Chapter 5
Graph Algorithms

Algorithm Theory
WS 2013/14

Fabian Kuhn

Graphs

Extremely important concept in computer science

Graph $G = (V, E)$

- V: node (or vertex) set
- $E \subseteq V^2$: edge set
  - Simple graph: no self-loops, no multiple edges
  - Undirected graph: we often think of edges as sets of size 2 (e.g., $\{u, v\}$)
  - Directed graph: edges are sometimes also called arcs
  - Weighted graph: (positive) weight on edges (or nodes)
- (simple) path: sequence $v_0, \dots, v_k$ of nodes such that $(v_i, v_{i+1}) \in E$ for all $i \in \{0, \dots, k-1\}$
- …

Many real-world problems can be formulated as optimization problems on graphs

Graph Optimization: Examples

Minimum spanning tree (MST):

- Compute min. weight spanning tree of a weighted undir. Graph

Shortest paths:

- Compute (length) of shortest paths (single source, all pairs, …)

Traveling salesperson (TSP):

- Compute shortest TSP path/tour in weighted graph

Vertex coloring:

- Color the nodes such that neighbors get different colors
- Goal: minimize the number of colors

Maximum matching:

- Matching: set of pair-wise non-adjacent edges
- Goal: maximize the size of the matching

Network Flow

Flow Network:

- Directed graph $G = (V, E)$, $E \subseteq V^2$
- Each (directed) edge e has a capacity $c_e \geq 0$
  - Amount of flow (traffic) that the edge can carry
- A single source node $s \in V$ and a single sink node $t \in V$

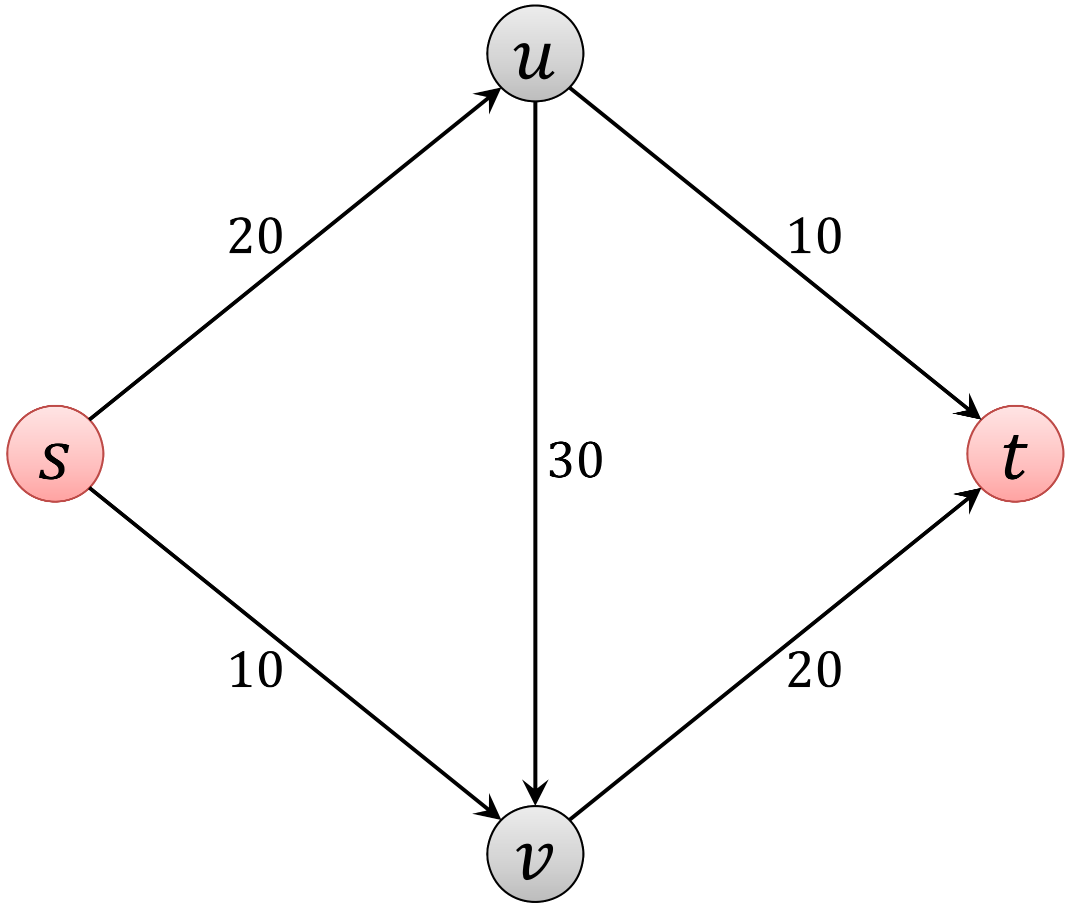
Flow: (informally)

- Traffic from s to t such that each edge carries at most its capacity

Examples:

- Highway system: edges are highways, flow is the traffic
- Computer network: edges are network links that can carry packets, nodes are switches
- Fluid network: edges are pipes that carry liquid

Example: Flow Network



Network Flow: Definition

Flow: function $\boldsymbol{f}: \boldsymbol{E} \rightarrow \mathbb{R}_{\geq 0}$

- $f(e)$ is the amount of flow carried by edge e

Capacity Constraints:

- For each edge $e \in E$, $f(e) \leq c_e$

Flow Conservation:

- For each node $v \in V \setminus \{s, t\}$,

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

Flow Value:

$$|f| := \sum_{e \text{ out of } s} f\big((s,u)\big) = \sum_{e \text{ into } t} f\big((v,t)\big)$$

Notation

We define:

$$f^{\text{in}}(v) \coloneqq \sum_{e \text{ into } v} f(e), \qquad f^{\text{out}}(v) \coloneqq \sum_{e \text{ out of } v} f(e)$$

For a set $S \subseteq V$:

$$f^{\text{in}}(S) \coloneqq \sum_{e \text{ into } S} f(e), \qquad f^{\text{out}}(S) \coloneqq \sum_{e \text{ out of } S} f(e)$$

Flow conservation: $\forall v \in V \setminus \{s, t\}$: $f^{\text{in}}(v) = f^{\text{out}}(v)$

Flow value: $|f| = f^{\text{out}}(s) = f^{\text{in}}(t)$

For simplicity: Assume that all capacities are positive integers

The Maximum-Flow Problem

Maximum Flow:

Given a flow network, find a flow of maximum possible value

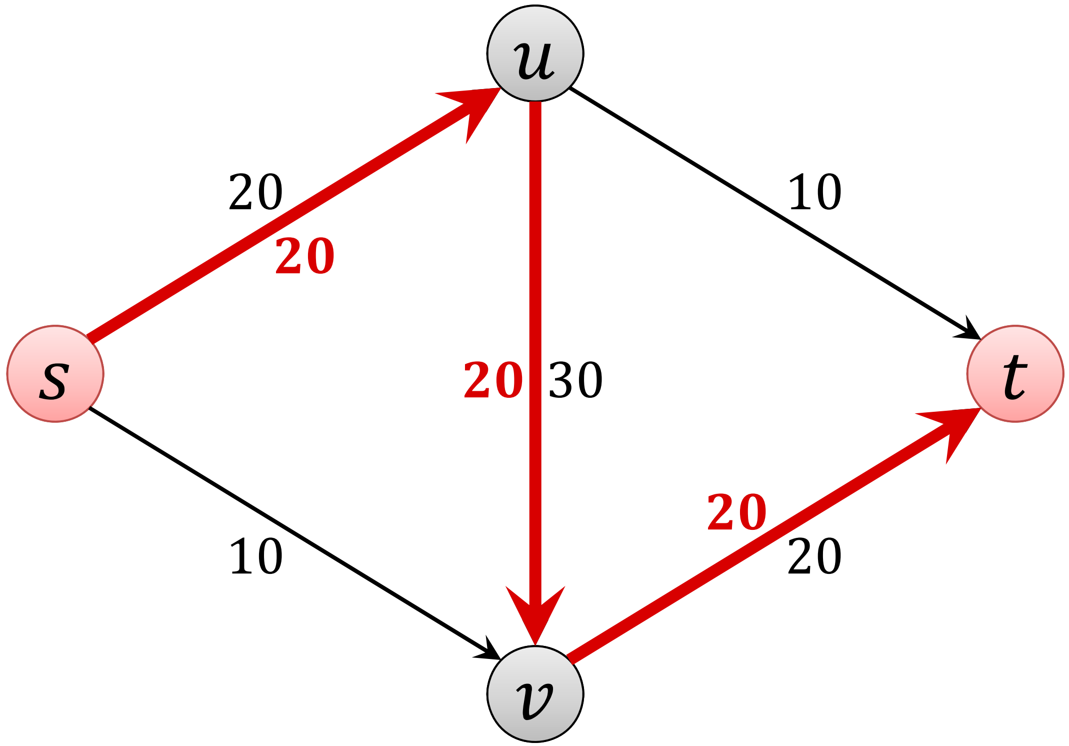
- Classical graph optimization problem
- Many applications (also beyond the obvious ones)
- Requires new algorithmic techniques

Maximum Flow: Greedy?

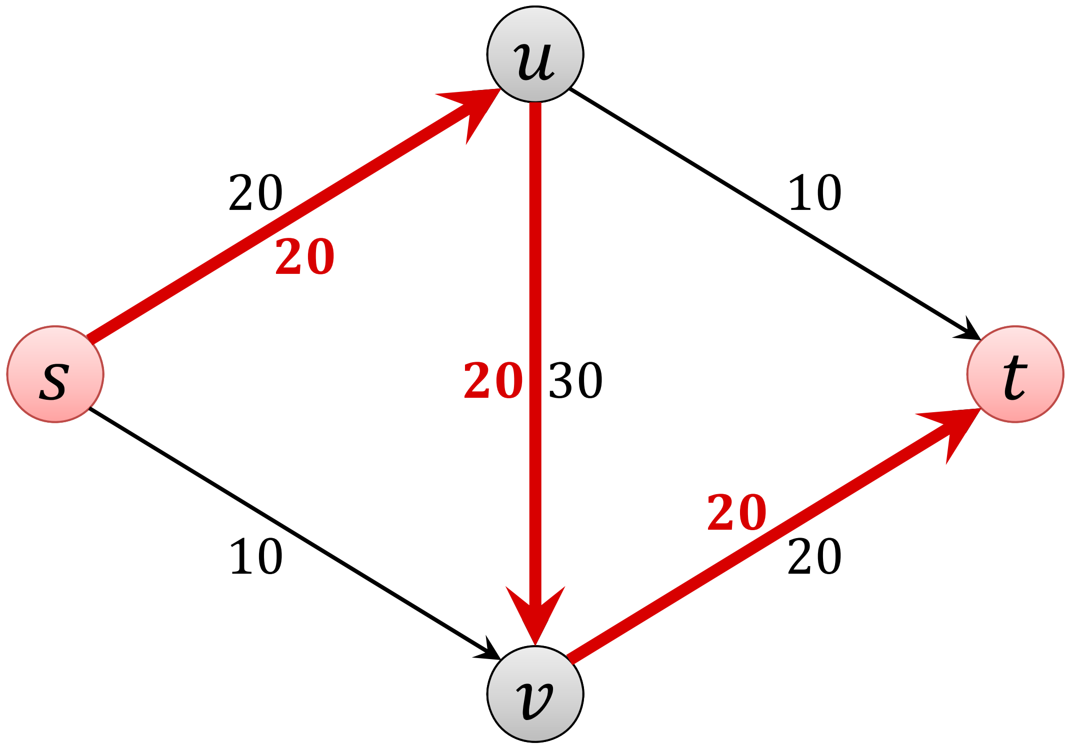
Does greedy work?

A natural greedy algorithm:

- As long as possible, find an s-t-path with free capacity and add as much flow as possible to the path



Improving the Greedy Solution



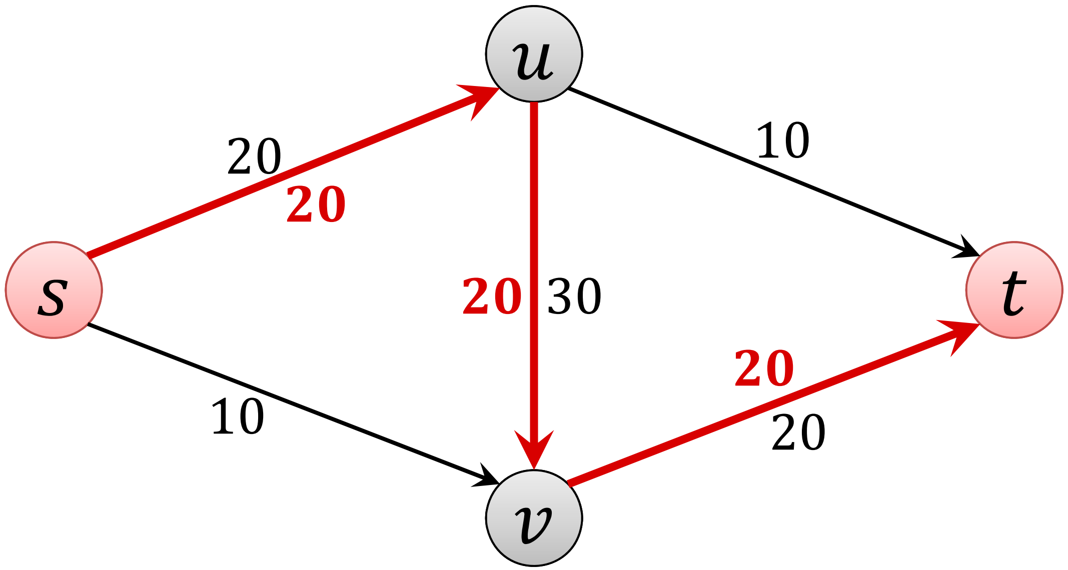
- Try to push 10 units of flow on edge (s, v)
- Too much incoming flow at v: reduce flow on edge (u, v)
- Add that flow on edge (u, t)

Residual Graph

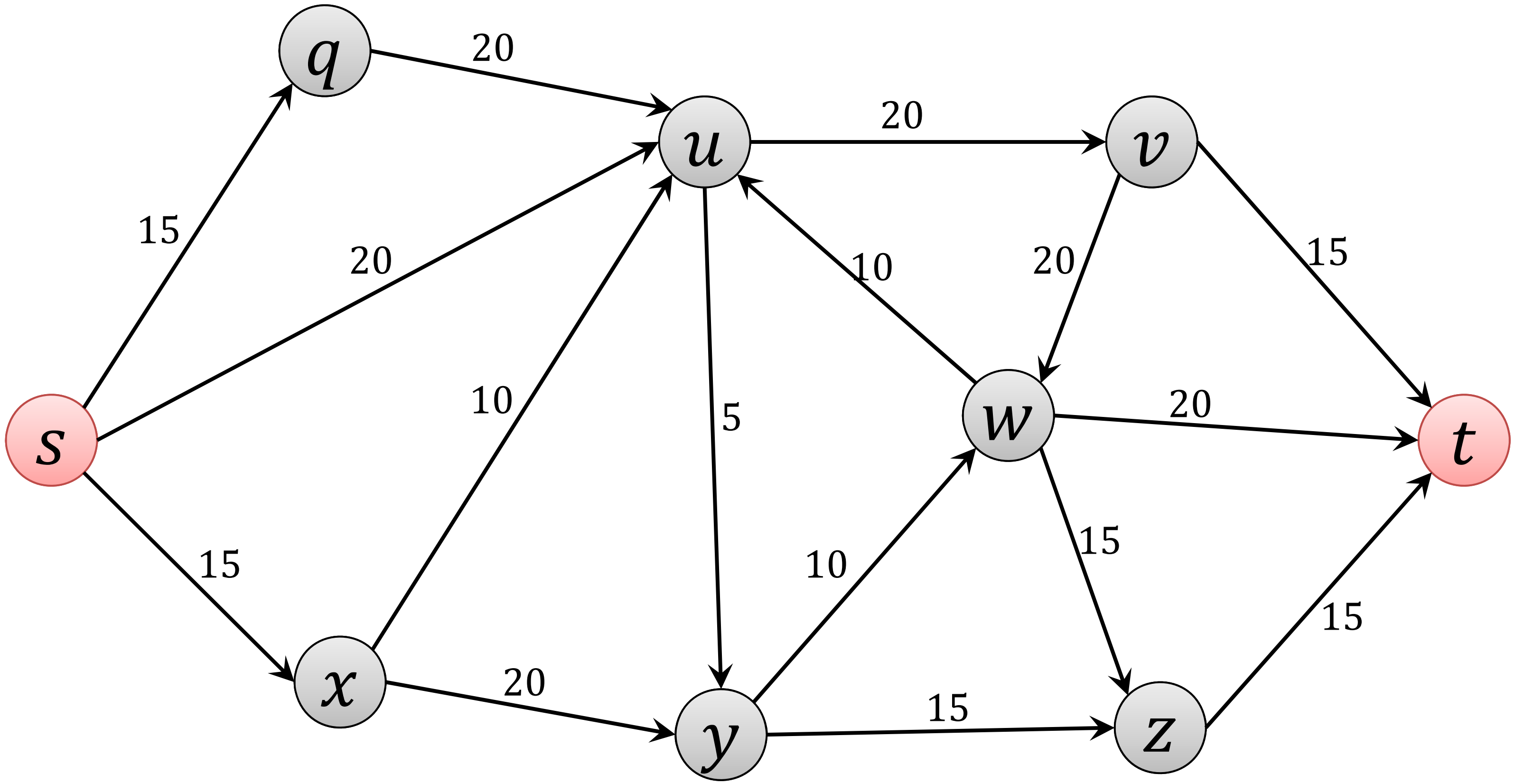
Given a flow network $G = (V, E)$ with capacities c_e (for $e \in E$)

For a flow f on G, define directed graph $G_f = (V_f, E_f)$ as follows:

- Node set $V_f = V$
- For each edge $e = (u, v)$ in E, there are two edges in E_f:
  - forward edge $e = (u, v)$ with residual capacity $c_e - f(e)$
  - backward edge $e' = (v, u)$ with residual capacity $f(e)$

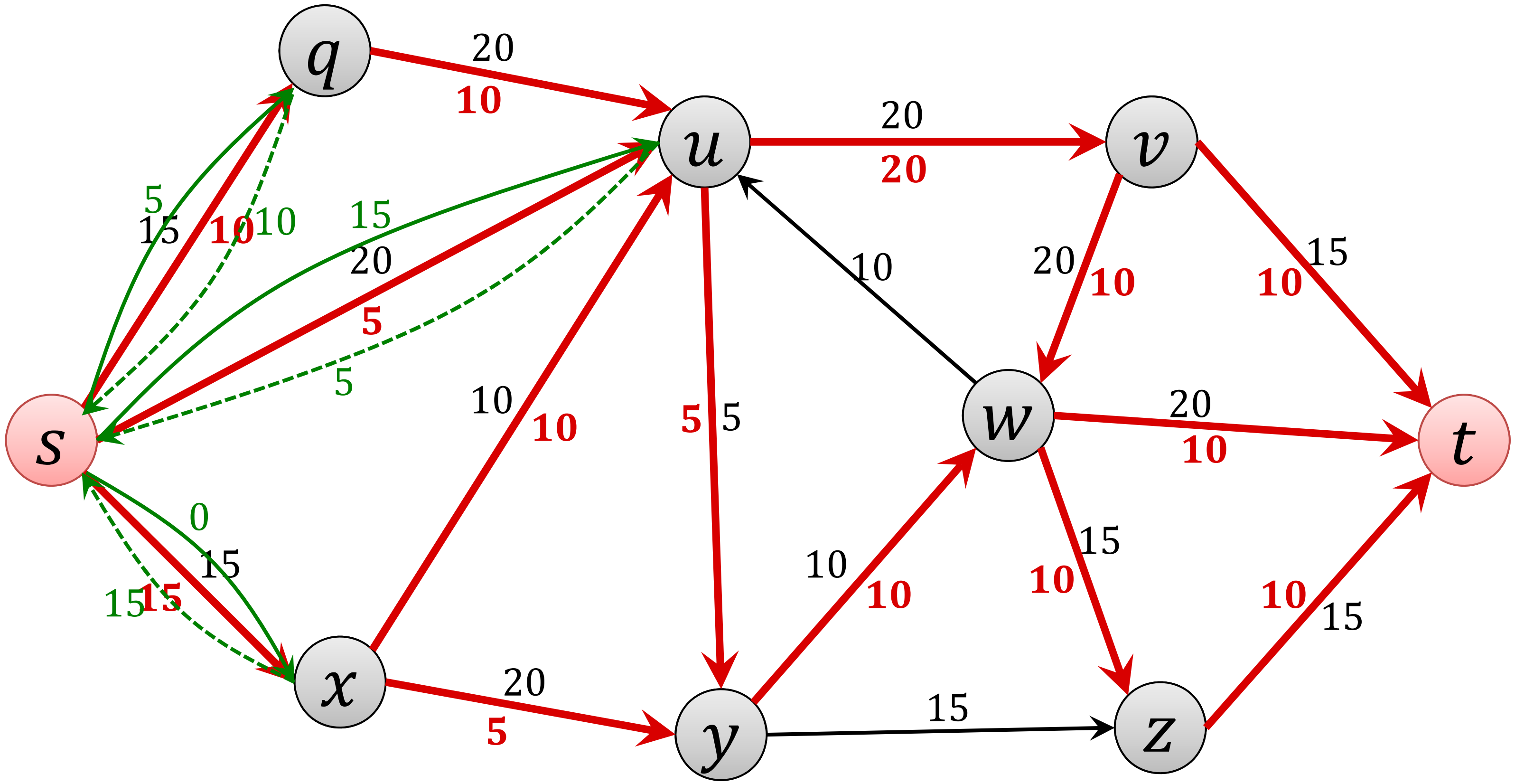


Residual Graph: Example



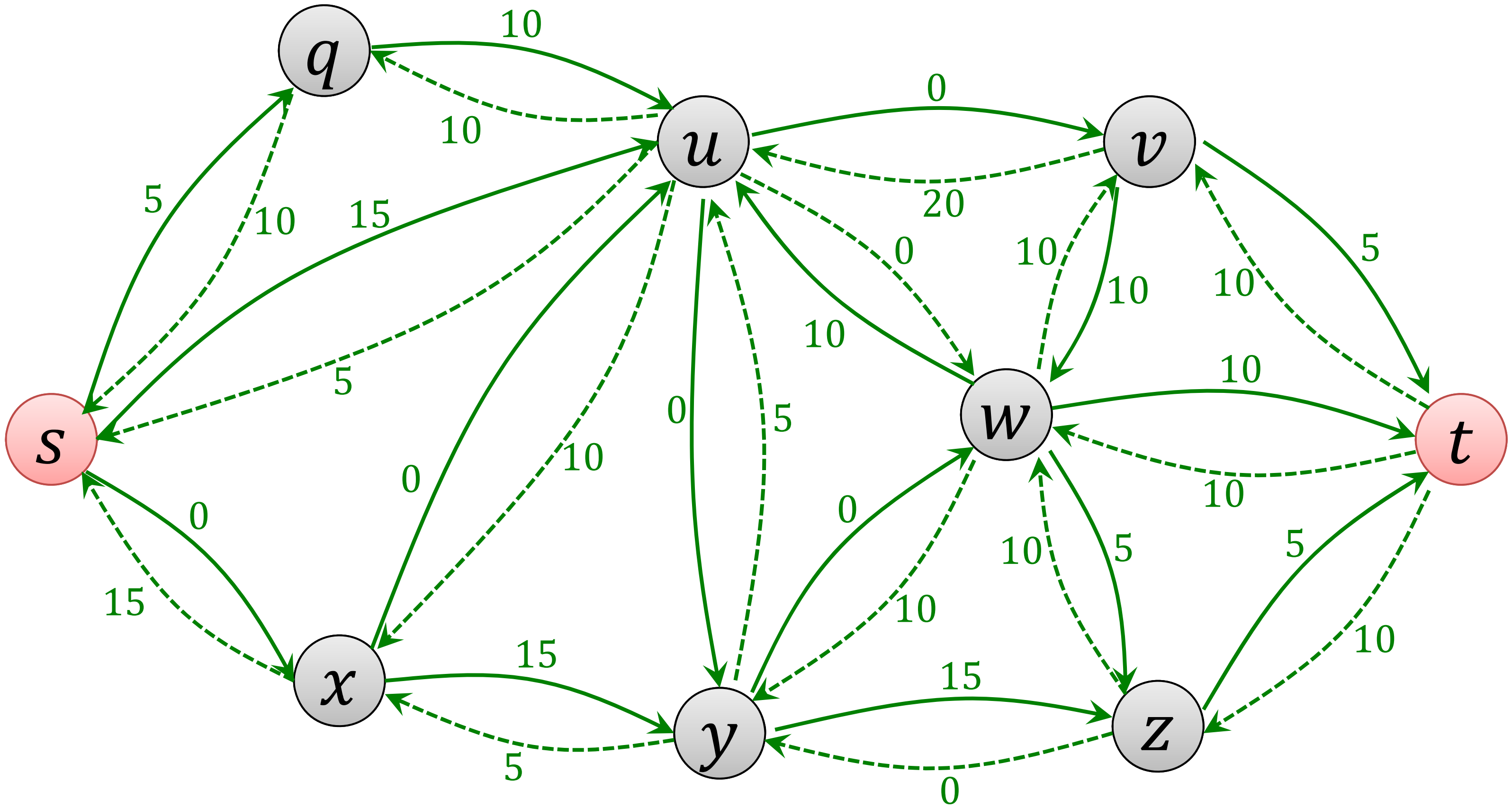
Residual Graph: Example

Flow f



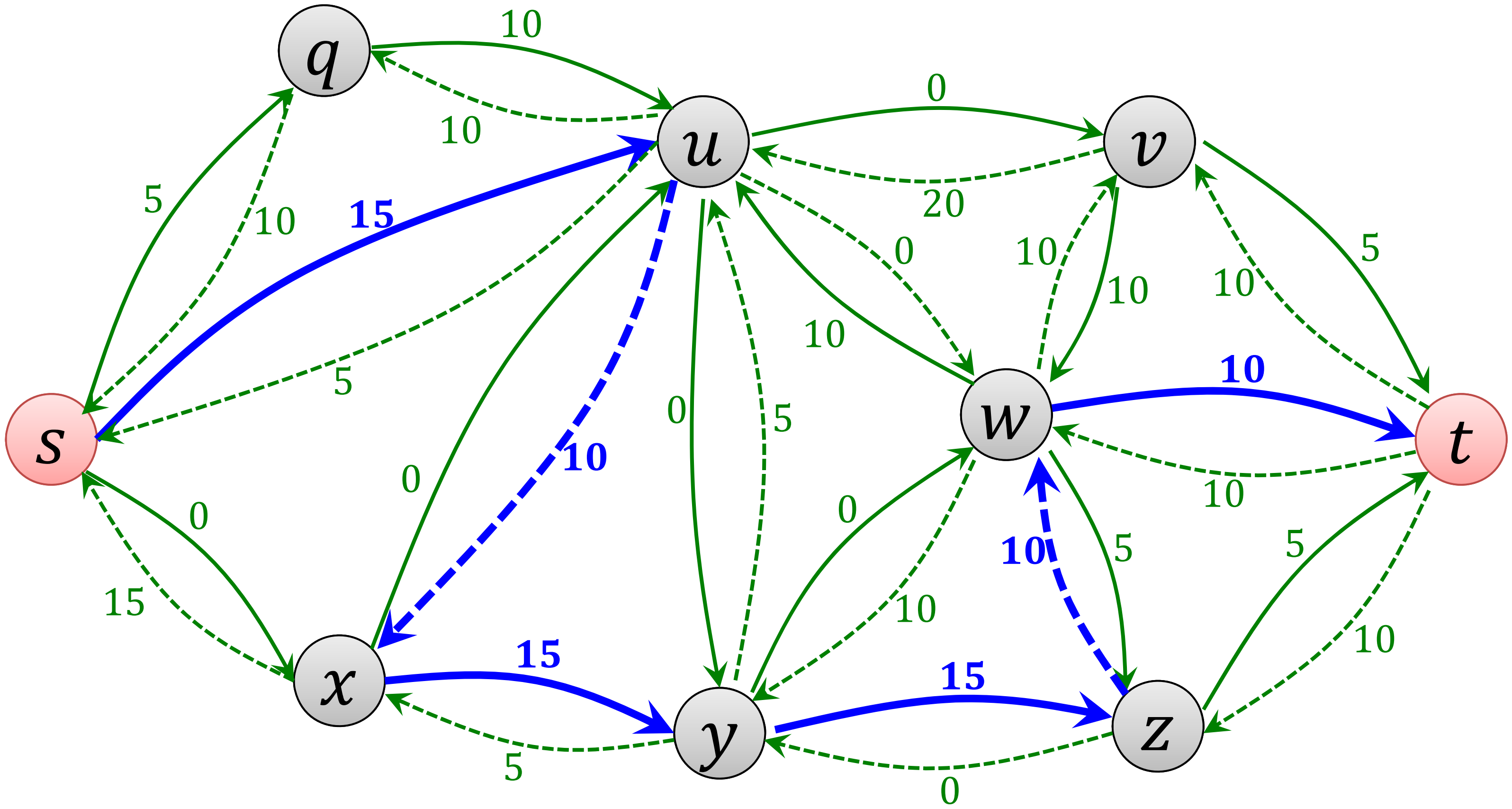
Residual Graph: Example

Residual Graph G_f



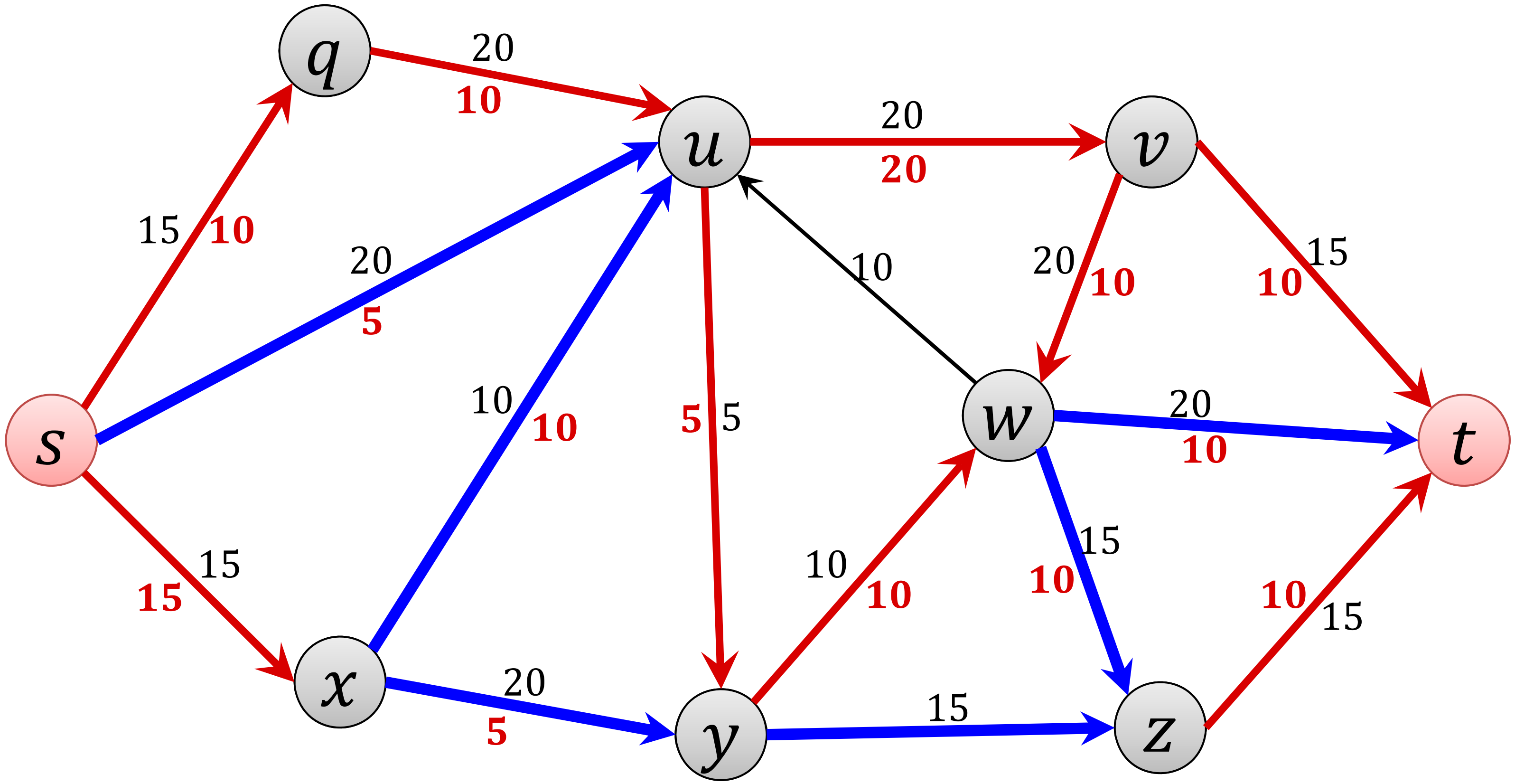
Augmenting Path

Residual Graph G_f



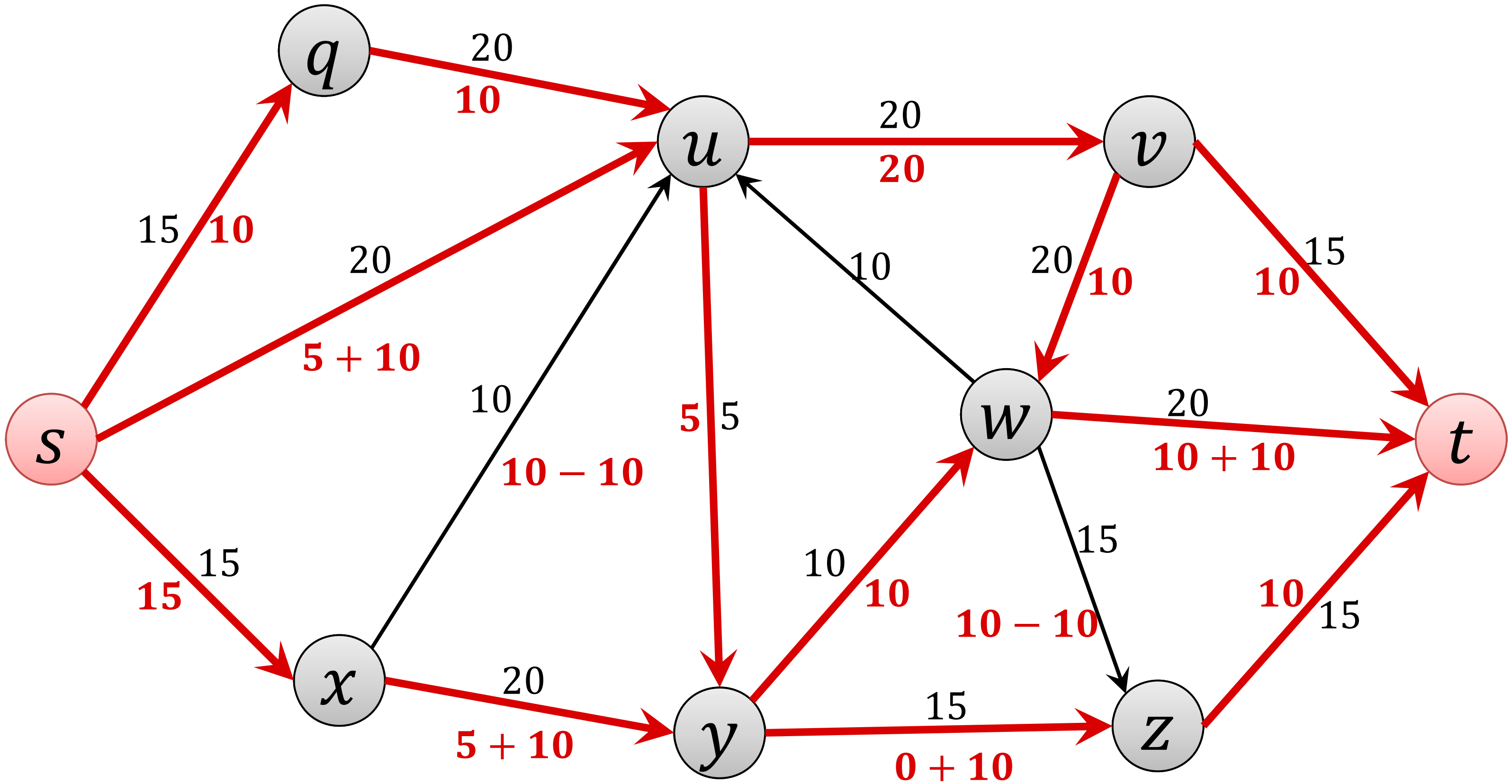
Augmenting Path

Augmenting Path



Augmenting Path

New Flow



Augmenting Path

Definition:
An augmenting path P is a (simple) s-t-path on the residual graph G_f on which each edge has residual capacity > 0.

$\text{bottleneck}(P, f)$: minimum residual capacity on any edge of the augmenting path P

Augment flow f to get flow f':

- For every forward edge (u, v) on P:

$$f'\big((u,v)\big) := f\big((u,v)\big) + \text{bottleneck}(P, f)$$

- For every backward edge (u, v) on P:

$$f'\big((v,u)\big) := f\big((v,u)\big) - \text{bottleneck}(P, f)$$

Augmented Flow

Lemma: Given a flow f and an augmenting path P, the resulting augmented flow f' is legal and its value is

$$|\boldsymbol{f}'| = |\boldsymbol{f}| + \mathbf{bottleneck}(\boldsymbol{P}, \boldsymbol{f}).$$

Proof:

Augmented Flow

Lemma: Given a flow f and an augmenting path P, the resulting augmented flow f' is legal and its value is

$$|\boldsymbol{f}'| = |\boldsymbol{f}| + \mathbf{bottleneck}(\boldsymbol{P}, \boldsymbol{f}).$$

Proof:

Ford-Fulkerson Algorithm

- Improve flow using an augmenting path as long as possible:

1. Initially, $f(e) = 0$ for all edges $e \in E$, $G_f = G$
2. **while** there is an augmenting s-t-path P in G_f **do**
3. Let P be an augmenting s-t-path in G_f;
4. $f' \coloneqq \text{augment}(f, P)$;
5. update f to be f';
6. update the residual graph G_f
7. **end**;

Ford-Fulkerson Running Time

Theorem: If all edge capacities are integers, the Ford-Fulkerson algorithm terminates after at most C iterations, where

$$C = \sum_{e \text{ out of } s} c_e .$$

Proof:

Ford-Fulkerson Running Time

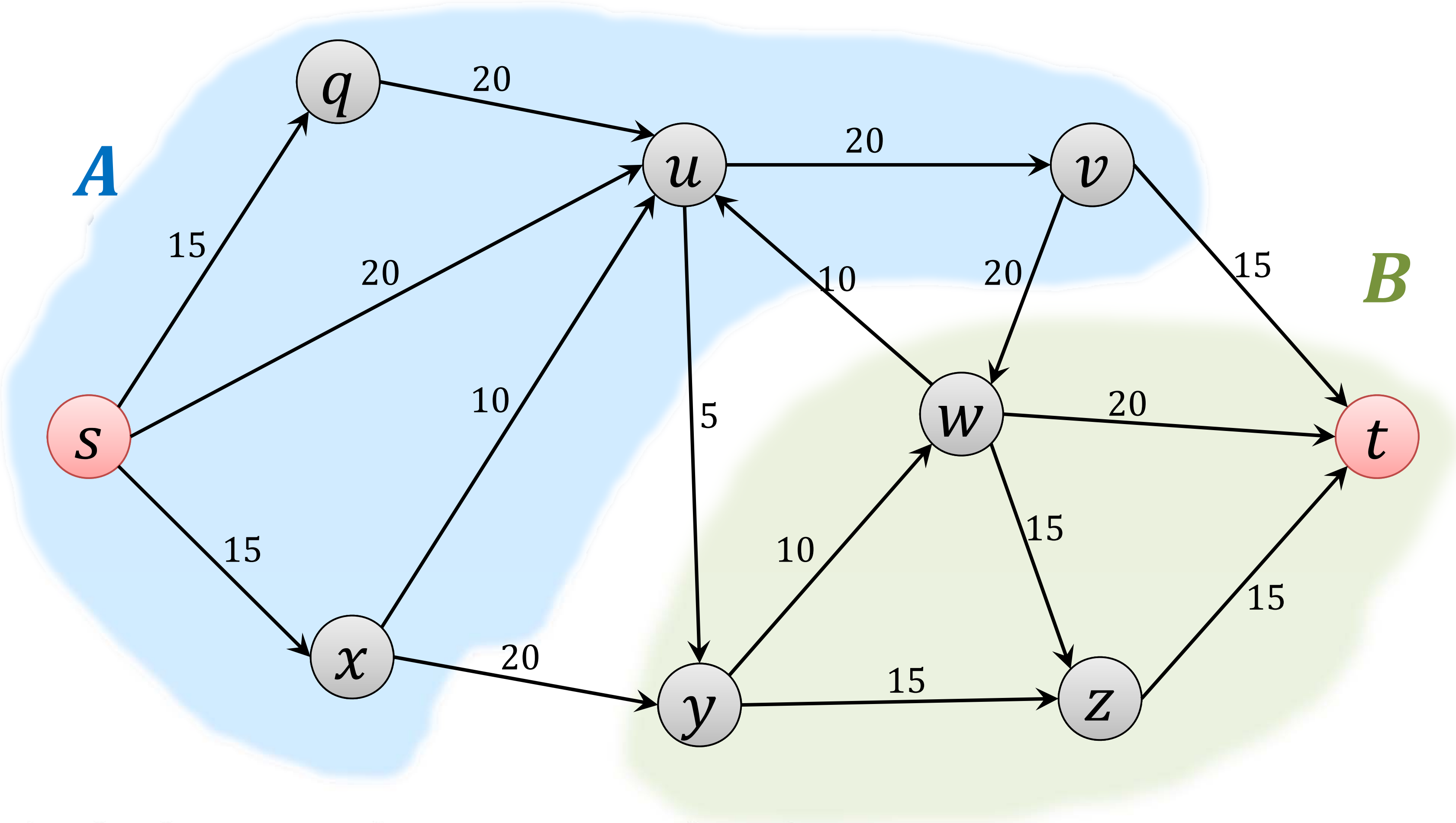
Theorem: If all edge capacities are integers, the Ford-Fulkerson algorithm can be implemented to run in $O(mC)$ time.

Proof:

s-t Cuts

Definition:

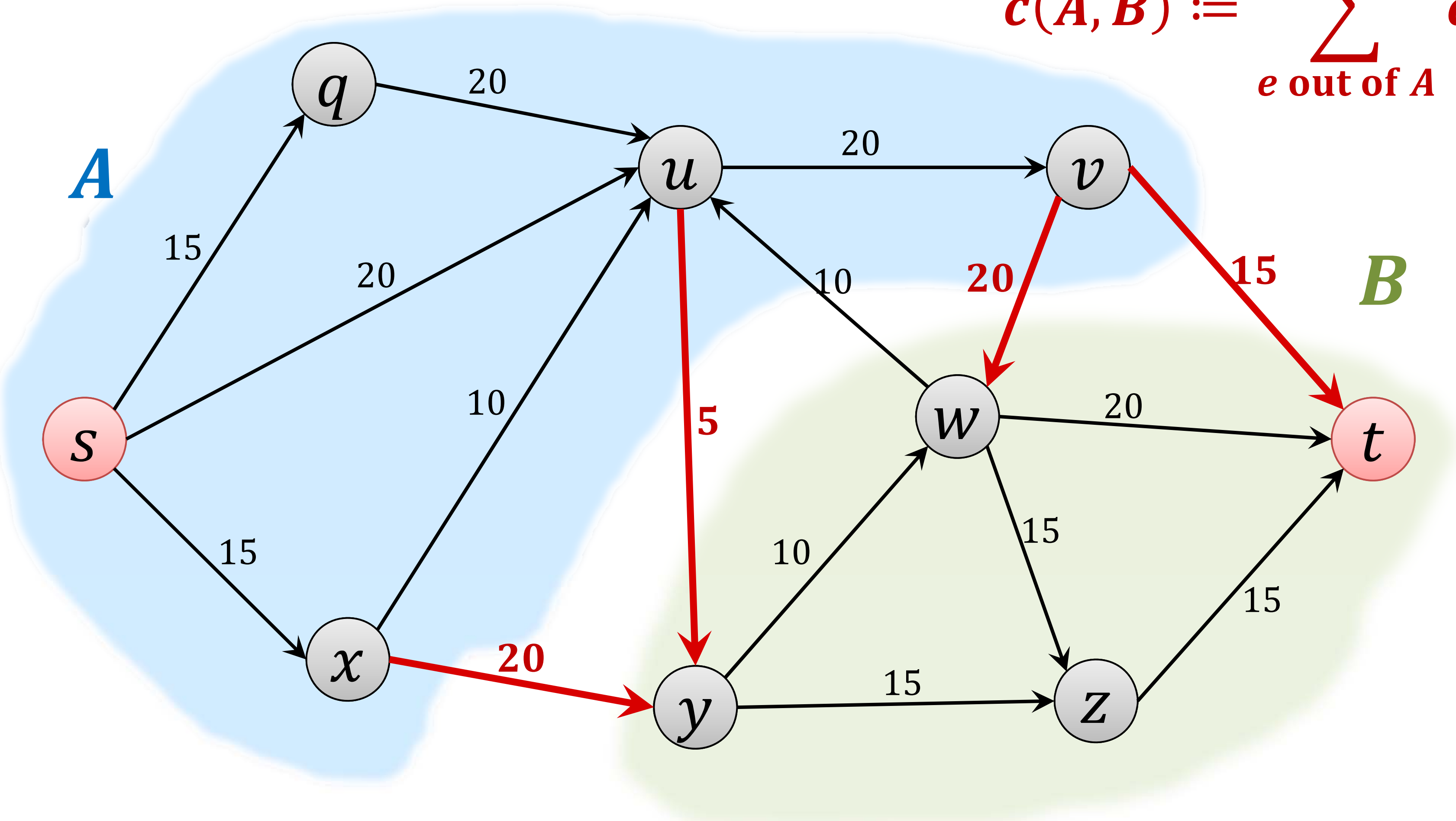
An s-t cut is a partition (A, B) of the vertex set such that $s \in A$ and $t \in B$



Cut Capacity

Definition:
The capacity $c(A,B)$ of an s-t-cut (A,B) is defined as

$$c(A,B) := \sum_{e \text{ out of } A} c_e .$$



Cuts and Flow Value

Lemma: Let f be any s-t flow, and (A, B) any s-t cut. Then,

$$|\boldsymbol{f}| = \boldsymbol{f}^{\mathbf{out}}(\boldsymbol{A}) - \boldsymbol{f}^{\mathbf{in}}(\boldsymbol{A}).$$

Proof:

Cuts and Flow Value

Lemma: Let f be any s-t flow, and (A, B) any s-t cut. Then,

$$|\boldsymbol{f}| = \boldsymbol{f}^{\mathbf{out}}(\boldsymbol{A}) - \boldsymbol{f}^{\mathbf{in}}(\boldsymbol{A}).$$

Lemma: Let f be any s-t flow, and (A, B) any s-t cut. Then,

$$|\boldsymbol{f}| = \boldsymbol{f}^{\mathbf{in}}(\boldsymbol{B}) - \boldsymbol{f}^{\mathbf{out}}(\boldsymbol{B}).$$

Proof:

Upper Bound on Flow Value

Lemma:

Let f be any s-t flow and (A, B) and s-t cut. Then $\boldsymbol{|f| \leq c(A, B)}$.

Proof:

Ford-Fulkerson Gives Optimal Solution

Lemma: If f is an s-t flow such that there is no augmenting path in G_f, then there is an s-t cut (A^*, B^*) in G for which

$$|\boldsymbol{f}| = \boldsymbol{c}(\boldsymbol{A}^*, \boldsymbol{B}^*).$$

Proof:

- Define $\boldsymbol{A}^*$: set of nodes that can be reached from s on a path with positive residual capacities in G_f:

- For $B^* = V \setminus A^*$, (A^*, B^*) is an s-t cut
  - By definition $s \in A^*$ and $t \notin A^*$

Ford-Fulkerson Gives Optimal Solution

Lemma: If f is an s-t flow such that there is no augmenting path in G_f, then there is an s-t cut (A^*, B^*) in G for which

$$|\boldsymbol{f}| = \boldsymbol{c}(\boldsymbol{A}^*, \boldsymbol{B}^*).$$

Proof:

Ford-Fulkerson Gives Optimal Solution

Lemma: If f is an s-t flow such that there is no augmenting path in G_f, then there is an s-t cut (A^*, B^*) in G for which

$$|\boldsymbol{f}| = \boldsymbol{c}(\boldsymbol{A}^*, \boldsymbol{B}^*).$$

Proof:

Ford-Fulkerson Gives Optimal Solution

Theorem: The flow returned by the Ford-Fulkerson algorithm is a maximum flow.

Proof:

Min-Cut Algorithm

Ford-Fulkerson also gives a min-cut algorithm:

Theorem: Given a flow f of maximum value, we can compute an s-t cut of minimum capacity in $O(m)$ time.

Proof:

Max-Flow Min-Cut Theorem

Theorem: (Max-Flow Min-Cut Theorem)

In every flow network, the maximum value of an s-t flow is equal to the minimum capacity of an s-t cut.

Proof:

Integer Capacities

Theorem: (Integer-Valued Flows)

If all capacities in the flow network are integers, then there is a maximum flow f for which the flow $f(e)$ of every edge e is an integer.

Proof:

Non-Integer Capacities

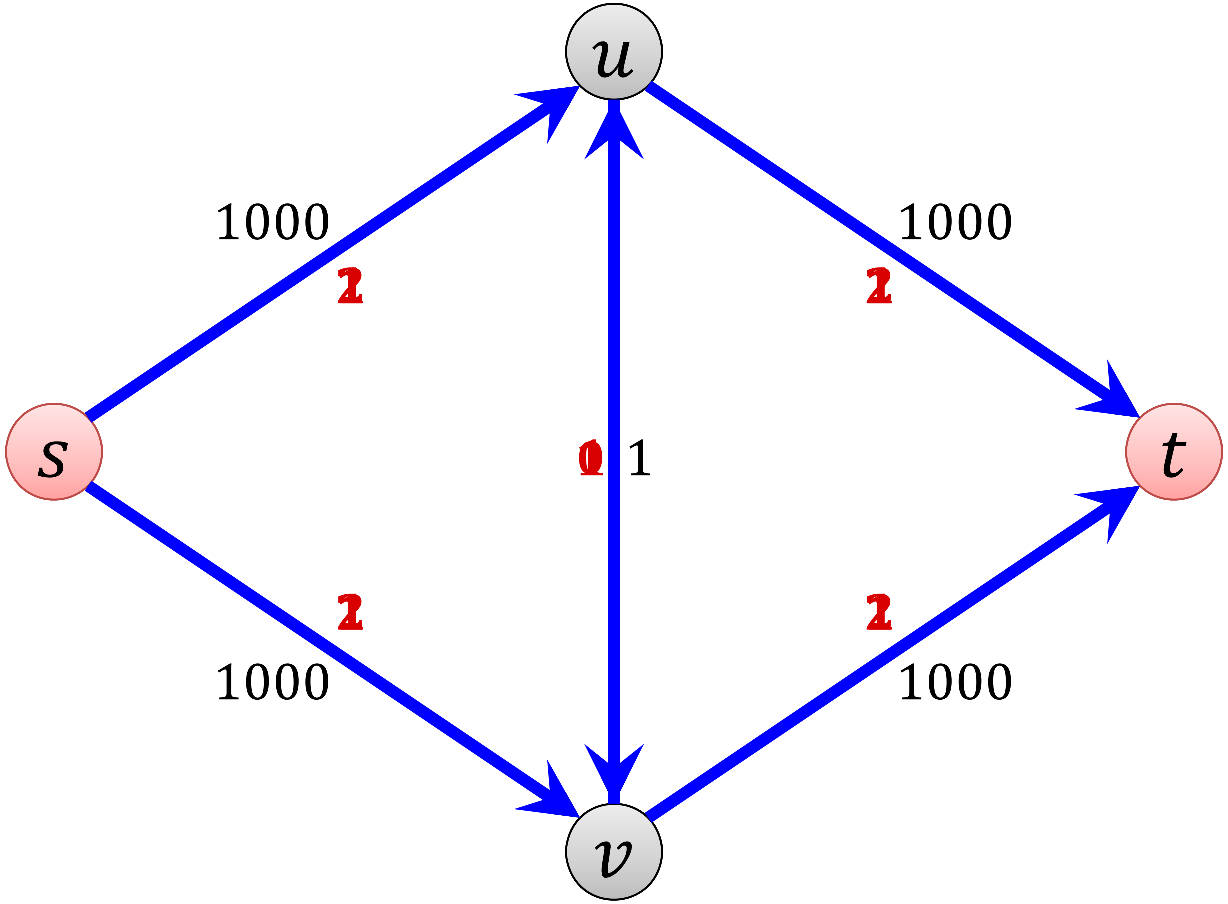
What if capacities are not integers?

- rational capacities:
  - can be turned into integers by multiplying them with large enough integer
  - algorithm still works correctly

- real (non-rational) capacities:
  - not clear whether the algorithm always terminates

- even for integer capacities, time can linearly depend on the value of the maximum flow

Slow Execution



- Number of iterations: 2000 (value of max. flow)

Improved Algorithm

Idea: Find the best augmenting path in each step

- best: path P with maximum $\mathrm{bottleneck}(P, f)$

- Best path might be rather expensive to find
  → find almost best path

- **Scaling parameter Δ:**
  (initially, $\Delta =$ "$\max c_e$ rounded down to next power of 2")

- As long as there is an augmenting path that improves the flow by at least Δ, augment using such a path

- If there is no such path: $\Delta \coloneqq \Delta/2$

Scaling Parameter Analysis

Lemma: If all capacities are integers, number of different scaling parameters used is $\leq 1 + \lfloor \log_2 C \rfloor$.

- **Δ-scaling phase:** Time during which scaling parameter is Δ

Length of a Scaling Phase

Lemma: If f is the flow at the end of the Δ-scaling phase, the maximum flow in the network has value at most $|f| + m\Delta$.

Length of a Scaling Phase

Lemma: The number of augmentation in each scaling phase is at most $2m$.

Running Time: Scaling Max Flow Alg.

Theorem: The number of augmentations of the algorithm with scaling parameter and integer capacities is at most $O(m \log C)$. The algorithm can be implemented in time $O(m^2 \log C)$.

Strongly Polynomial Algorithm

- Time of regular Ford-Fulkerson algorithm with integer capacities:

$$O(mC)$$

- Time of algorithm with scaling parameter:

$$O(m^2 \log C)$$

- $O(\log C)$ is polynomial in the size of the input, but not in n

- Can we get an algorithm that runs in time polynomial in n?

- Always picking a shortest augmenting path leads to running time

$$O(m^2 n)$$

Other Algorithms

- There are many other algorithms to solve the maximum flow problem, for example:

- **Preflow-push algorithm:**
  - Maintains a preflow (∀ nodes: inflow ≥ outflow)
  - Alg. guarantees: As soon as we have a flow, it is optimal
  - Detailed discussion in last year's lecture
  - Running time of basic algorithm: $O(m \cdot n^2)$
  - Doing steps in the "right" order: $O(n^3)$

- **Current best known complexity: $\boldsymbol{O(m \cdot n)}$**
  - For graphs with $m \geq n^{1+\epsilon}$ (for every constant $\epsilon > 0$) [King,Rao,Tarjan 1992/1994]
  - For sparse graphs with $m \leq n^{16/15-\delta}$ [Orlin, 2013]

Maximum Flow Applications

- Maximum flow has many applications

- Reducing a problem to a max flow problem can even be seen as an important algorithmic technique

- Examples:
  - related network flow problems
  - computation of small cuts
  - computation of matchings
  - computing disjoint paths
  - scheduling problems
  - assignment problems with some side constraints
  - …

Undirected Edges and Vertex Capacities

Undirected Edges:

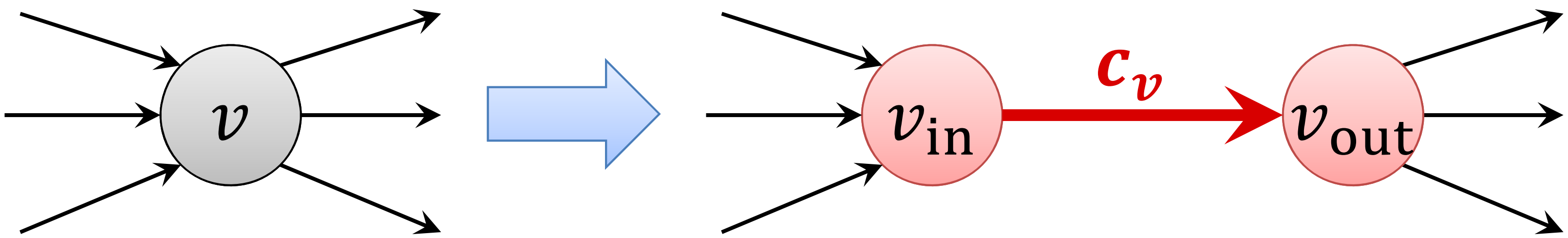
- Undirected edge $\{u, v\}$: add edges (u, v) and (v, u) to network

Vertex Capacities:

- Not only edge, but also (or only) nodes have capacities
- Capacity c_v of node $v \notin \{s, t\}$:

$$f^{\text{in}}(v) = f^{\text{out}}(v) \leq c_v$$

- Replace node v by edge $e_v = \{v_{\text{in}}, v_{\text{out}}\}$:



Minimum s-t Cut

Given: undirected graph $G = (V, E)$, nodes $s, t \in V$

s-t cut: Partition (A, B) of V such that $s \in A$, $t \in B$

Size of cut (A, B): number of edges crossing the cut

Objective: find s-t cut of minimum size

Edge Connectivity

Definition: A graph $G = (V, E)$ is k-edge connected for an integer $k \geq 1$ if the graph $G_X = (V, E \setminus X)$ is connected for every edge set

$$X \subseteq E, |X| \leq k - 1.$$

Goal: Compute edge connectivity $\lambda(G)$ of G
(and edge set X of size $\lambda(G)$ that divides G into ≥ 2 parts)

- minimum set X is a minimum s-t cut for some $s, t \in V$
  - Actually for all s, t in different components of $G_X = (V, E \setminus X)$
- Possible algorithm: fix s and find min s-t cut for all $t \neq s$

Minimum s-t Vertex-Cut

Given: undirected graph $G = (V, E)$, nodes $s, t \in V$

***s*-*t* vertex cut:** Set $X \subset V$ such that $s, t \notin X$ and s and t are in different components of the sub-graph $G[V \setminus X]$ induced by $V \setminus X$

Size of vertex cut: $|X|$

Objective: find s-t vertex-cut of minimum size

- Replace undirected edge $\{u, v\}$ by (u, v) and (v, u)
- Compute max s-t flow for edge capacities ∞ and node capacities

$$c_v = 1 \text{ for } v \neq s, t$$

- Replace each node v by v_{in} and v_{out}:
- Min edge cut corresponds to min vertex cut in G

Vertex Connectivity

Definition: A graph $G = (V, E)$ is k-vertex connected for an integer $k \geq 1$ if the sub-graph $G[V \setminus X]$ induced by $V \setminus X$ is connected for every edge set

$$X \subseteq V, |X| \leq k - 1.$$

Goal: Compute vertex connectivity $\kappa(G)$ of G

(and node set X of size $\kappa(G)$ that divides G into ≥ 2 parts)

- Compute minimum s-t vertex cut for fixed s and all $t \neq s$

Edge-Disjoint Paths

Given: Graph $G = (V, E)$ with nodes $s, t \in V$

Goal: Find as many edge-disjoint s-t paths as possible

Solution:

- Find max s-t flow in G with edge capacities $c_e = 1$ for all $e \in E$

Flow f induces $|f|$ edge-disjoint paths:

- Integral capacities → can compute integral max flow f
- Get $|f|$ edge-disjoint paths by greedily picking them
- Correctness follows from flow conservation $f^{\text{in}}(v) = f^{\text{out}}(v)$

Vertex-Disjoint Paths

Given: Graph $G = (V, E)$ with nodes $s, t \in V$

Goal: Find as many internally vertex-disjoint s-t paths as possible

Solution:

- Find max s-t flow in G with node capacities $c_v = 1$ for all $v \in V$

Flow f induces $|f|$ vertex-disjoint paths:

- Integral capacities → can compute integral max flow f
- Get $|f|$ vertex-disjoint paths by greedily picking them
- Correctness follows from flow conservation $f^{\text{in}}(v) = f^{\text{out}}(v)$

Menger's Theorem

Theorem: (edge version)
For every graph $G = (V, E)$ with nodes $s, t \in V$, the size of the minimum s-t (edge) cut equals the maximum number of pairwise edge-disjoint paths from s to t.

Theorem: (node version)
For every graph $G = (V, E)$ with nodes $s, t \in V$, the size of the minimum s-t vertex cut equals the maximum number of pairwise internally vertex-disjoint paths from s to t

- Both versions can be seen as a special case of the max flow min cut theorem

Baseball Elimination

Team i	Wins w_i	Losses ℓ_i	To Play r_i	Against = r_{ij}				
				NY	Balt.	T. Bay	Tor.	Bost.
New York	81	69	12	-	2	5	2	3
Baltimore	79	77	6	2	-	2	1	1
Tampa Bay	79	74	9	5	2	-	1	1
Toronto	76	80	6	2	1	1	-	2
Boston	70	85	7	3	1	1	2	-

- Only wins/losses possible (no ties), winner: team with most wins
- Which teams can still win (as least as many wins as top team)?
- Boston is eliminated (cannot win):
  - Boston can get at most 77 wins, New York already has 81 wins
- If for some $i, j: w_i + r_i < w_j$ → team i is eliminated
- Sufficient condition, but not a necessary one!

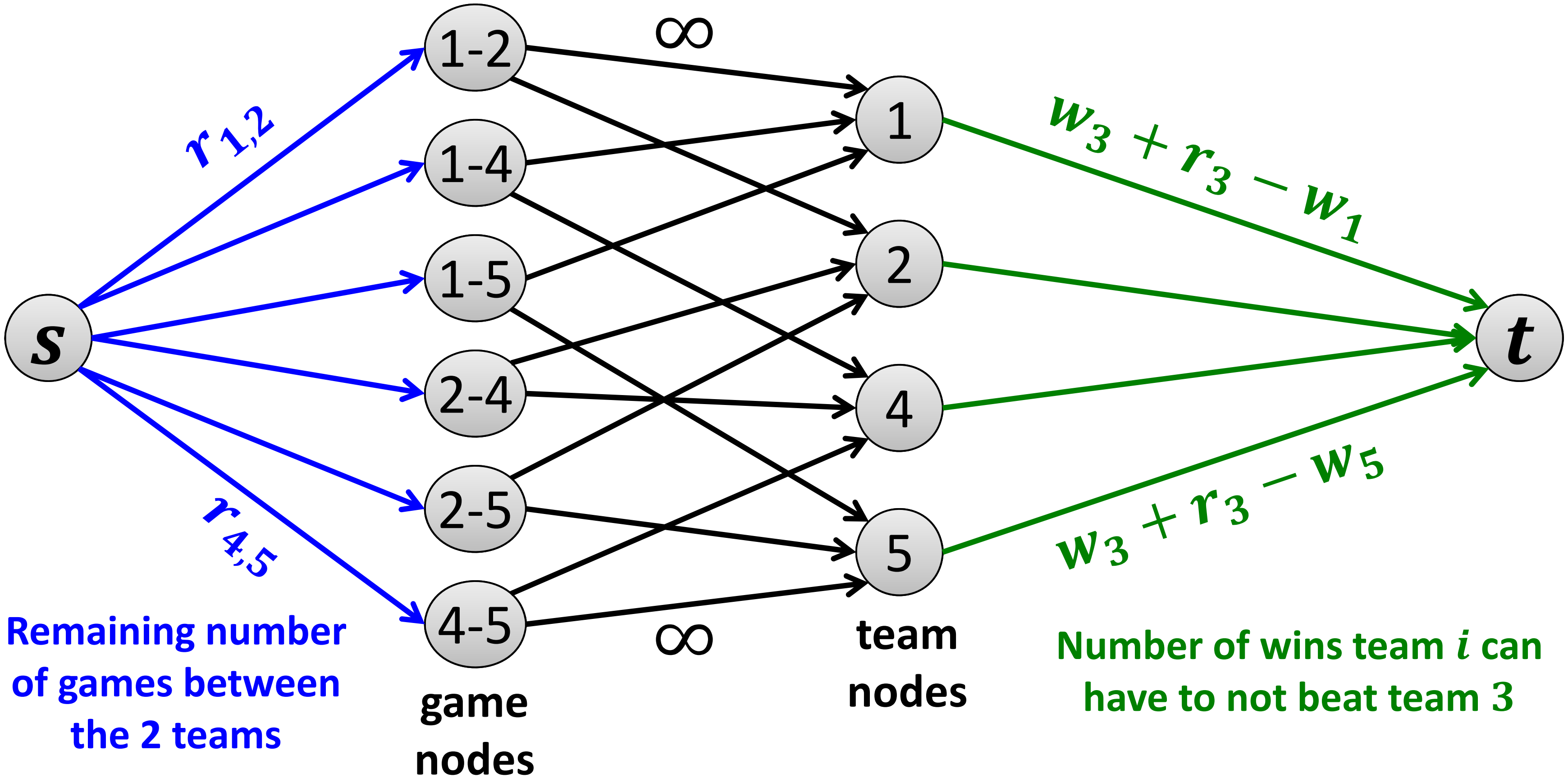
Baseball Elimination

Team i	Wins w_i	Losses ℓ_i	To Play r_i	Against = r_{ij}				
				NY	Balt.	T. Bay	Tor.	Bost.
New York	81	69	12	-	2	5	2	3
Baltimore	79	77	6	2	-	2	1	1
Tampa Bay	79	74	9	5	2	-	1	1
Toronto	76	80	6	2	1	1	-	2
Boston	70	85	7	3	1	1	2	-

- Can Toronto still finish first?
- Toronto can get $82 > 81$ wins, but:
  NY and Tampa have to play 5 more times against each other
  → if NY wins two, it gets 83 wins, otherwise, Tampa has 83 wins
- Hence: Toronto cannot finish first
- How about the others? How can we solve this in general?

Max Flow Formulation

- Can team 3 finish with most wins?



- Team 3 can finish first iff all source-game edges are saturated

Reason for Elimination

AL East: Aug 30, 1996

Team i	Wins w_i	Losses ℓ_i	To Play r_i	Against = r_{ij}				
				NY	Balt.	Bost.	Tor.	Detr.
New York	75	59	28	-	3	8	7	3
Baltimore	71	63	28	3	-	2	7	4
Boston	69	66	27	8	2	-	0	0
Toronto	63	72	27	7	7	0	-	0
Detroit	49	86	27	3	4	0	0	-

- Detroit could finish with $49 + 27 = 76$ wins
- Consider $R = \{\text{NY}, \text{Bal}, \text{Bos}, \text{Tor}\}$
  - Have together already won $w(R) = 278$ games
  - Must together win at least $r(R) = 27$ more games
- On average, teams in R win $\frac{278+27}{4} = 76.25$ games

Reason for Elimination

Certificate of elimination:

$$R \subseteq X, \qquad w(R) := \underbrace{\sum_{i \in R} w_i}_{\text{\#wins of nodes in } R}, \qquad r(R) := \underbrace{\sum_{i,j \in R} r_{i,j}}_{\text{\#remaining games among nodes in } R}$$

Team $x \in X$ is eliminated by R if

$$\frac{w(R) + r(R)}{|R|} > w_x + r_x.$$

Reason for Elimination

Theorem: Team x is eliminated if and only if there exists a subset $R \subseteq X$ of the teams X such that x is eliminated by R.

Proof Idea:

- Minimum cut gives a certificate…
- If x is eliminated, max flow solution does not saturate all outgoing edges of the source.
- Team nodes of unsaturated source-game edges are saturated
- Source side of min cut contains all teams of saturated team-dest. edges of unsaturated source-game edges
- Set of team nodes in source-side of min cut give a certificate R

Circulations with Demands

Given: Directed network with positive edge capacities

Sources & Sinks: Instead of one source and one destination, several sources that generate flow and several sinks that absorb flow.

Supply & Demand: sources have supply values, sinks demand values

Goal: Compute a flow such that source supplies and sink demands are exactly satisfied

- The circulation problem is a feasibility rather than a maximization problem

Circulations with Demands: Formally

Given: Directed network $G = (V, E)$ with

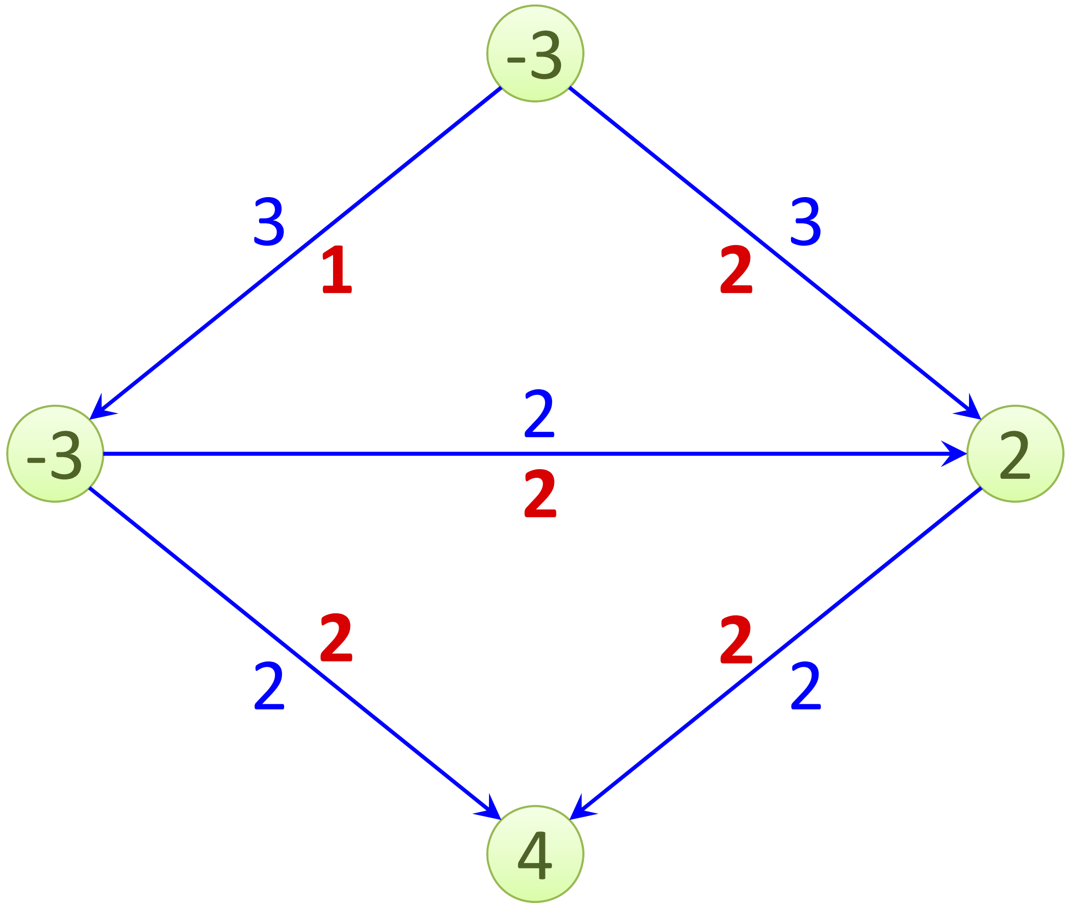
- Edge capacities $c_e > 0$ for all $e \in E$
- Node demands $d_v \in \mathbb{R}$ for all $v \in V$
  - $d_v > 0$: node needs flow and therefore is a sink
  - $d_v < 0$: node has a supply of $-d_v$ and is therefore a source
  - $d_v = 0$: node is neither a source nor a sink

Flow: Function $f: E \to \mathbb{R}_{\geq 0}$ satisfying

- *Capacity Conditions*: $\forall e \in E$: $\quad 0 \leq f(e) \leq c_e$
- *Demand Conditions*: $\forall v \in V$: $\quad f^{\text{in}}(v) - f^{\text{out}}(v) = d_v$

Objective: Does a flow f satisfying all conditions exist?
If yes, find such a flow f.

Example



Condition on Demands

Claim: If there exists a feasible circulation with demands d_v for $v \in V$, then

$$\sum_{v \in V} d_v = 0.$$

Proof:

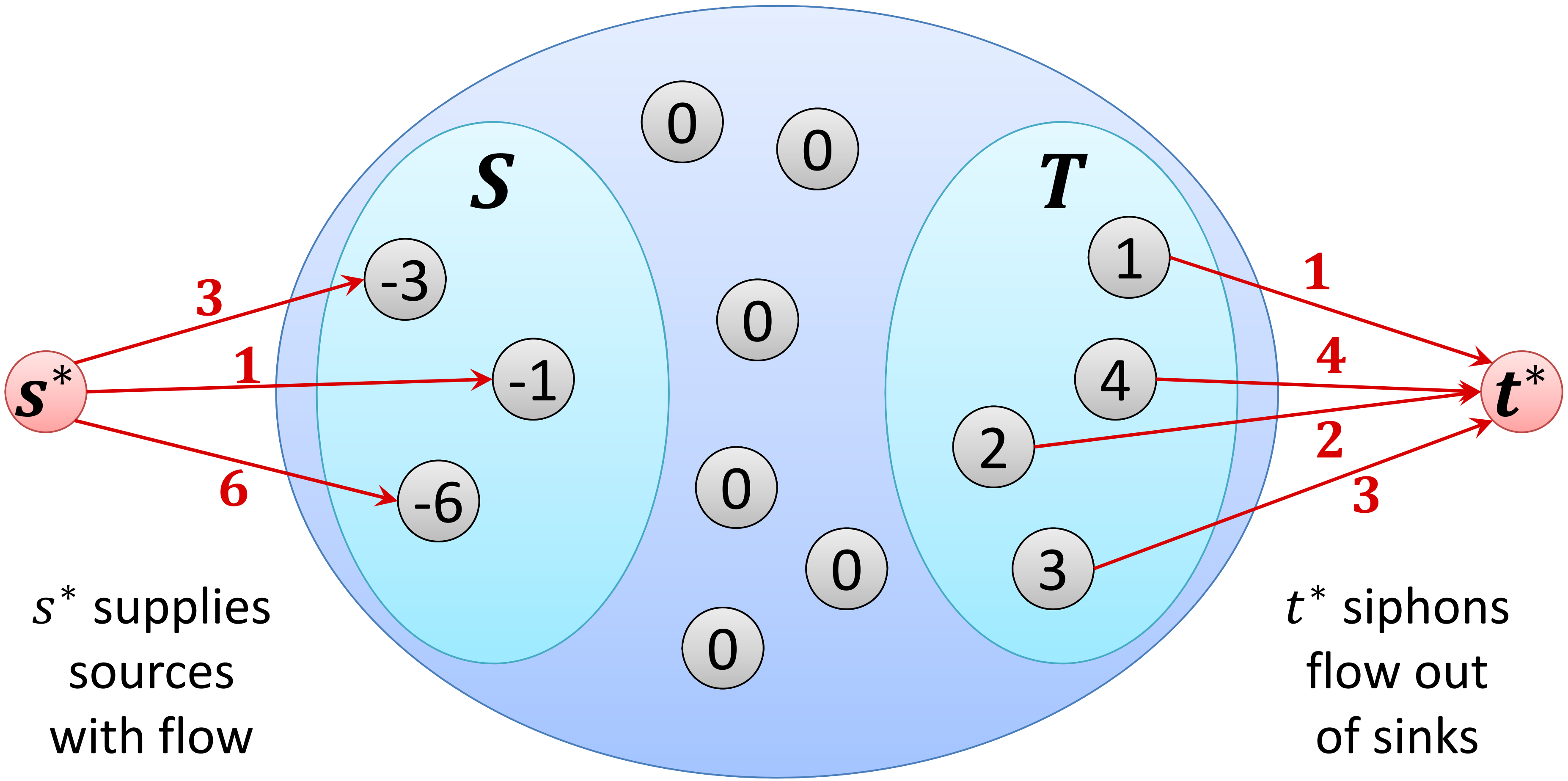
- $\sum_v d_v = \sum_v \left(f^{\text{in}}(v) - f^{\text{out}}(v)\right)$
- $f(e)$ of each edge e appears twice in the above sum with different signs → overall sum is 0

Total supply = total demand:

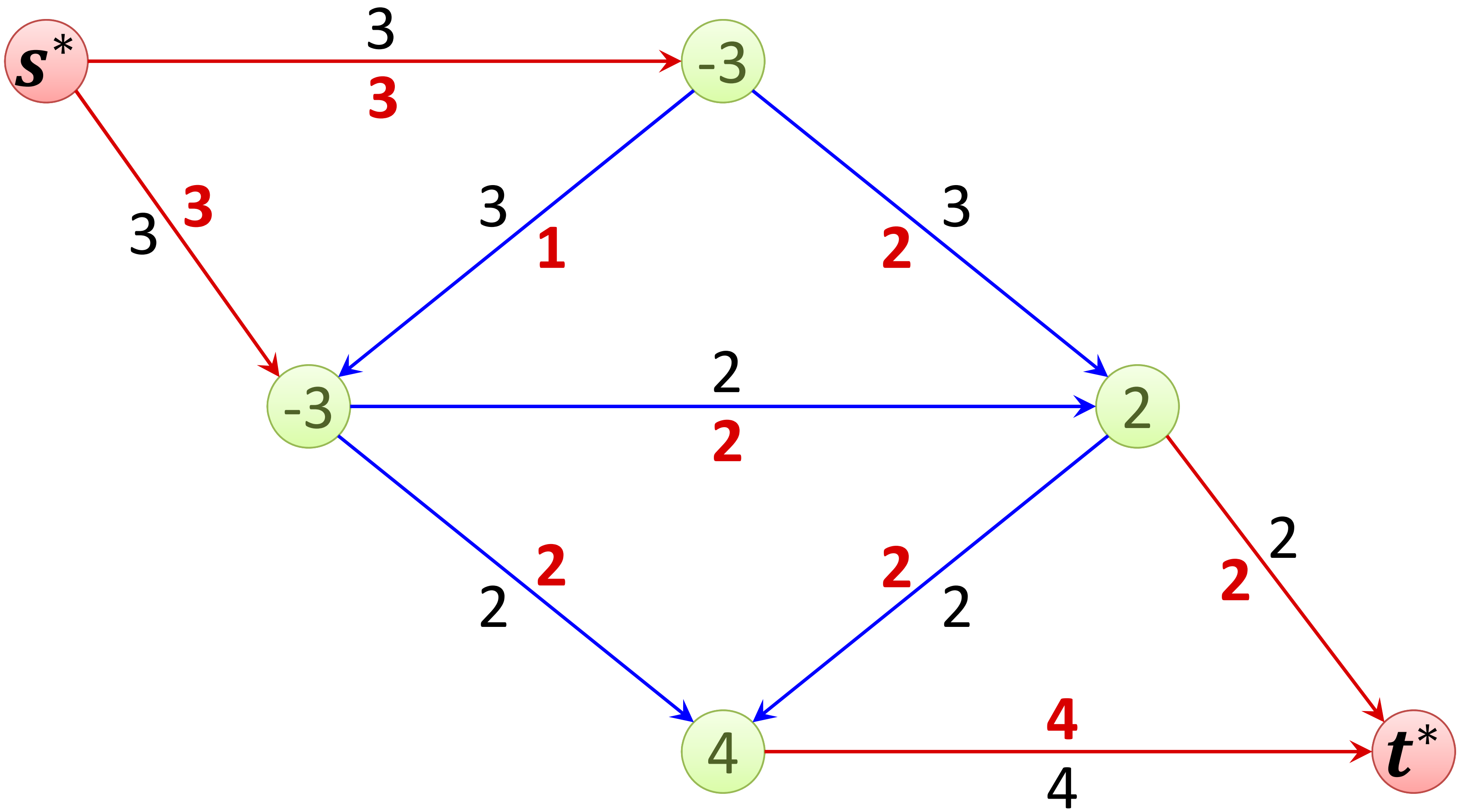
$$\text{Define } \boldsymbol{D} := \sum_{v : d_v > 0} \boldsymbol{d_v} = \sum_{v : d_v < 0} -\boldsymbol{d_v}$$

Reduction to Maximum Flow

- Add "super-source" s^* and "super-sink" t^* to network



Example



Formally…

Reduction: Get graph G' from graph as follows

- Node set of G' is $V \cup \{s^*, t^*\}$
- Edge set is E and edges
  - (s^*, v) for all v with $d_v < 0$, capacity of edge is d_v
  - (v, t^*) for all v with $d_v > 0$, capacity of edge is d_v

Observations:

- Capacity of min s^*-t^* cut is at most D (e.g., the cut $(s^*, V \cup \{t^*\})$)
- A feasible circulation on G can be turned into a feasible flow of value D of G' by saturating all (s^*, v) and (v, t^*) edges.
- Any flow of G' of value D induces a feasible circulation on G
  - (s^*, v) and (v, t^*) edges are saturated
  - By removing these edges, we get exactly the demand constraints

Circulation with Demands

Theorem: There is a feasible circulation with demands $d_v, v \in V$ on graph G if and only if there is a flow of value D on G'.

- If all capacities and demands are integers, there is an integer circulation

The max flow min cut theorem also implies the following:

Theorem: The graph G has a feasible circulation with demands $d_v, v \in V$ if and only if for all cuts (A, B),

$$\sum_{v \in B} d_v \leq c(A, B).$$

Circulation: Demands and Lower Bounds

Given: Directed network $G = (V, E)$ with

- Edge capacities $c_e > 0$ and **lower bounds $\mathbf{0 \leq \ell_e \leq c_e}$ for $\boldsymbol{e \in E}$**
- Node demands $d_v \in \mathbb{R}$ for all $v \in V$
  - $d_v > 0$: node needs flow and therefore is a sink
  - $d_v < 0$: node has a supply of $-d_v$ and is therefore a source
  - $d_v = 0$: node is neither a source nor a sink

Flow: Function $f: E \to \mathbb{R}_{\geq 0}$ satisfying

- *Capacity Conditions*: $\forall e \in E$: $\boldsymbol{\ell_e \leq f(e) \leq c_e}$
- *Demand Conditions*: $\forall v \in V$: $f^{\text{in}}(v) - f^{\text{out}}(v) = d_v$

Objective: Does a flow f satisfying all conditions exist?
If yes, find such a flow f.

Solution Idea

- Define initial circulation $f_0(e) = \ell_e$
  Satisfies capacity constraints: $\forall e \in E: \ell_e \leq f_0(e) \leq c_e$

- Define

$$L_v := f_0^{\text{in}}(v) - f_0^{\text{out}}(v) = \sum_{e \text{ into } v} \ell_e - \sum_{e \text{ out of } v} \ell_e$$

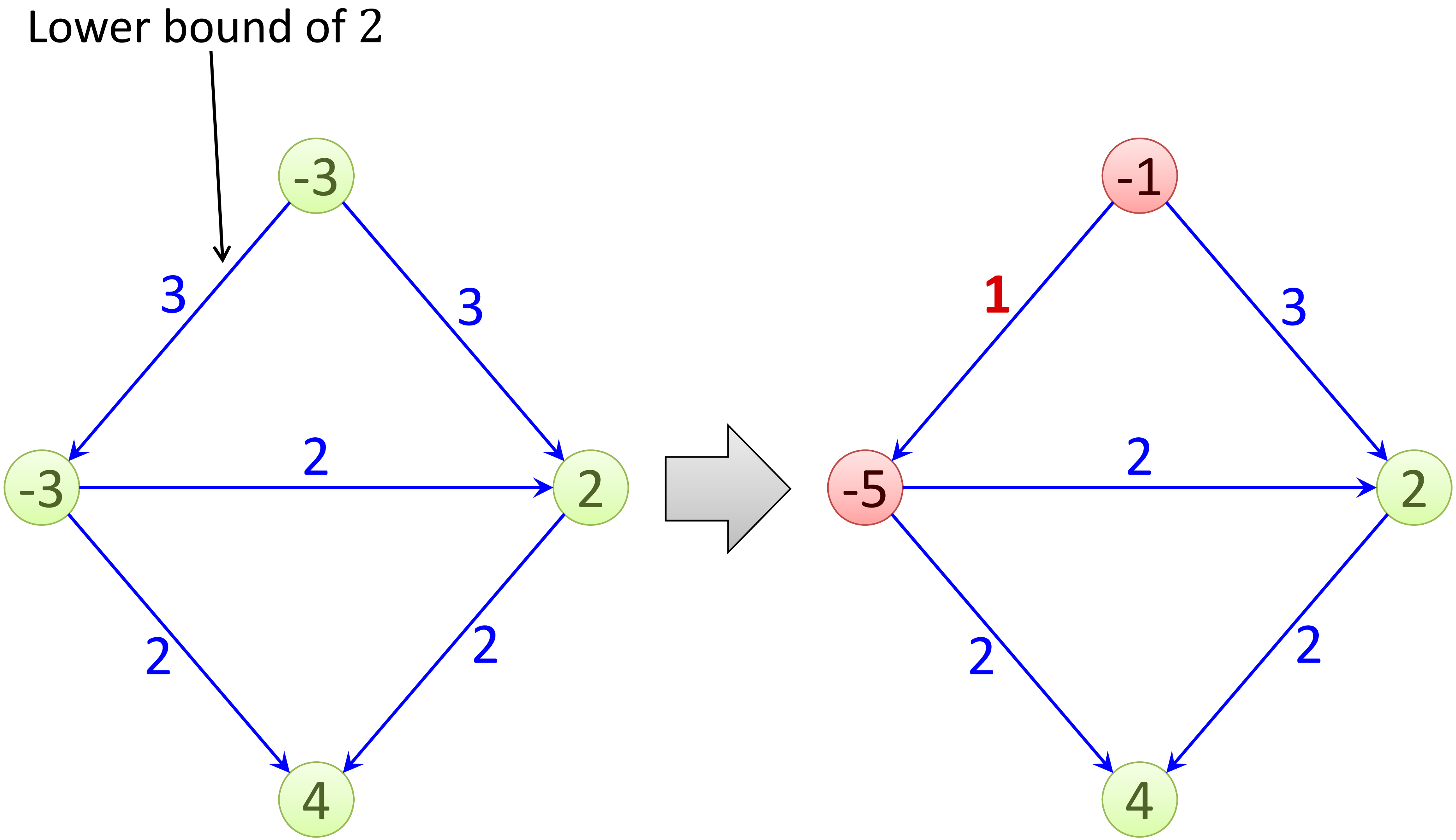
- If $L_v = d_v$, demand condition is satisfied at v by f_0, otherwise, we need to superimpose another circulation f_1 such that

$$d'_v := f_1^{\text{in}}(v) - f_1^{\text{out}}(v) = d_v - L_v$$

- Remaining capacity of edge e: $c'_e := c_e - \ell_e$

- We get a circulation problem with new demands d'_v, new capacities c'_e, and no lower bounds

Eliminating a Lower Bound: Example

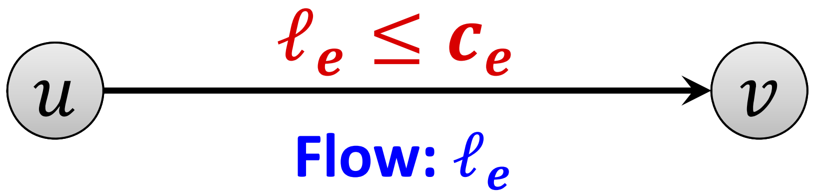


Reduce to Problem Without Lower Bounds

Graph $G = (V, E)$:

- Capacity: For each edge $e \in E$: $\ell_e \leq f(e) \leq c_e$
- Demand: For each node $v \in V$: $f^{\text{in}}(v) - f^{\text{out}}(v) = d_v$

Model lower bounds with supplies & demands:



Create Network G' (without lower bounds):

- For each edge $e \in E$: $c'_e = c_e - \ell_e$
- For each node $v \in V$: $d'_v = d_v - L_v$

Circulation: Demands and Lower Bounds

Theorem: There is a feasible circulation in G (with lower bounds) if and only if there is feasible circulation in G' (without lower bounds).

- Given circulation f' in G', $f(e) = f'(e) + \ell_e$ is circulation in G
  - The capacity constraints are satisfied because $f'(e) \leq c_e - \ell_e$
  - Demand conditions:

$$f^{\text{in}}(v) - f^{\text{out}}(v) = \sum_{e \text{ into } v} (\ell_e + f'(e)) - \sum_{e \text{ out of } v} (\ell_e + f'(e))$$
$$= L_v + (d_v - L_v) = d_v$$

- Given circulation f in G, $f'(e) = f(e) - \ell_e$ is circulation in G'
  - The capacity constraints are satisfied because $\ell_e \leq f(e) \leq c_e$
  - Demand conditions:

$$f'^{\text{in}}(v) - f'^{\text{out}}(v) = \sum_{e \text{ into } v} (f(e) - \ell_e) - \sum_{e \text{ out of } v} (f(e) - \ell_e)$$
$$= d_v - L_v$$

Integrality

Theorem: Consider a circulation problem with integral capacities, flow lower bounds, and node demands. If the problem is feasible, then it also has an integral solution.

Proof:

- Graph G' has only integral capacities and demands
- Thus, the flow network used in the reduction to solve circulation with demands and no lower bounds has only integral capacities
- The theorem now follows because a max flow problem with integral capacities also has an optimal integral solution

- It also follows that with the max flow algorithms we studied, we get an integral feasible circulation solution.

Matrix Rounding

- **Given:** $p \times q$ matrix $D = \{d_{i,j}\}$ of real numbers
- **row i sum:** $a_i = \sum_j d_{i,j}$,   **column j sum:** $b_j = \sum_i d_{i,j}$
- **Goal:** Round each $d_{i,j}$, as well as a_i and b_j up or down to the next integer so that the sum of rounded elements in each row (column) equals the rounded row (column) sum
- **Original application:** publishing census data

Example:

3.14	6.80	7.30	17.24
9.60	2.40	0.70	12.70
3.60	1.20	6.50	11.30
16.34	10.40	14.50	

original data

→

3	7	7	17
10	2	1	13
3	1	7	11
16	10	15	

possible rounding

Matrix Rounding

Theorem: For any matrix, there exists a feasible rounding.

Remark: Just rounding to the nearest integer doesn't work

0.35	0.35	0.35	1.05
0.55	0.55	0.55	1.65
0.90	0.90	0.90	

original data

0	0	0	0
1	1	1	3
1	1	1	

rounding to nearest integer

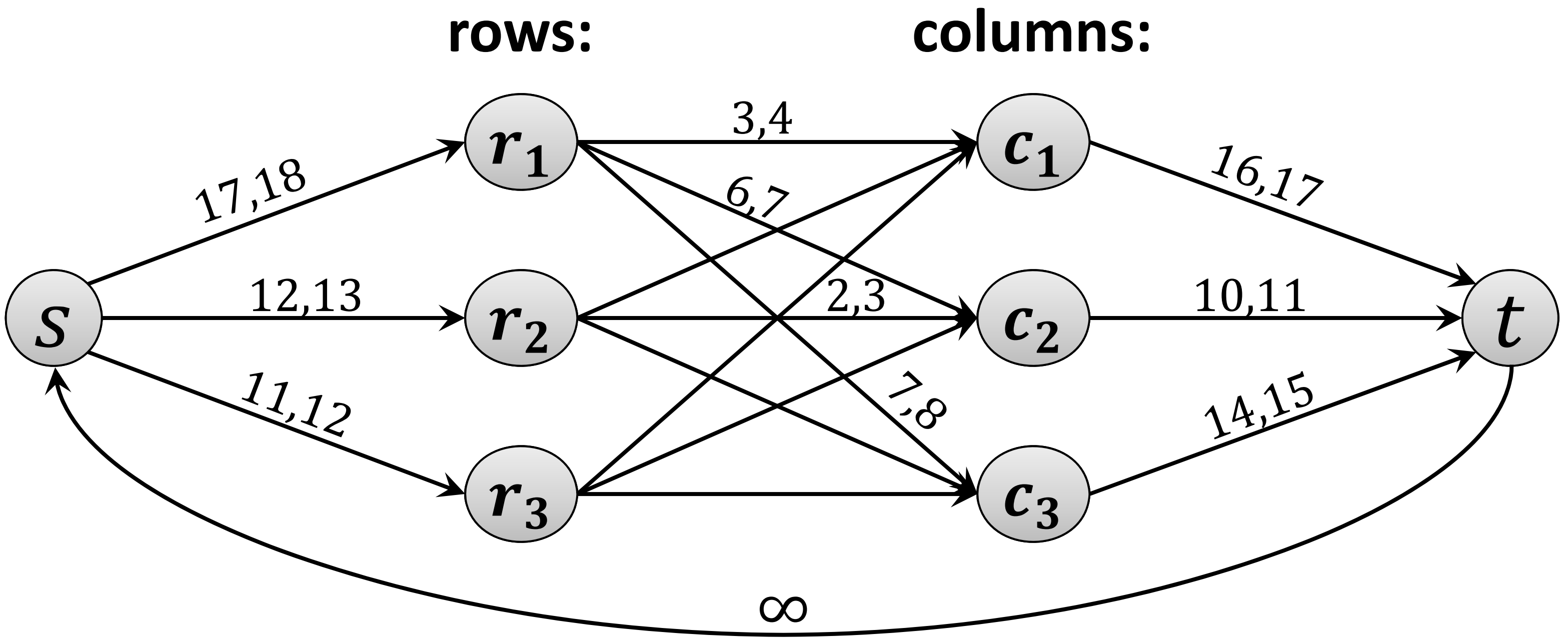
0	0	1	1
1	1	0	2
1	1	1	

feasible rounding

Reduction to Circulation

3.14	6.80	7.30	17.24
9.60	2.40	0.70	12.70
3.60	1.20	6.50	11.30
16.34	10.40	14.50	

Matrix elements and row/column sums give a feasible circulation that satisfies all lower bound, capacity, and demand constraints



all demands $d_v = 0$

Matrix Rounding

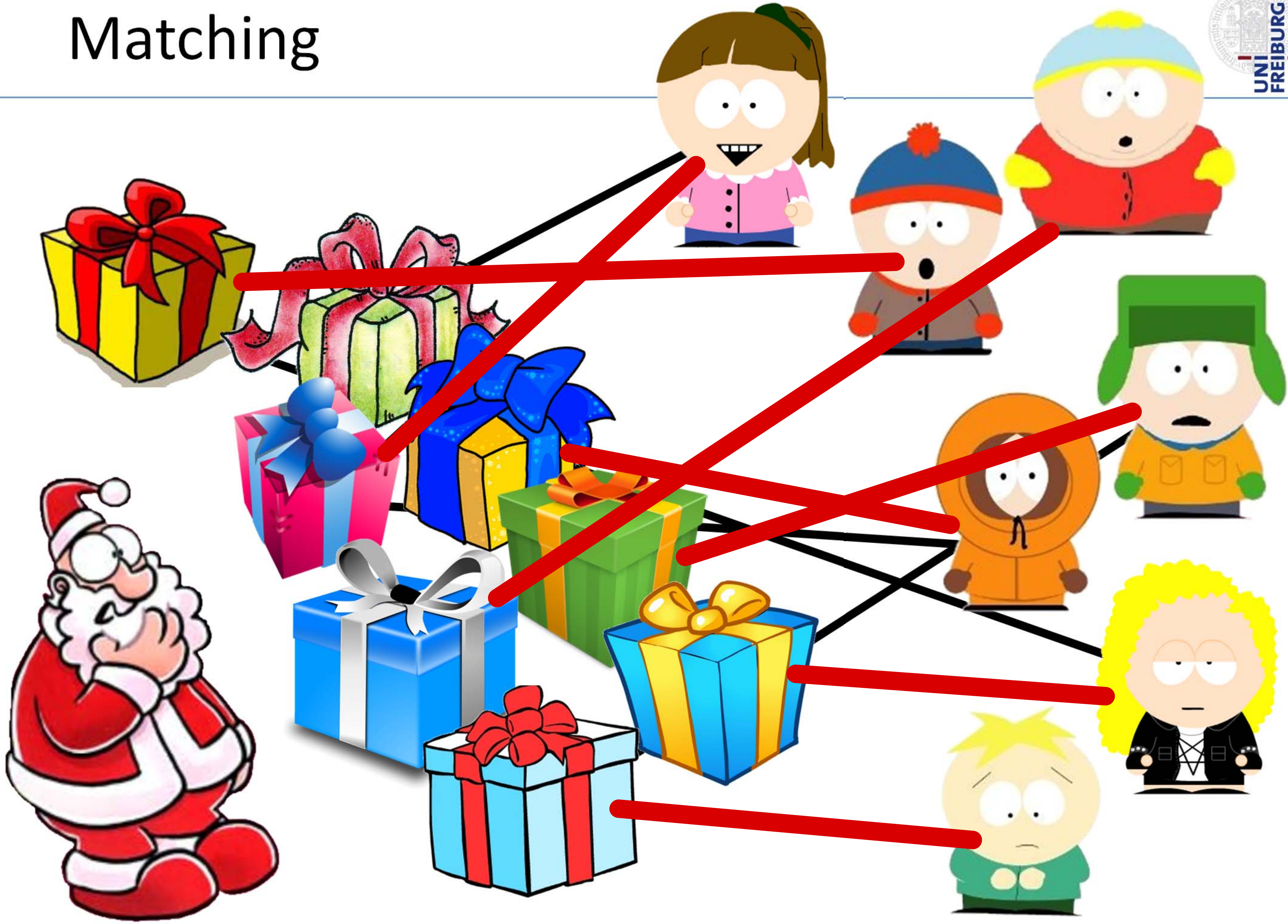
Theorem: For any matrix, there exists a feasible rounding.

Proof:

- The matrix entries $d_{i,j}$ and the row and column sums a_i and b_j give a feasible circulation for the constructed network
- Every feasible circulation gives matrix entries with corresponding row and column sums (follows from demand constraints)
- Because all demands, capacities, and flow lower bounds are integral, there is an integral solution to the circulation problem

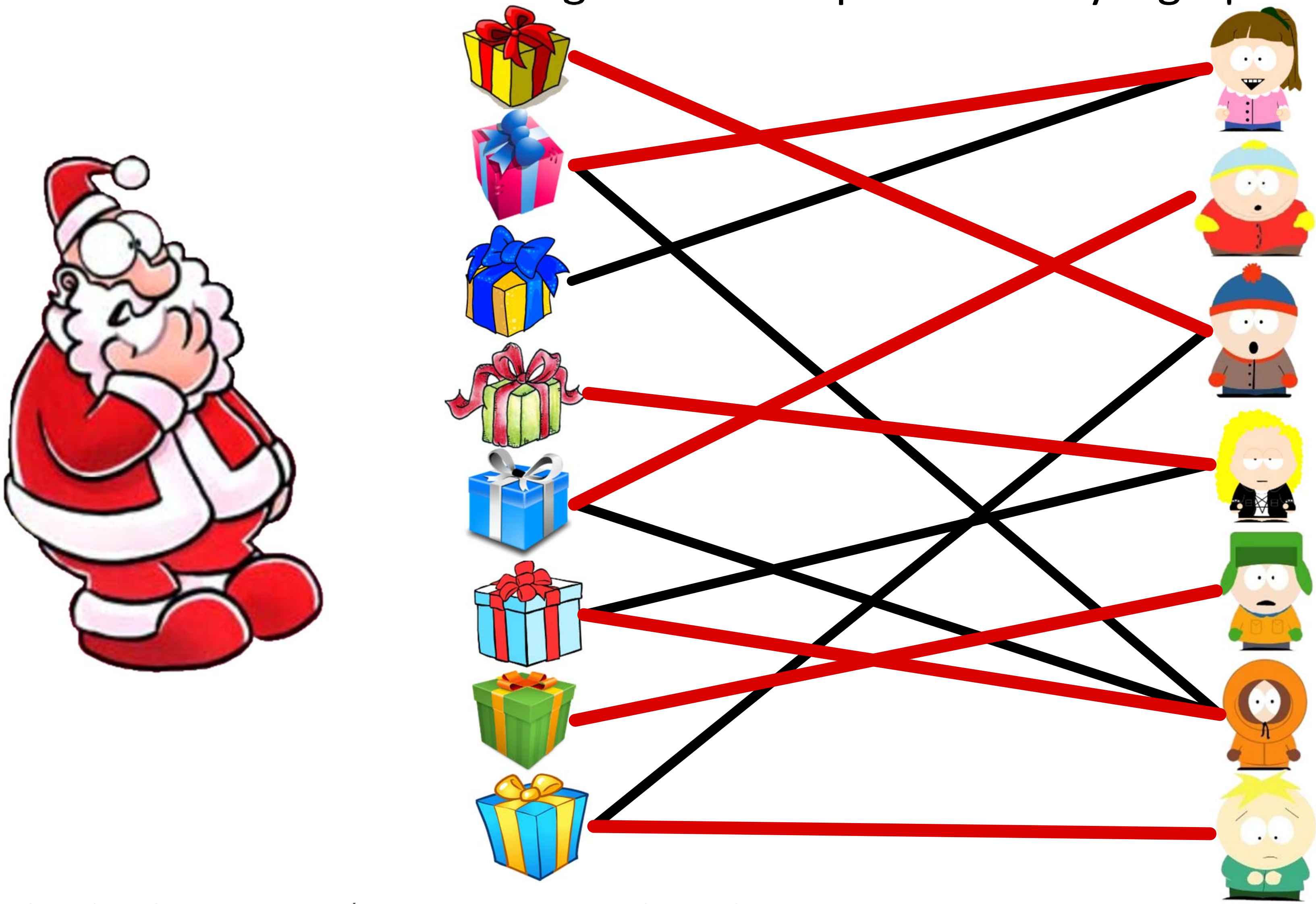
→ gives a feasible rounding!

Matching



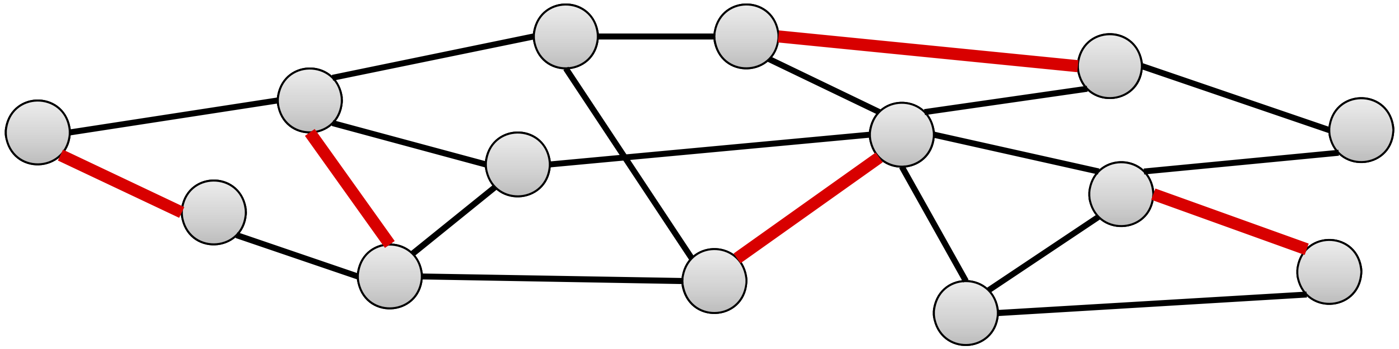
Gifts-Children Graph

- Which child likes which gift can be represented by a graph



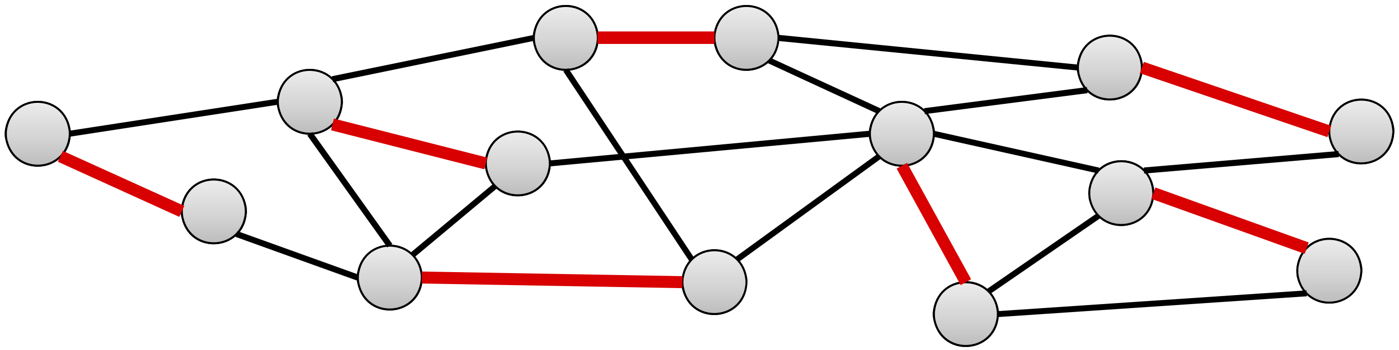
Matching

Matching: Set of pairwise non-incident edges



Maximal Matching: A matching s.t. no more edges can be added

Maximum Matching: A matching of maximum possible size



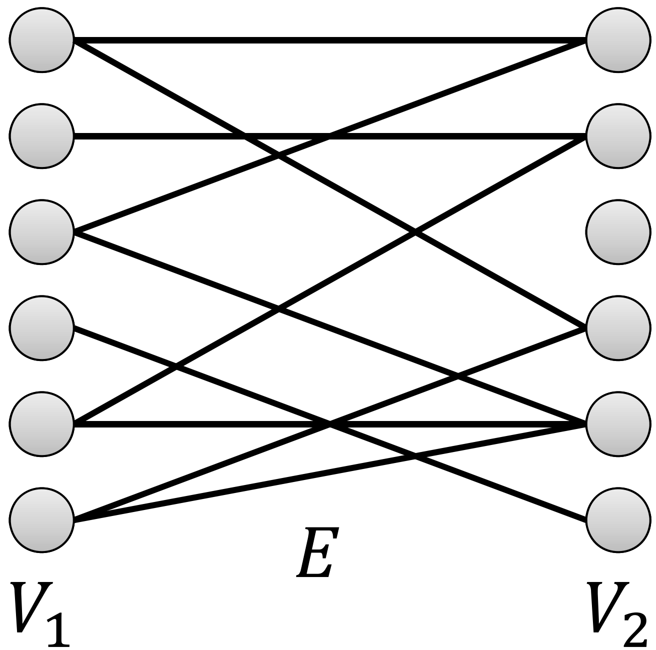
Perfect Matching: Matching of size $n/2$ (every node is matched)

Bipartite Graph

Definition: A graph $G = (V, E)$ is called bipartite iff its node set can be partitioned into two parts $V = V_1 \cupdot V_2$ such that for each edge $\{\mathrm{u}, \mathrm{v}\} \in E$,

$$|\{u, v\} \cap V_1| = 1.$$

- Thus, edges are only between the two parts



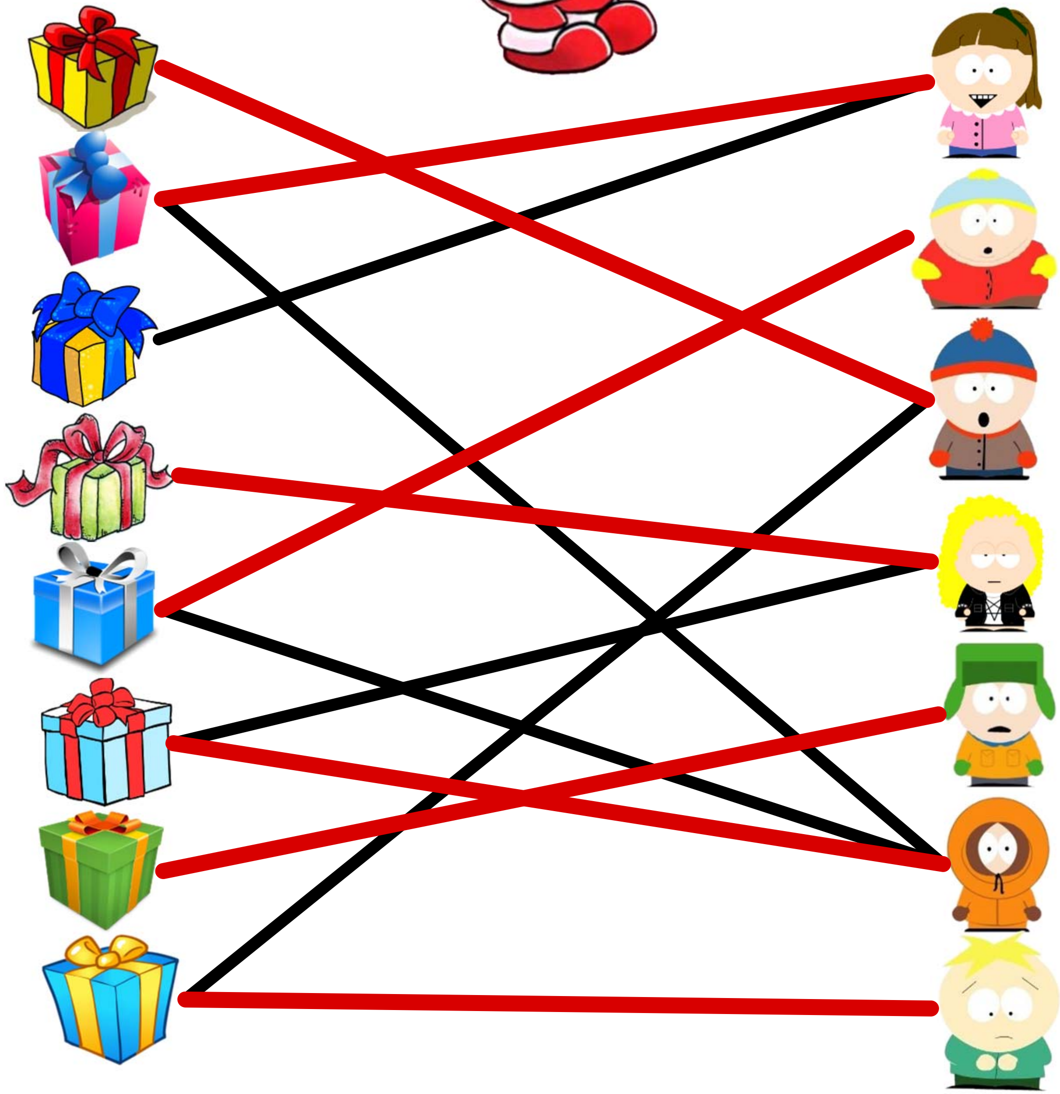
Santa's Problem

Maximum Matching in Bipartite Graphs:

Every child can get a gift iff there is a matching of size #children

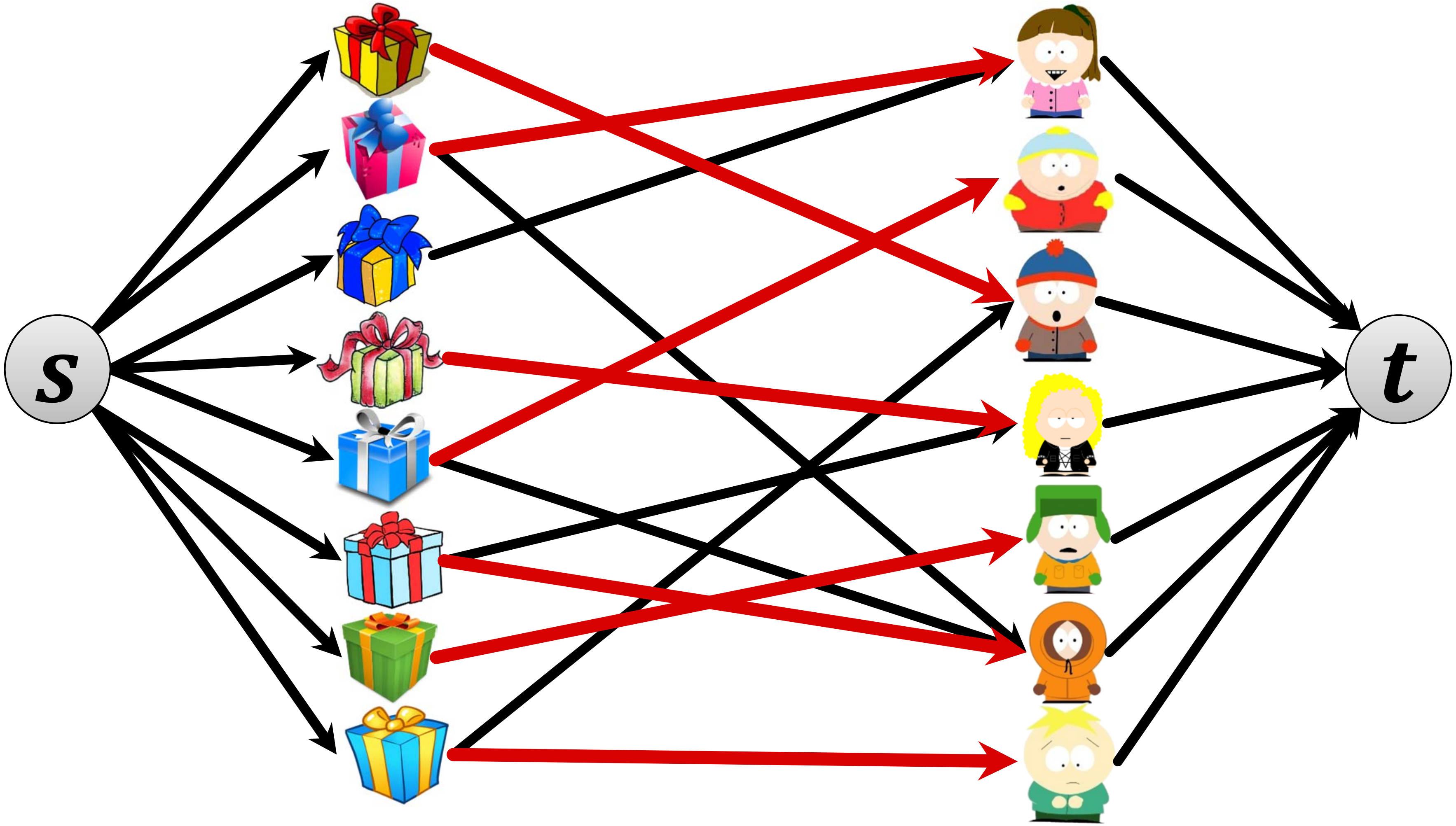
Clearly, every matching is at most as big

If #children = #gifts, there is a solution iff there is a perfect matching



Reducing to Maximum Flow

- Like edge-disjoint paths…



all capacities are 1

Reducing to Maximum Flow

Theorem: Every integer solution to the max flow problem on the constructed graph induces a maximum bipartite matching of G.

Proof:

1. An integer flow f of value $|f|$ induces a matching of size $|f|$
   - Left nodes (gifts) have incoming capacity 1
   - Right nodes (children) have outgoing capacity 1
   - Left and right nodes are incident to ≤ 1 edge e of G with $f(e) = 1$

2. A matching of size k implies a flow f of value $|f| = k$
   - For each edge $\{u, v\}$ of the matching:

   $$f\big((s,u)\big) = f\big((u,v)\big) = f\big((v,t)\big) = 1$$

   - All other flow values are 0

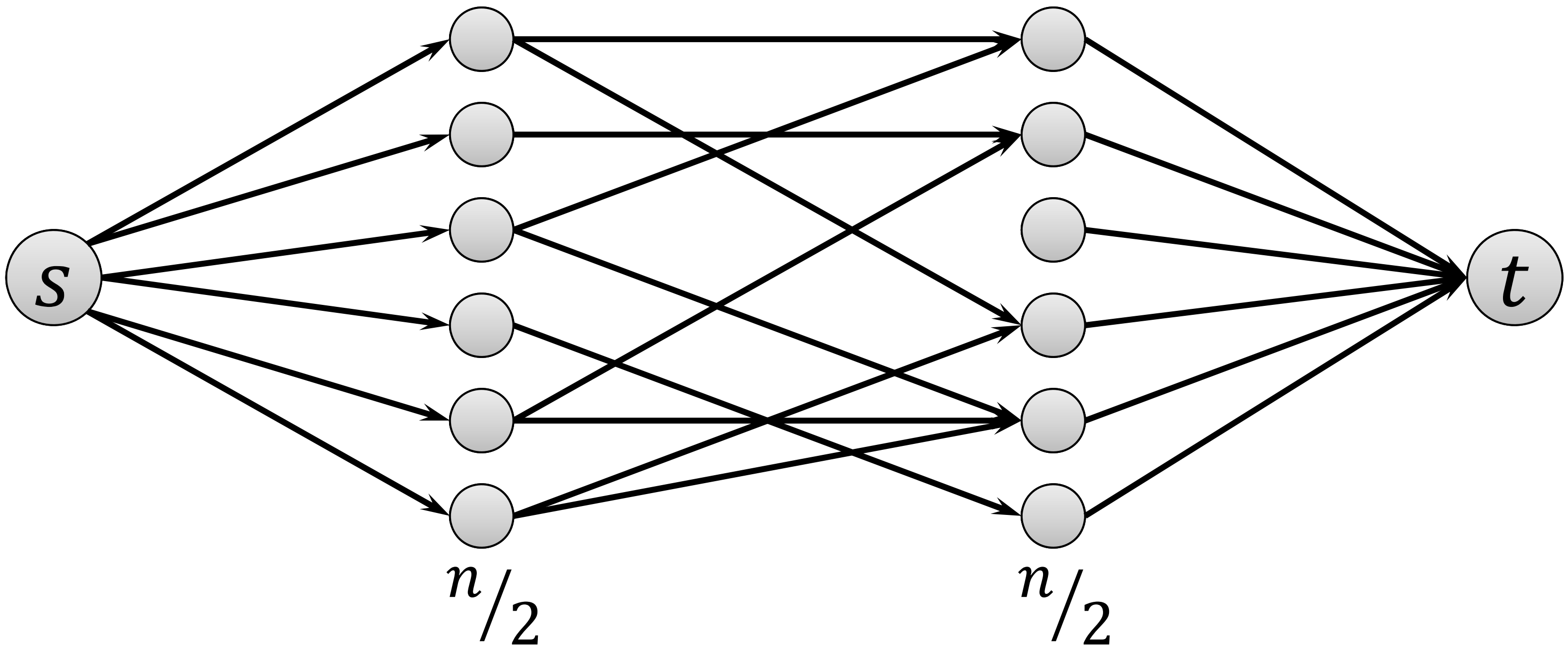
Running Time of Max. Bipartite Matching

Theorem: A maximum matching of a bipartite graph can be computed in time $O(m \cdot n)$.

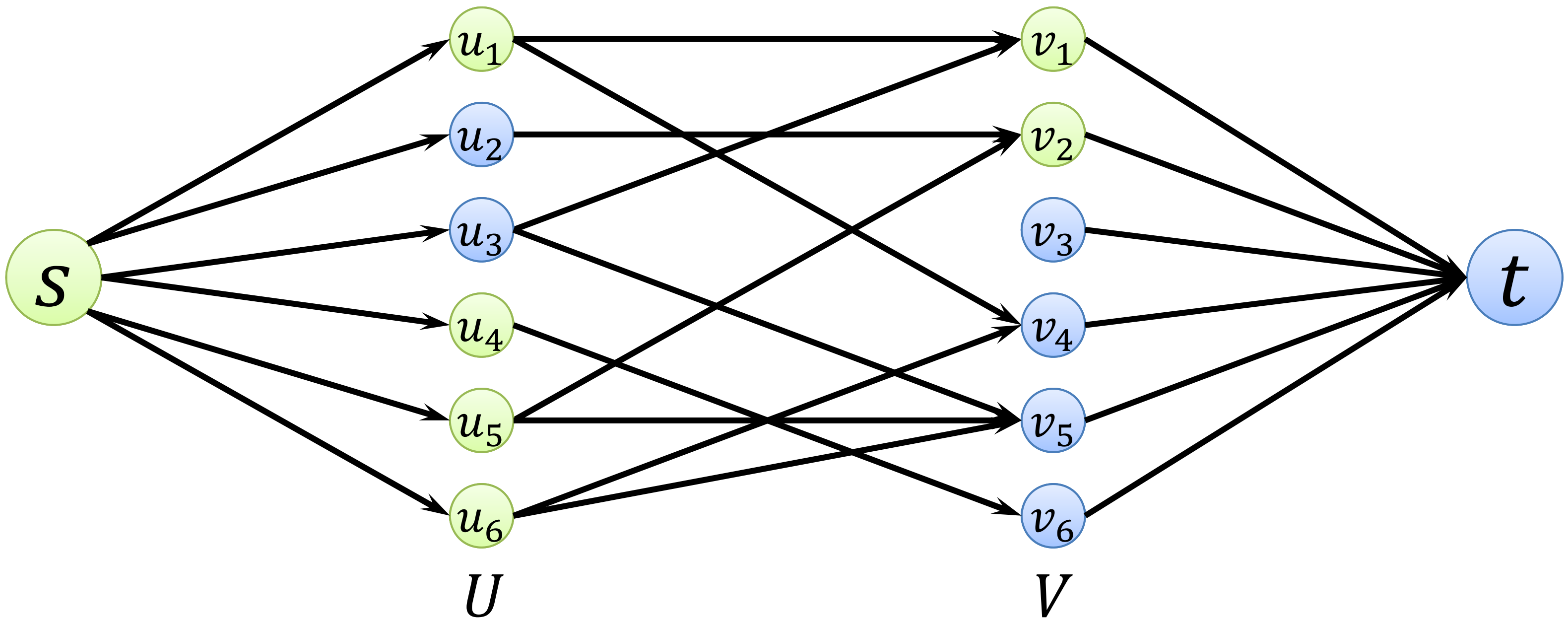
Perfect Matching?

- There can only be a perfect matching if both sides of the partition have size $n/_2$.

- There is no perfect matching, iff there is an s-t cut of size $< n/_2$ in the flow network.



s-t Cuts



Partition (A, B) of node set such that $s \in A$ and $t \in B$

- If $v_i \in A$: edge (v_i, t) is in cut (A, B)
- If $u_i \in B$: edge (s, u_i) is in cut (A, B)
- Otherwise (if $u_i \in A$, $v_i \in B$), all edges from u_i to some $v_j \in B$ are in cut (A, B)

Hall's Marriage Theorem

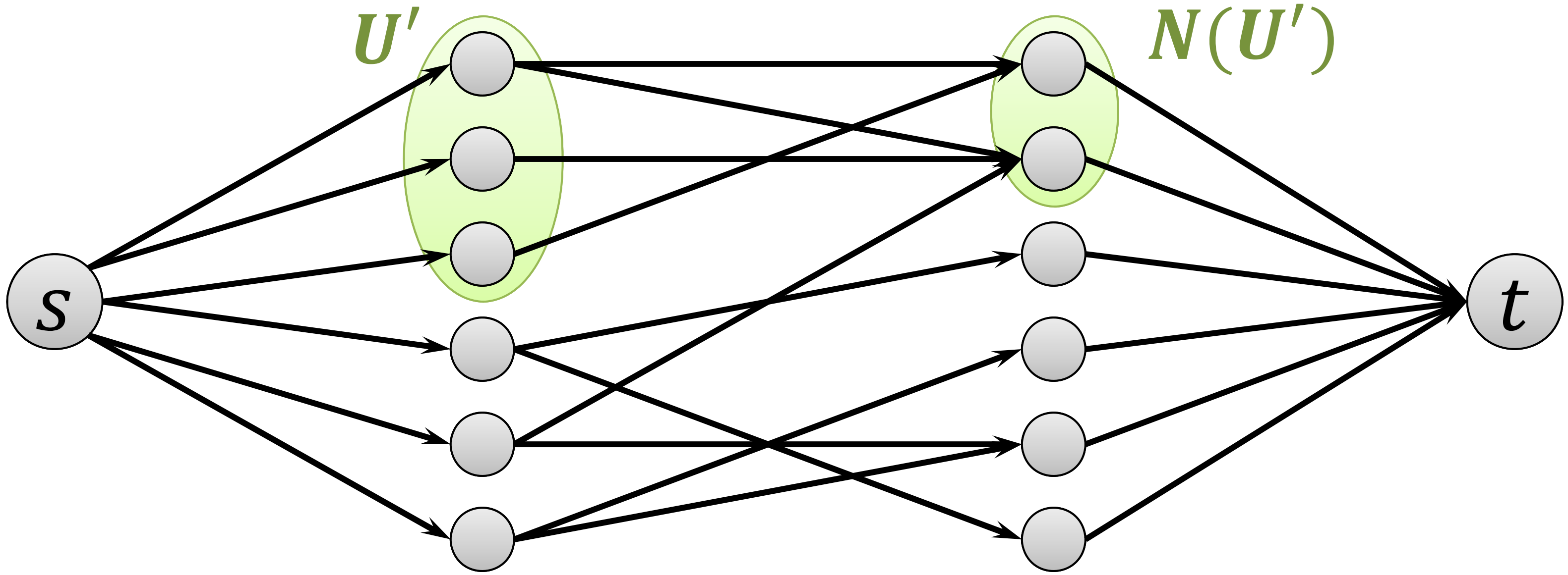
Theorem: A bipartite graph $G = (U \cup V, E)$ for which $|U| = |V|$ has a perfect matching if and only if

$$\forall \boldsymbol{U'} \subseteq \boldsymbol{U} : |\boldsymbol{N}(\boldsymbol{U'})| \geq |\boldsymbol{U'}|,$$

where $N(U') \subseteq V$ is the set of neighbors of nodes in U'.

Proof: No perfect matching $\Leftrightarrow$ some s-t cut has capacity $< n/2$

1. Assume there is U' for which $|N(U')| < |\mathrm{U}'|$:



Hall's Marriage Theorem

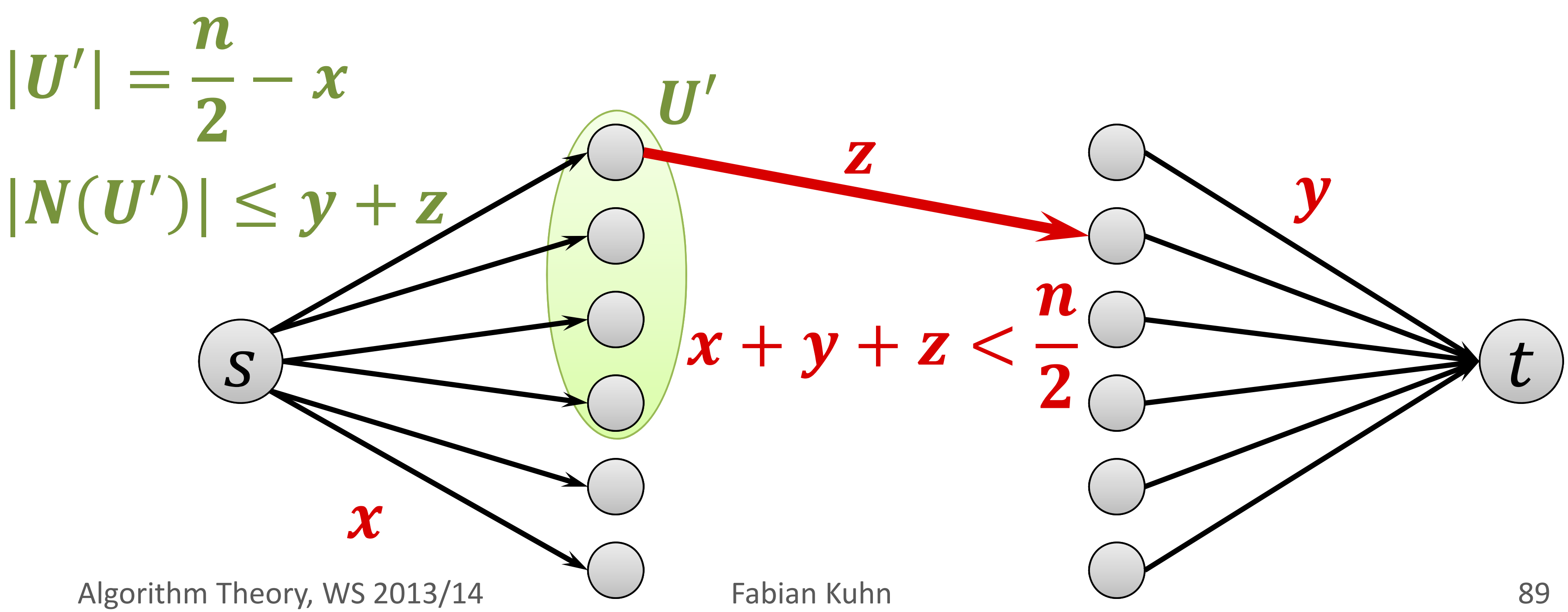
Theorem: A bipartite graph $G = (U \cup V, E)$ for which $|U| = |V|$ has a perfect matching if and only if

$$\forall U' \subseteq U: |N(U')| \geq |U'|,$$

where $N(U') \subseteq V$ is the set of neighbors of nodes in U'.

Proof: No perfect matching $\Leftrightarrow$ some s-t cut has capacity $< n/2$

2. Assume that there is a cut (A, B) of capacity $< n/2$



Hall's Marriage Theorem

Theorem: A bipartite graph $G = (U \cup V, E)$ for which $|U| = |V|$ has a perfect matching if and only if

$$\forall \boldsymbol{U'} \subseteq \boldsymbol{U}: |\boldsymbol{N}(\boldsymbol{U'})| \geq |\boldsymbol{U'}|,$$

where $N(U') \subseteq V$ is the set of neighbors of nodes in U'.

Proof: No perfect matching $\Leftrightarrow$ some s-t cut has capacity $< n$

2. Assume that there is a cut (A, B) of capacity $< n$

$$|\boldsymbol{U'}| = \frac{\boldsymbol{n}}{\boldsymbol{2}} - \boldsymbol{x}$$

$$|\boldsymbol{N}(\boldsymbol{U'})| \leq \boldsymbol{y} + \boldsymbol{z}$$

$$\boldsymbol{x} + \boldsymbol{y} + \boldsymbol{z} < \frac{\boldsymbol{n}}{\boldsymbol{2}}$$

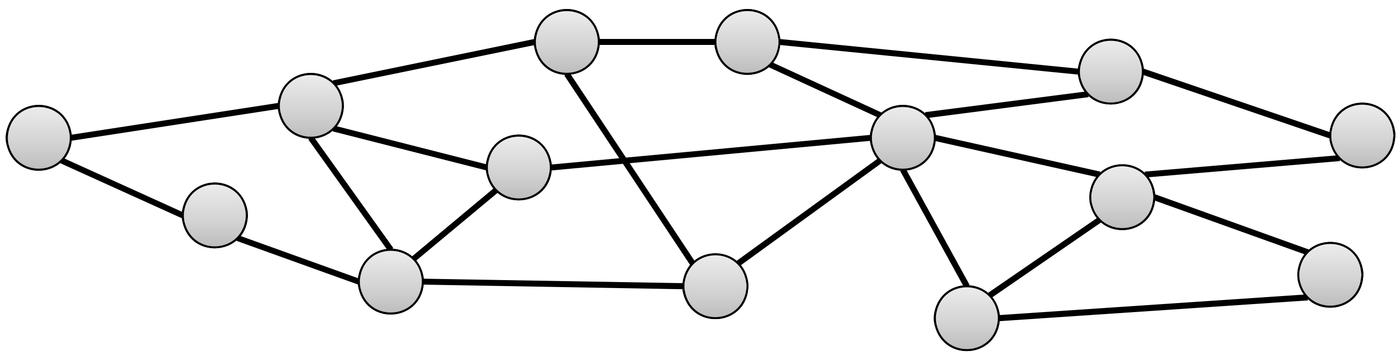
What About General Graphs

- Can we efficiently compute a maximum matching if G is not bipartite?

- How good is a maximal matching?
  - A matching that cannot be extended…

- **Vertex Cover:** set $S \subseteq V$ of nodes such that

$$\forall \{u, v\} \in E, \qquad \{u, v\} \cap S \neq \emptyset.$$



- A vertex cover covers all edges by incident nodes

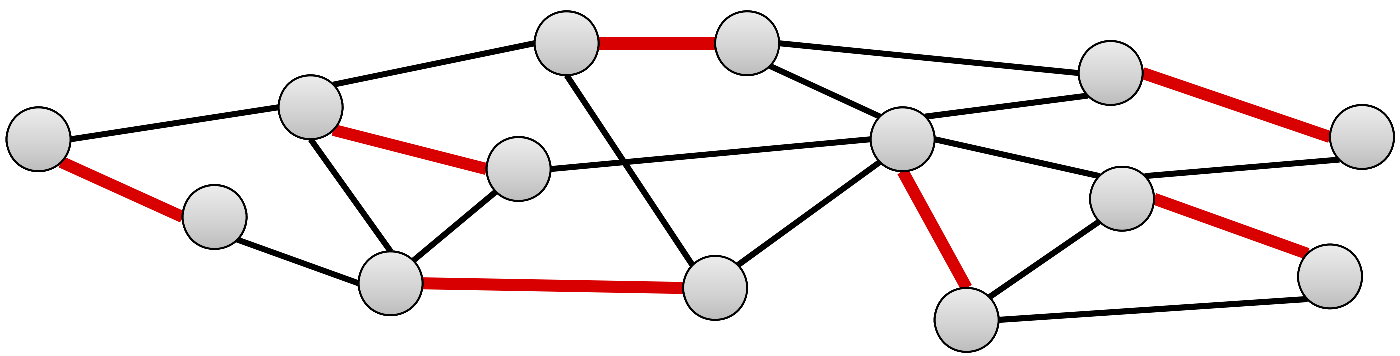
Vertex Cover vs Matching

Consider a matching M and a vertex cover S

Claim: $|M| \leq |S|$

Proof:

- At least one node of every edge $\{u, v\} \in M$ is in S
- Needs to be a different node for different edges from M



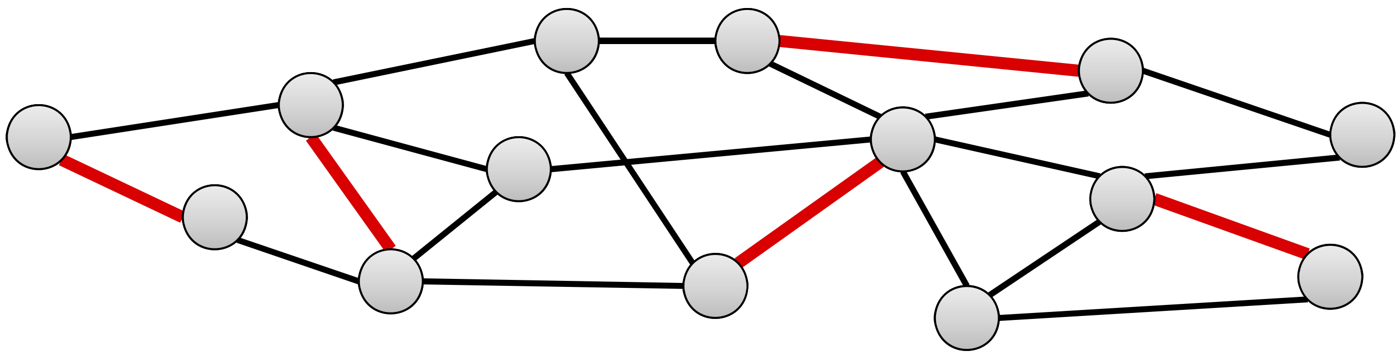
Vertex Cover vs Matching

Consider a matching M and a vertex cover S

Claim: If M is maximal and S is minimum, $|S| \leq 2|M|$

Proof:

- M is maximal: for every edge $\{u, v\} \in E$, either u or v (or both) are matched



- Every edge $e \in E$ is "covered" by at least one matching edge
- Thus, the set of the nodes of all matching edges gives a vertex cover S of size $|S| = 2|M|$.

Maximal Matching Approximation

Theorem: For any maximal matching M and any maximum matching M^*, it holds that

$$|M| \geq \frac{|M^*|}{2}.$$

Proof:

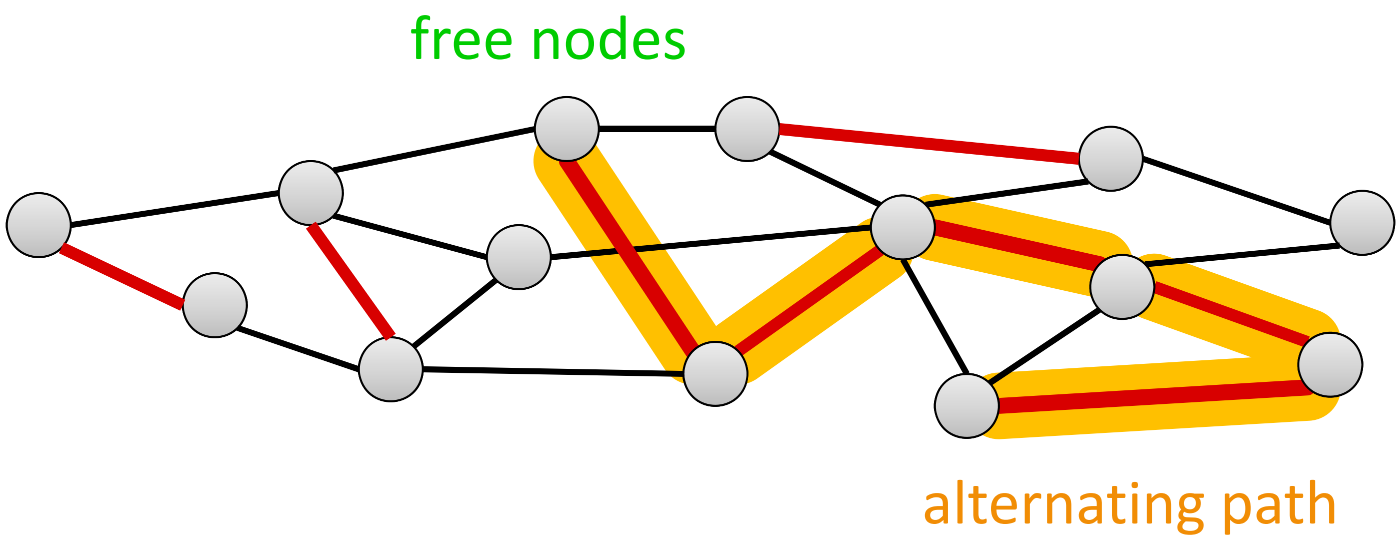
Theorem: The set of all matched nodes of a maximal matching M is a vertex cover of size at most twice the size of a min. vertex cover.

Augmenting Paths

Consider a matching M of a graph $G = (V, E)$:

- A node $v \in V$ is called **free** iff it is not matched

Augmenting Path: A (odd-length) path that starts and ends at a free node and visits edges in $E \setminus M$ and edges in M alternatingly.



- Matching M can be improved using an augmenting path by switching the role of each edge along the path

Augmenting Paths

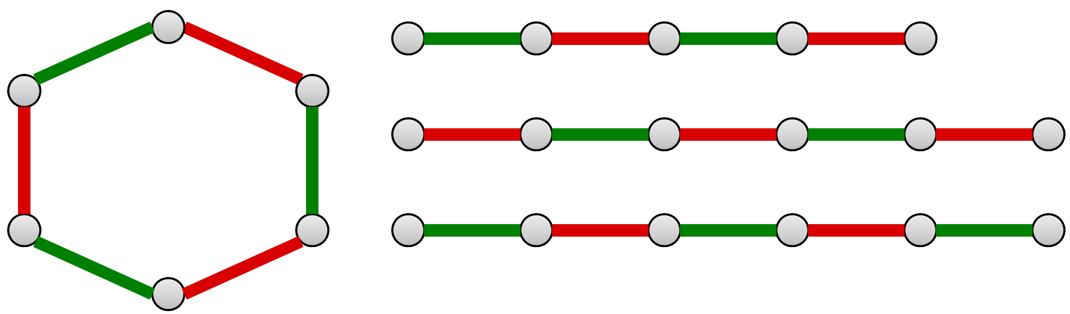
Theorem: A matching M of $G = (V, E)$ is maximum if and only if there is no augmenting path.

Proof:

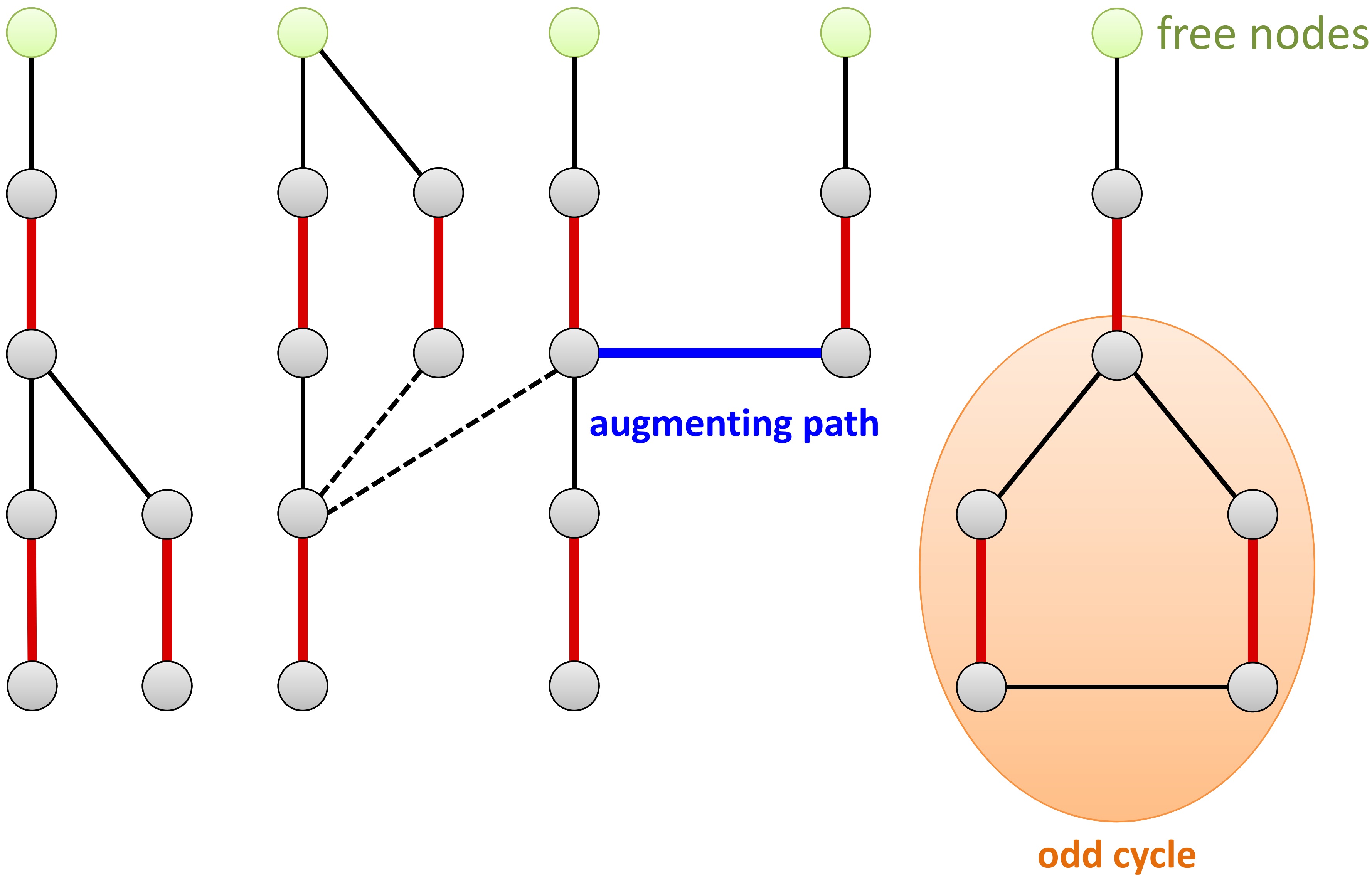
- Consider non-max. matching M and max. matching M^* and define

$$F := M \setminus M^*, \qquad F^* := M^* \setminus M$$

- Note that $F \cap F^* = \emptyset$ and $|F| < |F^*|$
- Each node $v \in V$ is incident to at most one edge in both F and F^*
- $F \cup F^*$ induces even cycles and paths

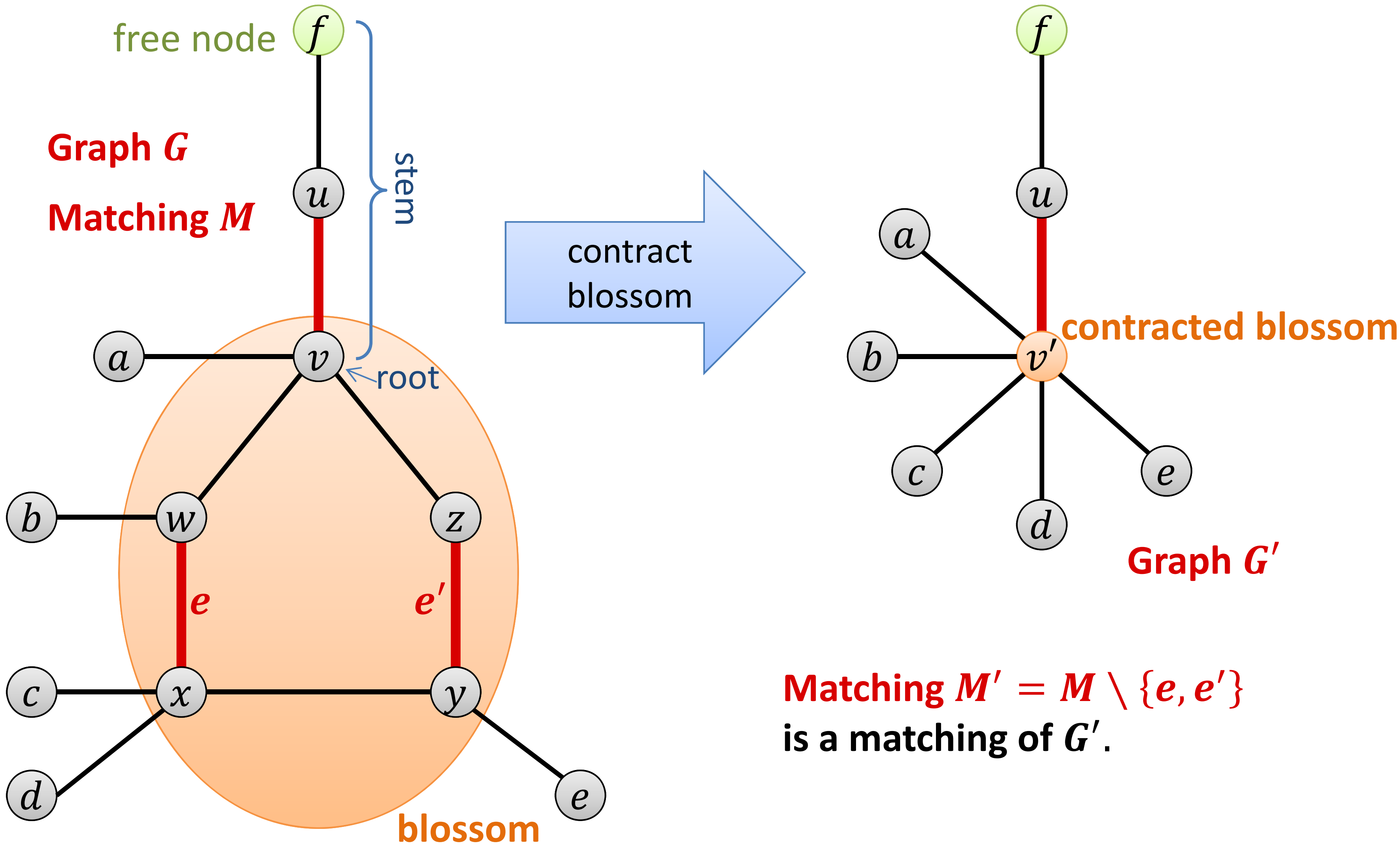


Finding Augmenting Paths



Blossoms

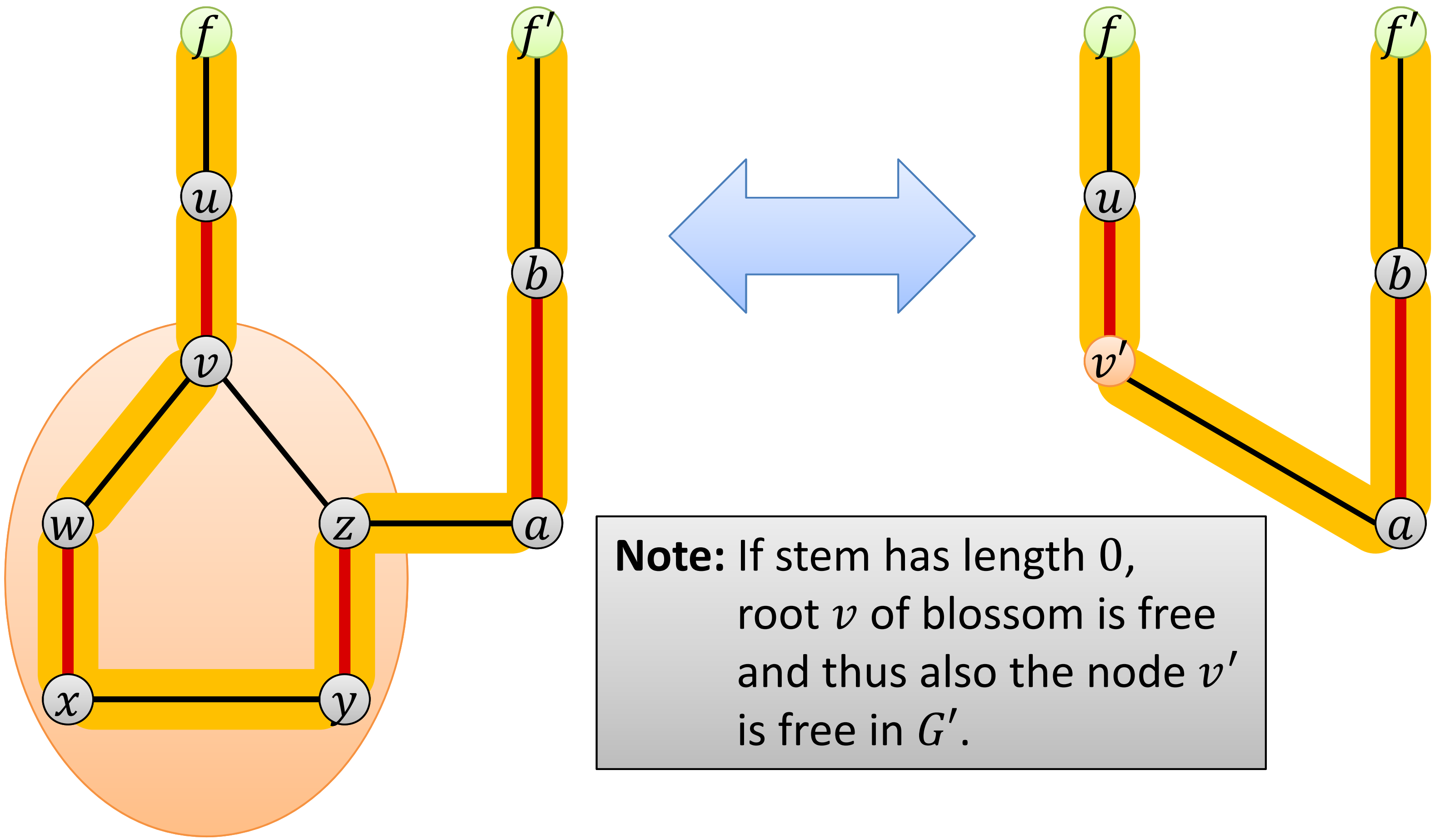
- If we find an odd cycle…



Matching $M' = M \setminus \{e, e'\}$ is a matching of G'.

Contracting Blossoms

Lemma: Graph G has an augmenting path w.r.t. matching M iff G' has an augmenting path w.r.t. matching M'



Note: If stem has length 0, root v of blossom is free and thus also the node v' is free in G'.

Also: The matching M can be computed efficiently from M'.

Edmond's Blossom Algorithm

Algorithm Sketch:

1. Build a tree for each free node
2. Starting from an explored node u at even distance from a free node f in the tree of f, explore some unexplored edge $\{u, v\}$:

    1. If v is an unexplored node, v is matched to some neighbor w: add w to the tree (w is now explored)

    2. If v is explored and in the same tree:
    at odd distance from root → ignore and move on
    at even distance from root → **blossom found**

    3. If v is explored and in another tree
    at odd distance from root → ignore and move on
    at even distance from root → **augmenting path found**

Running Time

Finding a Blossom: Repeat on smaller graph

Finding an Augmenting Path: Improve matching

Theorem: The algorithm can be implemented in time $O(mn^2)$.

Matching Algorithms

We have seen:

- $\boldsymbol{O(mn)}$ time alg. to compute a max. matching in *bipartite graphs*
- $\boldsymbol{O(mn^2)}$ time alg. to compute a max. matching in *general graphs*

Better algorithms:

- Best known running time (bipartite and general gr.): $\boldsymbol{O(m\sqrt{n})}$

Weighted matching:

- Edges have weight, find a matching of **maximum total weight**
- *Bipartite graphs*: flow reduction works in the same way
- *General graphs*: can also be solved in polynomial time
  (Edmond's algorithms is used as blackbox)

Happy Holidays!