http://algs4.cs.princeton.edu

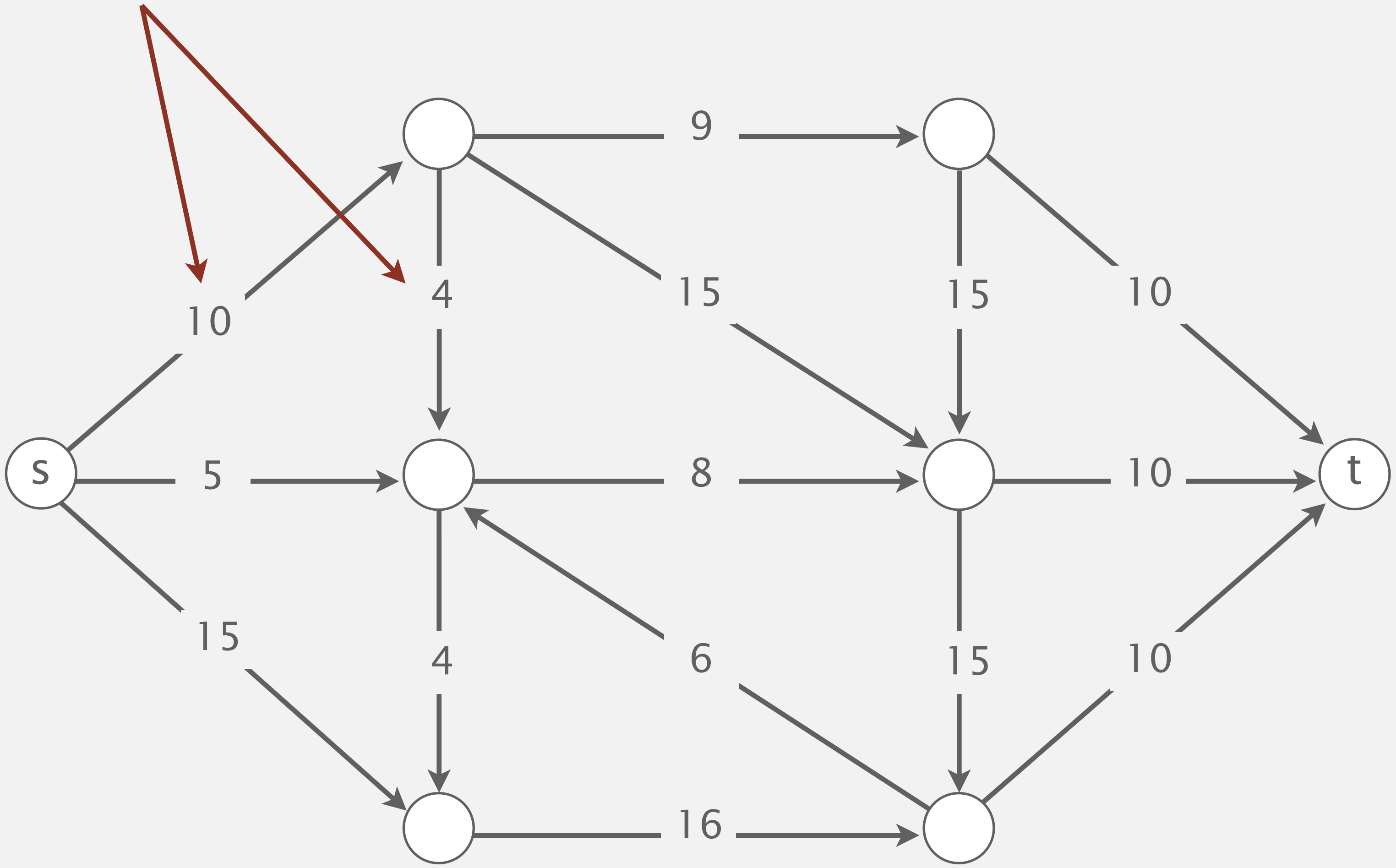
6.4 Maximum Flow

- *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.

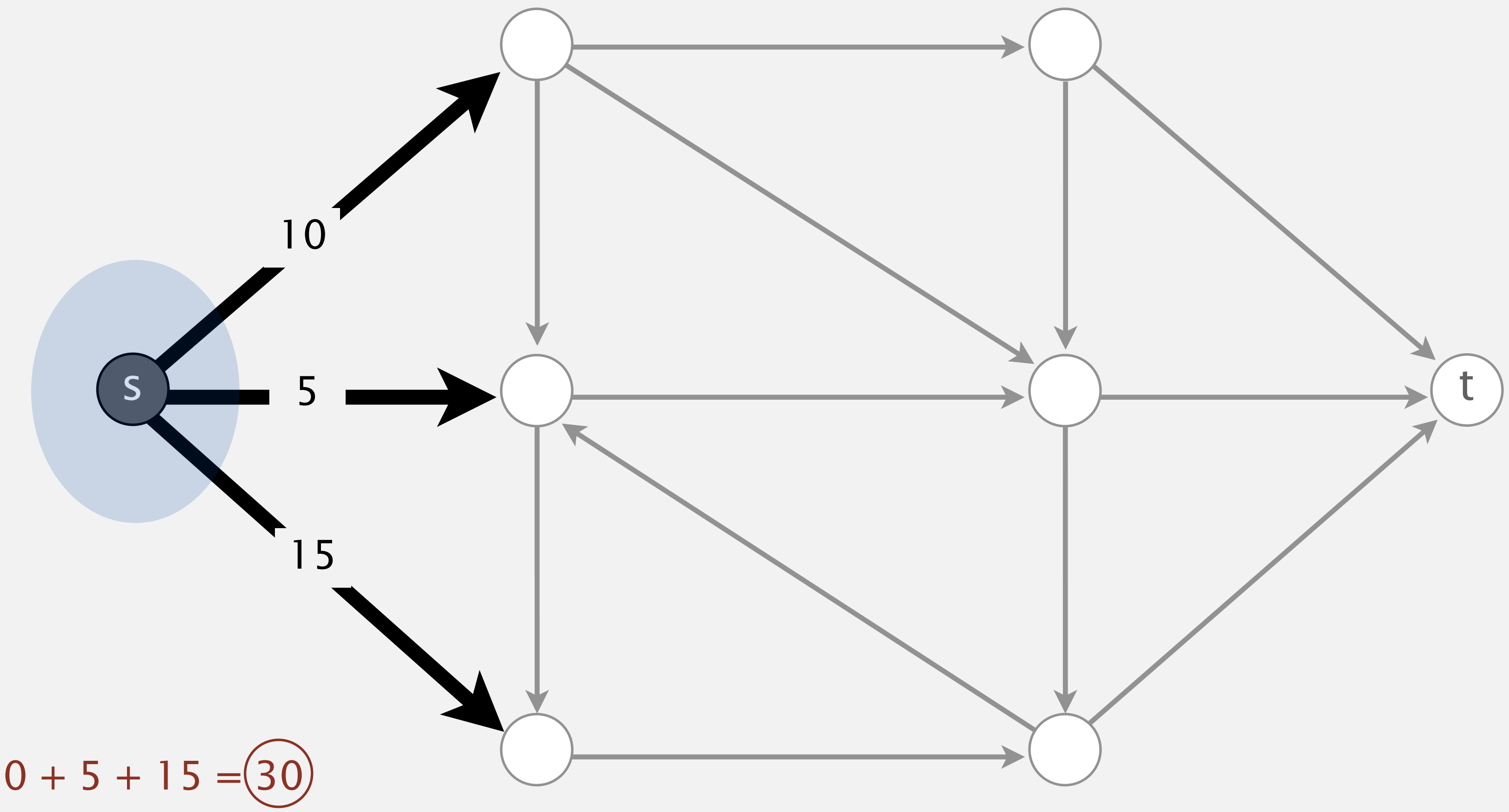
each edge has a positive capacity


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.

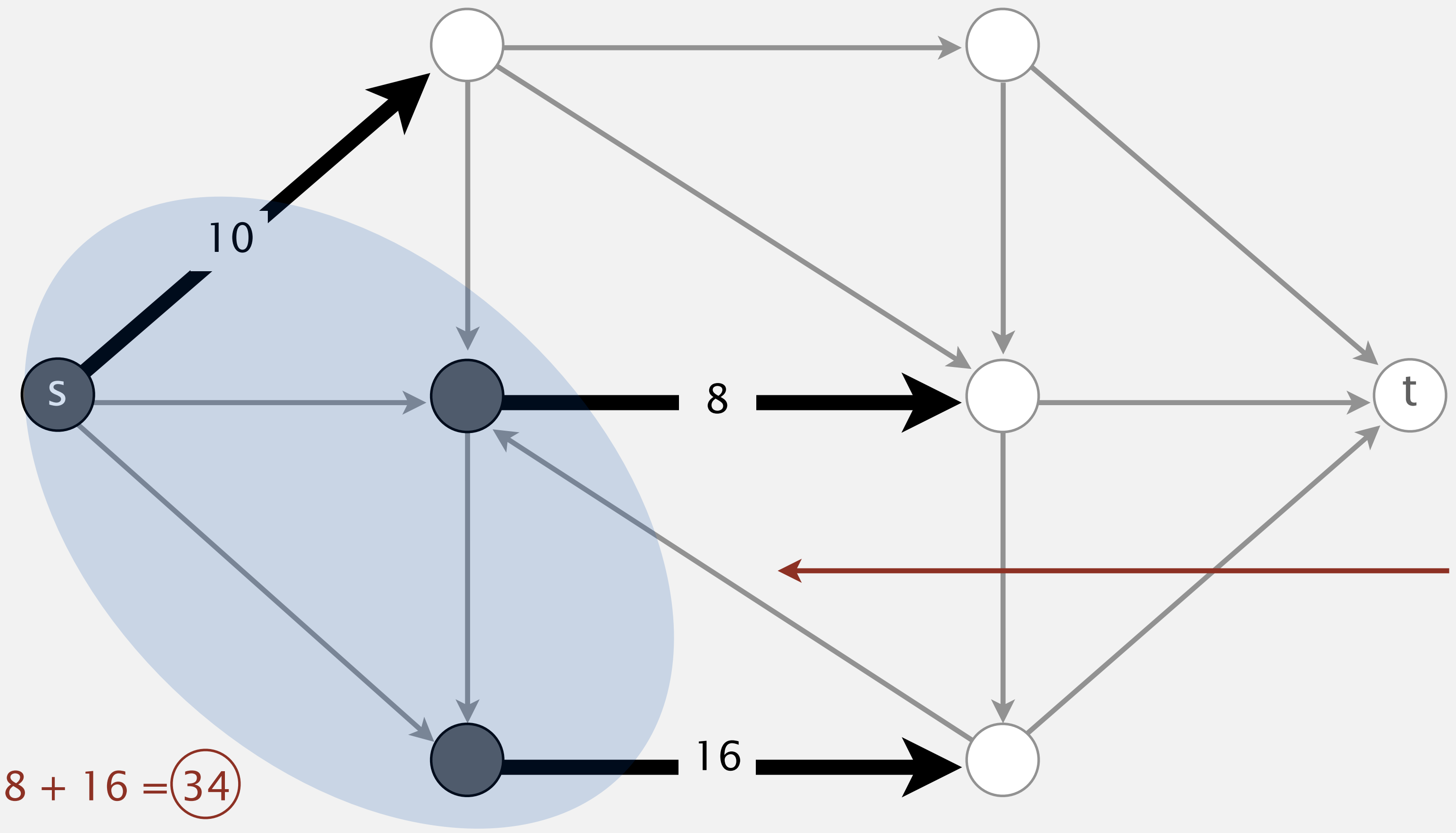


capacity = 10 + 5 + 15 = 30

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*.



don't count edges from B to A

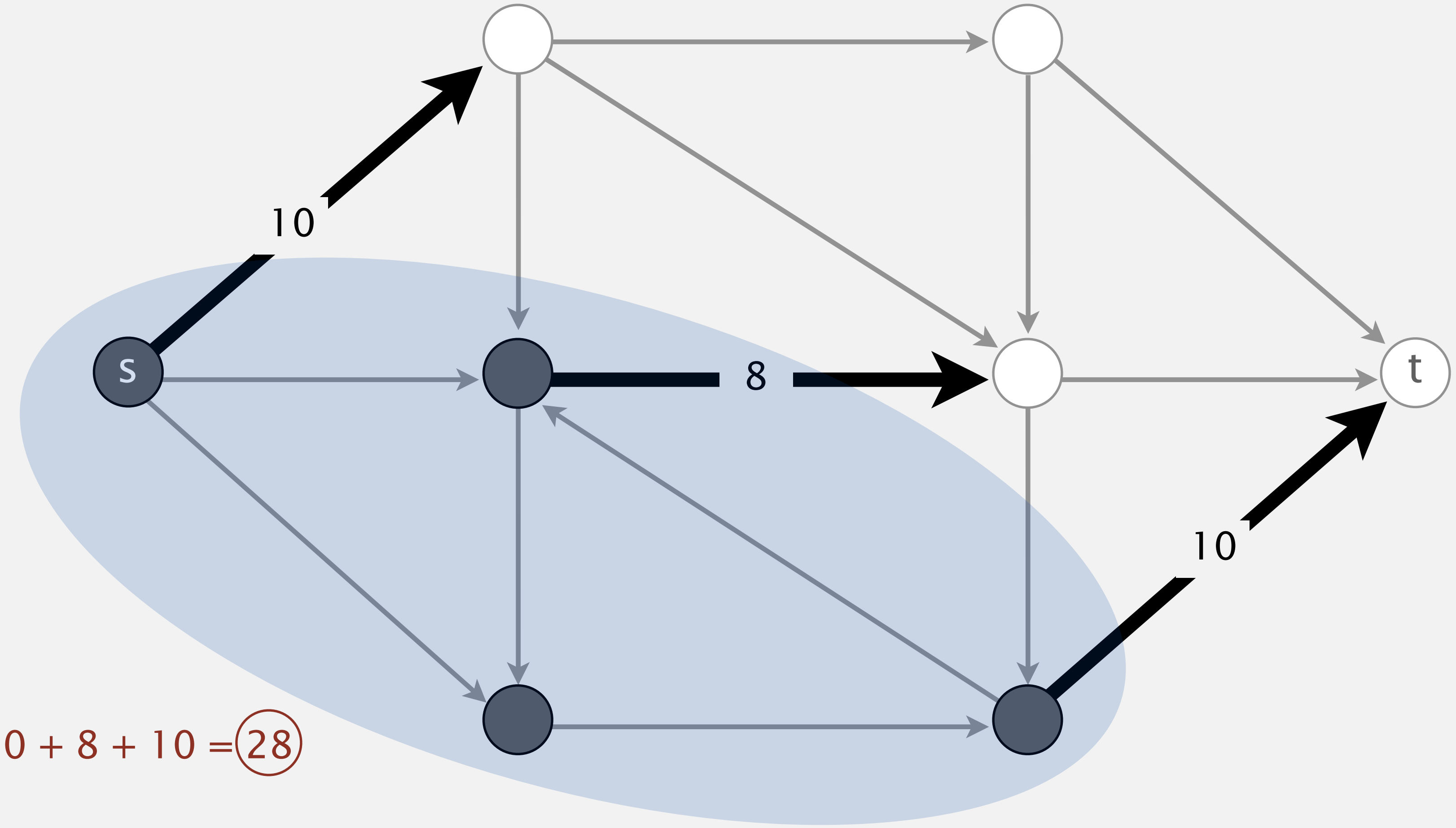
capacity = 10 + 8 + 16 = 34

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.



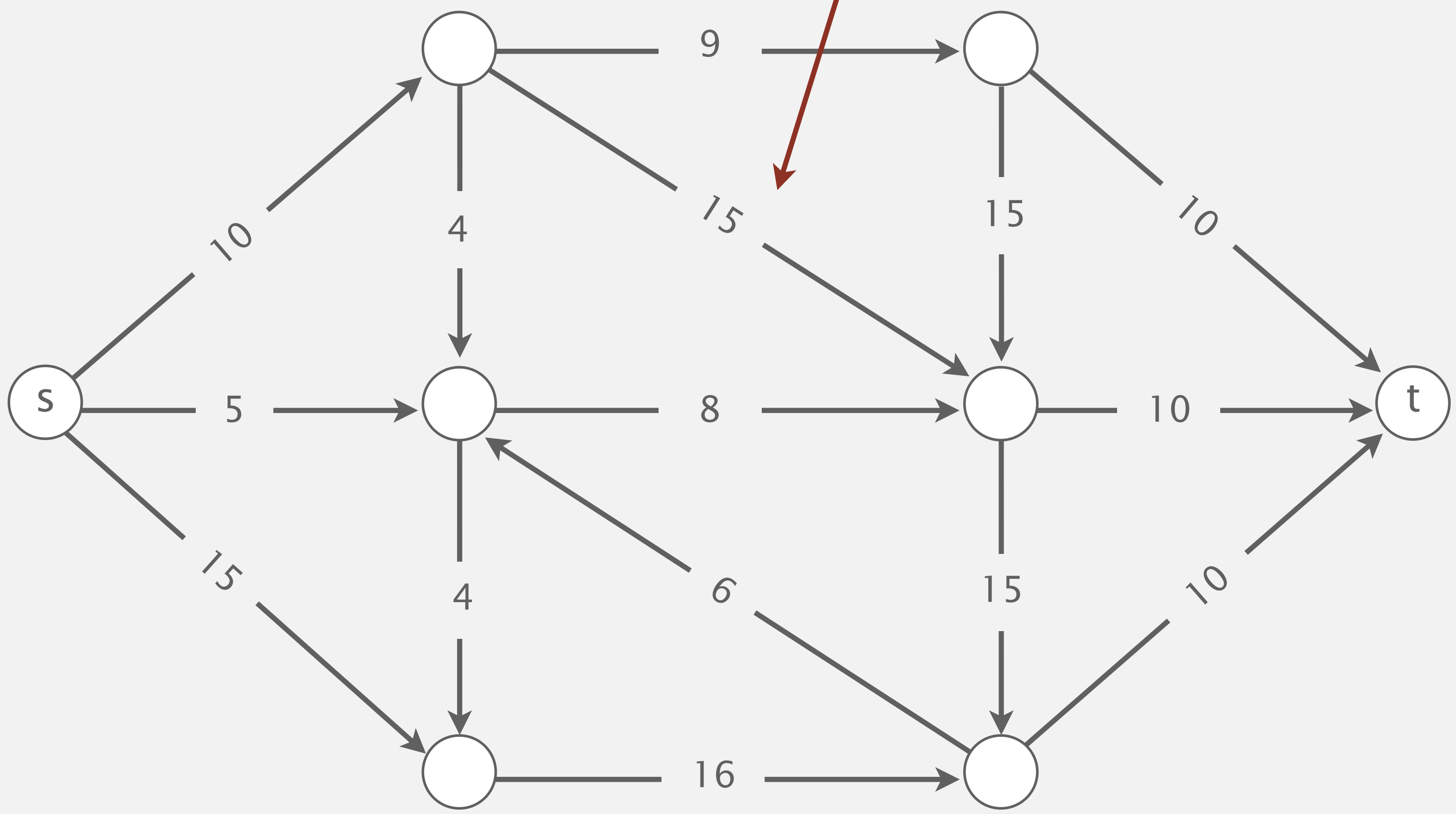
capacity = 10 + 8 + 10 = 28

Maxflow problem

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

each edge has a positive capacity

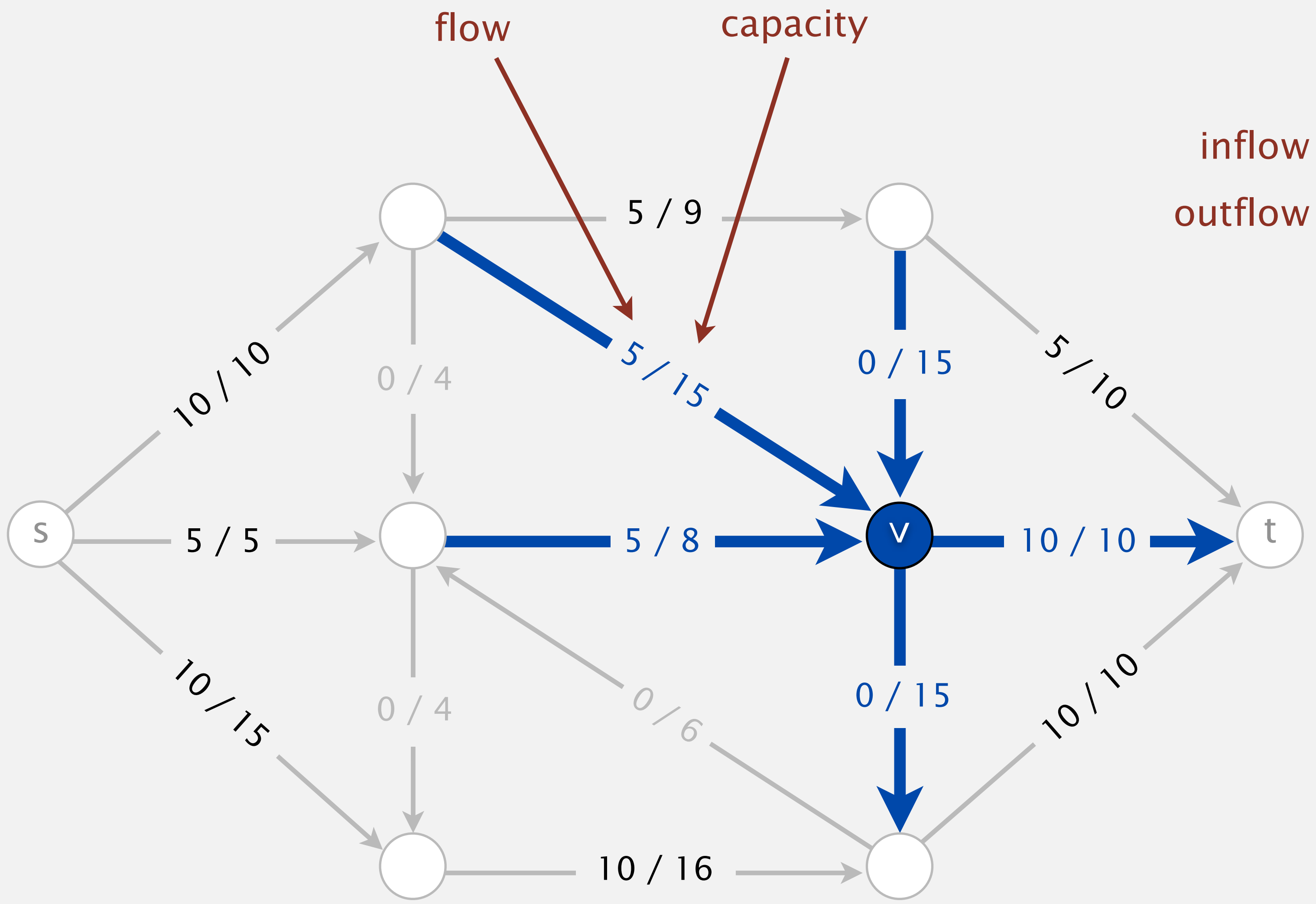
capacity



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*).



inflow at v = 5 + 5 + 0 = 10

outflow at v = 10 + 0 = 10

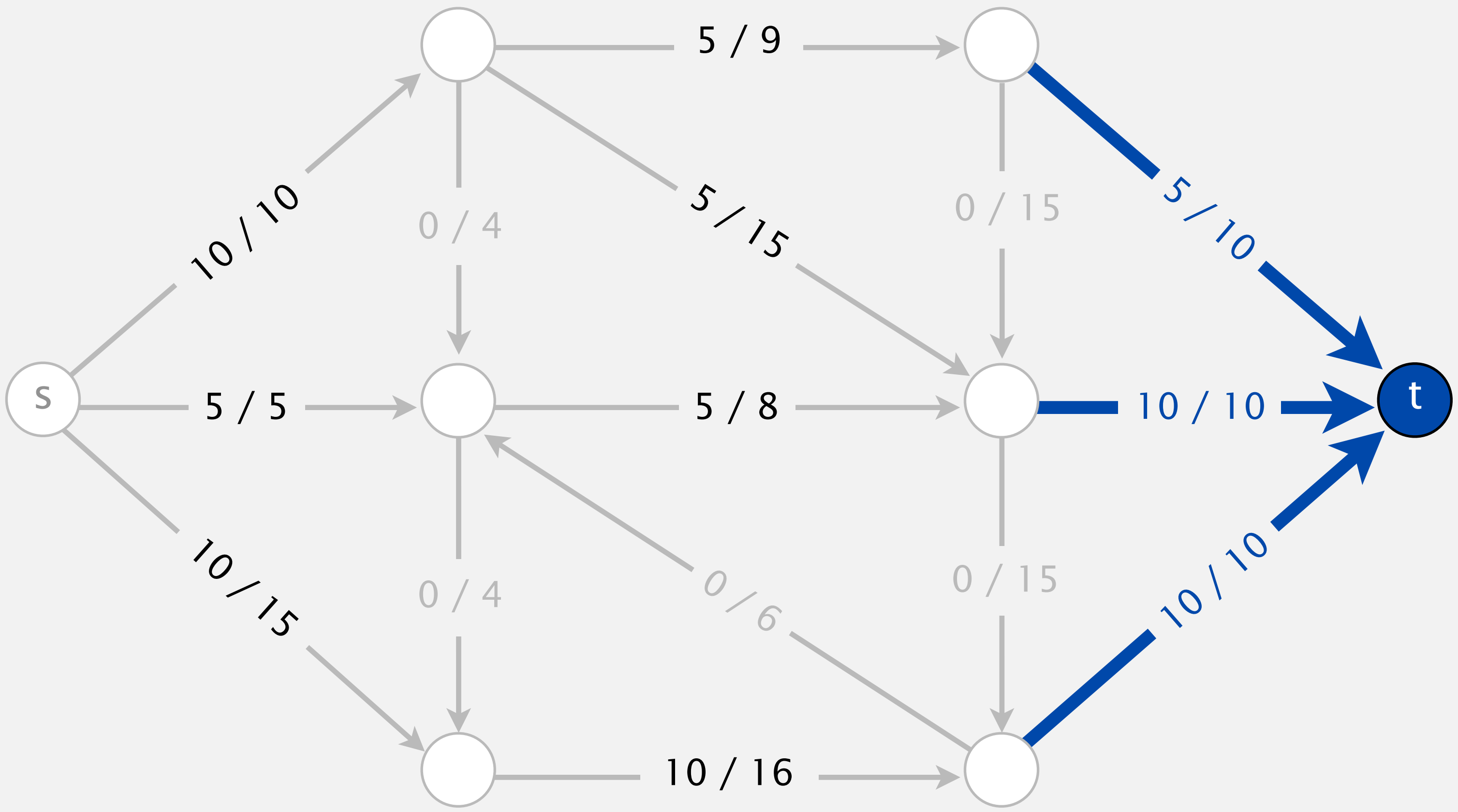
Maxflow problem

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

- Capacity constraint: 0 ≤ edge's flow ≤ 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



value = 5 + 10 + 10 = 25

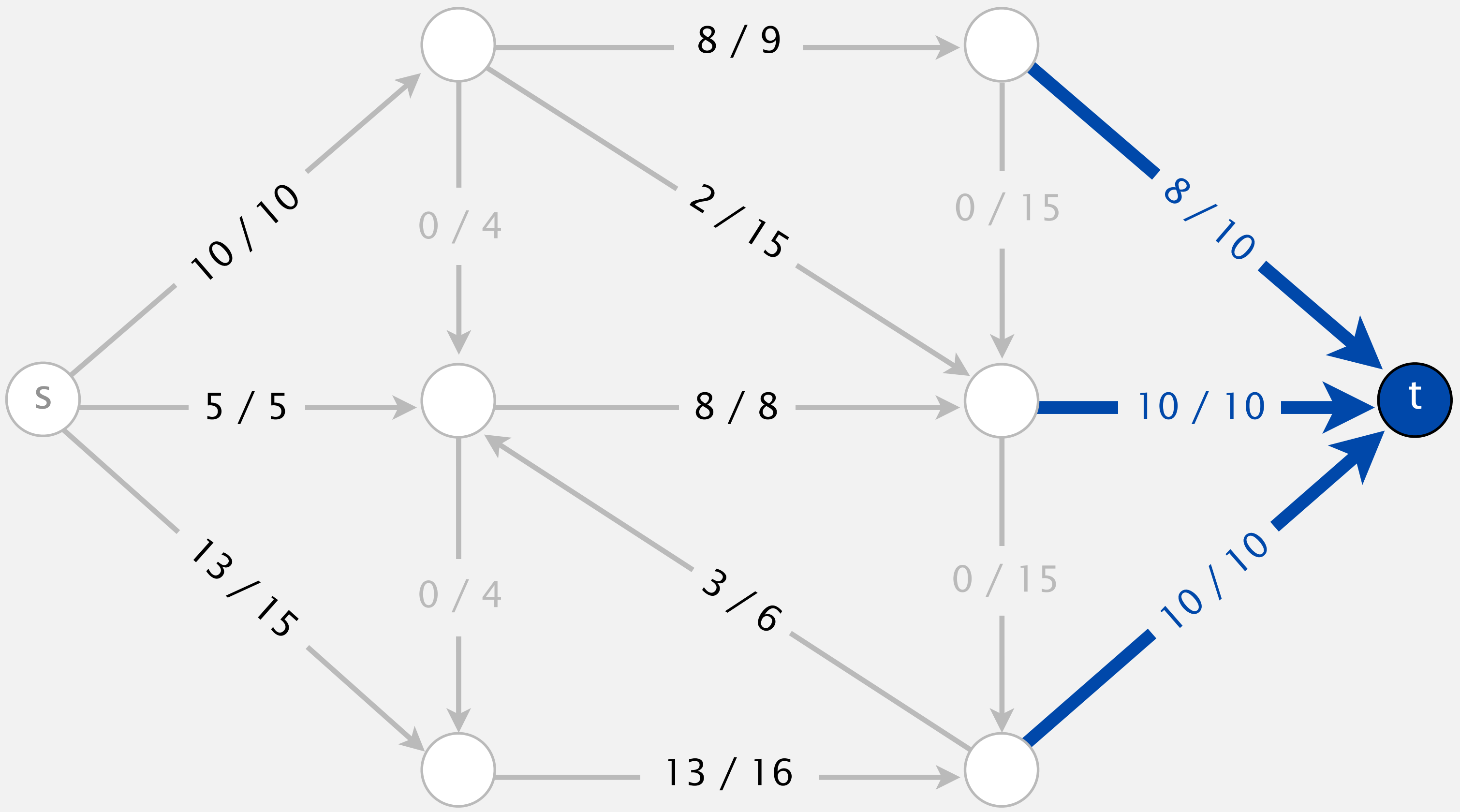
Maxflow problem

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.

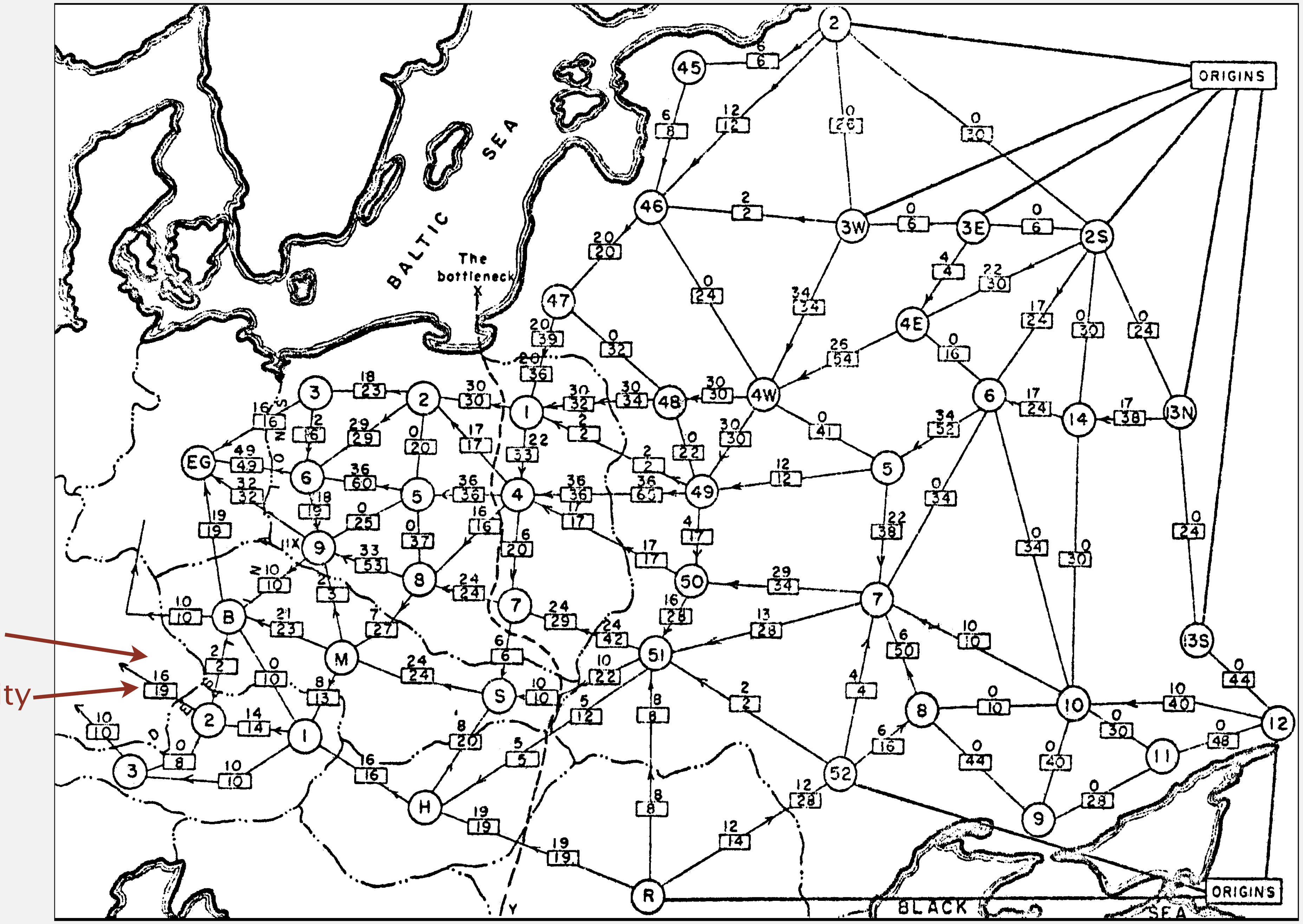
Maximum st-flow (maxflow) problem. Find a flow of maximum value.



value = 8 + 10 + 10 = $\boxed{28}$

Maxflow application (Tolstoǐ 1930s)

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



flow

capacity

rail network connecting Soviet Union with Eastern European countries

(map declassified by Pentagon in 1999)

Potential maxflow application (2010s)

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

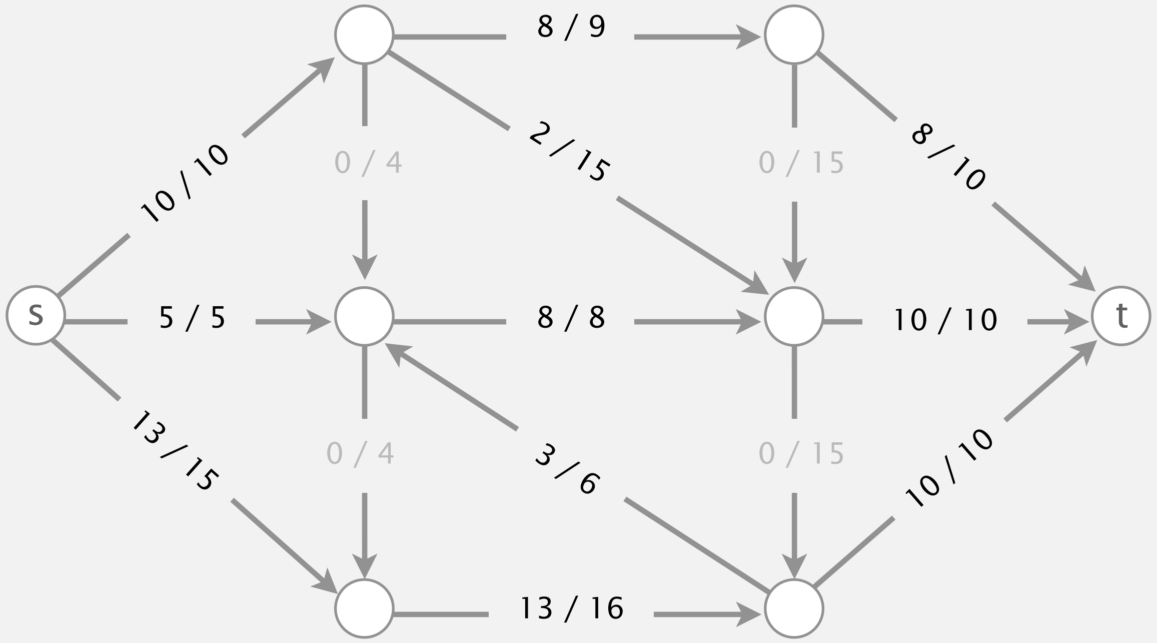
facebook graph

Summary

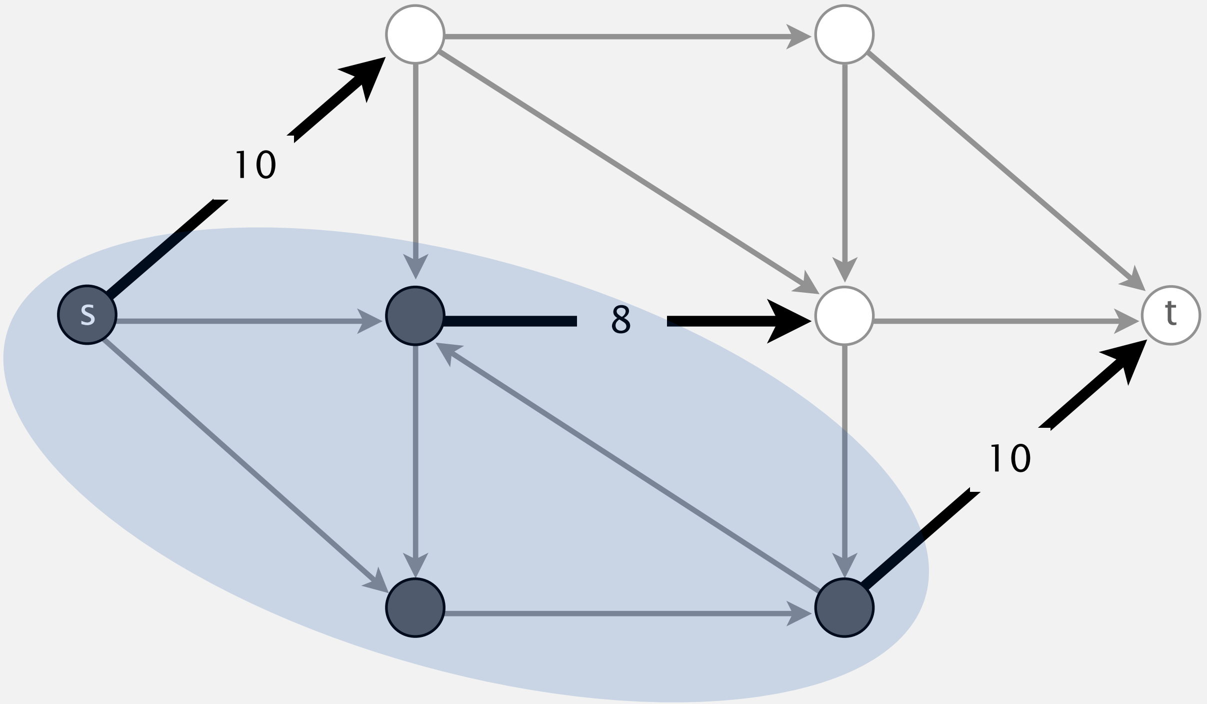
Input. A weighted digraph, source vertex s, and target vertex t.

Mincut problem. Find a cut of minimum capacity.

Maxflow problem. Find a flow of maximum value.



value of flow = 28



capacity of cut = 28

Remarkable fact. These two problems are same!

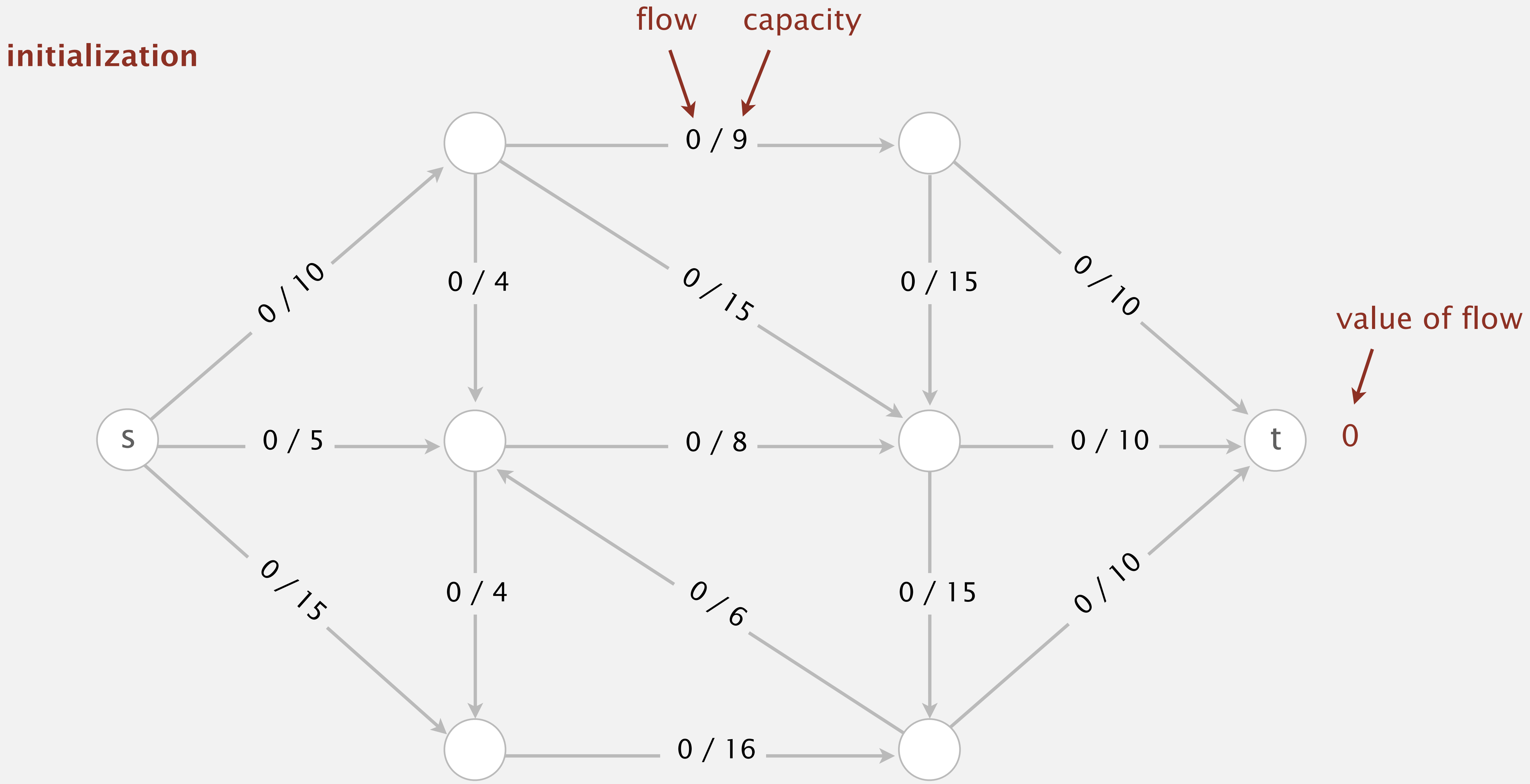
http://algs4.cs.princeton.edu

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.

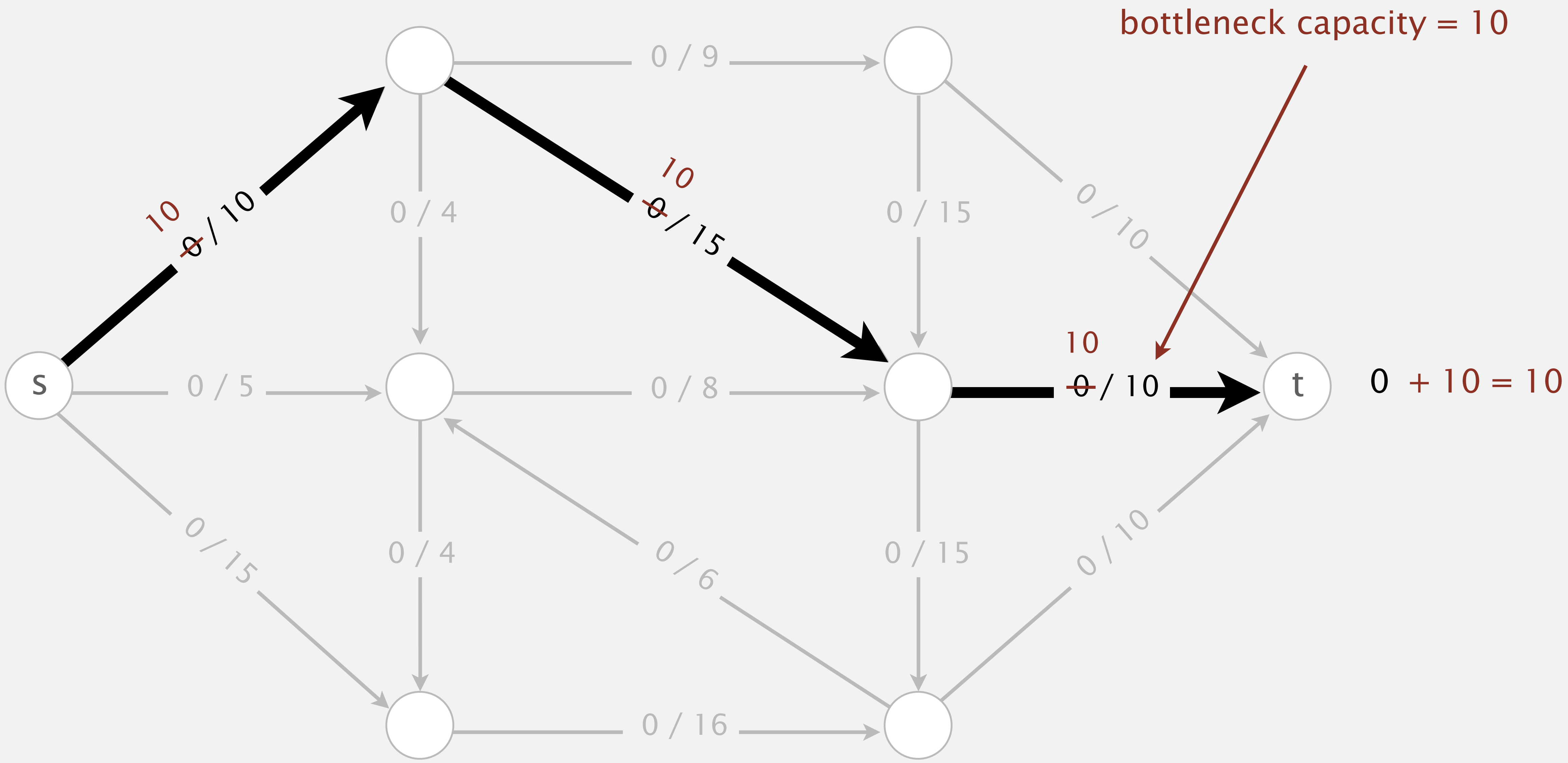


Idea: increase flow along augmenting paths

Augmenting path. 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).

1st augmenting path

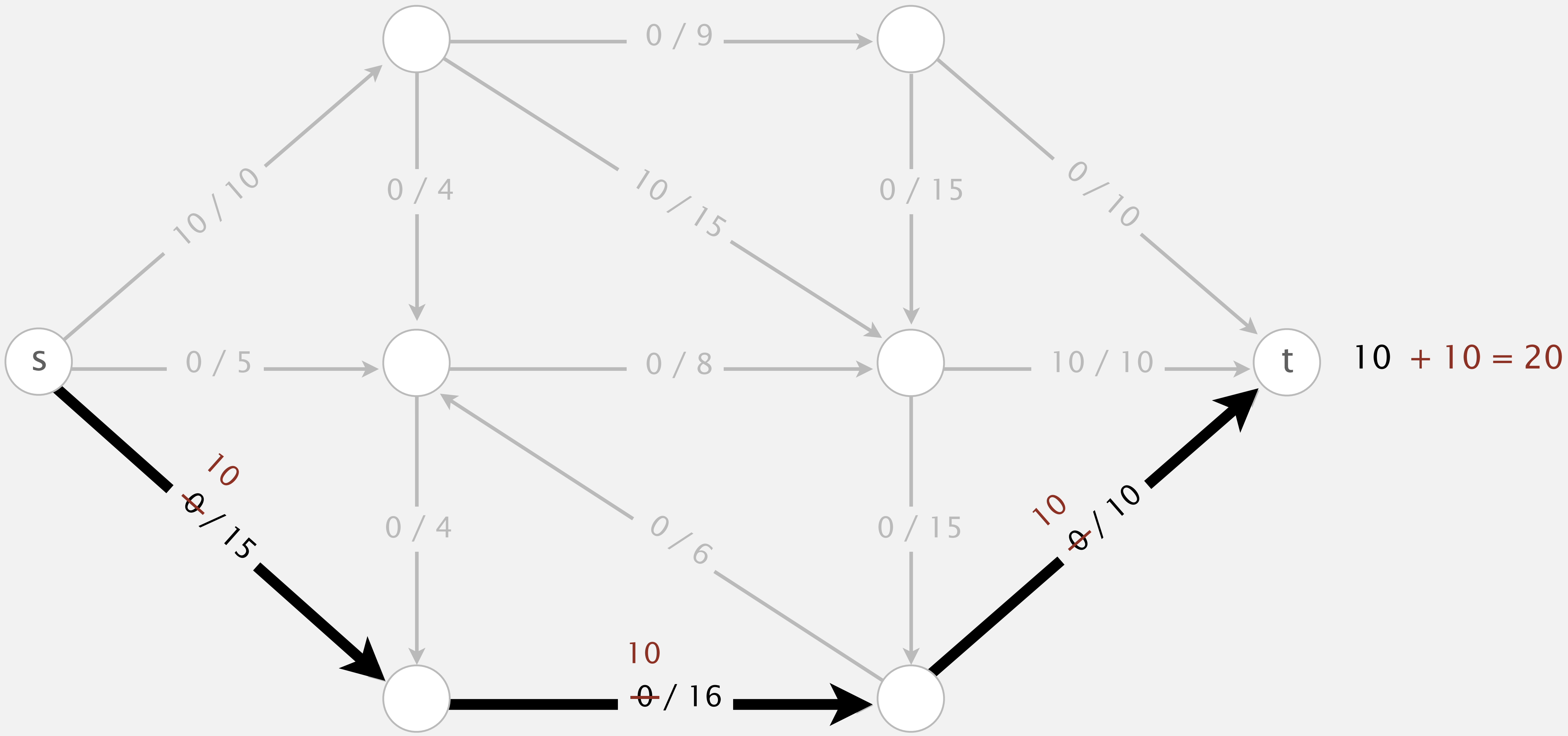


Idea: increase flow along augmenting paths

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).

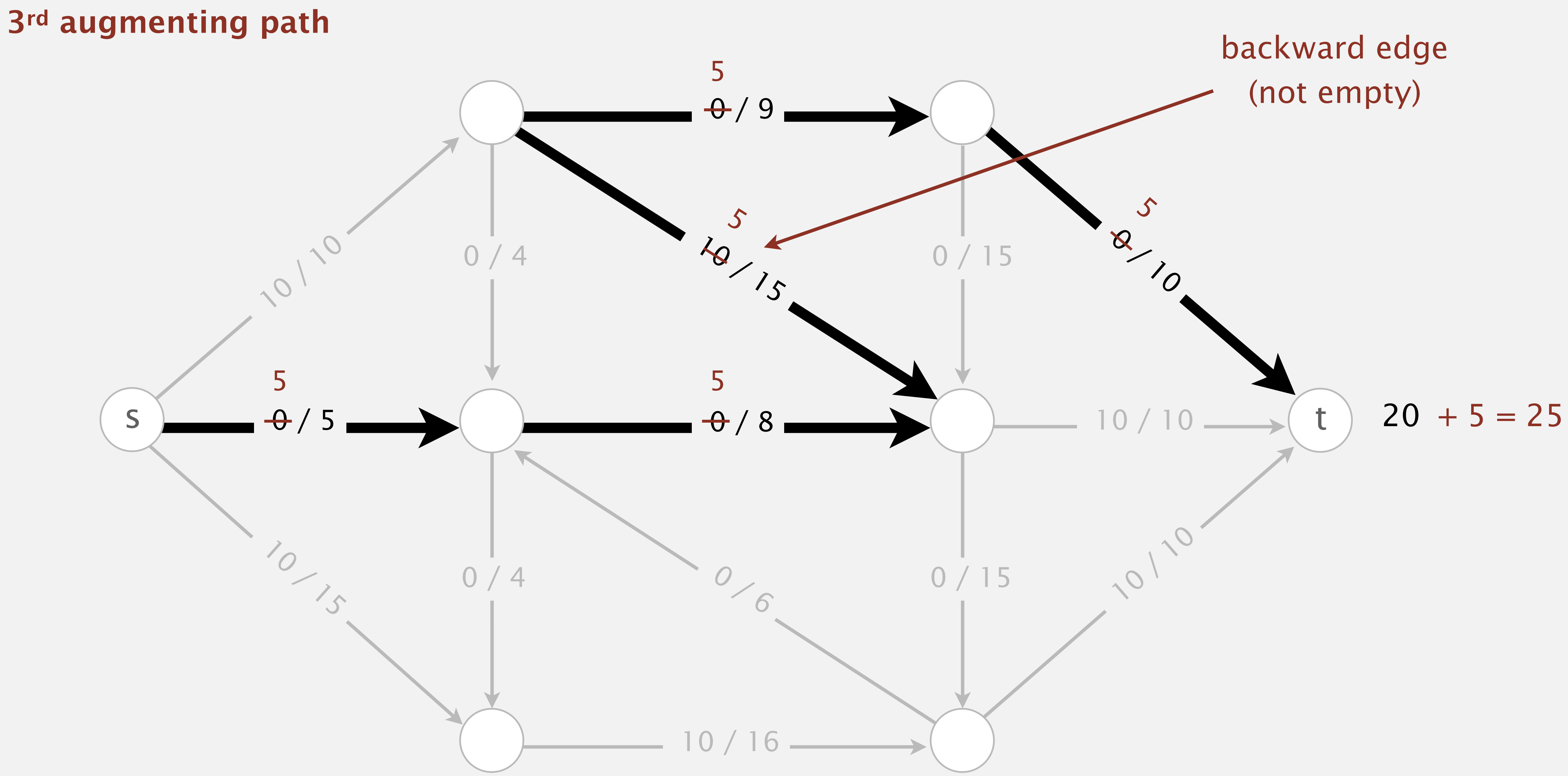
2nd augmenting path



Idea: increase flow along augmenting paths

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).

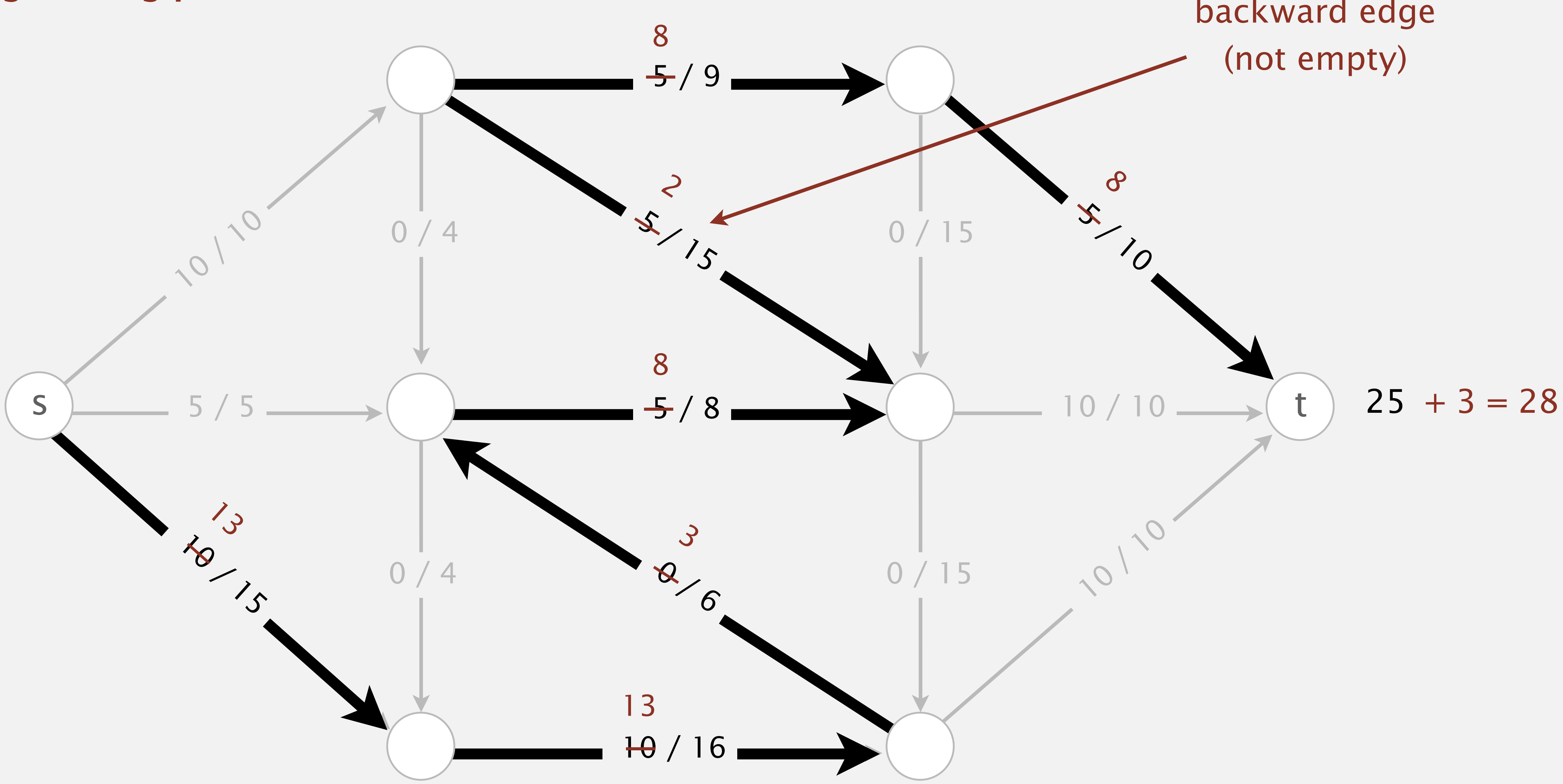


Idea: increase flow along augmenting paths

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).

4th augmenting path



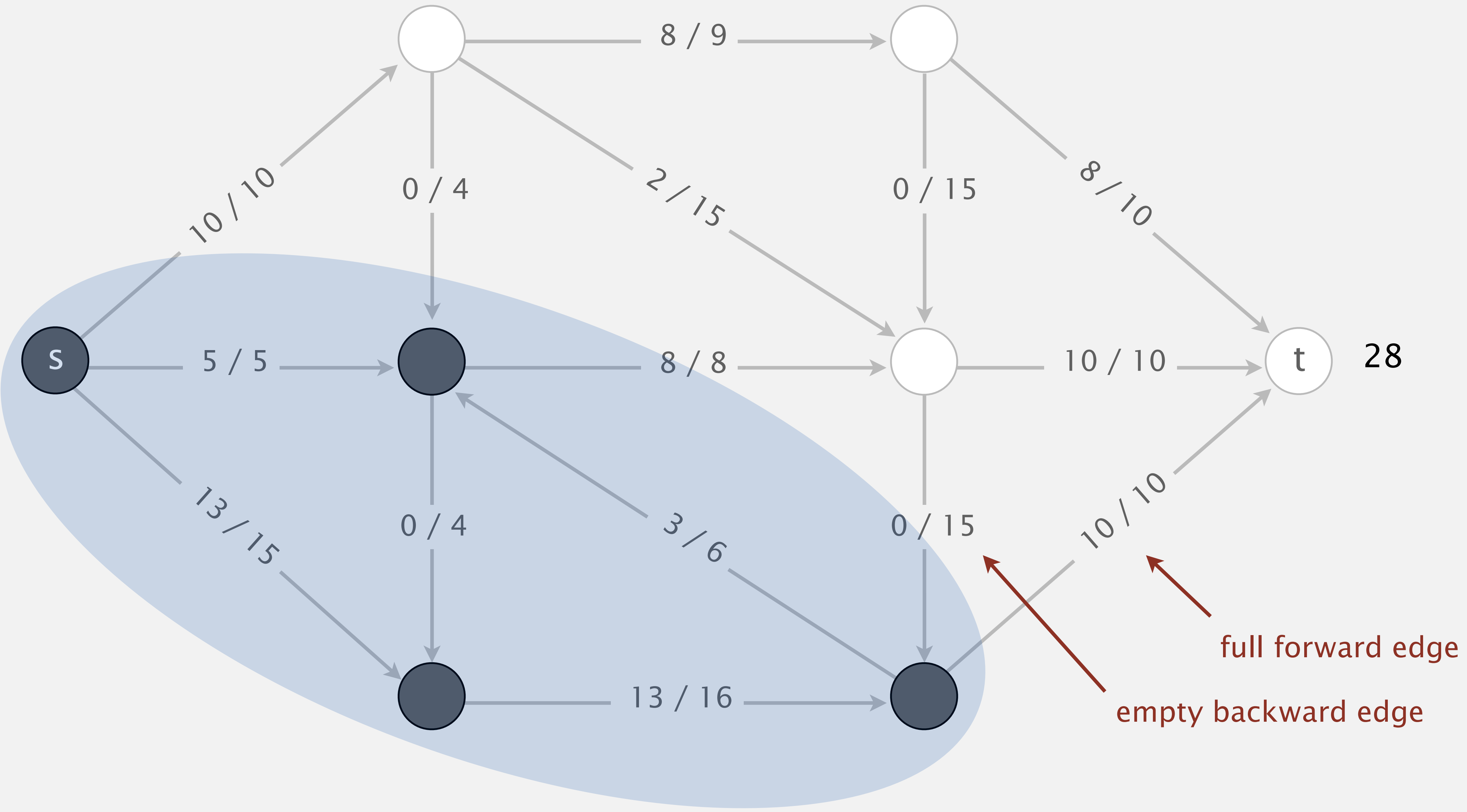
backward edge (not empty)

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



full forward edge

empty backward edge

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?
- 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?

http://algs4.cs.princeton.edu

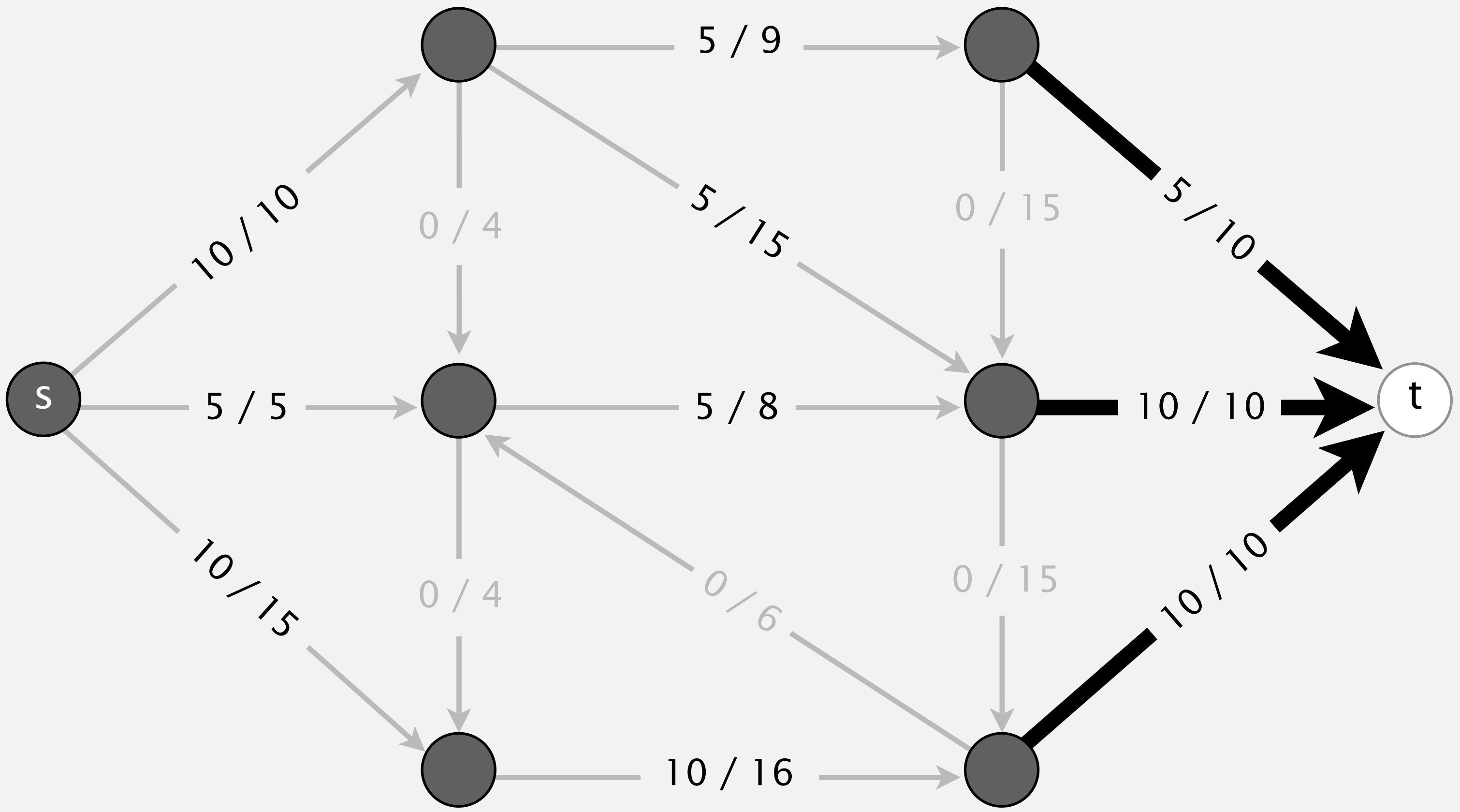
6.4 Maximum Flow

- *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.

net flow across cut = 5 + 10 + 10 = 25

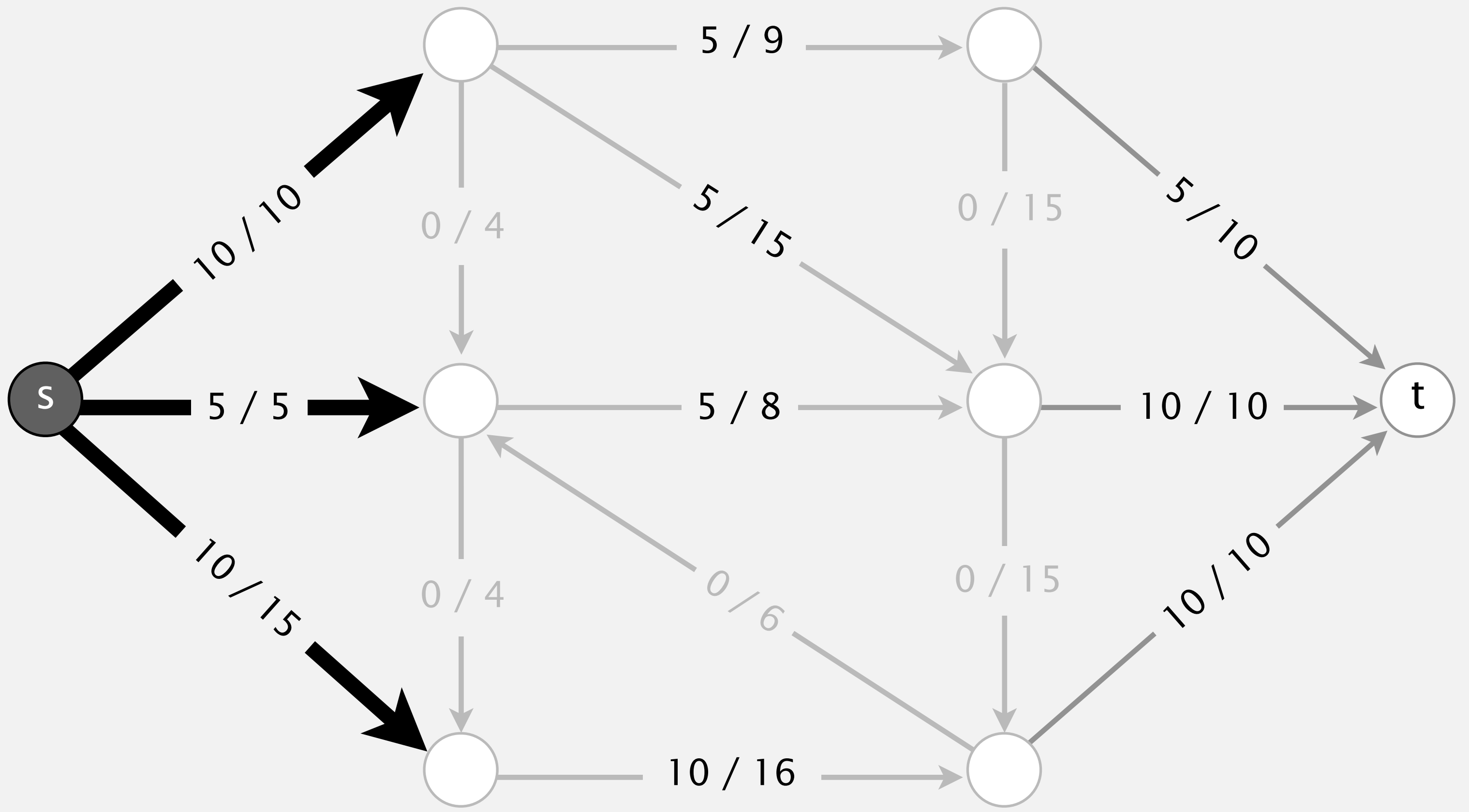


value of flow = 25

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.

net flow across cut = 10 + 5 + 10 = 25

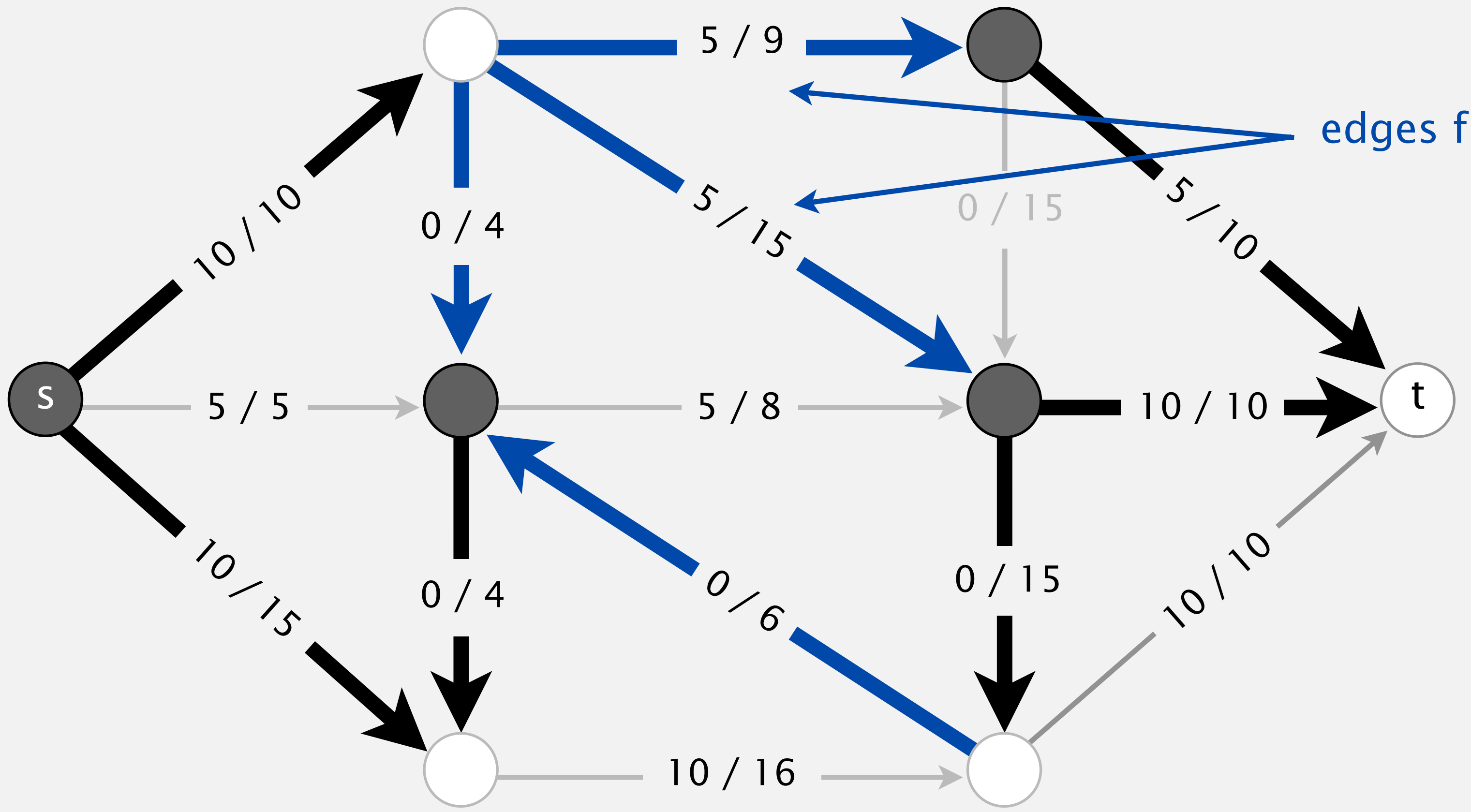


value of flow = 25

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.

net flow across cut = (10 + 10 + 5 + 10 + 0 + 0) – (5 + 5 + 0 + 0) = 25



value of flow = 25

Relationship between flows and cuts

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.

Key Idea. Outflow from s = inflow to t = value of flow.

Relationship between flows and cuts

Let f be any flow and let (A, B) be any cut.

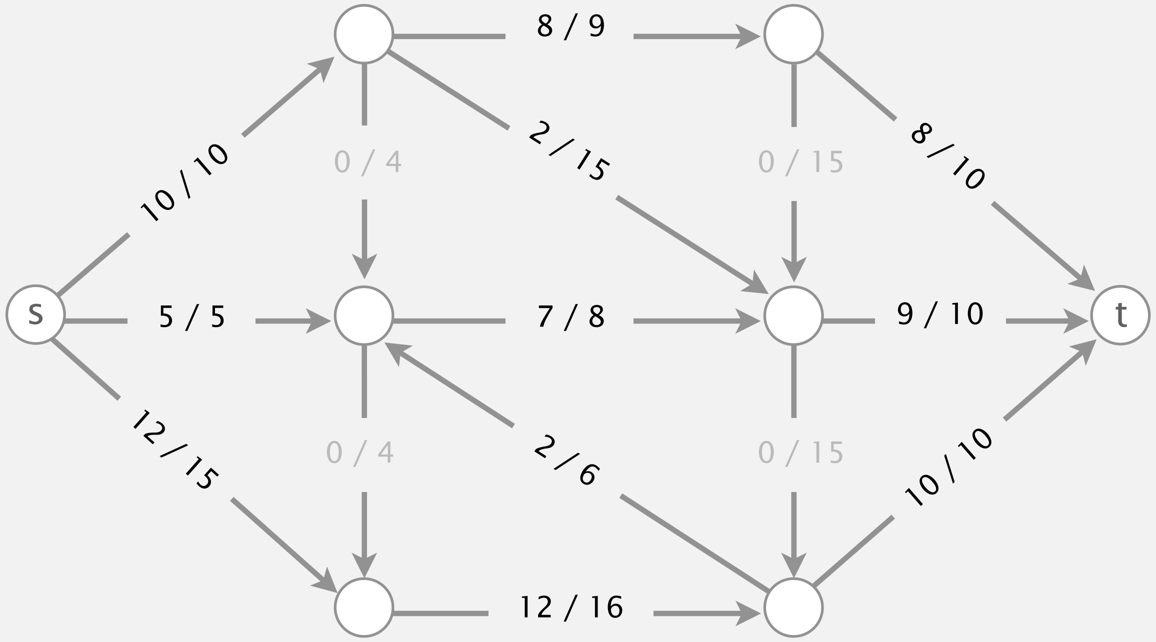
Then, the value of the flow ≤ the capacity of the cut.

Value of flow f = net flow across cut (A, B) ≤ capacity of cut (A, B).

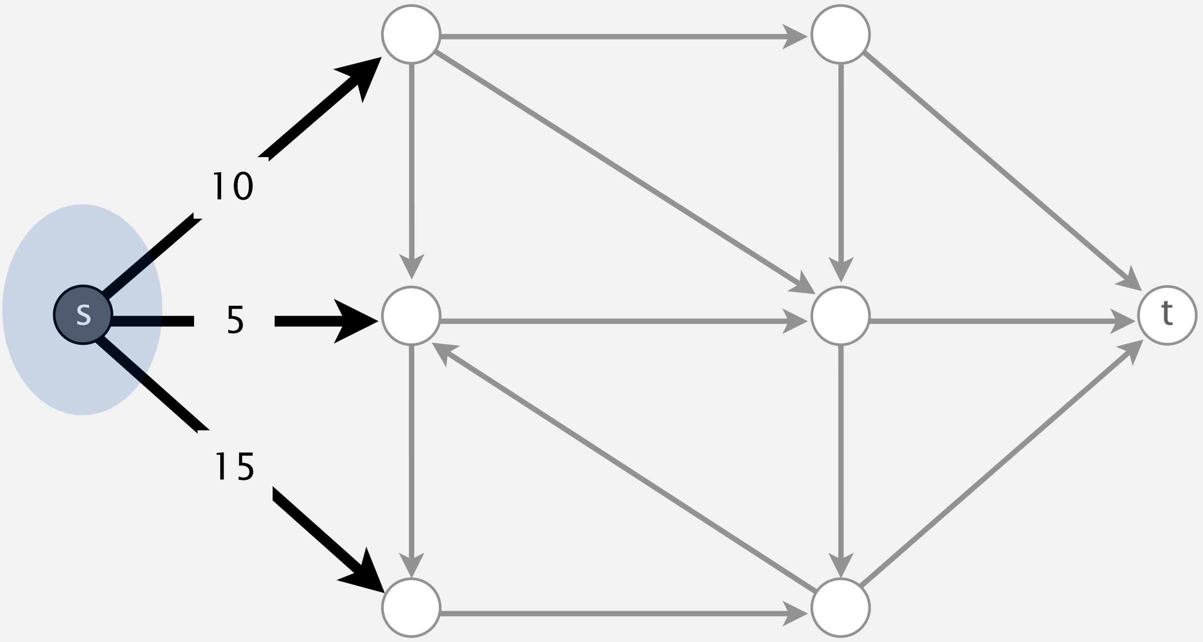
flow-value lemma

flow bounded by capacity

Think of the nodes collapsing on themselves.



value of flow = 27



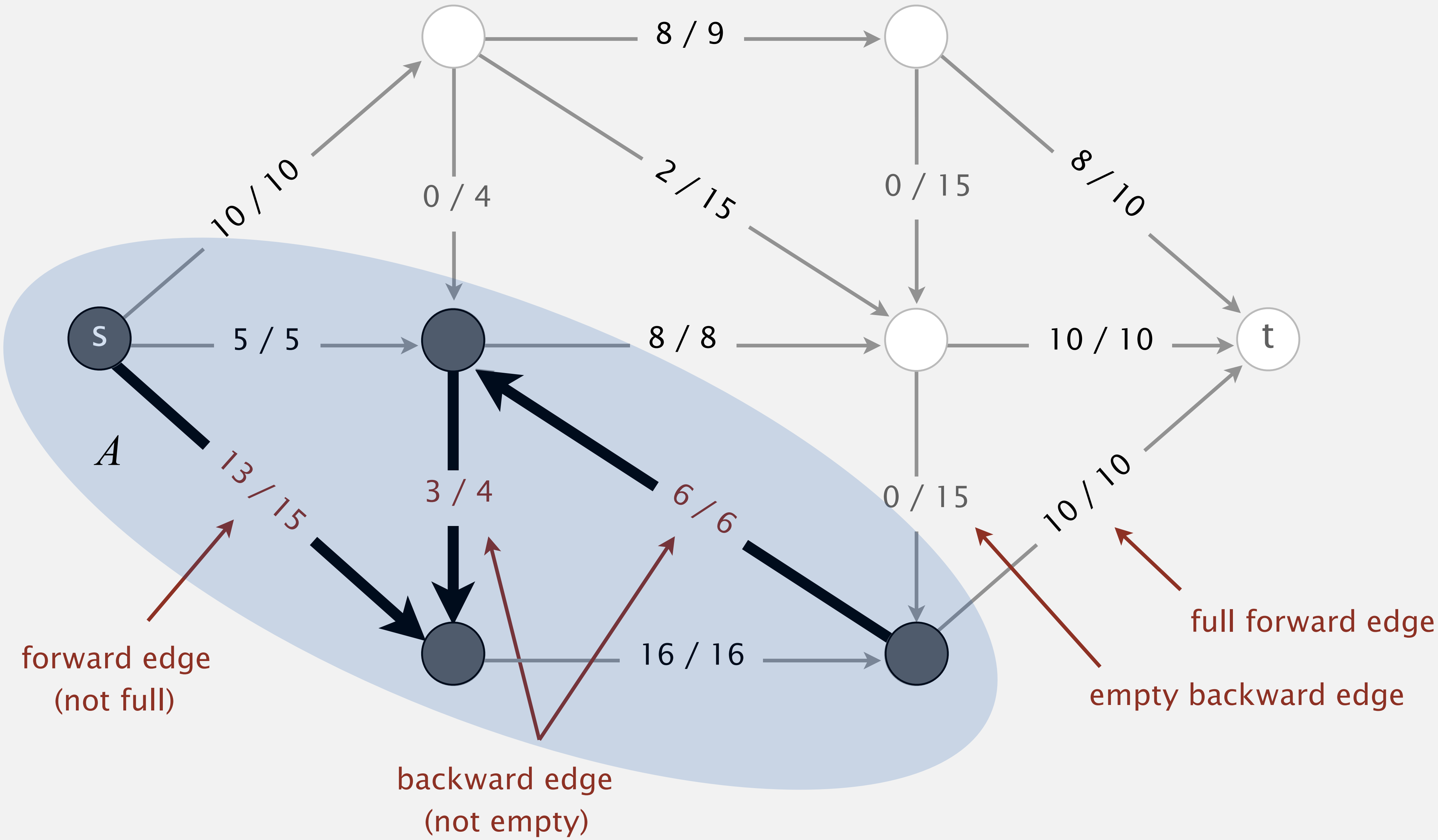
capacity of cut = 30

Computing a mincut from a maxflow

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

- Compute A = set of vertices connected to s by an undirected path with no full forward or empty backward edges.

Think of running DFS on the undirected graph that with full forward edges and empty backward edges removed



http://algs4.cs.princeton.edu

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?

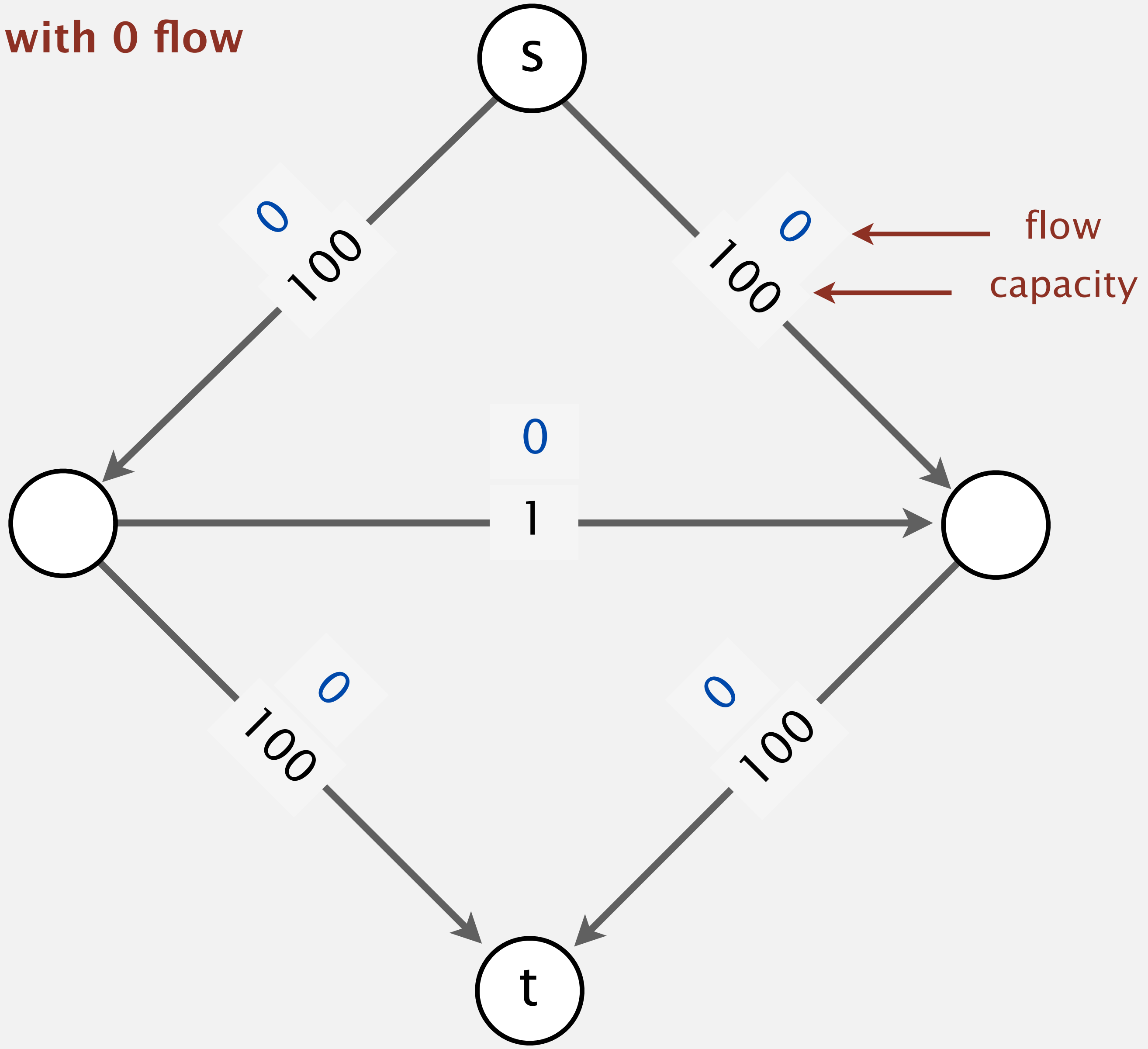
yes, provided edge capacities are integers
(or augmenting paths are chosen carefully)

requires clever analysis

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.

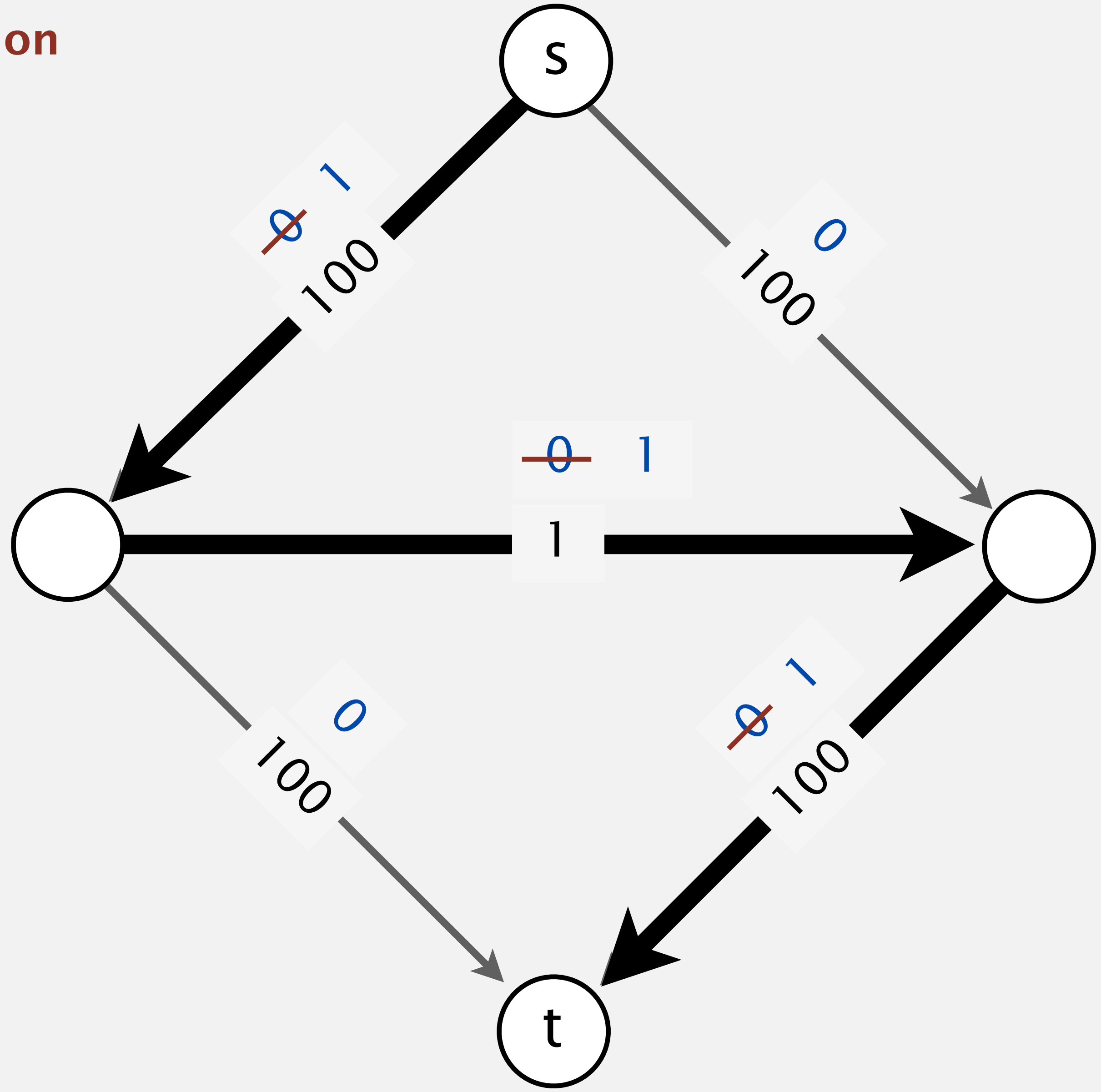
initialize with 0 flow



Bad case for Ford-Fulkerson

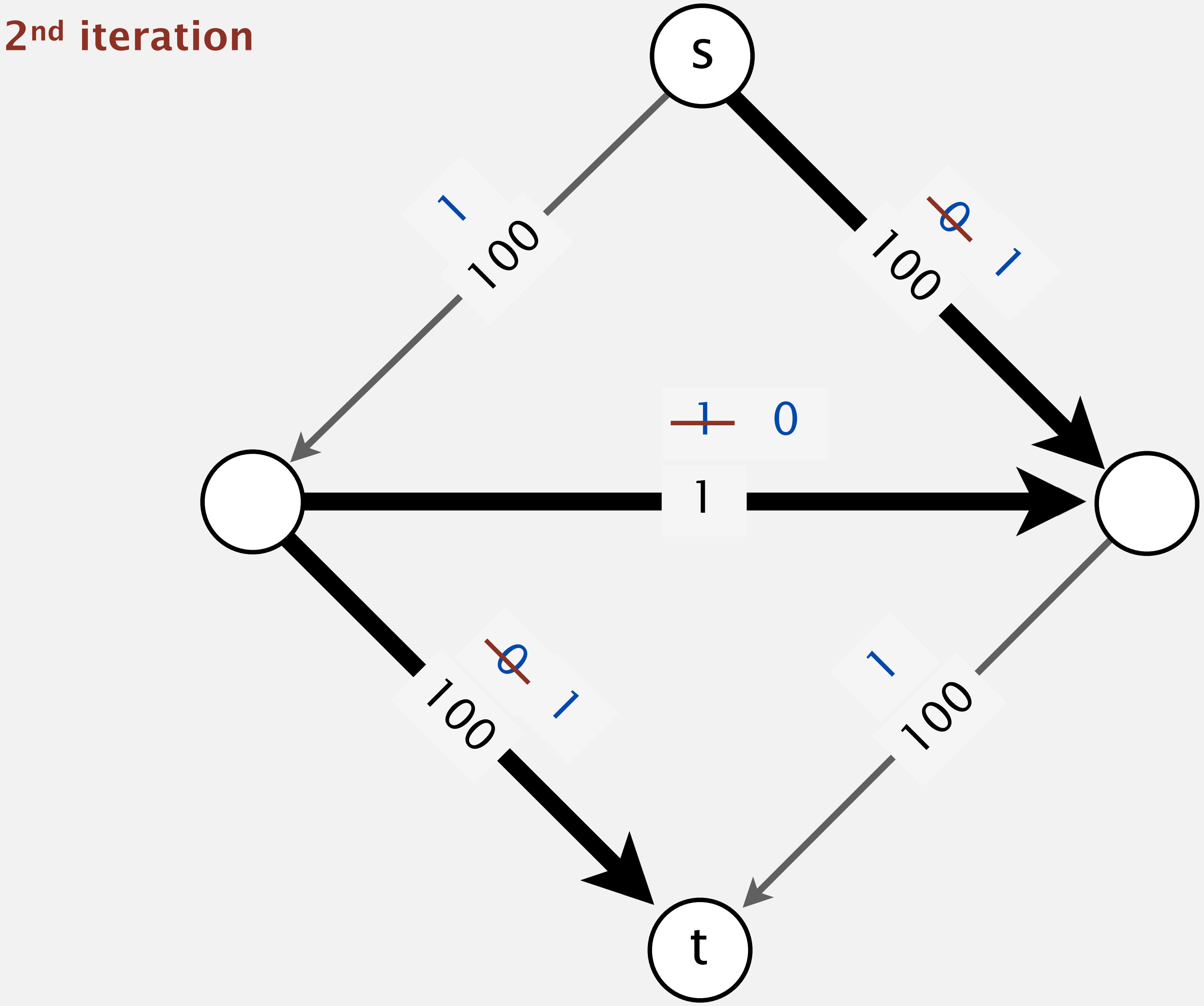
Bad news. Even when edge capacities are integers, number of augmenting paths could be equal to the value of the maxflow.

1st iteration



Bad case for Ford-Fulkerson

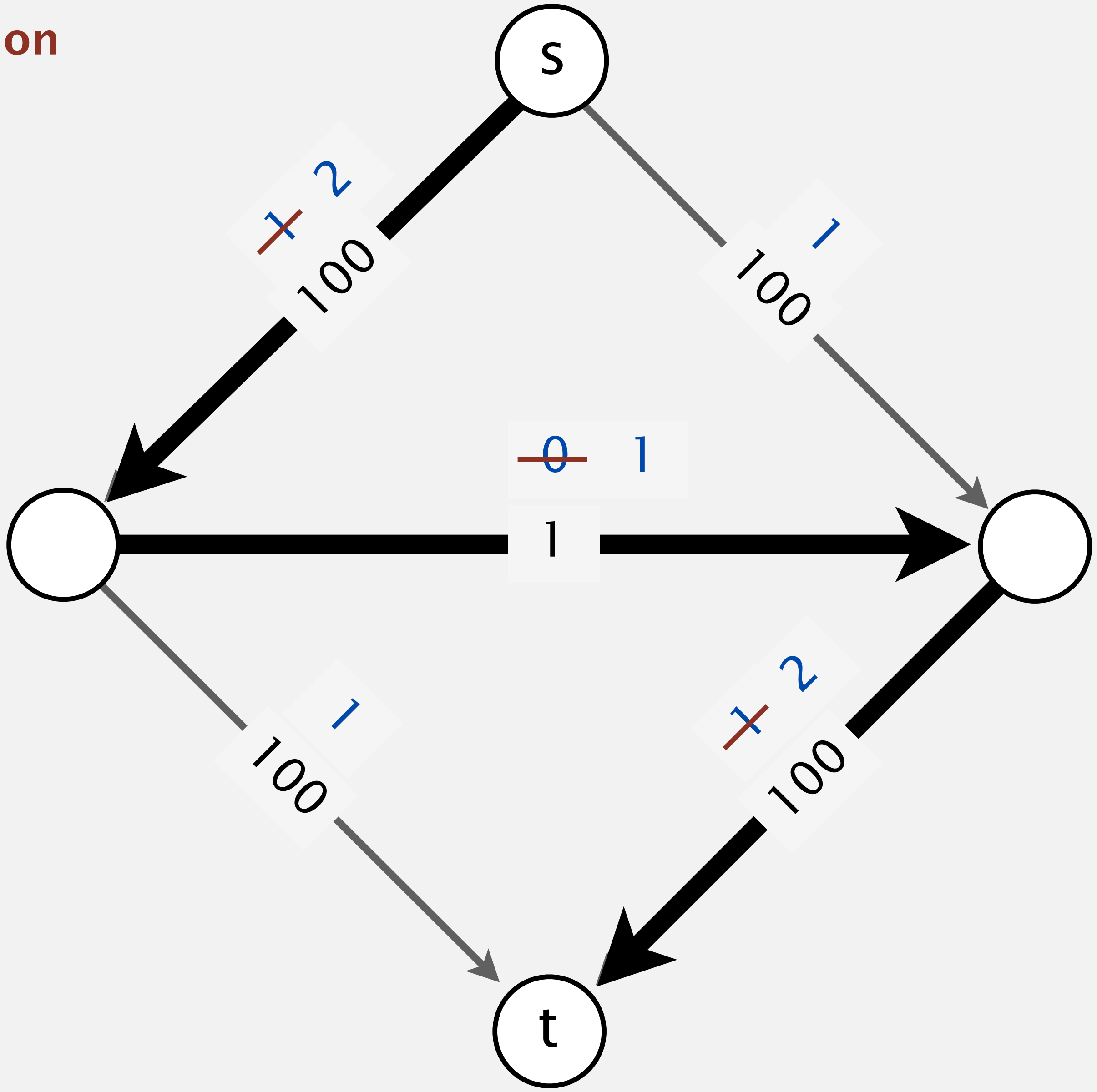
Bad news. Even when edge capacities are integers, number of augmenting paths could be equal to the value of the maxflow.



Bad case for Ford-Fulkerson

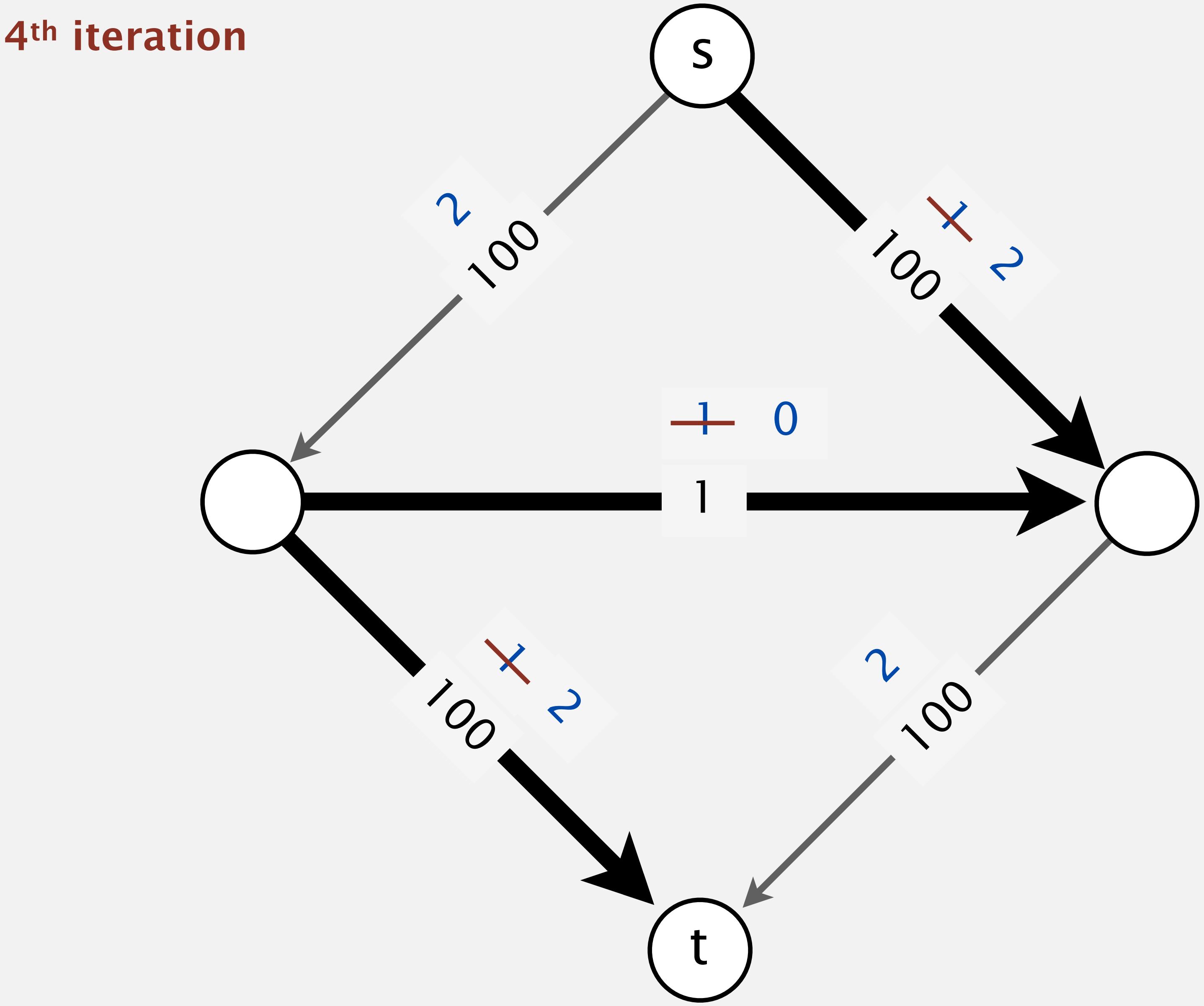
Bad news. Even when edge capacities are integers, number of augmenting paths could be equal to the value of the maxflow.

3rd iteration



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.



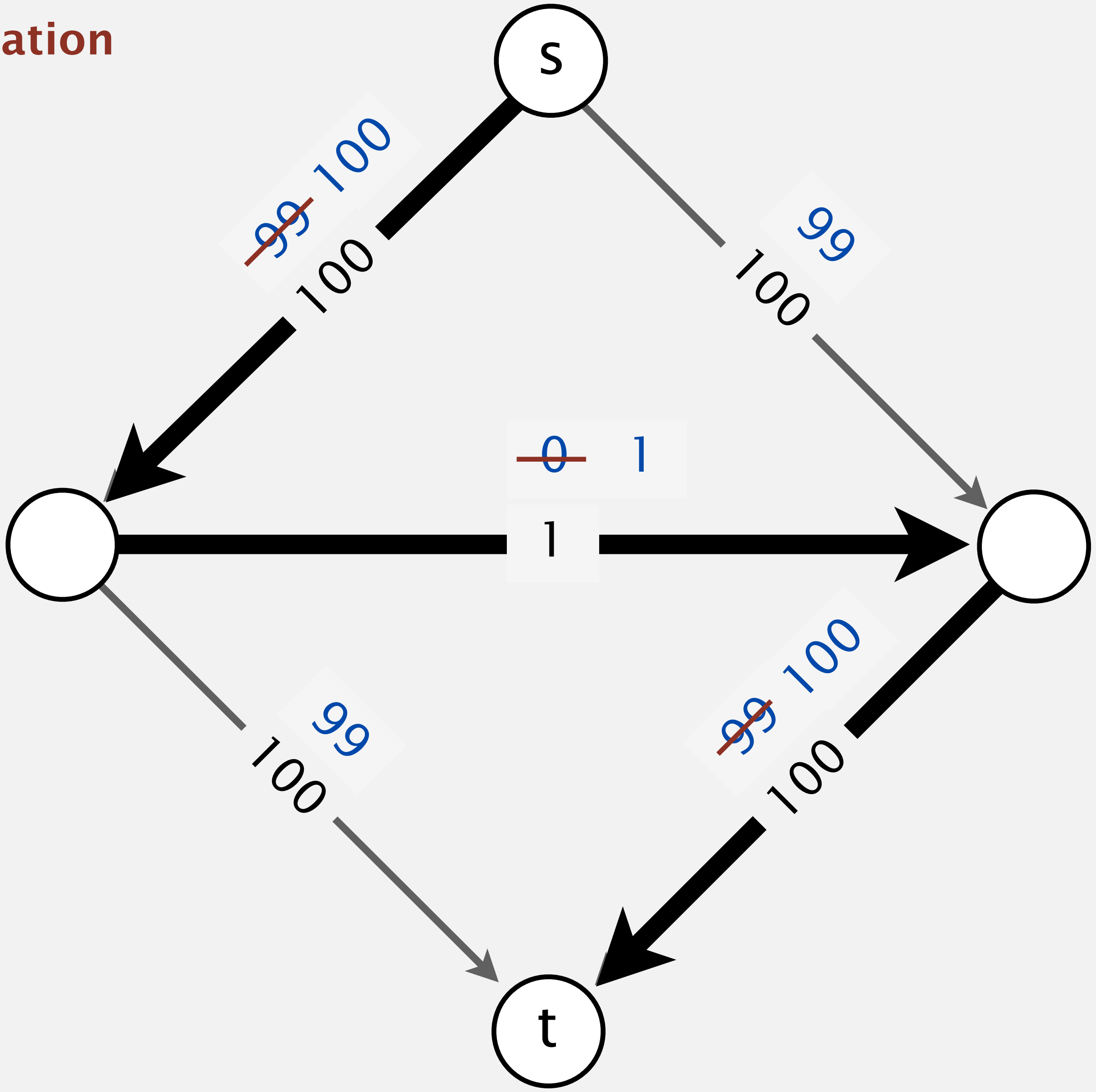
Bad case for Ford-Fulkerson

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

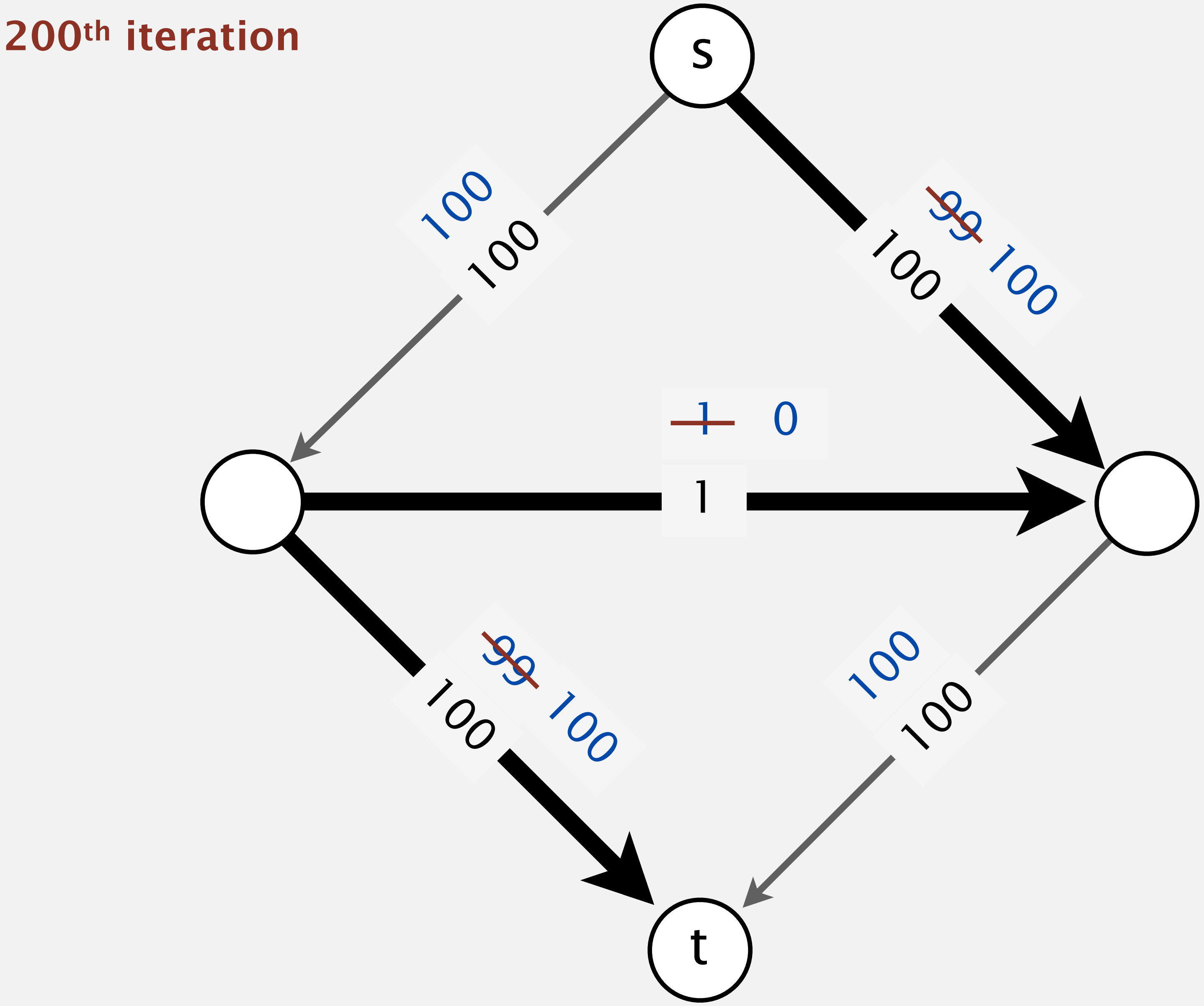
Bad news. Even when edge capacities are integers, number of augmenting paths could be equal to the value of the maxflow.

199th iteration



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.

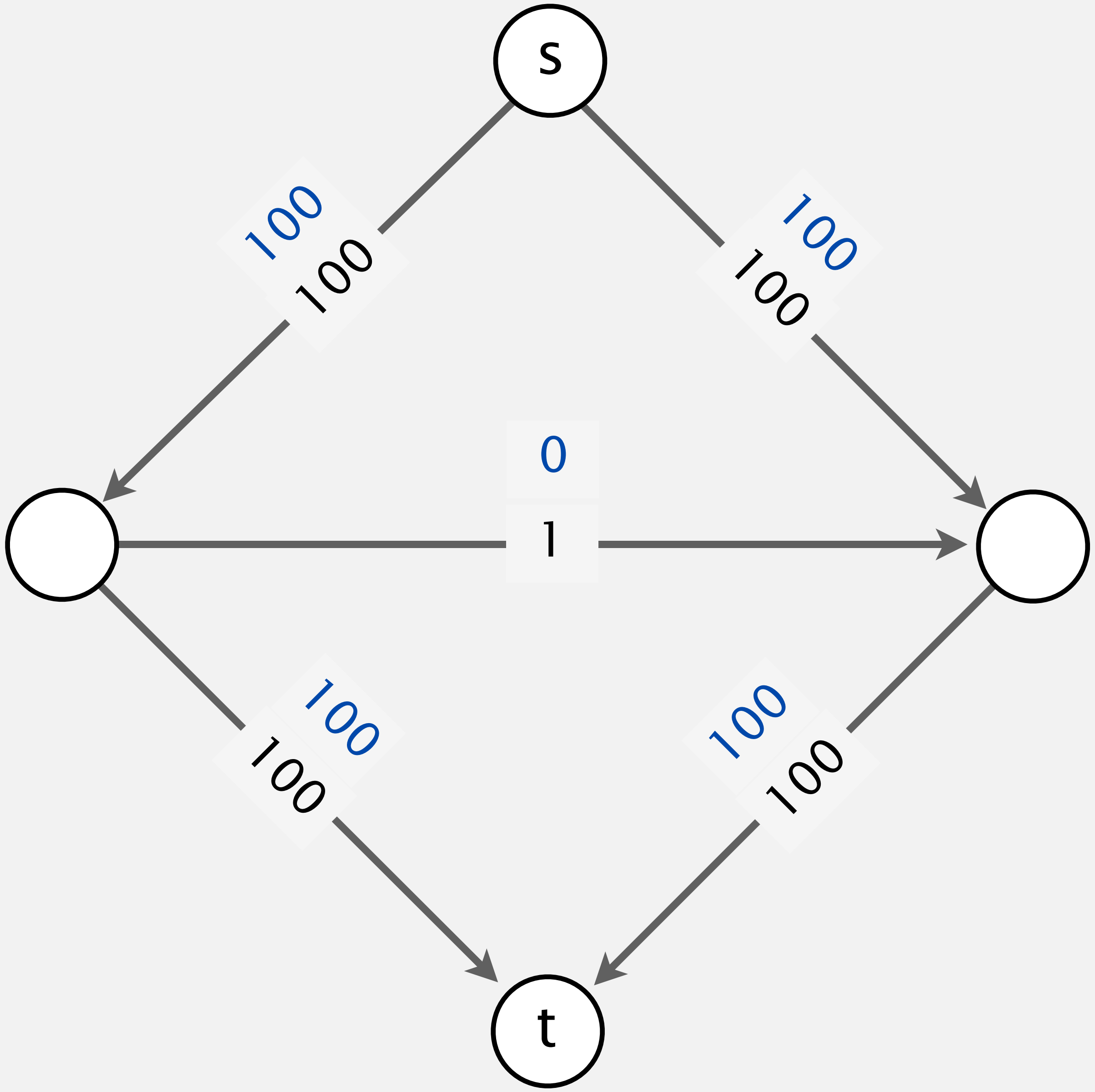


Bad case for Ford-Fulkerson

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

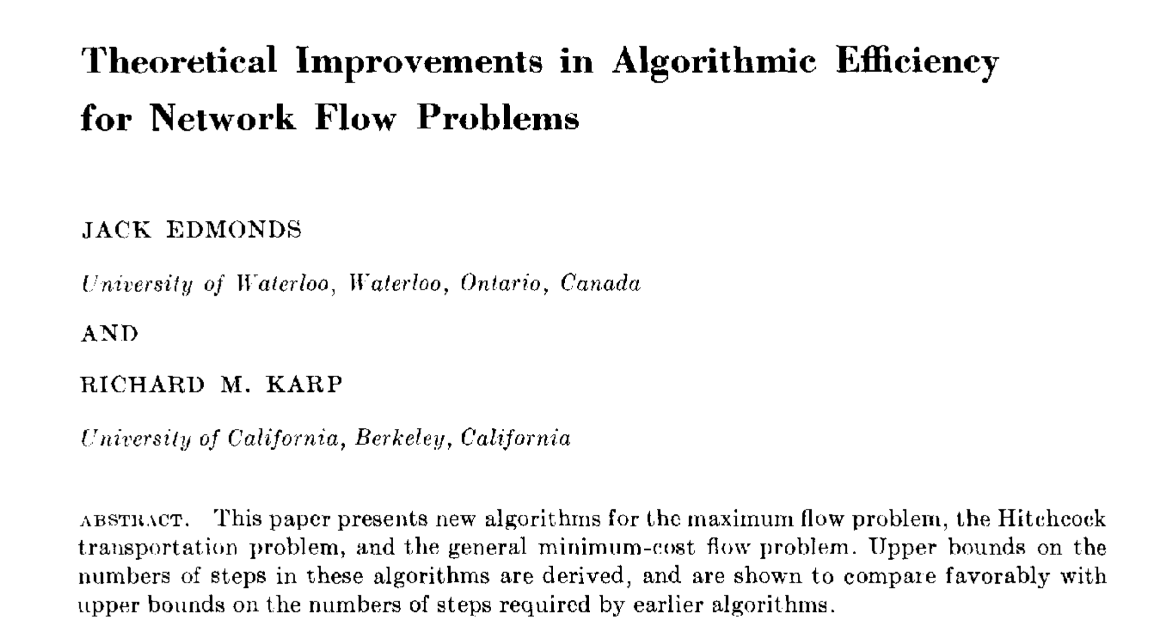
Good news. This case is easily avoided. [use shortest/fattest path]



How to choose augmenting paths?

Choose augmenting paths with:

- Shortest path: fewest number of edges.
- Fattest path: max bottleneck capacity.



Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems

JACK EDMONDS

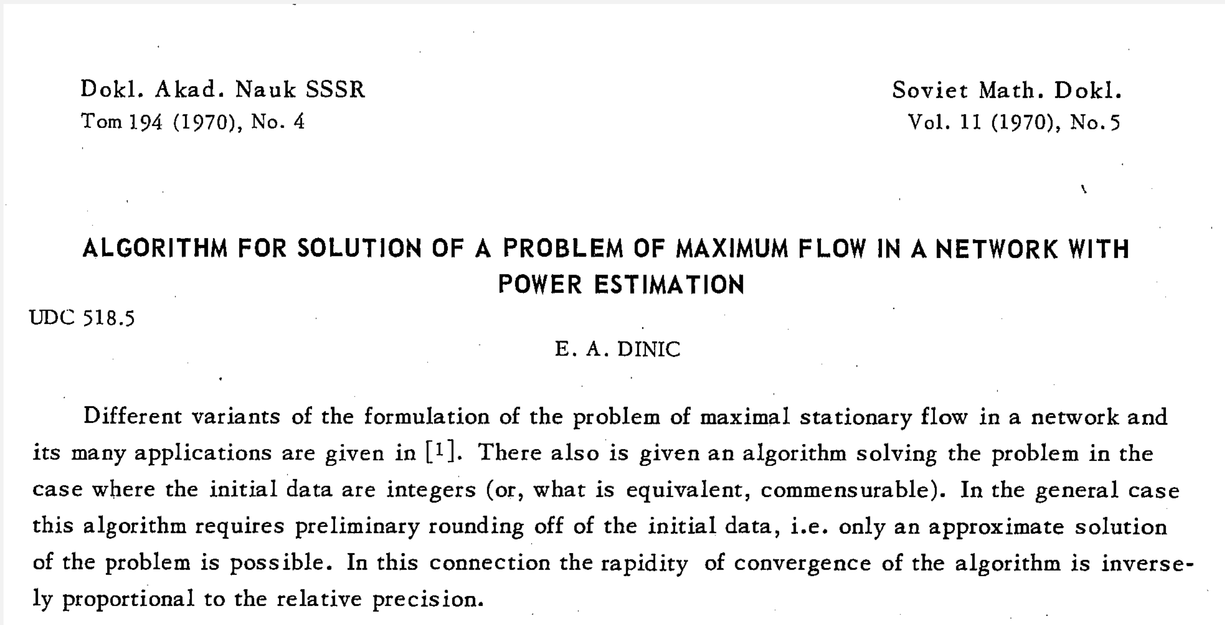
University of Waterloo, Waterloo, Ontario, Canada

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.



Dokl. Akad. Nauk SSSR
Tom 194 (1970), No. 4

Soviet Math. Dokl.
Vol. 11 (1970), No. 5

ALGORITHM FOR SOLUTION OF A PROBLEM OF MAXIMUM FLOW IN A NETWORK WITH POWER ESTIMATION

UDC 518.5

E. A. DINIC

Different variants of the formulation of the problem of maximal stationary flow in a network and its many applications are given in [1]. There also is given an algorithm solving the problem in the case where the initial data are integers (or, what is equivalent, commensurable). In the general case this algorithm requires preliminary rounding off of the initial data, i.e. only an approximate solution of the problem is possible. In this connection the rapidity of convergence of the algorithm is inversely proportional to the relative precision.

Edmonds-Karp 1972 (USA)

Dinic 1970 (Soviet Union)

How to choose augmenting paths?

Use care when selecting augmenting paths.

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

augmenting path	number of paths	implementation
random path	$\leq E\ U$	randomized queue
shortest path	$\leq \frac{1}{2}\ E\ V$	queue (BFS)
fattest path	$\leq E \ln(E\ U)$	priority queue

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

Algorithms

Robert Sedgewick | Kevin Wayne

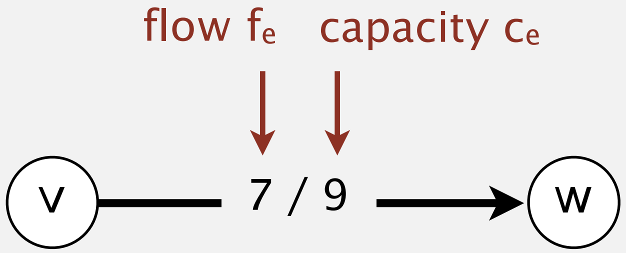
http://algs4.cs.princeton.edu

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 $e = v \to w$.



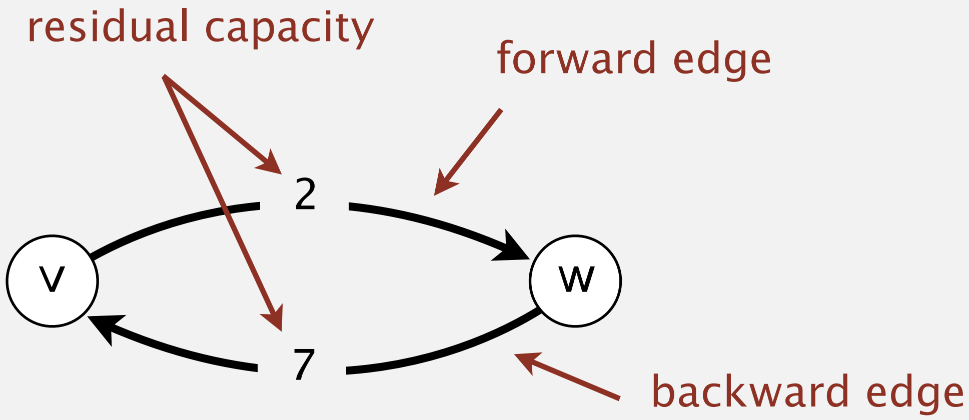
Flow network data type. Must be able to process edge $e = v \to w$ in either direction: include e in adjacency lists of both v and w.

Residual (spare) capacity.

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

Augment flow.

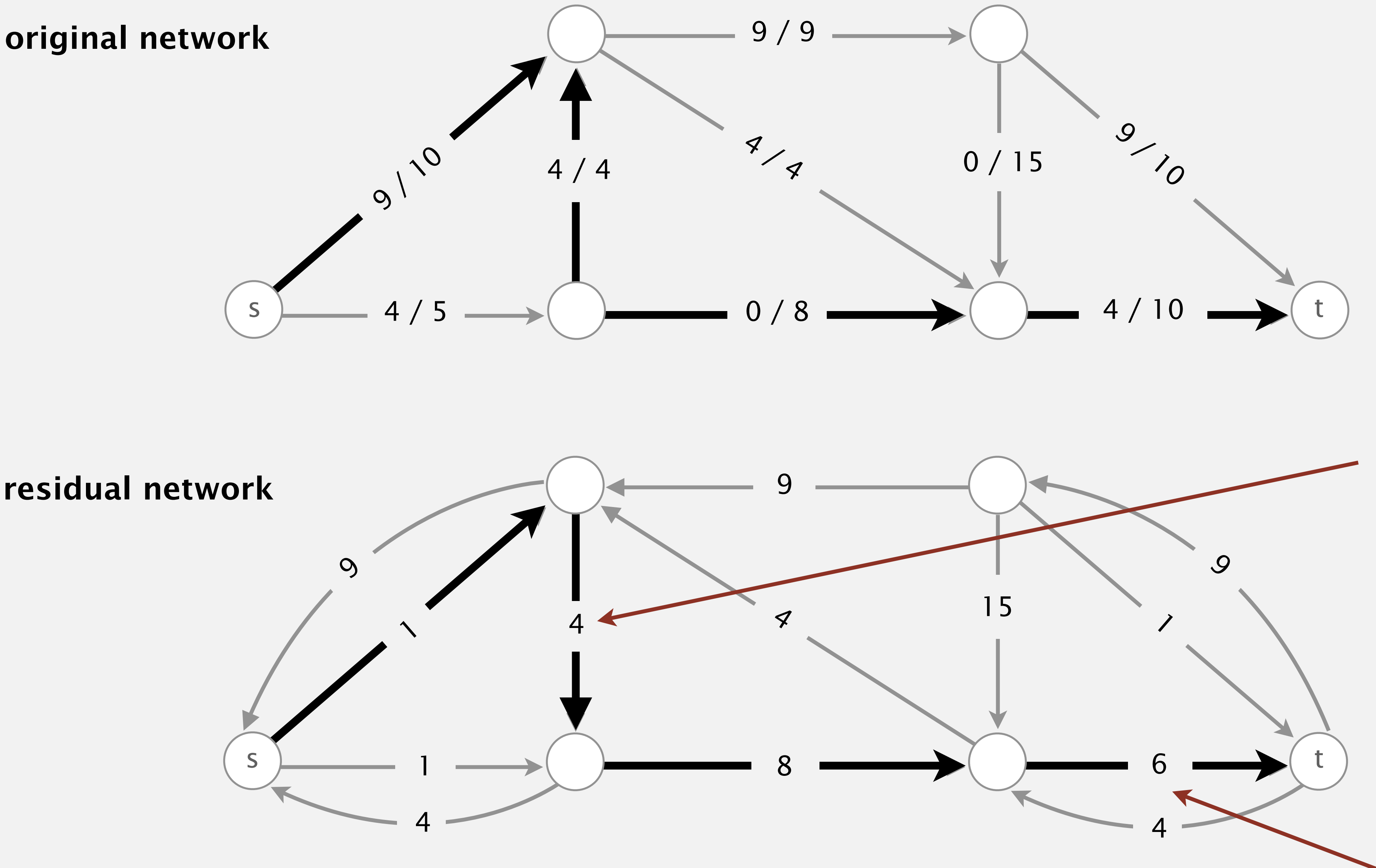
- Forward edge: add Δ.
- Backward edge: subtract Δ.



Flow network representation

Residual network. A useful view of a flow network.

includes all edges with positive residual capacity



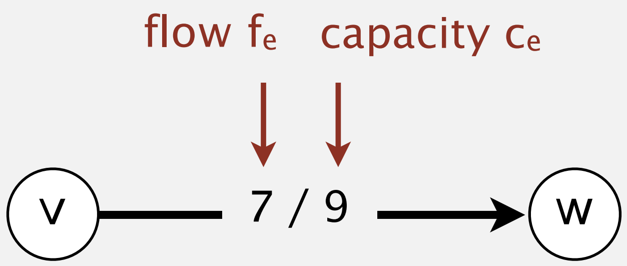
backward edge (not empty)

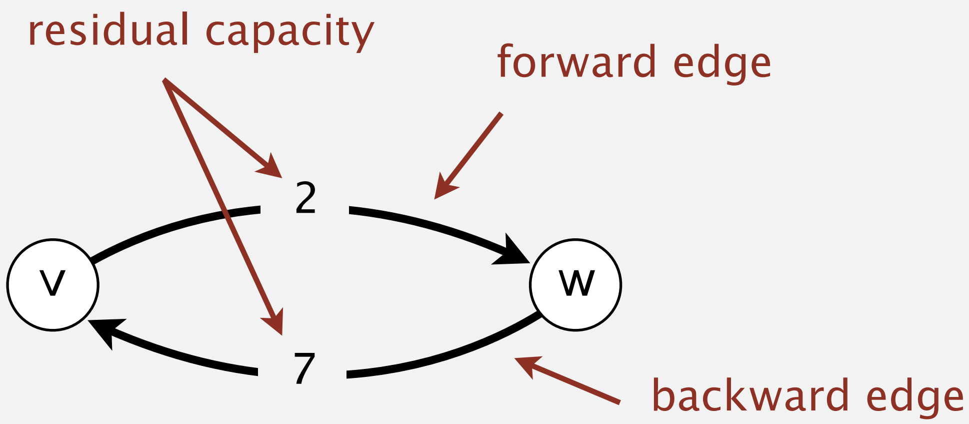
forward edge (not full)

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→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 (mutable)

next slide

Flow edge: Java implementation (continued)

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

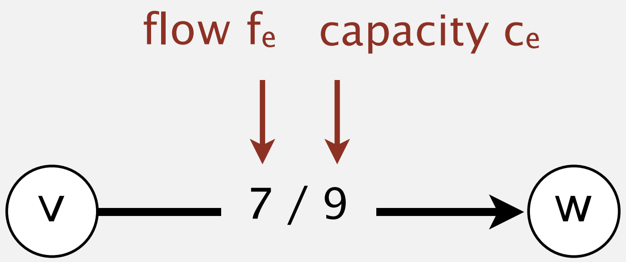
forward edge

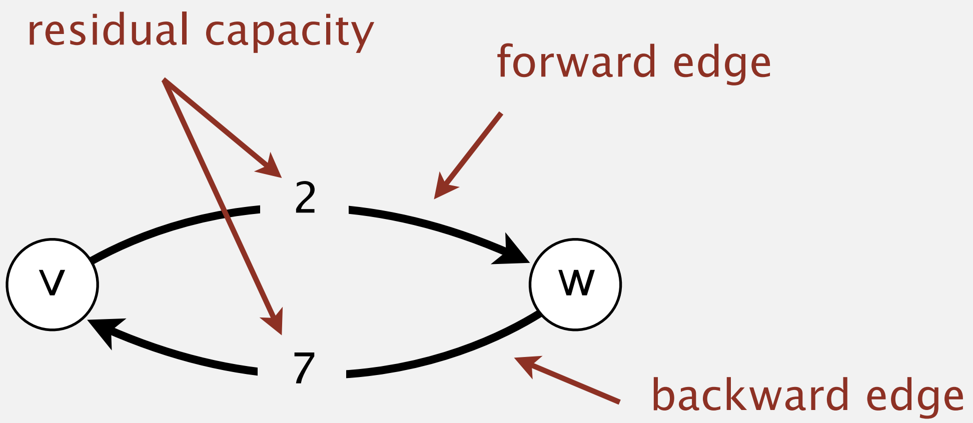
backward edge

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

forward edge

backward edge





Flow network API

	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*

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]; }
}
```

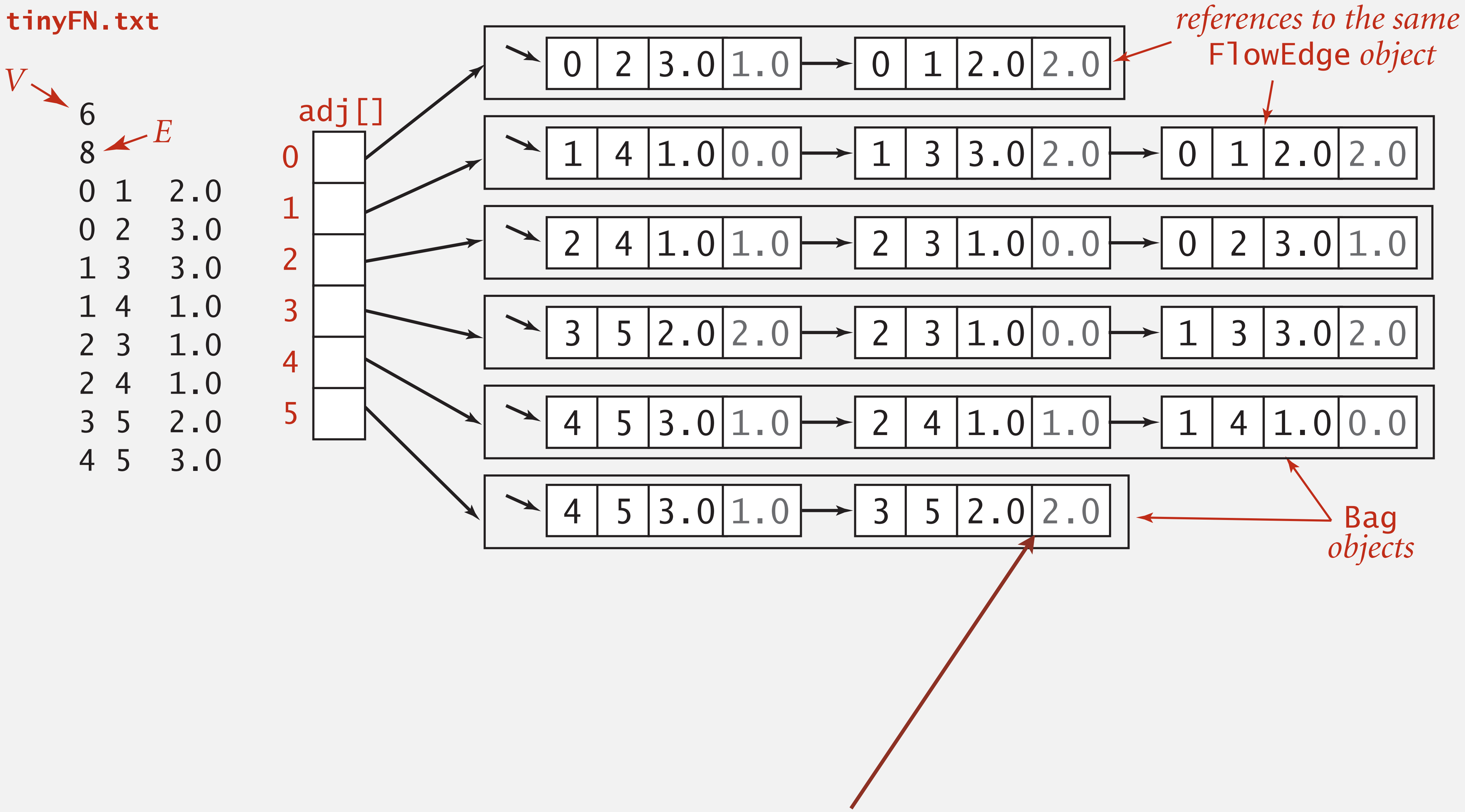
same as EdgeWeightedGraph, but adjacency lists of FlowEdges instead of Edges

add forward edge

add backward edge

Flow network: adjacency-lists representation

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



tinyFN.txt

```
6
8
0 1  2.0
0 2  3.0
1 3  3.0
1 4  1.0
2 3  1.0
2 4  1.0
3 5  2.0
4 5  3.0
```

Note. Adjacency list includes edges with 0 residual capacity.

(residual network is represented implicitly)

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];
}
```

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?

Ford-Fulkerson: Java implementation

```java
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));

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

         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];  }
}
```

compute edgeTo[] and marked[]

compute bottleneck capacity

augment flow

is v reachable from s in residual network?

Algorithms

Robert Sedgewick | Kevin Wayne

http://algs4.cs.princeton.edu

6.4 Maximum Flow

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

Bipartite matching problem

N students apply for N jobs.

Each gets several offers.

Is there a way to match all students to jobs?

bipartite matching problem

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

6 Adobe
    Alice
    Bob
    Carol
7 Amazon
    Alice
    Bob
    Dave
    Eliza
8 Facebook
    Carol
9 Google
    Alice
    Carol
10 Yahoo
    Dave
    Eliza

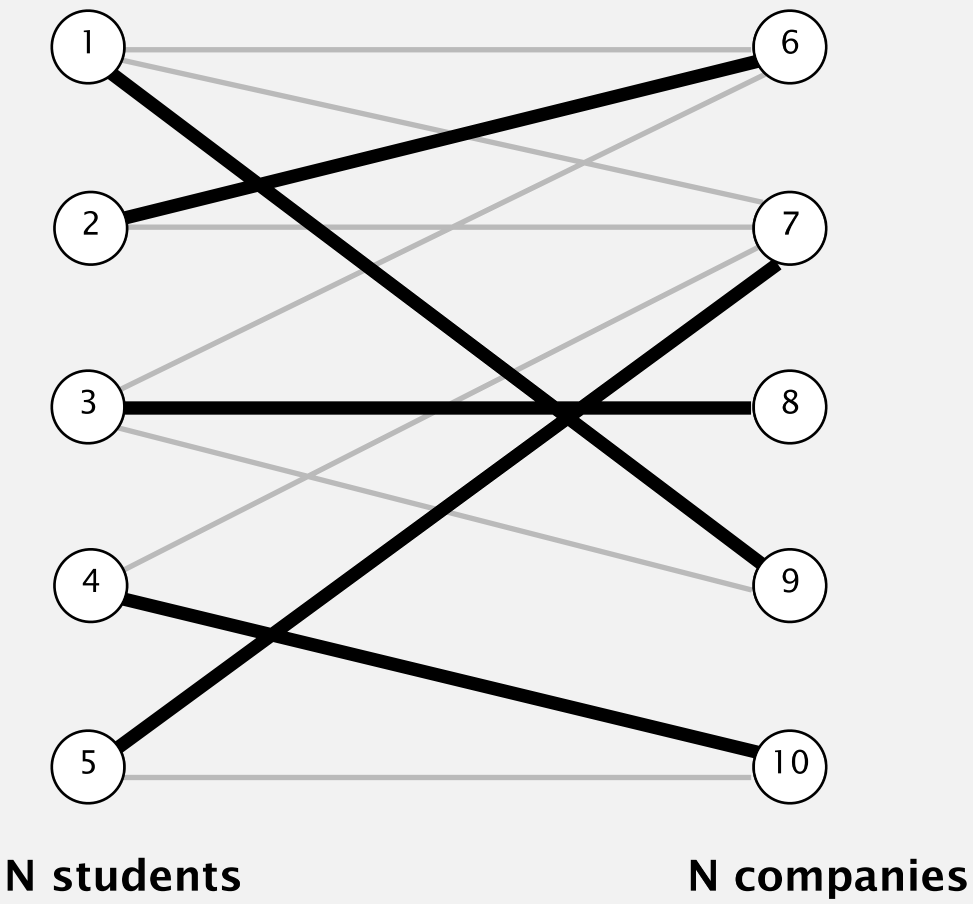
Bipartite matching problem

Given a bipartite graph, find a perfect matching.

perfect matching (solution)

Alice	—	Google
Bob	—	Adobe
Carol	—	Facebook
Dave	—	Yahoo
Eliza	—	Amazon

bipartite graph



bipartite matching problem

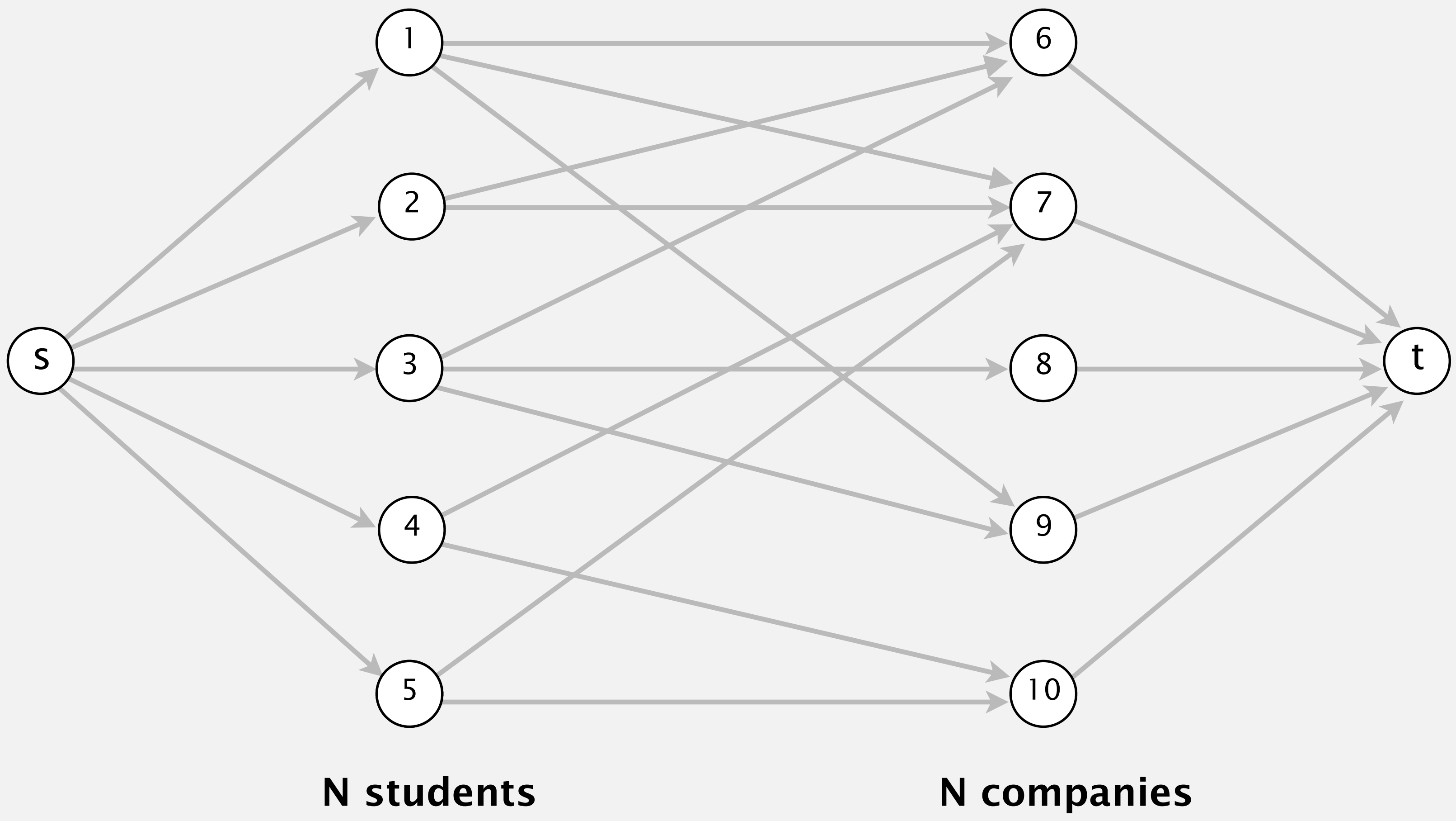
1 Alice
  Adobe
  Amazon
  Google
2 Bob
  Adobe
  Amazon
3 Carol
  Adobe
  Facebook
  Google
4 Dave
  Amazon
  Yahoo
5 Eliza
  Amazon
  Yahoo

6 Adobe
  Alice
  Bob
  Carol
7 Amazon
  Alice
  Bob
  Dave
  Eliza
8 Facebook
  Carol
9 Google
  Alice
  Carol
10 Yahoo
  Dave
  Eliza

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).

flow network



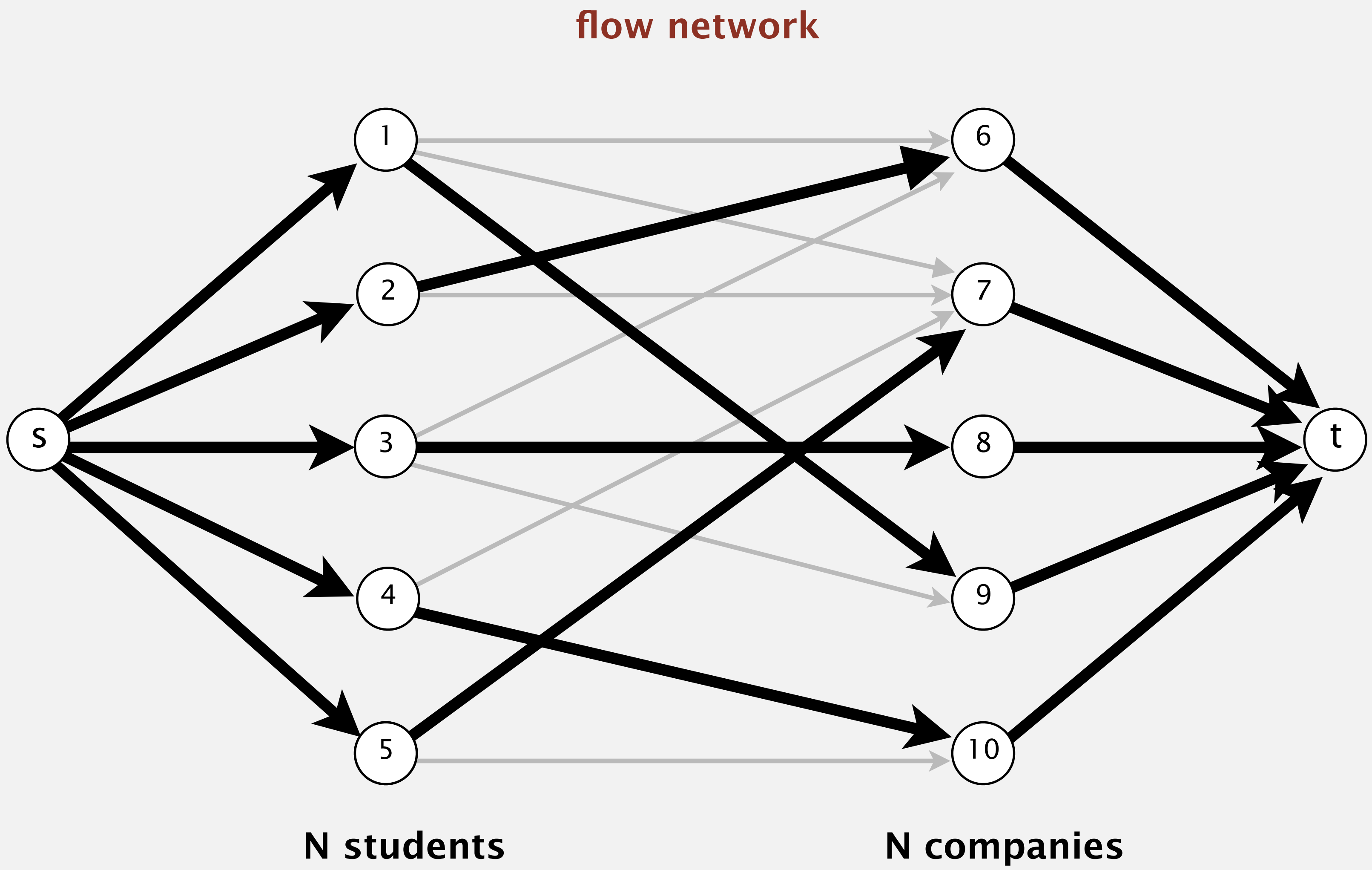
bipartite matching problem

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

6 Adobe
Alice
Bob
Carol
7 Amazon
Alice
Bob
Dave
Eliza
8 Facebook
Carol
9 Google
Alice
Carol
10 Yahoo
Dave
Eliza

Network flow formulation of bipartite matching

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



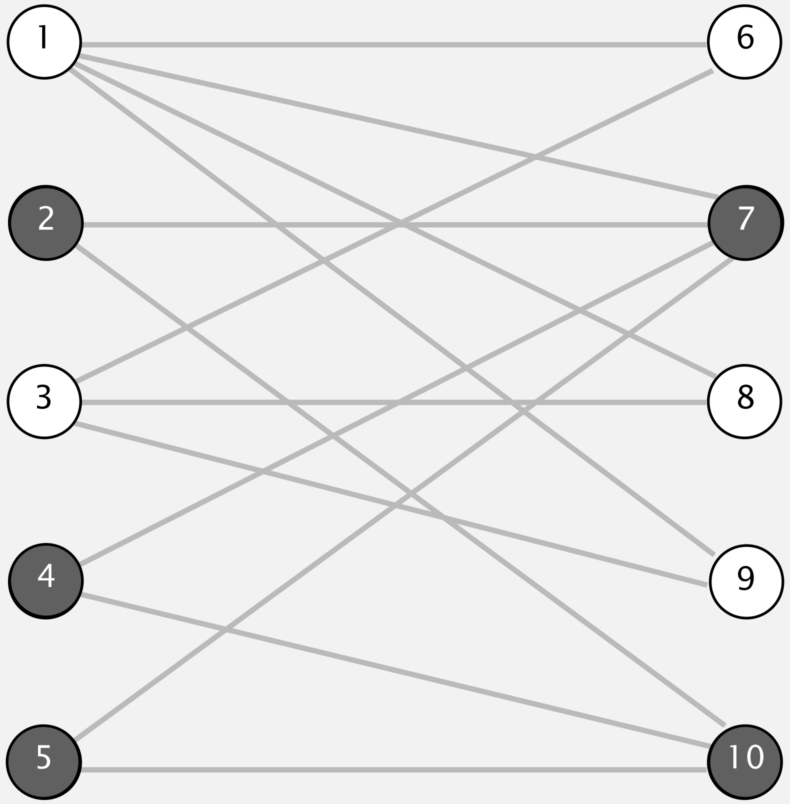
bipartite matching problem

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

6 Adobe
Alice
Bob
Carol
7 Amazon
Alice
Bob
Dave
Eliza
8 Facebook
Carol
9 Google
Alice
Carol
10 Yahoo
Dave
Eliza

What the mincut tells us

Goal. When no perfect matching, explain why.



no perfect matching exists

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

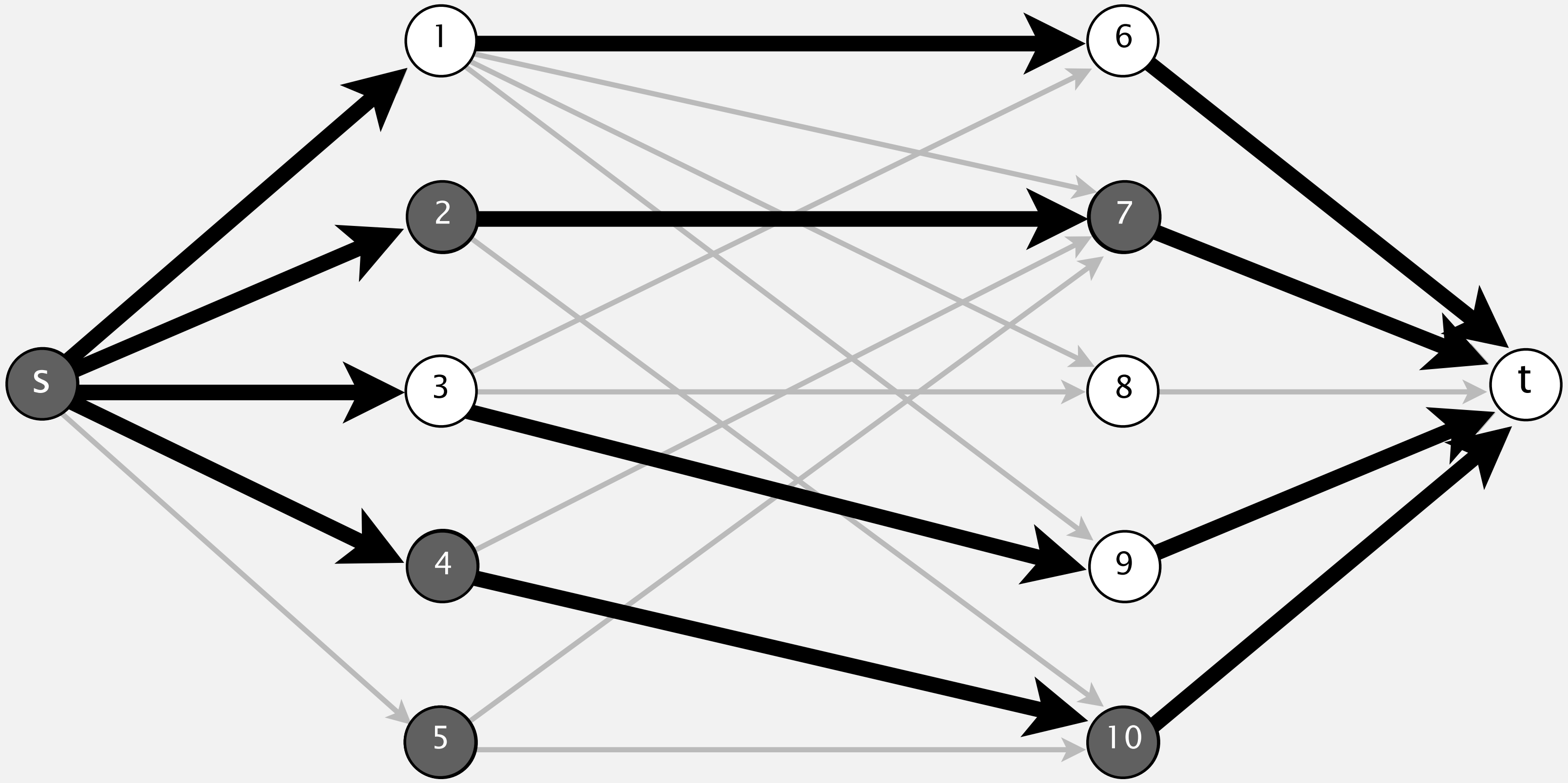
student in S
can be matched
only to
companies in T

| S | > | T |

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.



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

student in S can be matched only to companies in T

| S | > | T |

no perfect matching exists

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

Summary

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!