CS630 Graduate Algorithms

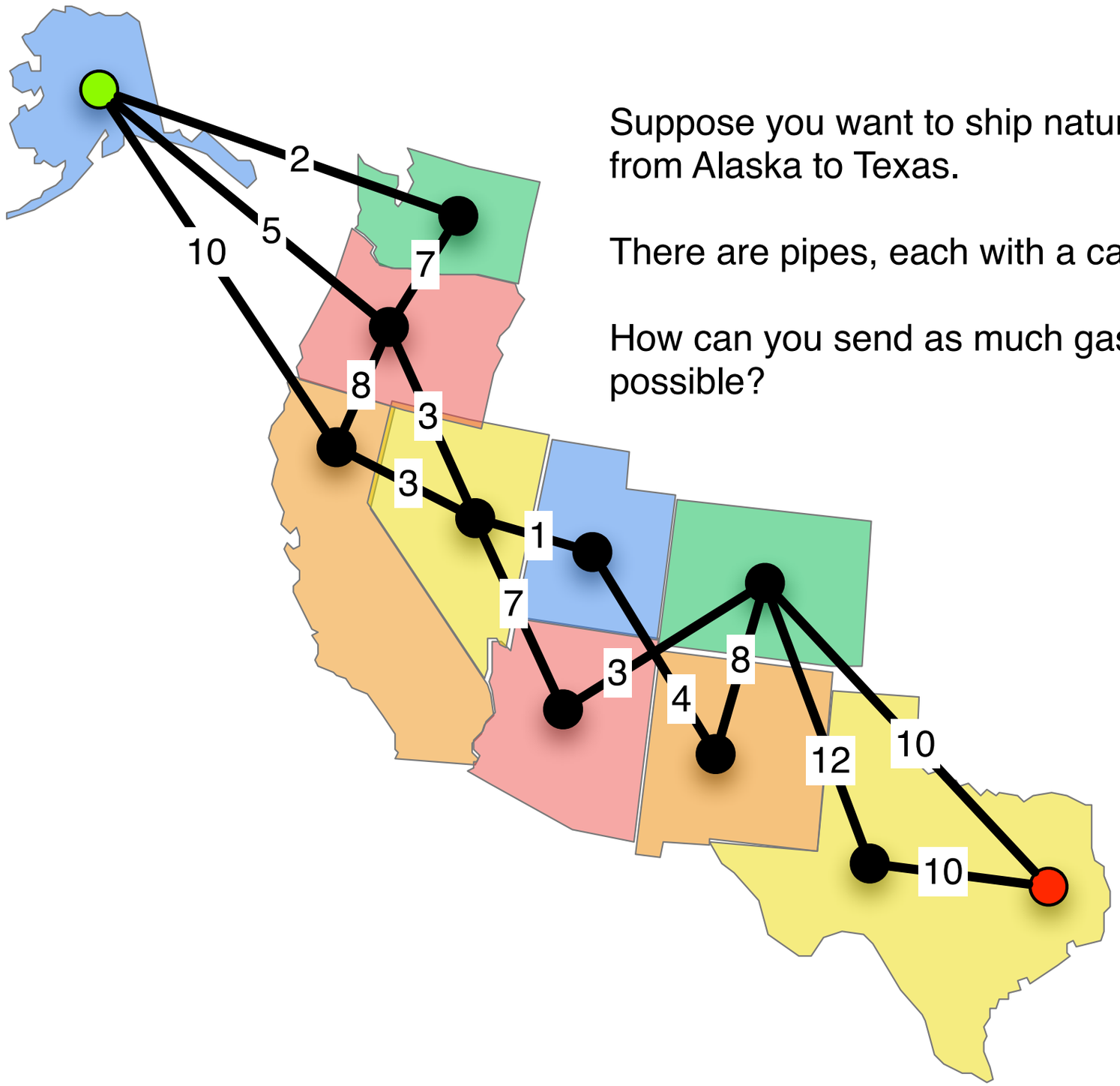
September 5, 2024

by Dora Erdos and Jeffrey Considine

Today:

- Maximum flow problems
- Reductions to maximum flow

Shipping through a pipeline



Suppose you want to ship natural gas from Alaska to Texas.

There are pipes, each with a capacity.

How can you send as much gas as possible?

image by Carl Kingsford , Univ. of Maryland, College Park

Network flow applications

Road network, in which each street has a throughput of how many cars can pass in a minute.

Communication network in which certain pairs of nodes are connected by communication links, each with a limit of how many Mb/s it can transmit. How much data can be sent through the network?

- Also called “bottleneck bandwidth” or “bisection bandwidth”.

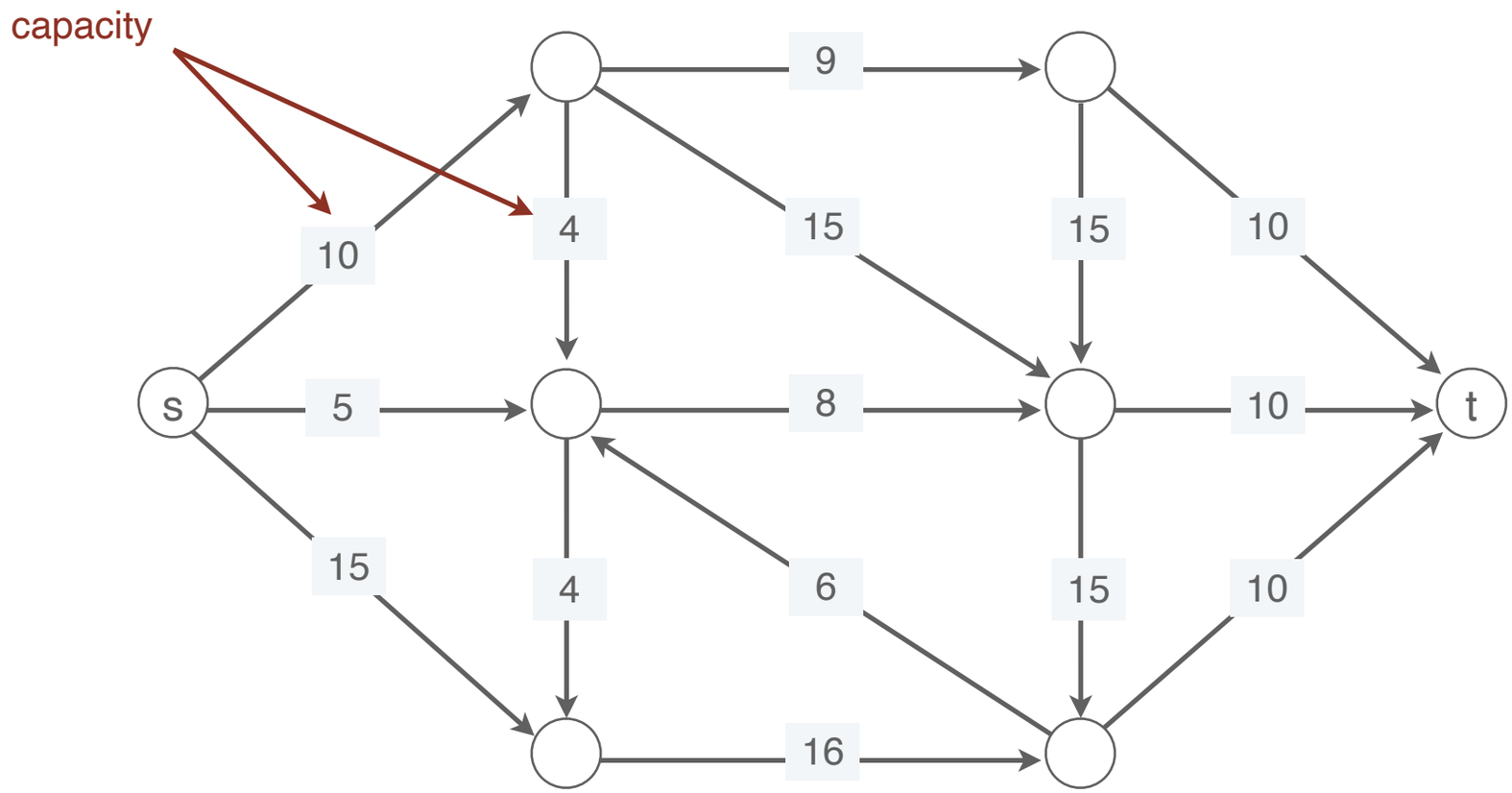
Supply chain network, in which goods are being shipped from factories to stores through a network of warehouses and trucks with given capacity carrying goods from one location to another.

- Amazon and logistics companies in general do a lot of this.
- More complicated with different kinds of goods.

Network flow - intuition

A flow network is a tuple $G = (V, E, s, t, c)$.

- Directed graph (digraph) (V, E) with source $s \in V$ and sink $t \in V$.
- Non-negative capacity $c(e)$ for each $e \in E$.
  - intuition: the capacity is the *throughput* of each edge

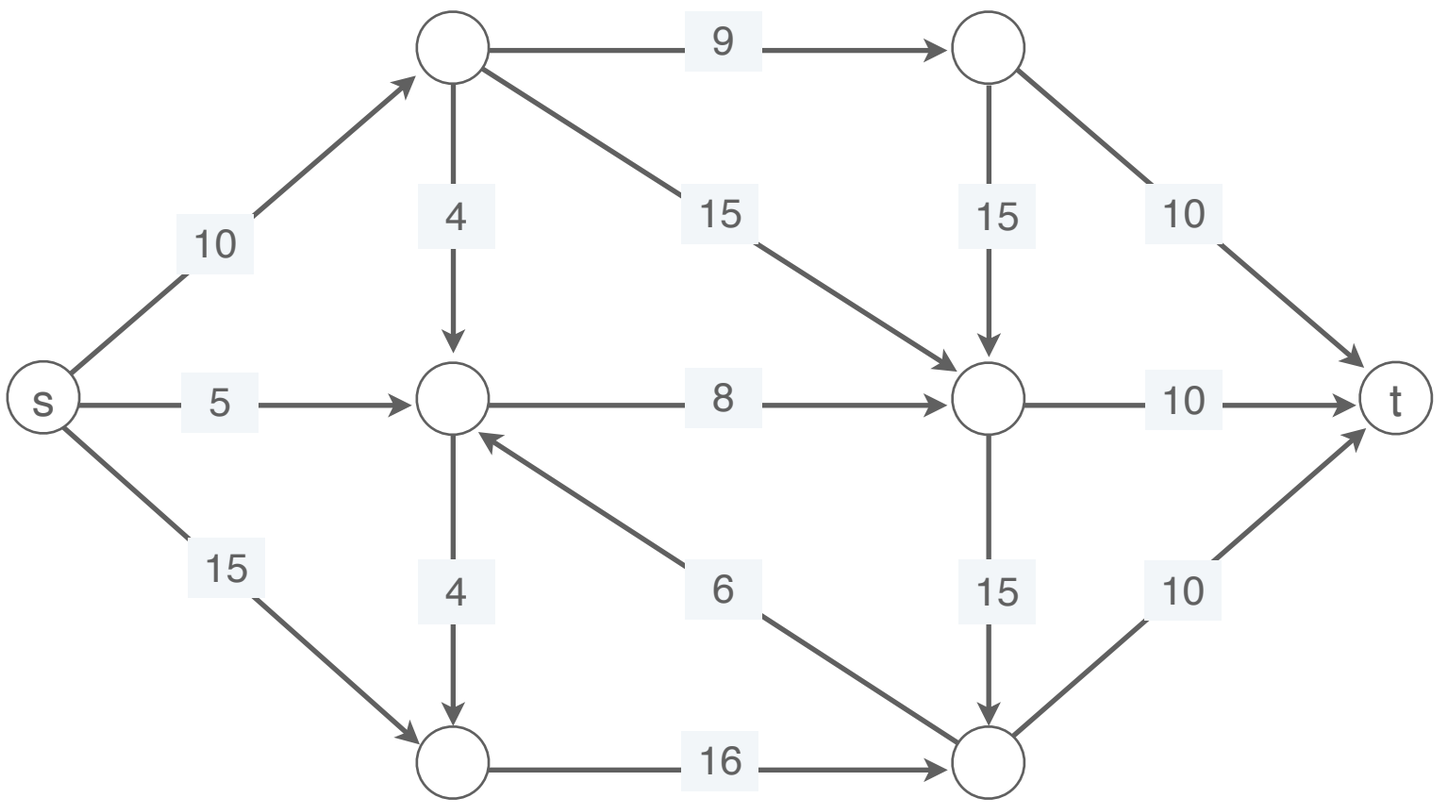


Network flow - intuition

A flow network is a tuple $G = (V, E, s, t, c)$.

- Directed graph (digraph) (V, E) with source $s \in V$ and sink $t \in V$.
- Non-negative capacity $c(e)$ for each $e \in E$.
  - intuition: the capacity is the *throughput* of each edge

st-flow. intuition: the amount of matter that can be sent from s to t through the network given the capacity of each edge.



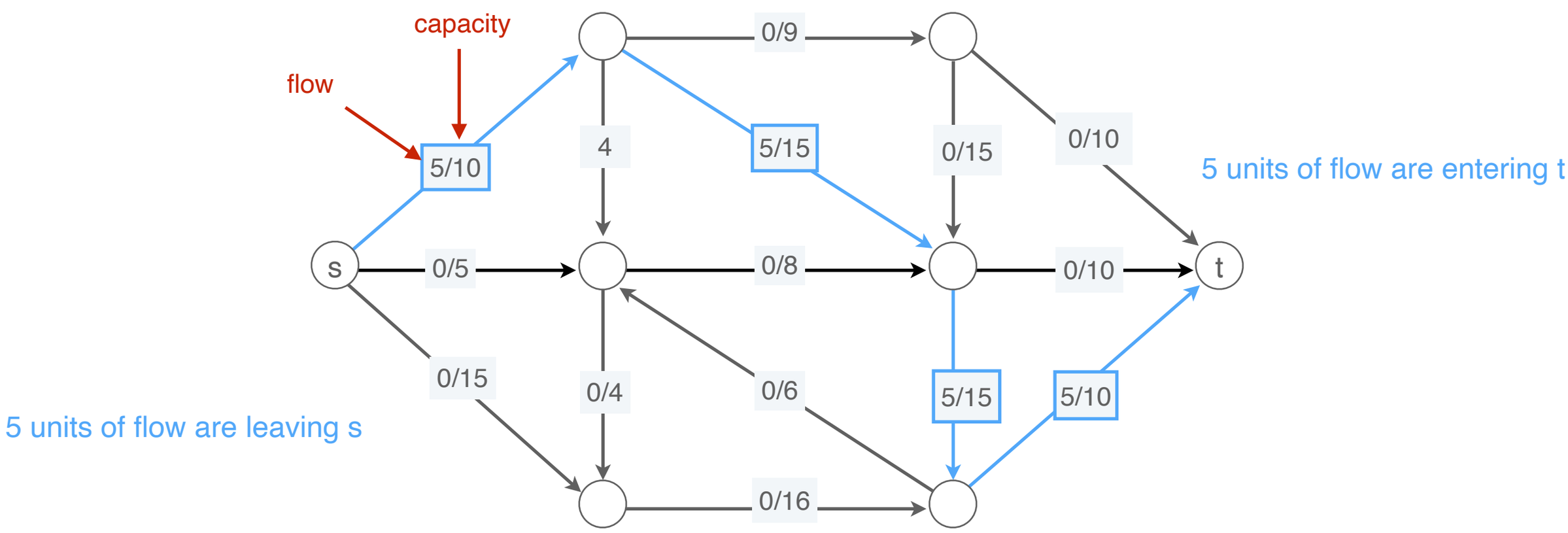
Network flow - intuition

A flow network is a tuple $G = (V, E, s, t, c)$.

- Directed graph (digraph) (V, E) with source $s \in V$ and sink $t \in V$.
- Non-negative capacity $c(e)$ for each $e \in E$.

st-flow. A *function* f on the edges, f(u,v) is the amount of flow on directed edge (u,v).

properties:



Network flow - intuition

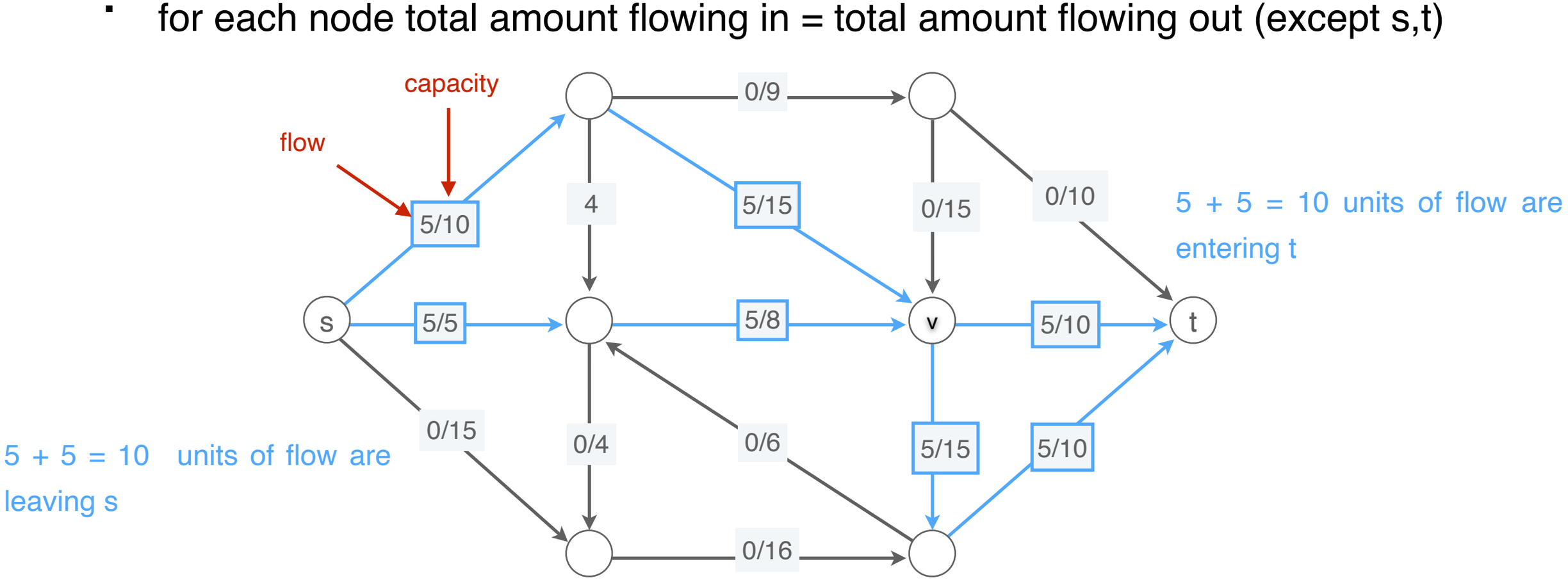
A flow network is a tuple $G = (V, E, s, t, c)$.

- Directed graph (digraph) (V, E) with source $s \in V$ and sink $t \in V$.
- Non-negative capacity $c(e)$ for each $e \in E$.

st-flow. A *function* f on the edges, f(u,v) is the amount of flow on directed edge (u,v).

properties:

- upper bounded by the capacity on each edge
- for each node total amount flowing in = total amount flowing out (except s,t)



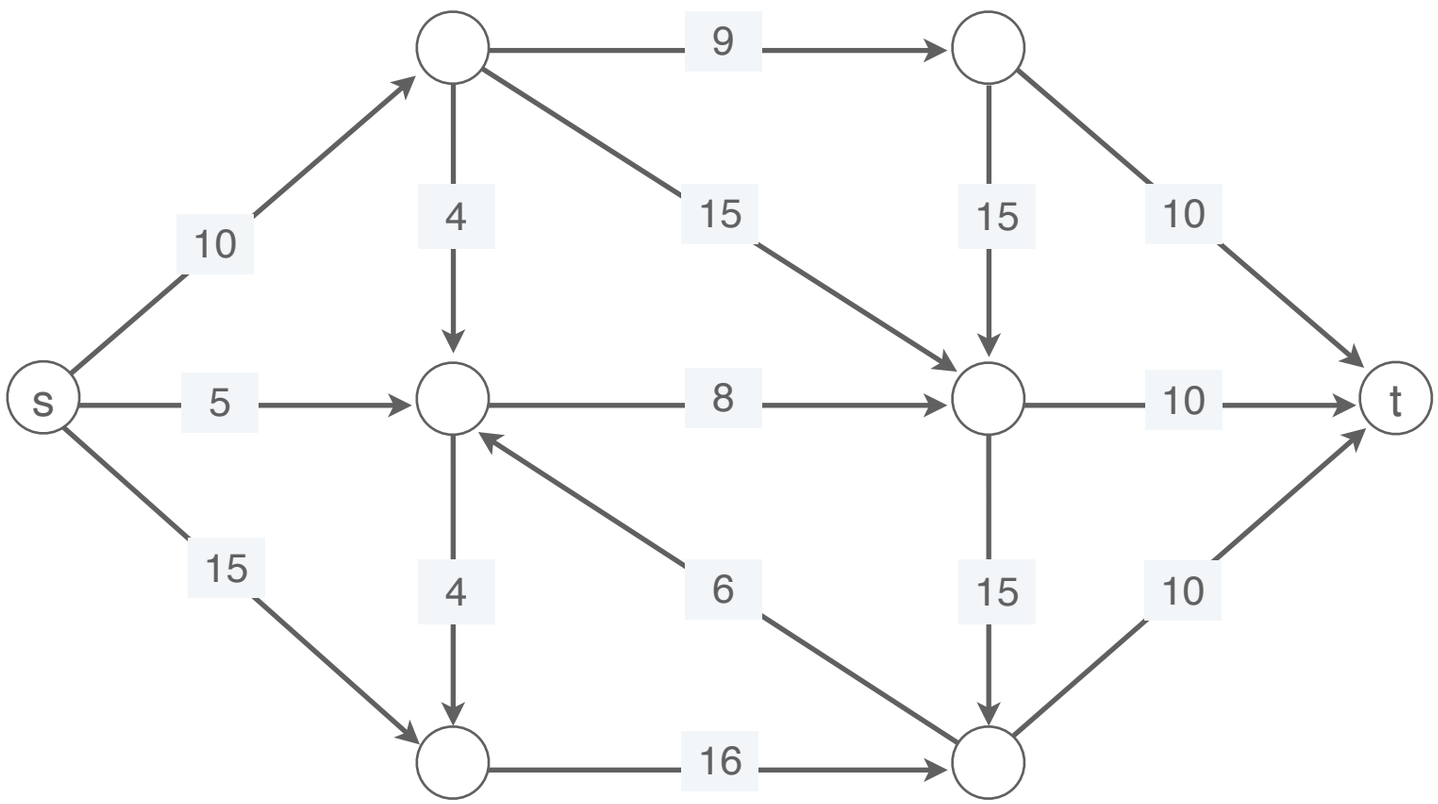
Network flow - intuition

A flow network is a tuple $G = (V, E, s, t, c)$.

- Directed graph (digraph) (V, E) with source $s \in V$ and sink $t \in V$.
- Non-negative capacity $c(e)$ for each $e \in E$.

st-flow. A *function* f on the edges, f(u,v) is the amount of flow on directed edge (u,v).

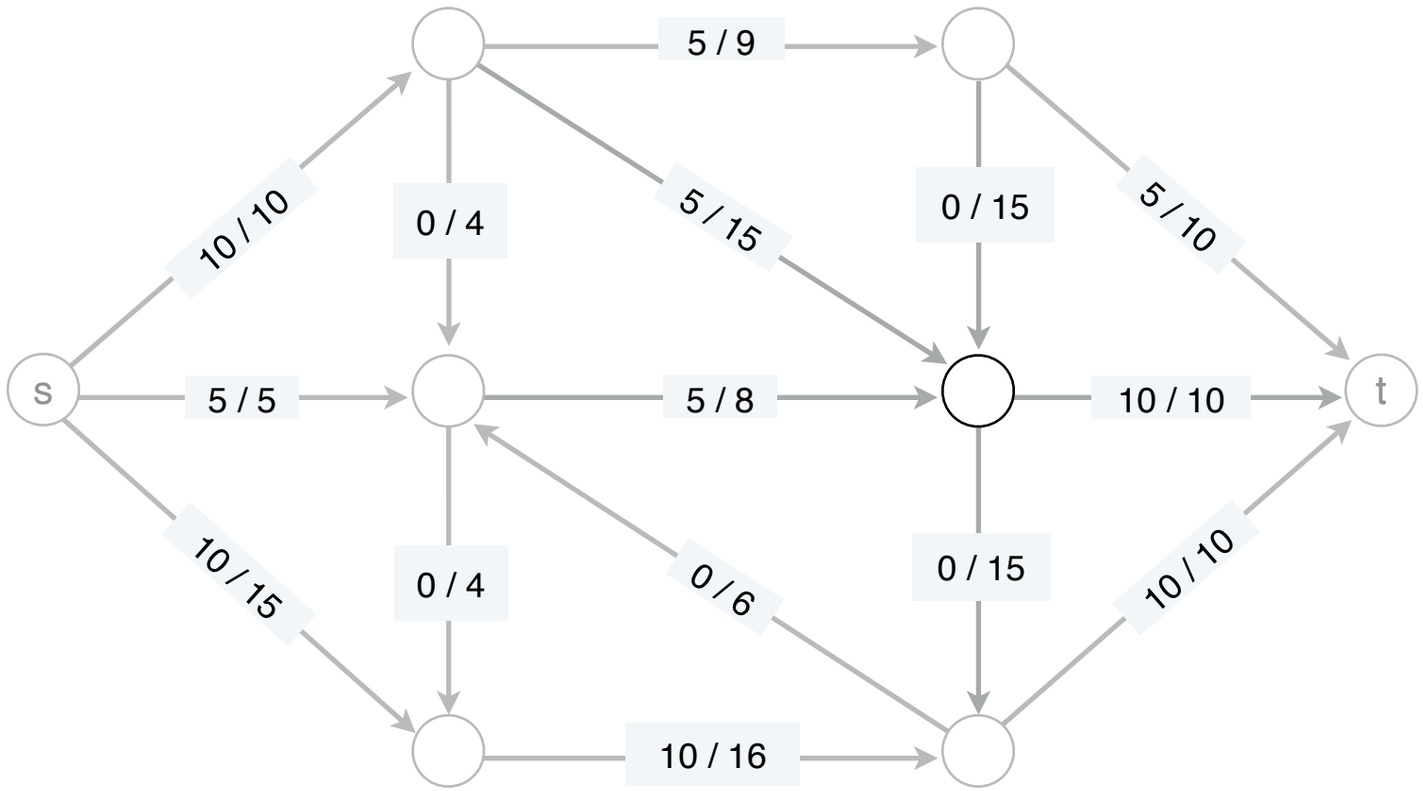
Value of the flow: Total amount of flow from s to t.



Network flow

Def. An st-flow (flow) f is a function that satisfies:

- For each $e \in E$: $0 \leq f(e) \leq c(e)$ [capacity]
- For each $v \in V - \{s, t\}$: $\sum_{e \text{ in to } v} f(e) = \sum_{e \text{ out of } v} f(e)$ [flow conservation]

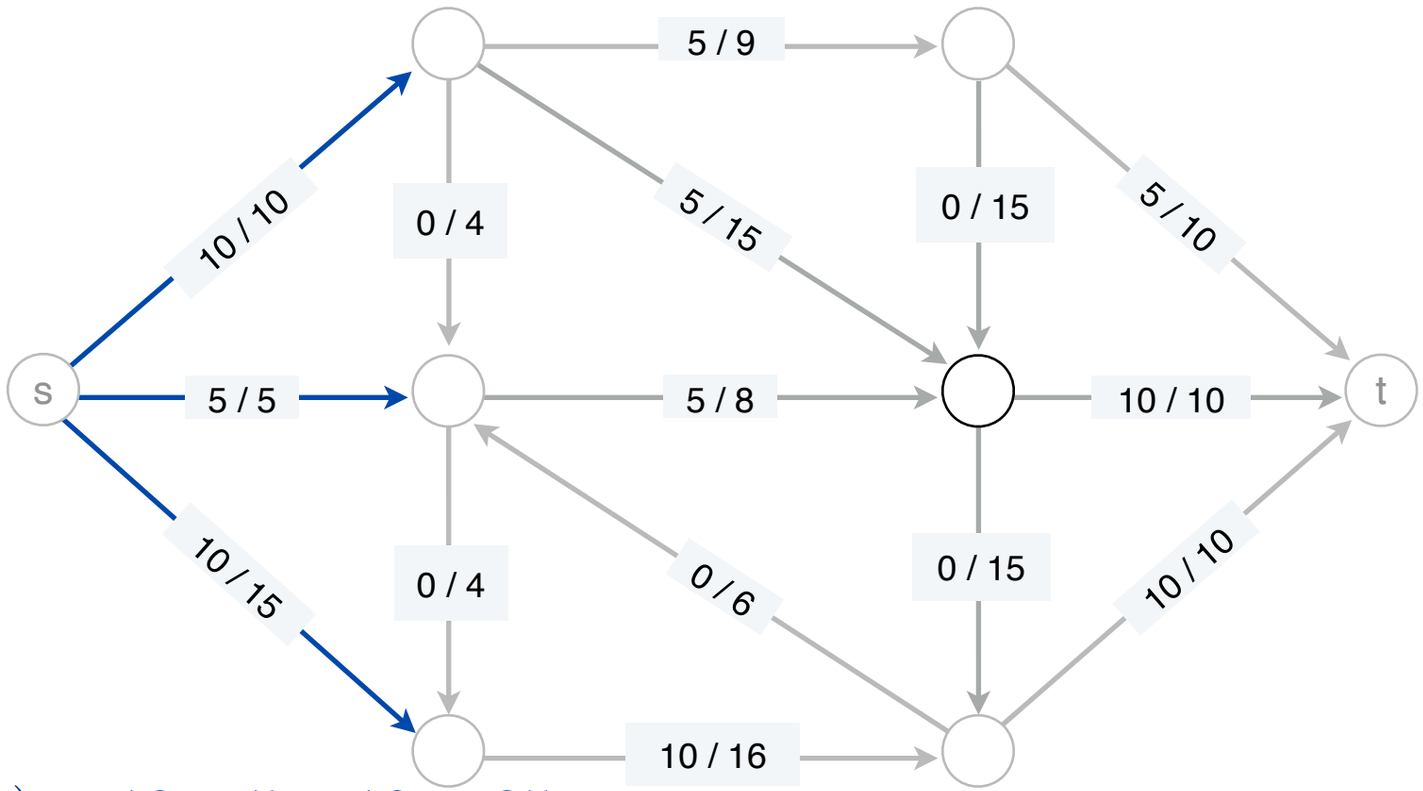


Network flow

Def. An st-flow (flow) f is a function that satisfies:

- For each $e \in E$: $0 \leq f(e) \leq c(e)$ [capacity]
- For each $v \in V - \{s, t\}$: $\sum_{e \text{ in to } v} f(e) = \sum_{e \text{ out of } v} f(e)$ [flow conservation]

Def. The value of a flow f is: $val(f) = \sum_{edges\ (s,u)} f(s,u)$



$val(f) = f_{out}(s) = 10 + 5 + 10 = 25$

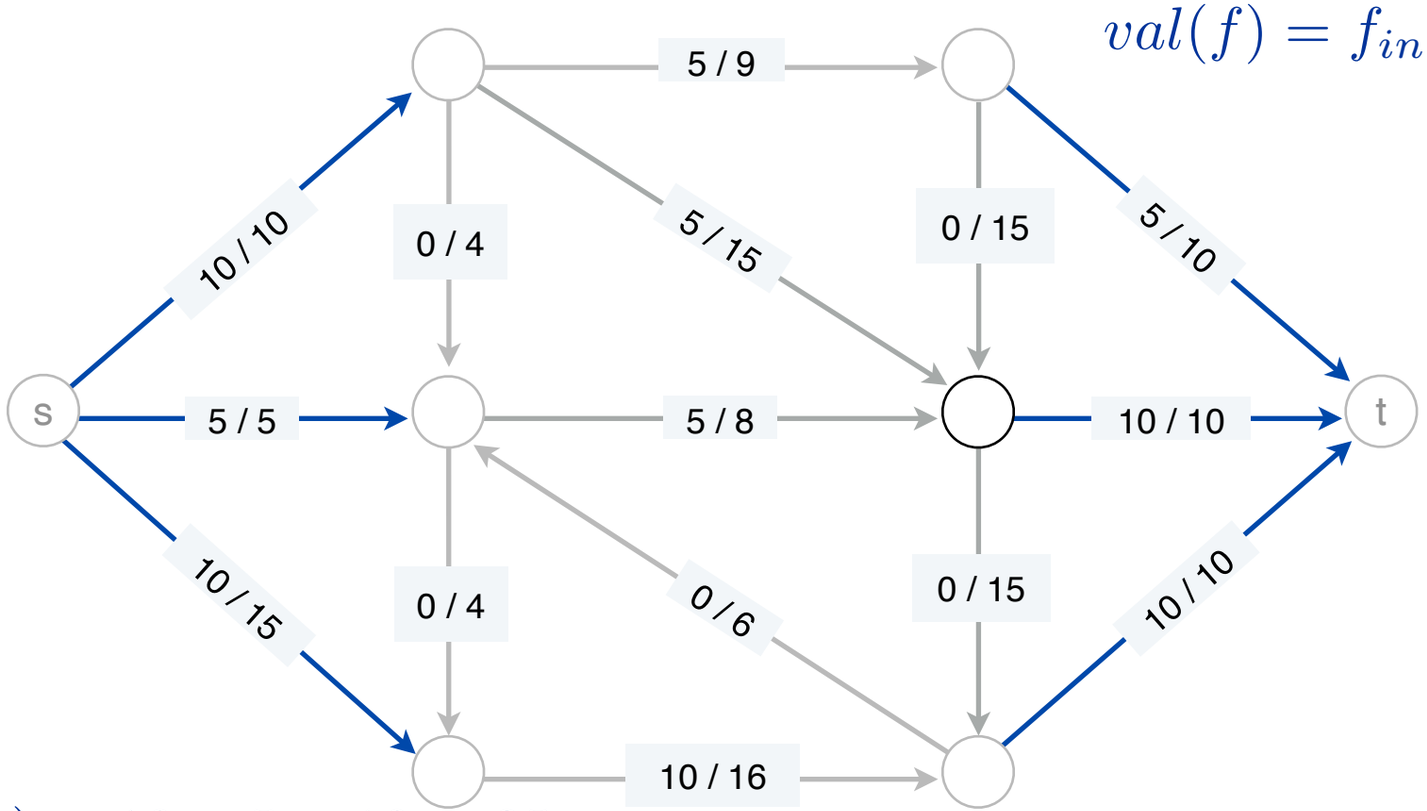
Network flow

Def. An st-flow (flow) f is a function that satisfies:

- For each $e \in E$: $0 \leq f(e) \leq c(e)$ [capacity]
- For each $v \in V - \{s, t\}$: $\sum_{e \text{ in to } v} f(e) = \sum_{e \text{ out of } v} f(e)$ [flow conservation]

Def. The value of a flow f is: $val(f) = \sum_{edges\ (s,u)} f(s,u) = \sum_{edges\ (v,t)} f(v,t)$

why?



$val(f) = f_{in}(t) = 5 + 10 + 10 = 25$

$val(f) = f_{out}(s) = 10 + 5 + 10 = 25$

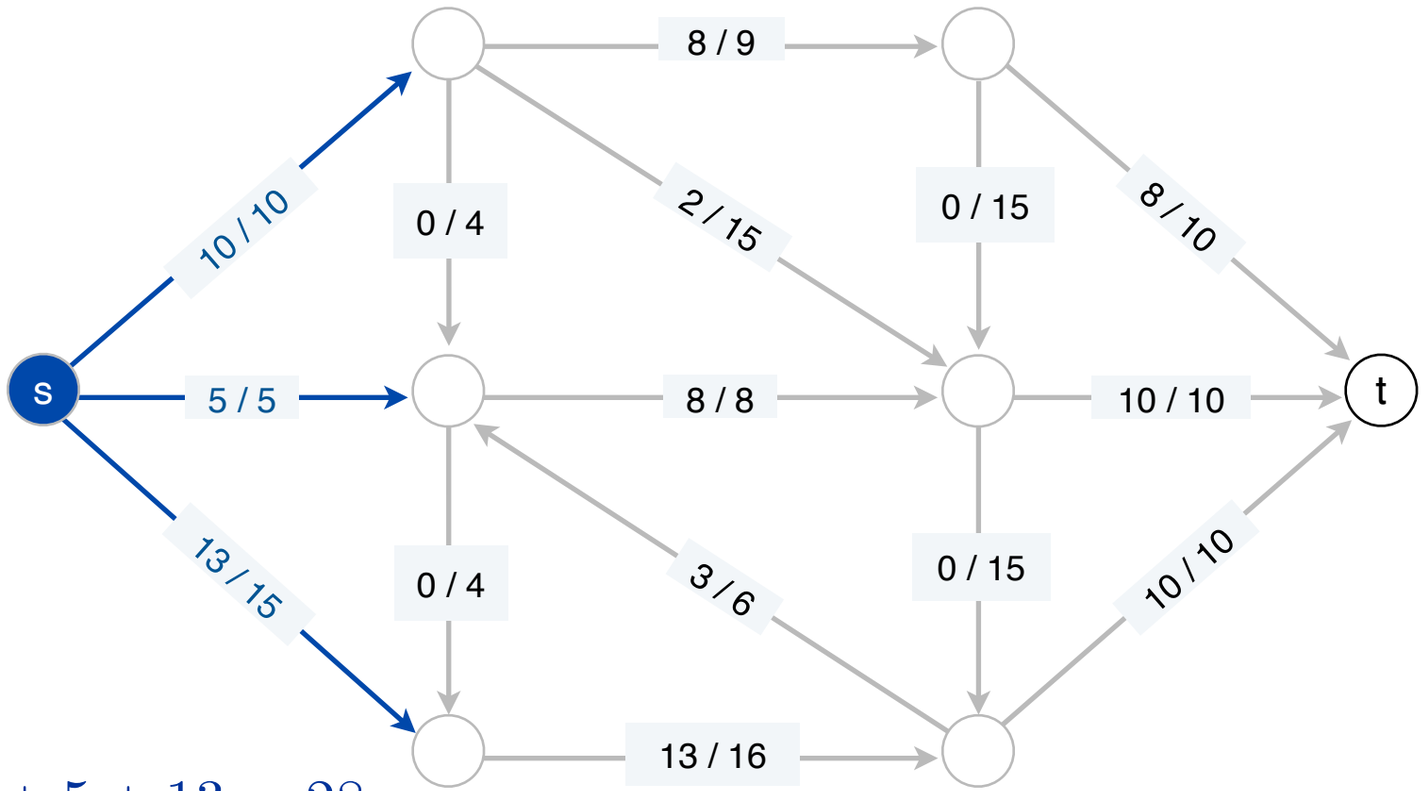
Maximum-flow problem

Def. An st-flow (flow) f is a function that satisfies:

- For each $e \in E$: $0 \le f(e) \le c(e)$ [capacity]
- For each $v \in V - \{s, t\}$: $\sum_{e \text{ in to } v} f(e) = \sum_{e \text{ out of } v} f(e)$ [flow conservation]

Def. The value of a flow f is: $val(f) = \sum_{edges\ (s,u)} f(s,u) = \sum_{edges\ (v,t)} f(v,t)$

new value of flow in this network:



$val(f) = 10 + 5 + 13 = 28$

Maximum-flow problem

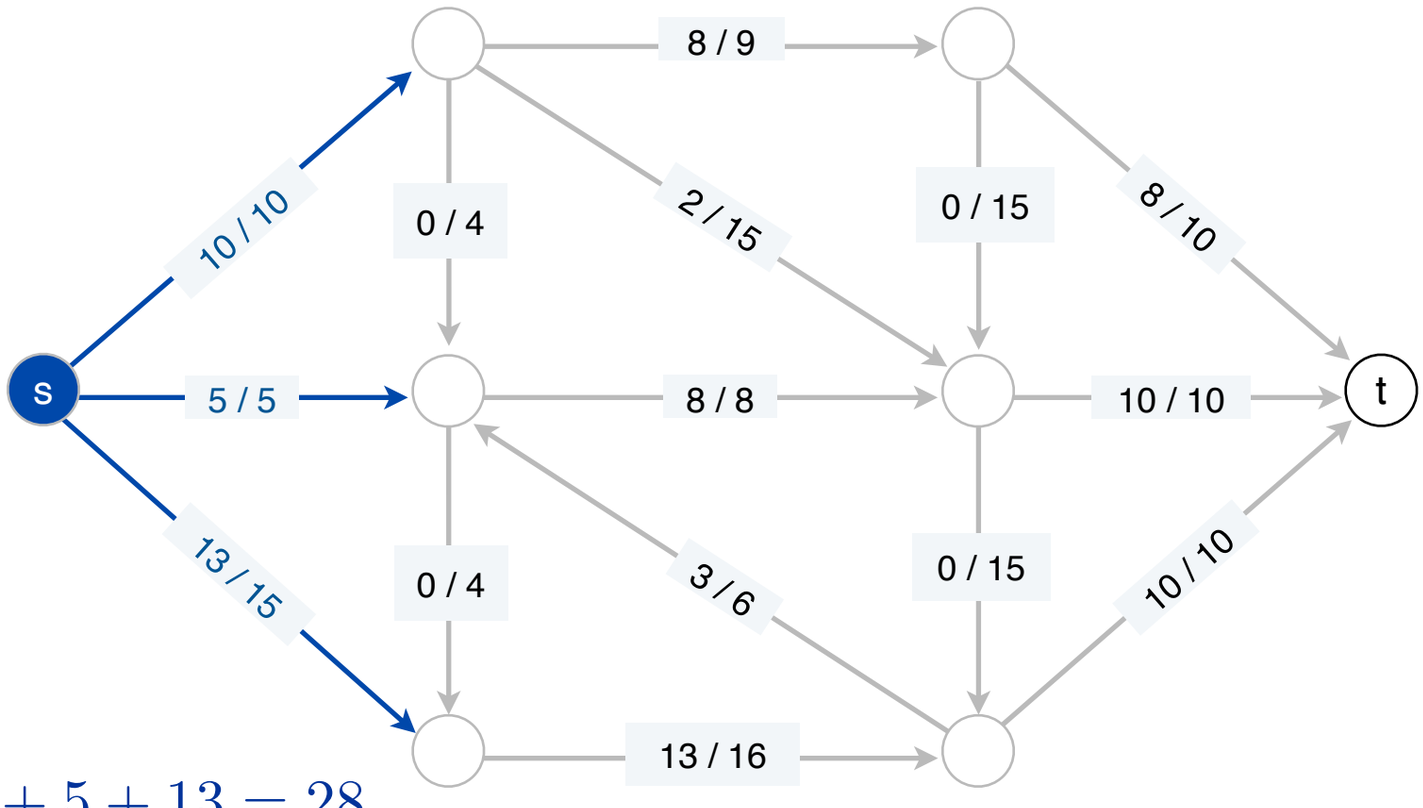
Def. An st-flow (flow) f is a function that satisfies:

- For each $e \in E$: $0 \le f(e) \le c(e)$ [capacity]
- For each $v \in V - \{s, t\}$: $\sum_{e \text{ in to } v} f(e) = \sum_{e \text{ out of } v} f(e)$ [flow conservation]

Def. The value of a flow f is: $val(f) = \sum_{edges\ (s,u)} f(s,u) = \sum_{edges\ (v,t)} f(v,t)$

Max-Flow problem:

Given a directed graph with edge capacities, find the maximum value flow.



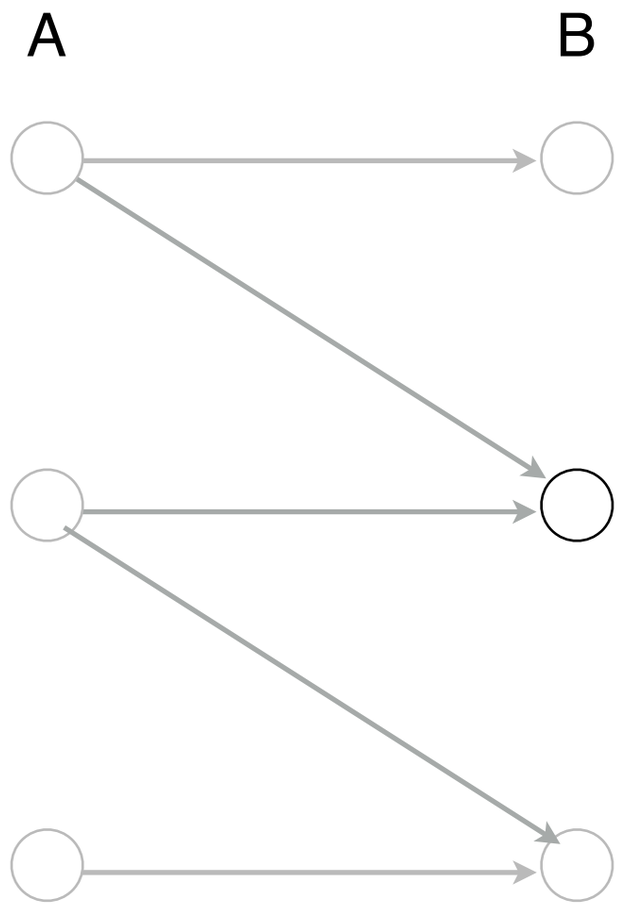
$val(f) = 10 + 5 + 13 = 28$

Pause for TopHat Quiz

More network flow applications - Maximum bipartite matching

Given a bipartite graph, pick a maximum set of edges with non-overlapping vertices.

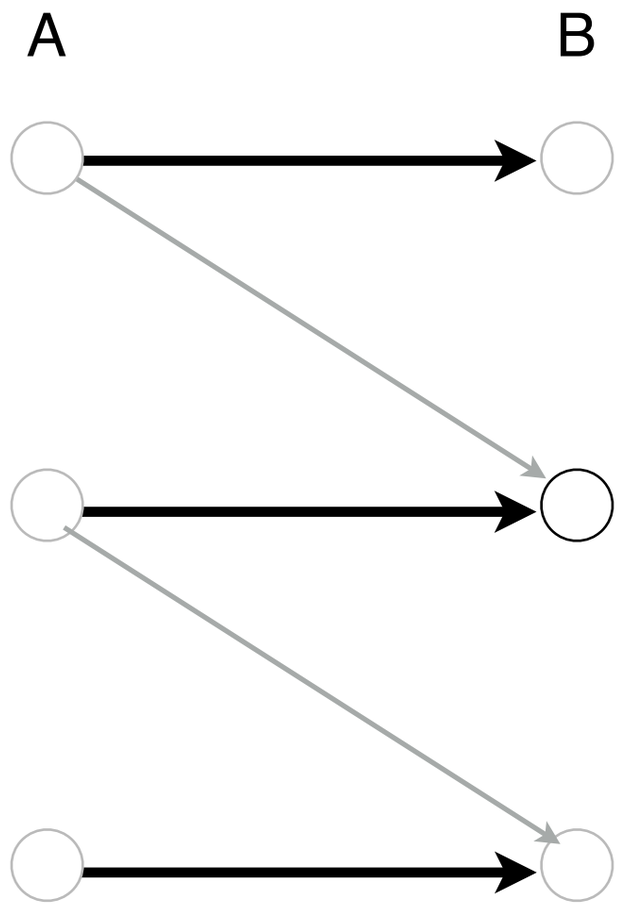
Sometimes called a marriage problem, but not the stable marriage problem.



More network flow applications - Maximum bipartite matching

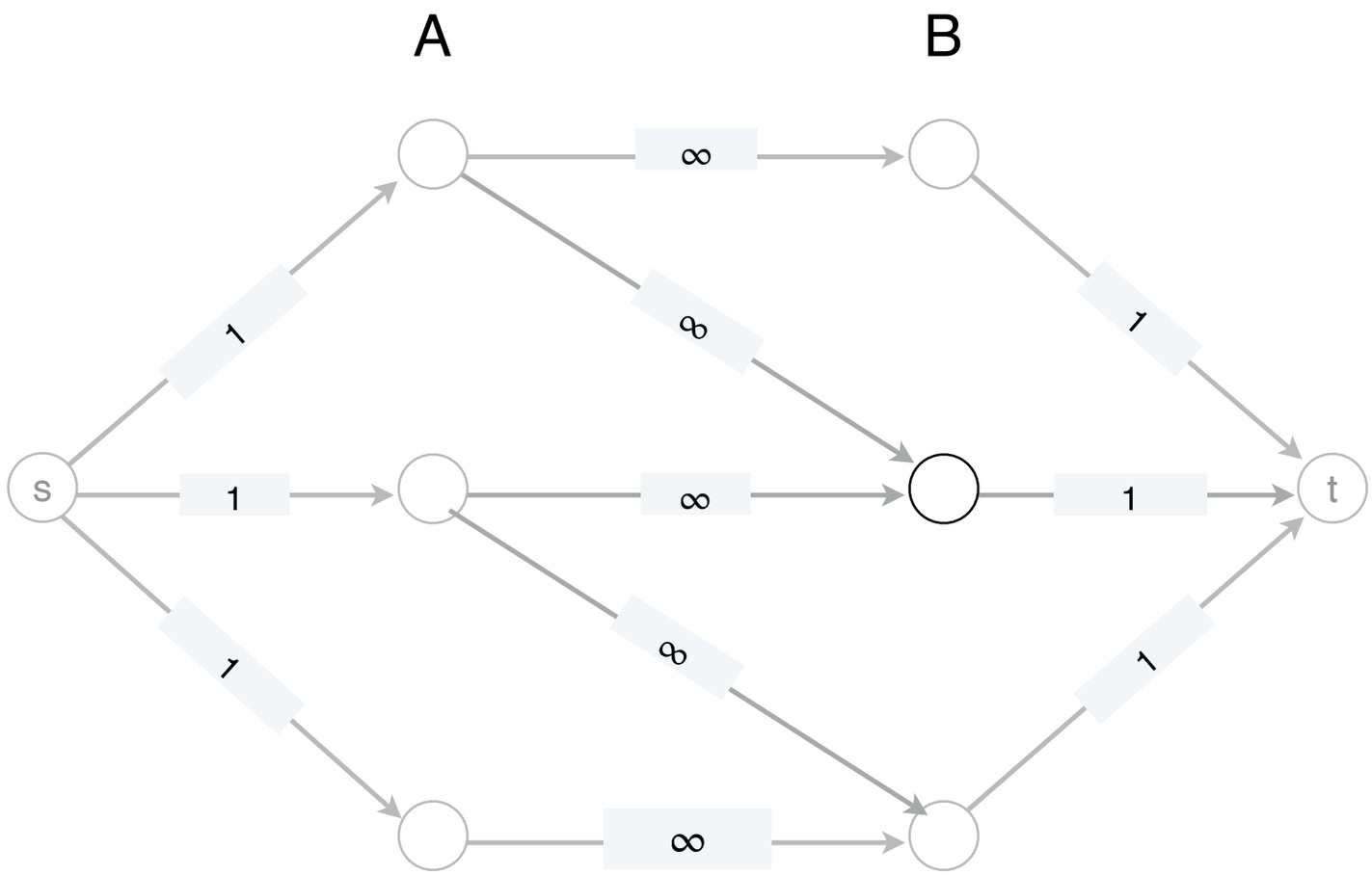
Given a bipartite graph, pick a maximum set of edges with non-overlapping vertices.

Sometimes called a marriage problem, but not the stable marriage problem.



Bipartite matching reduction to maximum network flow

This maximum network flow problem has the same answer as the previous maximum bipartite matching problem.

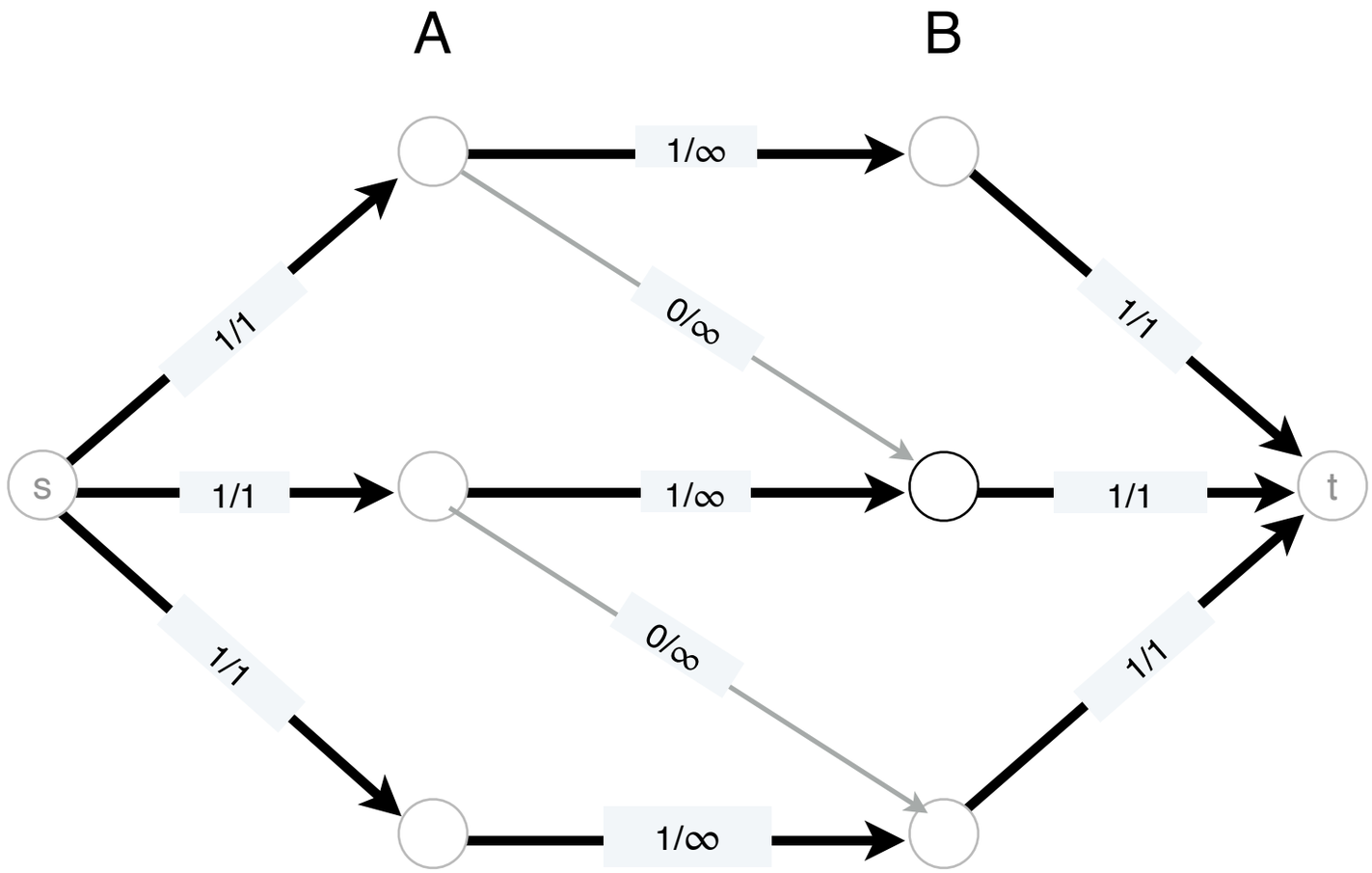


Integral flow theorem:

If each edge in a flow network has integral capacity, then there exists an integral maximal flow. Meaning each edge has an integral flow value.

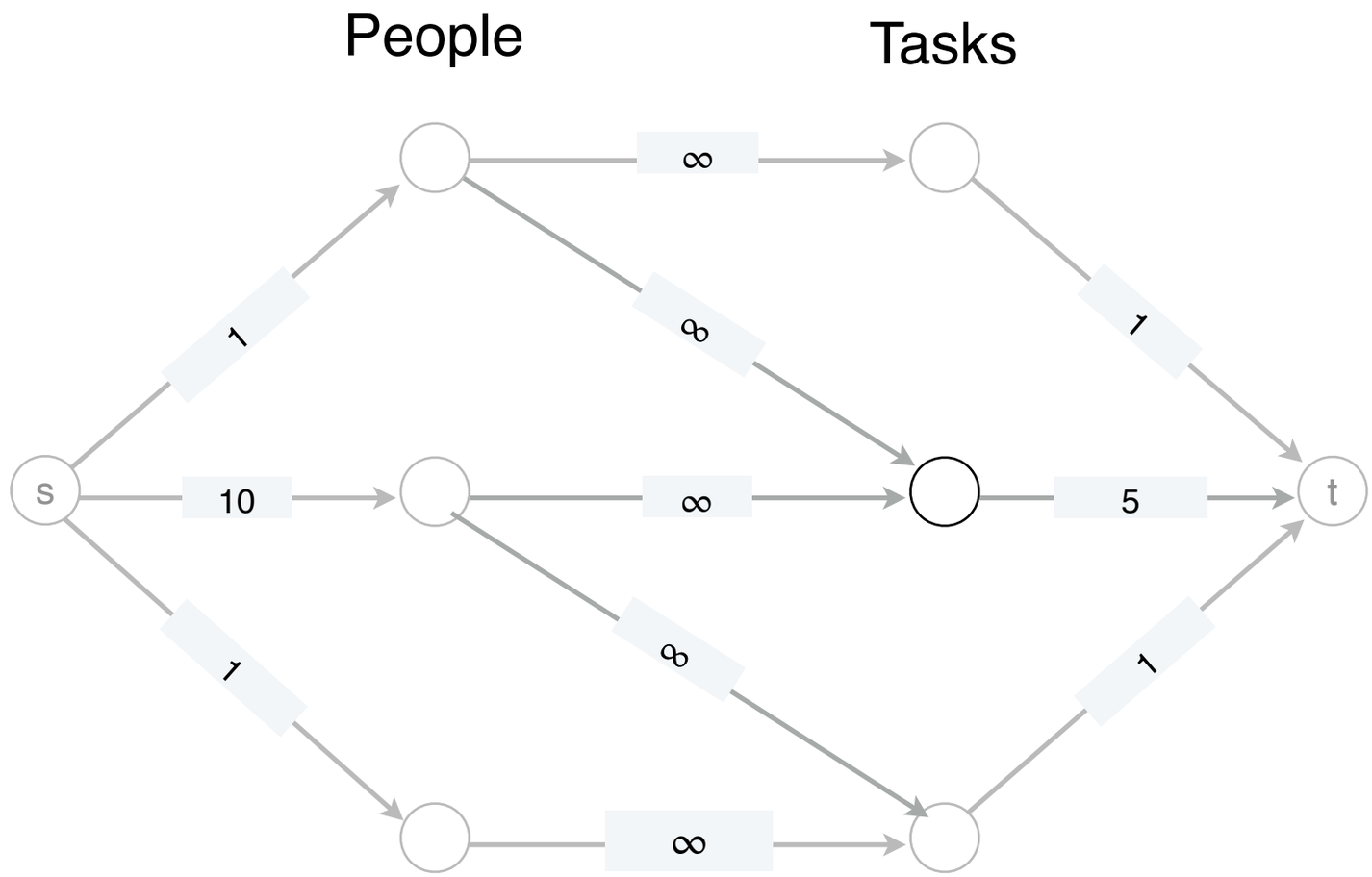
Bipartite matching reduction to maximum network flow

The flow value of the maximum network flow problem is the size of the maximal matching of the bipartite graph. If the flow is an integral maximal flow, then the edges between A and B with flow > 0 are a maximal matching.



Task assignments reduction to network flow

Modify bipartite matching to make sides represent people and tasks, and add weights to people or tasks.



- Setting weights on both side will let people share tasks.
- Have only heard of this actually being done with batch jobs and computers, not people.

How to find a maximum flow?

Max-Flow problem:

Given a directed graph with edge capacities, find the maximum value flow.

"Find" flow = assign valid flow values to each edge.

- valid = satisfies the capacity and flow conservation constraints

Algorithms:

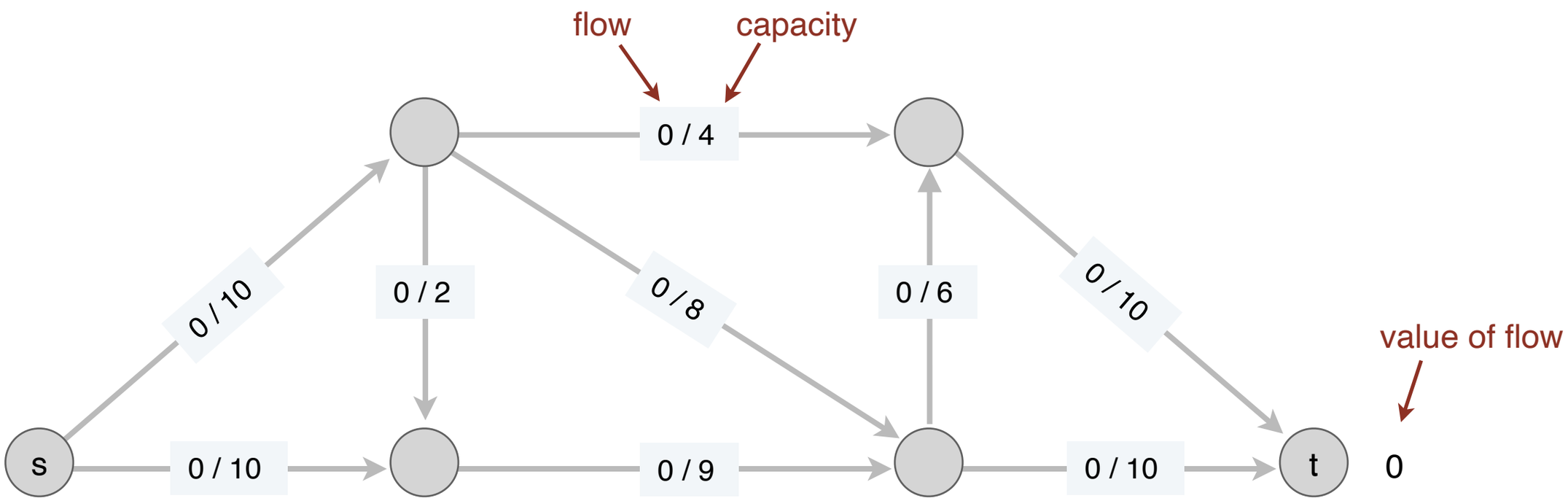
- Ford-Fulkerson
  - "augmenting paths" - intuitive, can be very slow on some input
  - speed up - capacity scaling algorithm, Edmonds-Karp algorithm

- we always analyze the running time of our algorithms
- always prove their correctness — what does it mean to be correct?

Augmenting paths

Observation: we can send additional flow along an st-path if there is free capacity along every edge.

Augmenting path: an st-path with free capacity, along which we augment (= increase) the flow.



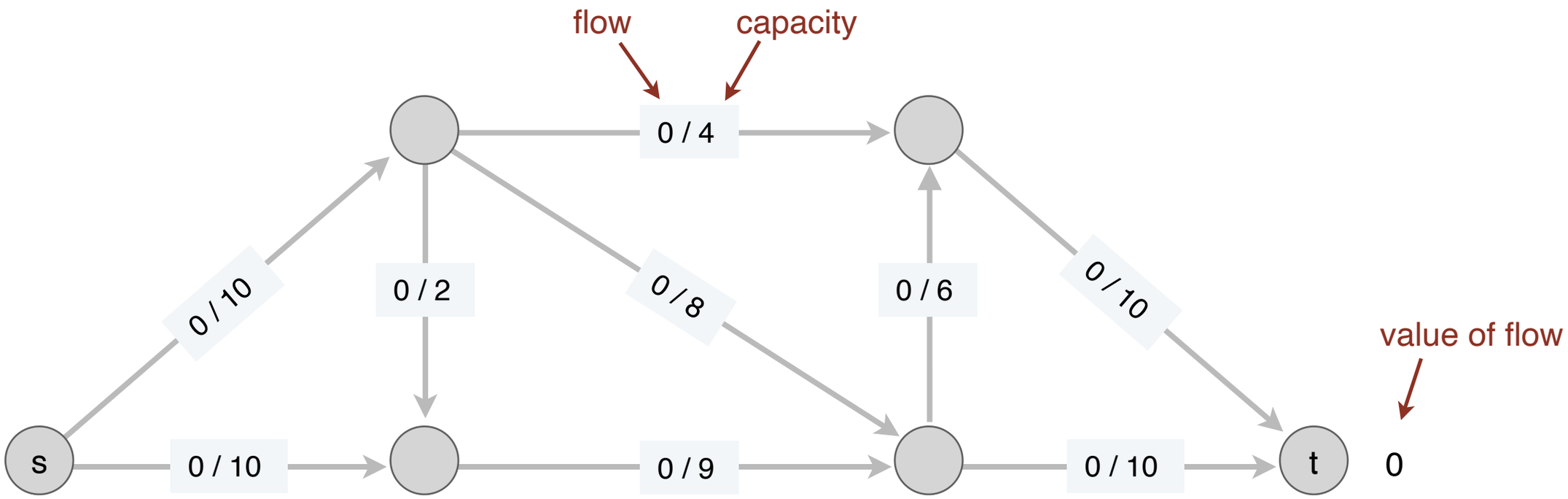
Augmenting paths

Observation: we can send additional flow along an st-path if there is free capacity along every edge.

Augmenting path: an st-path with free capacity, along which we augment the flow.

```
GreedyFlow(G(V,E,c)):

1.  While there is an augmenting path P:
2.     increase flow along P by the max available
3.  Return f along each edge
```

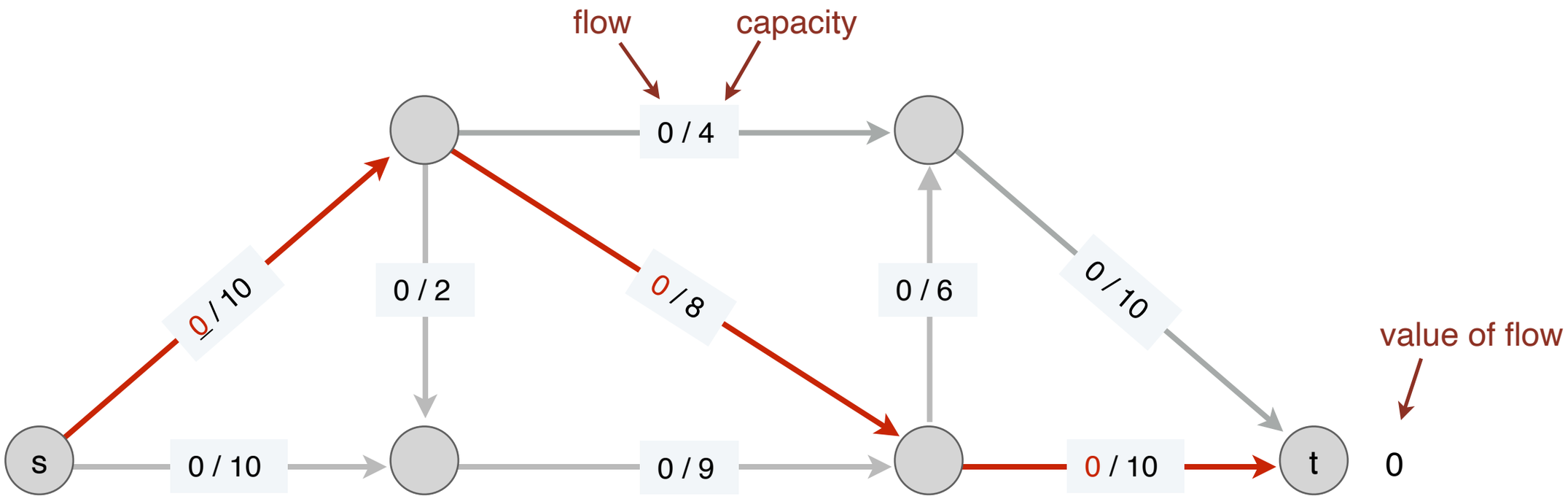


Augmenting paths

Observation: we can send additional flow along an st-path if there is free capacity along every edge.

Augmenting path: an st-path with free capacity, along which we augment the flow.

We can augment the flow by 8 units along the highlighted path.

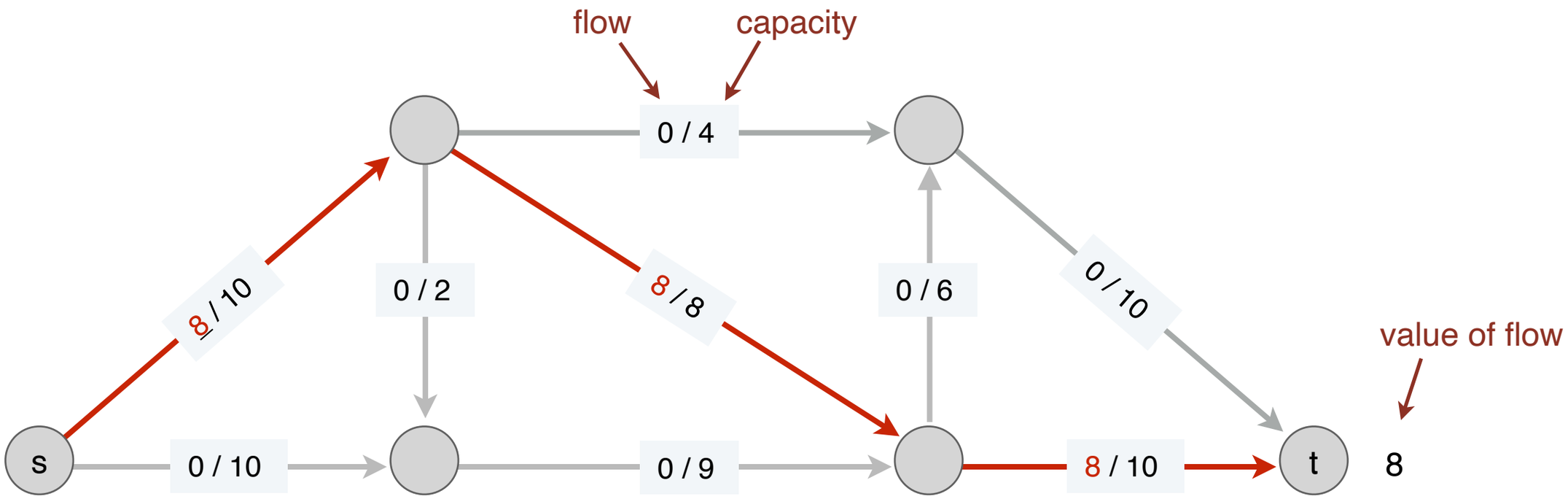


Augmenting paths

Observation: we can send additional flow along an st-path if there is free capacity along every edge.

Augmenting path: an st-path with free capacity, along which we augment the flow.

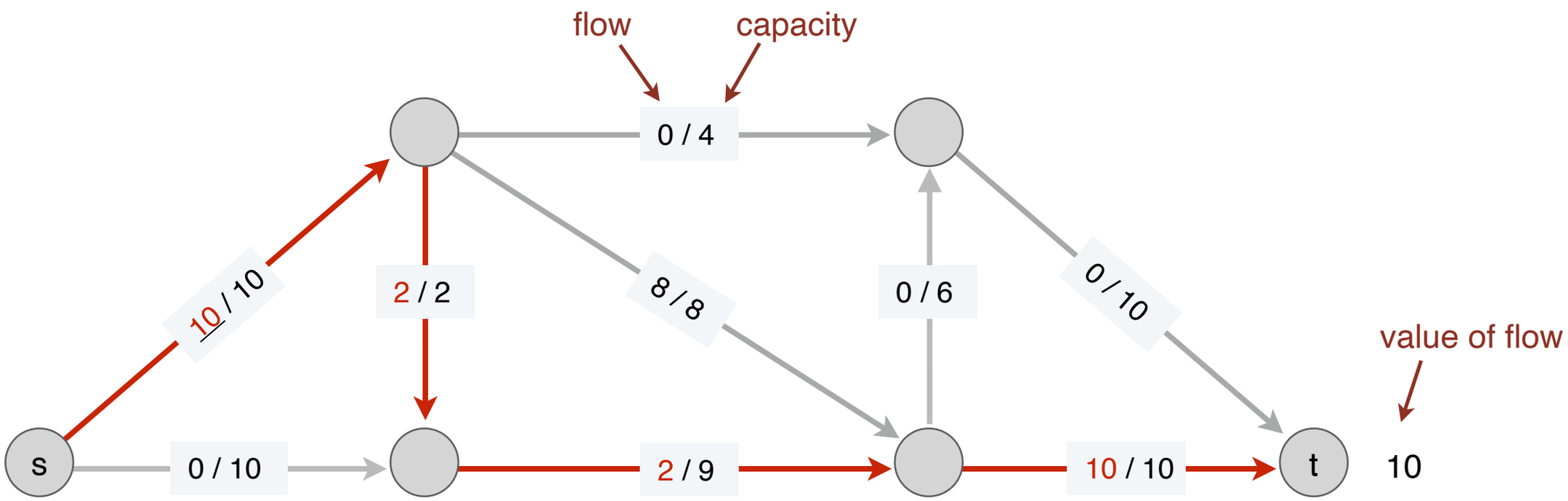
We can augment the flow by 8 units along the highlighted path.



Augmenting paths

Observation: we can send additional flow along an st-path if there is free capacity along every edge.

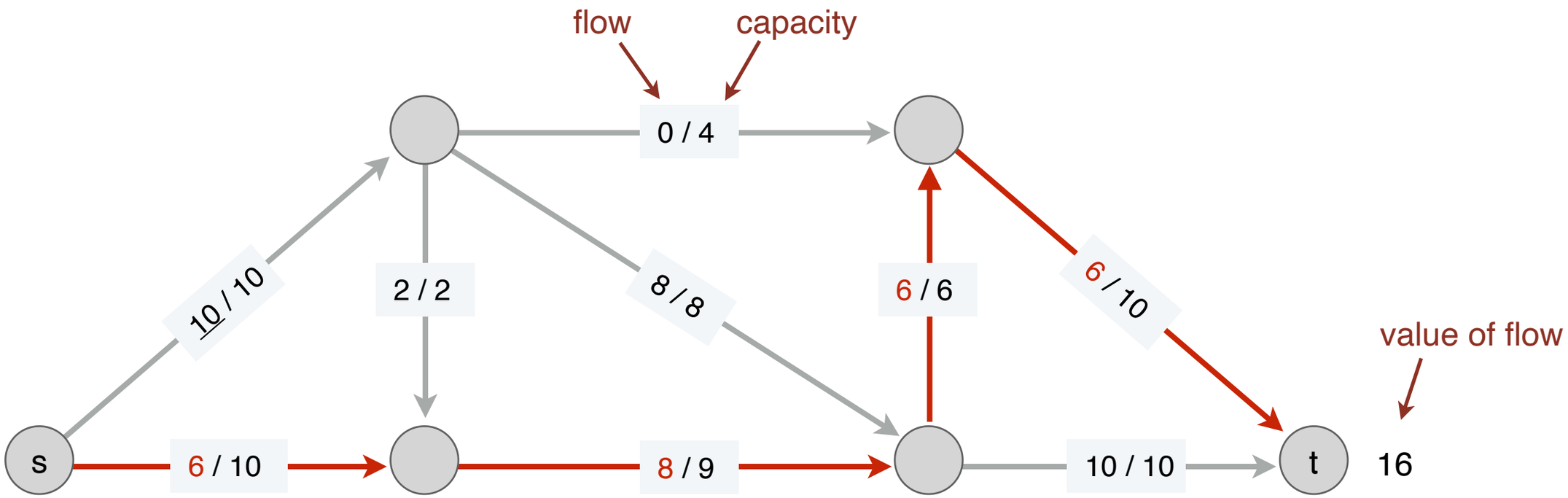
Augmenting path: an st-path with free capacity, along which we augment the flow.



Augmenting paths

Observation: we can send additional flow along an st-path if there is free capacity along every edge.

Augmenting path: an st-path with free capacity, along which we augment the flow.



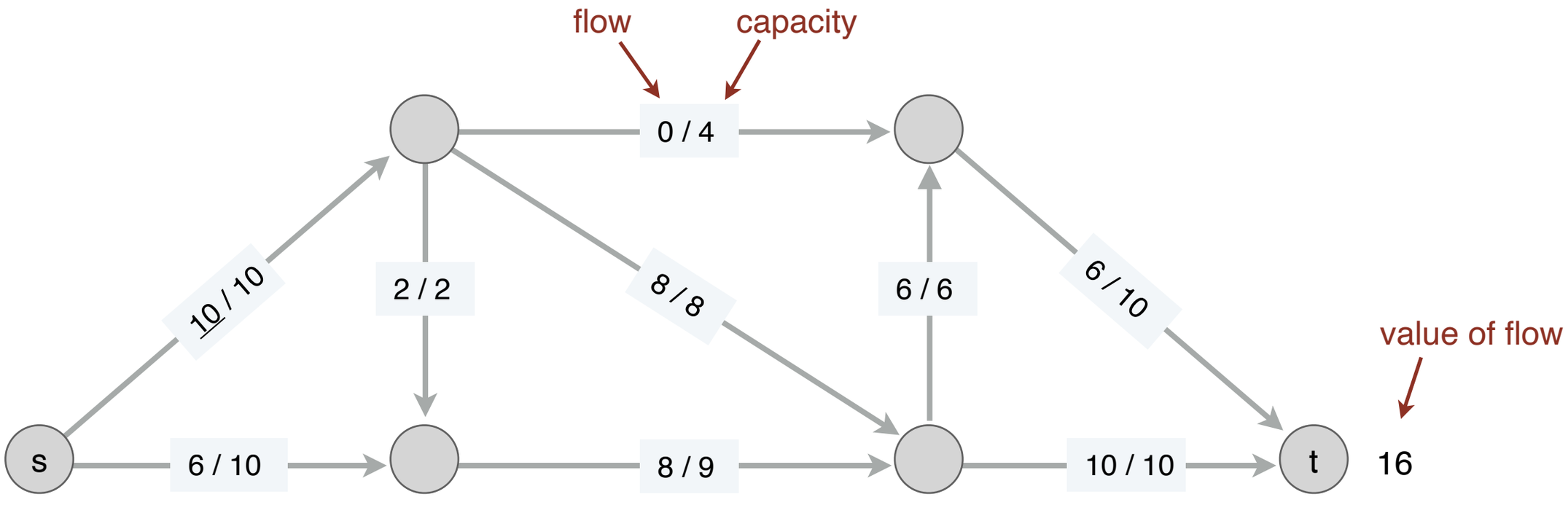
Augmenting paths

Observation: we can send additional flow along an st-path if there is free capacity along every edge.

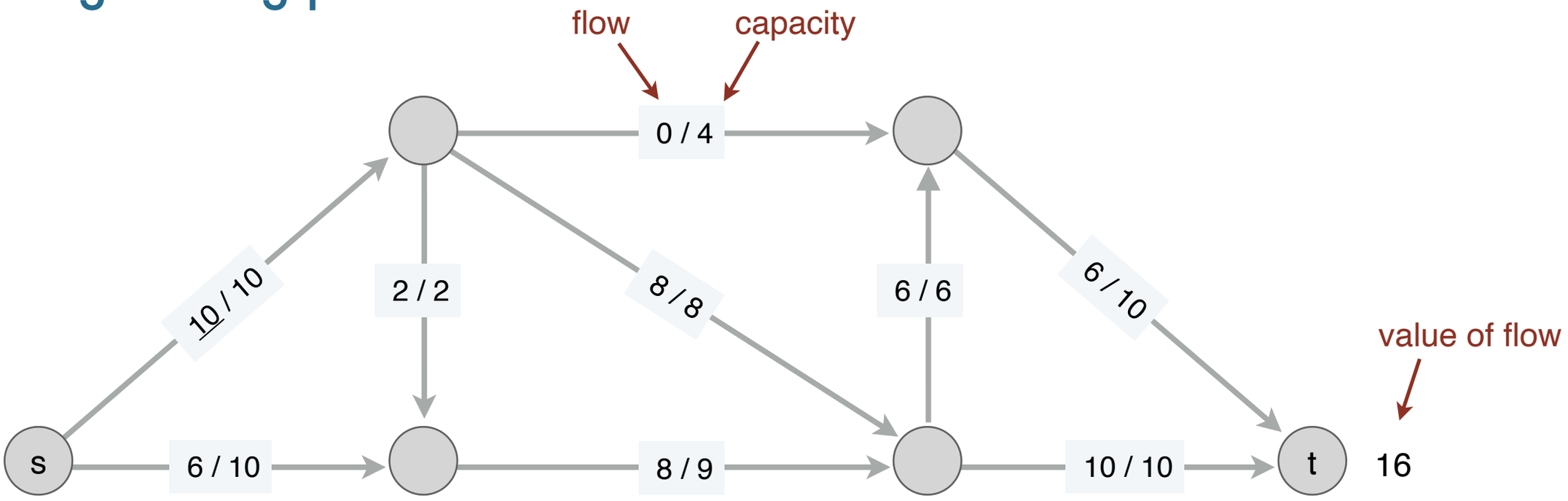
Augmenting path: an st-path with free capacity, along which we augment the flow.

The value of this flow is 16.

But this is not the max flow in this network.

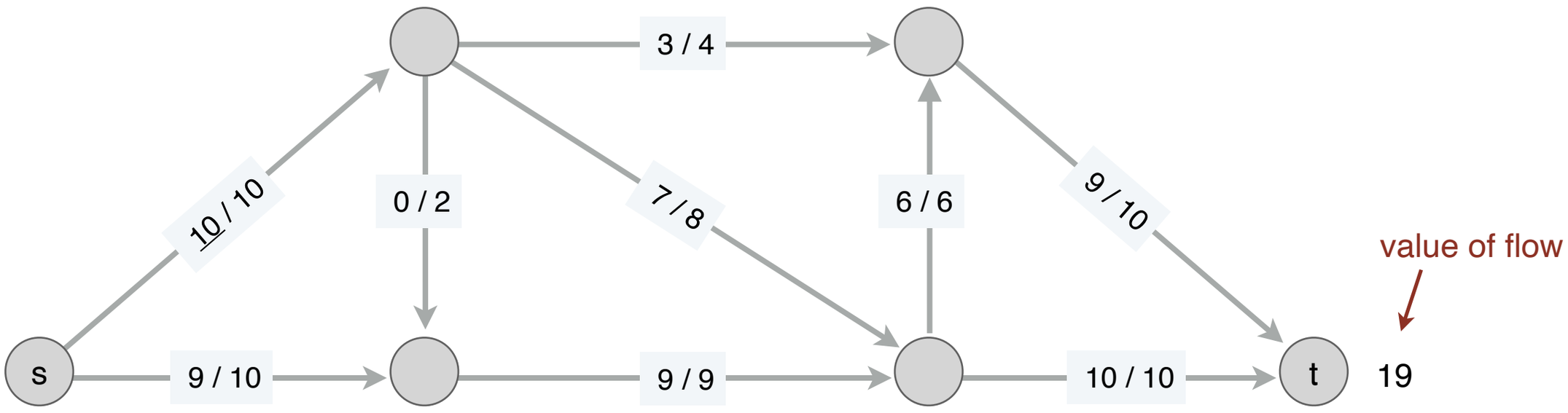


Augmenting paths and max flow



Value on top is 16

Below 19

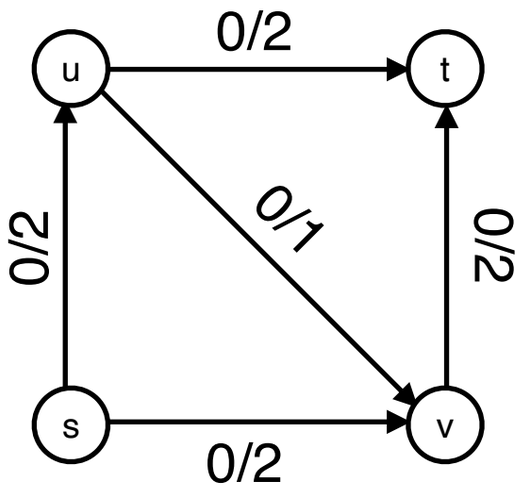


Choosing augmenting path greedily

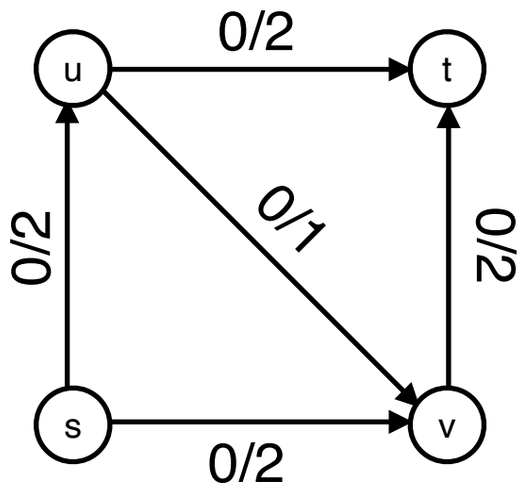
Greedy choice:

- at each step choose an augmenting path and increase the flow along it by as much as we can.
- greedy heuristic: never reverse on a decision, e.g. which path to send the flow

max flow solution:



suboptimal due to greedy choice:

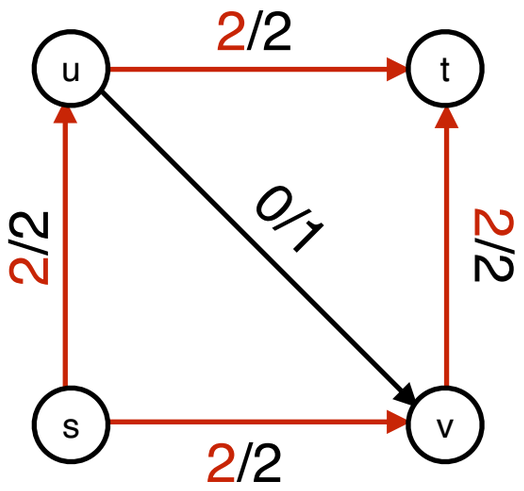


Choosing augmenting path greedily

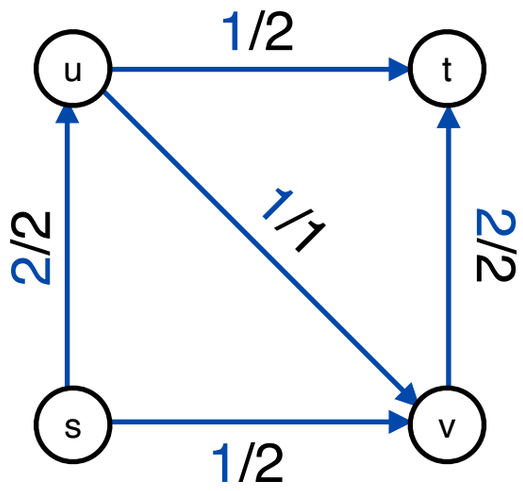
Greedy choice:

- at each step choose an augmenting path and increase the flow along it by as much as we can.
- greedy heuristic: never reverse on a decision, e.g. which path to send the flow

max flow solution:



suboptimal due to greedy choice:



Residual graph

G: nodes V, edges E, capacities c(e)

residual graph G_f:

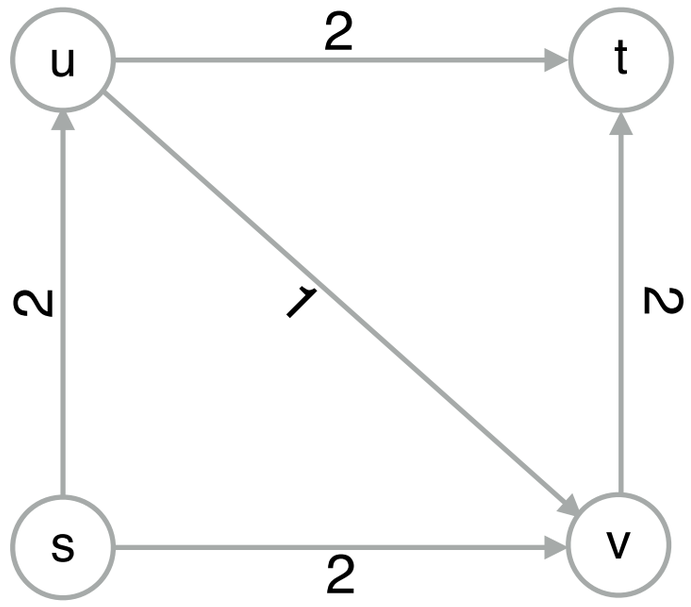
- nodes V
- edges $E \cup E^{reverse}$
- residual capacity

$$c_f(e) = \begin{cases} c(e) - f(e) & e \in E \\ f(e) & e \in E^{reverse} \end{cases}$$

The sum of the capacity on an edge and its reverse are equal to the original edge capacity, thus

$$c_f(e) + c_f(e^{reversed}) = c(e)$$

- Initially the flow is 0 on every edge, hence the reverse edge capacities are 0



Residual graph

G: nodes V, edges E, capacities c(e)

residual graph G_f:

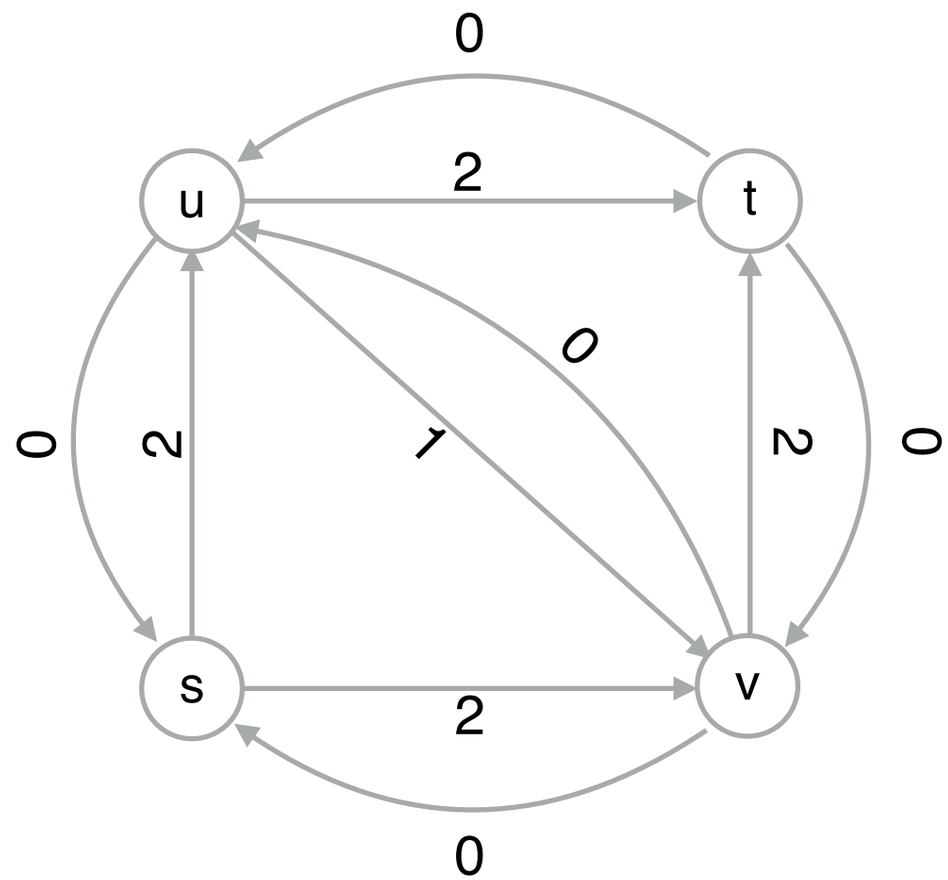
- nodes V
- edges $E \cup E^{reverse}$
- residual capacity

$$c_f(e) = \begin{cases} c(e) - f(e) & e \in E \\ f(e) & e \in E^{reverse} \end{cases}$$

The sum of the capacity on an edge and its reverse are equal to the original edge capacity, thus

$$c_f(e) + c_f(e^{reversed}) = c(e)$$

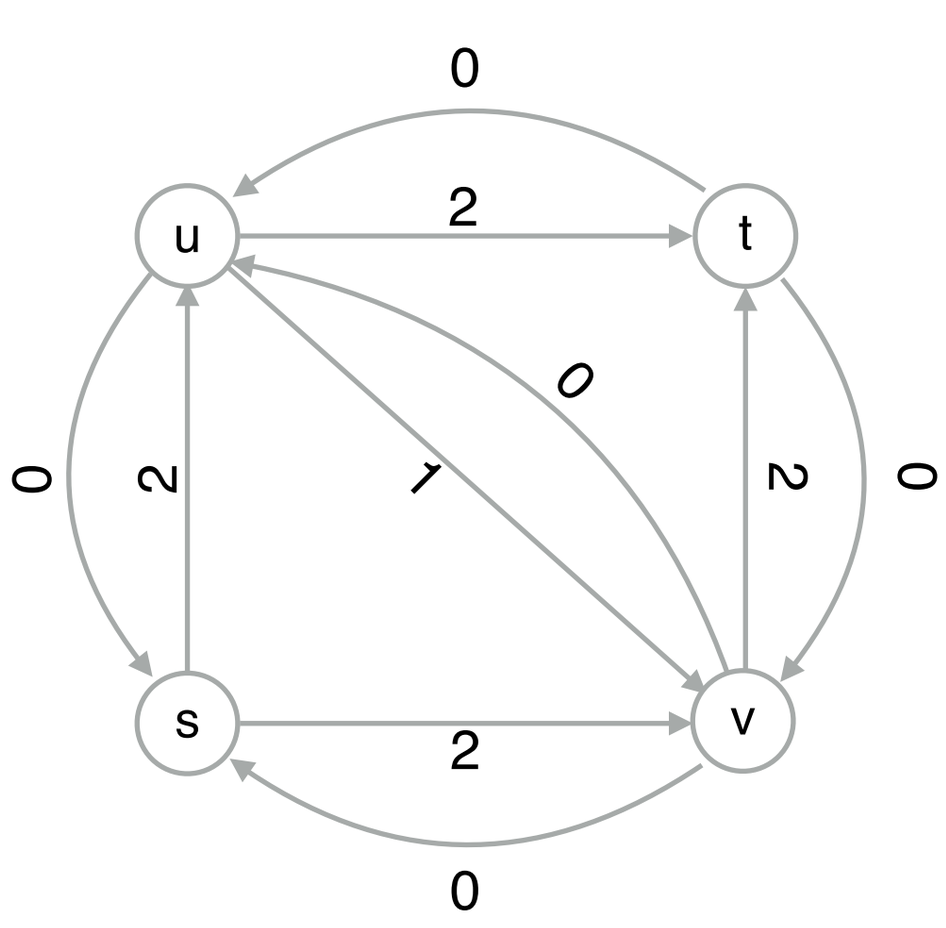
- Initially the flow is 0 on every edge, hence the reverse edge capacities are 0



Ford-Fulkerson algorithm

Algorithm:

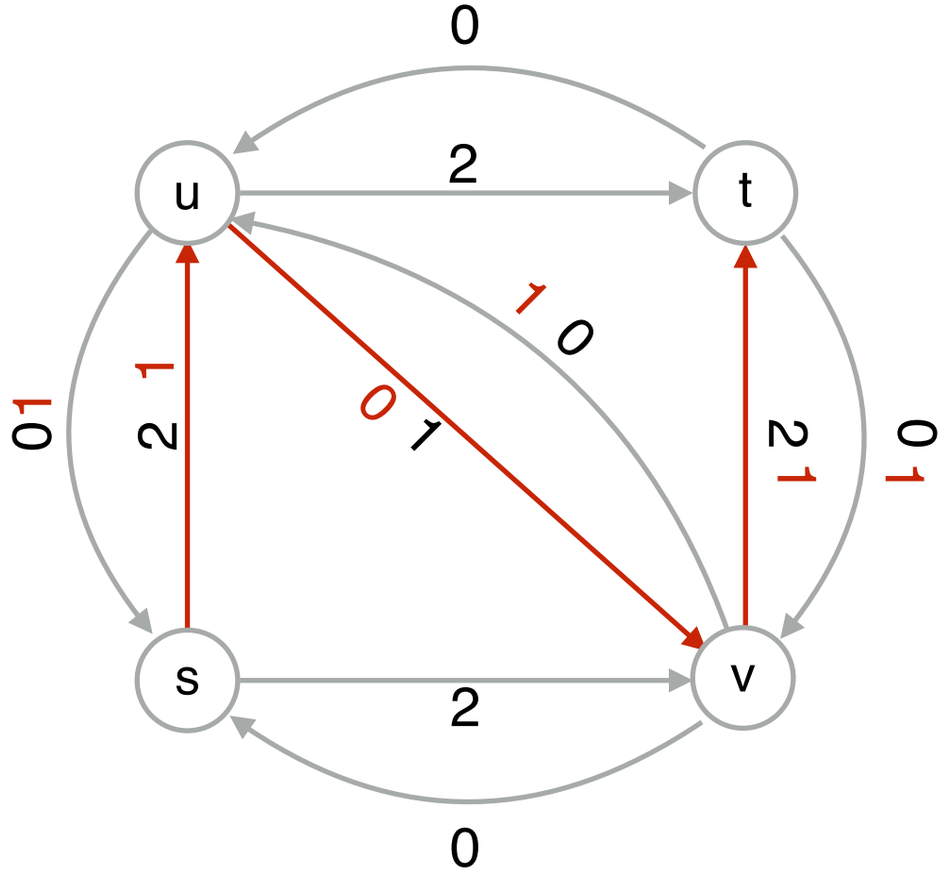
- In each iteration find an augmenting path in G_f and increase the flow along the path
  - Treat original and reversed edges the same in the paths
- For each residual edge capacity that is decrease along the path, increase the capacity of the opposite edge
  - maintain $c_f(e) + c_f(e^{reversed}) = c(e)$



Ford-Fulkerson algorithm

Algorithm:

- In each iteration find an augmenting path in G_f and increase the flow along the path
  - Treat original and reversed edges the same in the paths
- For each residual edge capacity that is decrease along the path, increase the capacity of the opposite edge
  - maintain $c_f(e) + c_f(e^{reversed}) = c(e)$



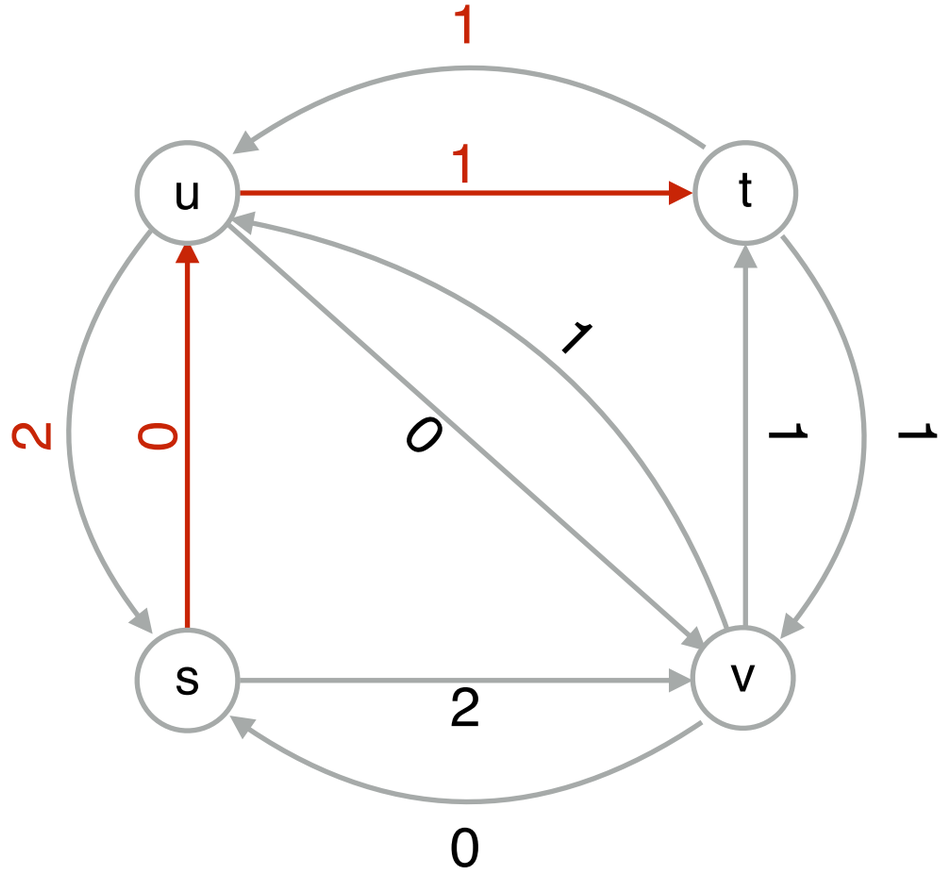
send 1 unit of flow along the path s->u->v->t

- updated capacities in red

Ford-Fulkerson algorithm

Algorithm:

- In each iteration find an augmenting path in G_f and increase the flow along the path
  - Treat original and reversed edges the same in the paths
- For each residual edge capacity that is decrease along the path, increase the capacity of the opposite edge
  - maintain $c_f(e) + c_f(e^{reversed}) = c(e)$

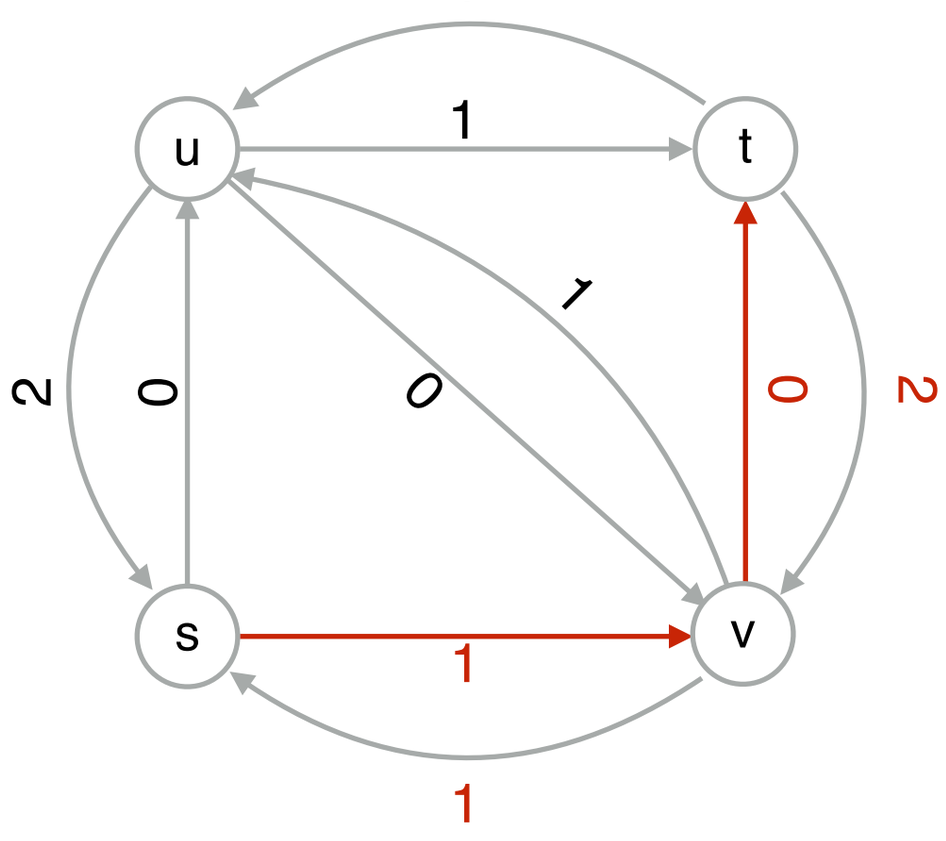


Ford-Fulkerson algorithm

Algorithm:

- In each iteration find an augmenting path in G_f and increase the flow along the path
  - Treat original and reversed edges the same in the paths
- For each residual edge capacity that is decrease along the path, increase the capacity of the opposite edge
  - maintain $c_f(e) + c_f(e^{reversed}) = c(e)$

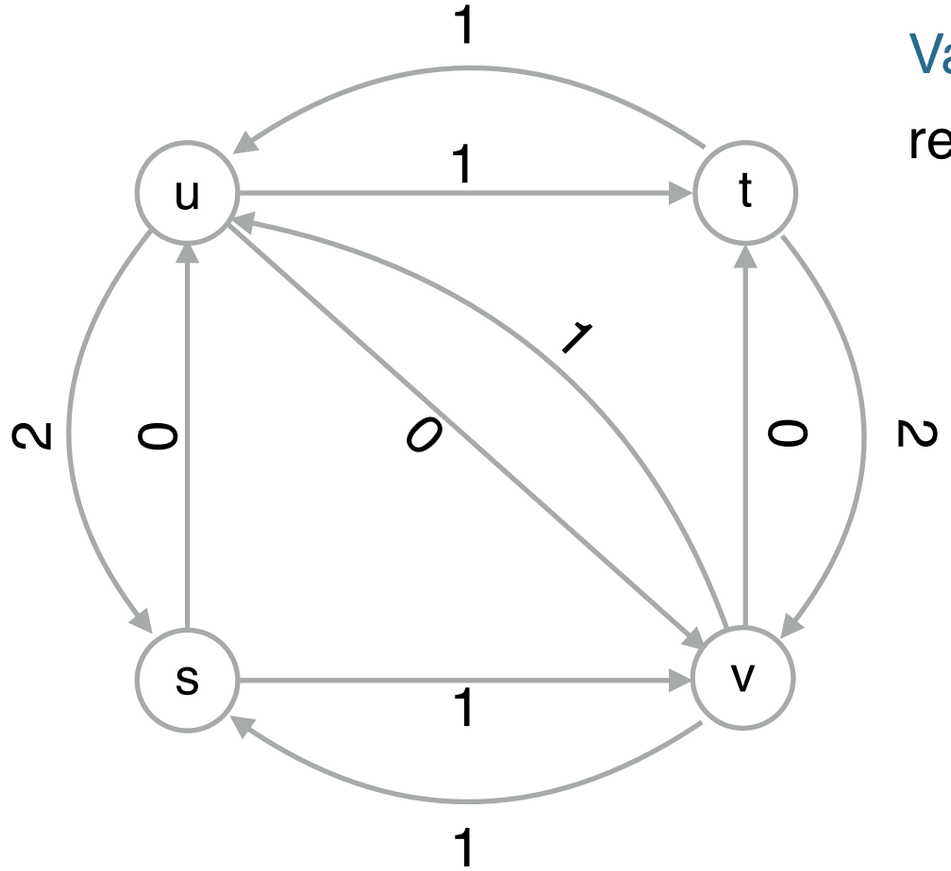
1



Ford-Fulkerson algorithm

Algorithm:

- In each iteration find an augmenting path in G_f and increase the flow along the path
  - Treat original and reversed edges the same in the paths
- For each residual edge capacity that is decrease along the path, increase the capacity of the opposite edge
  - maintain $c_f(e) + c_f(e^{reversed}) = c(e)$

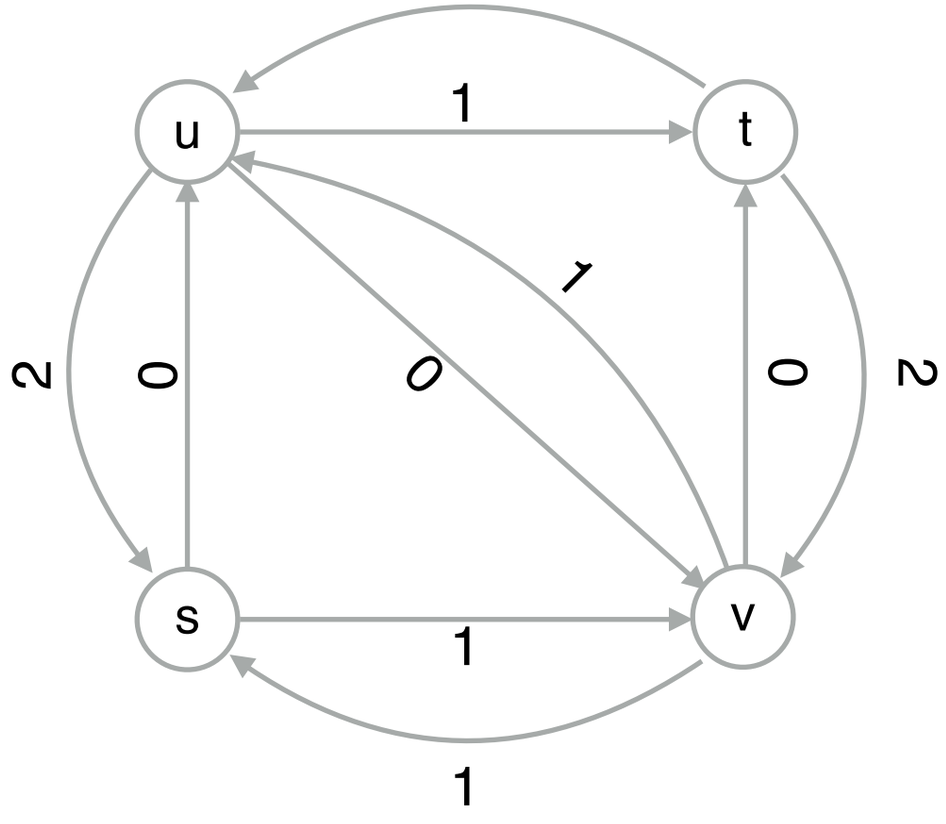


Value of the flow (in the original G) is the total residual capacity in G_f leaving t.

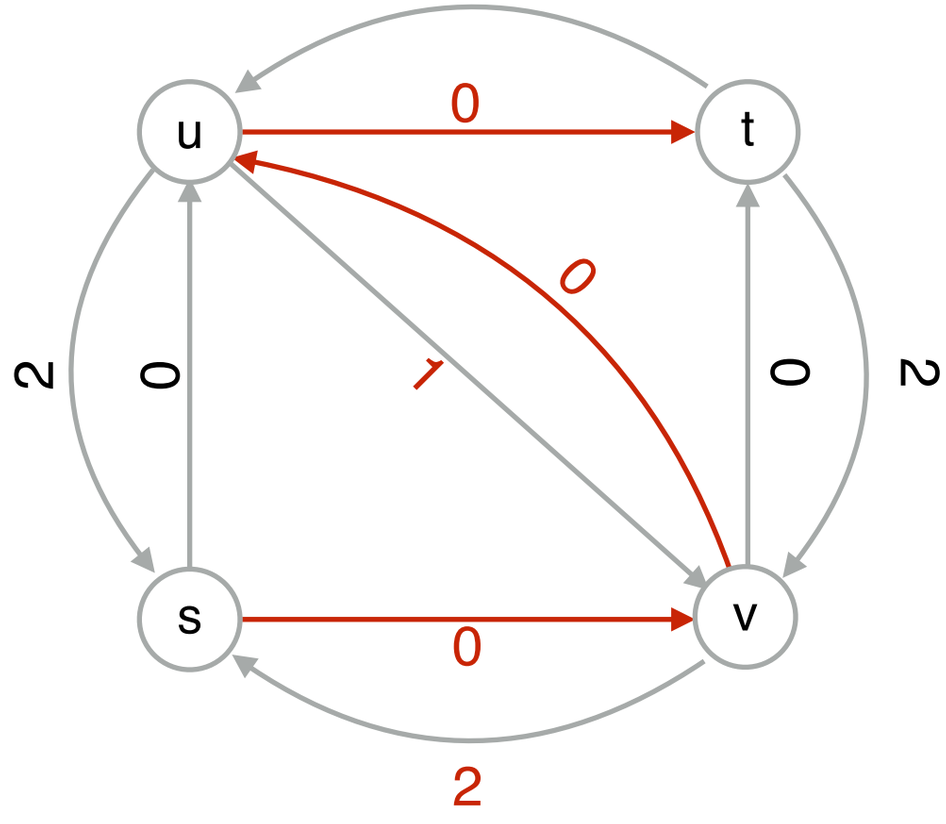
Ford-Fulkerson algorithm

- This is the situation where the greedy algorithm got stuck in a suboptimal solution
- However, there is an augmenting path using residual edges
- Using a residual edge is akin to “sending flow back” among an edge.

1

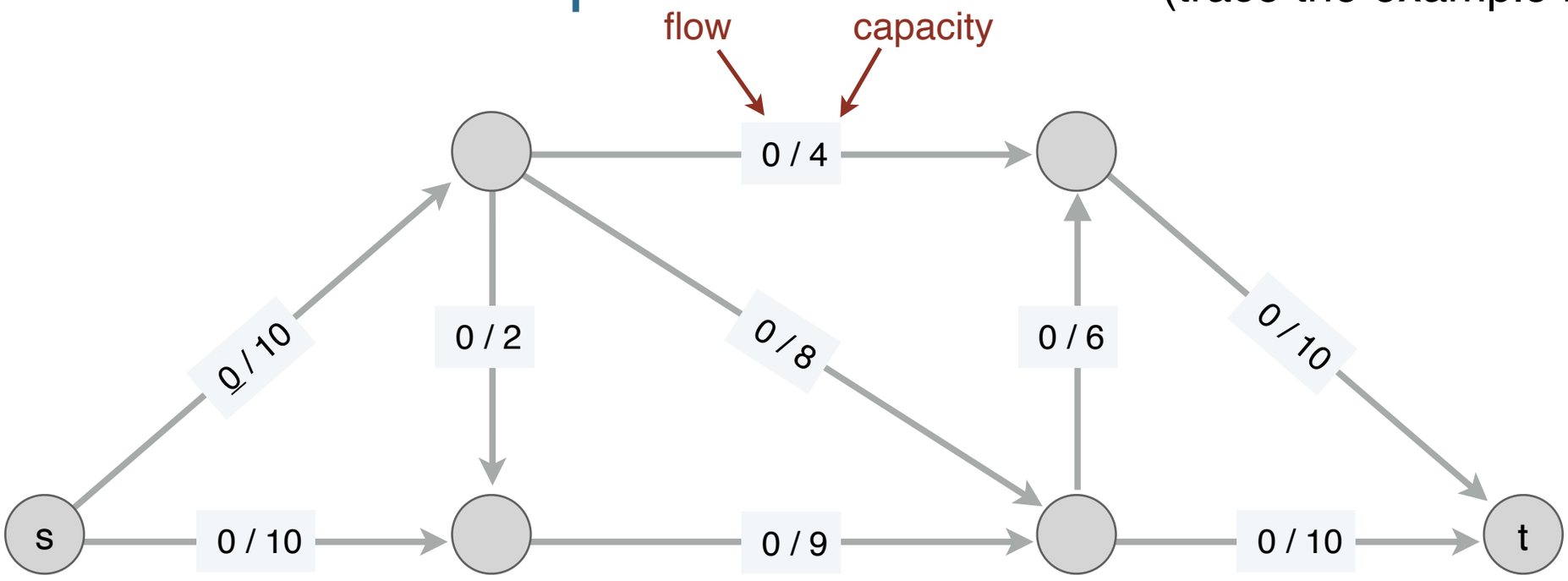


2

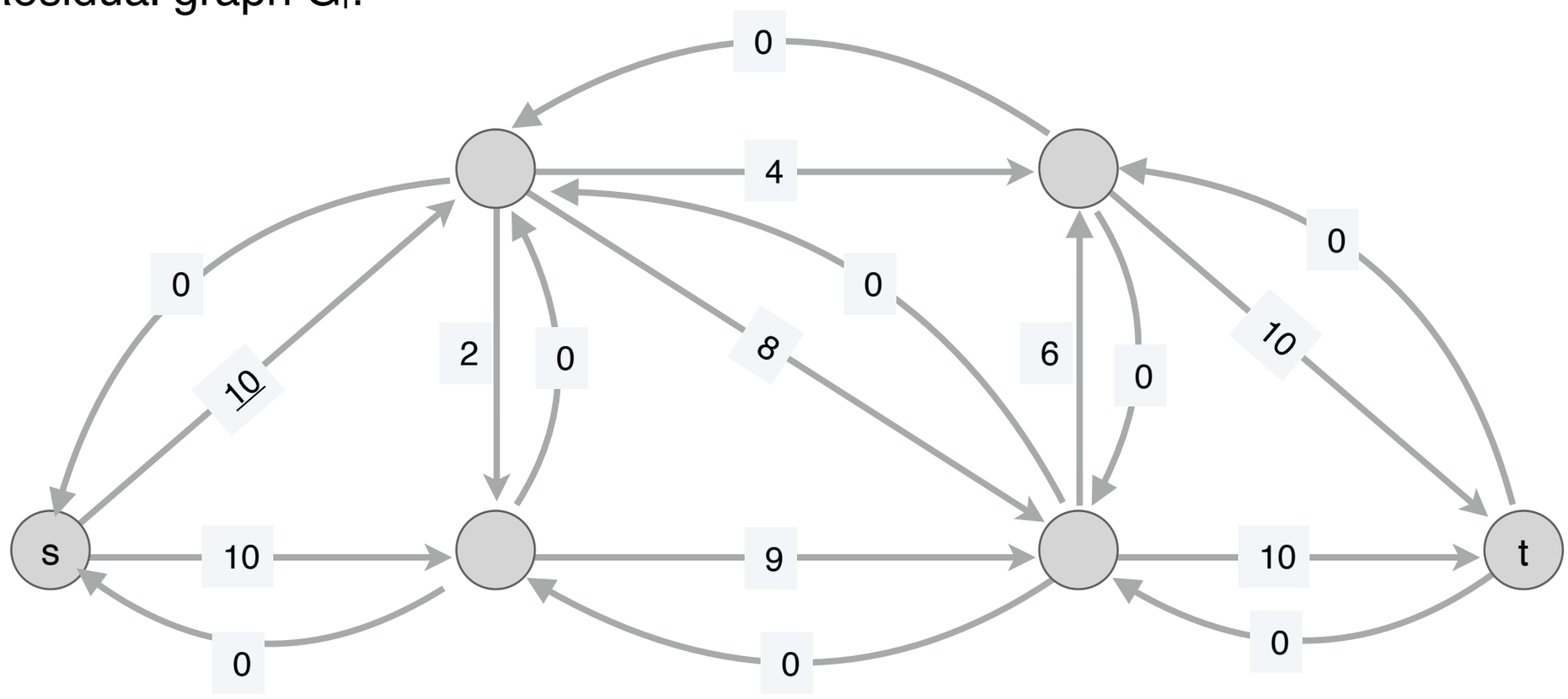


Ford-Fulkerson example

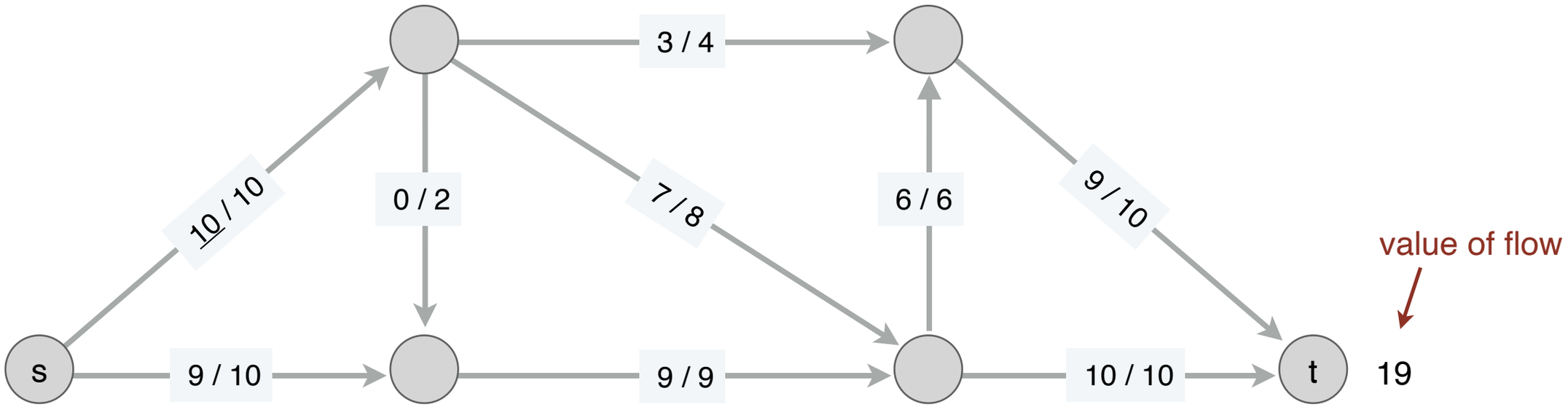
(trace the example in class)



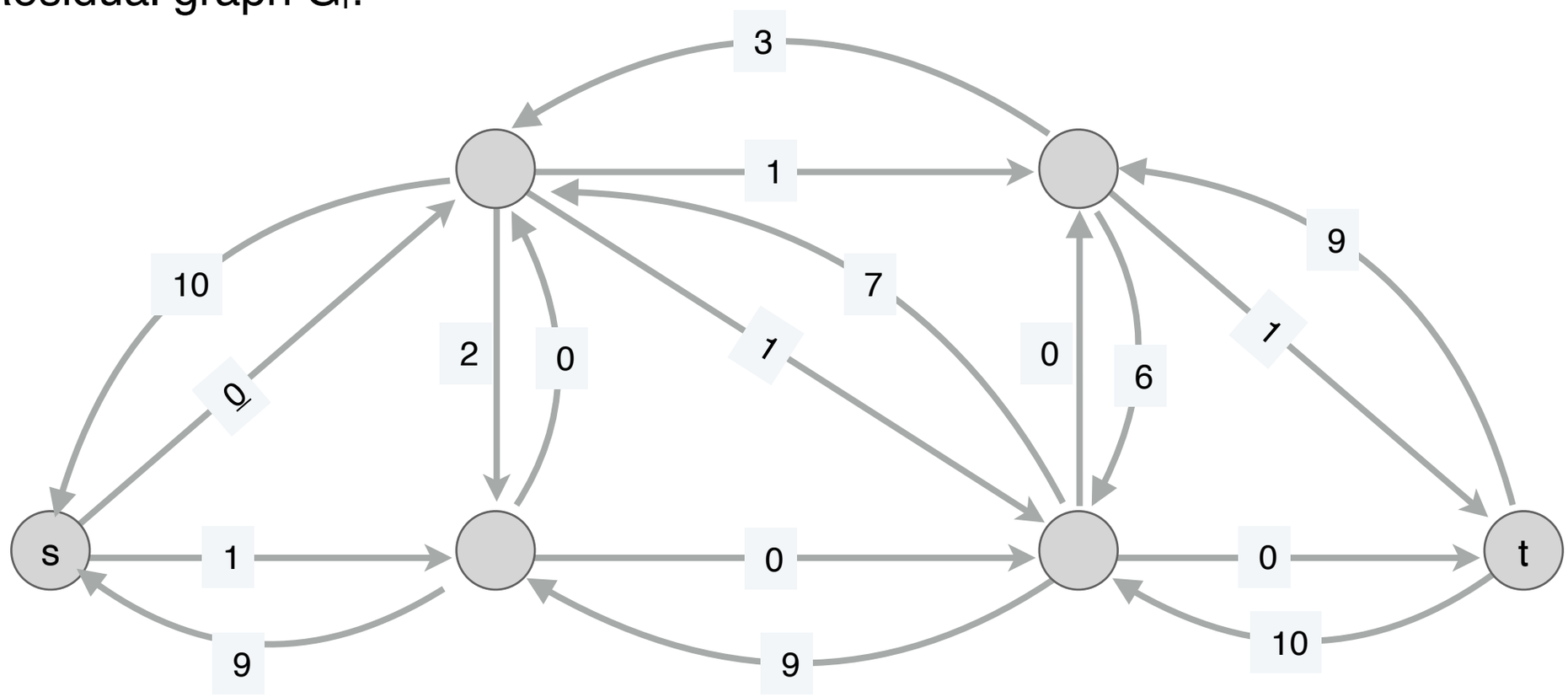
Residual graph G_f:



Ford-Fulkerson example



Residual graph G_f:



Ford - Fulkerson algorithm

Algorithm 1: FordFulkerson($G(V, E, c)$)

```
1 for e ∈ E do
2 |   f(e) = 0;
3 G_f ← residual graph with respect to f;
4 while There exist an s ⇝ s augmenting path P do
5 |   f ← Augment(f, c, P)/* augment the flow along the path with
  |      the bottleneck capacity along P                          */
6 |   update G_f;
7 return f
```

question: what do we need to consider to compute its running time?

Ford - Fulkerson algorithm – running time

Algorithm 1: FordFulkerson($G(V, E, c)$)

```
1 for e ∈ E do
2 |   f(e) = 0;
3 G_f ← residual graph with respect to f;
4 while There exist an s ⇝ s augmenting path P do
5 |   f ← Augment(f, c, P)/* augment the flow along the path with
  |      the bottleneck capacity along P                          */
6 |   update G_f;
7 return f
```

- What is the size of G_f?

- How do we find an augmenting path and what is the runtime?

- How long does it take to augment a path? (i.e. run *Augment(f,c,P)* in line 5)

Ford - Fulkerson algorithm – running time

Algorithm 1: FordFulkerson($G(V, E, c)$)

```
1 for e ∈ E do
2 |   f(e) = 0;
3 G_f ← residual graph with respect to f;
4 while There exist an s ⇝ s augmenting path P do
5 |   f ← Augment(f, c, P)/* augment the flow along the path with
  |      the bottleneck capacity along P                          */
6 |   update G_f;
7 return f
```

- What is the size of G_f?

  O(E)

- How do we find an augmenting path and what is the runtime?

- How long does it take to augment a path? (i.e. run *Augment(f,c,P)* in line 5)

Ford - Fulkerson algorithm – running time

Algorithm 1: FordFulkerson($G(V, E, c)$)

```
1 for e ∈ E do
2 |   f(e) = 0;
3 G_f ← residual graph with respect to f;
4 while There exist an s ⇝ s augmenting path P do
5 |   f ← Augment(f, c, P)/* augment the flow along the path with
  |      the bottleneck capacity along P                          */
6 |   update G_f;
7 return f
```

- What is the size of G_f?

  O(E)

- How do we find an augmenting path and what is the runtime?

  BFS or DFS along residual graph edges with positive capacity. O(E)

- How long does it take to augment a path? (i.e. run *Augment(f,c,P)* in line 5)

Ford - Fulkerson algorithm – running time

Algorithm 1: FordFulkerson($G(V, E, c)$)

```
1 for e ∈ E do
2 |   f(e) = 0;
3 G_f ← residual graph with respect to f;
4 while There exist an s ⇝ s augmenting path P do
5 |   f ← Augment(f, c, P)/* augment the flow along the path with
  |      the bottleneck capacity along P                          */
6 |   update G_f;
7 return f
```

- What is the size of G_f?

  O(E)

- How do we find an augmenting path and what is the runtime?

  BFS or DFS along residual graph edges with positive capacity. O(E)

- How long does it take to augment a path? (i.e. run *Augment(f,c,P)* in line 5)

  O(V)

Ford - Fulkerson algorithm - running time

Algorithm 1: FordFulkerson($G(V, E, c)$)

```
1 for e ∈ E do
2 |   f(e) = 0;
3 G_f ← residual graph with respect to f;
4 while There exist an s ⇝ s augmenting path P do
5 |   f ← Augment(f, c, P)/* augment the flow along the path with
  |      the bottleneck capacity along P                          */
6 |   update G_f;
7 return f
```

claim: if all capacities are integers, then FF terminates.

proof:

Ford - Fulkerson algorithm - running time

Algorithm 1: FordFulkerson($G(V, E, c)$)

```
1 for e ∈ E do
2 |   f(e) = 0;
3 G_f ← residual graph with respect to f;
4 while There exist an s ⇝ s augmenting path P do
5 |   f ← Augment(f, c, P)/* augment the flow along the path with
  |      the bottleneck capacity along P                          */
6 |   update G_f;
7 return f
```

claim: if all capacities are integers, then FF terminates.

proof (sketch):

- if all capacities are integers, then the value of the flow is strictly increased — and by an integer amount — in each iteration
- the number of iterations is at most

$$C = \sum_{edge\ (s,u)} c(s, u)$$

Ford - Fulkerson algorithm - running time

Algorithm 1: FordFulkerson($G(V, E, c)$)

```
1 for e ∈ E do
2 |   f(e) = 0;
3 G_f ← residual graph with respect to f;
4 while There exist an s ⇝ s augmenting path P do
5 |   f ← Augment(f, c, P)/* augment the flow along the path with
  |      the bottleneck capacity along P                          */
6 |   update G_f;
7 return f
```

claim: if all capacities are integers, then FF terminates.

proof (sketch):

- if all capacities are integers, then the value of the flow is strictly increased — and by an integer amount — in each iteration
- the number of iterations is at most

$$C = \sum_{edge\ (s,u)} c(s, u)$$

- the number of iterations is also bounded by val(f), but we don't know that yet

Ford - Fulkerson algorithm - running time

Algorithm 1: FordFulkerson($G(V, E, c)$)

```
1 for e ∈ E do
2 |   f(e) = 0;
3 G_f ← residual graph with respect to f;
4 while There exist an s ⇝ s augmenting path P do
5 |   f ← Augment(f, c, P)/* augment the flow along the path with
  |      the bottleneck capacity along P                          */
6 |   update G_f;
7 return f
```

Total running time:

Is this an efficient algorithm?

Ford - Fulkerson algorithm - running time

Algorithm 1: FordFulkerson($G(V, E, c)$)

```
1 for e ∈ E do
2 |   f(e) = 0;
3 G_f ← residual graph with respect to f;
4 while There exist an s ⇝ s augmenting path P do
5 |   f ← Augment(f, c, P)/* augment the flow along the path with
  |      the bottleneck capacity along P                          */
6 |   update G_f;
7 return f
```

Total running time:

1-2. *O(m)* initialization

Is this an efficient algorithm?

Ford - Fulkerson algorithm - running time

Algorithm 1: FordFulkerson($G(V, E, c)$)

```
1 for e ∈ E do
2 |   f(e) = 0;
3 G_f ← residual graph with respect to f;
4 while There exist an s ⇝ s augmenting path P do
5 |   f ← Augment(f, c, P)/* augment the flow along the path with
  |      the bottleneck capacity along P                          */
6 |   update G_f;
7 return f
```

Total running time:

1-2. *O(m)* initialization

3. *O(m)* construction of G_f

Is this an efficient algorithm?

Ford - Fulkerson algorithm - running time

Algorithm 1: FordFulkerson($G(V, E, c)$)

```
1 for e ∈ E do
2 |   f(e) = 0;
3 G_f ← residual graph with respect to f;
4 while There exist an s ⇝ s augmenting path P do
5 |   f ← Augment(f, c, P)/* augment the flow along the path with
  |      the bottleneck capacity along P                          */
6 |   update G_f;
7 return f
```

Total running time:

1-2. *O(m)* initialization

3. *O(m)* construction of G_f

4. Up to *C* repetitions, each check costs *O(m)*

Is this an efficient algorithm?

Ford - Fulkerson algorithm - running time

Algorithm 1: FordFulkerson($G(V, E, c)$)

```
1 for e ∈ E do
2 |   f(e) = 0;
3 G_f ← residual graph with respect to f;
4 while There exist an s ⇝ s augmenting path P do
5 |   f ← Augment(f, c, P)/* augment the flow along the path with
  |      the bottleneck capacity along P                          */
6 |   update G_f;
7 return f
```

Total running time:

1-2. *O(m)* initialization

3. *O(m)* construction of G_f

4. Up to *C* repetitions, each check costs *O(m)*

5. Up to *C* repetitions, augmenting costs *O(n)*

Is this an efficient algorithm?

Ford - Fulkerson algorithm - running time

Algorithm 1: FordFulkerson($G(V, E, c)$)

```
1 for e ∈ E do
2 |   f(e) = 0;
3 G_f ← residual graph with respect to f;
4 while There exist an s ⇝ s augmenting path P do
5 |   f ← Augment(f, c, P)/* augment the flow along the path with
  |      the bottleneck capacity along P                          */
6 |   update G_f;
7 return f
```

Total running time:

1-2. *O(m)* initialization

3. *O(m)* construction of G_f

4. Up to *C* repetitions, each check costs *O(m)*

5. Up to *C* repetitions, augmenting costs *O(n)*

6. Up to *C* repetitions, update costs *O(m)*

Is this an efficient algorithm?

Ford - Fulkerson algorithm - running time

Algorithm 1: FordFulkerson($G(V, E, c)$)

```
1 for e ∈ E do
2 |   f(e) = 0;
3 G_f ← residual graph with respect to f;
4 while There exist an s ⇝ s augmenting path P do
5 |   f ← Augment(f, c, P)/* augment the flow along the path with
  |      the bottleneck capacity along P                          */
6 |   update G_f;
7 return f
```

Total running time: *O(Cm)*

1-2. *O(m)* initialization

3. *O(m)* construction of G_f

4. Up to *C* repetitions, each check costs *O(m)*

5. Up to *C* repetitions, augmenting costs *O(n)*

6. Up to *C* repetitions, update costs *O(m)*

Is this an efficient algorithm?

FF speedup — choosing the best augmenting paths

Goal: limit the number of iterations in which we augment the flow

method: instead of using *any* augmenting paths, make a clever selection

Edmonds-Karp algorithm — augmenting paths with fewest edges

Idea: in each iteration select the path with the fewest number of edges

How?

Edmonds-Karp — running time

Lemma 1: Throughout the algorithm, the length of a shortest augmenting path never decreases.

Lemma 2: After at most m shortest path augmentations the number of edges in the shortest augmenting path strictly increases.

Theorem: The shortest augmenting paths algorithm runs in time $O(nm^2)$

A concise proof of the lemmas can be found in CLRS, ch. 26, pages 727-730

Edmonds-Karp — running time

Lemma 1: Throughout the algorithm, the length of a shortest augmenting path never decreases.

Lemma 2: After at most m shortest path augmentations the number of edges in the shortest augmenting path strictly increases.

Theorem: The shortest augmenting paths algorithm runs in time $O(nm^2)$

(Augmenting path length from 1 to n)

A concise proof of the lemmas can be found in CLRS, ch. 26, pages 727-730

Edmonds-Karp — running time

Lemma 1: Throughout the algorithm, the length of a shortest augmenting path never decreases.

Lemma 2: After at most m shortest path augmentations the number of edges in the shortest augmenting path strictly increases.

Theorem: The shortest augmenting paths algorithm runs in time $O(nm^2)$

(Augmenting path length from 1 to n) *

(m augmentations to increase augmenting path length)

A concise proof of the lemmas can be found in CLRS, ch. 26, pages 727-730

Edmonds-Karp — running time

Lemma 1: Throughout the algorithm, the length of a shortest augmenting path never decreases.

Lemma 2: After at most m shortest path augmentations the number of edges in the shortest augmenting path strictly increases.

Theorem: The shortest augmenting paths algorithm runs in time $O(nm^2)$

(Augmenting path length from 1 to n) *

(m augmentations to increase augmenting path length) *

O(m) per augmentation

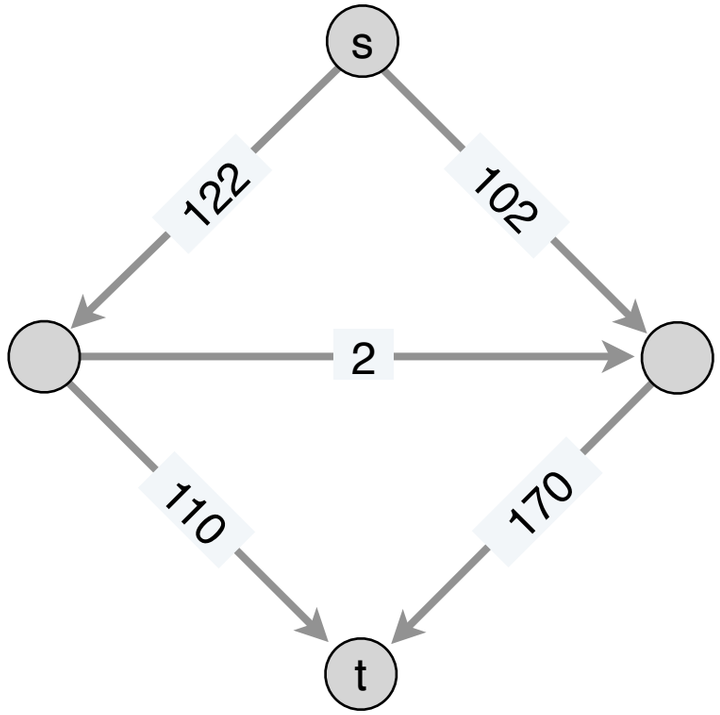
A concise proof of the lemmas can be found in CLRS, ch. 26, pages 727-730

FF speedup – choosing the best augmenting path II.

Goal. Make as few iterations in Ford-Fulkerson as possible

- depending on which augmenting paths we choose we may reach the max flow sooner

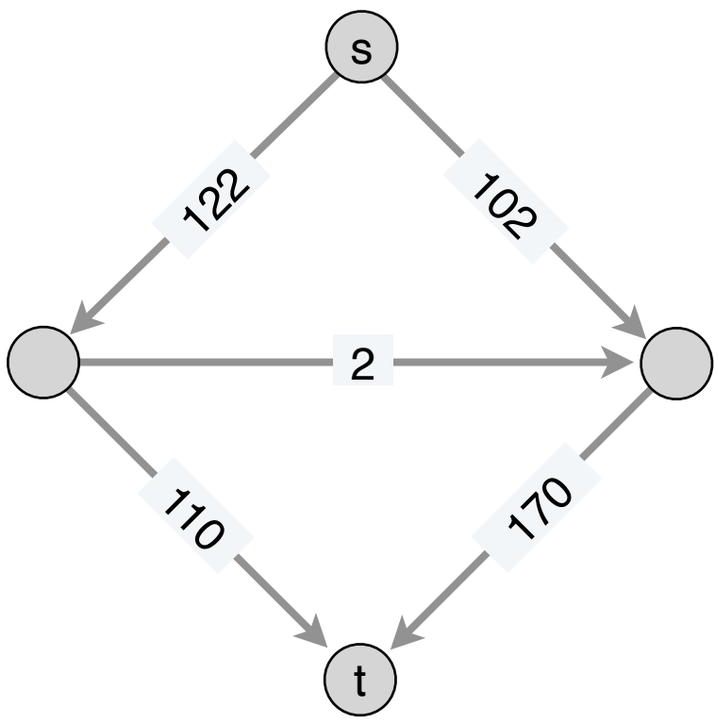
Intuition. In each iteration choose an augmenting path that increases the flow by the highest possible amount.



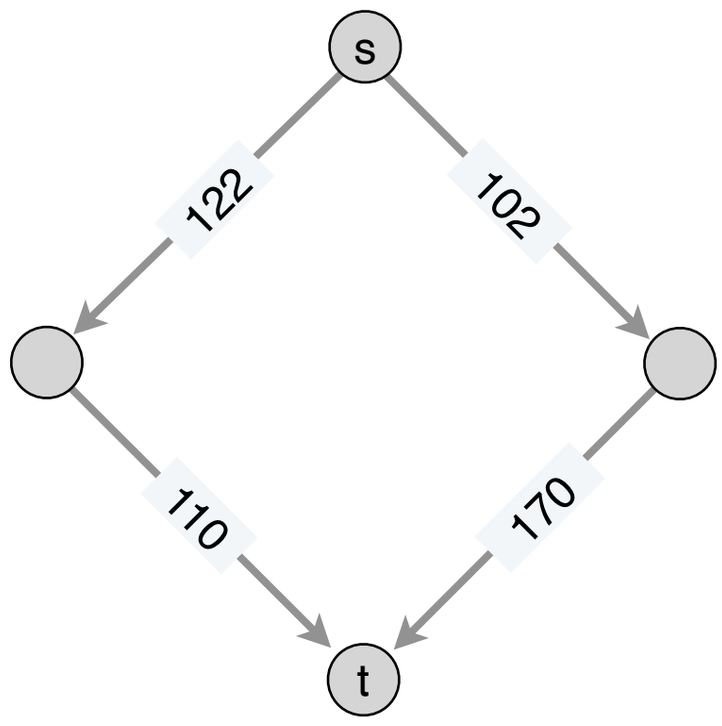
Capacity-scaling algorithm

Intuition.

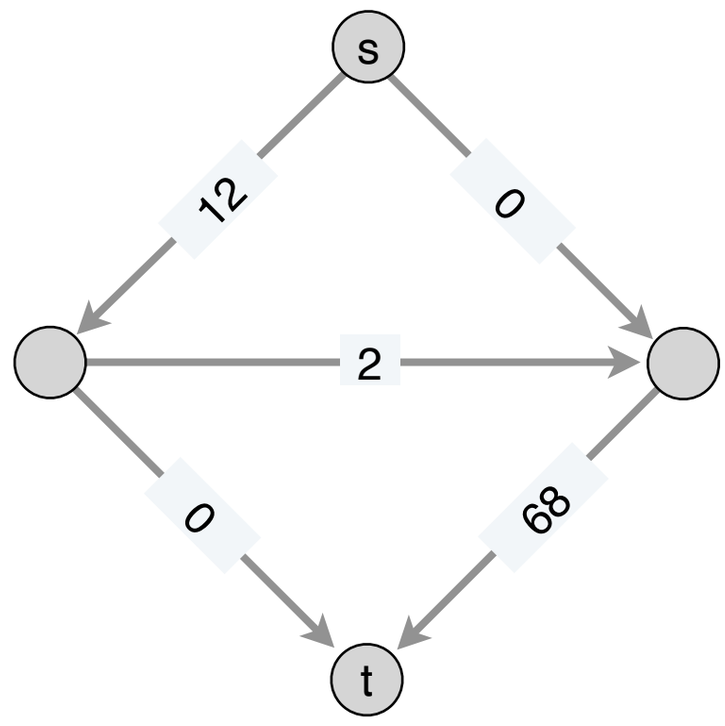
- Ignore all the edges in Gf with capacity lower than some value Δ
- choose an augmenting path among the high-capacity edges
- when all paths are exhausted, add back the edges of lower capacity



G_f



$G_f(\Delta)$, $\Delta = 100$



$\Delta = 1$

Capacity-scaling algorithm

Algorithm 1: capacity-scaling($G(V, E, s, t, c)$)

```
1  for each edge e do
2  |   f(e) ← 0;
3  Δ ← ⌊log(C)⌋ /* largest power of 2 that is ≤ C                    */
4  create G_f Δ /* Δ-residual network                                 */
5  while Δ ≥ 1 do
6  |   while there is an augmenting path P in G_f(Δ) do
7  |   |   f ←Augment(f, c, P);
8  |   |   Update G_f(Δ);
9  |   Δ ← Δ/2;
10 return f
```

Running time: key idea — there are at most 2m augmentations for each scaling phase (m refers to the scaled graph). There are at most O(log C) phases — $O(m^2 \log C)$

Capacity-scaling algorithm: proof of correctness

Assumption. All edge capacities are integers between 1 and C.

Integrality invariant. All flow and residual capacity values are integral.

Theorem. If capacity-scaling algorithm terminates, then f is a max flow.

Pf.

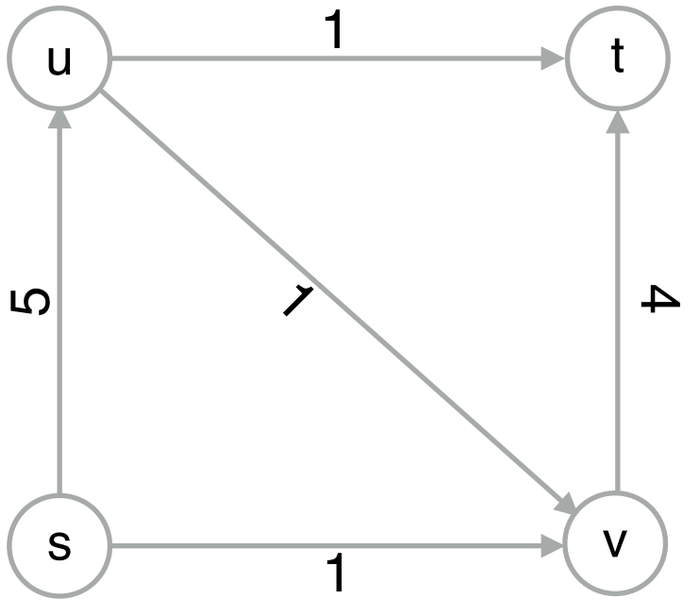
- By integrality invariant, when $\Delta = 1 \Rightarrow G_f(\Delta) = G_f$.
- Upon termination of $\Delta = 1$ phase, there are no augmenting paths. ■

Correctness of Ford-Fulkerson

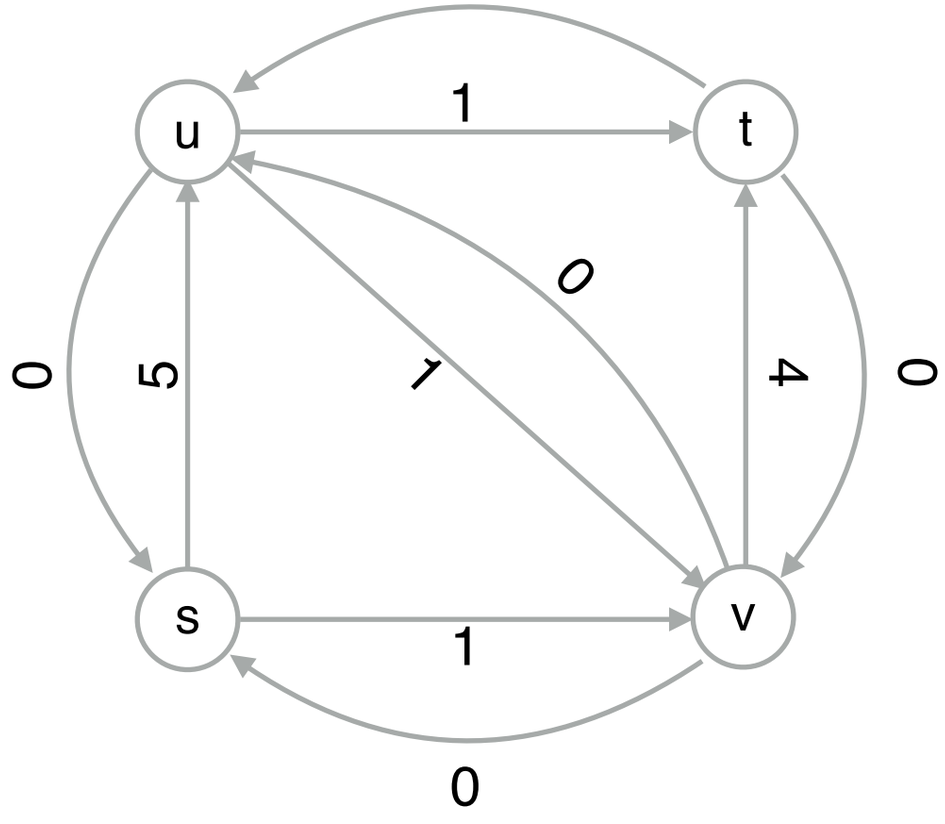
Max Flow Min Cut theorem

Bottleneck cuts

- What is the maximum flow in the graph on the left?
- Find the max flow by running Ford-Fulkerson on the residual graph.
- Can you see some “proof” in either graph why the value of the max flow cannot be more?



0



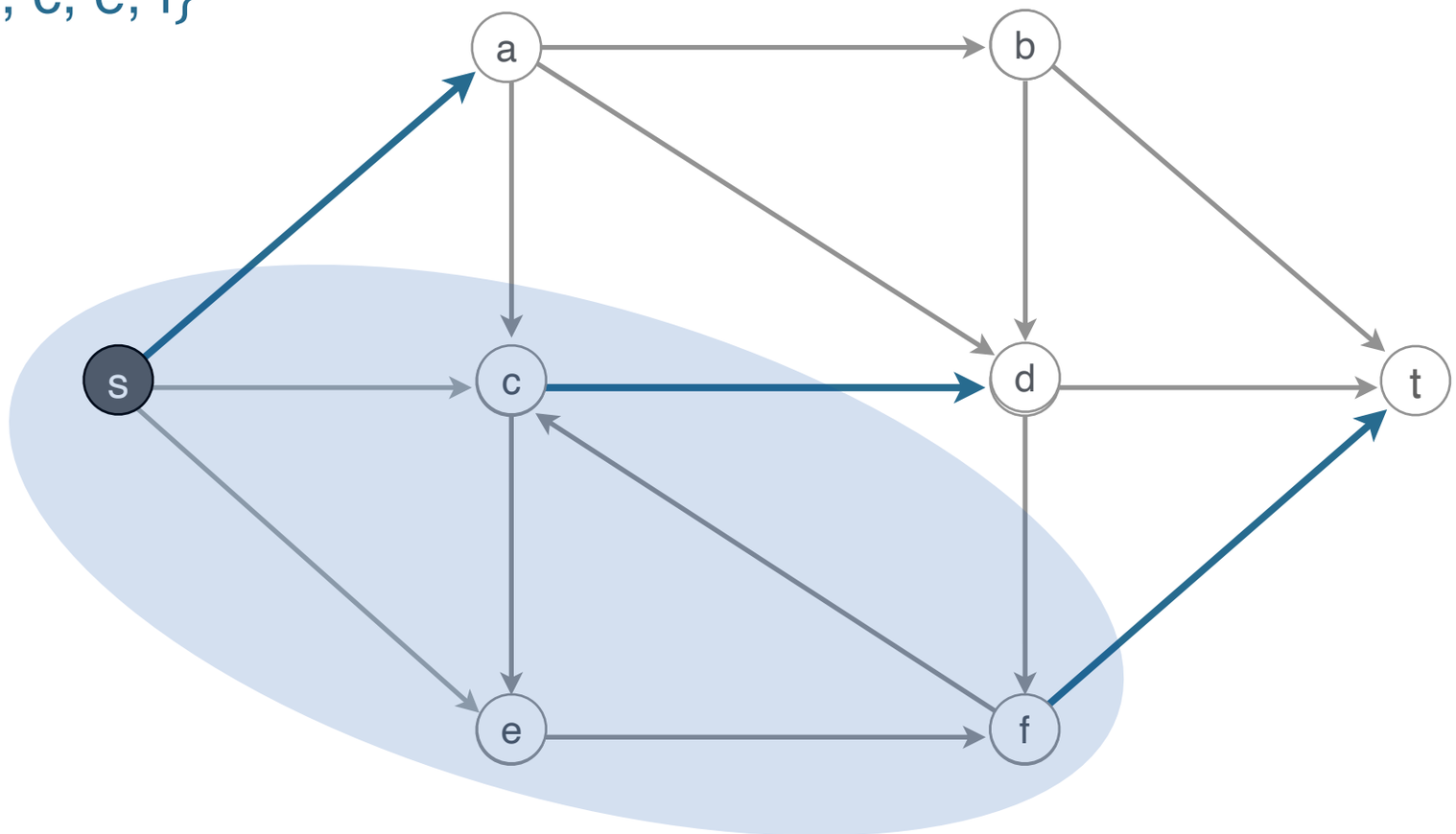
Minimum-cut problem

Def. An st-cut (cut) is a partition (A, B) of the vertices with $s \in A$ and $t \in B$.

cut-set: directed edges from nodes in A to B {(s,a), (c,d), (f,t)}

- Note, that this only contains edges *directed from A to B*

cut: A = {s, c, e, f}

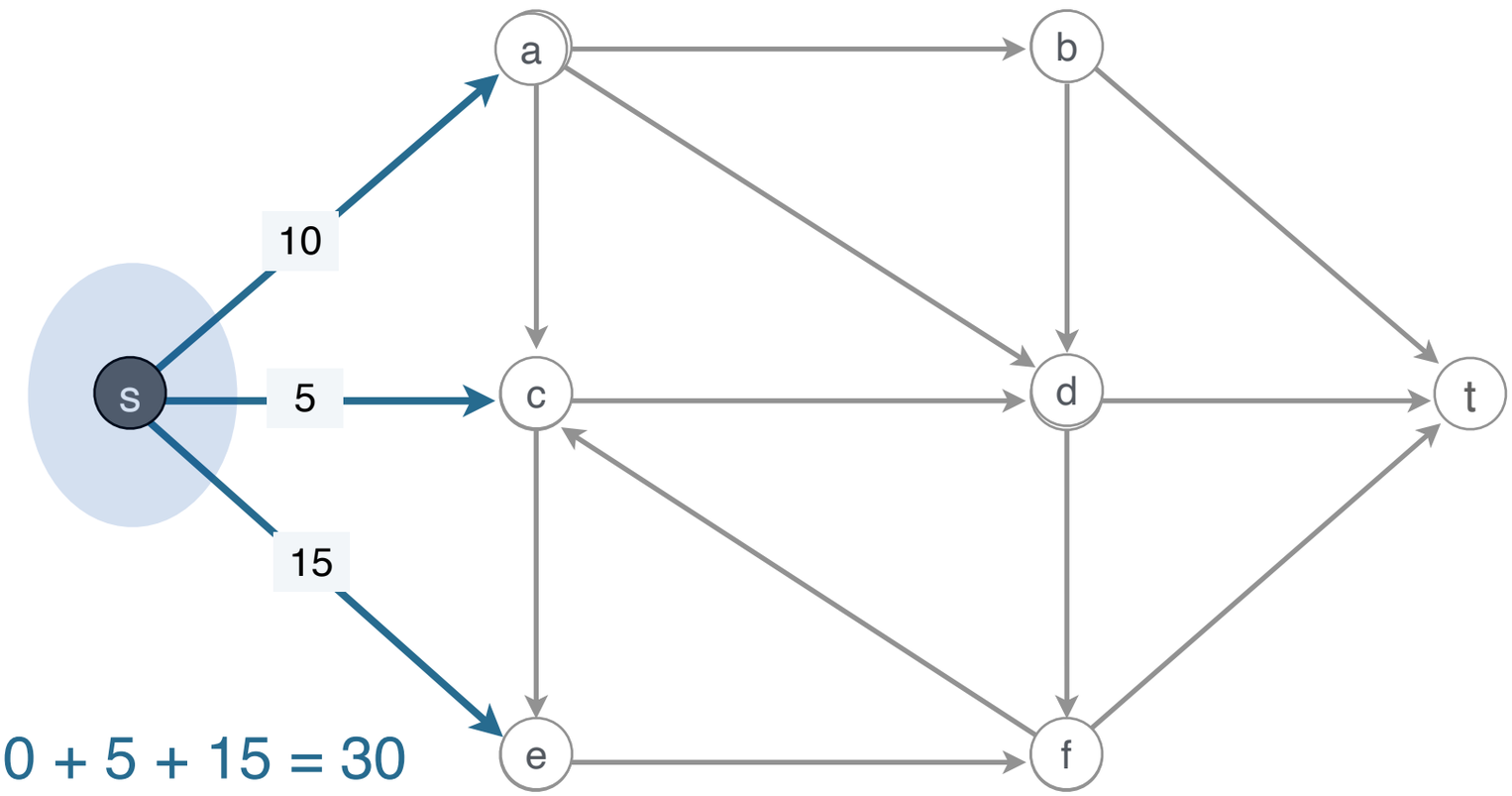


Minimum-cut problem

Def. An st-cut (cut) is a partition (A, B) of the vertices with $s \in A$ and $t \in B$.

Def. Its capacity is the sum of the capacities of the edges from A to B. (i.e. the capacity of edges in the cut-set)

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



A = {s}

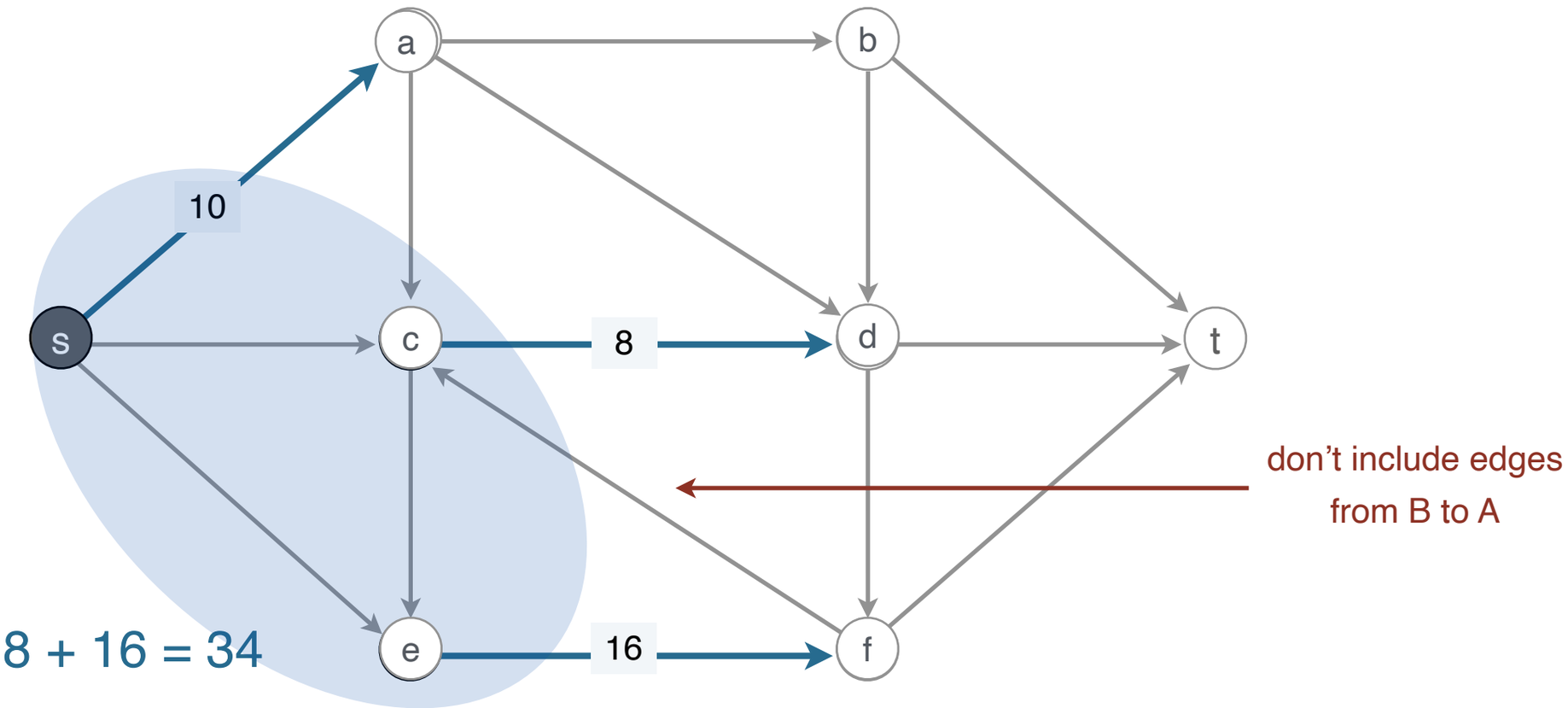
capacity = 10 + 5 + 15 = 30

Minimum-cut problem

Def. An st-cut (cut) is a partition (A, B) of the vertices with $s \in A$ and $t \in B$.

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

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



A = {s, c, e}

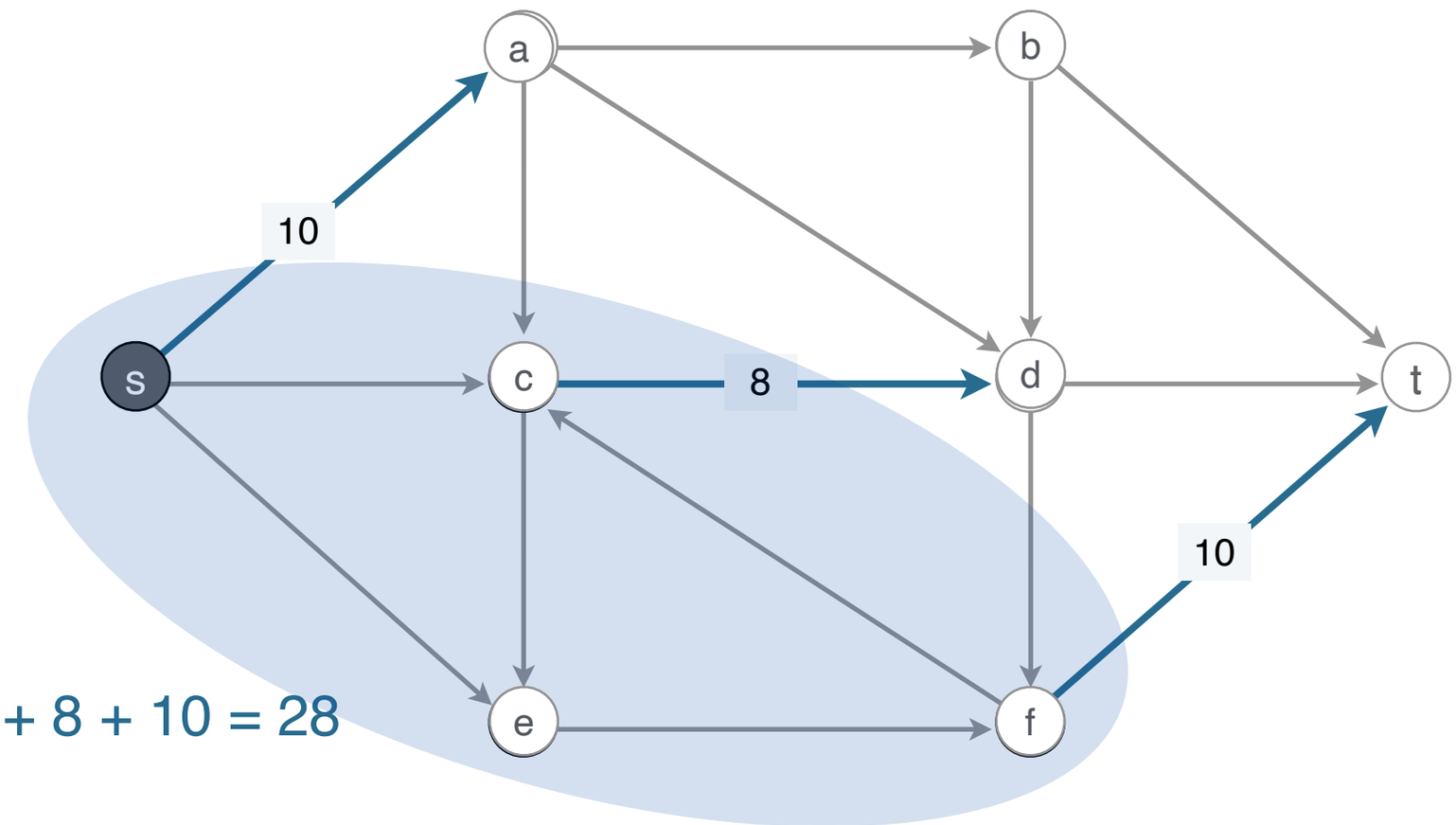
capacity = 10 + 8 + 16 = 34

Minimum-cut problem

Def. An st-cut (cut) is a partition (A, B) of the vertices with $s \in A$ and $t \in B$.

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

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



A = {s, c, e, f}

capacity = 10 + 8 + 10 = 28

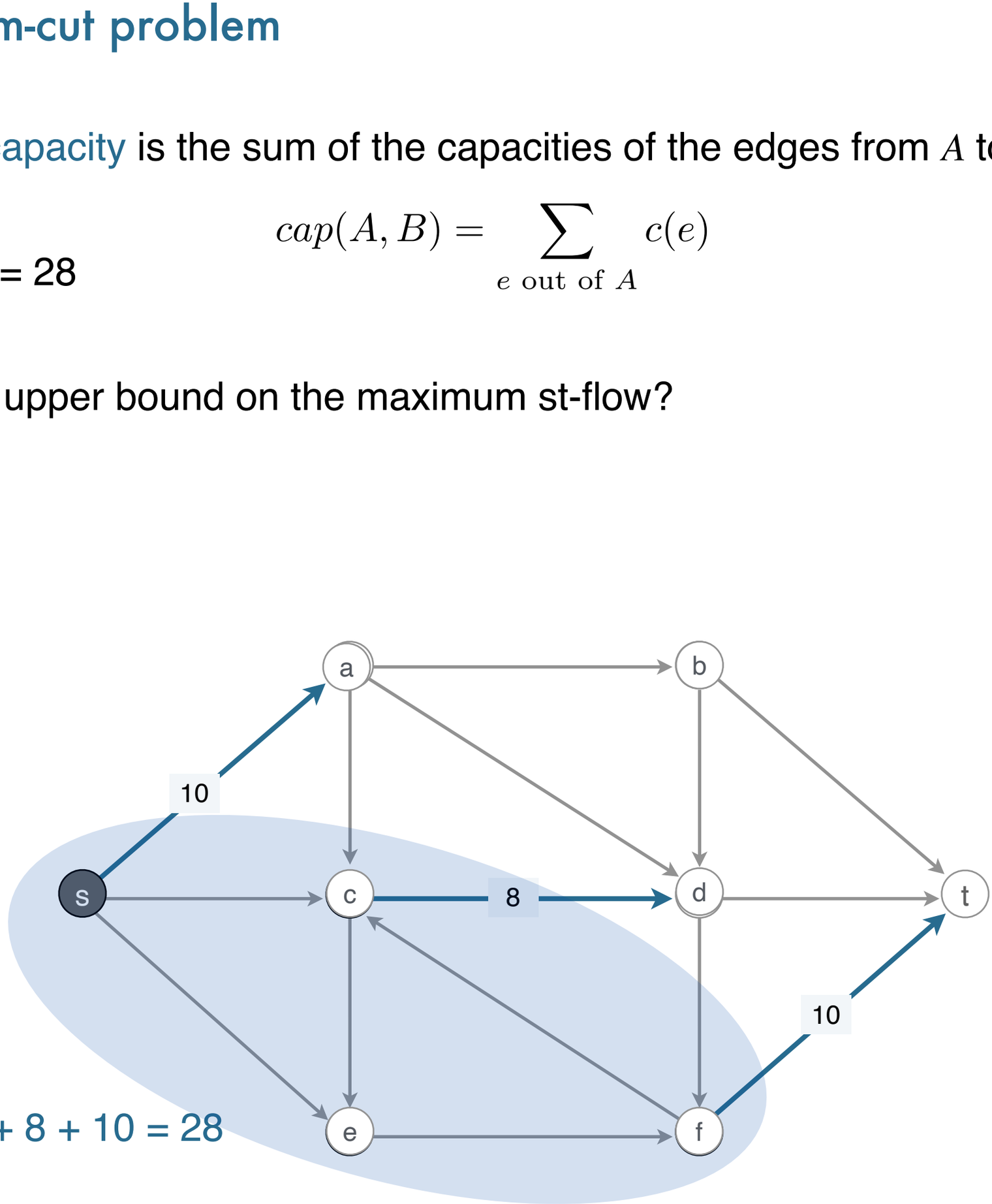
Minimum-cut problem

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

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

cap(A,B) = 28

Is this an upper bound on the maximum st-flow?



A = {s, c, e, f}

capacity = 10 + 8 + 10 = 28

Minimum-cut problem

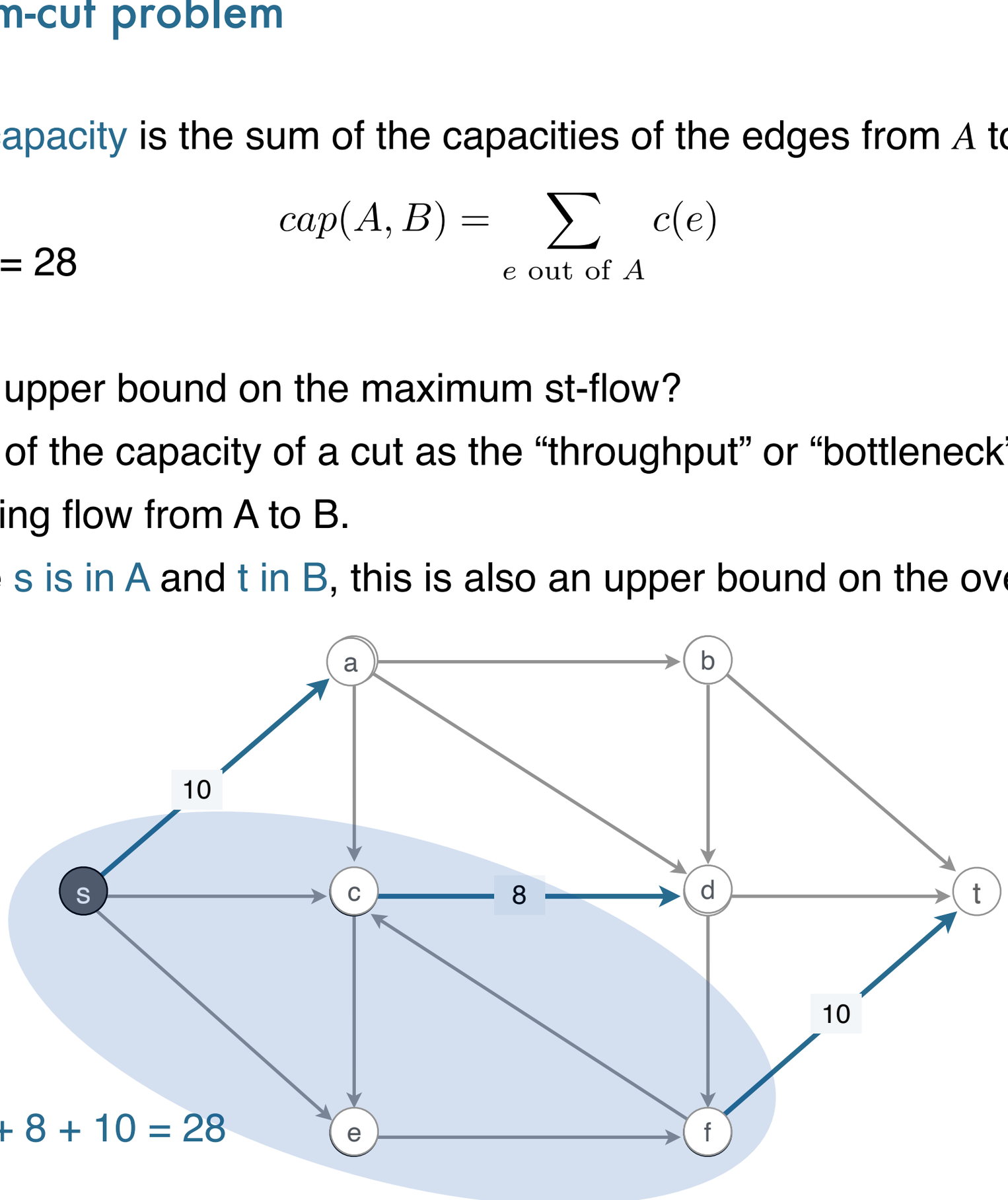
Def. Its capacity is the sum of the capacities of the edges from A to B.

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

cap(A,B) = 28

Is this an upper bound on the maximum st-flow?

- think of the capacity of a cut as the “throughput” or “bottleneck” of the edges carrying flow from A to B.
- since s is in A and t in B, this is also an upper bound on the over flow value



A = {s, c, e, f}

capacity = 10 + 8 + 10 = 28

Minimum-cut problem

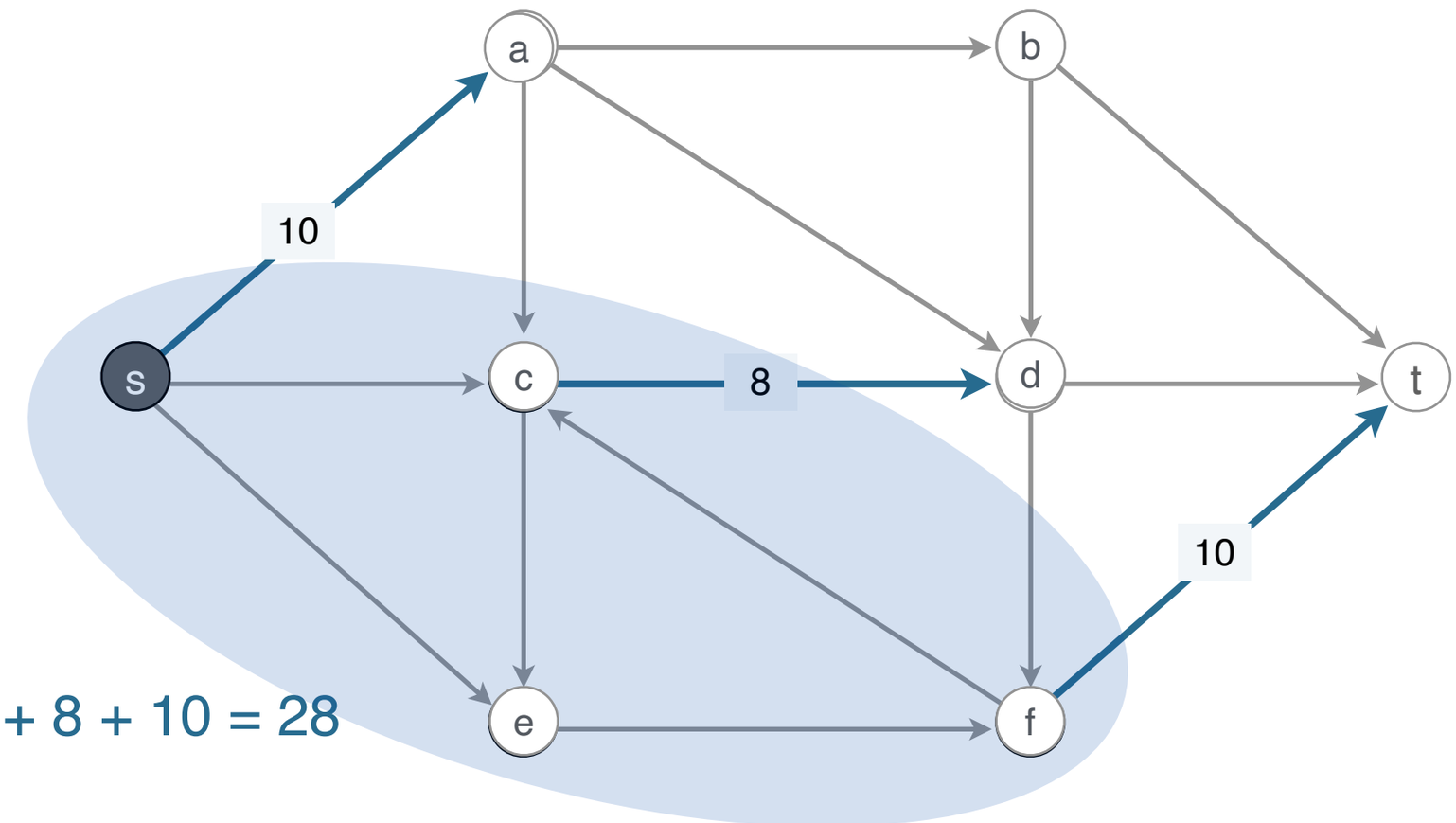
Def. Its capacity is the sum of the capacities of the edges from A to B.

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

cap(A,B) = 28

Min-cut: the st-cut with the lowest capacity in a graph

Min-cut problem: find the minimum capacity st-cut.

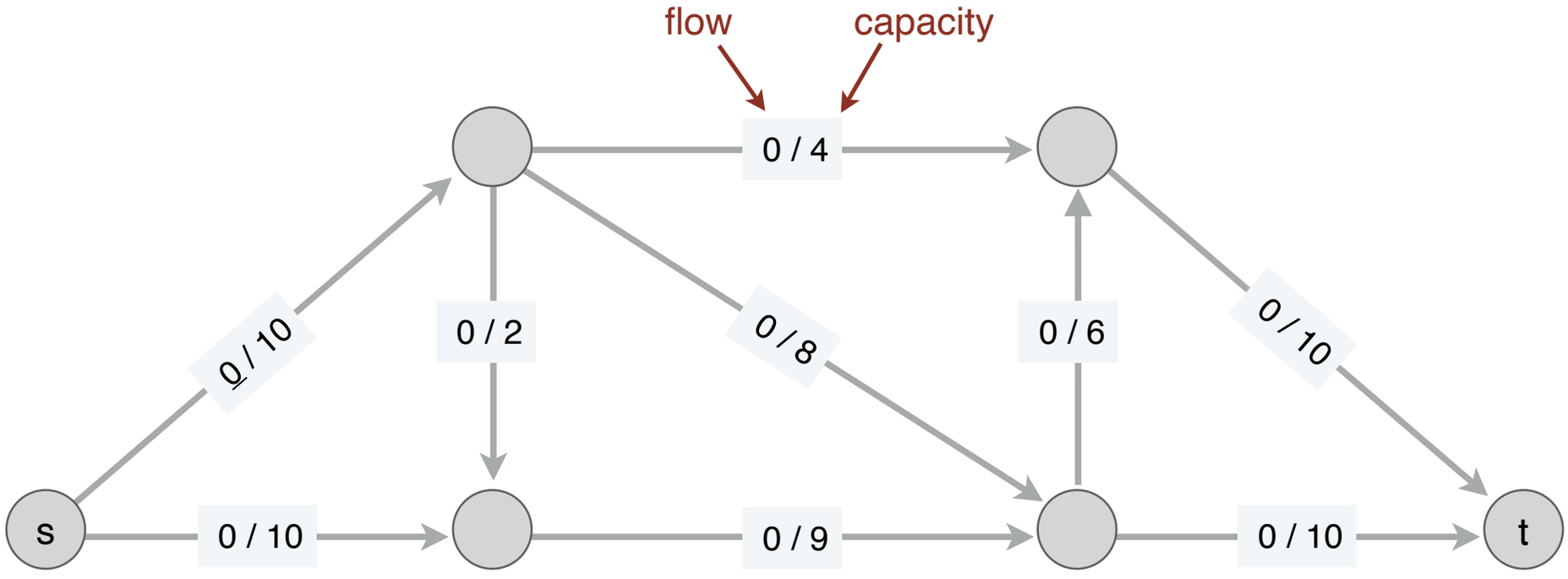


A = {s, c, e, f}

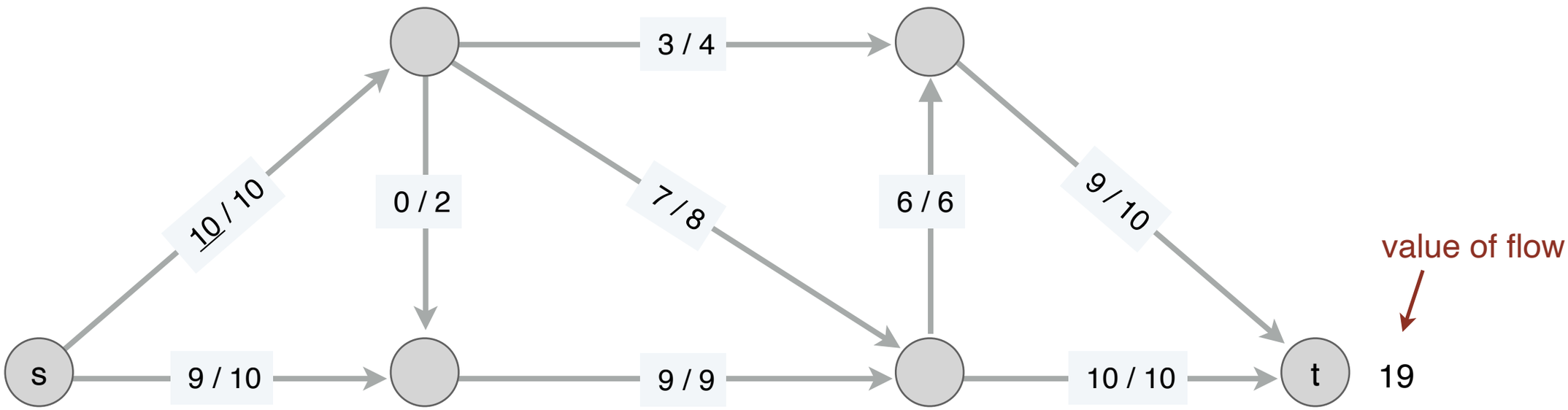
capacity = 10 + 8 + 10 = 28

Certificate for the max flow

- Can you find some proof/certificate for the value of the max flow in this graph?

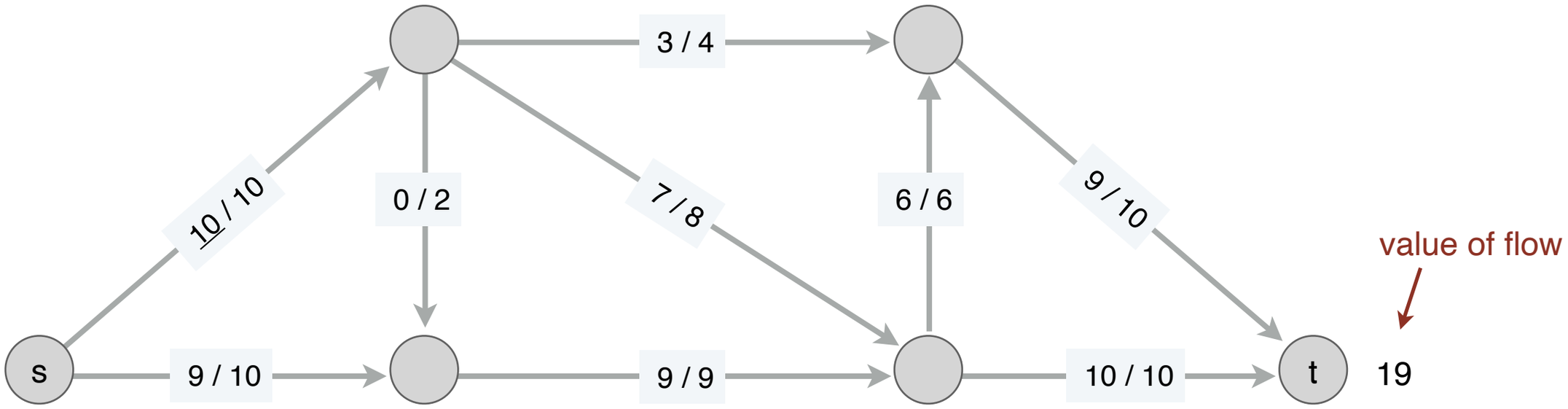


max flow:

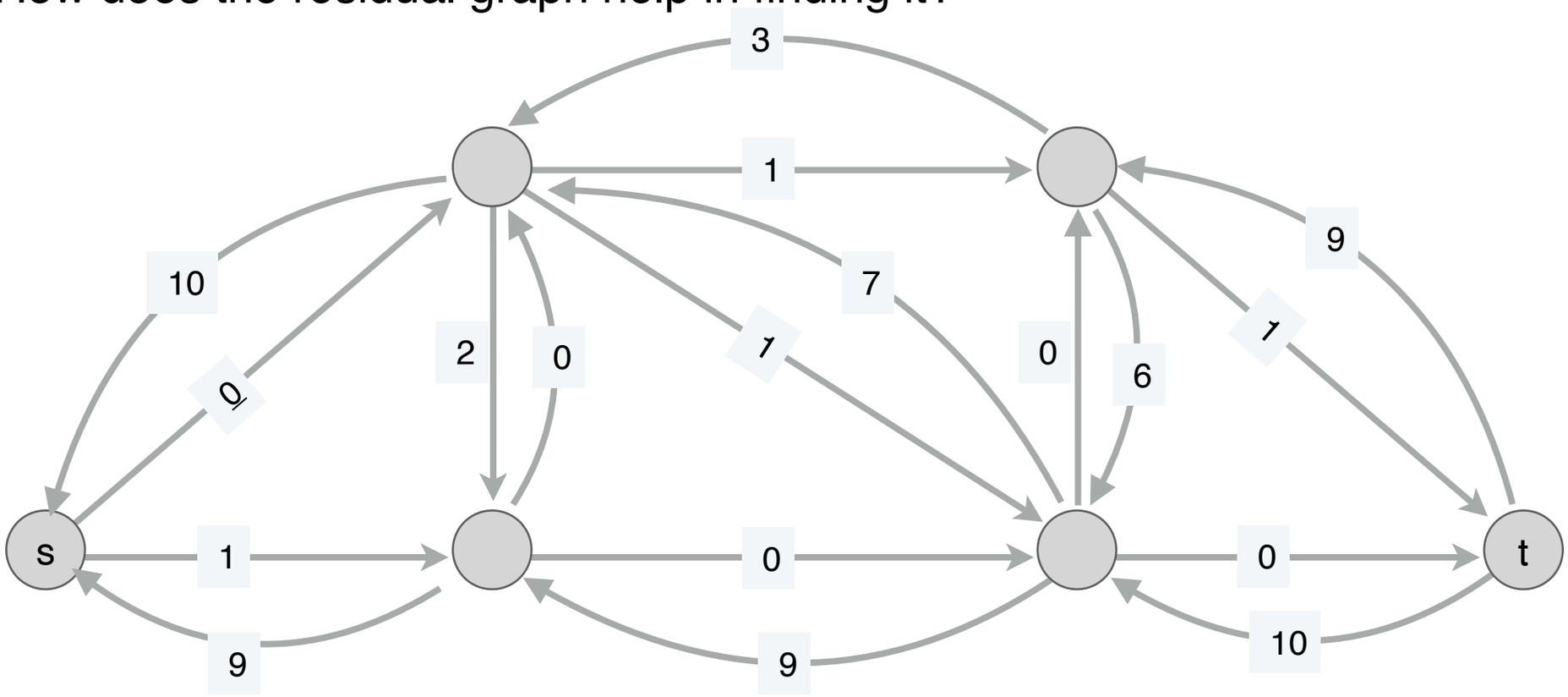


Certificate for the max flow

- Can you find some proof/certificate for the value of the max flow in this graph?



- How does the residual graph help in finding it?



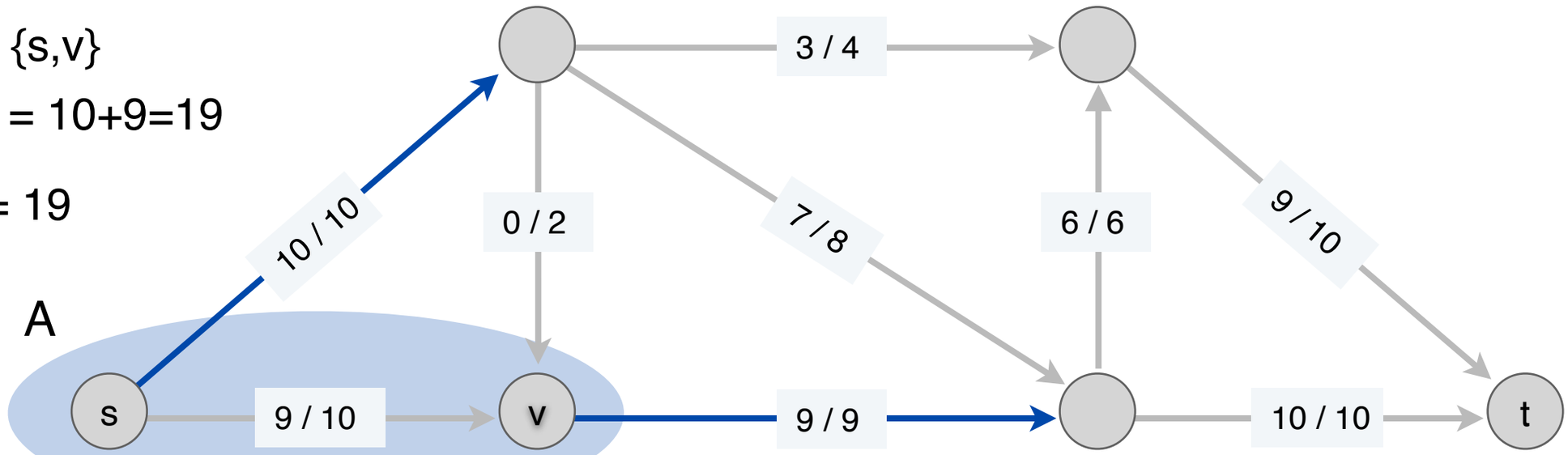
Certificate for the max flow

network G and flow f

cut A = {s,v}

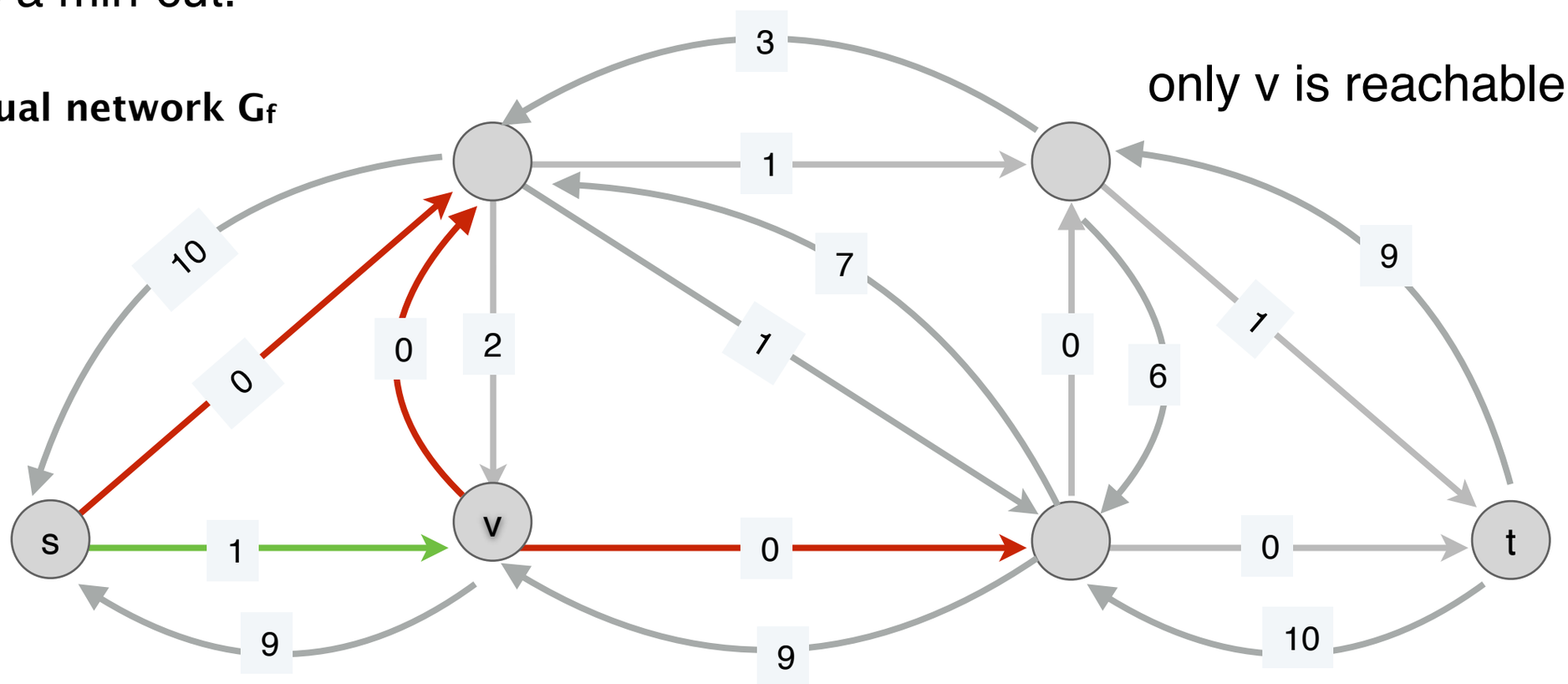
cap(A) = 10+9=19

val(f) = 19



claim: A is a min-cut.

residual network G_f



only v is reachable from s in G_f

Max Flow Min Cut

Max Flow Min Cut (MFMC) theorem:

Given a directed graph *G(V,E)* with source *s*, sink *t* and non-negative capacities *c(e)*, the value of the maximum flow in *G* is equal to the capacity of the minimum st-cut.

$$\max_{f\ flow} val(f) = \min_{A \subseteq V,\ s \in A} cap(A, B)$$

Certificate of optimality: We can use the MFMC theorem to prove that a flow f is maximum;

val(f) is maximum if there is a cut with its capacity equal to *val(f)*.

Finding the min-cut

1. find the maximum flow in G , i.e. run Ford-Fulkerson
2. find the set A of all nodes that are still reachable from the source
   - run BFS from s in G_f to find A
   - A has at least one element, s

Nodes in A form the minimum capacity cut.

Properties of min-cuts

1. is the min-cut always unique?

2. what happens to the max flow if we decrease the capacity of an edge in the min-cut by 1?

Properties of min-cuts

1. is the min-cut always unique?

No - this graph has 3 min cuts.

2. what happens to the max flow if we decrease the capacity of an edge in the min-cut by 1?

Properties of min-cuts

1. is the min-cut always unique?

No - this graph has 3 min cuts.

2. what happens to the max flow if we decrease the capacity of an edge in the min-cut by 1?

The max flow must decrease by one.

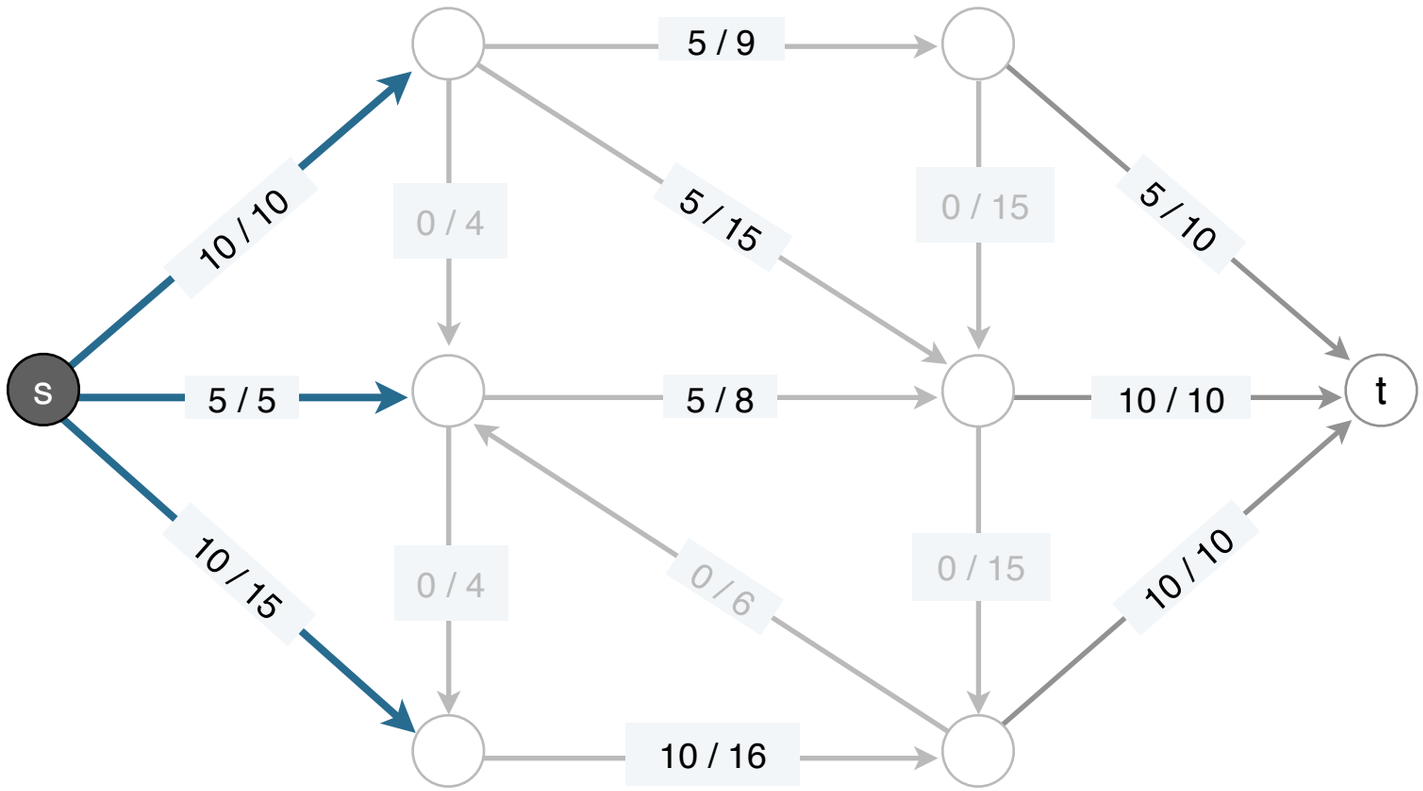
Relationship between flows and cuts — MFMC proof

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

$$val(f) = \sum_{e \text{ leaving } A} f(e) - \sum_{e \text{ entering } A} f(e)$$

cut: A = {s}

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



value of flow = 25

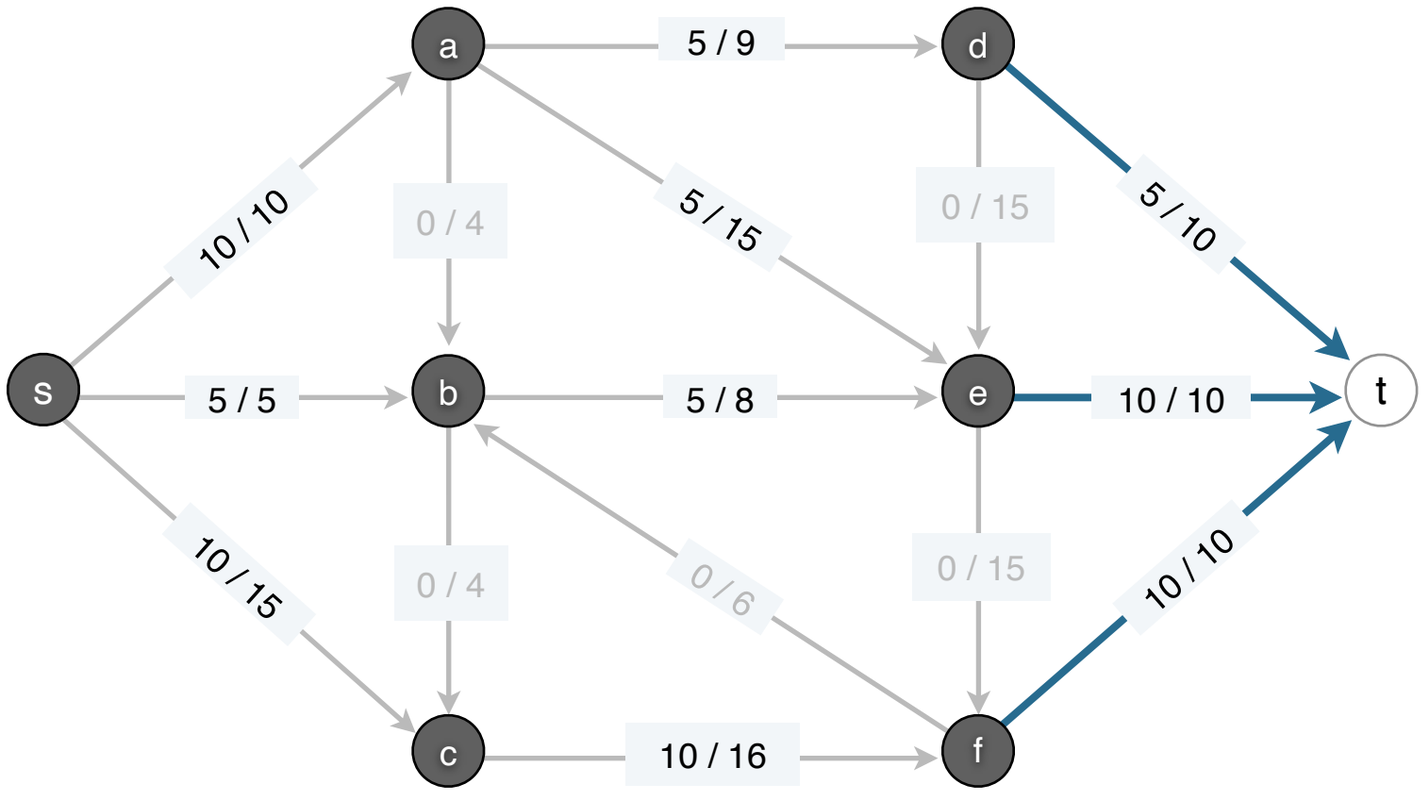
Relationship between flows and cuts — MFMC proof

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

$$val(f) = \sum_{e \text{ leaving } A} f(e) - \sum_{e \text{ entering } A} f(e) = \sum_{u \in A, v \notin A} f(u, v) - \sum_{v \notin A, u \in A} f(v, u)$$

cut: A = {s, a, b, c, d, e, f}

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



value of flow = 25

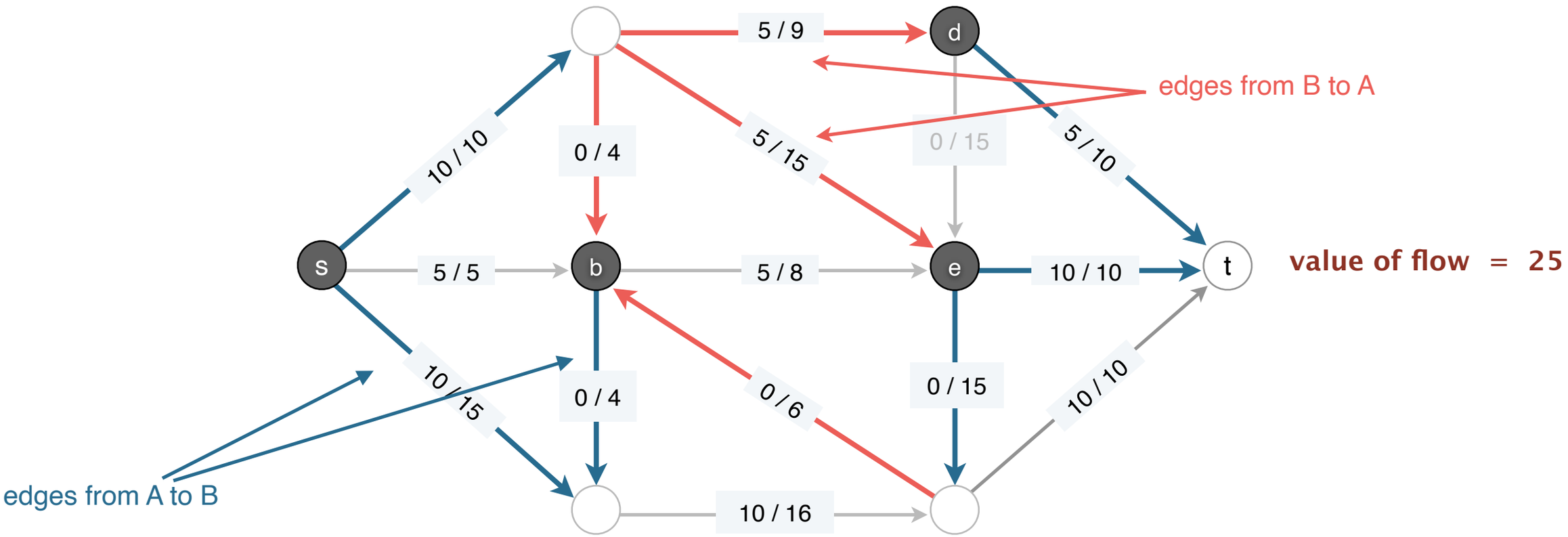
Relationship between flows and cuts — MFMC proof

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

$$val(f) = \sum_{e \text{ leaving } A} f(e) - \sum_{e \text{ entering } A} f(e)$$

cut: A = {s, b, d, e}

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



Relationship between flows and cuts — MFMC proof

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

$$val(f) = \sum_{e \text{ leaving } A} f(e) - \sum_{e \text{ entering } A} f(e)$$

Pf.

Relationship between flows and cuts – MFMC proof

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

$$val(f) = \sum_{e \text{ leaving } A} f(e) - \sum_{e \text{ entering } A} f(e)$$

Pf.

$$val(f) = \sum_{e \text{ leaving } s} f(e) - \sum_{e \text{ entering } s} f(e) =$$

definition of val(f)

edges pointing from v

$$= \sum_{v \in A} \left(\sum_{e=(v,w) \in E} f(e) - \sum_{e=(w,v) \in E} f(e) \right) =$$

by flow conservation, all terms in the sum are 0, except for v = s

edges pointing towards v

$$= \sum_{e \text{ leaving } A} f(e) - \sum_{e \text{ entering } A} f(e)$$

edges with both ends in A appear once with '+' once with '-' in the sum and cancel out. Edges with only one end in A contribute to the sum.

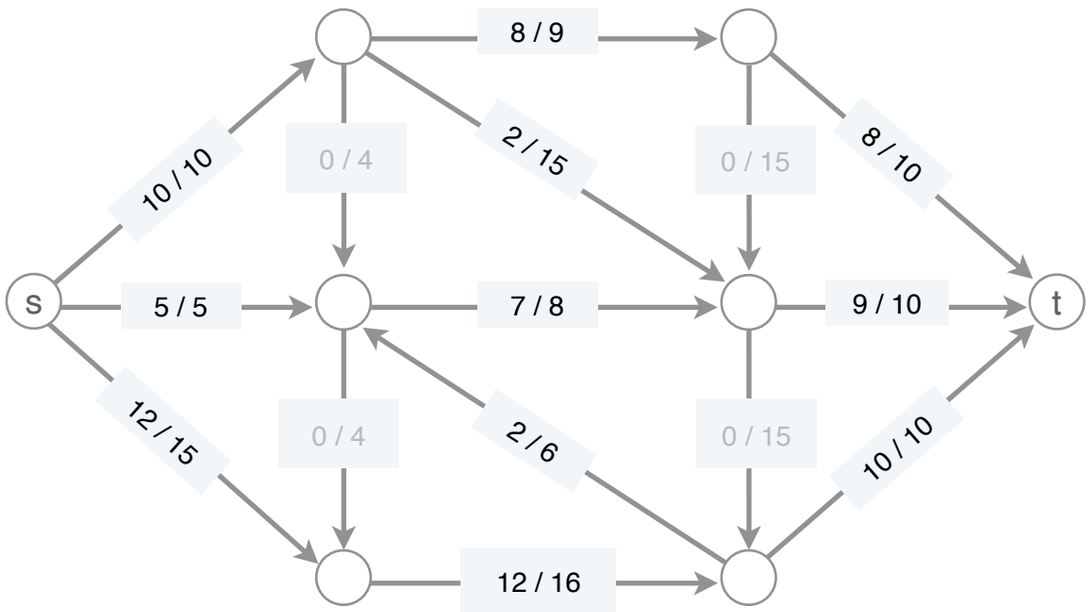
Relationship between flows and cuts — MFMC proof

Weak duality. Let f be *any* flow and (A, B) be *any* cut. Then, $v(f) \leq cap(A, B)$.

Pf.

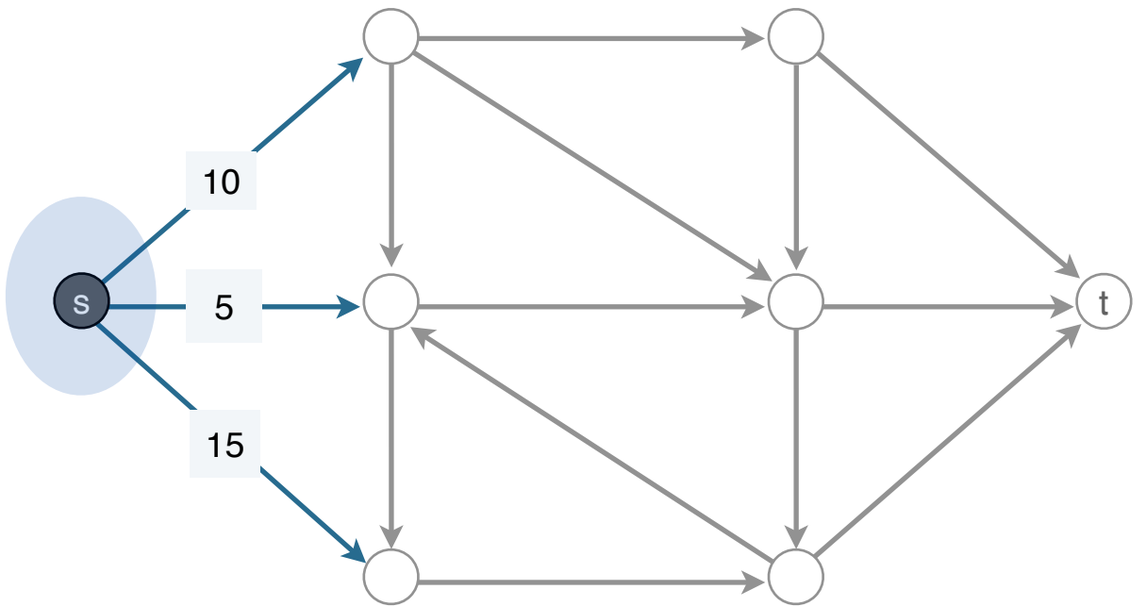
$$\begin{aligned} val(f) &= \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e) \\ &\leq \sum_{e \text{ out of } A} f(e) \\ &\leq \sum_{e \text{ out of } A} c(e) \\ &= cap(A, B) \quad \blacksquare \end{aligned}$$

flow-value lemma



value of flow = 27

≤



capacity of cut = 30

Certificate of optimality – MFMC

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

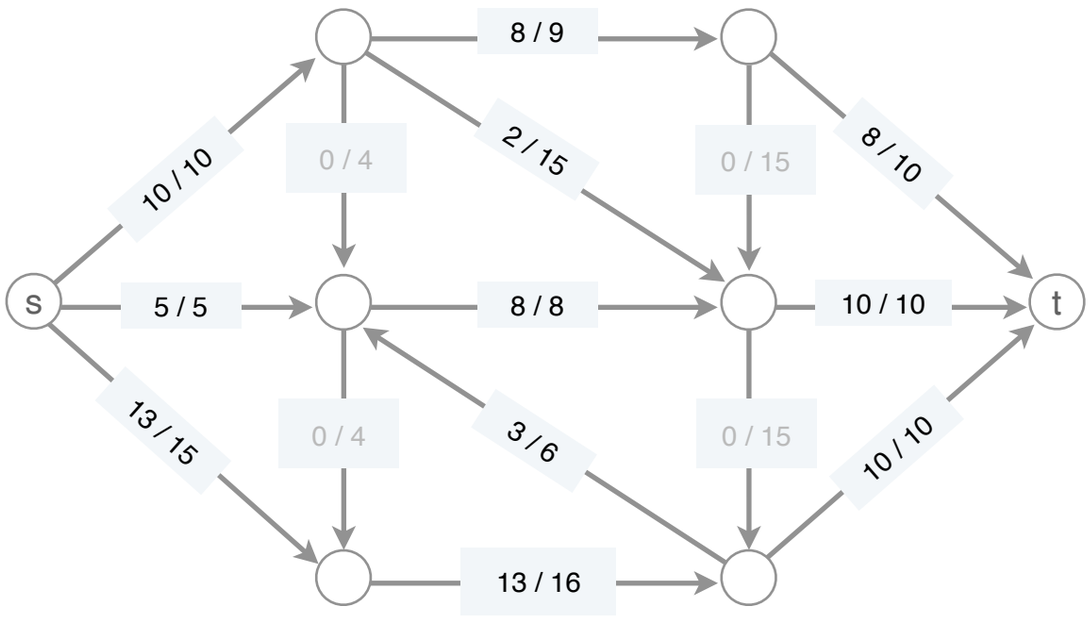
If $val(f) = cap(A, B)$, then f is a max flow and (A, B) is a min cut.

Pf.

weak duality

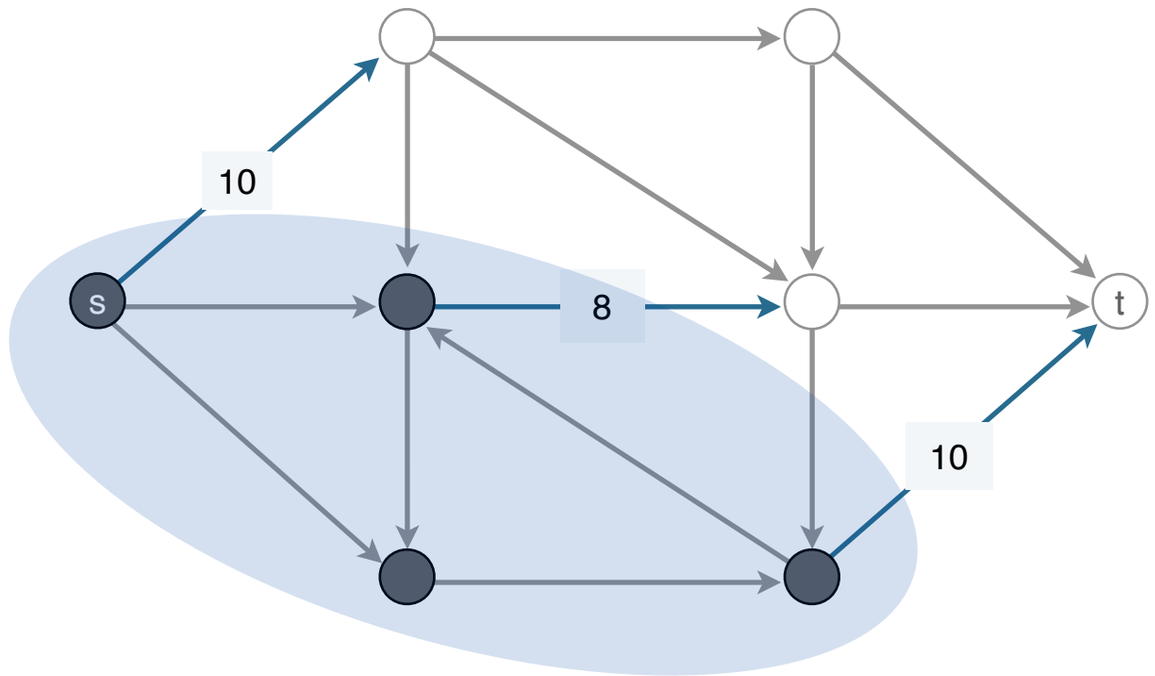
- For any flow f', $val(f') \leq cap(A, B) = val(f)$.
- For any cut (A', B'), $cap(A', B') \geq val(f) = cap(A, B)$.

Conclusion: if we can find a cut with the same capacity as the flow, then it's a maximum flow.



value of flow = 28

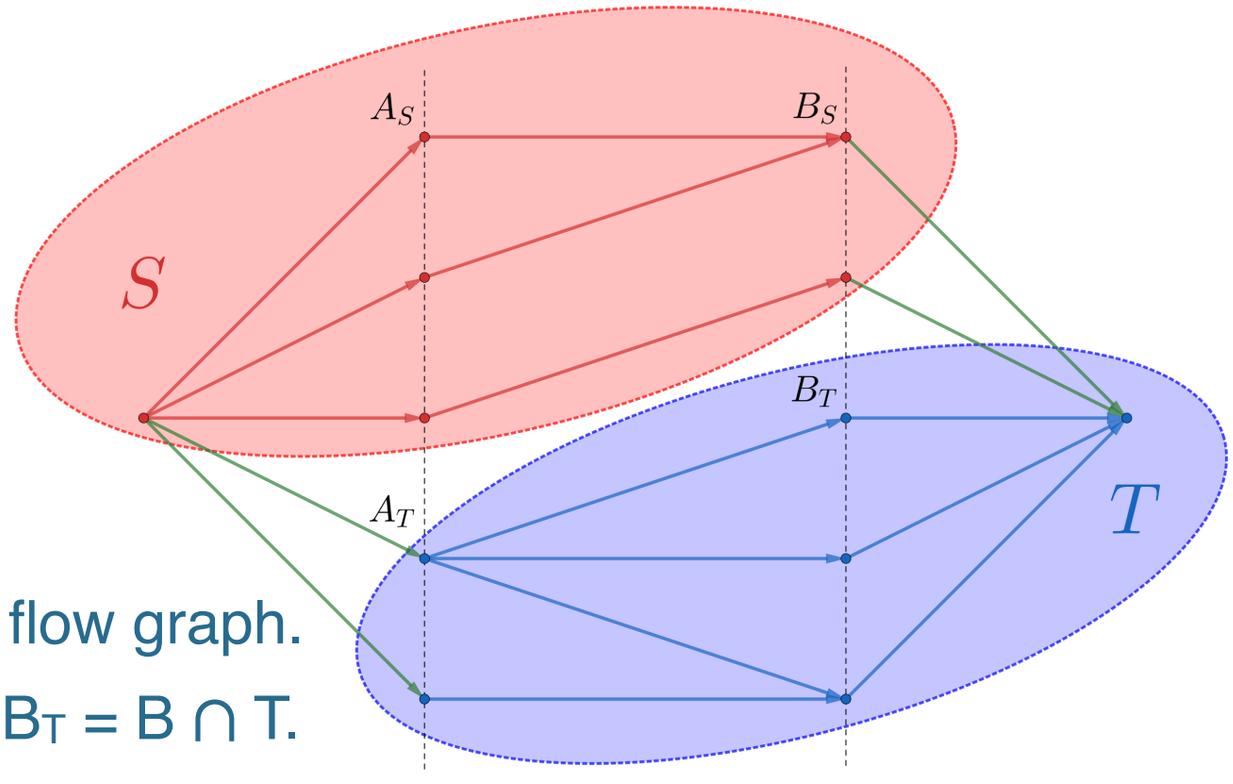
=



capacity of cut = 28

Kőnig's theorem (1931)

The size of the maximum matching of a bipartite graph is the same size as its minimum vertex cover (not set cover!).



1. Let (S, T) be a min-cut of the maximum flow graph.
2. Let $A_S = A \cap S$, $A_T = A \cap T$, $B_S = B \cap S$, $B_T = B \cap T$.
3. The only cut edges are from s to A_T, and B_S to T.
    1. A_S and B_T cannot be connected, since the edge weights would be ∞ and the cut would not be a min-cut. Same for A_T and B_S.
    2. A_S to B_S are internal to S, and A_T to B_T are internal to T.
4. The size of the min-cut is $|A_T| + |B_S|$.
    1. Also the maximum flow and maximum matching.
    2. Lower bound on vertex cover since maximum matching edges are disjoint.
5. $A_T \cup B_S$ is a vertex cover of the bipartite graph.
    1. Any missing edge would have to be from As to B_T, but rejected those above.
    2. Matches lower bound, so this is minimum vertex cover.

Image source: https://en.wikipedia.org/wiki/Kőnig's_theorem_(graph_theory)

Faster Maximum Flow Algorithms

"A new approach to the maximum flow problem"

by Goldberg and Tarjan (1986). "A new approach to the maximum flow problem"

- Preflow-push (Push-relabel) algorithm
- $O(n^2m)$ with basic implementation
- $O(n^2m^{1/2})$ with highest label node selection rule (fastest in practice)
- $O(nm \log(n^2/m))$ using dynamic trees (slow in practice)

Max Flows in $O(nm)$ Time, or Better

by Orlin (2013)

Maximum Flow and Minimum-Cost Flow in Almost-Linear Time

by Chen et al (2022)

- "There is an algorithm that on a graph G with m edges with integral capacities in [1, C] computes a maximum flow between two vertices in time $O(m^{1+o(1)} \log C)$ with high probability."
  (variables tweaked to match these slides, added O() to statement)

Faster Maximum Flow Algorithms

"A new approach to the maximum flow problem"

by Goldberg and Tarjan (1986). "A new approach to the maximum flow problem"

- Preflow-push (Push-relabel) algorithm
- $O(n^2m)$ with basic implementation
- $O(n^2m^{1/2})$ with highest label node selection rule (fastest in practice)
- $O(nm \log(n^2/m))$ using dynamic trees (slow in practice)

Max Flows in $O(nm)$ Time, or Better

by Orlin (2013)

Maximum Flow and Minimum-Cost Flow in Almost-Linear Time

by Chen et al (2022)

- "There is an algorithm that on a graph G with m edges with integral capacities in [1, C] computes a maximum flow between two vertices in time $O(m^{1+o(1)}$ ***log C***$)$ **with high probability**."

  (variables tweaked to match these slides, added O() to statement)