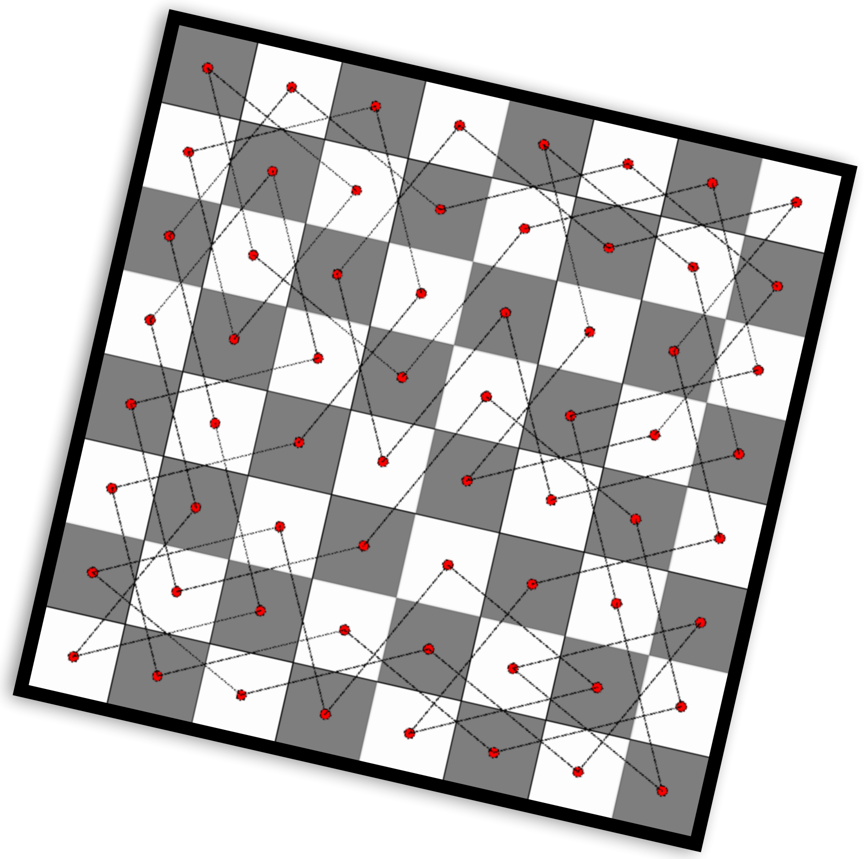
Network Flows*

Pierre Schaus



*Many slides and figures of this presentation are coming from Claude Guy Quimper and are used with his agreement for INGI2266

Outline

- Maximum Flows
  - Augmenting paths
  - Ford-Fulkerson algo
  - Scaling algo
  - Max-Flow Min Cut
- Min Cost Flow
  - Iterative Shortest Paths
- Flows and Linear Programming

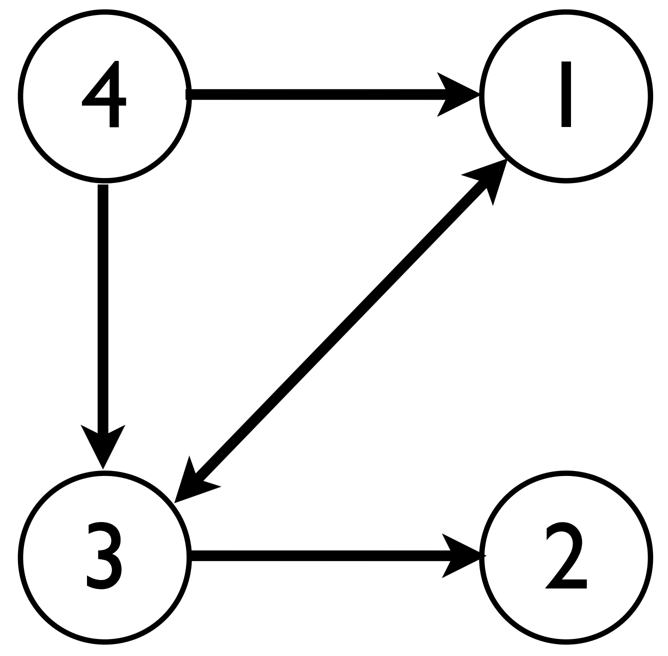
Introduction

- Max Flow is a combinatorial problem on Graphs.
- Many problems can be solved with Max Flows on a graph: transports, distributions, network designs
- Max Flow is a very useful building block occurring in many context of optimization (generation of cut for MIP, filtering algorithm for global constraints, computation of lower bounds, etc).

Reminder: Directed Graph

- A directed graph is tuple (V, E) where V is the set of vertices (also called nodes) and E ⊆ V x V is the set of edges (also called arcs).
- A pair (a, b) ∈ E represents a directed edge from node a to b.

Example of directed graph

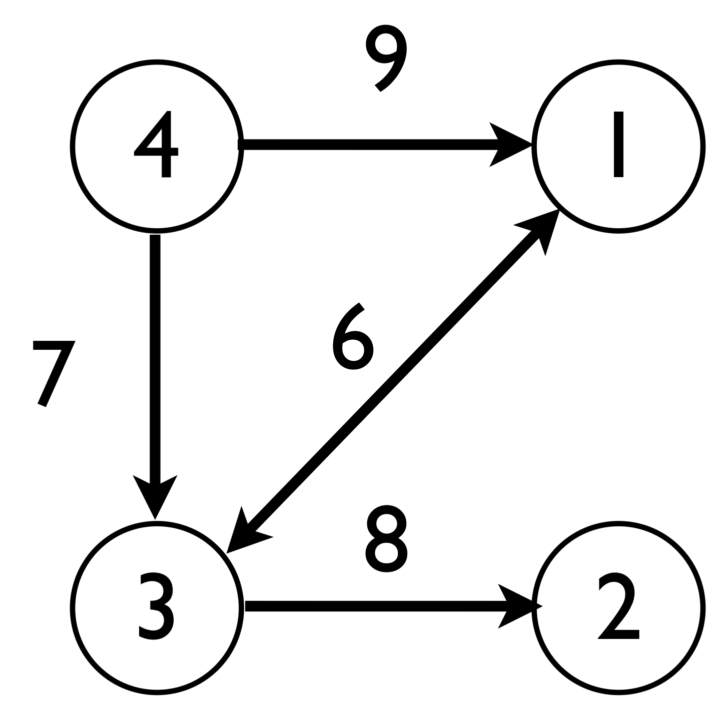


$V = \{1, 2, 3, 4\}$
$E = \{(1, 3), (3, 1), (3, 2), (4, 1), (4, 3)\}$

Weighted directed graph

- A node *a* is **adjacent** to node *b* if there exists an edge (a, b) ∈ E.
- Some graphs have weighted edges given by a function *f*(*a*, *b*) taking two nodes as input and returning a number (integer or real depending on the application).
- A weight can represent the distance, the cost, the capacity, etc.

Exemple de fonction de poids



a	b	f(a,b)
1	3	6
3	1	6
3	2	8
4	1	9
4	3	7

Path and cycles

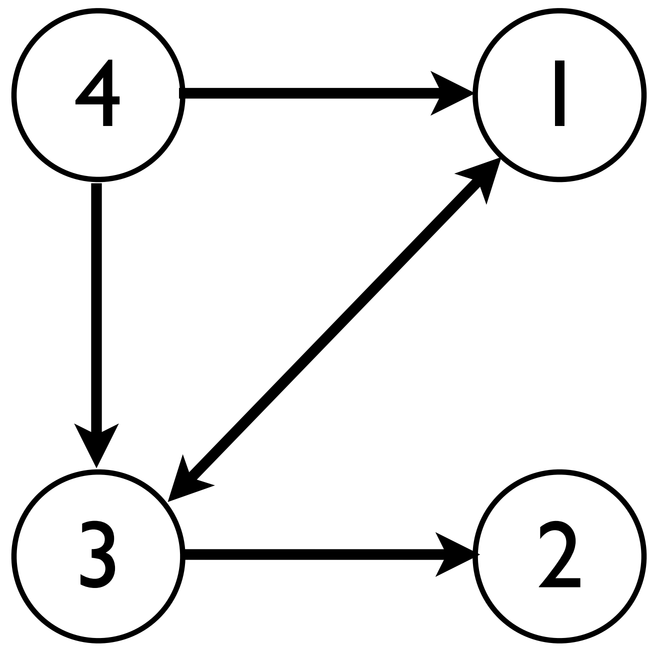
- A **path** is a suite of distinct nodes $n_0, n_1, \dots n_{k-1}$ with (n_i, n_{i+1}) an edge for all $0 \leqslant i < k-1$. Node n_0 is the **origin** and node n_{k-1} is the **destination**.
- Un **cycle** is suite of distinct nodes $n_0, n_1, \dots, n_{k-1}$ with $(n_i, n_{i+1 \bmod k})$ an edge for all $0 \leqslant i < k$.
- Sometimes, paths and cycles are identified with the sequence of edges instead of the sequence of nodes.

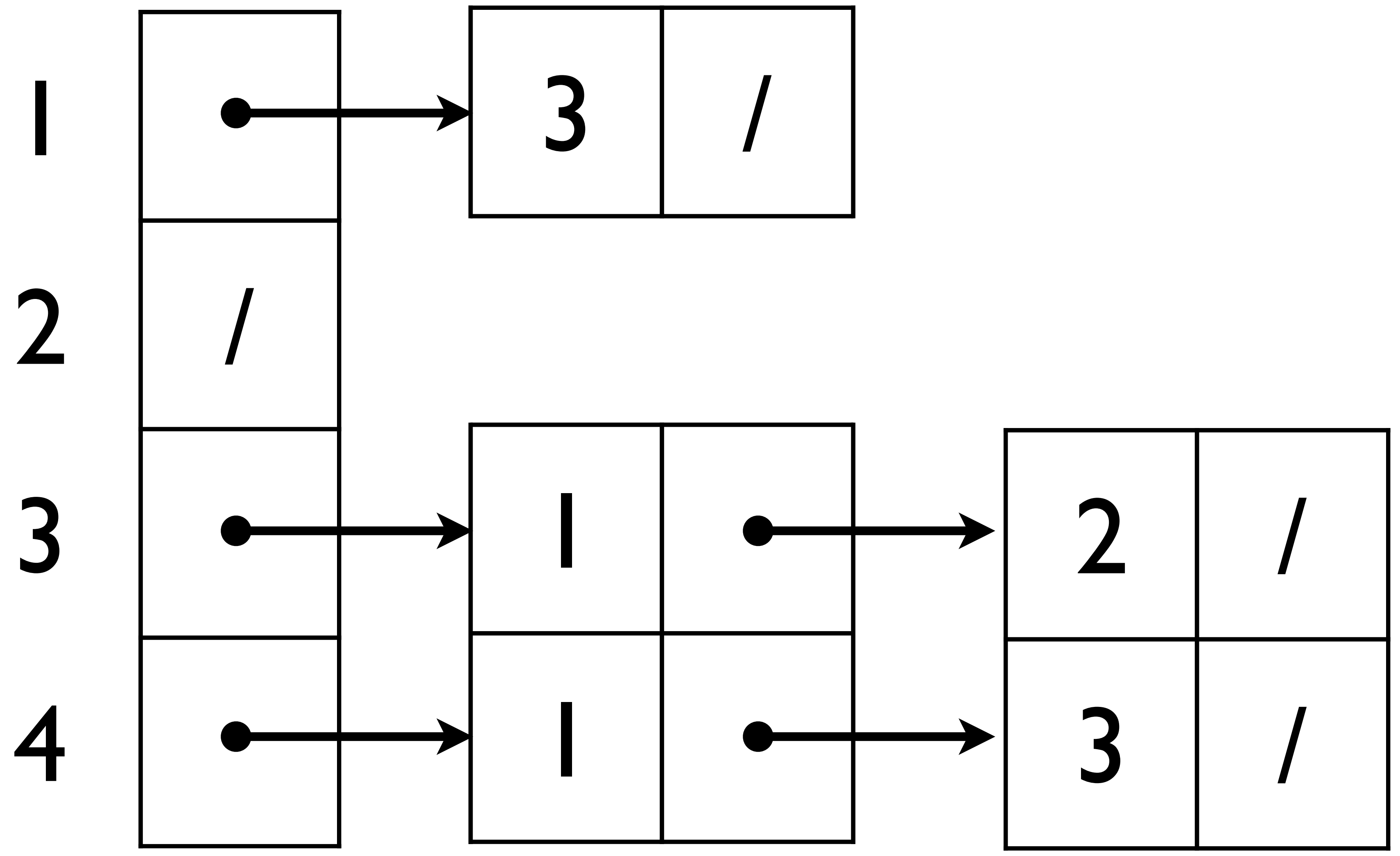
ADT for graphs: Adjacency Matrix

- There are several possible data structures to encode a graph.
- The simplest one is the adjacency matrix M où M[a,b] = 1 if (a,b) is an edge M[a,b] = 0 if (a, b) is not an edge.
- Drawback: Does not take advantage of sparsity, linear time to iterate over adjacent nodes.

ADT for graphs: Adjacency Lists

- We can use adjacency lists
- For each node, an array Adj[1..n] with one entry per node.
- An entry A[i] is a pointer to the beginning of a linked list with adjacent (out) nodes to node i.
- This list in practice can be double linked for more efficient update operations on the graph.





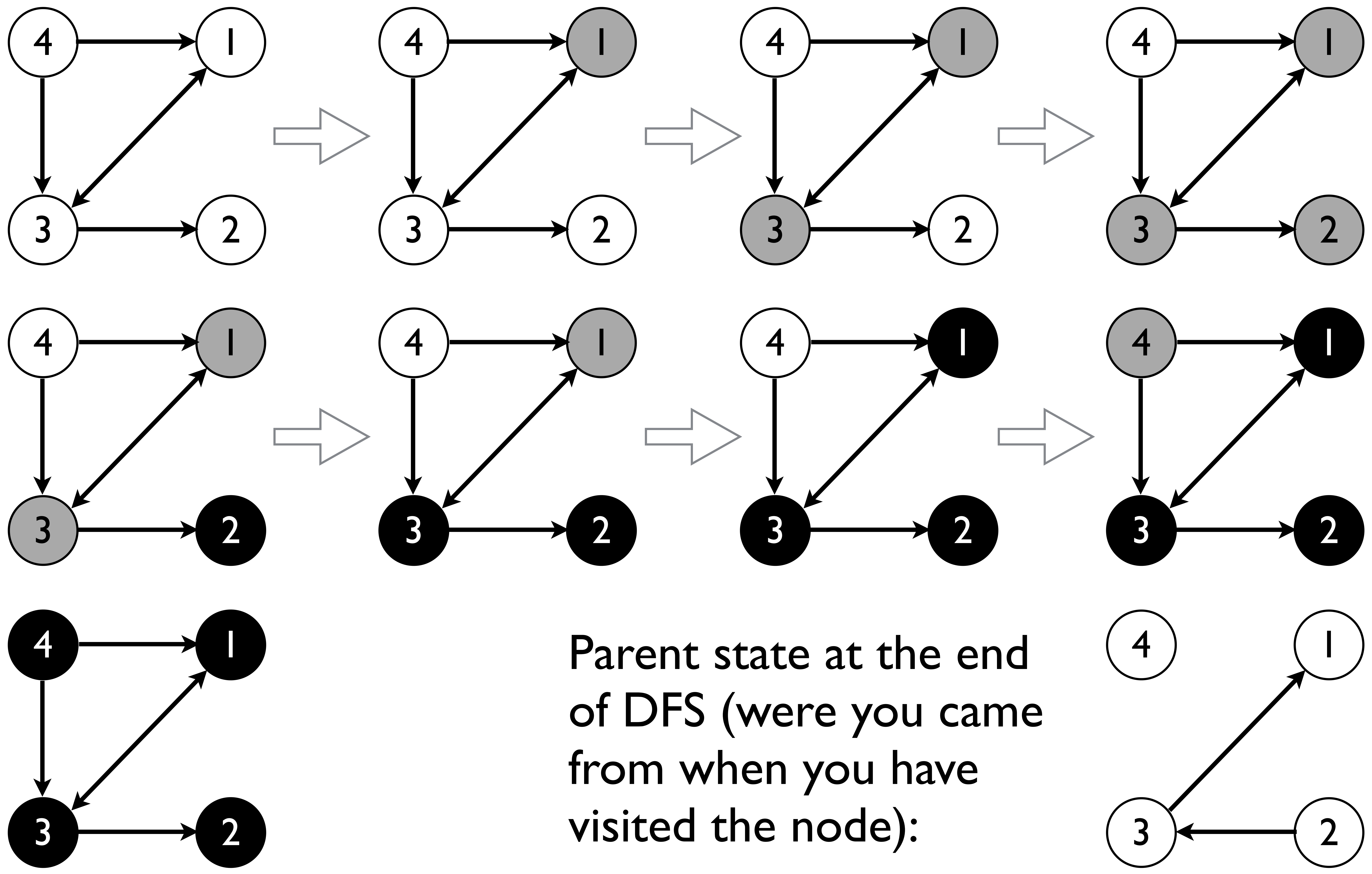
DFS on graph

Depth-first search (**DFS**) is an algorithm for traversing a graph data structures:

- One starts at the root (selecting some arbitrary node as the root in the case of a graph) and explores as far as possible along each branch before backtracking.

Color	Meaning
White	The node is not yet visited
Grey	The node has been visited but all its adjacent edges are not yet visited.
Black	The node is visited and all its adjacent edges have been processed.

DFS Example



Parent state at the end of DFS (were you came from when you have visited the node):

DFS Parent Code

- The Parent vector encodes a forest, i.e. a collection of trees.
- Each node of the graph is part of the forest.
- An edge (v, u) is part of the foret if the node v was visited during the processing of edge (u,v).
- Then u is called the parent of v.
- The forest is encoded using an Array representation Parent where Parent[v] = u.

Algorithm 1: DFS(V, E)

```
// Initialization;
Create global variable $Color[1..|V|]$;
Create global variable $Parent[1..|V|]$;
for each node $u \in V$ do
    $Color[u] \leftarrow white$;
    $Parent[u] \leftarrow Nul$;
// Start the search;
for each node $u \in V$ do
    if $Color[u] = white$ then
        Visit($u, V, E$);
```

Algorithm 2: Visit(u, V, E)

```
$Color[u] \leftarrow Grey$;
for $v \in Adj[u]$ do
    // Explore arc $(u, v)$;
    if $Color[v] = white$ then
        $Parent[v] \leftarrow u$;
        Visit($v, V, E$);
$Color[u] \leftarrow black$;
```

- Algorithm *DFS* initializes the vector of colors and parents then starts the visit for each node
- Algorithm *Visit* visits a node of the graph and every edge adjacent to this node. If node *u* is adjacent to a node *v* not yet visited, then this node is also visited.

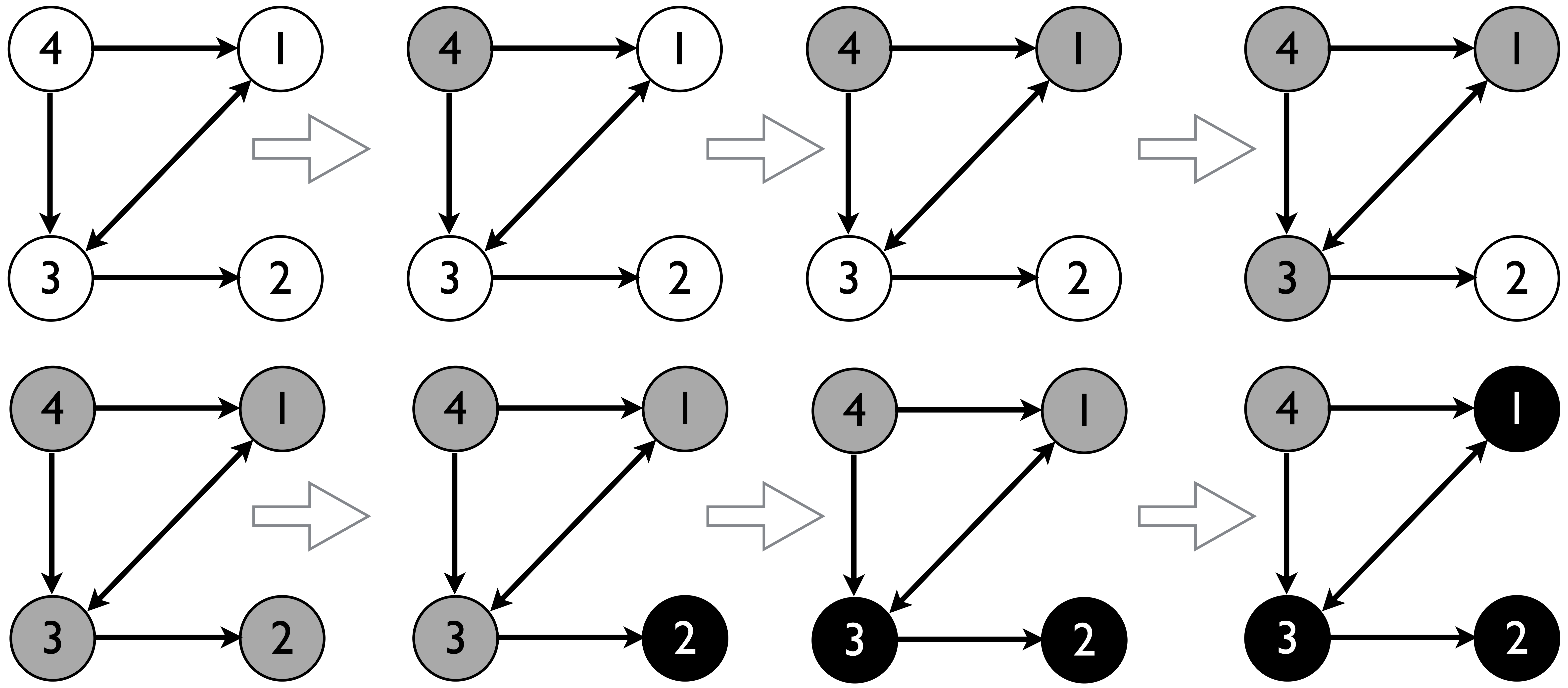
DFS Time Comlexity

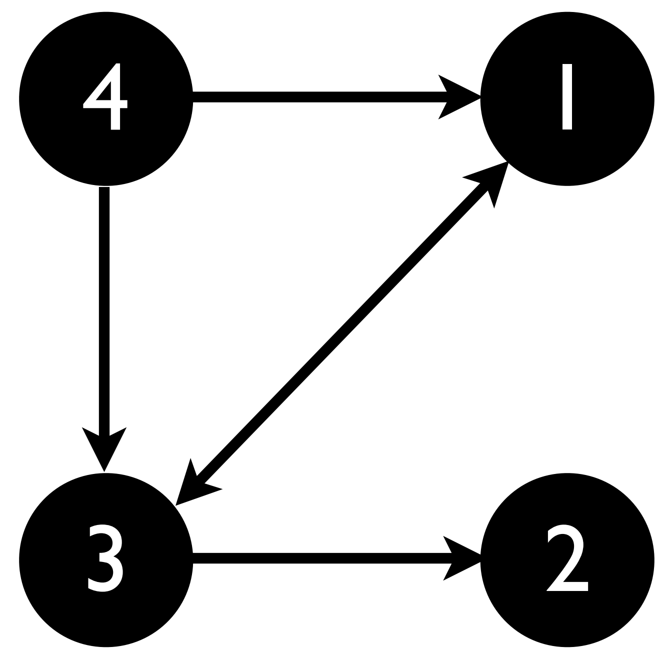
- Initialization in O(|V|)
- Exactly |V| calls to procedure Visit.
- Each call to procedure Visit treats each adjacent edge to node *u* only once. Since Visit is called only once per node, each edge is treated only once.
- The time complexity is thus O(|V| + |E|).

Find a Path in Directed Graph

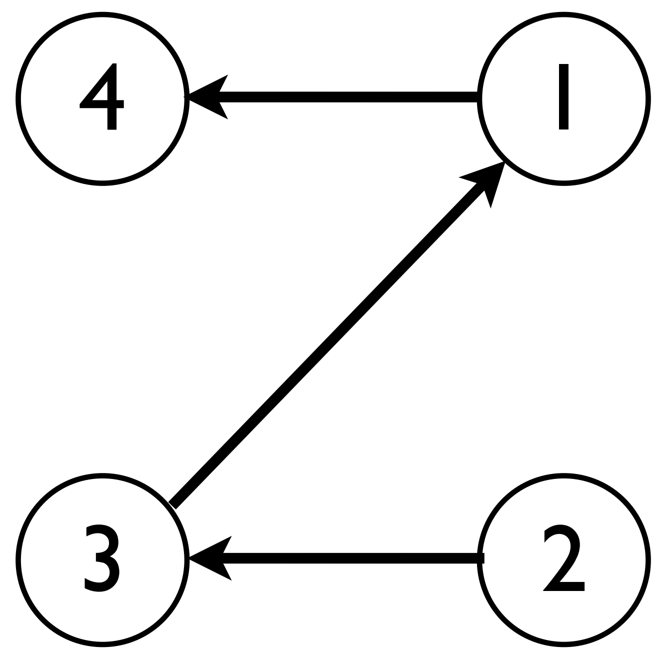
- **Problem:** Find a path from origin *s* to destination *t*.
- **Solution:** Adapt DFS to this end
  - Instead of calling Visit on every node, just call it on origin node *s*.
  - Nodes t, Parent[t], Parent[Parent[t]], ..., s is the reverse path from origin *s* to destination *t*.
  - If Parent[t] is nul, then there is no path from *s* to *t*.

Example: Find a path from 4 to 3





State of vector Parent at the end of the (modified) DFS

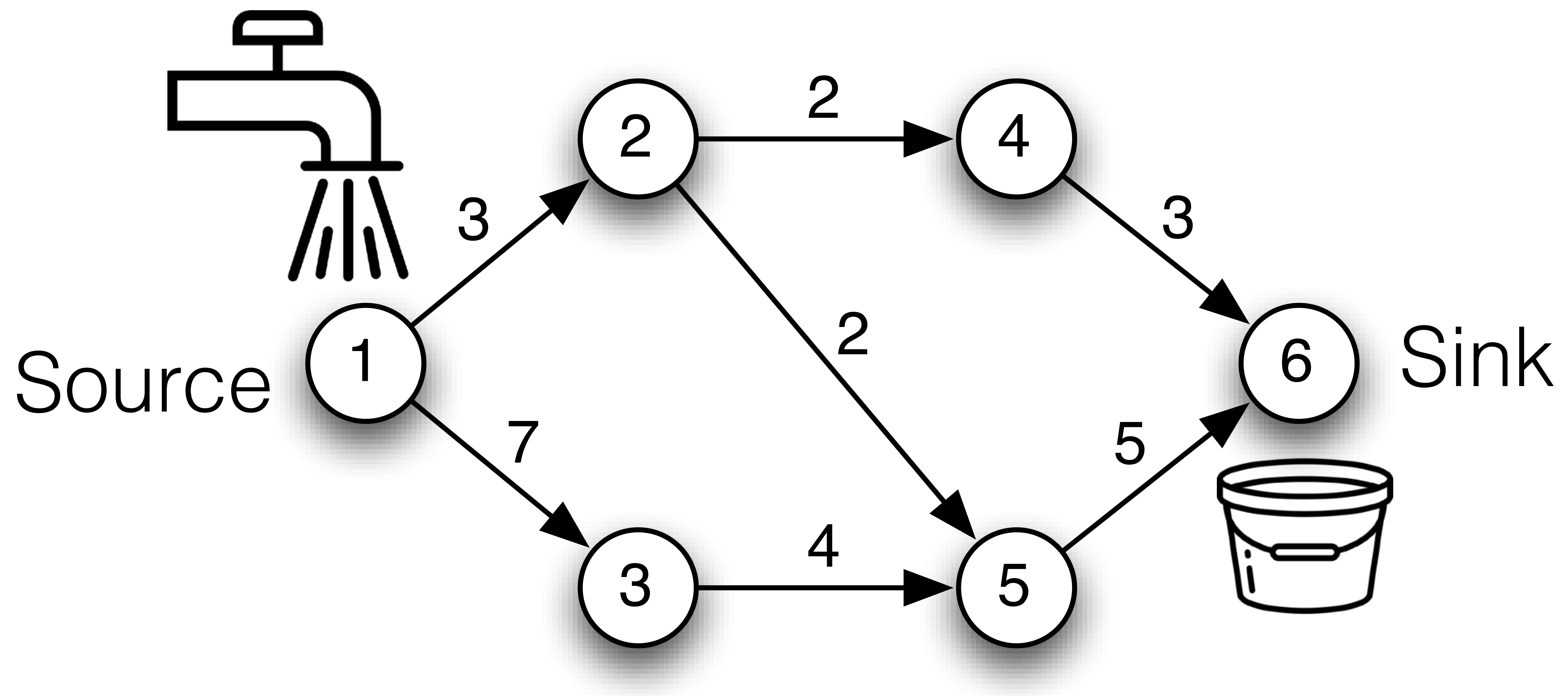


Observation

- This algo finds a path from origin *s* to all other nodes reachable from s (and not only toward destination *t*).
- And the representation of all these paths is O(|V|) (the vector Parent)

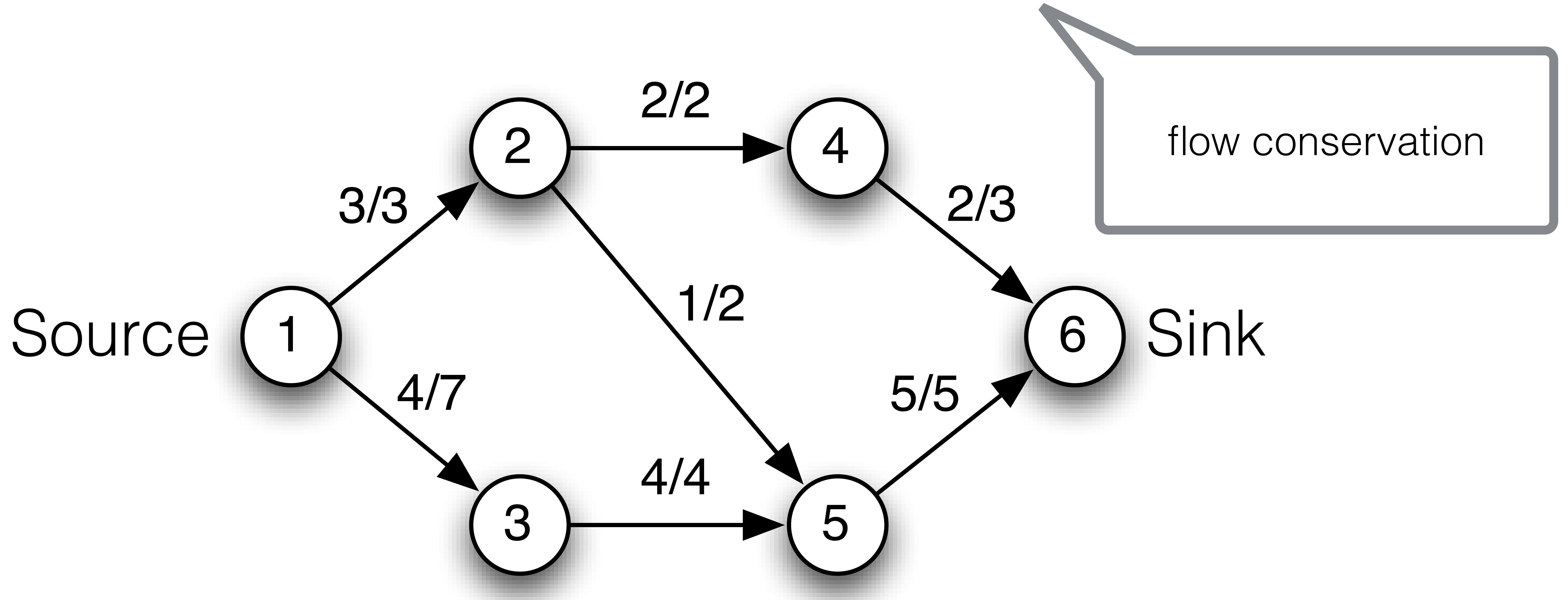
Max Flow

- Consider following graph and imagine that the edges are pipes and nodes are junctions between pipes. Each pipe has a capacity expressed in number of litres/minute.
- **Question:** How much litres/minutes can flow at most between node 1 and node 6 ?



Max Flow

- **Answer:** 7 litres / minutes
- Solution shown on graph: for each edge (flow/capacity)
- Note that nothing is created or lost at intermediate nodes: Input flow = Output Flow (like Kirchhoff law in electronic circuits)



Max Flow: Motivation

- Max Flow problem can be solved in polynomial time.
- We will show an algorithm to solve this problem.
- Max Flow can be used to model many different complex problems (so not only water flows in pipes, think about computer networks).
- Max Flow can be used as a subroutine in NP Hard problems.

Problem Definition

- We assume edges are unidirectional: (a,b)∈E => (b,a)∉E
- Node *s* provides water and is called the the **source**.
- Node *t* absorbs water and is called the **sink**.
- The **capacity** between two nodes *a* and *b* is denoted as *c(a,b)*. This value is positive if *(a, b)* is an edge of the graph, *0* if *(a, b)* is not an edge.
- A **flow** is a vector *f* such that each composant is associated to a pair of nodes. We denote by *f(a,b)* the flow quantity between nodes *a* and *b*. We have following relation:

$$f(a,b) = -f(b,a)$$

- An instance of the Max Flow problem is characterized by the graph G = (V, E), the source *s*, the sink *t* and the vector of capacity *c*.

Problem Definition

- The **flow conservation constraint** requires that the quantity of water entering into a node (different from source and sink) is exactly the same as the quantity of of water exiting this node.

$$\sum_{b|(b,a)\in E} f(b,a) = \sum_{b|(a,b)\in E} f(a,b) \qquad \forall a \in V \setminus \{s,t\}$$

- Or equivalently we have.

$$\sum_{b\in V} f(a,b) = 0 \qquad \forall a \in V \setminus \{s,t\}$$

- The **capacity constraint** requires that the flow through each edge does not exceed the capacity of this edge

$$f(a,b) \leq c(a,b) \qquad \forall (a,b) \in E$$
$$f(a,b) \leq 0 \qquad \forall (a,b) \notin E$$

Problem Definition

- A **valid flow** is a flow that satisfies the conservation constraint and the capacity constraint.
- A **null flow** is a valid flow where *f(a,b) = 0* for each edge.
- The **value of a valid flow** is the water quantity out of the source. Because of the conservation flow constraint, this value is also equal to the quantity entering the sink.

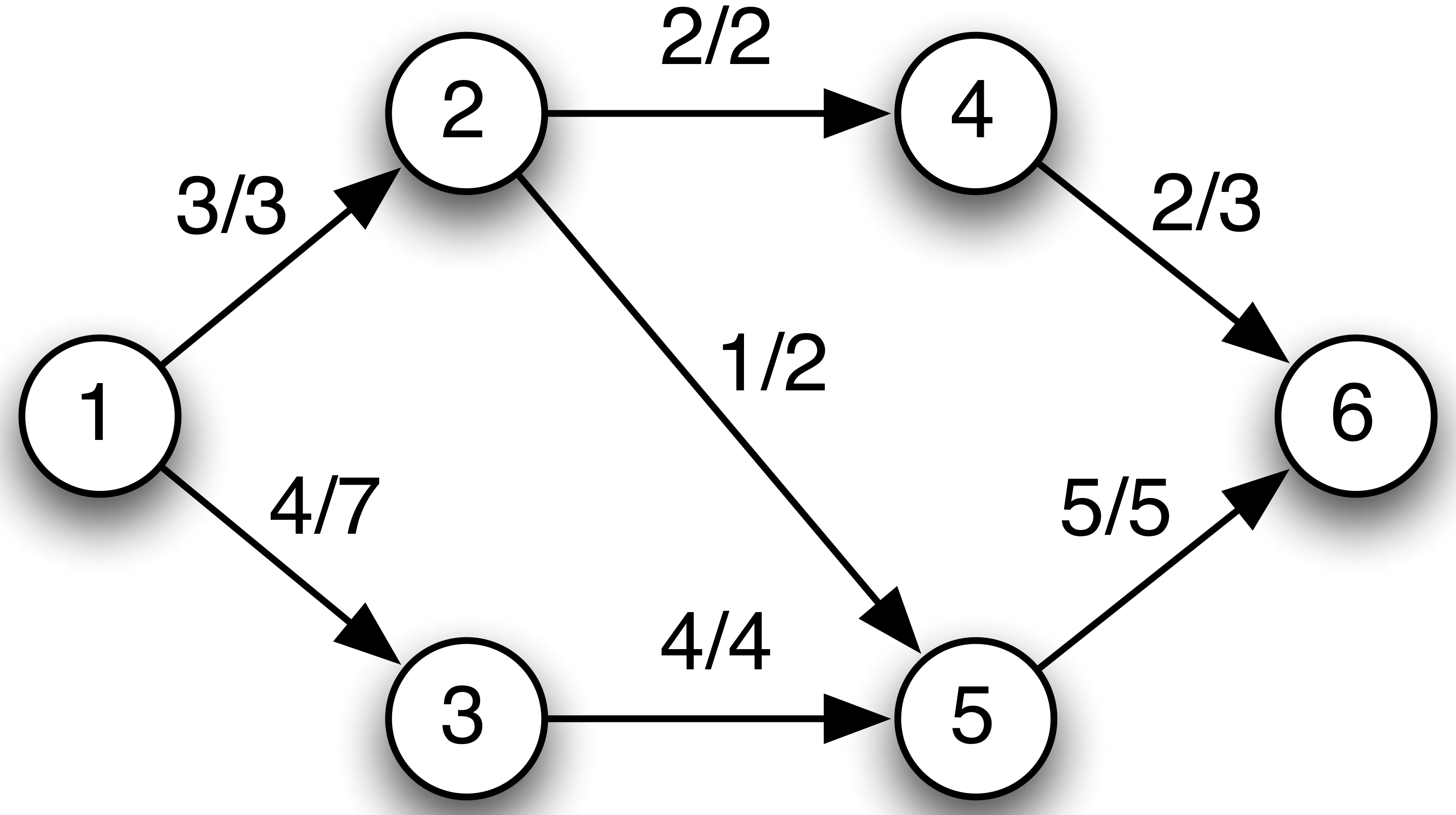
$$v(f) = \sum_{a \in V | (s,a) \in E} f(s,a) = \sum_{a \in V | (a,t) \in E} f(a,t)$$

- The **Max Flow problem** is to to discover a valid flow of maximal value.

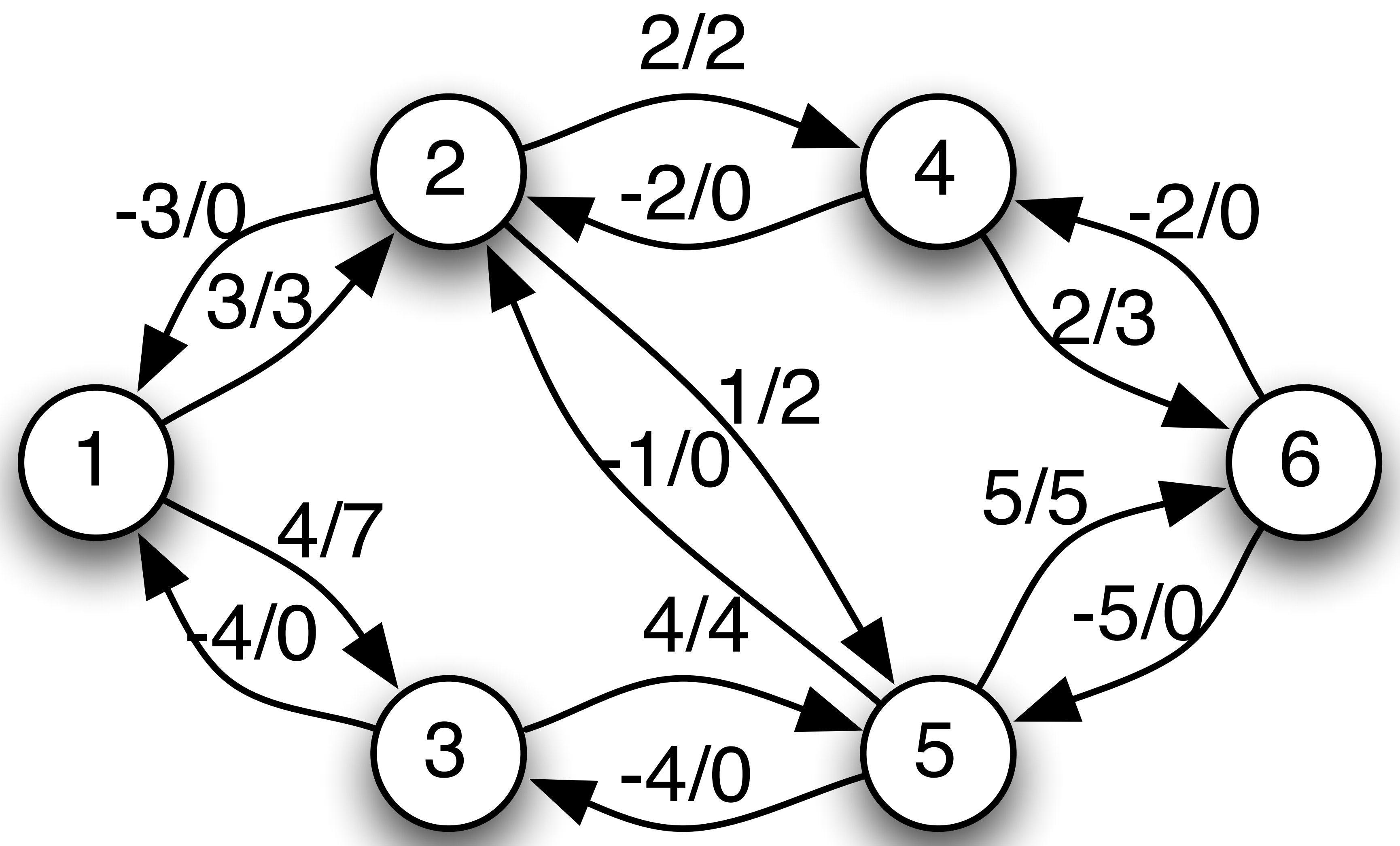
$$v(f) = \sum_{a \in V | (s,a) \in E} f(s,a) = \sum_{a \in V | (a,t) \in E} f(a,t)$$

Graphical Representation

- We usually only represent edges with positive capacity and non negative flows. Hence these two representations are equivalent.



For now, we assume only one directional edge in the input graph



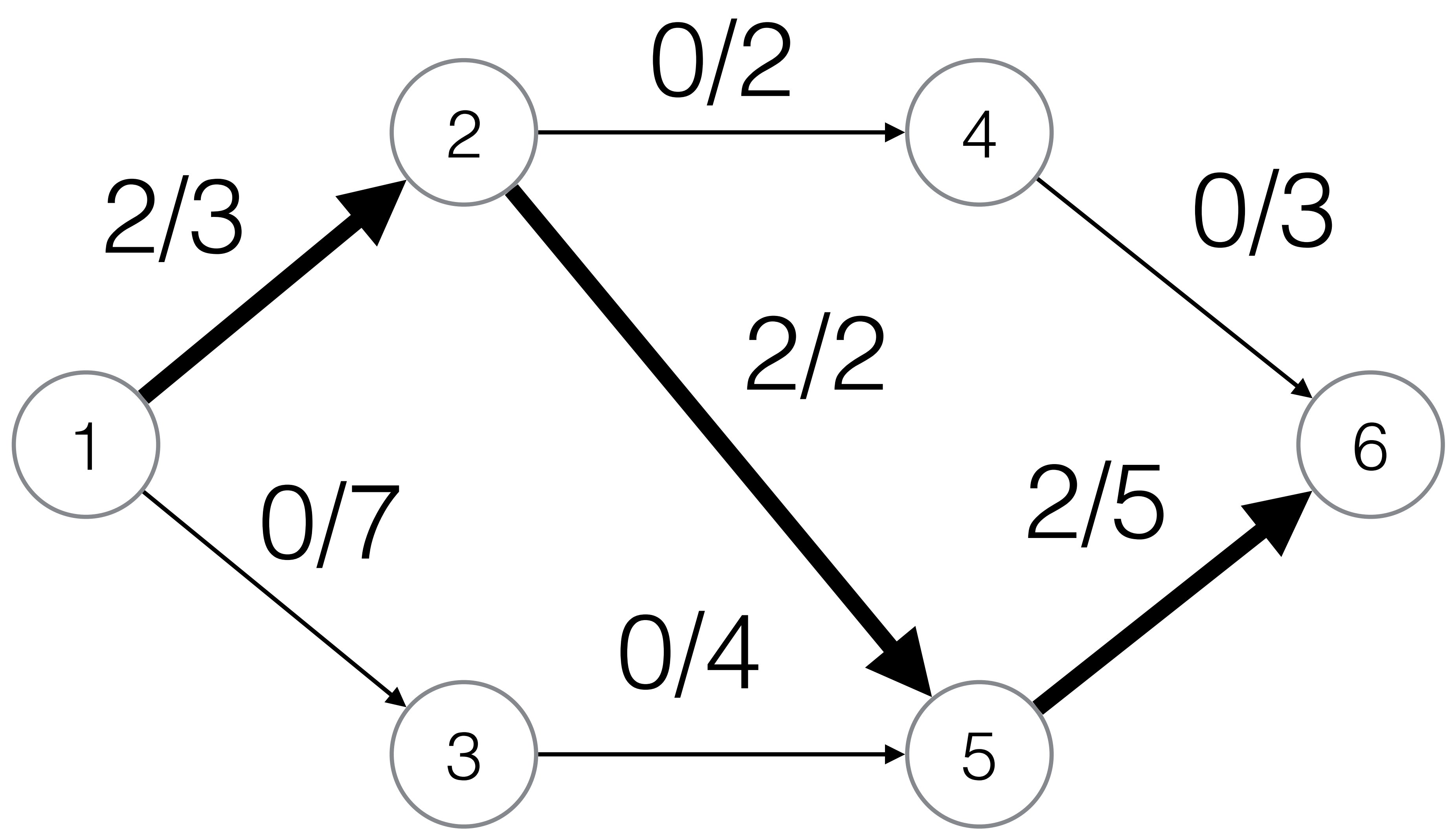
Implicitly add the reverse edges with capacity 0

Let us try a simple algorithm

- As long as I can find a path P with residual capacity in G, augment as much as possible the flow on P

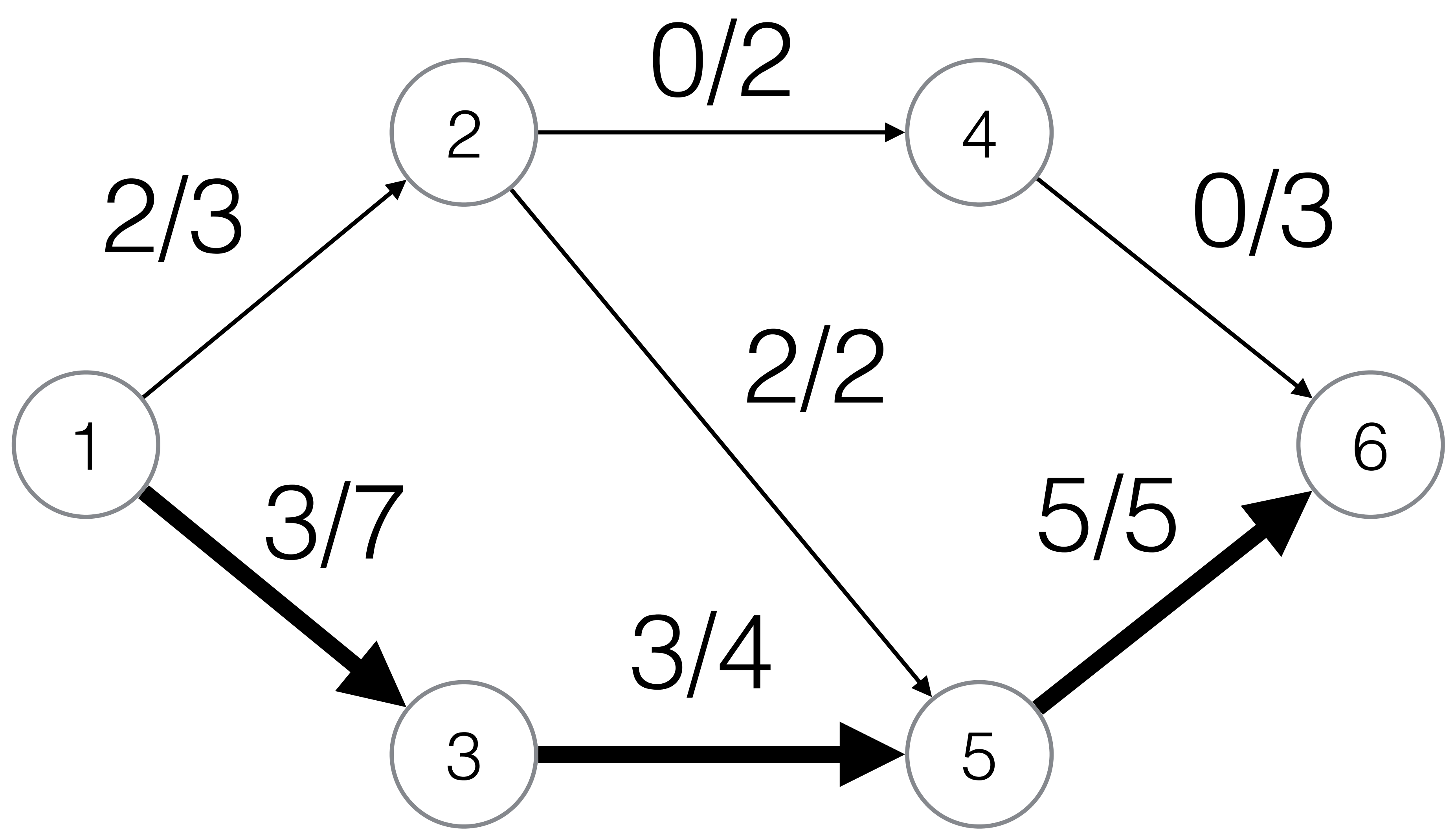
Intuition

- By how much can I increase at most the flow along this path?



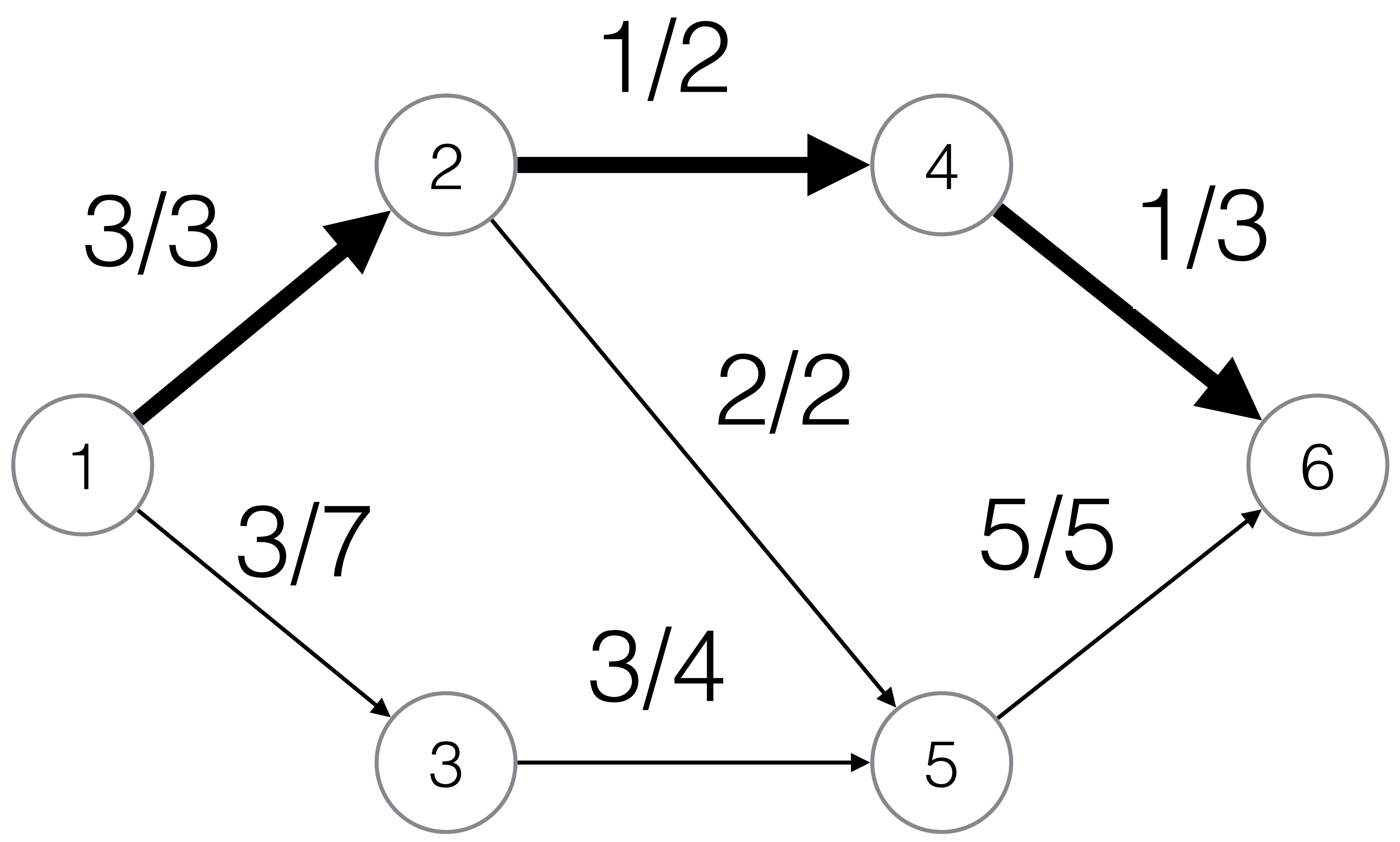
Intuition

- By how much can I increase at most the flow along this path?



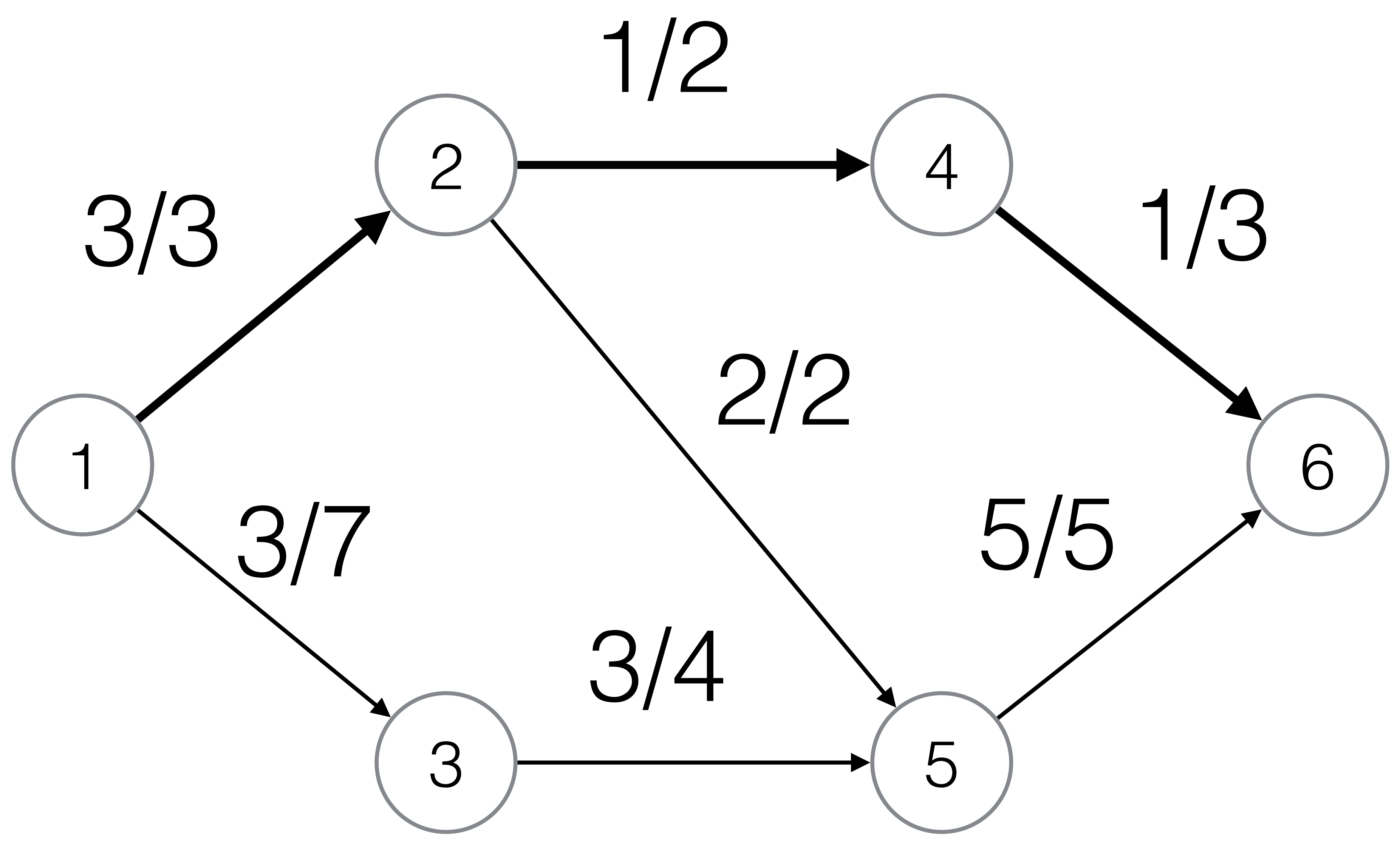
Intuition

- By how much can I increase at most the flow along this path?



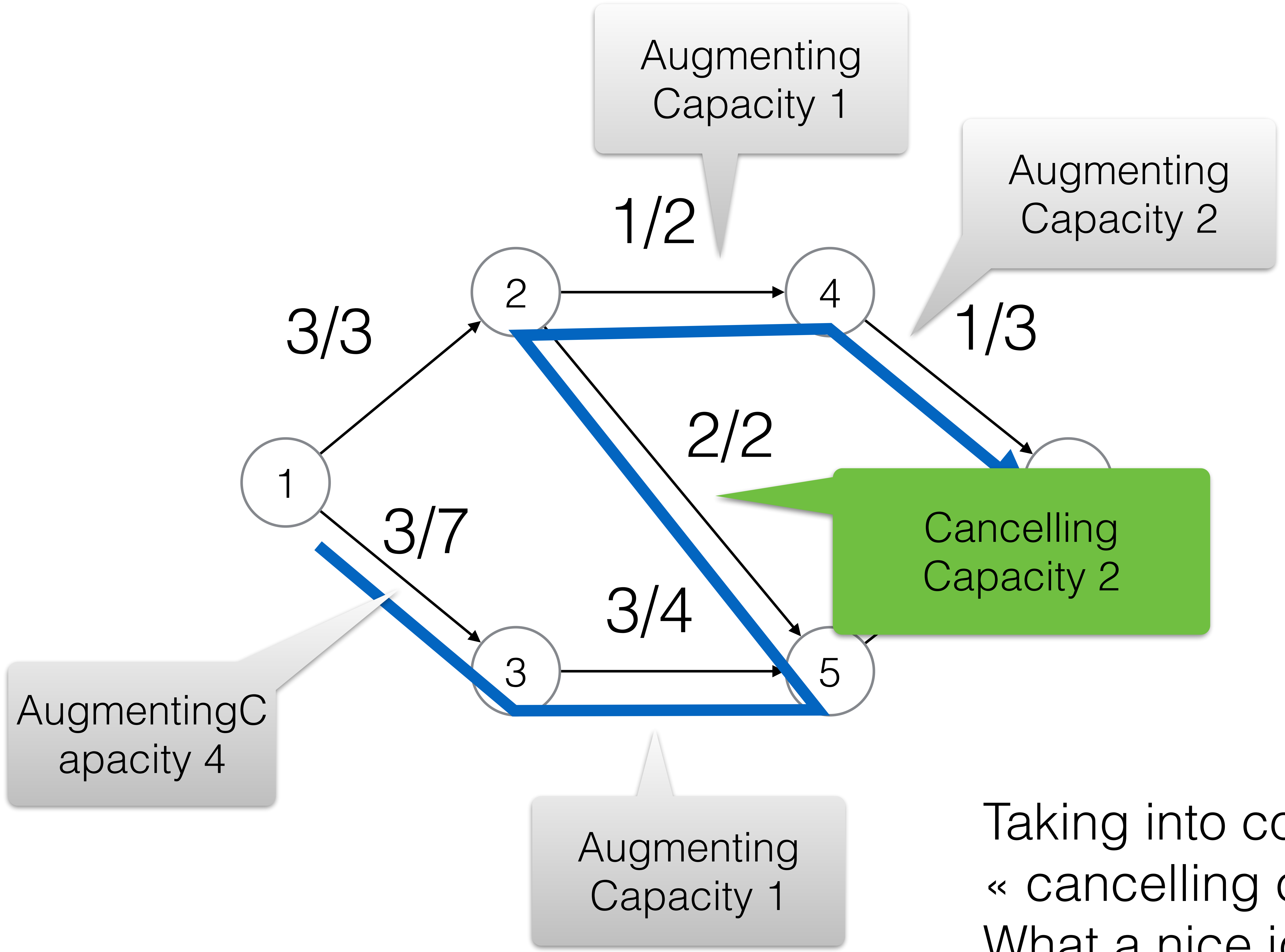
Intuition

- There is no path any more on G with residual capacity.
- My flow value is 6. Is this optimal?



Intuition

- The blue path is interesting!



Taking into consideration the « cancelling capacity »
What a nice idea!

Residual Graph

- Residual graph $G_f = (V, E_f)$ is used to discover paths from the source to the sink on which it is possible to « push » an additional quantity of water 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

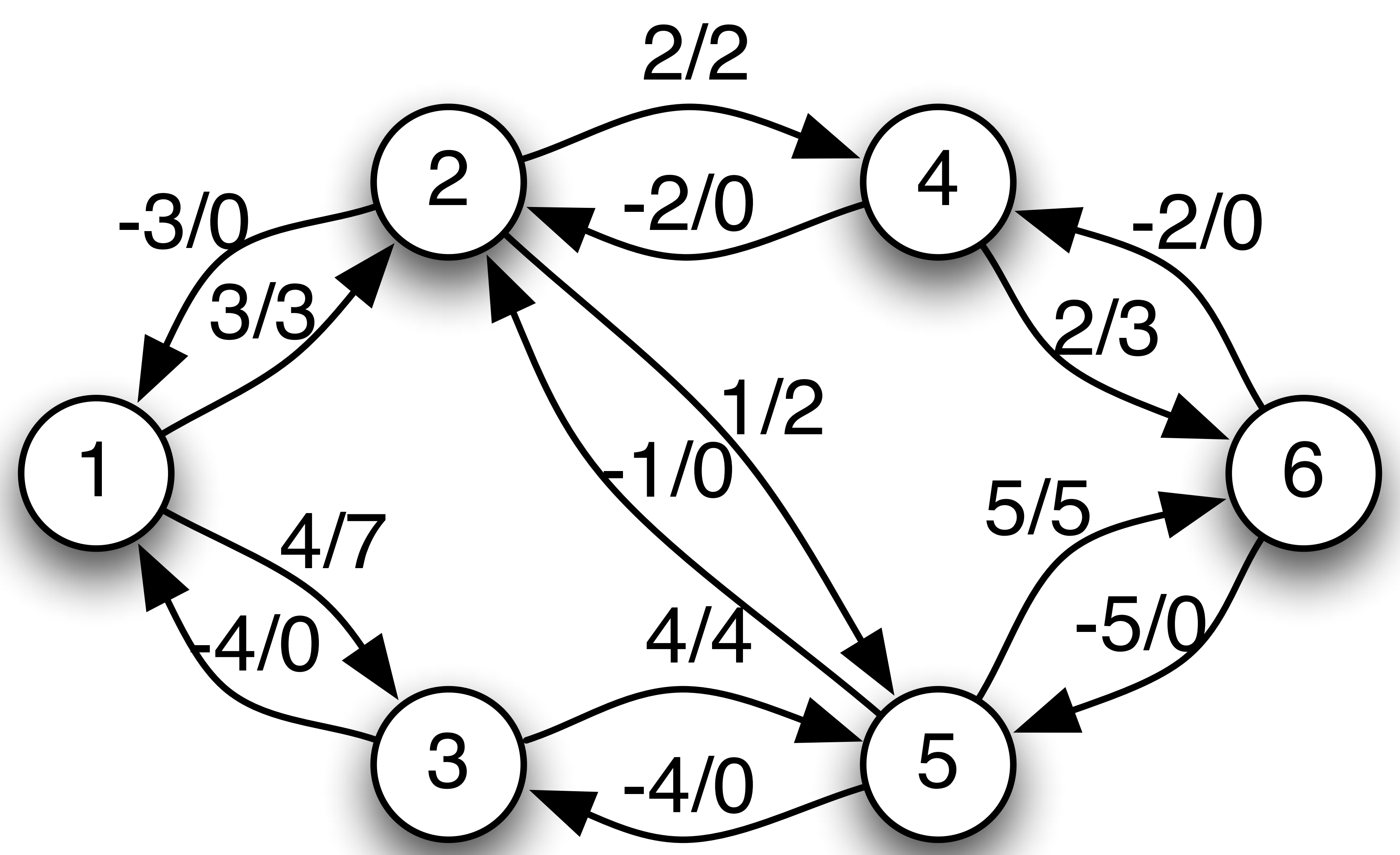
- We have an edge (a, b) in the residual graph if and only if the edge (a, b) is not saturated:

$$(a, b) \in E_f \iff f(a, b) < c(a, b)$$

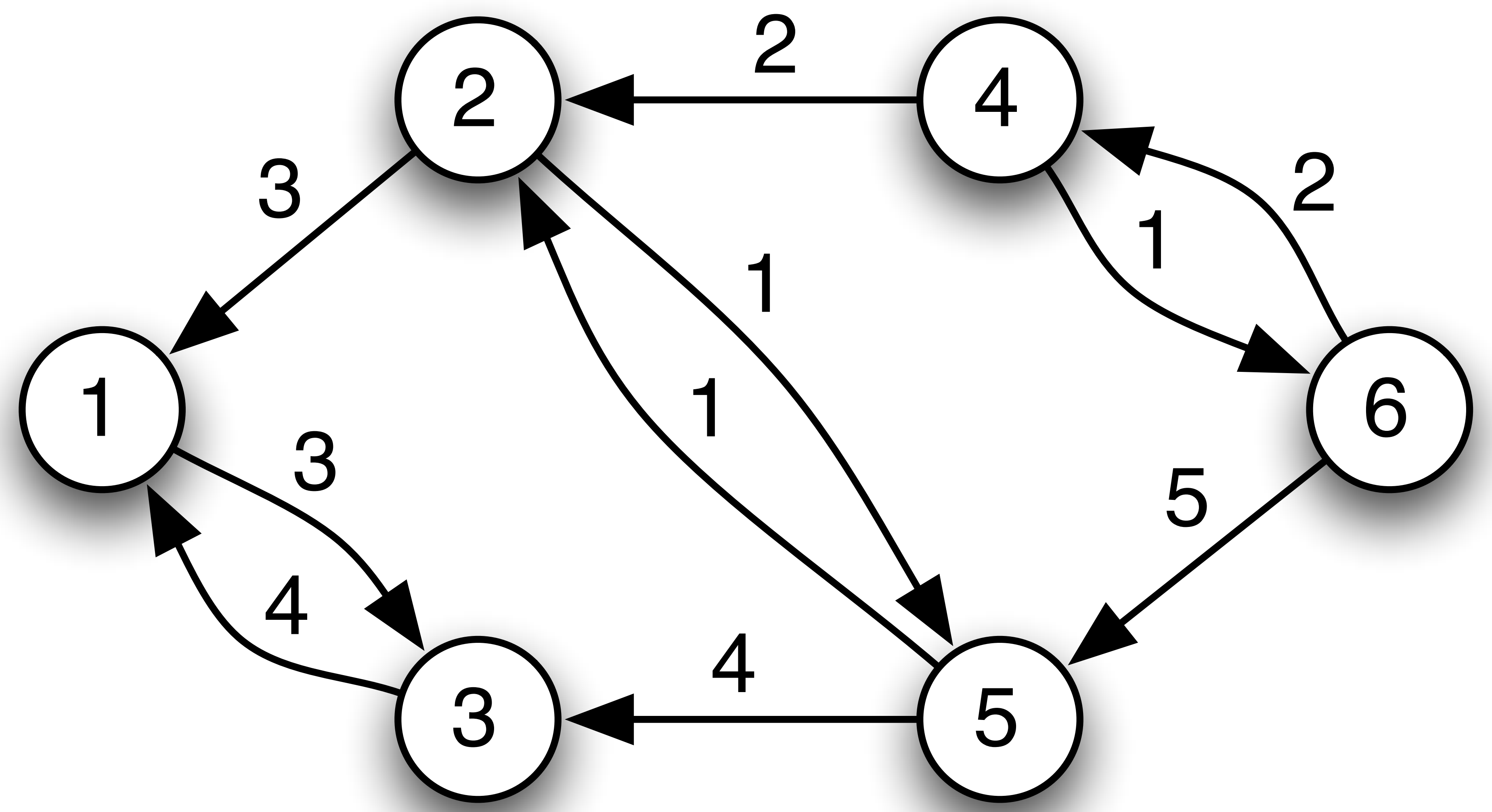
- The residual capacity is given by the quantity of flow that can be added without violating the capacity of the edge.

$$c_f(a, b) = c(a, b) - f(a, b)$$

Example Residual Graph



Graph G with flow f



Residual Graph G_f

Augmenting Path

- A path joining the source *s* to the sink *t* in the residual graph G_f is called **augmenting path**.
- It is a path on which it is possible to « push » at least one more unit of flow along the edges, 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 $c_f(a,b)$ associated to an edge on this path. The following flow *f'* is a valid flow with value $v(f) + q$.

$$f'(a,b) = \begin{cases} f(a,b) + q & \text{if (a,b) is on the augmenting path} \\ f(a,b) - q & \text{if (b,a) is on the augmenting path} \\ f(a,b) & \text{otherwise} \end{cases}$$

Ford-Fulkerson Algoritm

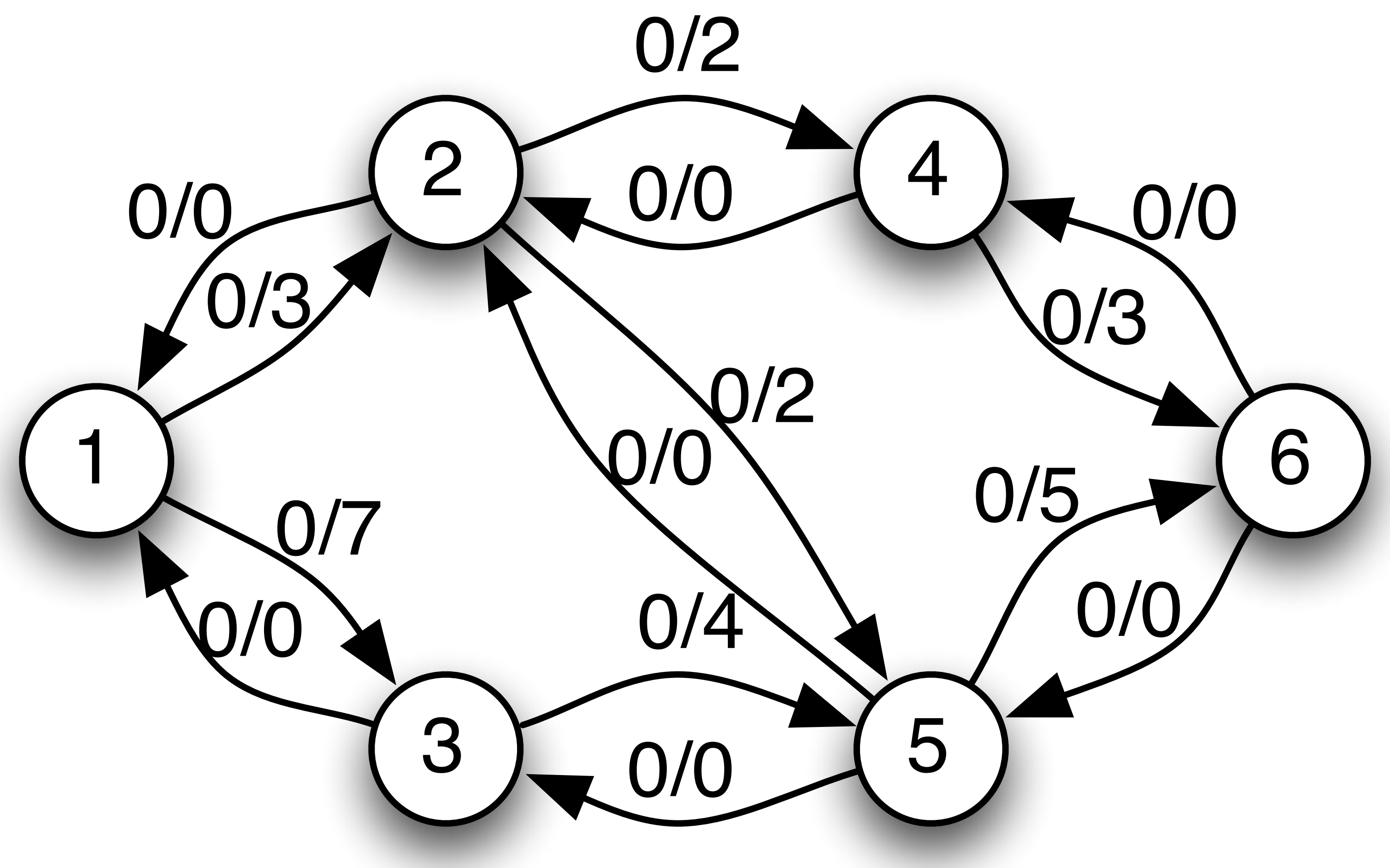
- Requires the capacities to be integers
- Is an iterative algorithm
- At each iteration, the algo takes a valid flow and transform it into another valid flow. This new flow has a strictly larger value, or the algo proves that there is no flow with a larger value.
- The flow increase is done along a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

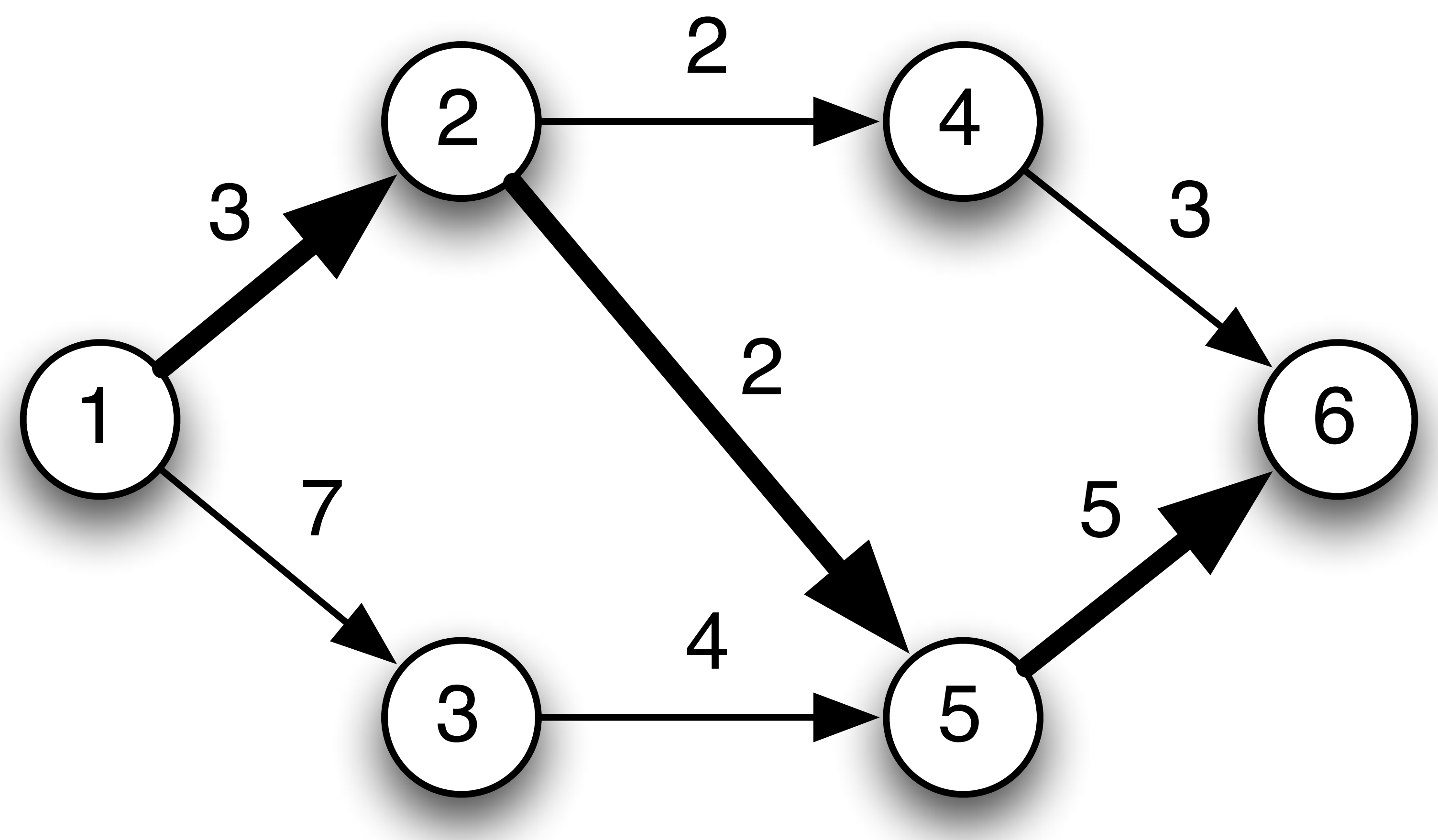
Ford-Fulkerson

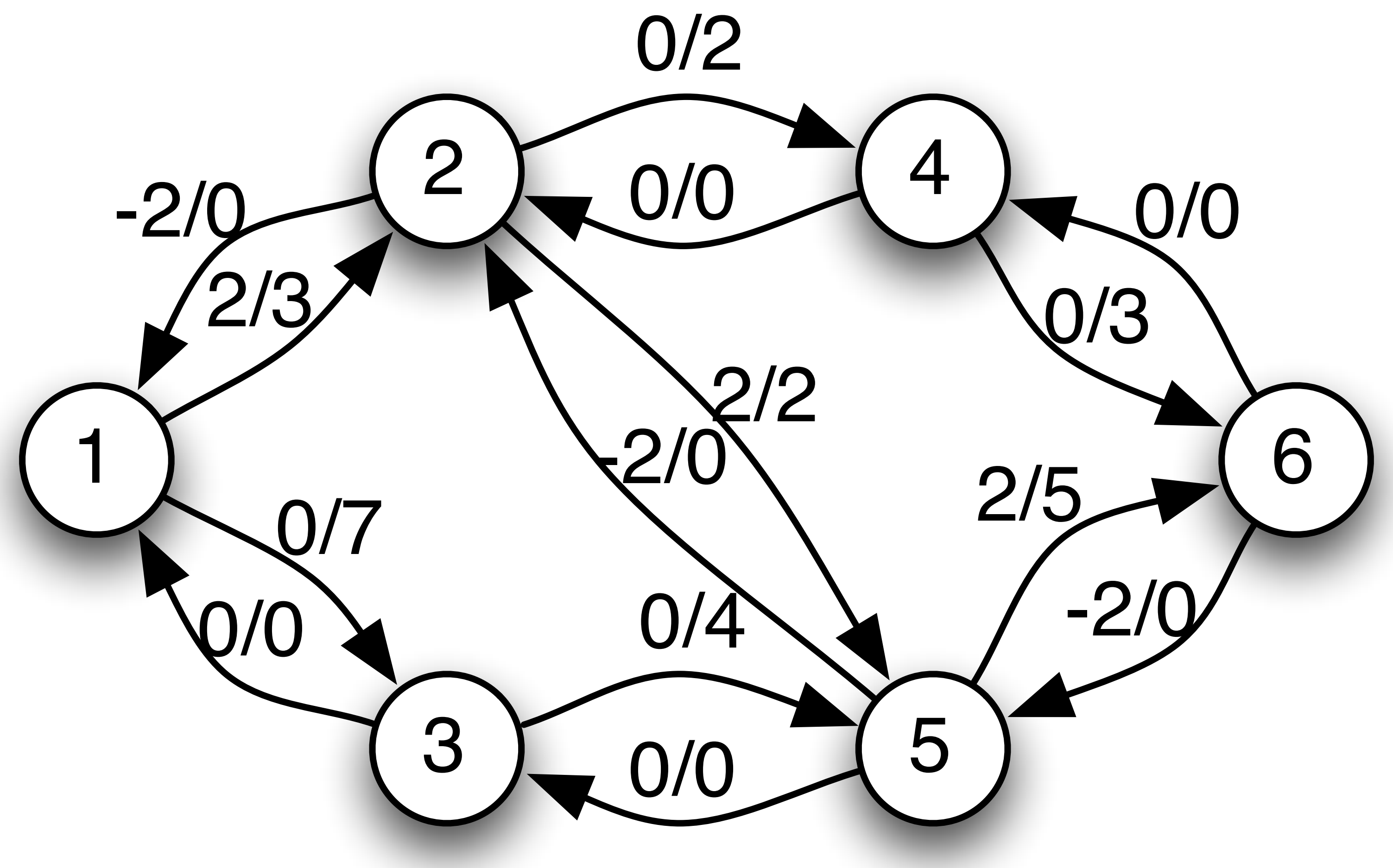
Algorithm 1: Ford-Fulkerson(V, E, c, s, t)

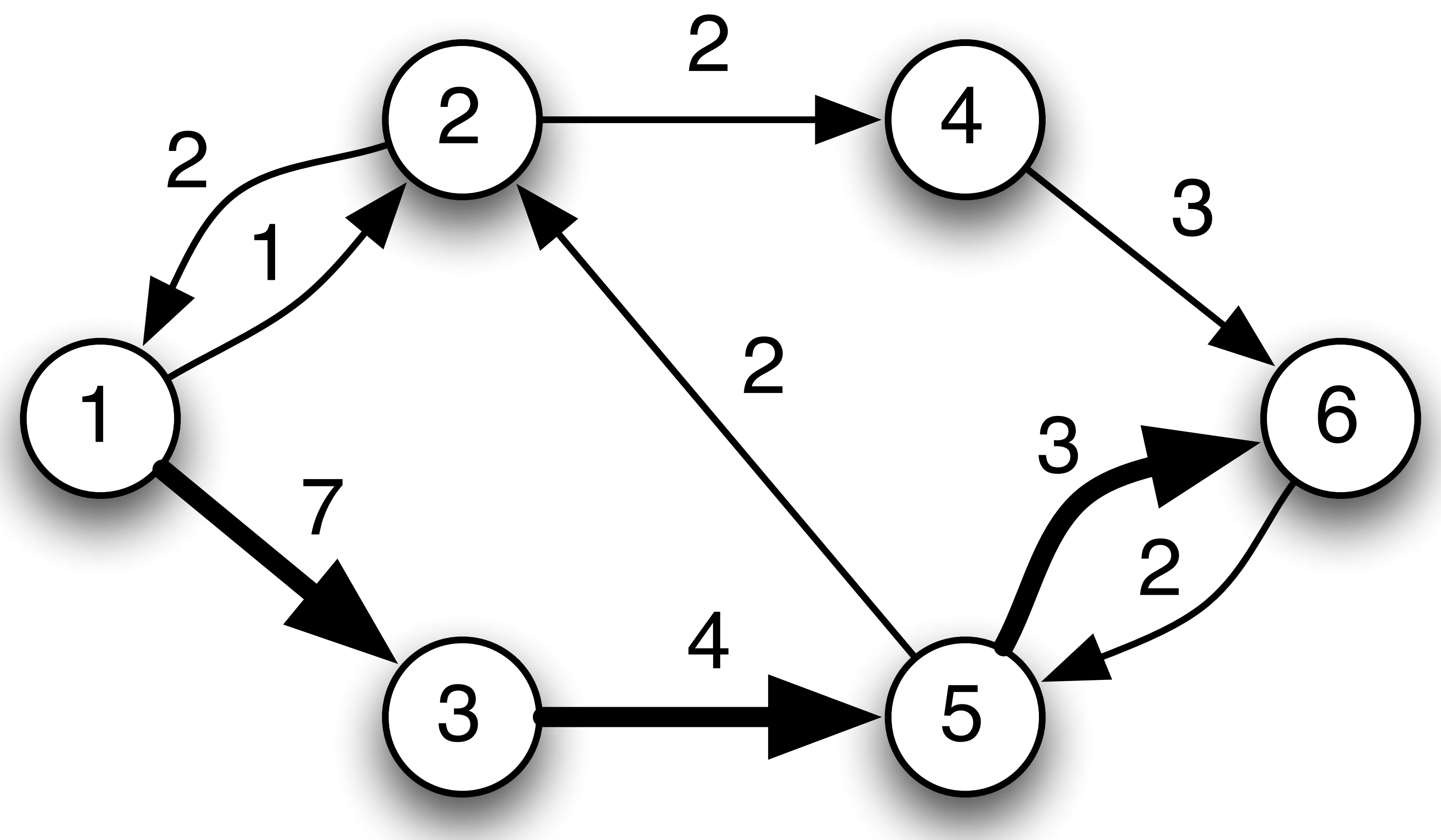
```
Build a vector $f$ with $\binom{|V|}{2}$ entries initialized at 0.;
repeat
    Build the residual graph $G_f$;
    Find a path $C$ from $s$ to $t$ in $G_f$;
    if such a path exists then
        Let $q$ be the smallest residual capacity on an edge of path $C$;
        for every edges $(a, b)$ of path $C$ do
            $f(a, b) \leftarrow f(a, b) + q$;
            $f(b, a) \leftarrow f(b, a) - q$;
until we cannot find a path between $s$ and $t$ in $G_f$;
return $f$;
```

Execution Ford-Fulkerson 1/3

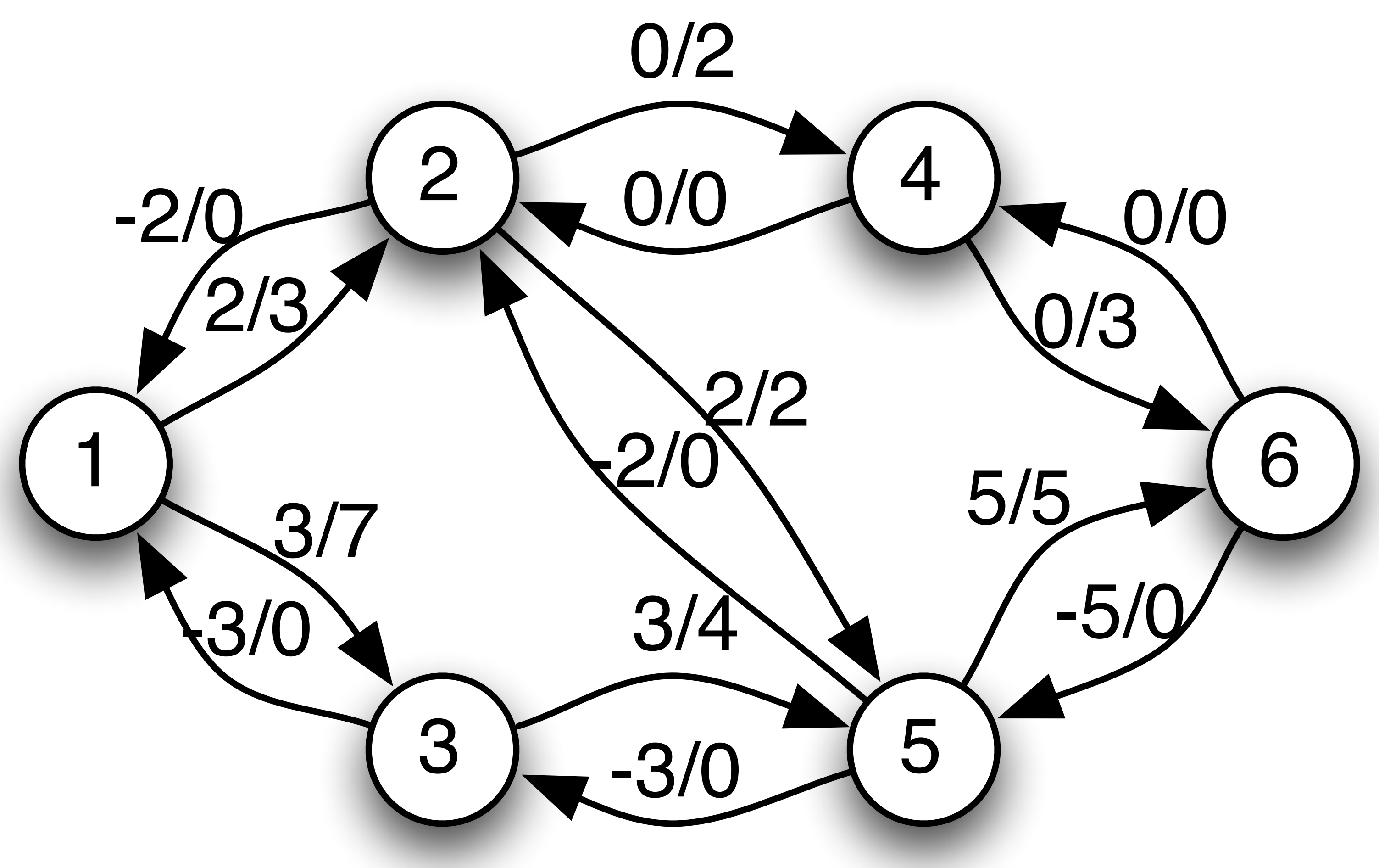


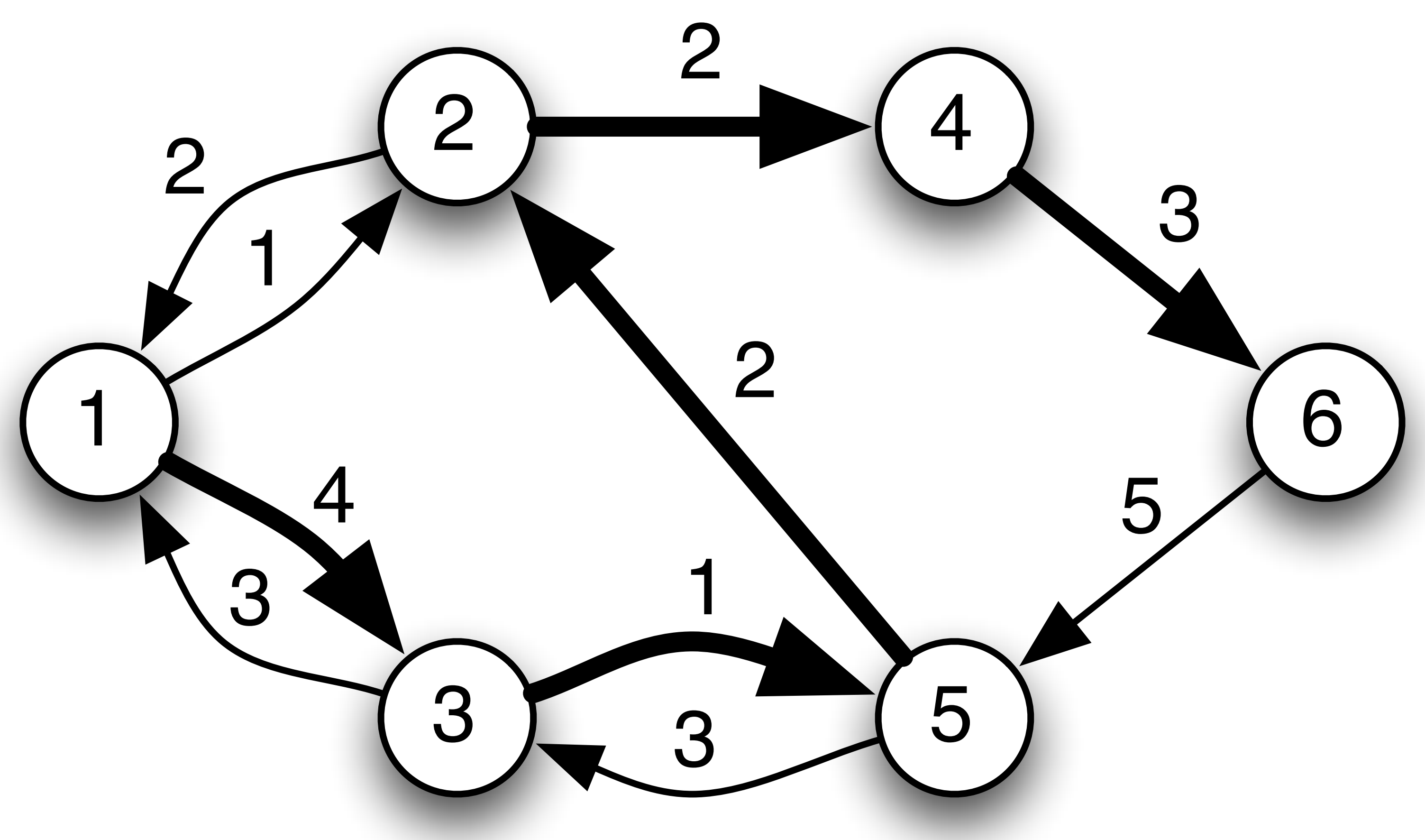


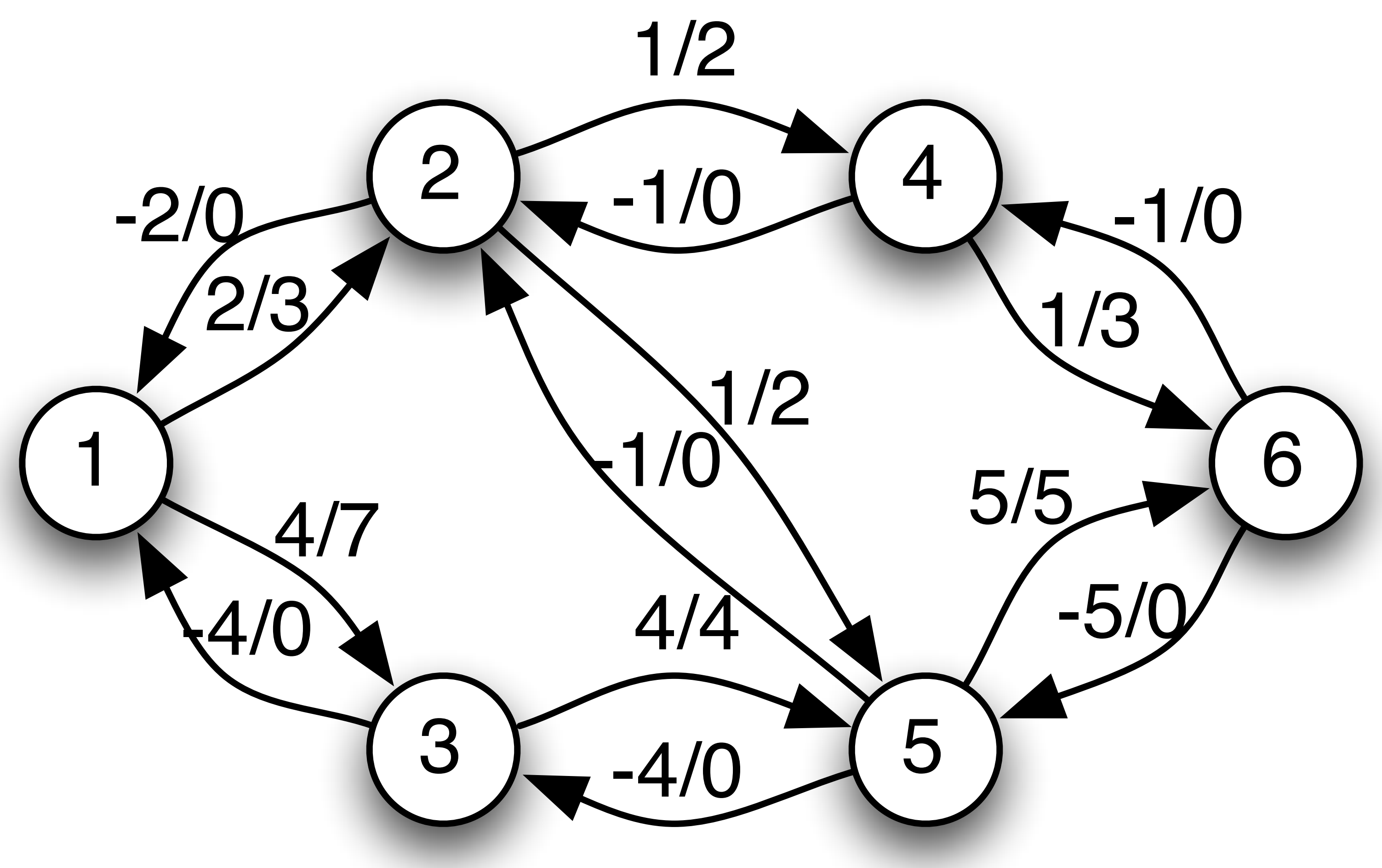


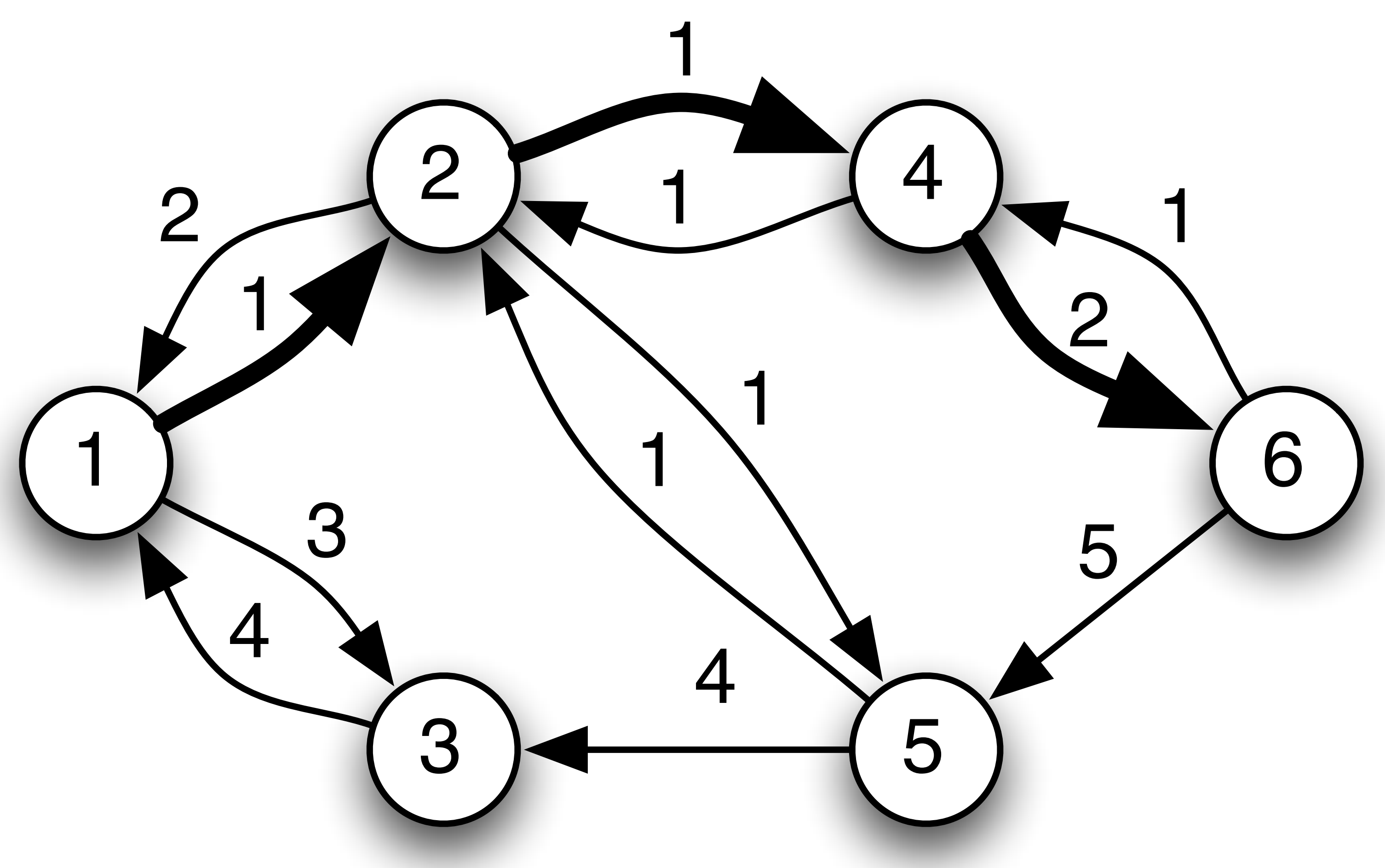


Execution Ford-Fulkerson 2/3

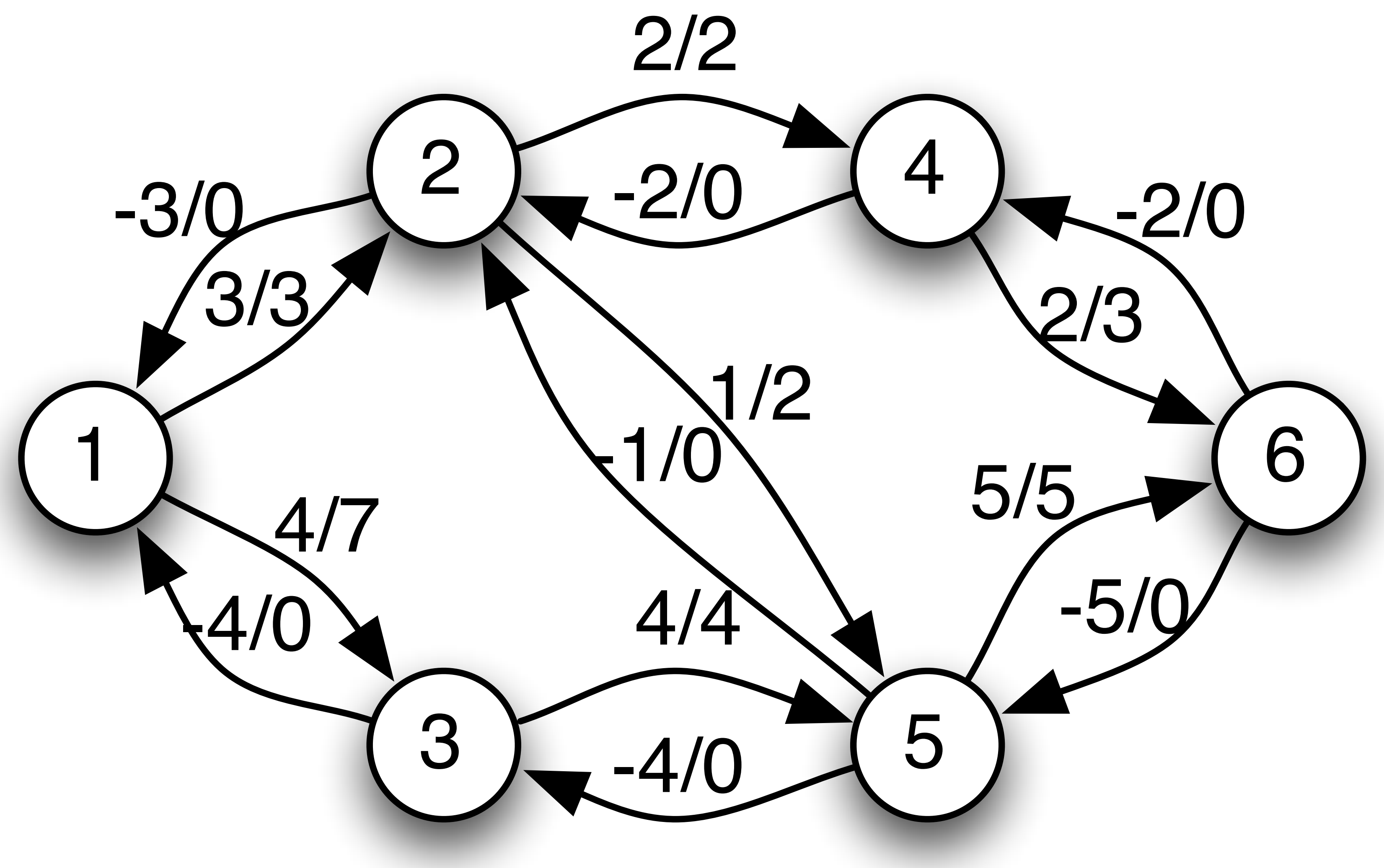


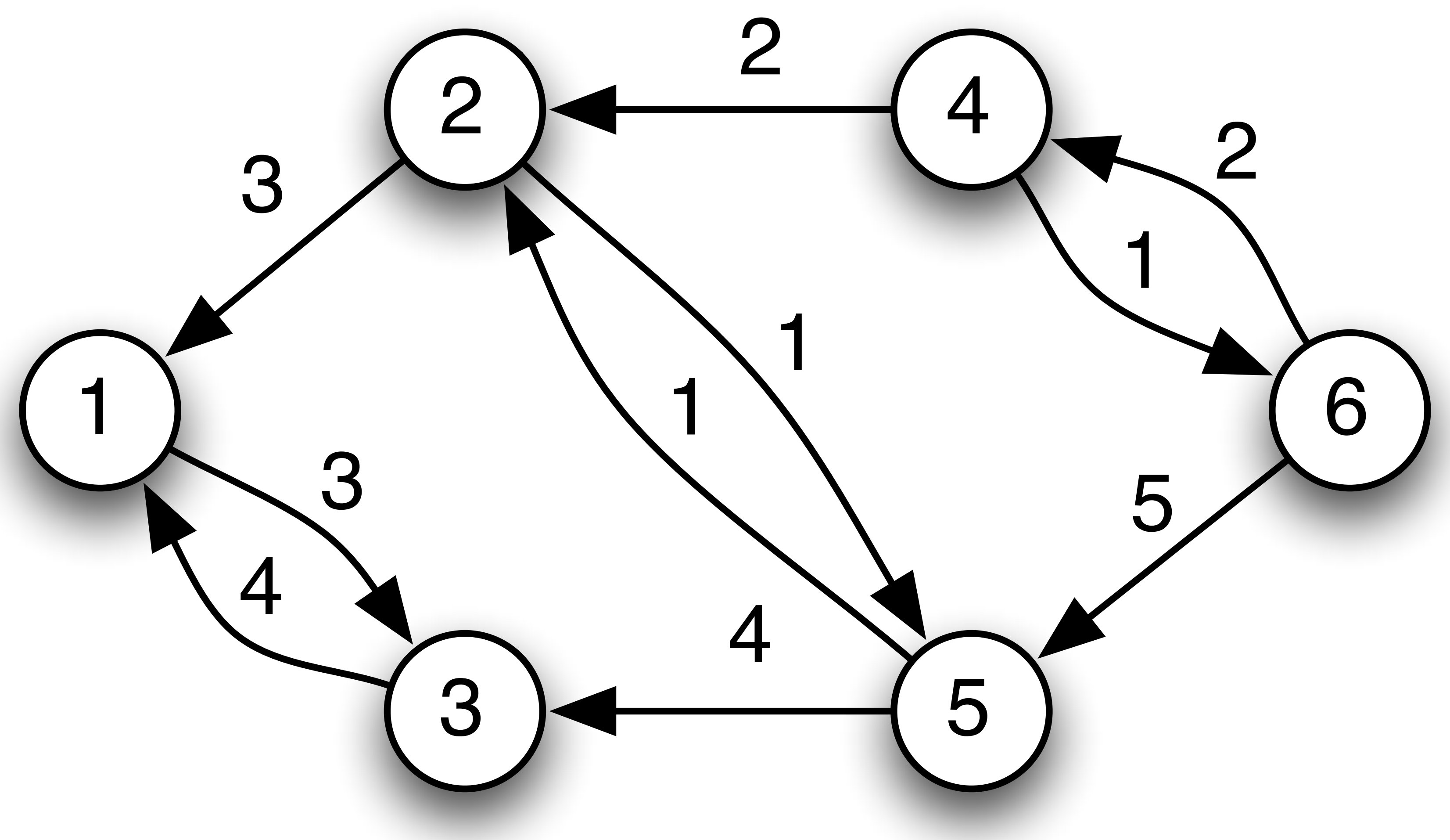






Execution Ford-Fulkerson 3/3





Ford-Fulkerson Time Complexity

- At each iteration, the algorithms of Ford-Fulkerson searches for a path in the residual graph, it takes $O(|E| + |V|)$ (DFS). Since the graph is connected we have $|V| - 1 \leq |E|$. Looking for a path is thus in $O(|E|)$.
- We then need to update the residual graph by processing each edge of the augmenting path $O(|V|)$.
- In the worst case, we need as many iterations as the value of the final maximal flow i.e. $v(f)$.
- The total complexity is thus $O(v(f)|E|)$.

Dealing with Bidirectional Edges

- We drop the hypothesis that edges are uni-directional.
- The input graph is now assumed to have bi-directional edges (set capacity to 0 if the edge does not exist in one direction).
- The flow is defined in both directions of an edge keeping the relation:

$$f(a,b) + f(b,a) = 0$$

- A negative flow on edge (a, b) means that the water flows in the reverse direction i.e. from b to a.
- The flow conservation constraint is written:

$$\sum_{v \in V} f(u,v) = \sum_{v \in V} f(v,u) = 0$$

Residual Graph with Bi-directional Edges

- The residual graph $G_f = (V, E_f)$ with bidirectional edges is defined as follows.
- Edge (a, b) belongs to E_f if and only if it is not saturated:

$$(a, b) \in E_f \iff f(a, b) < c(a, b)$$

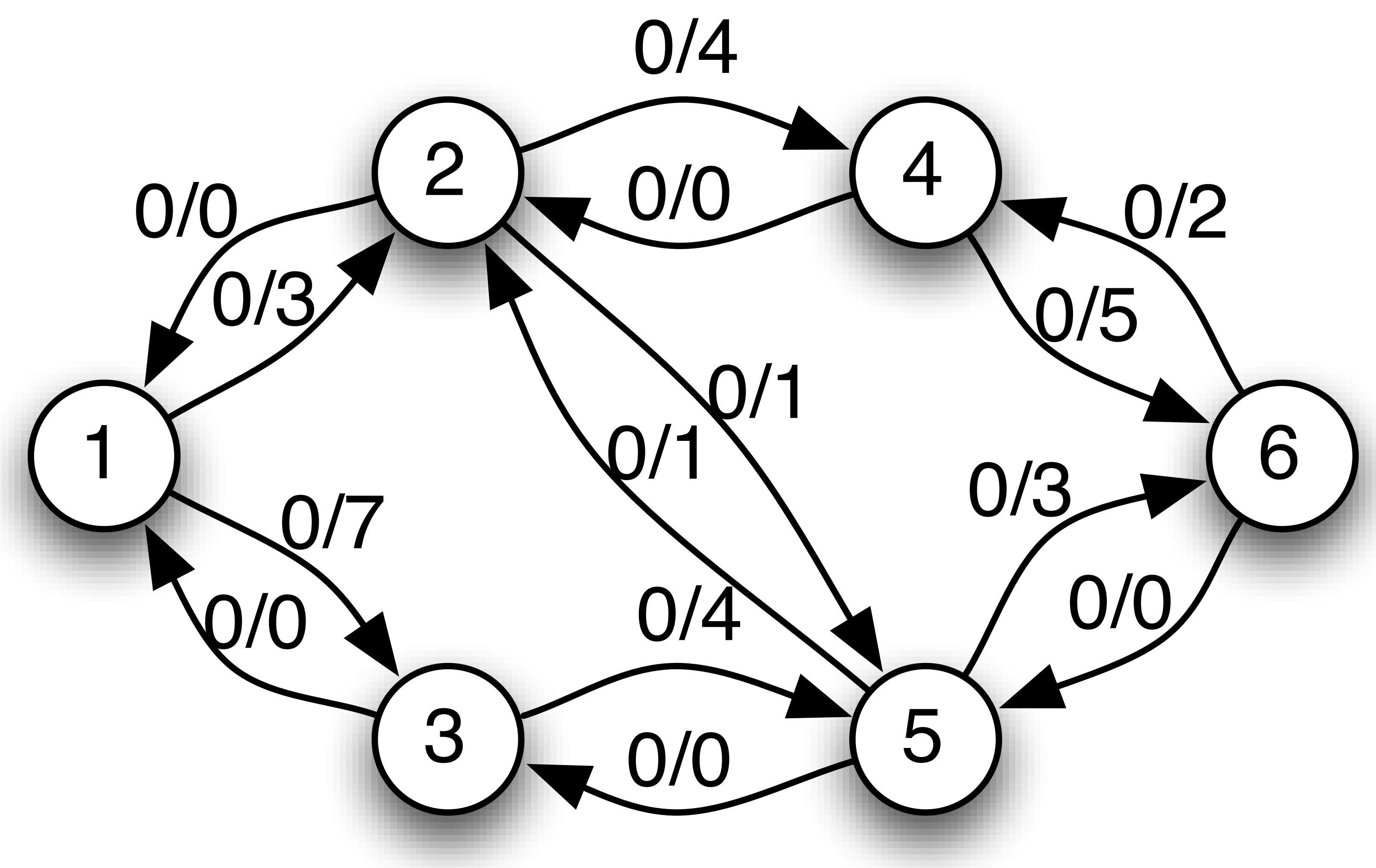
- The residual capacity of edge (a, b) is the additional flow quantity that can be pushed on this edge:

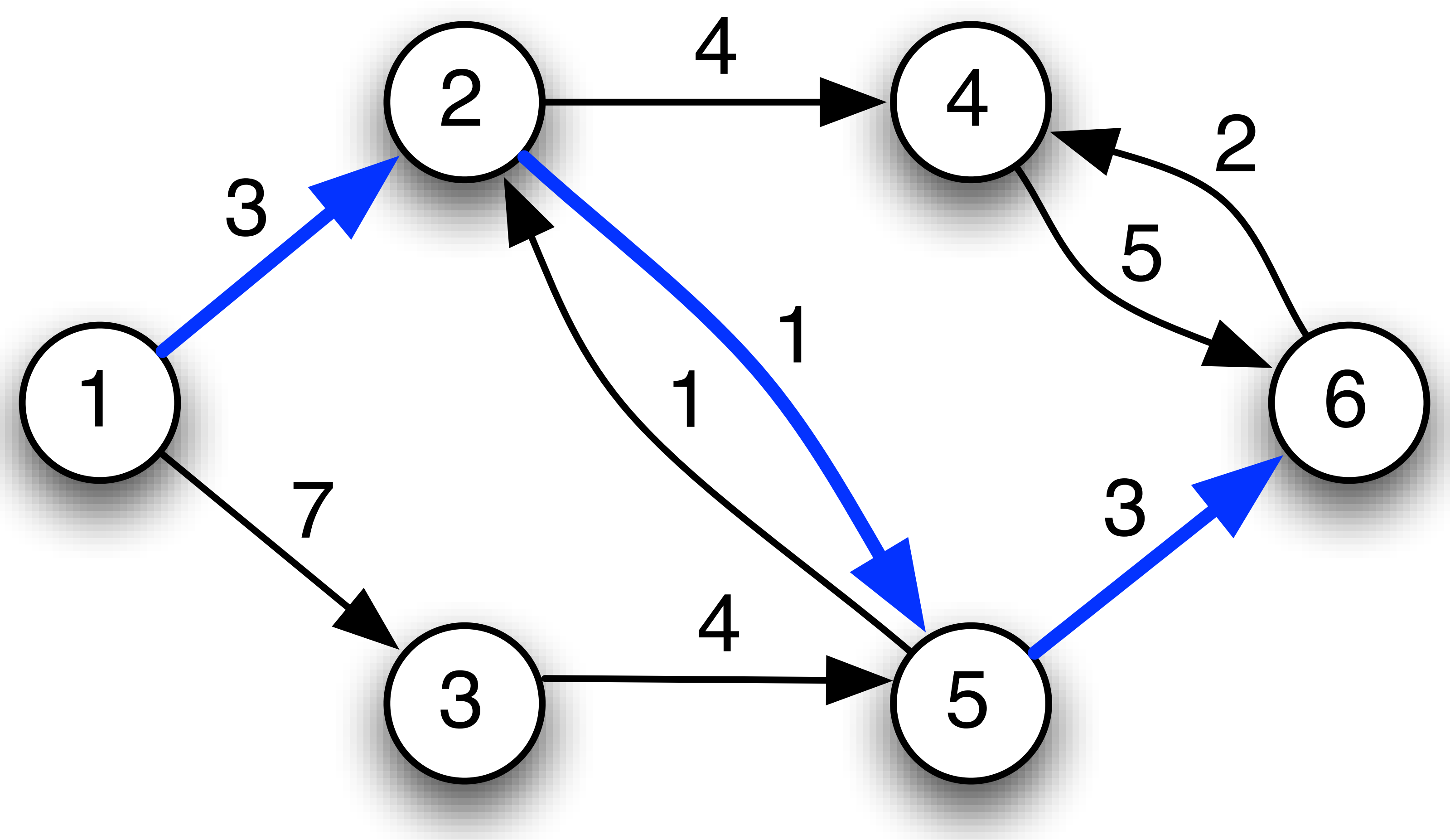
$$c_f(a, b) = c(a, b) - f(a, b)$$

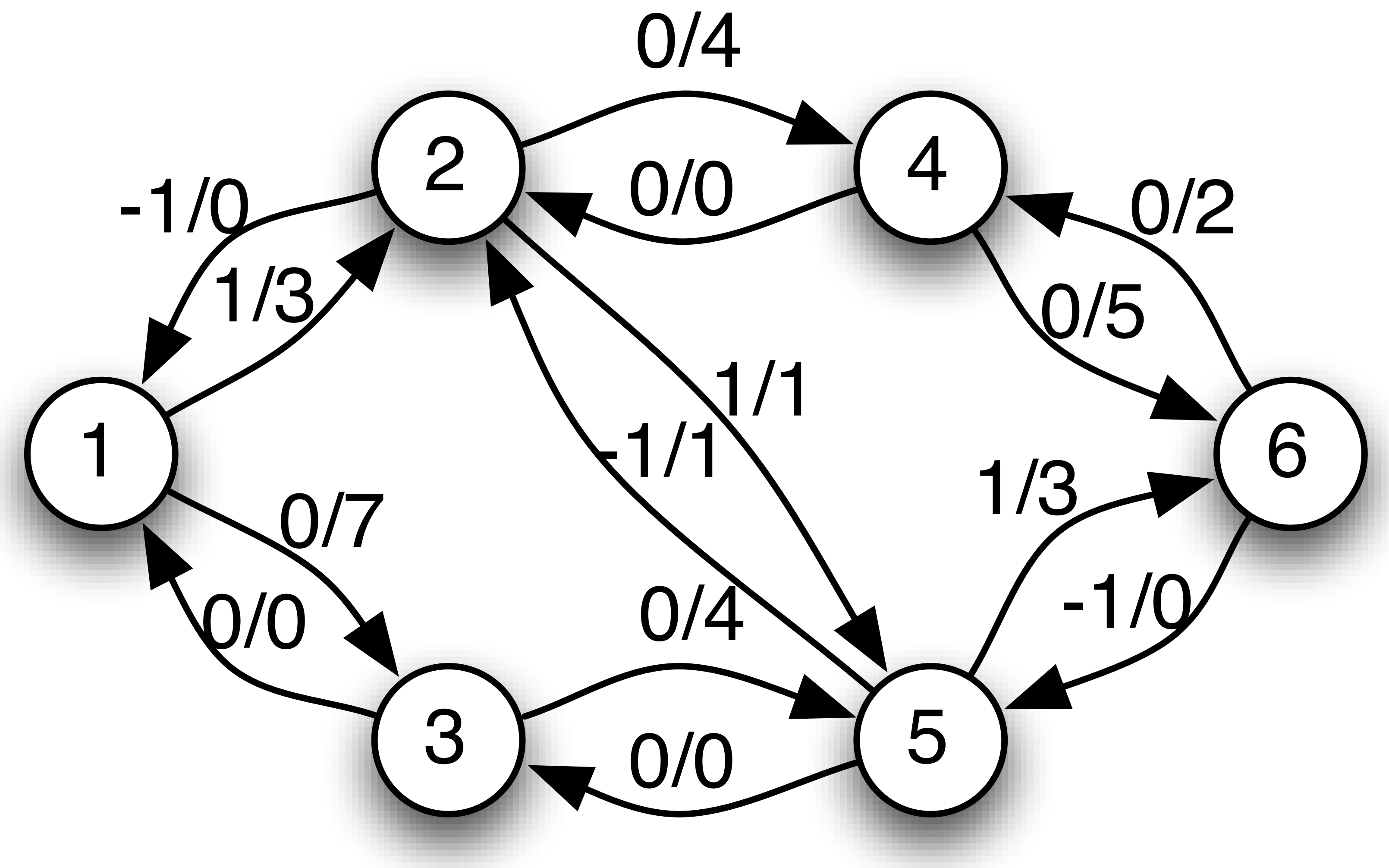
- Let C be an augmenting path in G_f and q be the smallest residual capacity on this path, the new augmented flow f' is defined as follow:

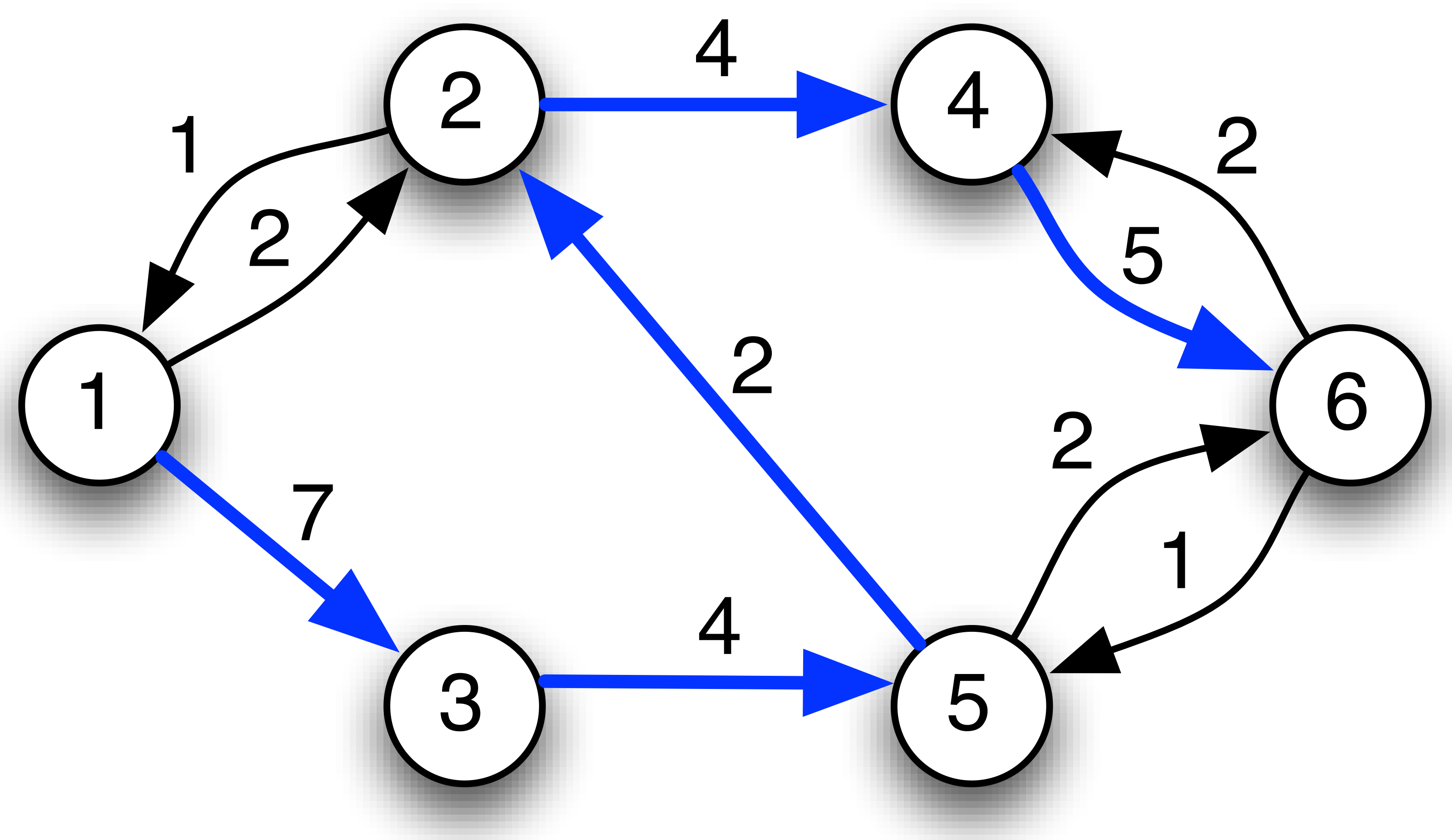
$$f'(a, b) = \begin{cases} f(a, b) + q & \text{if (a,b) in C} \\ f(a, b) - q & \text{if (b,a) in C} \\ f(a, b) & \text{otherwise} \end{cases}$$

Example Bidirectional Flows 1/3

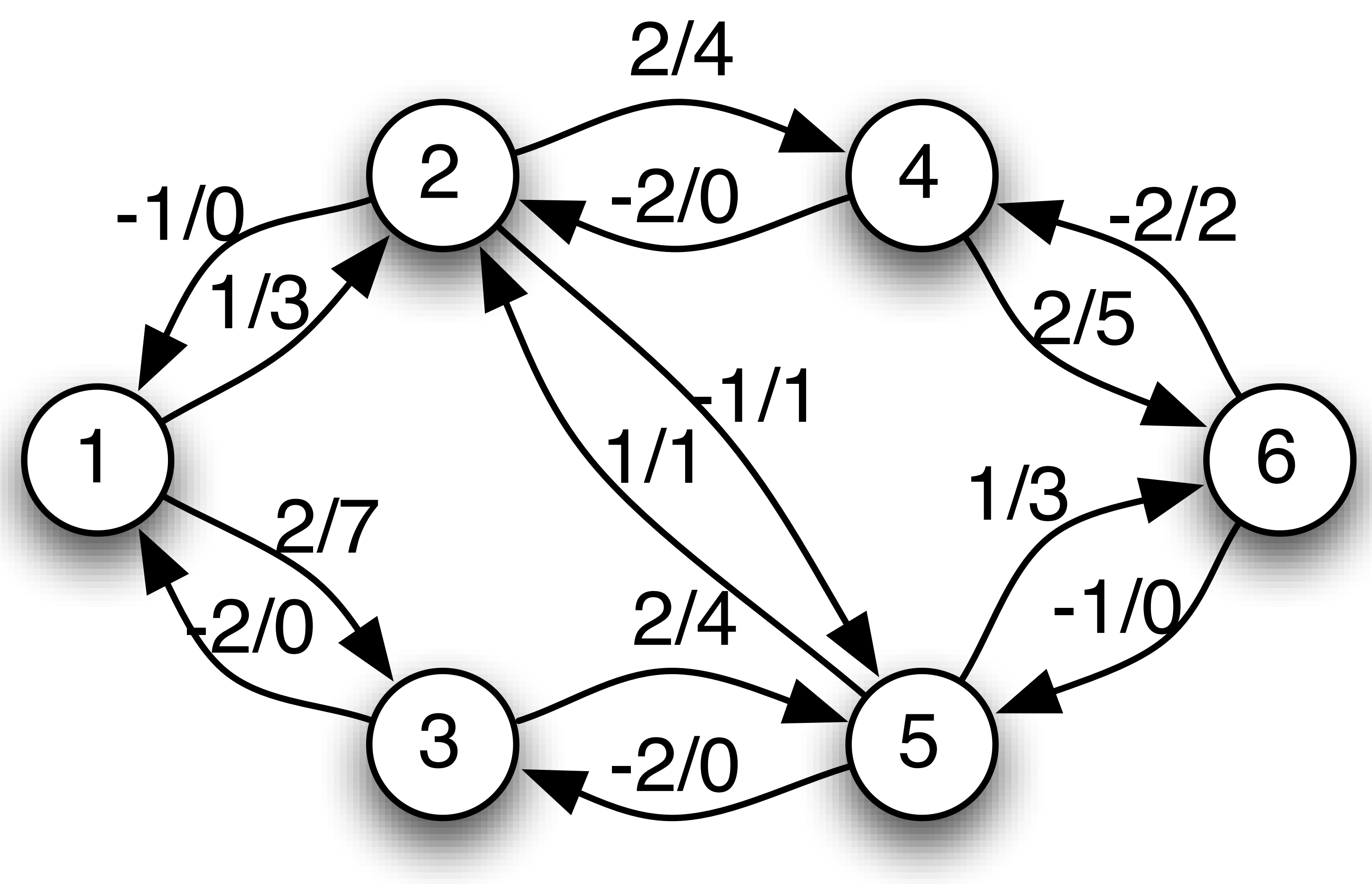


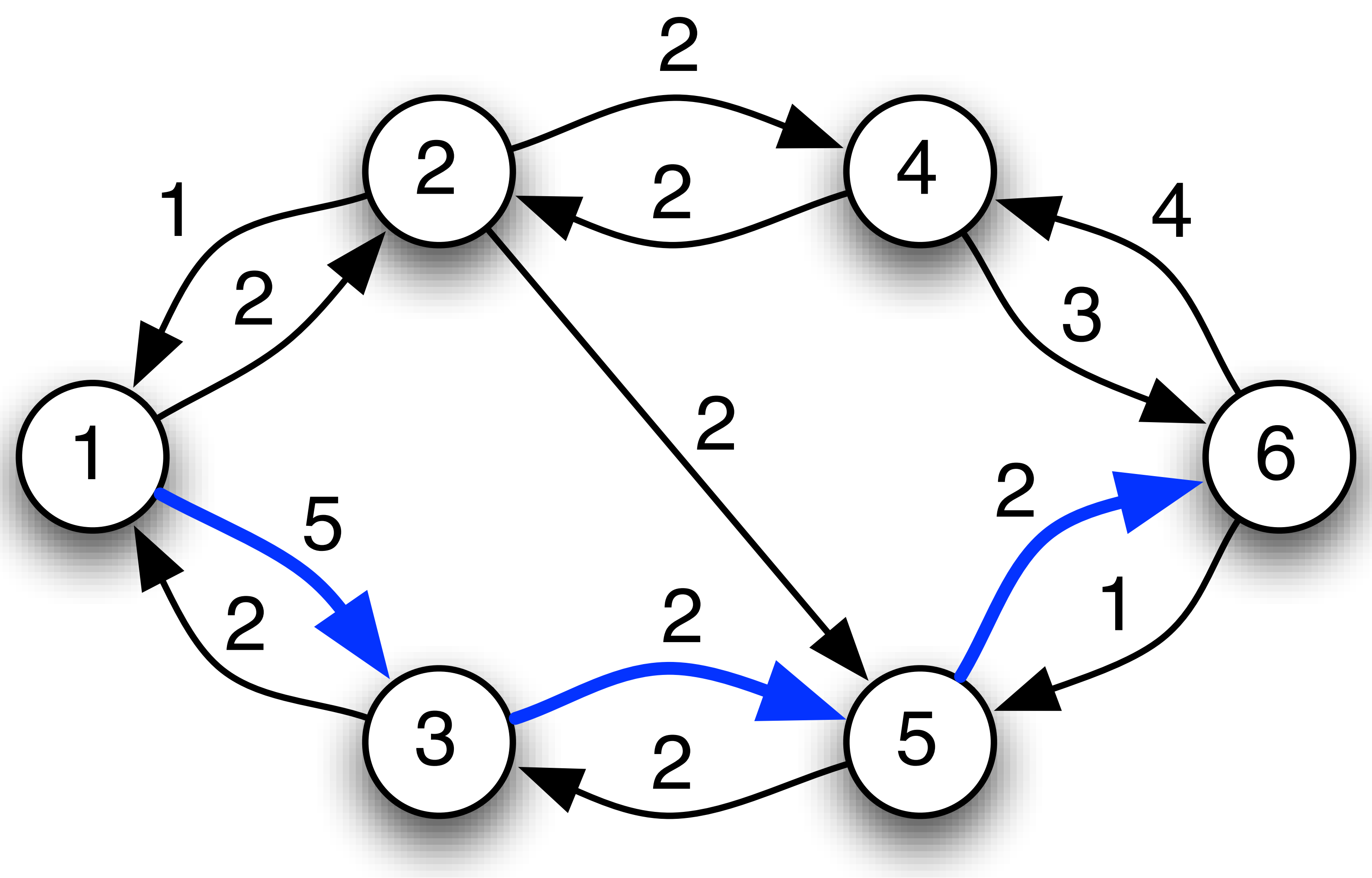


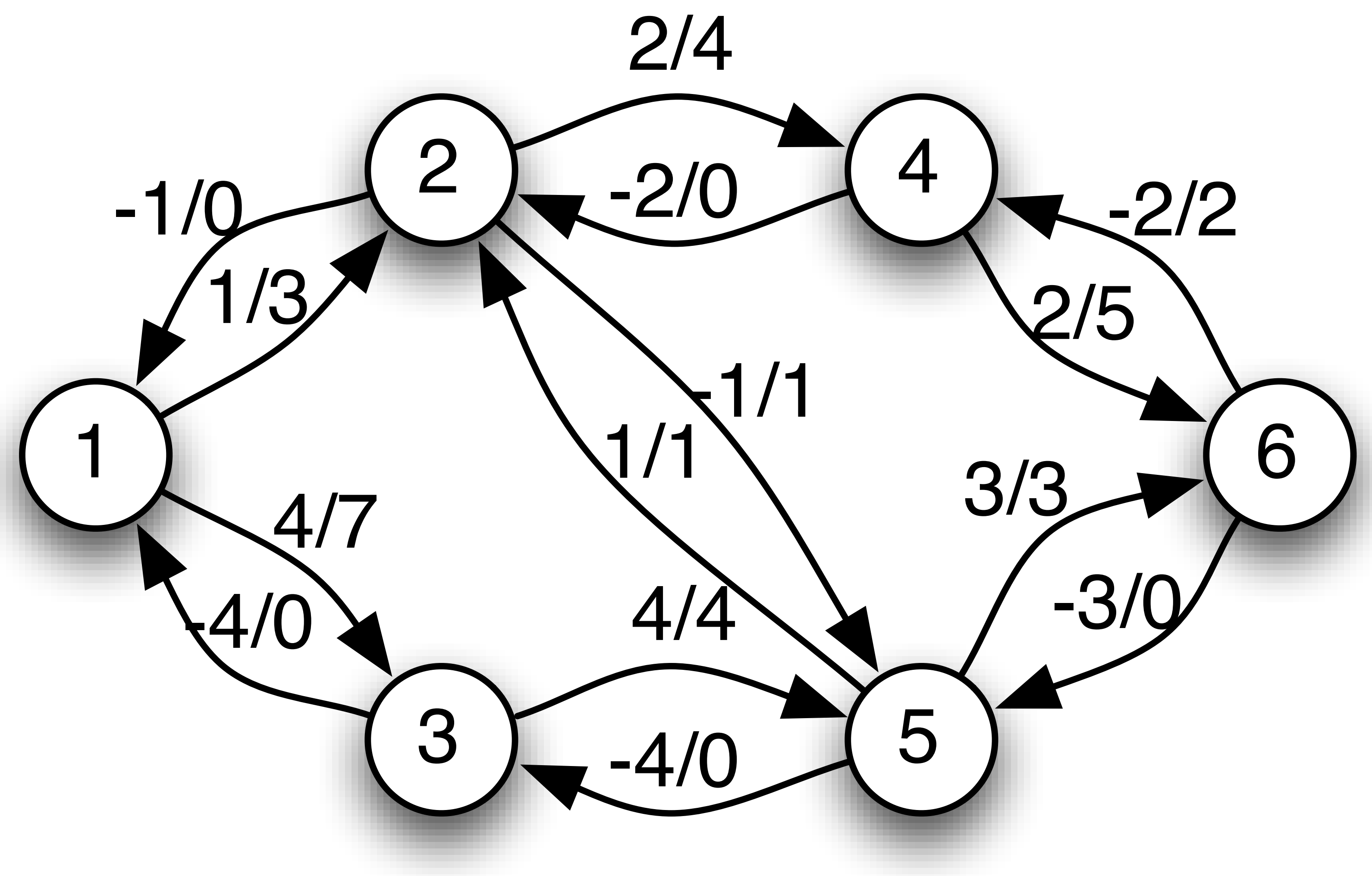


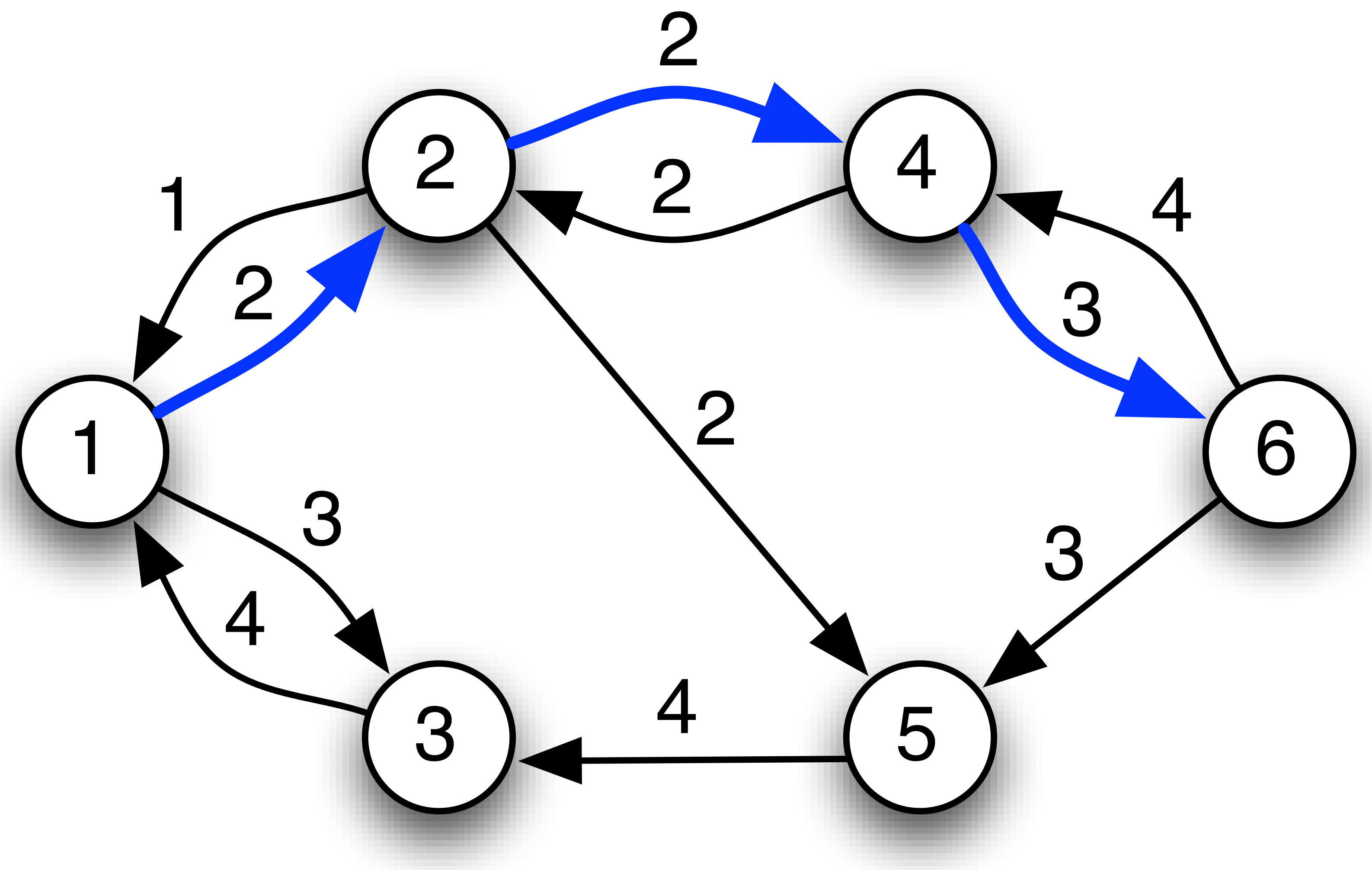


Example Bidirectional Flows 2/3

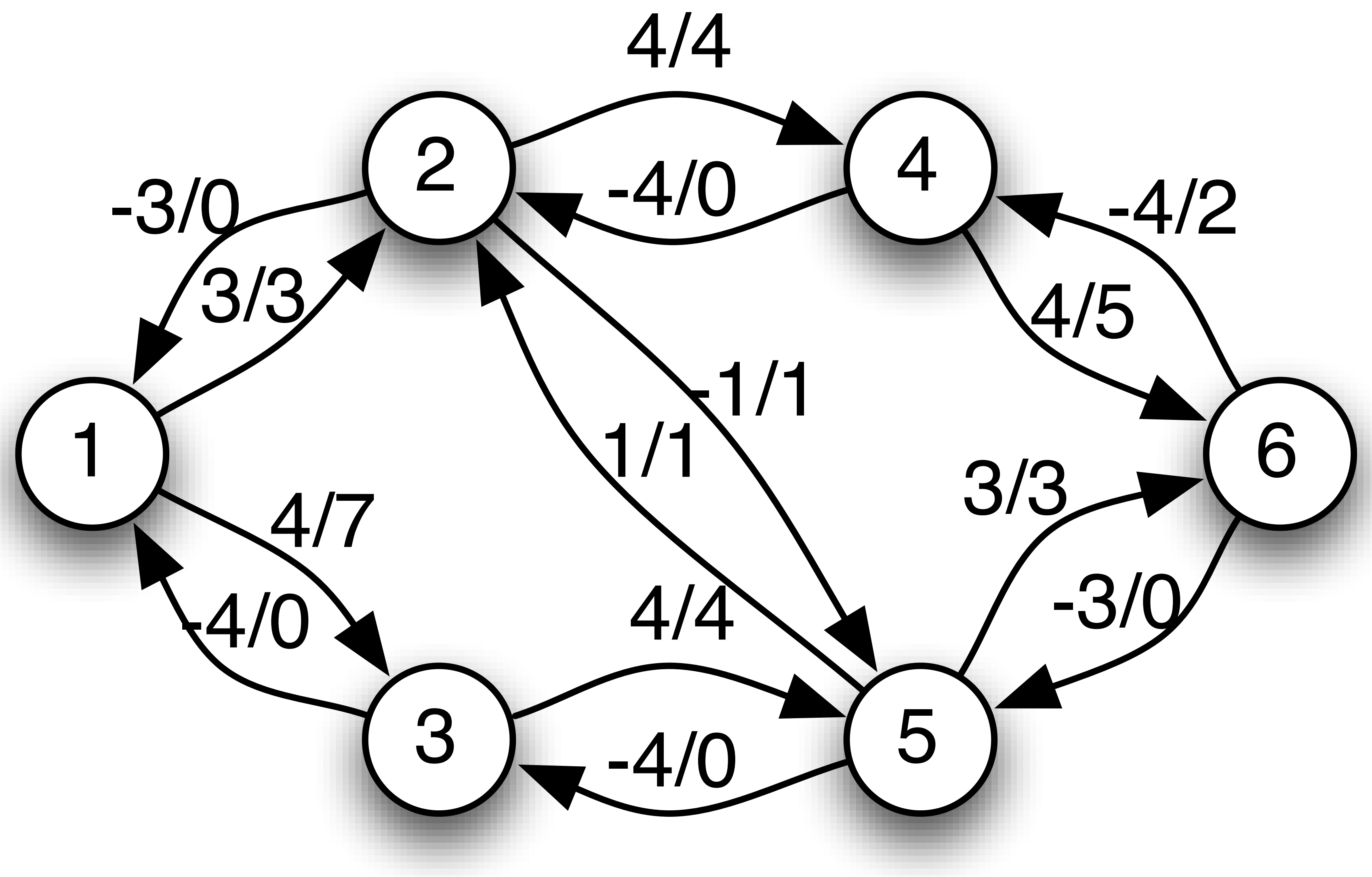


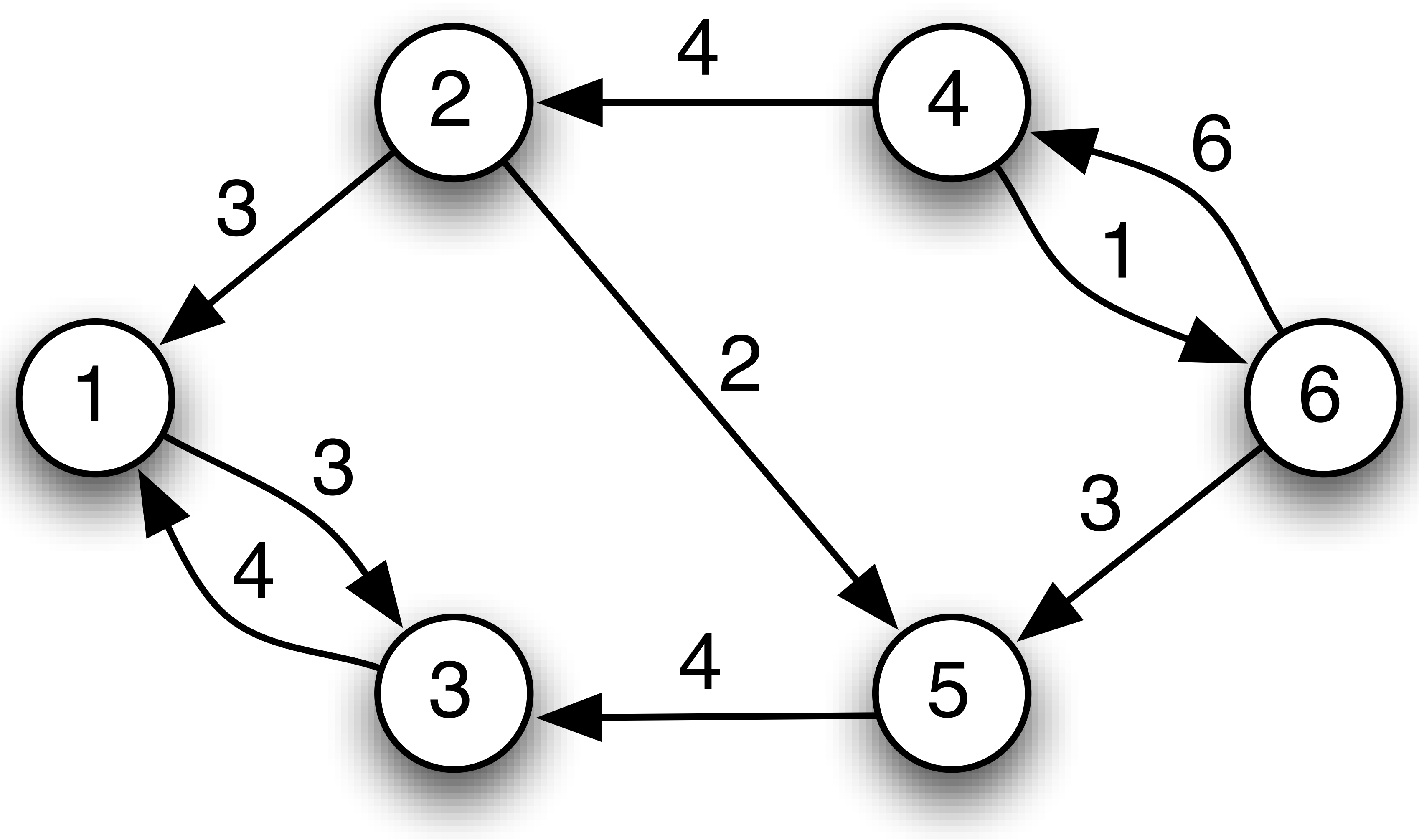






Example Bidirectional Flows 3/3





Still Alive?

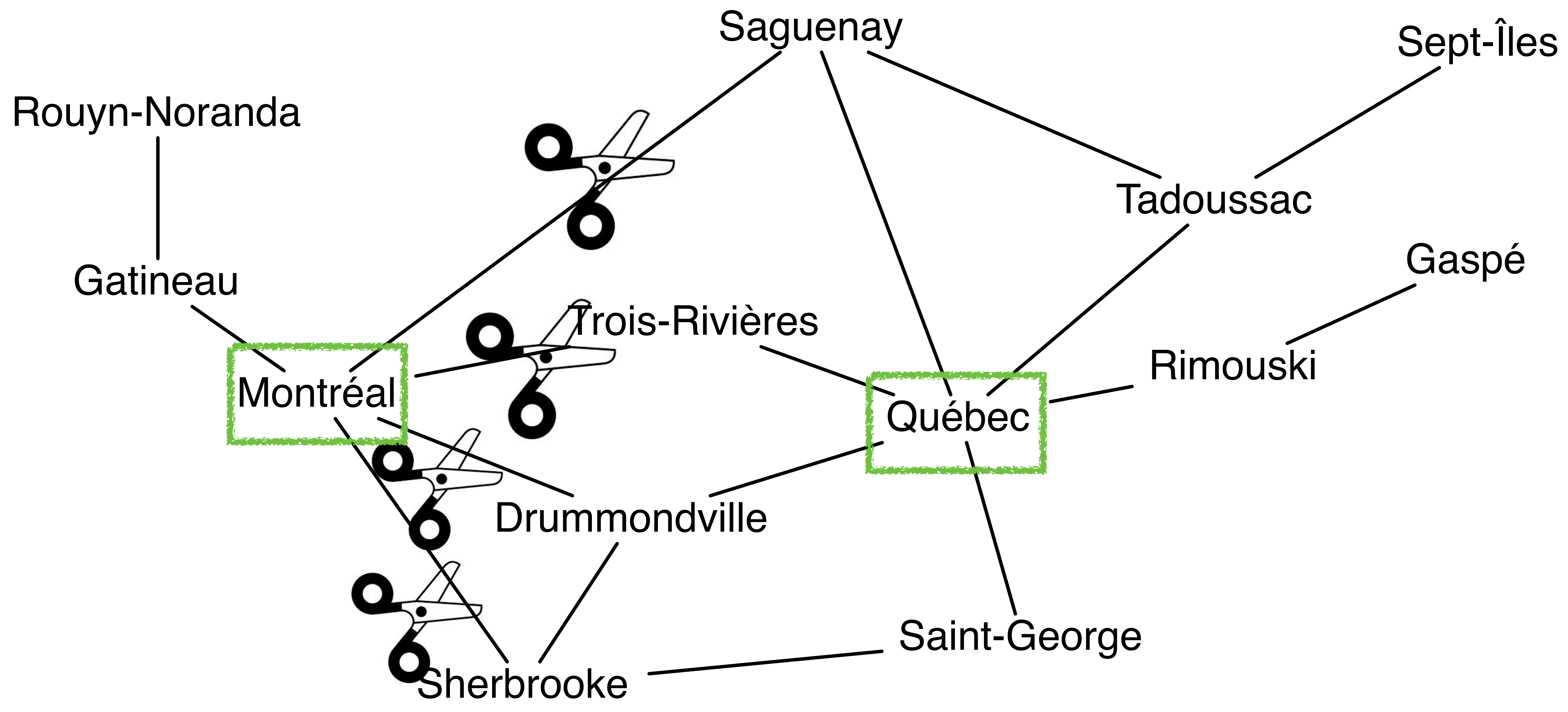
- If all of the edge capacities in a graph are an integer multiple of 7, then the value of the maximum flow will be a multiple of 7 (Yes or No?)

Still Alive?

- the maximum (s,t)-flow of some graph has value f. Now we increase the capacity of every edge by 1. Then the maximum (s,t)-flow in this modified graph will have value at most f + 1. (Yes/No?)

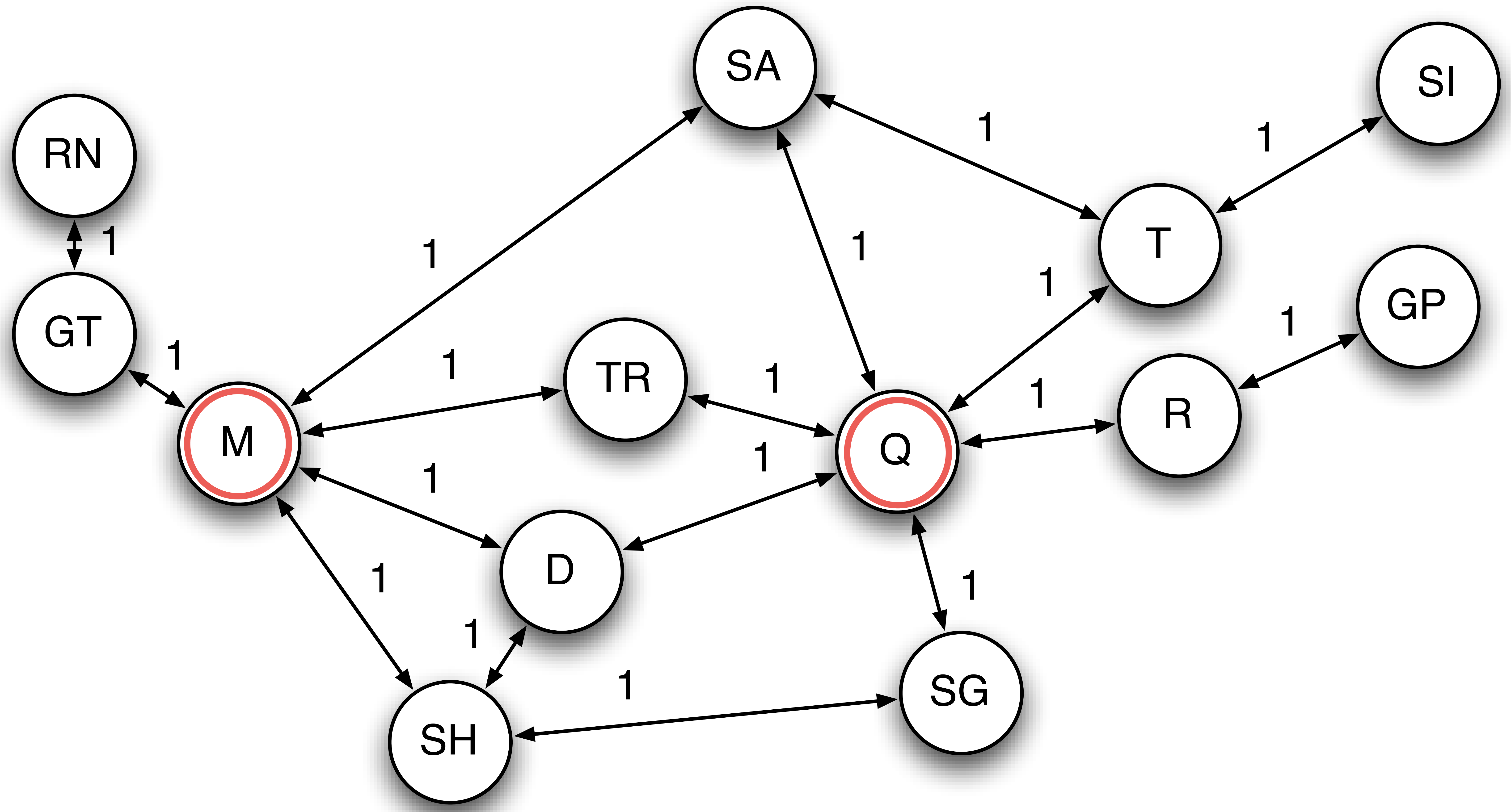
Evaluating Robustness of a Network

A telecom network linked cities of Québec with Optical Fiber.
What is the minimum number of fibres you need to cut to disconnect Montréal from Québec?



Evaluating Robustness of a Network (Cont)

Build a graph with one unit capacity on each edge. The maximum flow between Montréal (M) and Québec (Q) is the minimum number of fibers to be cut if you want to disconnect the two cities. Why?



Cut

- A cut is a bipartition of a set V formed of two disjoint sets S and T such that V is the union of S and T.

$$S \cup T = V \qquad S \cap T = \emptyset$$

- A **cut** in a network is a bipartition (S, T) of the nodes such that the source is in the set S and the sink is in the set T.

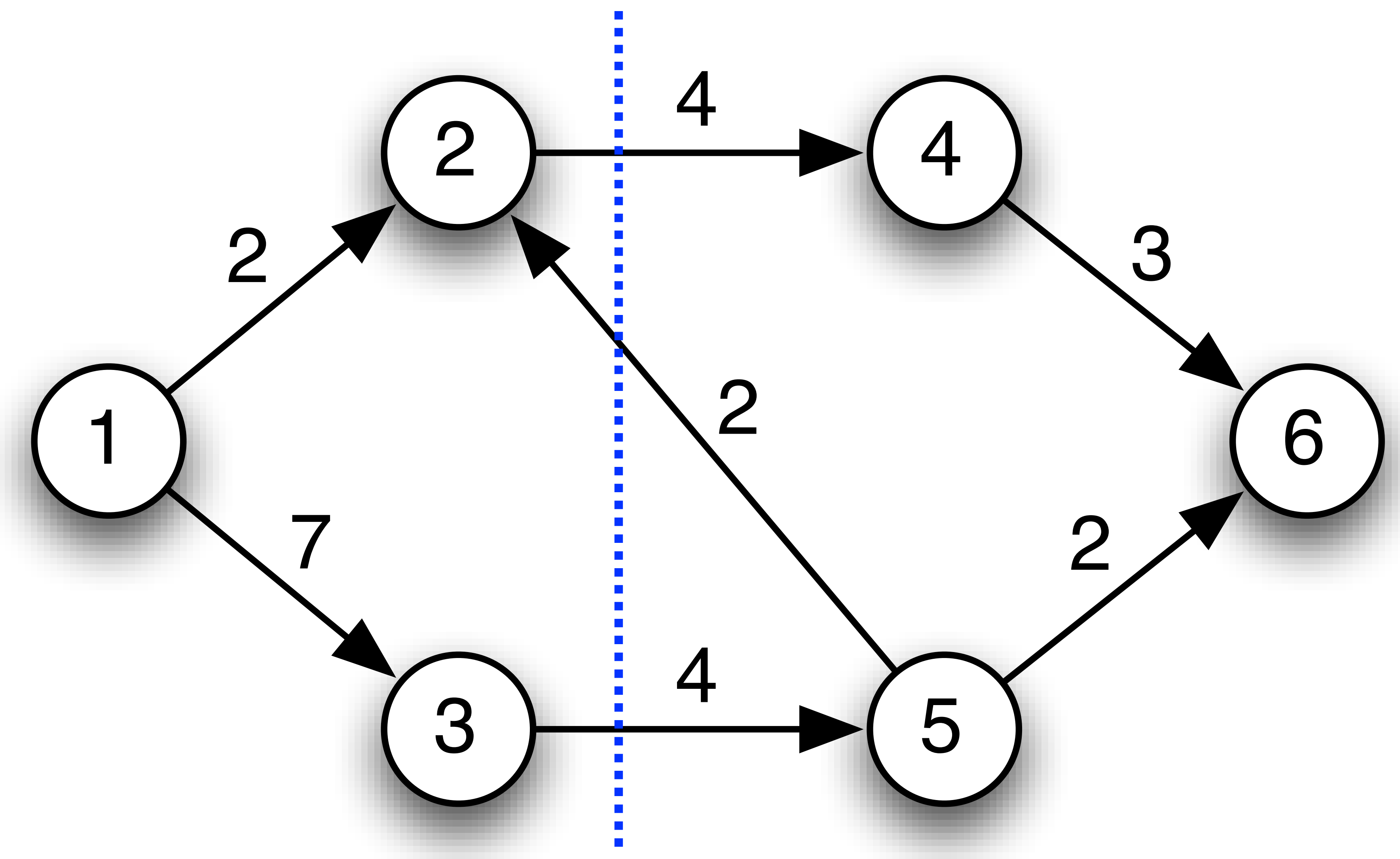
$$S \cup T = V \qquad S \cap T = \emptyset$$

$$s \in S \qquad t \in T$$

- The **capacity of a cut** (S, T) is the capacity of the edges linking a node from S to a node in T.

$$c(S,T) = \sum_{a \in S} \sum_{b \in T} c(a,b)$$

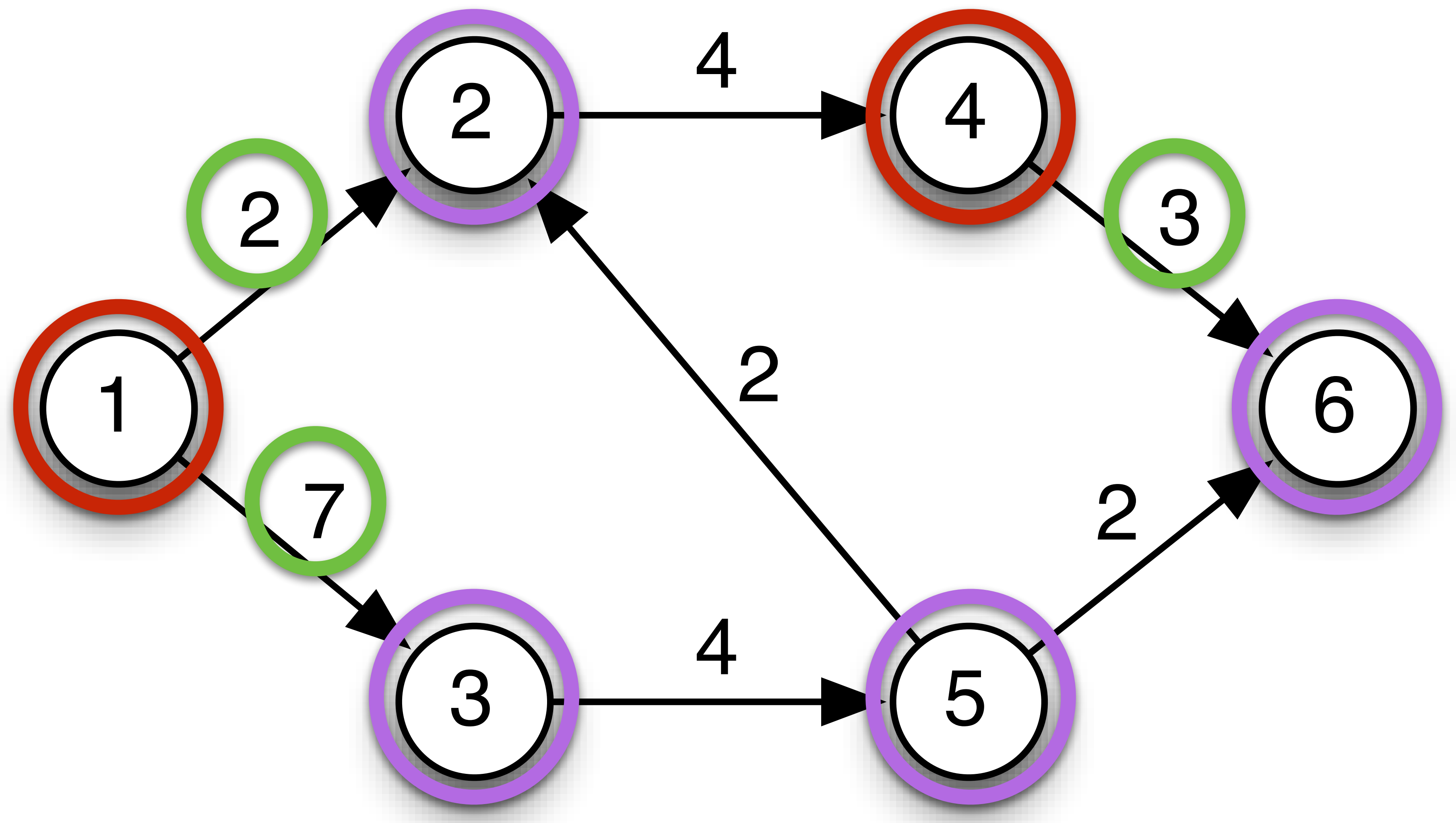
Cut example



S = {1, 2, 3} T = {4, 5, 6}

- The capacity of this cut is 8.
- We write c({1, 2, 3}, {4, 5, 6}) = 8.

Wooclap



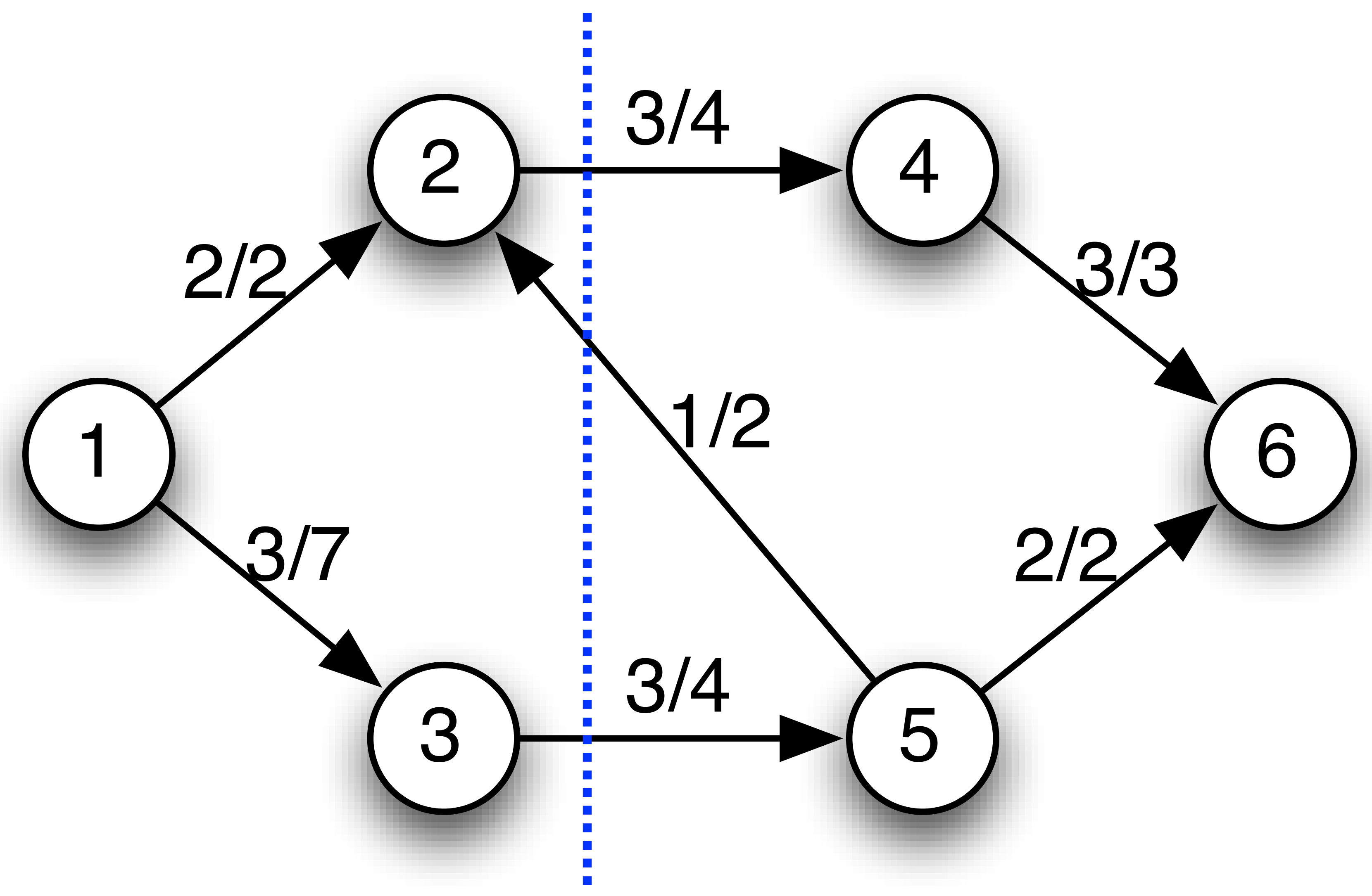
S = {1, 4}   T = {2, 3, 5, 6}

- What is c({1,4,}, {2, 3, 5, 6}) ?

Net flow of a cut

- The **net flow of a cut** (S, T) is the quantity of flow (positive or negative) from a node in S to a node in T.

$$f(S,T) = \sum_{a \in S, b \in T} f(a,b)$$



$$\begin{aligned} f(\{1,2,3\}, \{4,5,6\}) \\ &= f(2,4) + f(2,5) + f(3,5) \\ &= 3 - 1 + 3 \\ &= 5 \end{aligned}$$

Net flow of a cut

- **Theorem:** For every cut (S,T) and valid flow f, the value v(f) of the flow is equal to the net flow of the cut f(S,T).

- **Proof**:

$$\begin{aligned} v(f) &= \sum_{v \in V} f(s,v) \\ &= \sum_{v \in V} f(s,v) + \sum_{u \in S \setminus \{s\}} \sum_{v \in V} f(u,v) \qquad \text{(flow conservation in } u\text{)} \\ &= \sum_{v \in V} \left(f(s,v) + \sum_{u \in S \setminus \{s\}} f(u,v) \right) \\ &= \sum_{v \in V} \sum_{u \in S} f(u,v) \end{aligned}$$

$V = S \cup T$

Proof cont.

$$
\begin{aligned}
v(f) &= \sum_{v\in T}\sum_{u\in S} f(u,v) + \sum_{v\in S}\sum_{u\in S} f(u,v) \\
&= \sum_{v\in T}\sum_{u\in S} f(u,v) + \sum_{v\in S, u\in S, v<u} \left(f(u,v) + f(v,u)\right) \\
&= \sum_{v\in T}\sum_{u\in S} f(u,v) \\
&= f(S,T)
\end{aligned}
$$

And this is zero!

Max Flow, Min Cut Theorem (very famous)

- Given a feasible flow f, these properties are equivalent

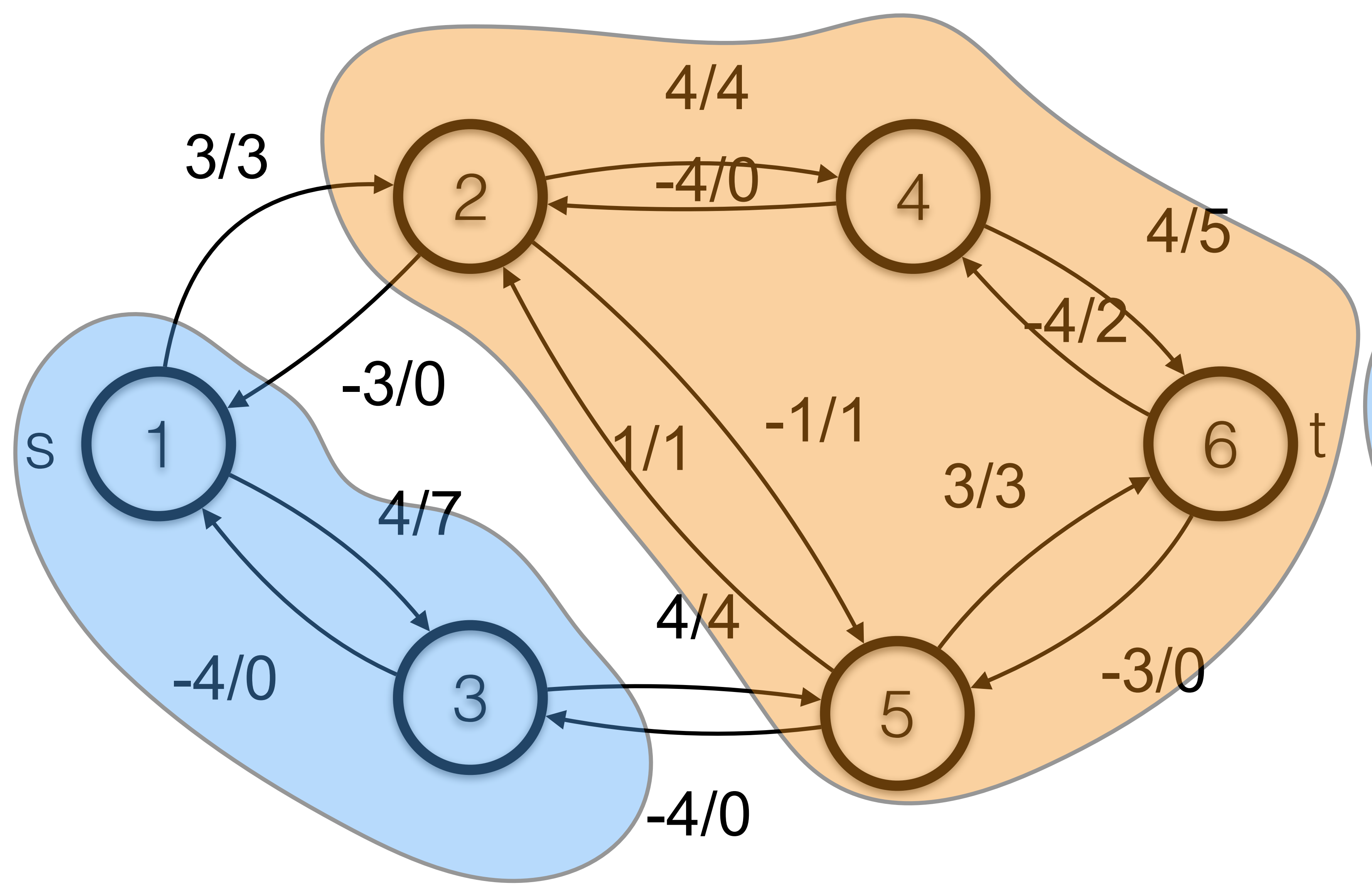
1. f is maximum flow in G;
2. The residual graph G_f has no augmenting path
3. There exists a cut (S, T) with capacity c(S, T) equal to v(f).

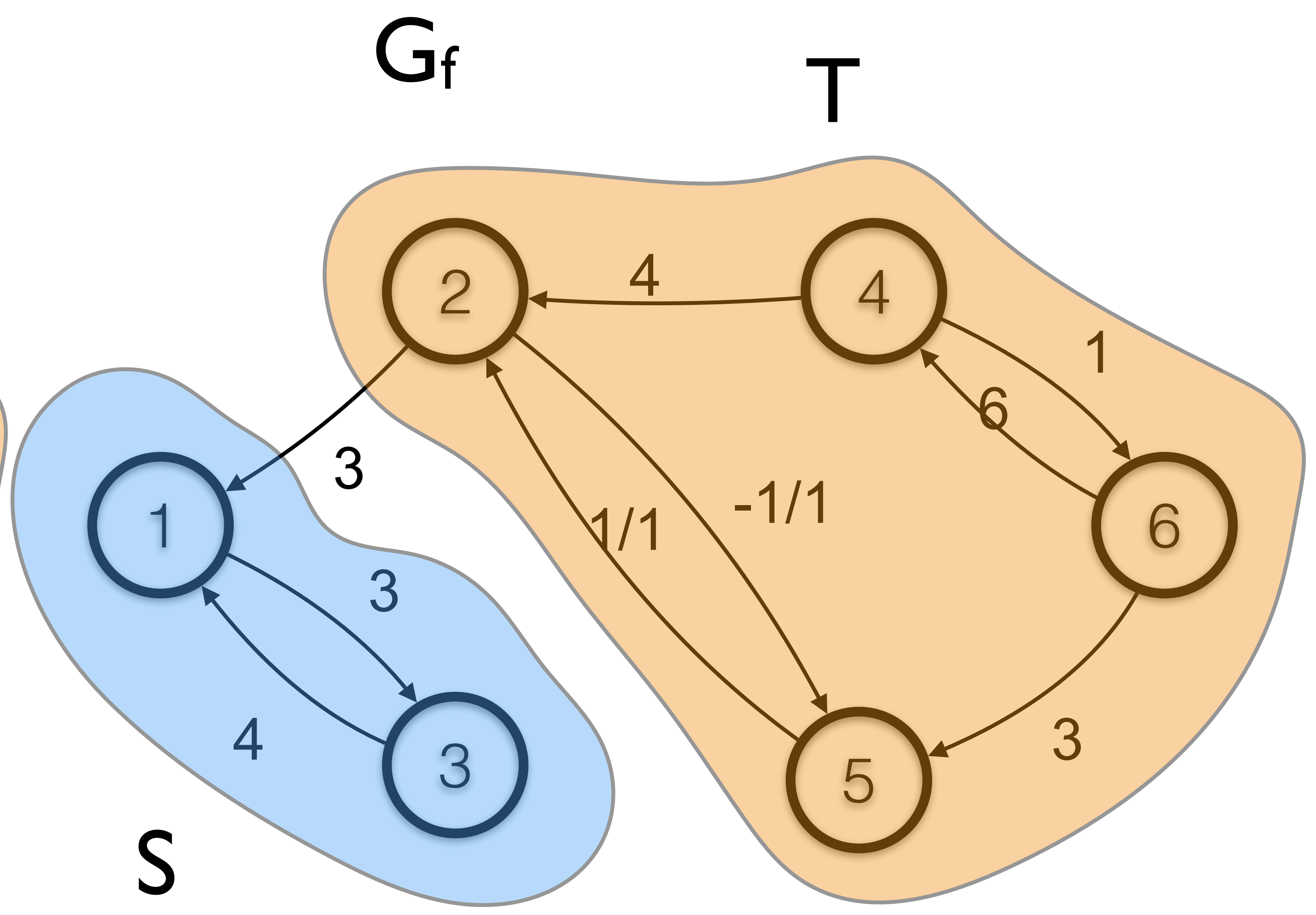
We proof 1=>2, 2 => 3 and 3 => 1

Proof Cont

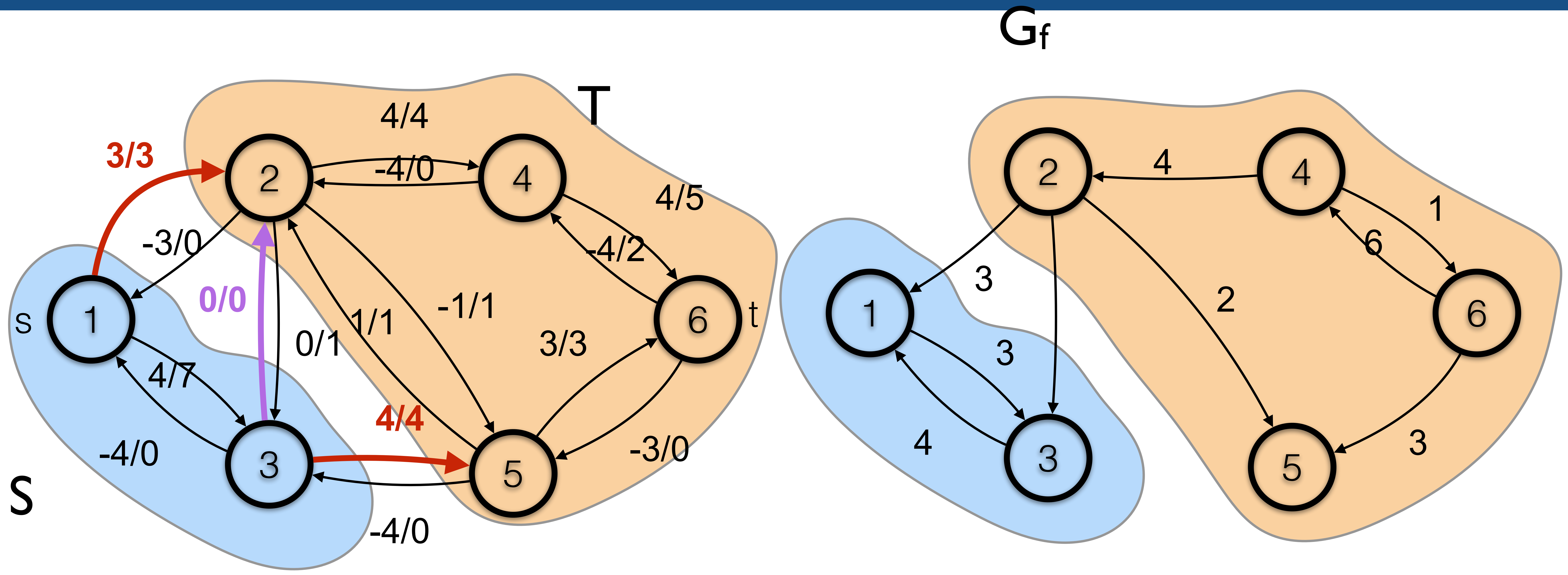
- (1 => 2) If this path would exists, we could build a larger flow (contradicts the fact that it is maximal).

- (2 => 3) Let S the set of nodes that can be reached from the source s in the residual graph G_f and let $T = V \setminus S$. Partition (S,T) is a cut since s is in S and t in T.





Proof Cont

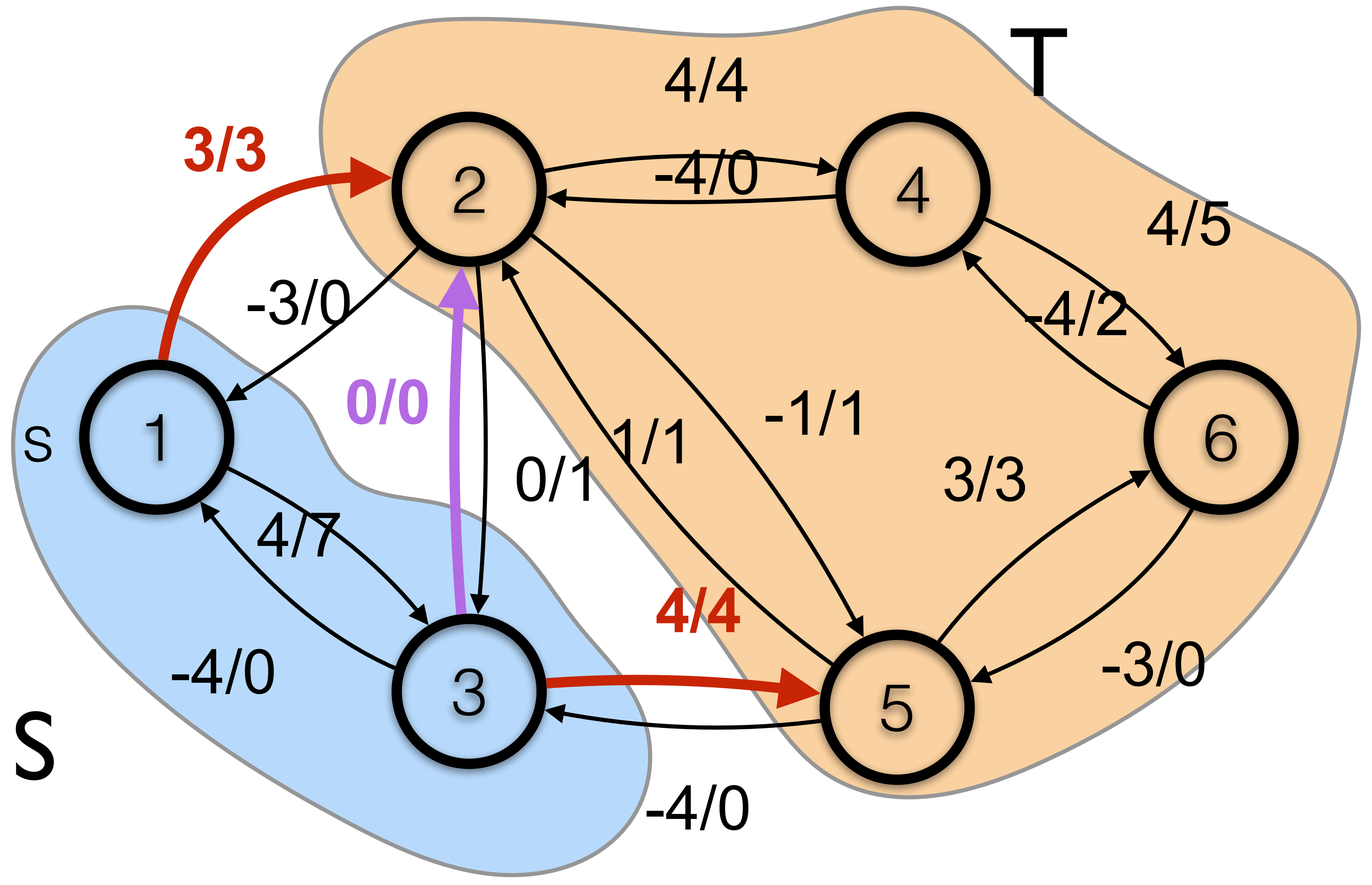


- Let two nodes u ∈ S and v ∈ T.
  - If (u, v) ∈ E then f(u, v) = c(u, v) otherwise s could reach v in G_f.
  - If (v, u) ∈ E then f(u, v) = 0 otherwise s could reach v in G_f.
  - Otherwise, f(u, v) = 0 since no flow can flow between two nodes not linked by an edge.

2 => 3 (Cont)

For every cut (S,T) and valid flow f, the value v(f) of the flow is equal to the net flow of the cut f(S,T).

$$
\begin{aligned}
v(f) &= f(S,T) \\
&= \sum_{a\in S}\sum_{b\in T} f(a,b) \\
&= \boxed{\sum_{(a,b)\in E \mid a\in S, b\in T} f(a,b)} + \boxed{\sum_{(b,a)\in E \mid a\in S, b\in T} f(a,b)} \\
&= \sum_{(a,b)\in E \mid a\in S, b\in T} c(a,b) + 0 \\
&= c(S,T)
\end{aligned}
$$



Proof Cont

- (3 => 1) The value of a flow cannot exceed the capacity of one of its cut.
- **Proof:**

For every cut (S,T) and valid flow f, the value v(f) of the flow is equal to the net flow of the cut f(S,T).

$$\begin{aligned} v(f) &= f(S,T) \\ &= \sum_{a \in S, b \in T} f(a,b) \\ &\leq \sum_{a \in S, b \in T} c(a,b) \quad \text{because of the capacity constraint} \\ &= c(S,T) \end{aligned}$$

- A flow with a value equal to the capacity of one of its cut is thus maximum.

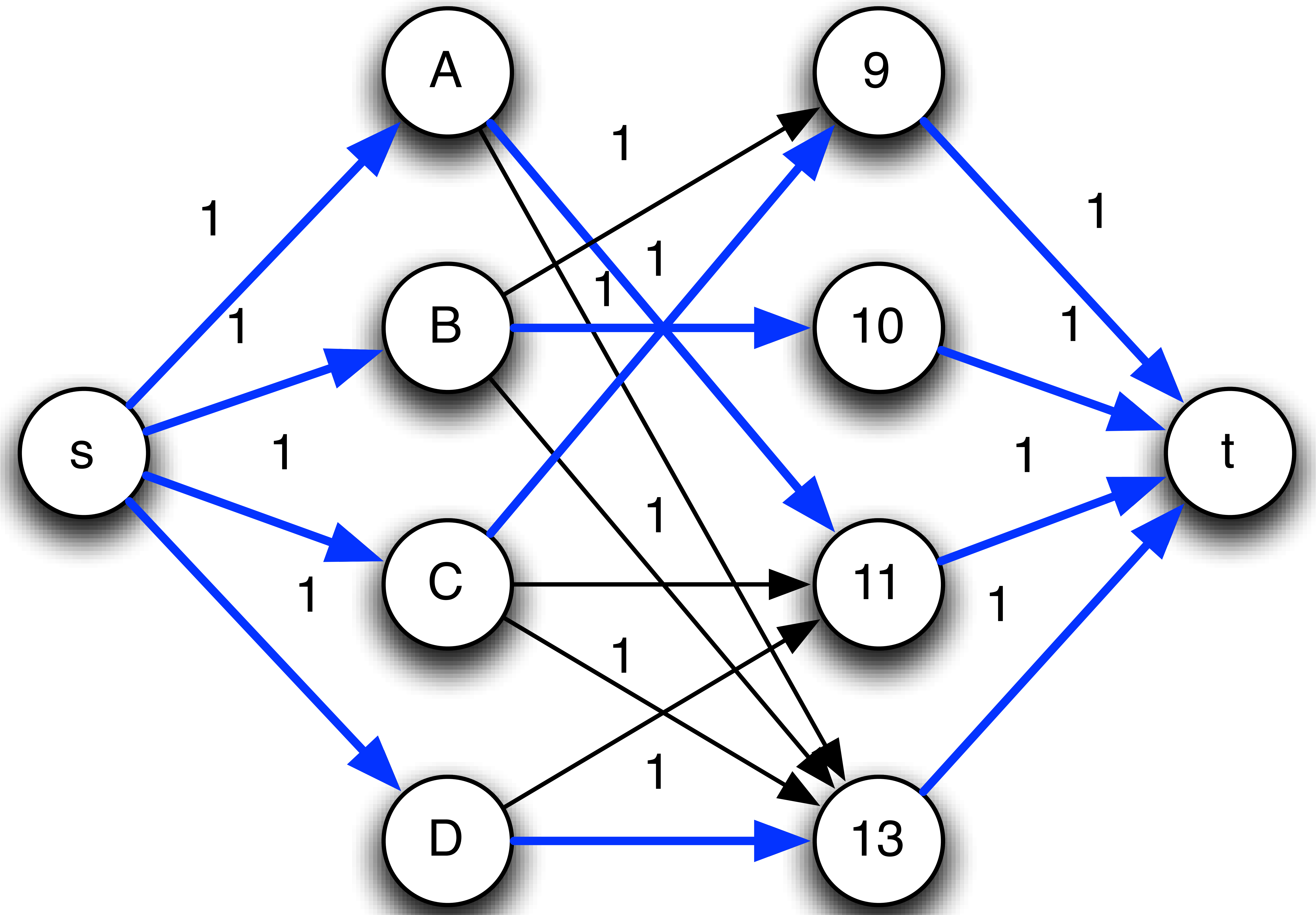
Exercise Scheduling Problem

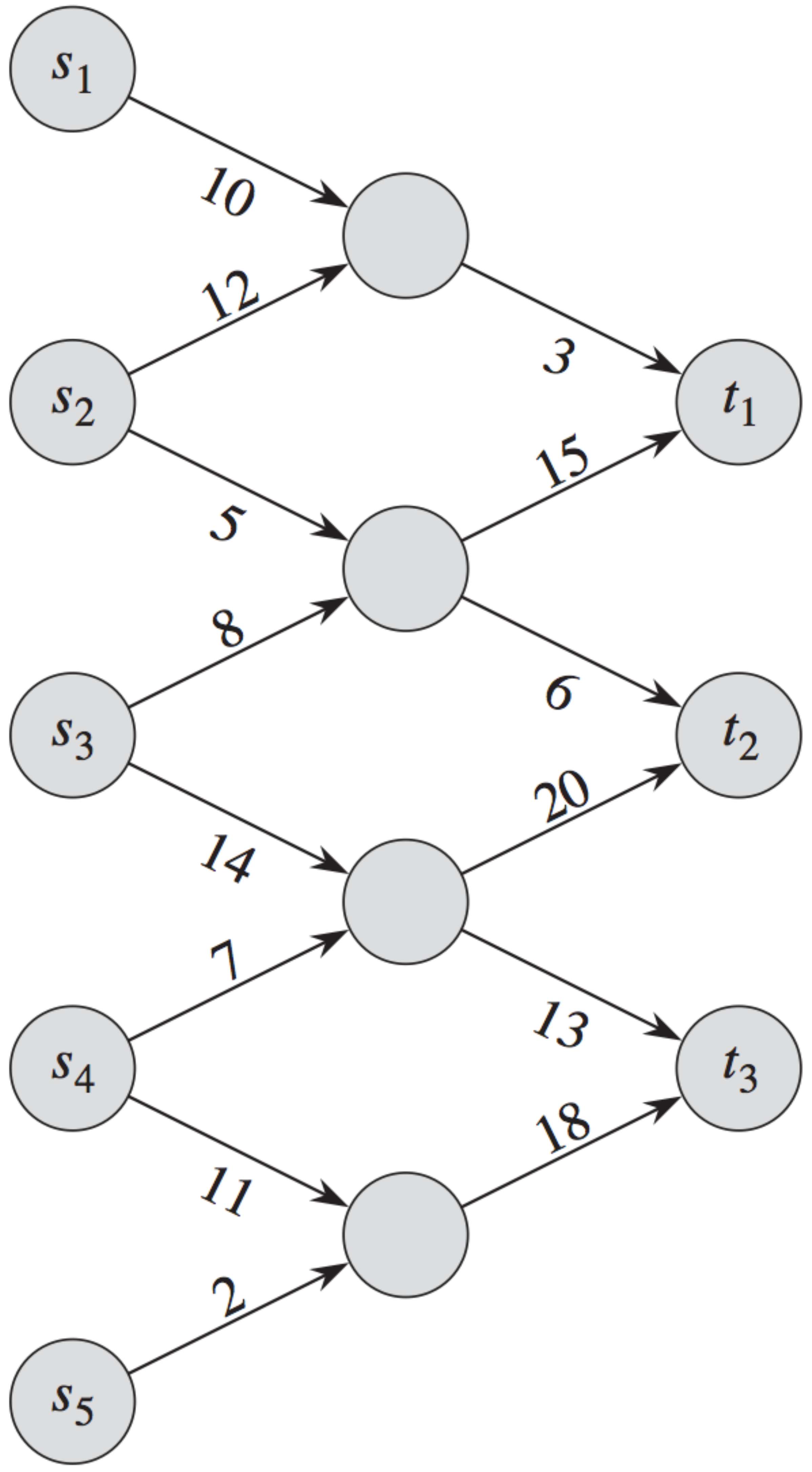
Person	Available
Alice	11h00 13h00
Benoît	9h00 10h00 13h00
Clotilde	9h00 11h00 13h00
Dany	11h00 13h00

- 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

Can you model this as a max flow problem?

Solution





The Lucky Puck Company has a set of m factories {s1,s2,…sm} and a set of n warehouses {t1,t2,…,tn}.

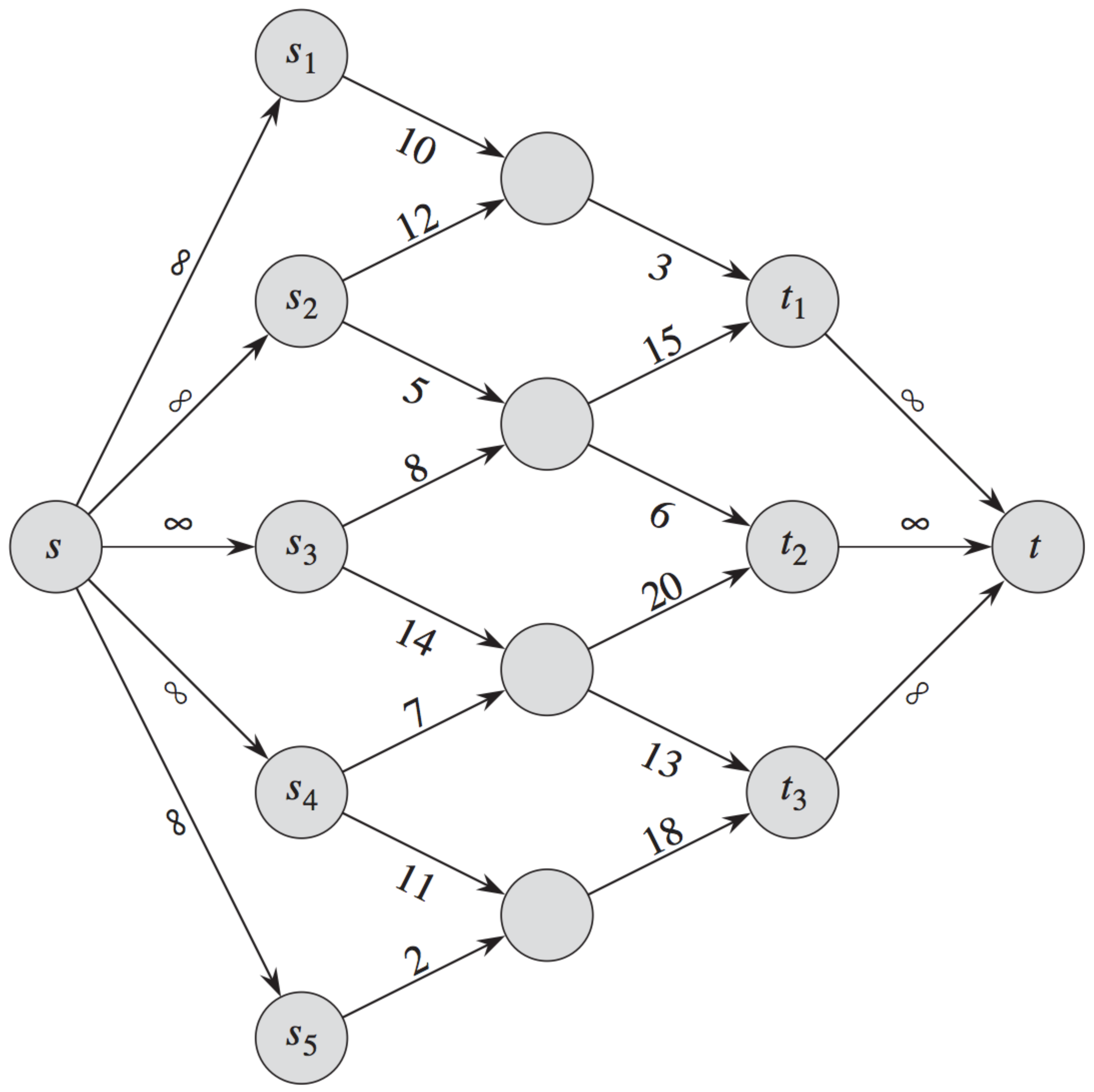
The Lucky Puck manufactures hockey pucks in the factories stocks them in the warehouses. **Lucky Puck leases space on trucks from another firm to ship the pucks from the factory to the warehouse. Because the trucks travel over specified routes (edges) between cities (vertices) and have a limited capacity, Lucky Puck can ship at most c(u,v) crates per day between each pair of cities u and v.**

Lucky Puck has no control over these routes and capacities, and so the company cannot alter the flow network

They need to determine the largest number p of crates per day that they can ship and then to produce this amount, since there is no point in producing more pucks than they can ship to their warehouses. Lucky Puck is not concerned with how long it takes for a given puck to get from a factory to the warehouses; they care only that p crates per day leave the factories and p crates per day arrive at the warehouses.

Can you model this as a max flow problem (one source, one sink)?

Solution



we can always reduce the problem of determining a maximum flow in a network with multiple sources and multiple sinks to an ordinary maximum-flow problem.

Ford-Fulkerson Time Complexity Revisited

- We have seen that the total complexity is O(v(f)|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 O(|V|.U) (by the max-flow min cut theorem)
- Ford-Fulkerson complexity can thus be expressed as

$$O(|V||E|U)$$

Is this polynomial in the size of the input?

No because the input takes log(U) bits to represent U

Ford-Fulkerson with Scaling

Idea: augment the flow along a path with sufficiently large residual capacity ($\geq \Delta$)

1. Let $G_{f,\Delta}$ be G_f with edges with residual capa $< \Delta$ filtered out. Then find augmenting paths in $G_{f,\Delta}$ until not possible with current value of Δ. Call this phase a Δ-*scaling phase.*

2. If $\Delta > 1$, divide Δ by 2 and go to step 1.

- Start with initial $\Delta = 2^{\lfloor \log U \rfloor}$
- There are at most O(log U) scaling phases.
- Question: How many augmenting paths can we discover at most in a Δ-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-S) is an s-t cut. By definition of S, residual capacity of the cut is c(S,V-S) $\leq |E|\Delta$.

- If f* is the optimal flow, then $v(f^*)-v(f) \leq |E|\Delta$.

- In the next Δ-scaling phase, each augmentation carries at least $\Delta/2$ units.

- So the next Δ-scaling phase can perform at most 2|E| augmentations: number of augmentations $\leq |E|\Delta/(\Delta/2)= 2|E|$

Ford-Fulkerson with scaling: Complexity

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

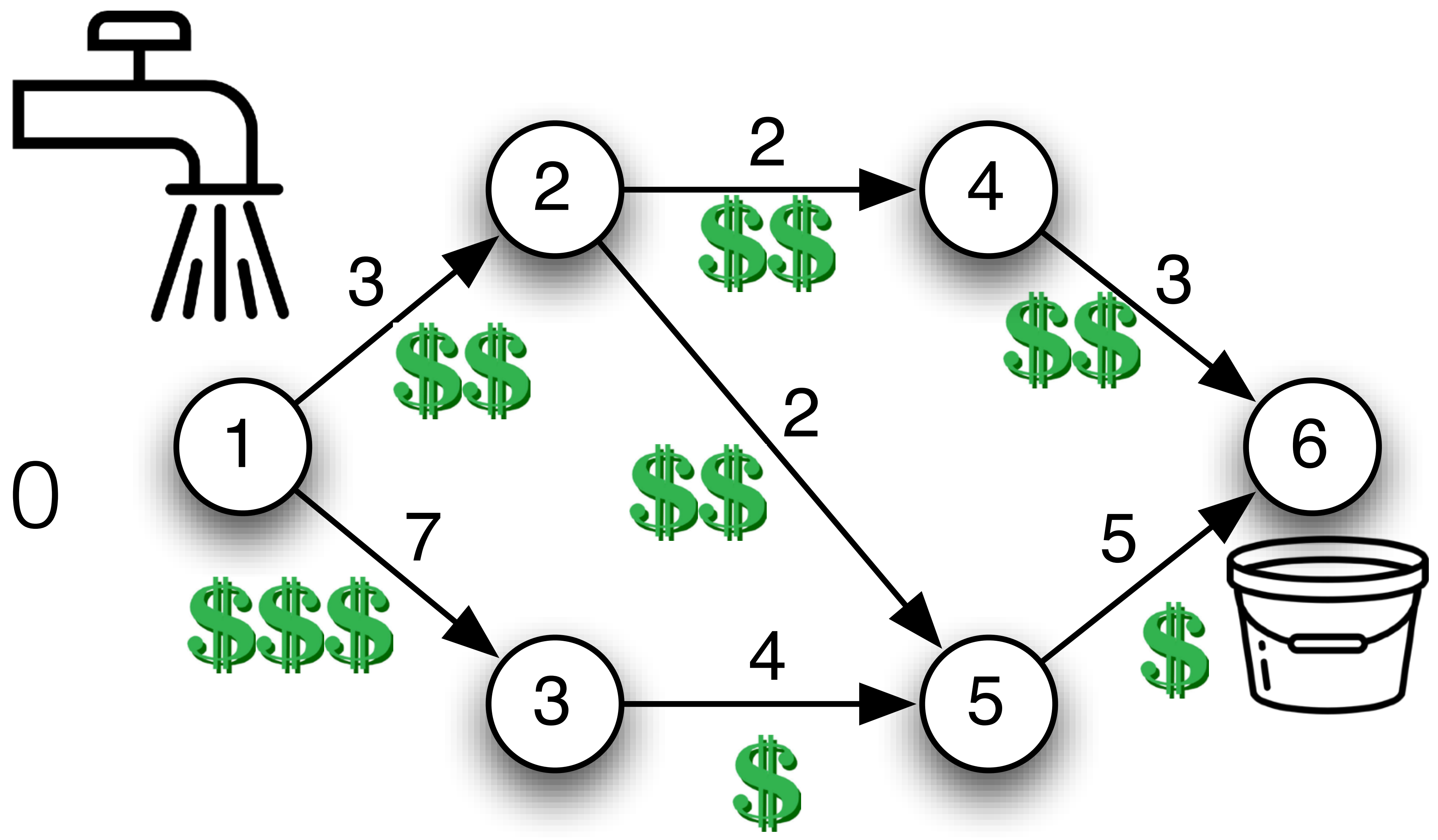
$$O(|E|^2 \log U)$$

Is this polynomial in the size of the input?

Yes it is!

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.

Flows, cut and Linear Programming

- Flows can be formulated as Integer Linear Programming problems.
- They have the particularity that if the capacities are integers, the optimal solution of the linear relaxation will be integer (totally unimodular matrix theory).
- The simplex algorithm that we have see can thus also be used to solve network flow problems (dedicated version exist only for flows: network simplex).
- Min Cut problem is the dual of the Max Flow problem (strong duality, same optimum objective)

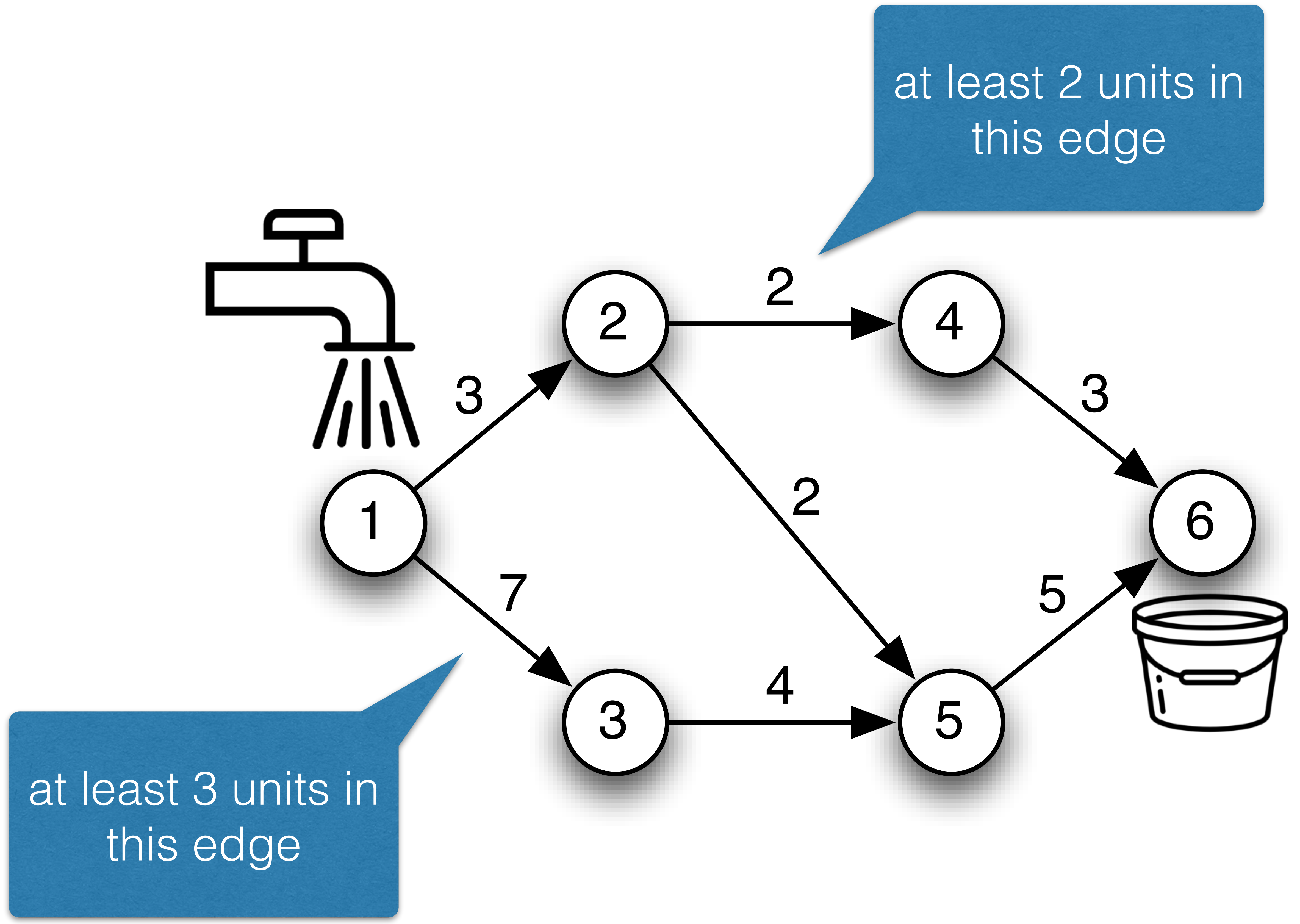
Bibliography

- « 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

Exerice

- How to find a maximum flow with minimum demand and capacity in some edges?



Hint: Try first to find a feasible flow, then maximize it

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