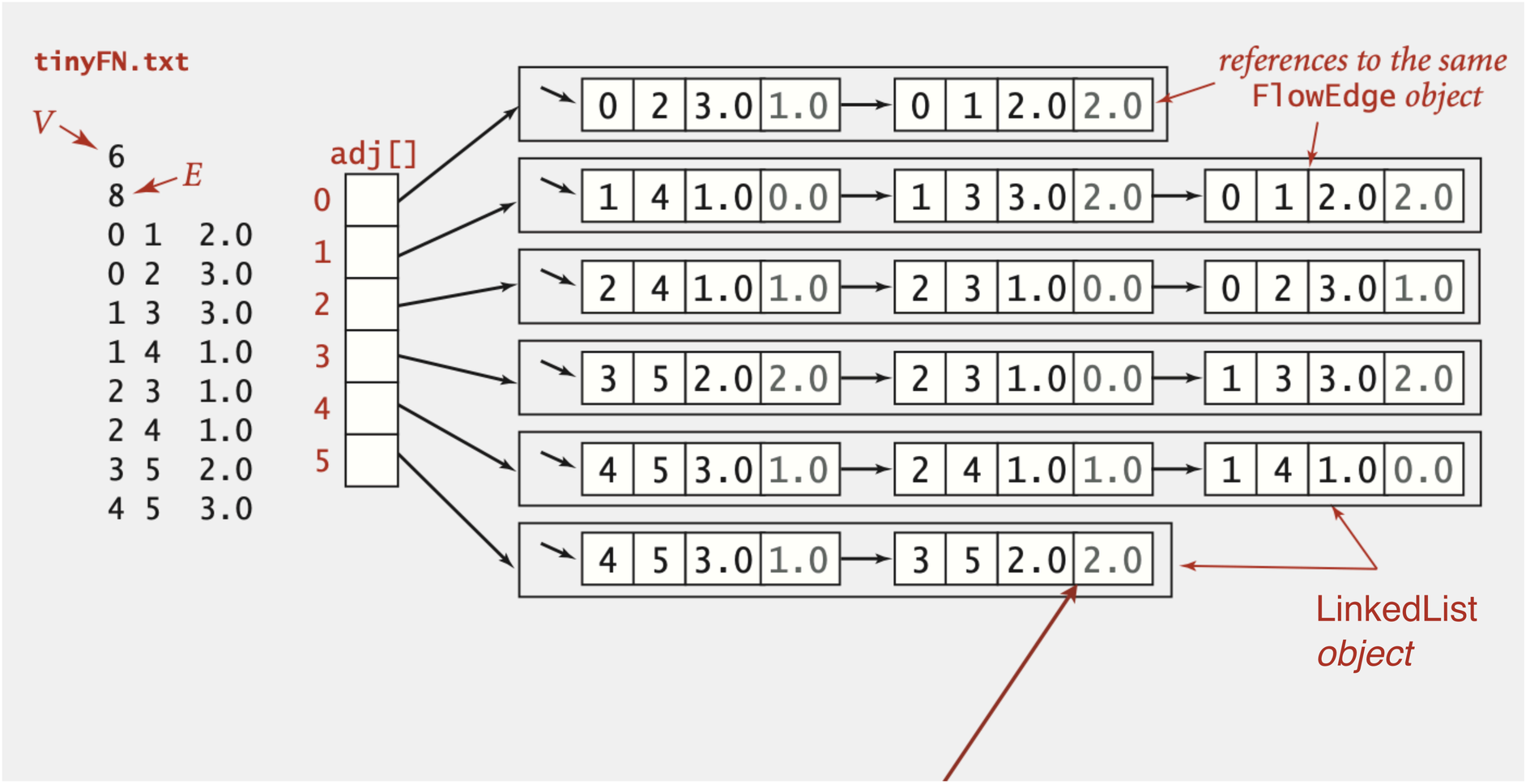
# Java Implementation: FlowNetwork.java

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

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

Adjacency List implementation of a Graph

Add forward and backward edges

# Java Implementation: FlowNetwork.java (Adjacency List)




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

# Searching for an augmenting path with DFS

```
public class FordFulkerson {

      private final int V;           // number of vertices
      private boolean[] marked;      // marked[v] = true iff s->v path in residual graph
      private FlowEdge[] edgeTo;     // edgeTo[v] = last edge on shortest residual s->v path
      private double value;          // current value of max flow
      private FlowNetwork G;

      private boolean hasAugmentingPath(FlowNetwork G, int s, int t) {
          edgeTo = new FlowEdge[G.V()];
          marked = new boolean[G.V()];
          // depth-first search
          Stack<Integer> stack = new Stack<>();
          stack.add(s);
          marked[s] = true;
          while (!stack.isEmpty() && !marked[t]) {
              int v = stack.pop();

              for (FlowEdge e : G.adj(v)) {
                  int w = e.other(v);

                  // if residual capacity from v to w
                  if (e.residualCapacityTo(w) > 0) {
                      if (!marked[w]) {
                          edgeTo[w] = e;
                          marked[w] = true;
                          stack.push(w);
                      }
                  }
              }
          }
          // is there an augmenting path?
          return marked[t];
      }
}
```

Save last edge on the path to w

# Java Implem: FordFulkerson

```
public class FordFulkerson {

     private final int V;           // number of vertices
     private boolean[] marked;      // marked[v] = true iff s->v path in residual graph
     private FlowEdge[] edgeTo;     // edgeTo[v] = last edge on shortest residual s->v path
     private double value;          // current value of max flow
     private FlowNetwork G;

    public FordFulkerson(FlowNetwork G, int s, int t) {
        this.G = G;
        V = G.V();

        // while there exists an augmenting path, use it
        value = excess(G, t);
        while (hasAugmentingPath(G, s, t)) {

            // compute bottleneck capacity
            double bottle = Double.POSITIVE_INFINITY;
            for (int v = t; v != s; v = edgeTo[v].other(v)) {
                bottle = Math.min(bottle, edgeTo[v].residualCapacityTo(v));
            }

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

            value += bottle;
        }
    }
}
```

# Ford-Fulkerson with Integer Capacities

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

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 ≤ the value of the maxflow.

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

Integrality theorem. There exists an integer-valued maxflow.

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

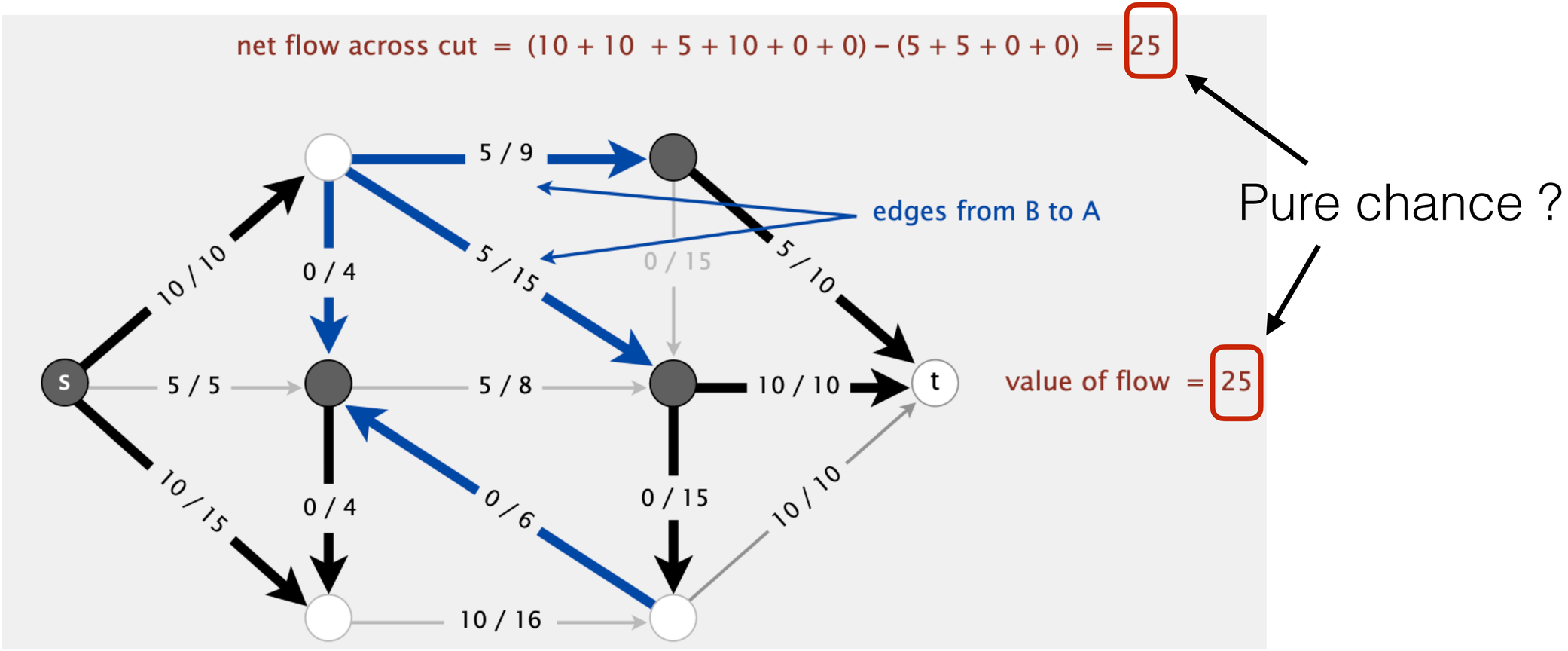
# Flow and Cut Relation: Flow of a cut

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 $A$.

$$\text{inflow}(A,B) = \sum_{(u,v)\in E: u\in A, v\in B} f_{u,v}$$

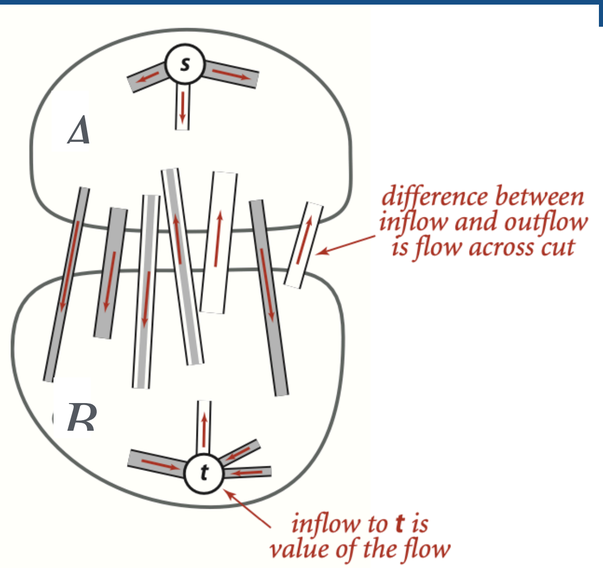
$$\text{outflow}(A,B) = \sum_{(u,v)\in E: u\in B, v\in A} f_{u,v}$$

$$\text{netflow}(A,B) = \text{inflow}(A,B) - \text{outflow}(A,B)$$



# Flow and Cut Relation

**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.

Proof. 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.

# Max-Flow, Min-Cut Theorem

## Theorem Max-Flow, Min-Cut

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 $f$

Proof (in three parts):

[ i ⇒ ii ] [ ii ⇒ iii ] [ iii ⇒ i ]

# Max-Flow, Min-Cut Theorem

## Theorem Max-Flow, Min-Cut

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 $f$

Proof: [ i $\Rightarrow$ ii ] (Weak Duality of Linear Programming)

- 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.

# Max-Flow, Min-Cut Theorem

## Theorem Max-Flow, Min-Cut

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 $f$

Proof: [ ii $\Rightarrow$ iii ]

- If it was true, then we could improve the flow $f$ (contradiction)

# Max-Flow, Min-Cut Theorem

## Theorem Max-Flow, Min-Cut

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 $f$

Proof: [ iii $\Rightarrow$ i ]

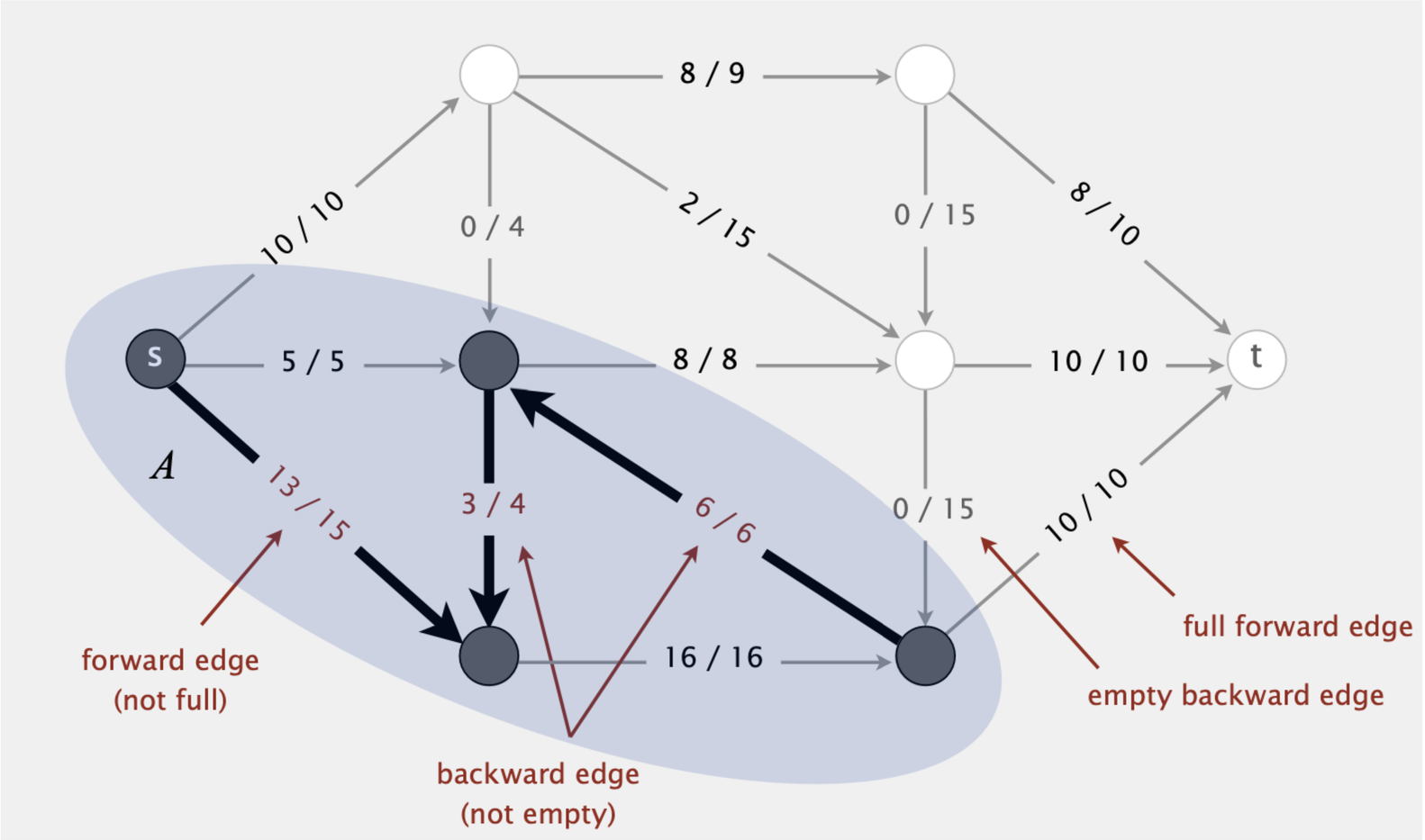
Suppose that there is no augmenting path with respect to $f$.

- 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 (forward edges are full, backward edges are empty)
- = value of flow $f$.

# Computing a mincut from a maxflow

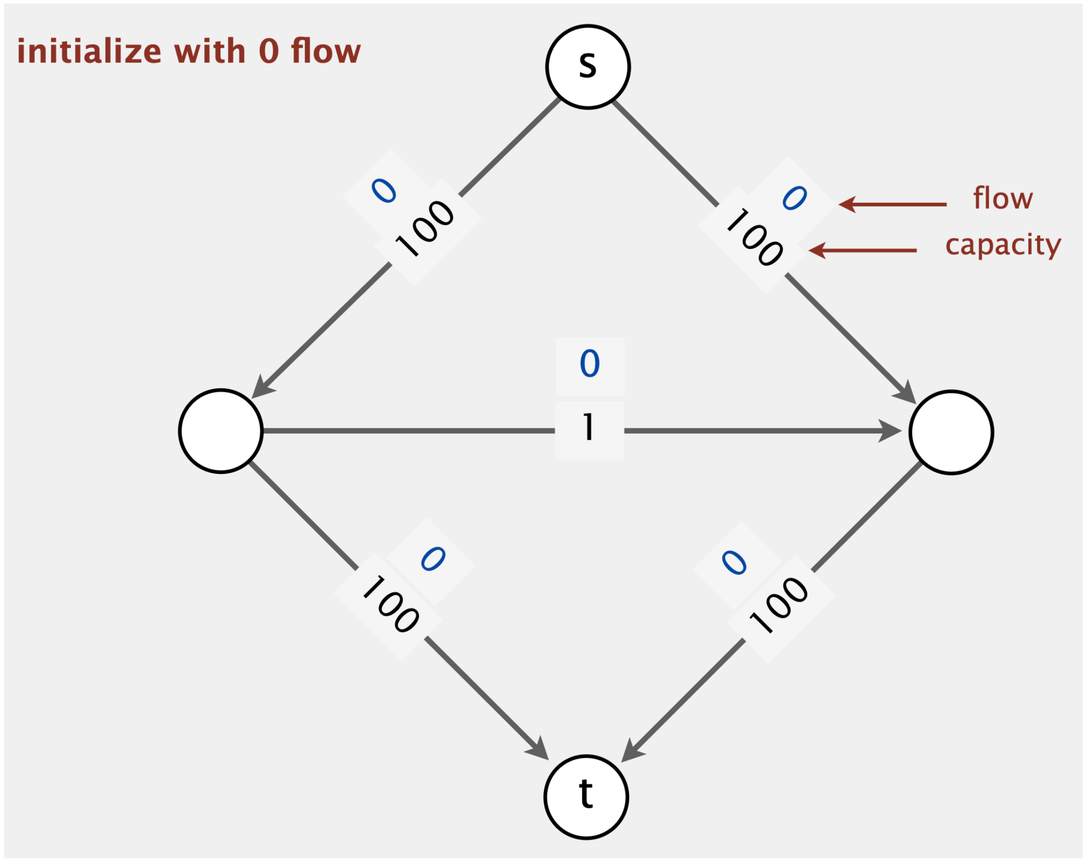
Maxflow, thus no augmenting paths with respect to $f$.

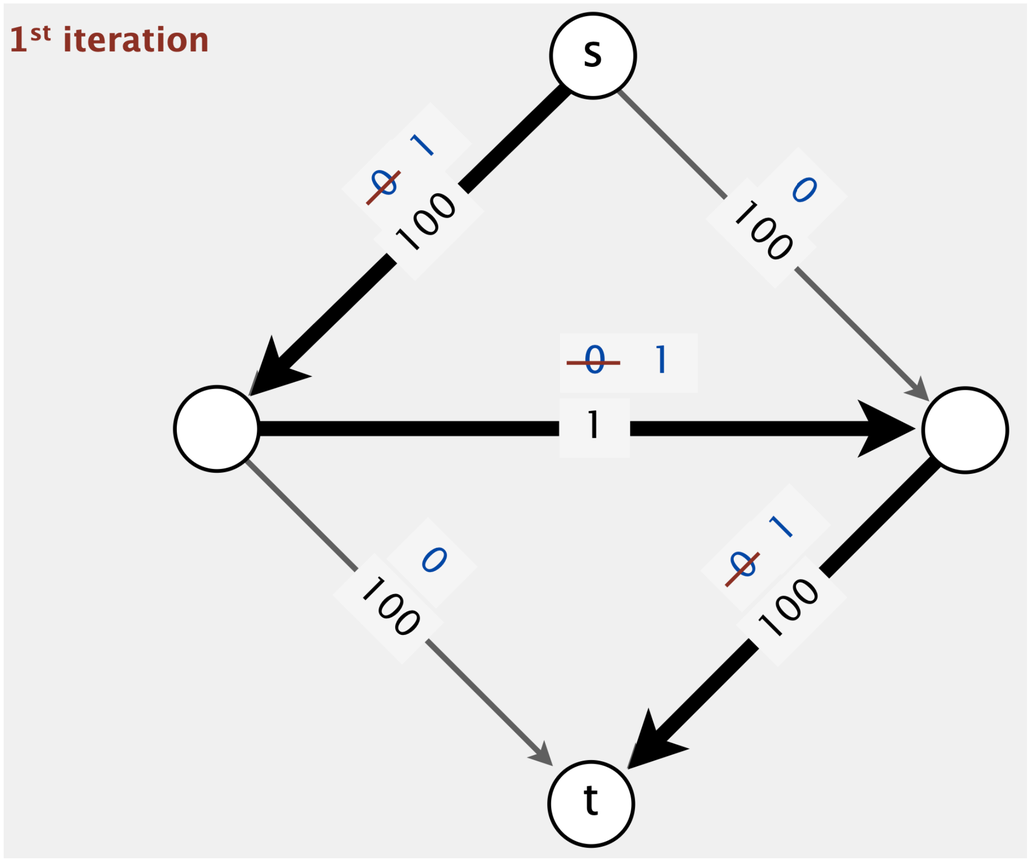
- Compute $A$ = set of vertices connected to $s$ in $G_f$ (residual graph)



# Ford-Fulkerson Time Complexity: A very bad case

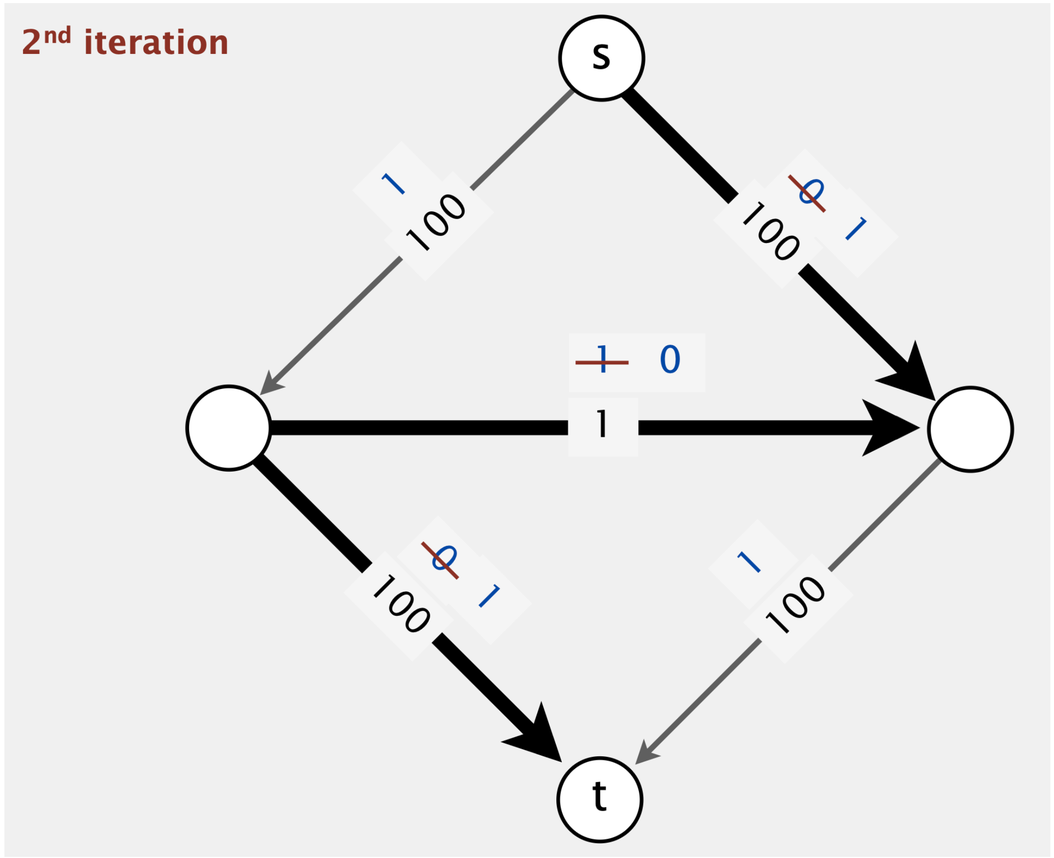
Bad news. Even when edge capacities are integers, number of augmenting paths could be equal to the value of the maxflow.

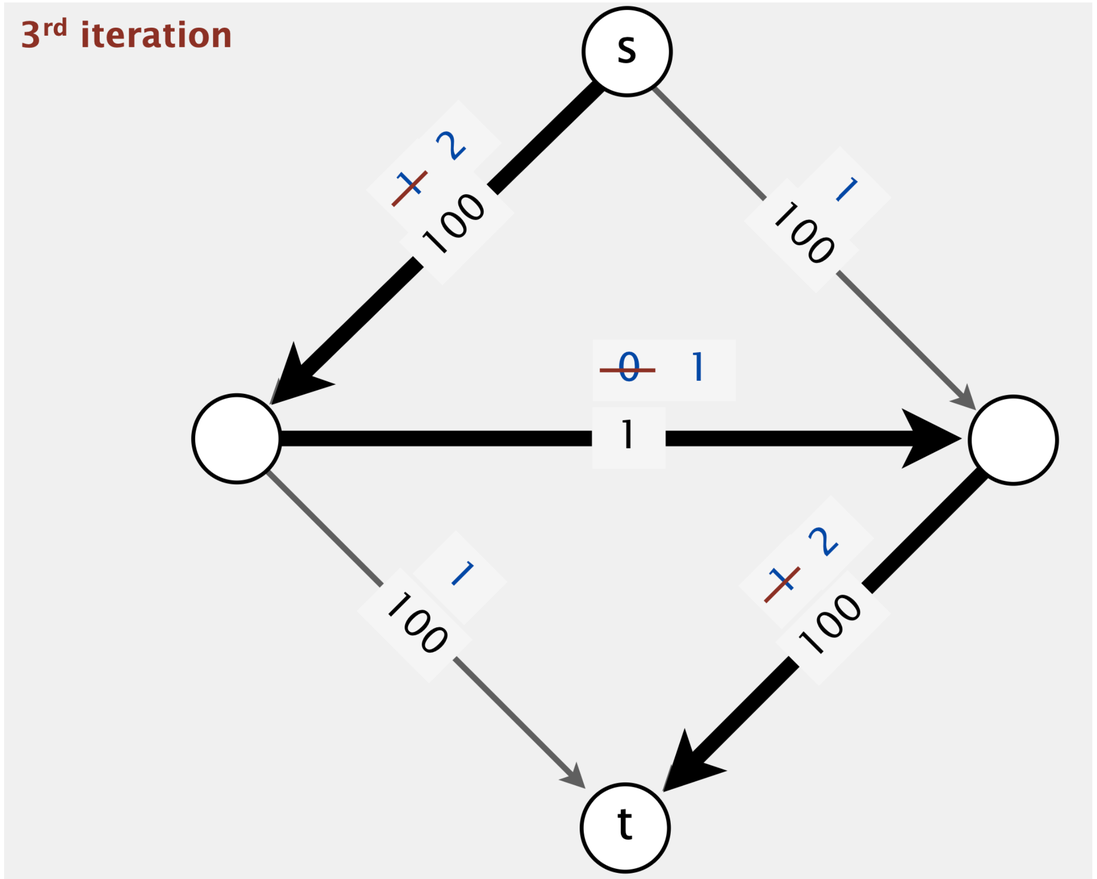


# Ford-Fulkerson Time Complexity: A very bad case

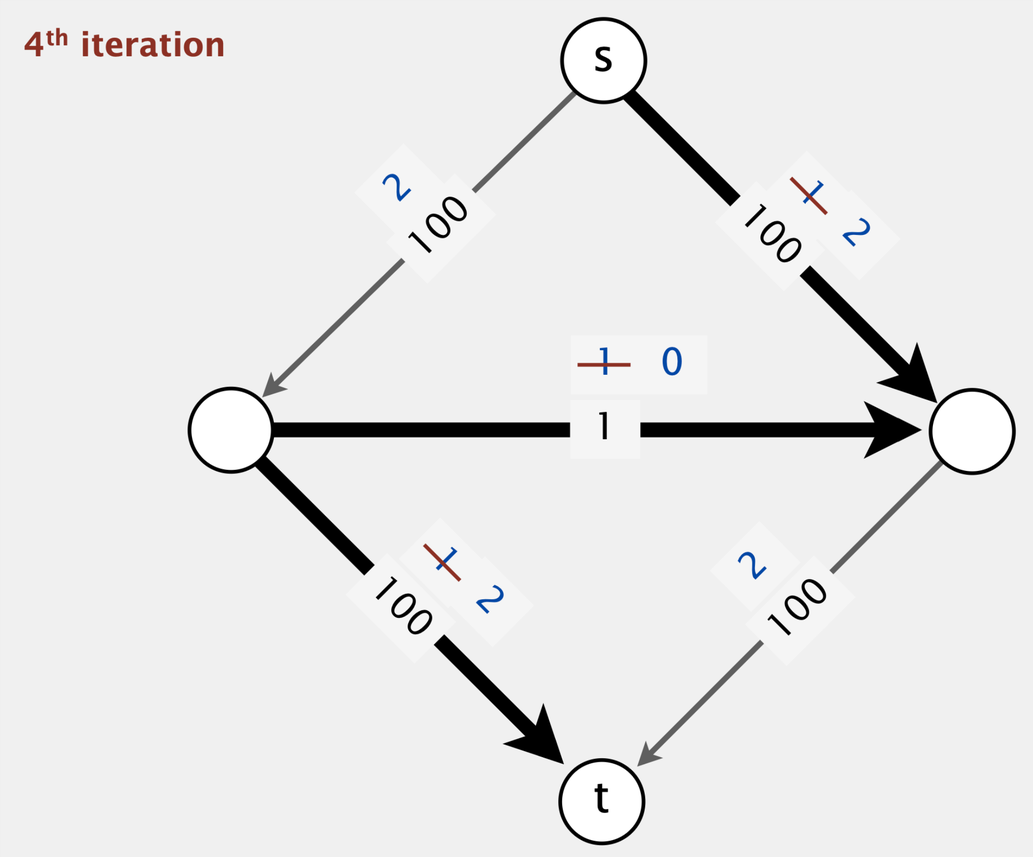
Bad news. Even when edge capacities are integers, number of augmenting paths could be equal to the value of the maxflow.



# Ford-Fulkerson Time Complexity: A very bad case

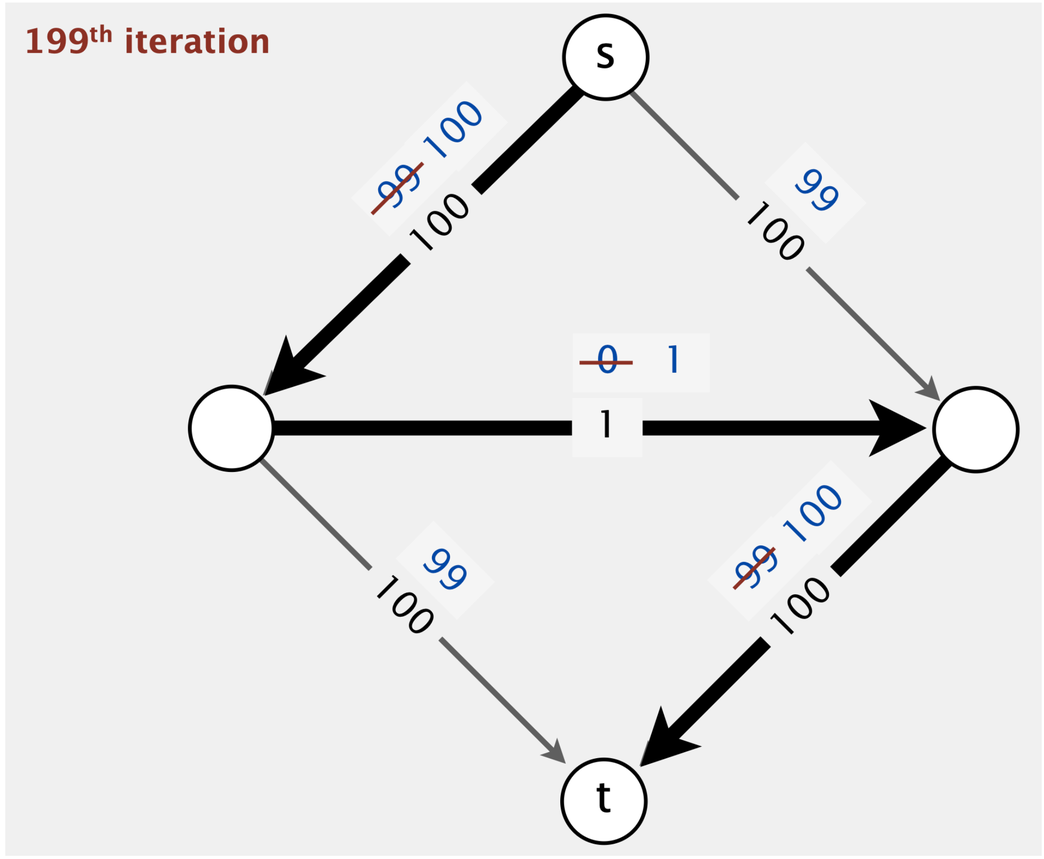
Bad news. Even when edge capacities are integers, number of augmenting paths could be equal to the value of the maxflow.

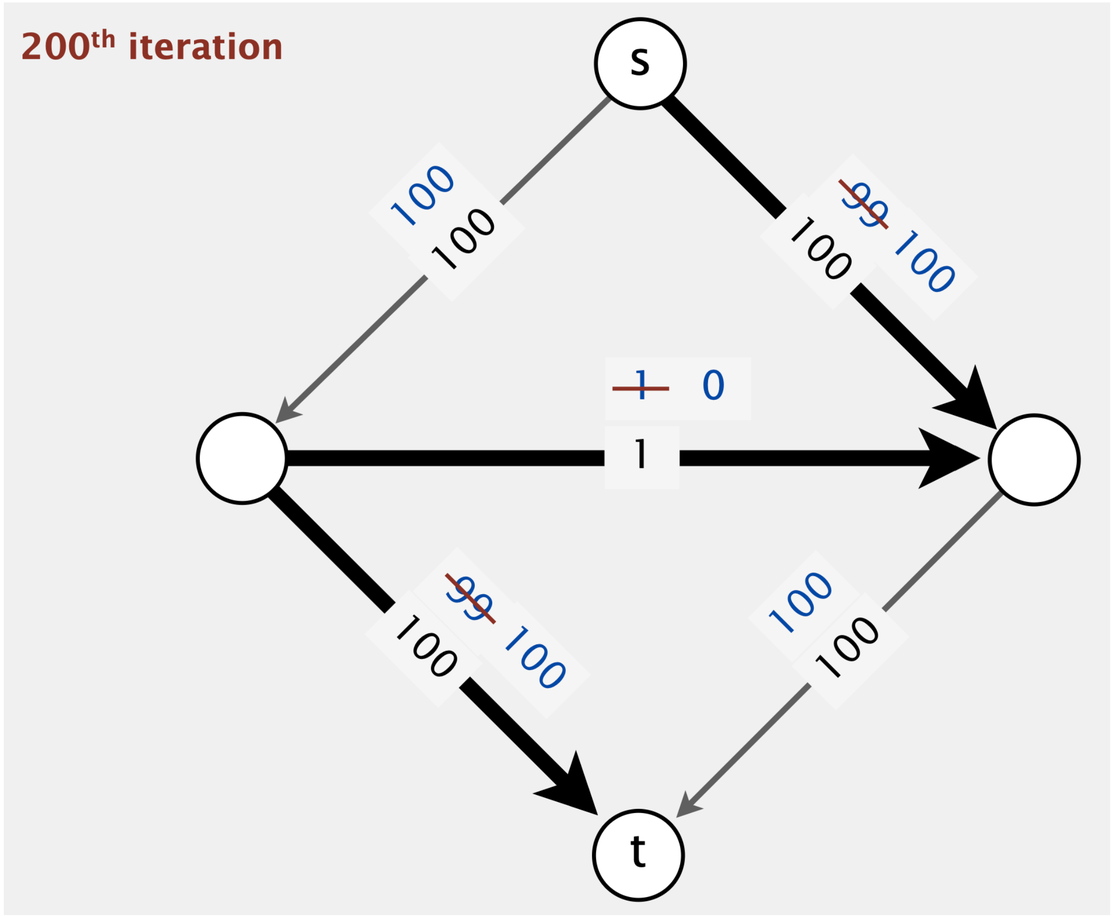



...

# Ford-Fulkerson Time Complexity: A very bad case

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



# Ford-Fulkerson Time Complexity

- Time Complexity is $\mathcal{O}(f_s \cdot |E|)$
  - It is strange to have the time complexity expressed in terms of the final result.
  - Let's try to reformulate the complexity in terms of the input.
- Let $U$ be the maximum capacity of the graph.
- $c(S = s, T = V - s)$ is in $\mathcal{O}(|V| \cdot U)$ (by the max-flow min cut theorem)
- Ford-Fulkerson complexity can thus be expressed as $\mathcal{O}(|V| \cdot |E| \cdot U)$

Is this polynomial in the size of the input?

No because only $\log U$ bits necessary to represent value of $U$

# Making Ford-Fulkerson (Strongly) Polynomial

- Try to discover first augmenting path with large bottleneck capacities 💡
- How ? By filtering out the edges in $G_f$ (residual graph) with a too small capacity

**Max-Flow by Scaling (G,s,t)**

$U = \max_{e \in E} c_e$

Initialize $f = 0$

$\Delta = 2^{\lfloor \log U \rfloor}$

**While** $\Delta \geq 1$

$\log U$ iterations

  **While** there exists an augmenting path $p$ of bottleneck capacity at least $\Delta$

    Increase flow on $p$ by bottleneck capacity

  $\Delta = \Delta/2$

**return** $f$

How many augmenting paths can we discover at most in a $\Delta$-scaling phase?

# Number of steps in a Δ-scaling phase

- Consider the flow f at the end of the Δ-scaling phase and let $v(f)$ denote its flow value.
- Let $S$ be the set of nodes reachable from $s$ in $G_{f,\Delta}$
- Then $(S, V \backslash S)$ is an $s - t$ cut. By definition of $S$, the residual capacity of the cut is $c(S, V \backslash S) \leq |E| \cdot \Delta$
- If $f^*$ is the optimal flow, then $v(f^*) - v(f) \leq |E| \cdot \Delta$
- In the next Δ-scaling phase, each augmentation carries at least $\frac{\Delta}{2}$ units.
- So the next Δ-scaling phase can perform at most $\frac{|E| \cdot \Delta}{\frac{\Delta}{2}} = 2 \cdot |E|$ augmentations

# Ford-Fulkerson with scaling: Complexity

- $\mathcal{O}(\log U)$ Δ-scaling phases
- At most $\mathcal{O}(|E|)$ augmenting paths in each Δ-scaling phase
- Discovery of one augmenting path in $\mathcal{O}(|E|)$
- Total complexity is thus

$$O(|E|^2 \log U)$$

Is this polynomial in the size of the input?

Yes it is!

# Another way to fix the time-complexity of Ford-Fulkerson

```
public class FordFulkerson {

      private final int V;           // number of vertices
      private boolean[] marked;      // marked[v] = true iff s->v path in residual graph
      private FlowEdge[] edgeTo;     // edgeTo[v] = last edge on shortest residual s->v path
      private double value;          // current value of max flow
      private FlowNetwork G;

      private boolean hasAugmentingPath(FlowNetwork G, int s, int t) {
          edgeTo = new FlowEdge[G.V()];
          marked = new boolean[G.V()];
          // breadth-first search
          Queue<Integer> queue = new LinkedList<>();
          queue.add(s);
          marked[s] = true;
          while (!queue.isEmpty() && !marked[t]) {
              int v = stack.remove();

              for (FlowEdge e : G.adj(v)) {
                  int w = e.other(v);

                  // if residual capacity from v to w
                  if (e.residualCapacityTo(w) > 0) {
                      if (!marked[w]) {
                          edgeTo[w] = e;
                          marked[w] = true;
                          queue.add(w);
                      }
                  }
              }
          }
          // is there an augmenting path?
          return marked[t];
      }
}
```

Use a Queue instead of Stack so that the path is the shortest one (number of edges). BFS Algo

**Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems**

JACK EDMONDS

*University of Waterloo, Waterloo, Ontario, Canada*

AND

RICHARD M. KARP

*University of California, Berkeley, California*

ABSTRACT. This paper presents new algorithms for the maximum flow problem, the Hitchcock transportation problem, and the general minimum-cost flow problem. Upper bounds on the numbers of steps in these algorithms are derived, and are shown to compare favorably with upper bounds on the numbers of steps required by earlier algorithms.

**Edmonds-Karp 1972 (USA)**

# Ford-Fulkerson with shortest paths (Edmond-Karp)

- The number of augmenting paths needed is at most $\frac{|E| \cdot |V|}{2}$
  - Proof main argument: every time you augment, the bottleneck edge will see its shortest distance through it augmenting by at least two edges.
- Time complexity is $\mathcal{O}(|V| \cdot |E|^2)$
- Another idea: looking the “fattest path” instead. Augmenting path with the largest bottleneck capacity (Dijkstra modification, use a priority queue). This idea is very close to the $\Delta$-scaling method, it actually get the same time-complexity

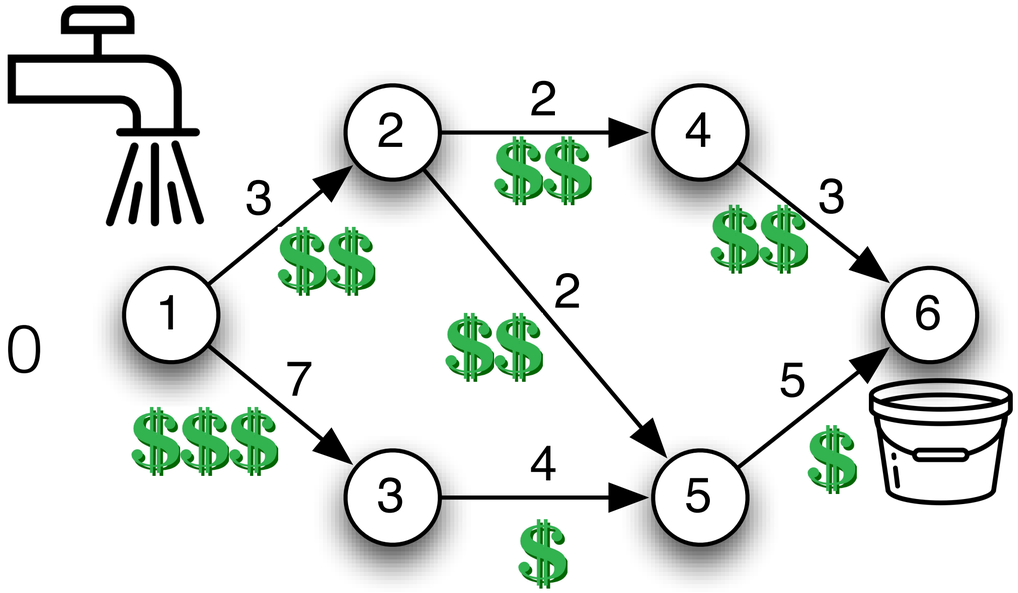
# Ford-Fulkerson and its variants: Complexity Summary

- $U$ = max capa in the graph

| Variant | Complexity |
|---|---|
| DFS Path | $\mathcal{O}(U \cdot \lvert E \rvert^2)$ |
| BFS (shortest) Path | $\mathcal{O}(\lvert V \rvert \cdot \lvert E \rvert^2)$ |
| Dijkstra maximum capa path | $\mathcal{O}(\lvert E \rvert^2 \log U)$ |
| Delta scaling (with DFS) | $\mathcal{O}(\lvert E \rvert^2 \log U)$ |

# Min Cost Flow (quick intro)

Input: B = 10
You must flow 10 litres/second




- Decide how to flow this quantity B to from the source to the sink at minimal cost without exceeding capacities

# Successive Shortest Path Algorithm

- Start with a flow of zero
- Find an s-t path in $G_f$ with minimum weight (**shortest path** problem)
- Augment along this path as much as possible (limited by smallest residual capacity of the path).
- Repeat two previous steps until initial demand B of the source is met.

- Moore-Bellman-Ford must be used (negative weights possible) O(|V||E|)
- Total complexity is thus O(B|V||E|)
- Remark: A similar tric for as for the capacity scaling is possible to make it strongly polynomial.

# Bibliography


- « Algorithms»  Sedgewick, R., & Wayne, K. (2011).. Addison-wesley
- « Introduction to algorithms » Thomas H. Cormen, Charles E. Leiserson, Ronald Rivest et Clifford Stein. Third edition. Chapter26
- « Network Flows » Ahuja, Maganti and Orlin Chapter 7 & 9.
- « Combinatorial Optimization Theory and Algorithms », Korte, Bernhard, Vygen, Jens. 5th Edition. Chapter 9.


Remark: We have seen the Ford-Fulkerson method for solving maximum flow problems. It is the most famous method but many other algorithms exists (push relabel, blocking flow, etc). Two know more about state of the art, have a look at http://cacm.acm.org/magazines/2014/8/177011-efficient-maximum-flow-algorithms/fulltext

# History

**D. R. Fulkerson**

1924 – 1976

**L. R. Ford**

1927

**Richard Manning Karp**

1935

**Jack Edmonds**

1934

Turing Award

https://www.youtube.com/watch?v=D36MJCXT4Qk

Nice video on network flows