Karlsruhe Institute of Technology

Hasso Plattner Institut
IT Systems Engineering | Universität Potsdam

Competitive Programming

Winter Term 23/24

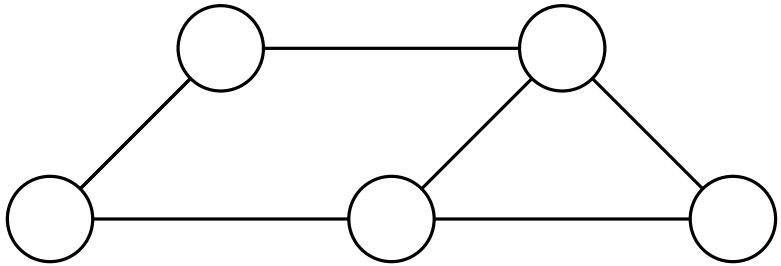
MaxFlow & MinCostFlow

Kirill Simonov
Algorithm Engineering Group
HPI

Christopher Weyand
Scalable Algorithms Group
KIT

Reminder: Flows

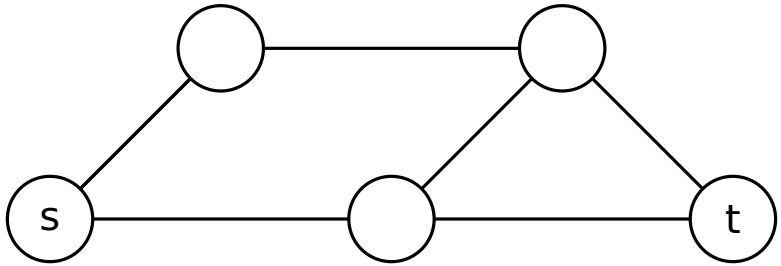
undirected network $G = (V, E)$



Reminder: Flows

undirected network $G = (V, E)$

a source and a sink $s, t \in V$

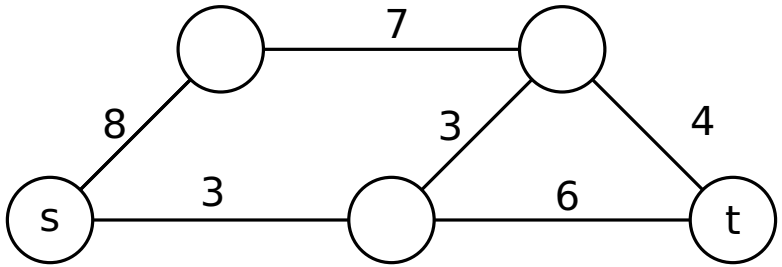


Reminder: Flows

undirected network $G = (V, E)$

a source and a sink $s, t \in V$

a capacity function $c : E \mapsto \mathbb{N}$



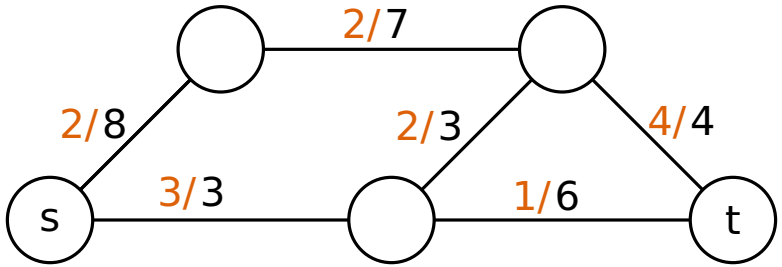
Reminder: Flows

undirected network $G = (V, E)$

a source and a sink $s, t \in V$

a capacity function $c : E \mapsto \mathbb{N}$

a flow $f : E \to \mathbb{Z}$ with...



Reminder: Flows

undirected network $G = (V, E)$

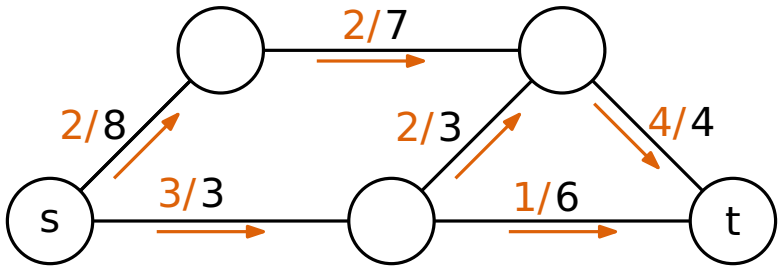
a source and a sink $s, t \in V$

a capacity function $c : E \mapsto \mathbb{N}$

a flow $f : E \to \mathbb{Z}$ with...

- consistency

$$f(u, v) = -f(v, u)$$



Reminder: Flows

undirected network $G = (V, E)$

a source and a sink $s, t \in V$

a capacity function $c : E \mapsto \mathbb{N}$

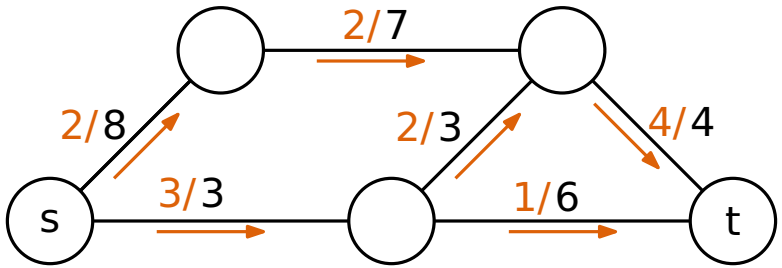
a flow $f : E \to \mathbb{Z}$ with...

- consistency

  $$f(u, v) = -f(v, u)$$

- capacity constraint

  $$f(u, v) \leq c(\{u, v\})$$



Reminder: Flows

undirected network $G = (V, E)$

a source and a sink $s, t \in V$

a capacity function $c : E \mapsto \mathbb{N}$

a flow $f : E \to \mathbb{Z}$ with...

- consistency

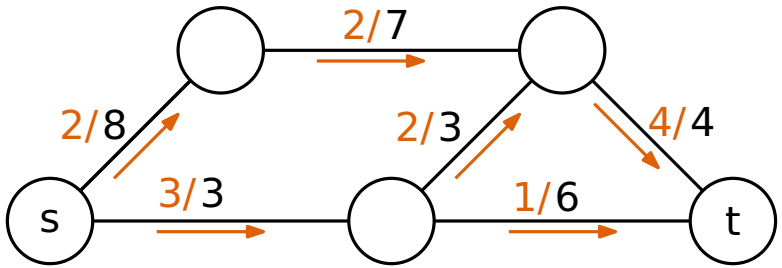
  $$f(u, v) = -f(v, u)$$

- capacity constraint

  $$f(u, v) \leq c(\{u, v\})$$

- conservation of flow $\quad \forall v \in V \setminus \{s, t\}$:

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



Reminder: Flows

undirected network $G = (V, E)$

a source and a sink $s, t \in V$

a capacity function $c : E \mapsto \mathbb{N}$

a flow $f : E \to \mathbb{Z}$ with...

- consistency

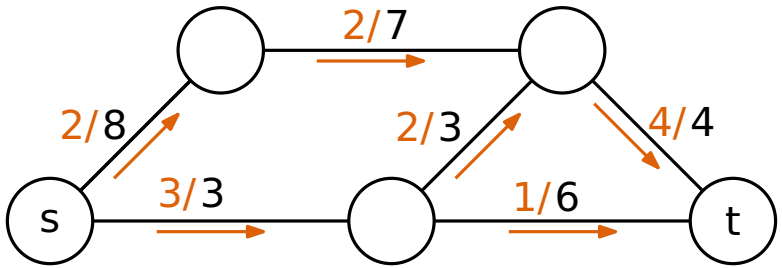
  $$f(u, v) = -f(v, u)$$

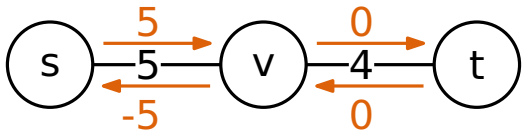
- capacity constraint

  $$f(u, v) \leq c(\{u, v\})$$

- conservation of flow $\forall v \in V \setminus \{s, t\}$:

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





Reminder: Flows

undirected network $G = (V, E)$

a source and a sink $s, t \in V$

a capacity function $c : E \mapsto \mathbb{N}$

a flow $f : E \to \mathbb{Z}$ with...

- consistency

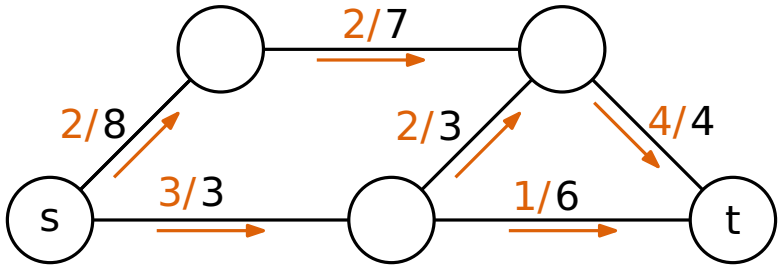
  $$f(u, v) = -f(v, u)$$

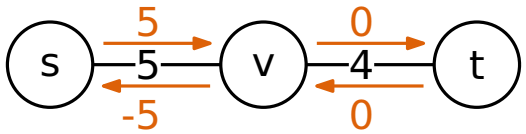
- capacity constraint

  $$f(u, v) \leq c(\{u, v\})$$

- conservation of flow $\quad \forall v \in V \setminus \{s, t\}$:

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





$$f(s, v) - f(v, s) = 5 - (-5) \neq 0$$

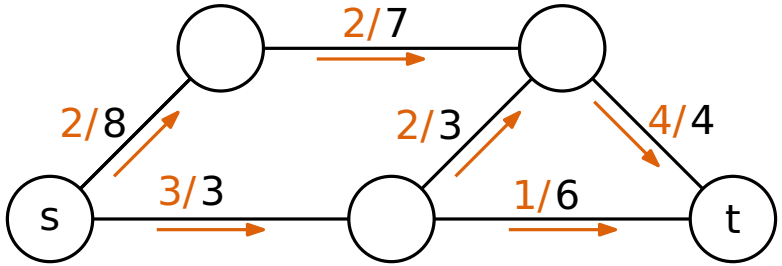
Reminder: Flows

undirected network $G = (V, E)$

a source and a sink $s, t \in V$

a capacity function $c : E \mapsto \mathbb{N}$

a flow $f : E \to \mathbb{Z}$ with...



- consistency

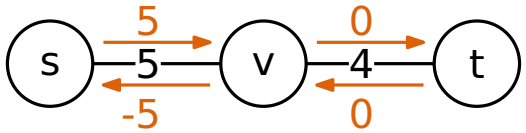
  $$f(u, v) = -f(v, u)$$

- capacity constraint

  $$f(u, v) \leq c(\{u, v\})$$

- conservation of flow $\forall v \in V \setminus \{s, t\}$:

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



$$f(s, v) - f(v, s) = 5 - (-5) \neq 0$$

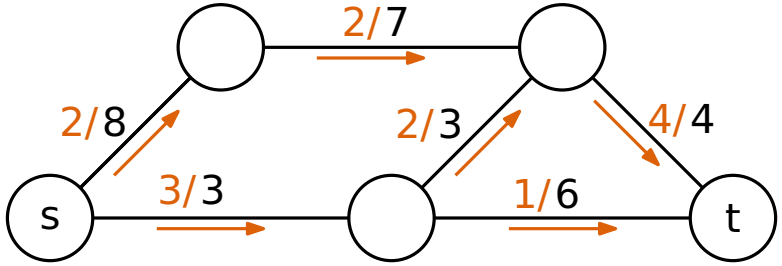
- value of flow is amount of flow outgoing from s

Reminder: Flows

- a flow f in network G implies a **residual network** G_f
- G_f contains the directed edge (u, v) iff $f(u, v) < c(\{u, v\})$
- edges in G_f have capacity $c(\{u, v\}) - f(u, v)$

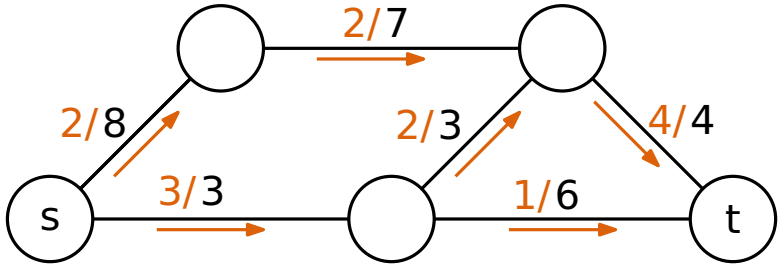
Reminder: Flows

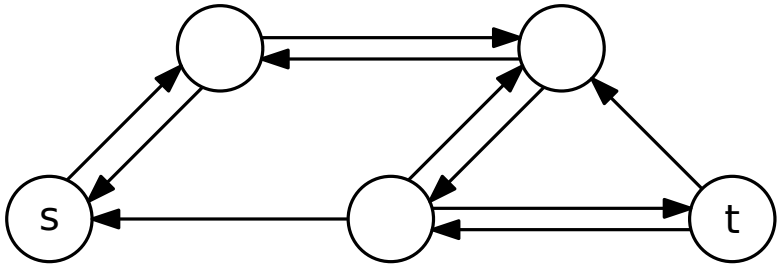
- a flow f in network G implies a **residual network** G_f
- G_f contains the directed edge (u, v) iff $f(u, v) < c(\{u, v\})$
- edges in G_f have capacity $c(\{u, v\}) - f(u, v)$



Reminder: Flows

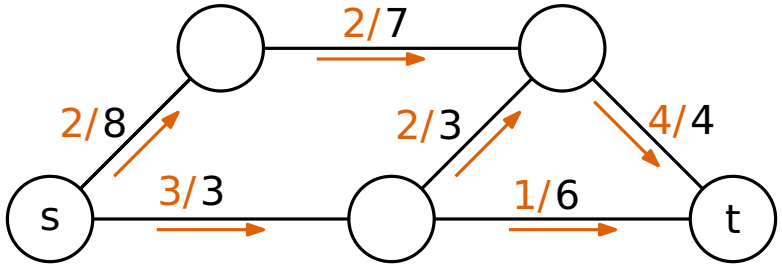
- a flow f in network G implies a **residual network** G_f
- G_f contains the directed edge (u, v) iff $f(u, v) < c(\{u, v\})$
- edges in G_f have capacity $c(\{u, v\}) - f(u, v)$

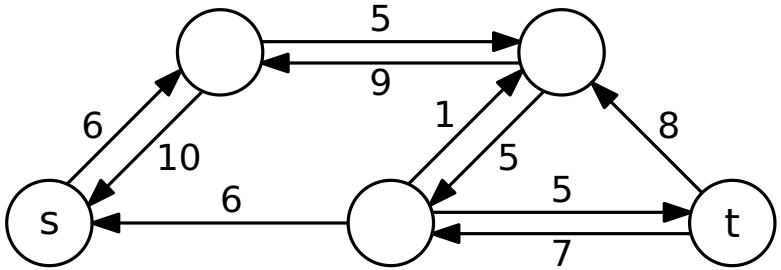




Reminder: Flows

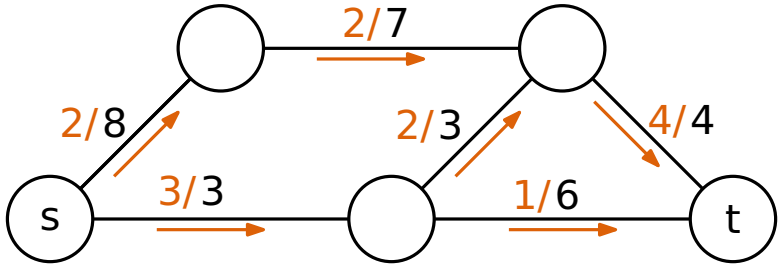
- a flow f in network G implies a **residual network** G_f
- G_f contains the directed edge (u, v) iff $f(u, v) < c(\{u, v\})$
- edges in G_f have capacity $c(\{u, v\}) - f(u, v)$

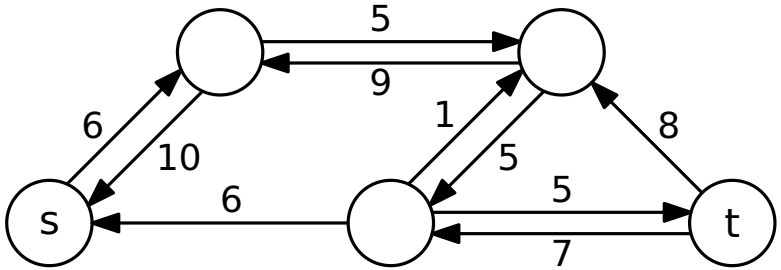




Reminder: Flows

- a flow f in network G implies a **residual network** G_f
- G_f contains the directed edge (u, v) iff $f(u, v) < c(\{u, v\})$
- edges in G_f have capacity $c(\{u, v\}) - f(u, v)$



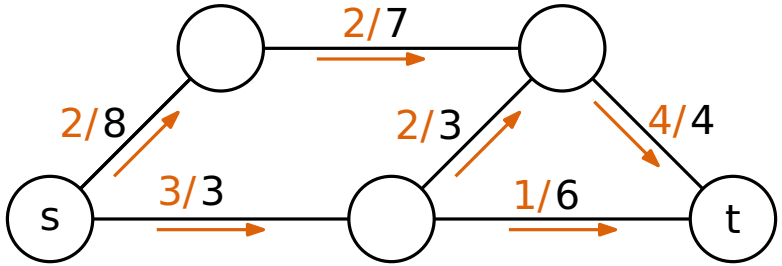


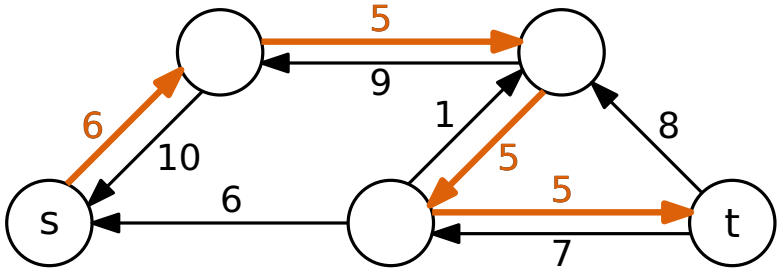
- let a s-t path in G_f be called an **augmenting path**

Observation: If there is an augmenting path in G_f, we can increase the flow.

Reminder: Flows

- a flow f in network G implies a **residual network** G_f
- G_f contains the directed edge (u, v) iff $f(u, v) < c(\{u, v\})$
- edges in G_f have capacity $c(\{u, v\}) - f(u, v)$





- let a s-t path in G_f be called an **augmenting path**

Observation: If there is an augmenting path in G_f, we can increase the flow.

Ford-Fulkerson

- idea: find augmenting path, augment, repeat

Ford-Fulkerson

- idea: find augmenting path, augment, repeat

Ford-Fulkerson MaxFlow Algorithm

```
int flow = 0;
while (true) {
    // find augmenting path
    // break if no s-t path found
    // find path bottleneck
    // augment path
}
```

Ford-Fulkerson

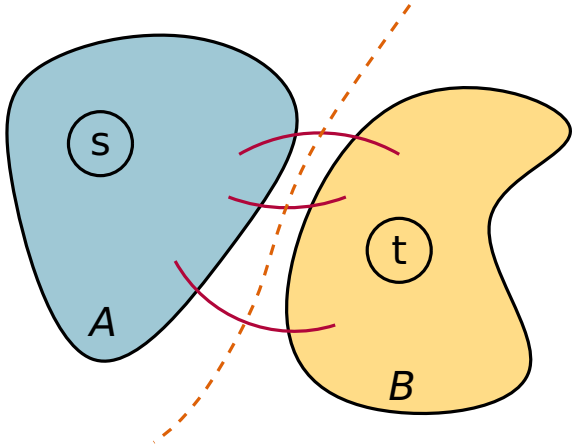
- idea: find augmenting path, augment, repeat
- running time: $O(mC)$
- C is value of MaxFlow

Ford-Fulkerson MaxFlow Algorithm

$\Theta(1)$	`int flow = 0;`
$\leq C$ times	`while (true) {`
$\Theta(m)$	`  // find augmenting path`
$\Theta(1)$	`  // break if no s-t path found`
$O(m)$	`  // find path bottleneck`
$O(m)$	`  // augment path`
	`}`

Ford-Fulkerson

- idea: find augmenting path, augment, repeat
- running time: $O(mC)$
- C is value of MaxFlow
- correctness:

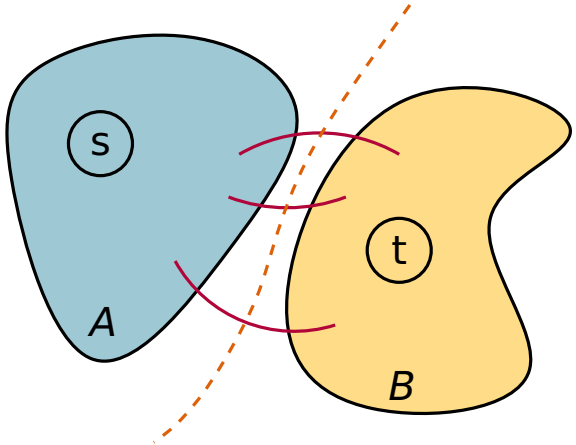


Ford-Fulkerson MaxFlow Algorithm

$\Theta(1)$	`int flow = 0;`
$\leq C$ times	`while (true) {`
$\Theta(m)$	`  // find augmenting path`
$\Theta(1)$	`  // break if no s-t path found`
$O(m)$	`  // find path bottleneck`
$O(m)$	`  // augment path`
	`}`

Ford-Fulkerson

- idea: find augmenting path, augment, repeat
- running time: $O(mC)$
- C is value of MaxFlow
- correctness:



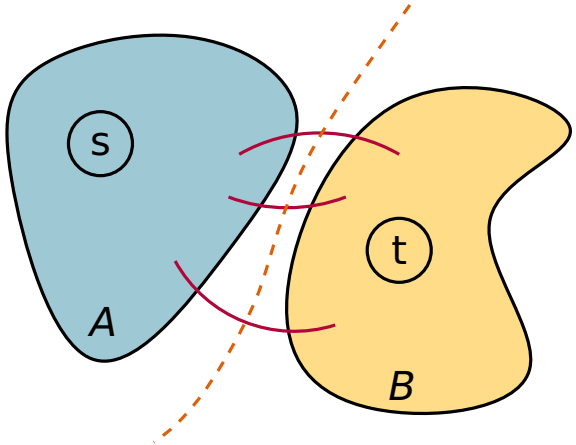
Ford-Fulkerson MaxFlow Algorithm

$\Theta(1)$	`int flow = 0;`
$\leq C$ times	`while (true) {`
$\Theta(m)$	`  // find augmenting path`
$\Theta(1)$	`  // break if no s-t path found`
$O(m)$	`  // find path bottleneck`
$O(m)$	`  // augment path`
	`}`

- capacity of a cut is sum of edge capacities going from A to B

Ford-Fulkerson

- idea: find augmenting path, augment, repeat
- running time: $O(mC)$
- C is value of MaxFlow
- correctness:



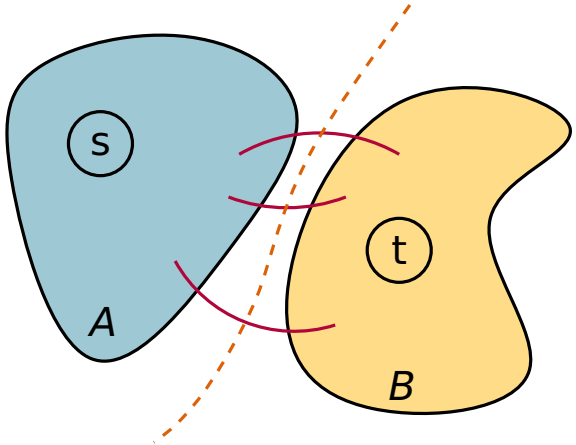
Ford-Fulkerson MaxFlow Algorithm

$\Theta(1)$	`int flow = 0;`
$\leq C$ times	`while (true) {`
$\Theta(m)$	`// find augmenting path`
$\Theta(1)$	`// break if no s-t path found`
$O(m)$	`// find path bottleneck`
$O(m)$	`// augment path`
	`}`

- capacity of a cut is sum of edge capacities going from A to B
- a flow without augmenting path implies a cut of same size (later) $\rightarrow$ max-flow $\geq$ min-cut

Ford-Fulkerson

- idea: find augmenting path, augment, repeat
- running time: $O(mC)$
- C is value of MaxFlow
- correctness:



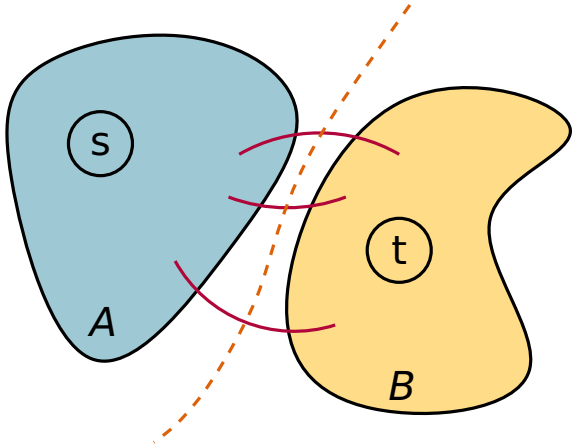
Ford-Fulkerson MaxFlow Algorithm

$\Theta(1)$	`int flow = 0;`
$\leq C$ times	`while (true) {`
$\Theta(m)$	`// find augmenting path`
$\Theta(1)$	`// break if no s-t path found`
$O(m)$	`// find path bottleneck`
$O(m)$	`// augment path`
	`}`

- capacity of a cut is sum of edge capacities going from A to B
- a flow without augmenting path implies a cut of same size (later) $\rightarrow$ max-flow $\geq$ min-cut
- every s-t cut limits the maximum flow $\rightarrow$ max-flow $\leq$ min-cut

Ford-Fulkerson

- idea: find augmenting path, augment, repeat
- running time: $O(mC)$
- C is value of MaxFlow
- correctness:



Ford-Fulkerson MaxFlow Algorithm

$\Theta(1)$	`int flow = 0;`
$\leq C$ times	`while (true) {`
$\Theta(m)$	`// find augmenting path`
$\Theta(1)$	`// break if no s-t path found`
$O(m)$	`// find path bottleneck`
$O(m)$	`// augment path`
	`}`

- capacity of a cut is sum of edge capacities going from A to B
- a flow without augmenting path implies a cut of same size (later) $\rightarrow$ max-flow $\geq$ min-cut
- every s-t cut limits the maximum flow $\rightarrow$ max-flow $\leq$ min-cut
- **MaxFlow = MinCut**

Ford-Fulkerson Implementation

```
struct edge {
    int from,to;
    int flow,cap;
    edge* twin;
}
// resize me pls
vector<vector<edge*>> adj;
```

Ford-Fulkerson Implementation

```
struct edge {
    int from,to;
    int flow,cap;
    edge* twin;
}
// resize me pls
vector<vector<edge*>> adj;

void add_edge(int a, int b, int cap) {
    auto ab = new edge{a, b, 0, cap, nullptr};
    auto ba = new edge{b, a, 0, cap, nullptr};
    // auto ba = new edge{b, a, 0, 0, nullptr};
    ab->twin = ba;
    ba->twin = ab;
    adj[a].push_back(ab);
    adj[b].push_back(ba);
}
```

Ford-Fulkerson Implementation

```
struct edge {
    int from,to;
    int flow,cap;
    edge* twin;
}
// resize me pls
vector<vector<edge*>> adj;

void add_edge(int a, int b, int cap) {
    auto ab = new edge{a, b, 0, cap, nullptr};
    auto ba = new edge{b, a, 0, cap, nullptr};
    // auto ba = new edge{b, a, 0, 0, nullptr};
    ab->twin = ba;
    ba->twin = ab;
    adj[a].push_back(ab);
    adj[b].push_back(ba);
}
```

uncomment to make it directed

Ford-Fulkerson Impl. (Edmonds-Karp)

```
// find augmenting path
vector<edge*> inc(n, nullptr);
queue<int> q{{s}};
while (!q.empty()) {
    auto v = q.front(); q.pop();
    for (auto e : adj[v]) {
        if (!inc[e->to] &&
            e->flow < e->cap) {
            q.push(e->to);
            inc[e->to] = e;
        }
    }
}
```

Ford-Fulkerson Impl. (Edmonds-Karp)

```
// find augmenting path
vector<edge*> inc(n, nullptr);
queue<int> q{{s}};
while (!q.empty()) {
    auto v = q.front(); q.pop();
    for (auto e : adj[v]) {
        if (!inc[e->to] &&
            e->flow < e->cap) {
            q.push(e->to);
            inc[e->to] = e;
        }
    }
}

// break if no s-t path found
if (!inc[t]) break;
```

Ford-Fulkerson Impl. (Edmonds-Karp)

```
// find augmenting path
vector<edge*> inc(n, nullptr);
queue<int> q{{s}};
while (!q.empty()) {
    auto v = q.front(); q.pop();
    for (auto e : adj[v]) {
        if (!inc[e->to] &&
            e->flow < e->cap) {
            q.push(e->to);
            inc[e->to] = e;
        }
    }
}

// break if no s-t path found
if (!inc[t]) break;
```

- simple BFS in residual network
- saves edges to reconstruct the path
- also use incoming edges to track visited nodes

Ford-Fulkerson Impl. (Edmonds-Karp)

```
// find augmenting path
vector<edge*> inc(n, nullptr);
queue<int> q{{s}};
while (!q.empty()) {
    auto v = q.front(); q.pop();
    for (auto e : adj[v]) {
        if (!inc[e->to] &&
            e->flow < e->cap) {
            q.push(e->to);
            inc[e->to] = e;
        }
    }
}

// break if no s-t path found
if (!inc[t]) break;
```

- simple BFS in residual network
- saves edges to reconstruct the path
- also use incoming edges to track visited nodes
- by augmenting shortest paths we accidentally implemented Edmonds-Karp

$$O(nm^2)$$

Ford-Fulkerson Impl. (Edmonds-Karp)

```
// find augmenting path
vector<edge*> inc(n, nullptr);
queue<int> q{{s}};
while (!q.empty()) {
    auto v = q.front(); q.pop();
    for (auto e : adj[v]) {
        if (!inc[e->to] &&
            e->flow < e->cap) {
            q.push(e->to);
            inc[e->to] = e;
        }
    }
}

// break if no s-t path found
if (!inc[t]) break;
```

- simple BFS in residual network
- saves edges to reconstruct the path
- also use incoming edges to track visited nodes
- by augmenting shortest paths we accidentally implemented Edmonds-Karp

$$O(nm^2)$$

see a bug?

Ford-Fulkerson Implementation

```
// find path bottleneck
int aug = INF;
for (int v = t; v != s; v = inc[v]->from)
    aug = min(aug, inc[v]->cap - inc[v]->flow);
```

Ford-Fulkerson Implementation

```
// find path bottleneck
int aug = INF;
for (int v = t; v != s; v = inc[v]->from)
    aug = min(aug, inc[v]->cap - inc[v]->flow);
```

- follow edges back from t to s
- we must do this 2 times

Ford-Fulkerson Implementation

```
// find path bottleneck
int aug = INF;
for (int v = t; v != s; v = inc[v]->from)
    aug = min(aug, inc[v]->cap - inc[v]->flow);

// augment path
flow += aug;
for (int v = t; v != s; v = inc[v]->from) {
    inc[v]->flow += aug;
    inc[v]->twin->flow -= aug;
}
```

- follow edges back from t to s
- we must do this 2 times

Ford-Fulkerson Implementation

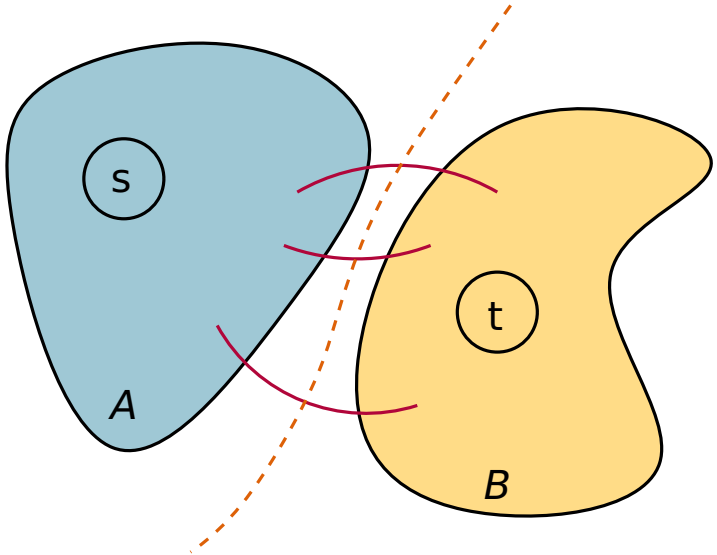
```
// find path bottleneck
int aug = INF;
for (int v = t; v != s; v = inc[v]->from)
    aug = min(aug, inc[v]->cap - inc[v]->flow);

// augment path
flow += aug;
for (int v = t; v != s; v = inc[v]->from) {
    inc[v]->flow += aug;
    inc[v]->twin->flow -= aug;
}
```

- follow edges back from t to s
- we must do this 2 times
- code works the same for directed flow network!!

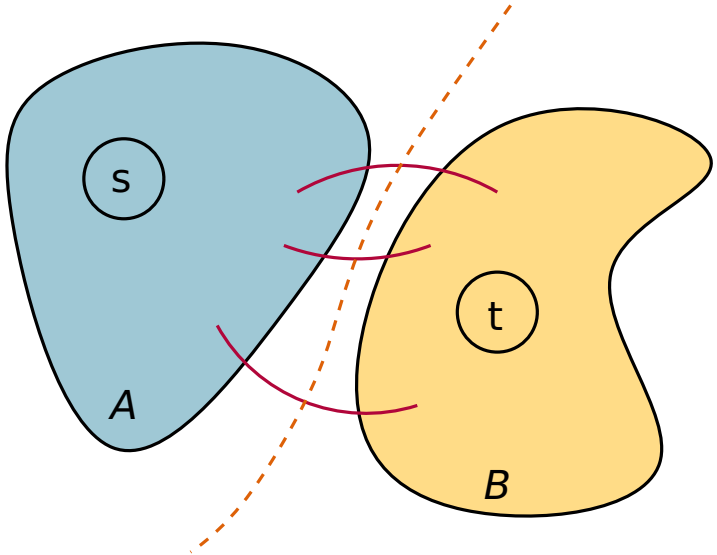
MinCut

Problem extract edges in MinCut from given MaxFlow



MinCut

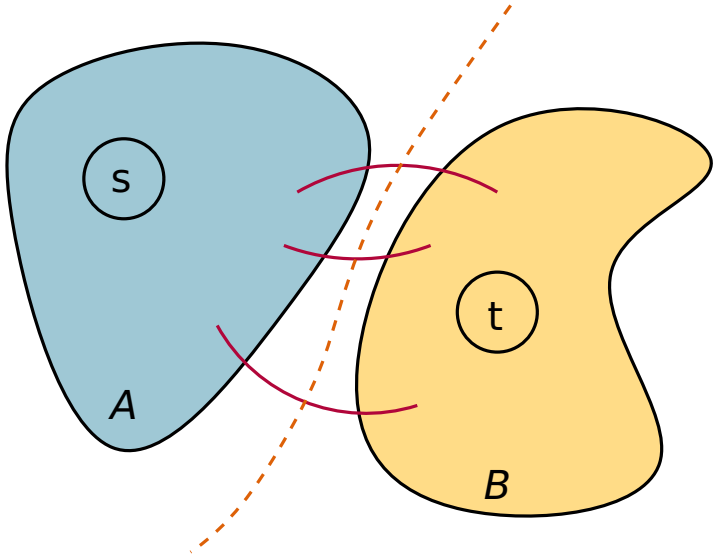
Problem extract edges in MinCut from given MaxFlow



- reachability from s in residual network partitions nodes

MinCut

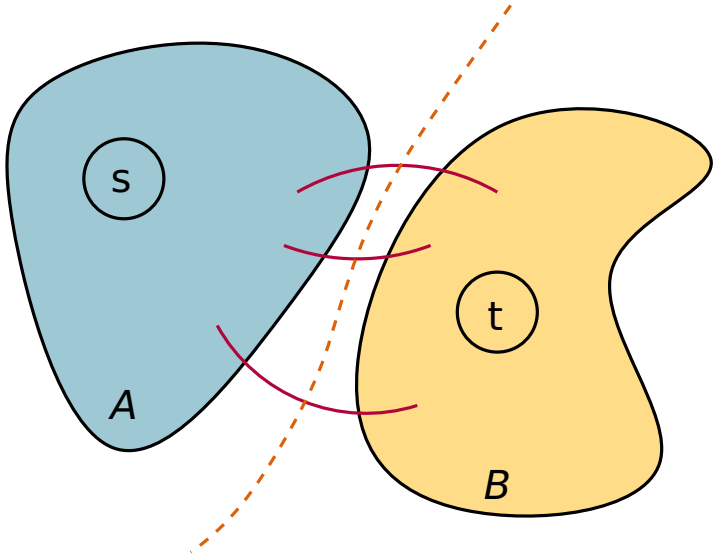
Problem extract edges in MinCut from given MaxFlow



- reachability from s in residual network partitions nodes
- t is in other partition than s

MinCut

Problem extract edges in MinCut from given MaxFlow

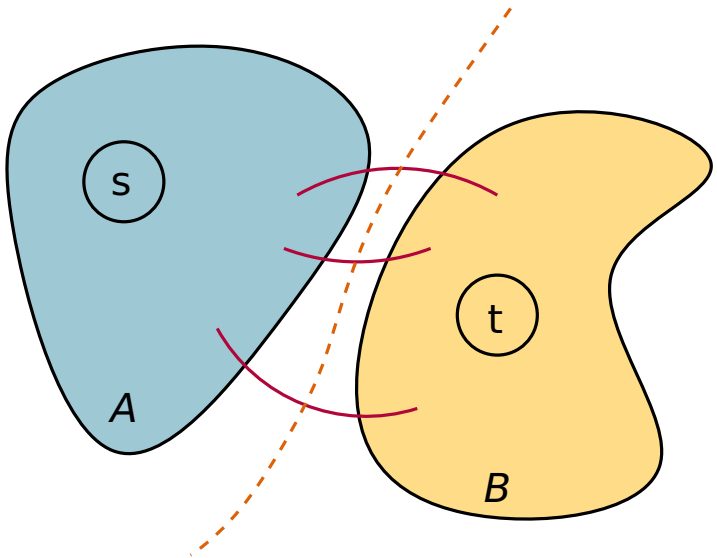


- reachability from s in residual network partitions nodes
- t is in other partition than s

  $\rightarrow$ otherwise flow not maximal

MinCut

Problem extract edges in MinCut from given MaxFlow

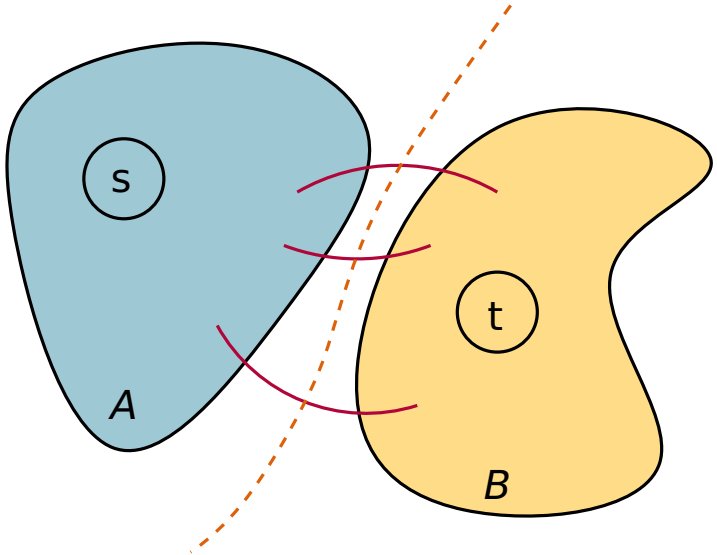


- reachability from s in residual network partitions nodes
- t is in other partition than s

  $\rightarrow$ otherwise flow not maximal
- there are no edges from partition A to B in residual network (they're fully used)

MinCut

Problem extract edges in MinCut from given MaxFlow



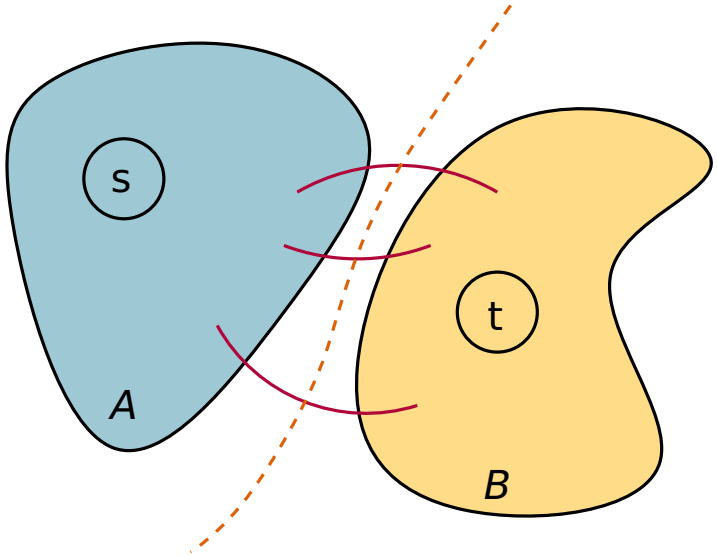
- reachability from s in residual network partitions nodes
- t is in other partition than s

  → otherwise flow not maximal
- there are no edges from partition A to B in residual network (they're fully used)

  → by choice of partitioning

MinCut

Problem extract edges in MinCut from given MaxFlow



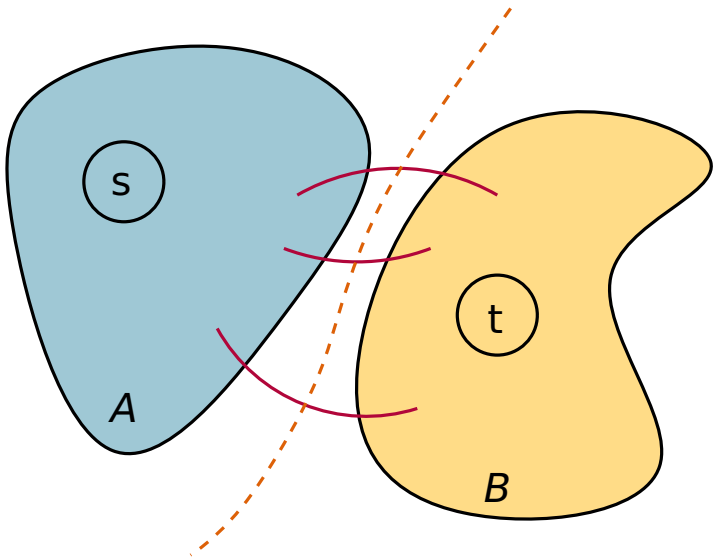
- reachability from s in residual network partitions nodes
- t is in other partition than s

  → otherwise flow not maximal
- there are no edges from partition A to B in residual network (they're fully used)

  → by choice of partitioning
- all flow from s to t must pass through this cut

MinCut

Problem extract edges in MinCut from given MaxFlow



- reachability from s in residual network partitions nodes
- t is in other partition than s

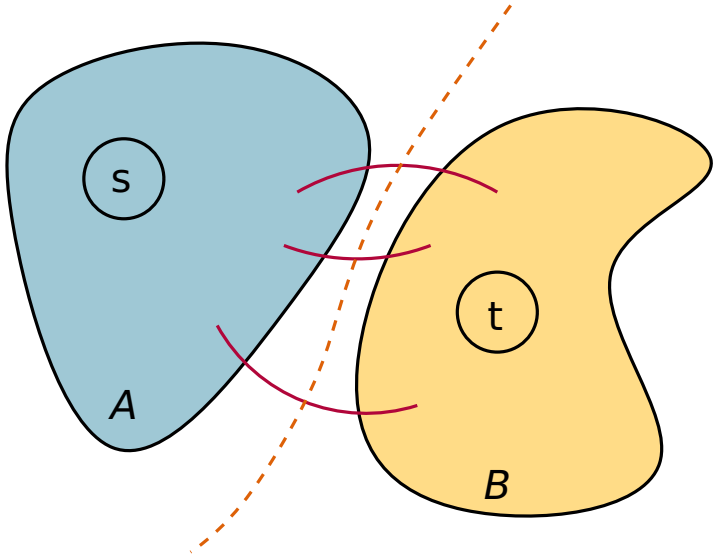
  → otherwise flow not maximal
- there are no edges from partition A to B in residual network (they're fully used)

  → by choice of partitioning
- all flow from s to t must pass through this cut

  → by conservation of flow

MinCut

Problem extract edges in MinCut from given MaxFlow



- reachability from s in residual network partitions nodes
- t is in other partition than s

  → otherwise flow not maximal
- there are no edges from partition A to B in residual network (they're fully used)

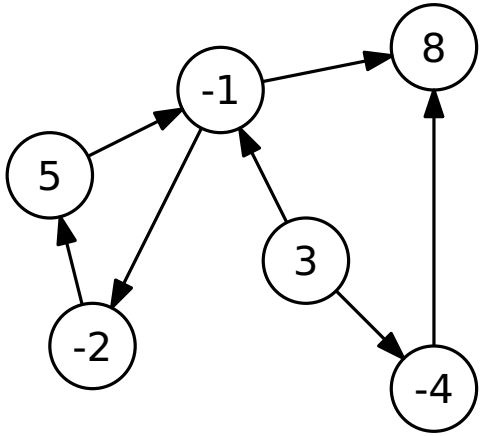
  → by choice of partitioning
- all flow from s to t must pass through this cut

  → by conservation of flow
- I guess we found our MinCut

Applications: Closure

Problem: Given a directed graph with vertex weights, find the maximum weight closure.

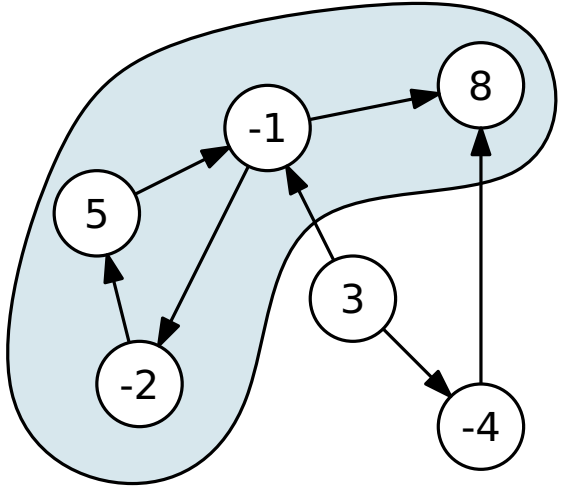
- a **closure** of a directed graph is a set of nodes with no outgoing edges



Applications: Closure

Problem: Given a directed graph with vertex weights, find the maximum weight closure.

- a **closure** of a directed graph is a set of nodes with no outgoing edges

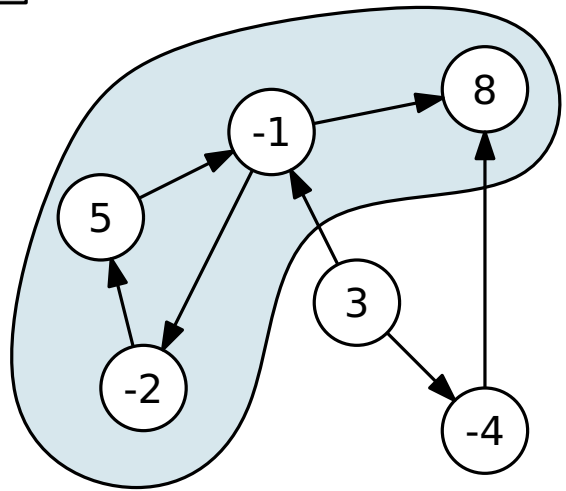


Applications: Closure

Problem: Given a directed graph with vertex weights, find the maximum weight closure.

- a **closure** of a directed graph is a set of nodes with no outgoing edges

min. closure?



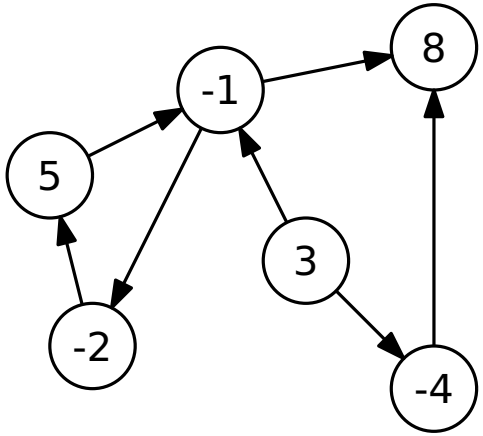
Applications: Closure

Problem: Given a directed graph with vertex weights, find the maximum weight closure.

- a **closure** of a directed graph is a set of nodes with no outgoing edges

Reduction to MinCut:

min. closure?



Applications: Closure

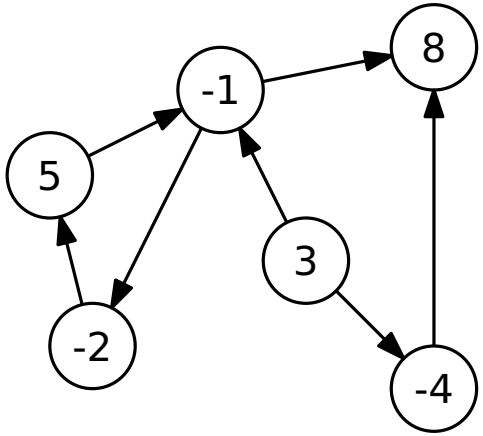
Problem: Given a directed graph with vertex weights, find the maximum weight closure.

- a **closure** of a directed graph is a set of nodes with no outgoing edges

Reduction to MinCut:

- all old edges weight ∞

min. closure?



Applications: Closure

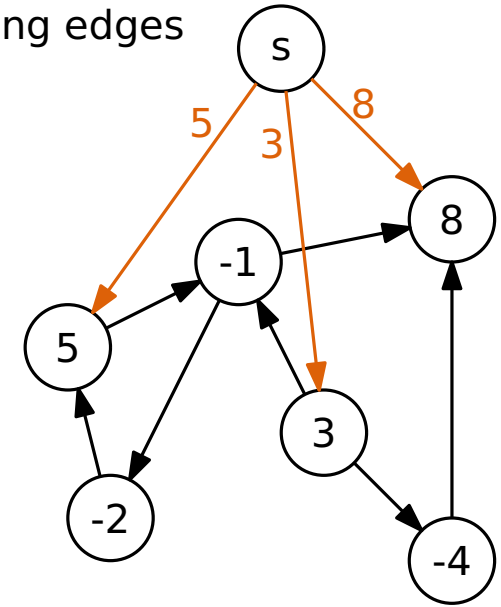
Problem: Given a directed graph with vertex weights, find the maximum weight closure.

- a **closure** of a directed graph is a set of nodes with no outgoing edges

Reduction to MinCut:

- all old edges weight ∞
- new node s connects to vertices with $w_i > 0$ using w_i as edge weight

min. closure?



Applications: Closure

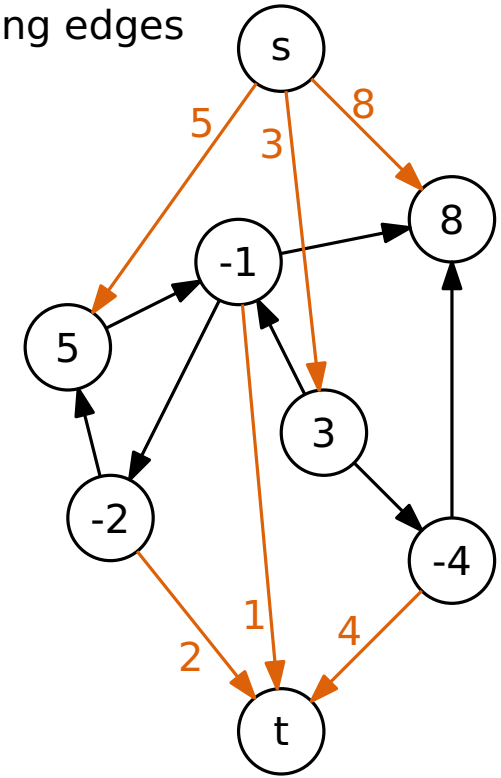
Problem: Given a directed graph with vertex weights, find the maximum weight closure.

- a **closure** of a directed graph is a set of nodes with no outgoing edges

Reduction to MinCut:

- all old edges weight ∞
- new node s connects to vertices with $w_i > 0$ using w_i as edge weight
- new node t connects to vertices with $w_i < 0$ using $-w_i$ as edge weight

min. closure?



Applications: Closure

Problem: Given a directed graph with vertex weights, find the maximum weight closure.

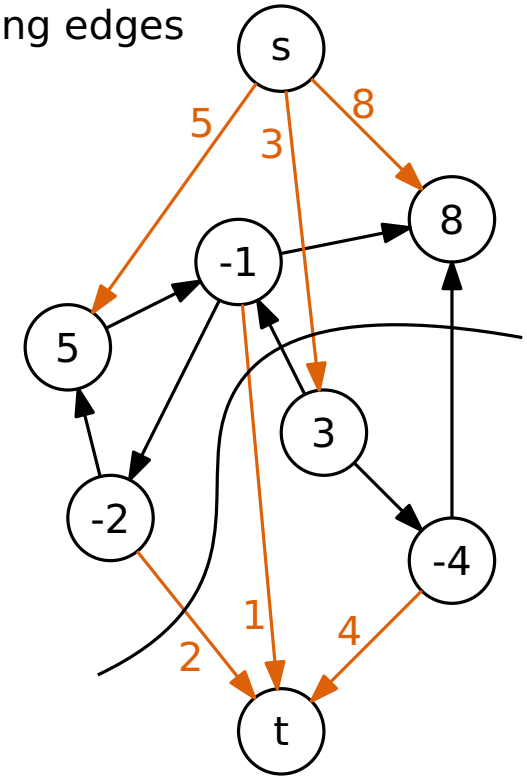
- a **closure** of a directed graph is a set of nodes with no outgoing edges

Reduction to MinCut:

- all old edges weight ∞
- new node s connects to vertices with $w_i > 0$ using w_i as edge weight
- new node t connects to vertices with $w_i < 0$ using $-w_i$ as edge weight

Lemma: The partition of s in min. s-t cut is a max. closure C.

min. closure?



Applications: Closure

Problem: Given a directed graph with vertex weights, find the maximum weight closure.

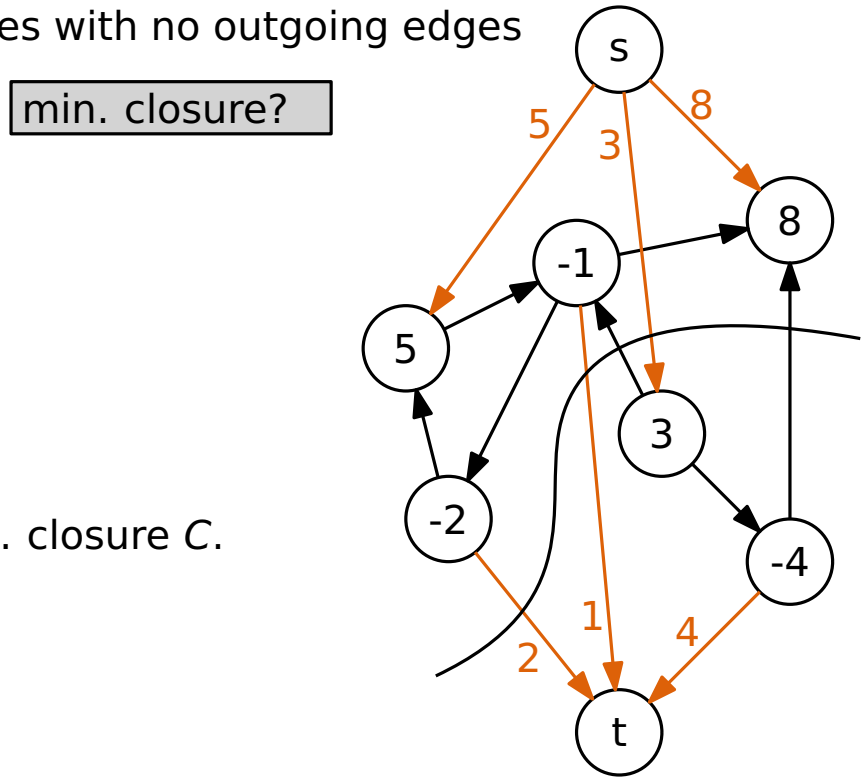
- a **closure** of a directed graph is a set of nodes with no outgoing edges

Reduction to MinCut:

- all old edges weight ∞
- new node s connects to vertices with $w_i > 0$ using w_i as edge weight
- new node t connects to vertices with $w_i < 0$ using $-w_i$ as edge weight

Lemma: The partition of s in min. s-t cut is a max. closure C.

Why is C a valid closure?



Applications: Closure

Problem: Given a directed graph with vertex weights, find the maximum weight closure.

- a **closure** of a directed graph is a set of nodes with no outgoing edges

Reduction to MinCut:

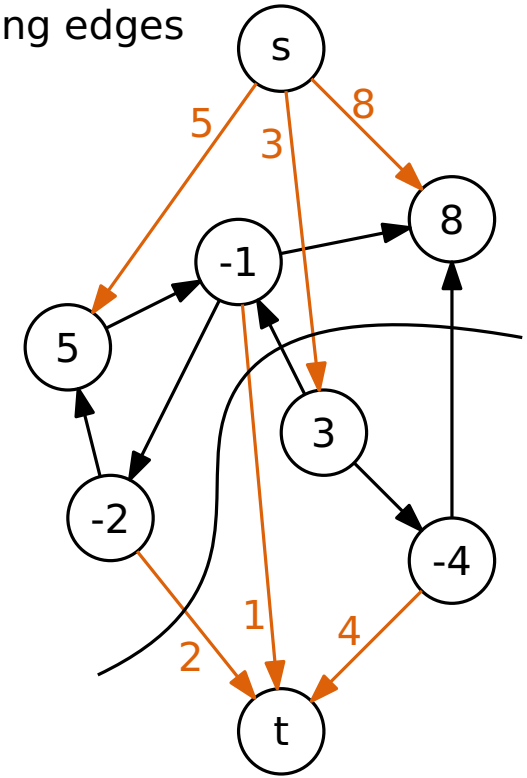
- all old edges weight ∞
- new node s connects to vertices with $w_i > 0$ using w_i as edge weight
- new node t connects to vertices with $w_i < 0$ using $-w_i$ as edge weight

Lemma: The partition of s in min. s-t cut is a max. closure C.

Why is C a valid closure?

Why C's weight maximized?

min. closure?



Applications: Closure

Problem: Given a directed graph with vertex weights, find the maximum weight closure.

- a **closure** of a directed graph is a set of nodes with no outgoing edges

Reduction to MinCut:

- all old edges weight ∞
- new node s connects to vertices with $w_i > 0$ using w_i as edge weight
- new node t connects to vertices with $w_i < 0$ using $-w_i$ as edge weight

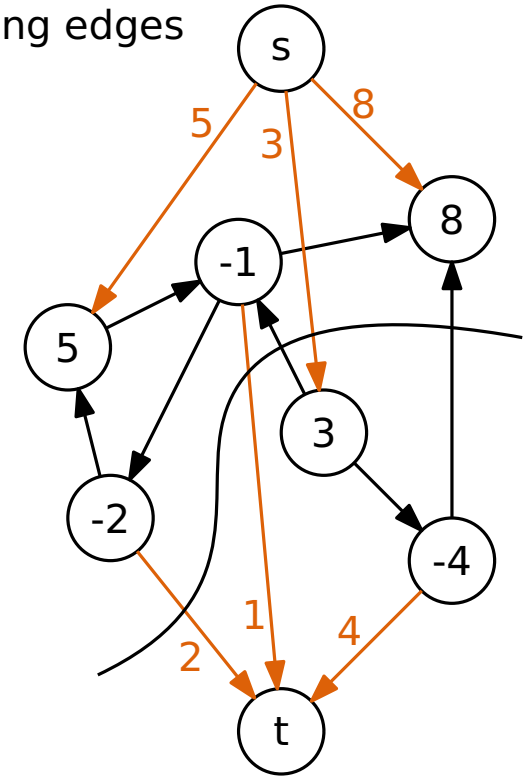
Lemma: The partition of s in min. s-t cut is a max. closure C.

Why is C a valid closure?

Why C's weight maximized?

value of cut =
sum of positive vertices $\notin C$
+ sum of negative vertices $\in C$

min. closure?

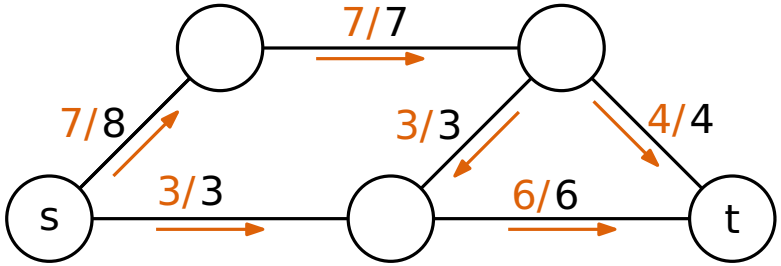


Flow Decomposition

- each flow can be decomposed in a collection of s-t paths

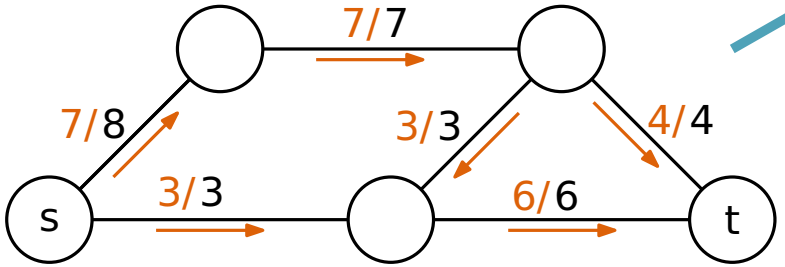
Flow Decomposition

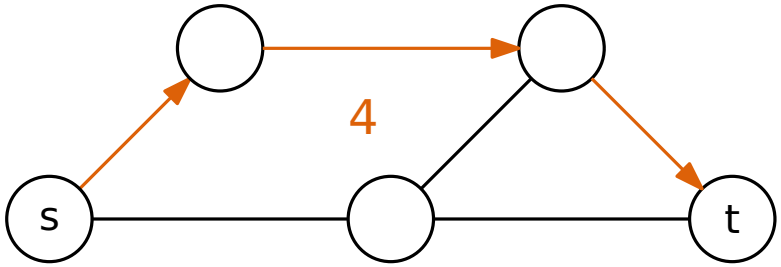
- each flow can be decomposed in a collection of s-t paths



Flow Decomposition

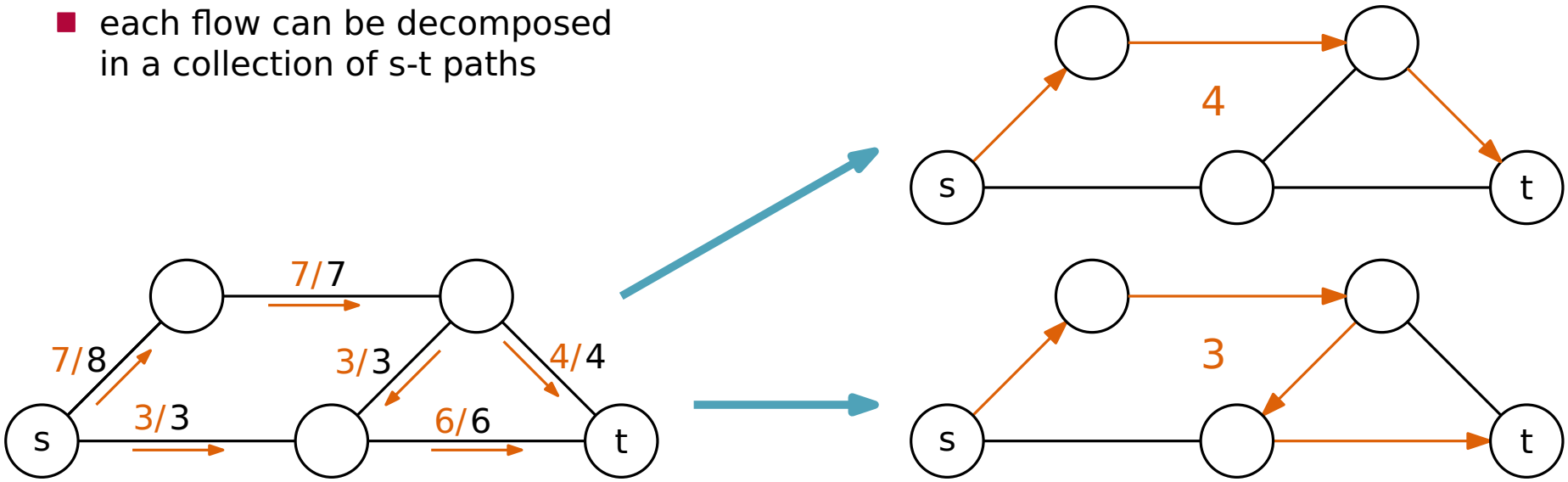
- each flow can be decomposed in a collection of s-t paths





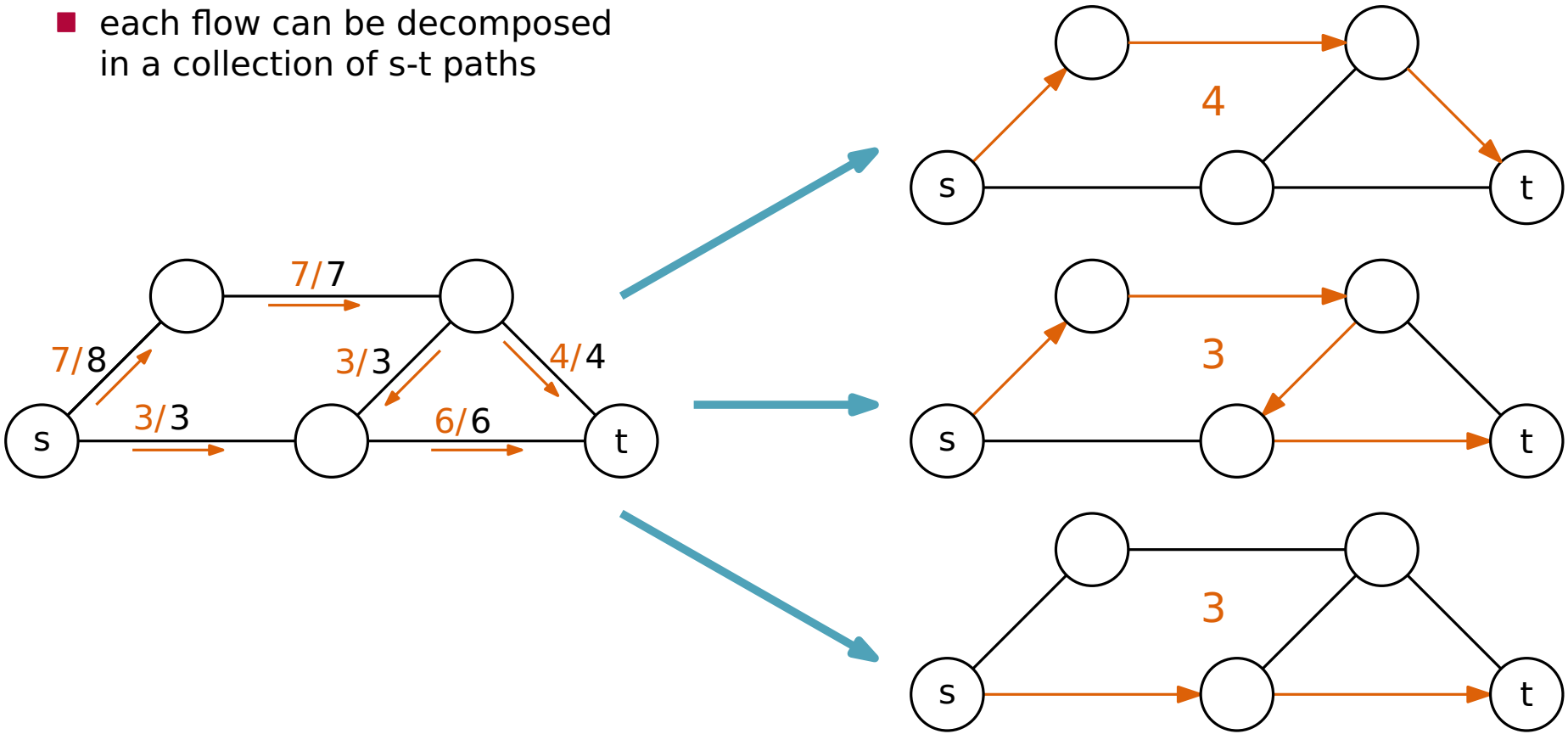
Flow Decomposition

- each flow can be decomposed in a collection of s-t paths



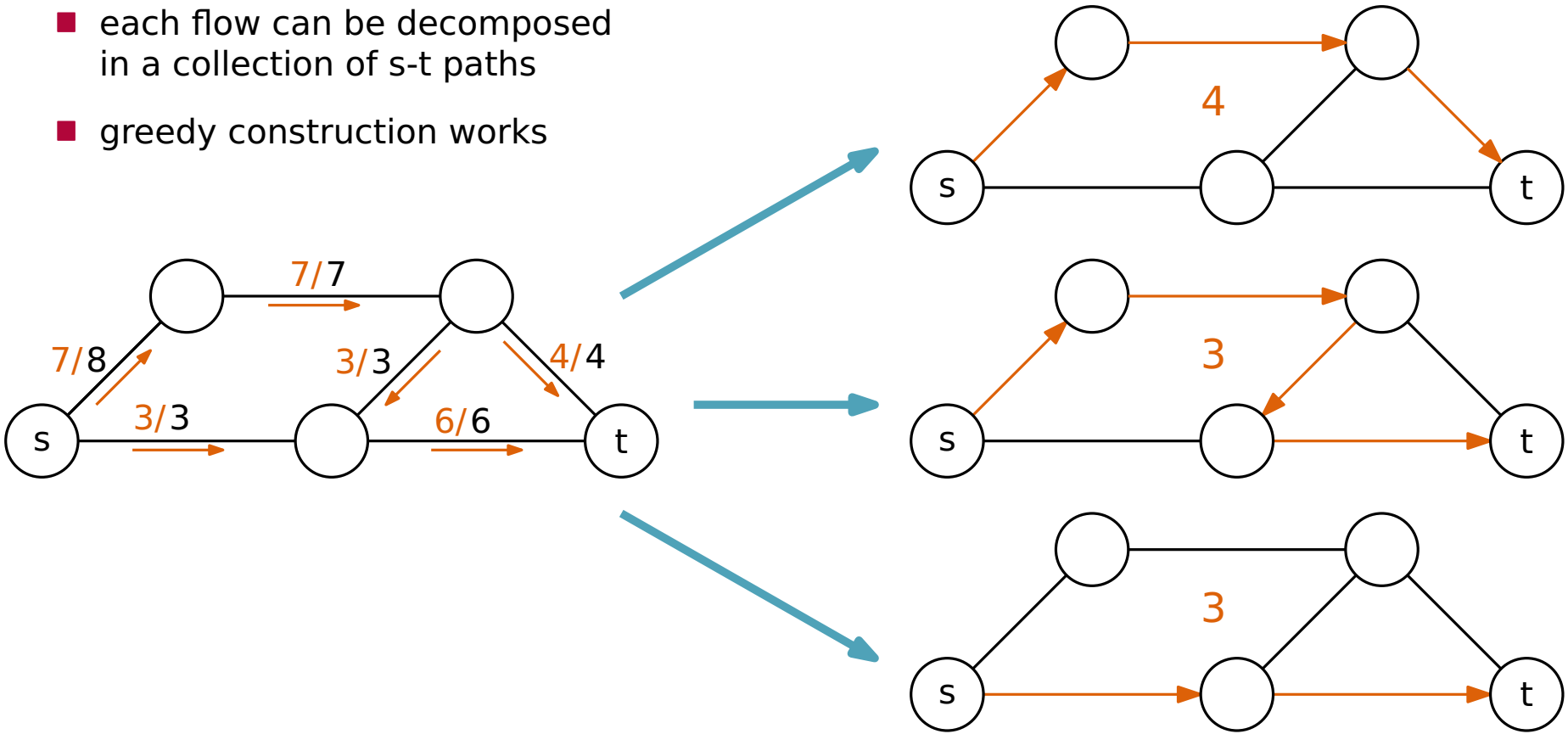
Flow Decomposition

- each flow can be decomposed in a collection of s-t paths



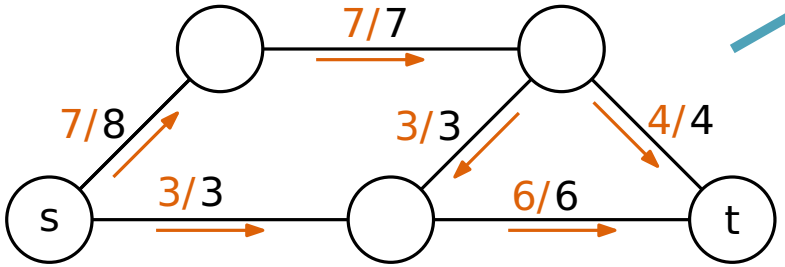
Flow Decomposition

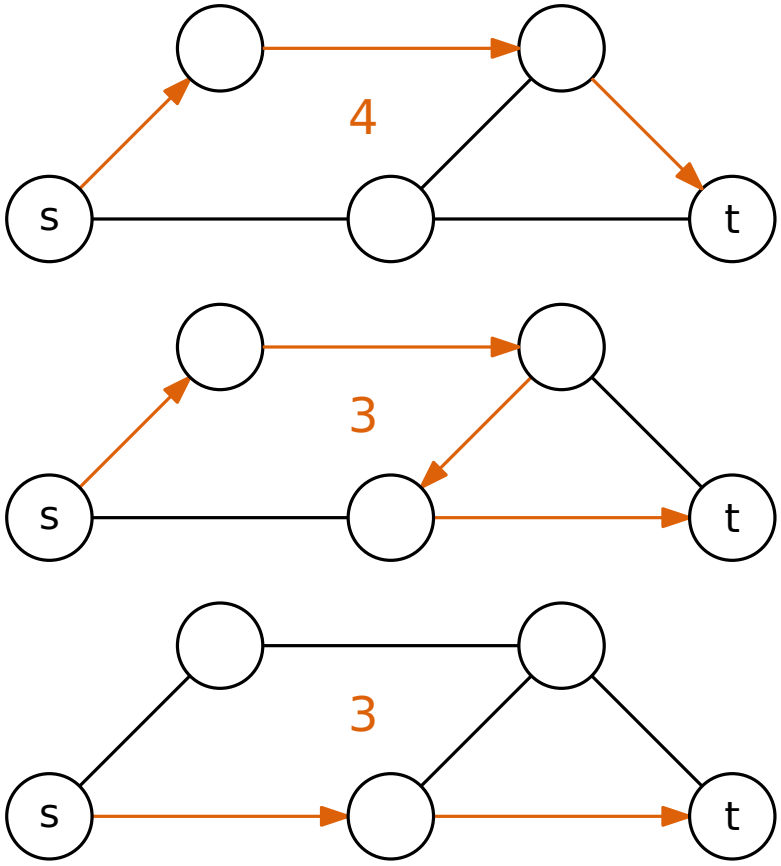
- each flow can be decomposed in a collection of s-t paths
- greedy construction works



Flow Decomposition

- each flow can be decomposed in a collection of s-t paths
- greedy construction works





- idea: repeatedly find some s-t path and remove it from the flow

Flow Decomposition

Flow Decomposition Algorithm:

- **while** there is s-t flow **do**
  - find s-t path P with flow
  - remove flow from P

Flow Decomposition

- running time $O(m^2)$

Flow Decomposition Algorithm:

- $\leq m$ times **while** there is s-t flow **do**
  - $\Theta(m)$ find s-t path P with flow
  - $O(n)$ remove flow from P

Flow Decomposition

- running time $O(m^2)$
- **Observation**: we don't need edges without flow; why look at them m times?

Flow Decomposition Algorithm:

- $\leq m$ times **while** there is s-t flow **do**
  - $\Theta(m)$ find s-t path P with flow
  - $O(n)$ remove flow from P

Flow Decomposition

- running time $O(m^2)$

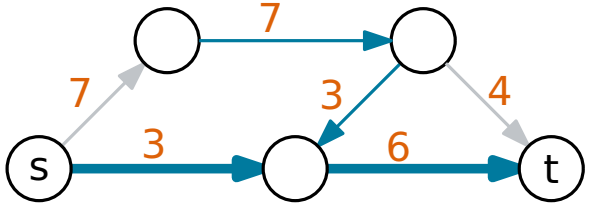
- **Observation**: we don't need edges without flow; why look at them m times?

- for each node, maintain the first outgoing edge with positive flow

- can be done with an index into its ajacency list

Flow Decomposition Algorithm:

- $\leq m$ times — **while** there is s-t flow **do**
  - $\Theta(m)$ — find s-t path P with flow
  - $O(n)$ — remove flow from P

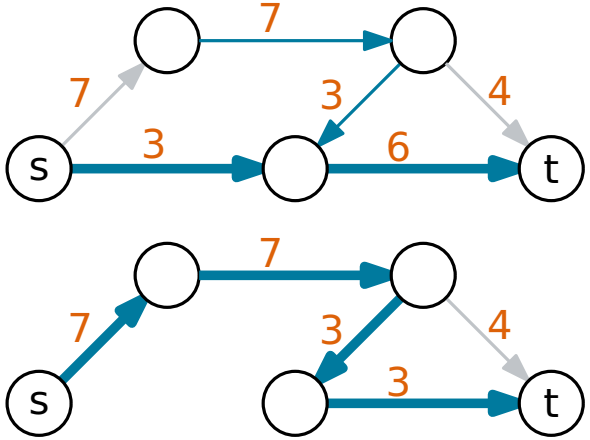


Flow Decomposition

- running time $O(m^2)$
- **Observation**: we don't need edges without flow; why look at them m times?
- for each node, maintain the first outgoing edge with positive flow
- can be done with an index into its ajacency list

Flow Decomposition Algorithm:

- $\leq m$ times **while** there is s-t flow **do**
  - $\Theta(m)$ find s-t path P with flow
  - $O(n)$ remove flow from P

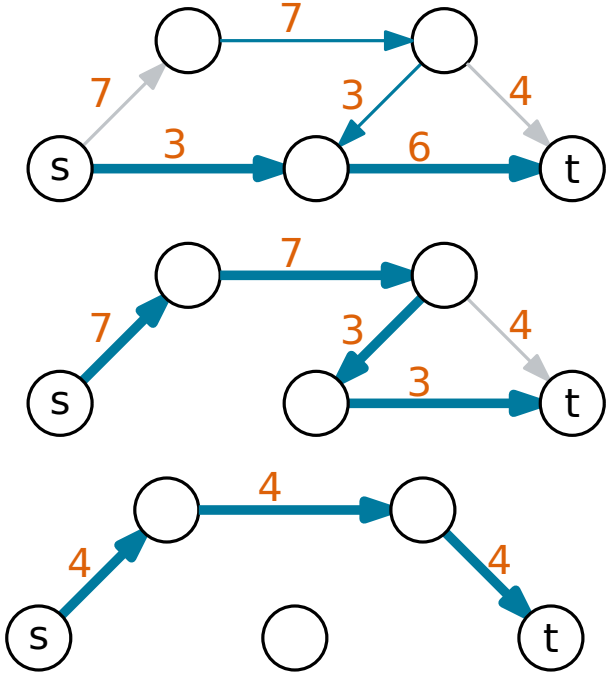


Flow Decomposition

- running time $O(m^2)$
- **Observation**: we don't need edges without flow; why look at them m times?
- for each node, maintain the first outgoing edge with positive flow
- can be done with an index into its ajacency list

Flow Decomposition Algorithm:

- $\leq m$ times — **while** there is s-t flow **do**
  - $\Theta(m)$ — find s-t path P with flow
  - $O(n)$ — remove flow from P

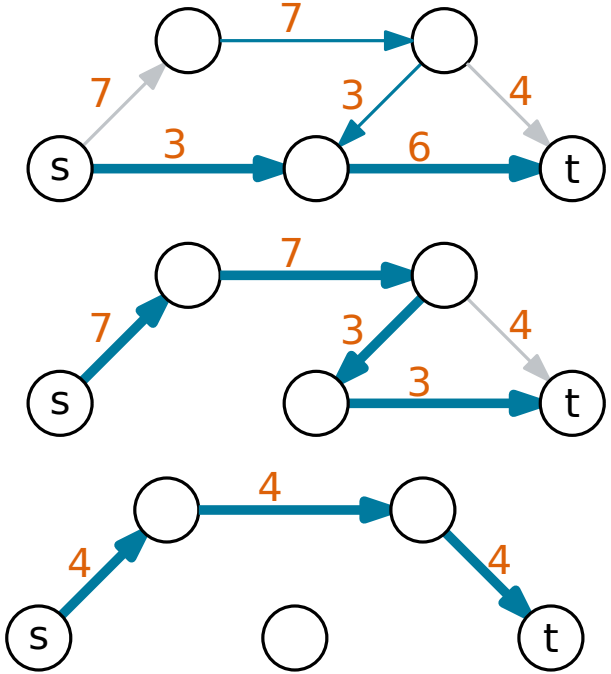


Flow Decomposition

- running time $O(m^2)$
- **Observation**: we don't need edges without flow; why look at them m times?
- for each node, maintain the first outgoing edge with positive flow
- can be done with an index into its ajacency list
- maintaining the next edge for each node takes $O(m)$ time over all graph traversals combined

Flow Decomposition Algorithm:

- $\leq m$ times **while** there is s-t flow **do**
  - $\Theta(m)$ find s-t path P with flow
  - $O(n)$ remove flow from P



Flow Decomposition

- running time $O(m^2)$
- **Observation**: we don't need edges without flow; why look at them m times?
- for each node, maintain the first outgoing edge with positive flow
- can be done with an index into its ajacency list
- maintaining the next edge for each node takes $O(m)$ time over all graph traversals combined
- each graph traversal (e.g. DFS) considers only one edge per node $\rightarrow O(n)$ instead of $O(m)$

$\leq m$ times

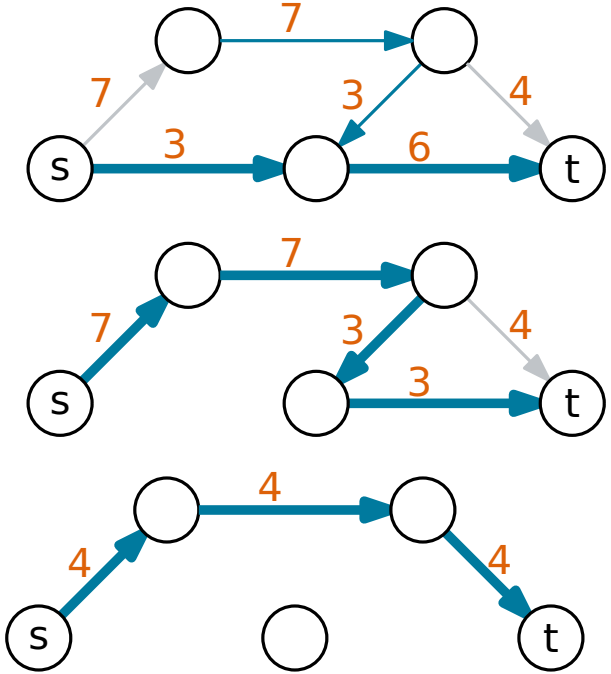
$\Theta(n) \leftarrow \Theta(m)$

* amortized

$O(n)$

Flow Decomposition Algorithm:

- **while** there is s-t flow **do**
  - find s-t path P with flow
  - remove flow from P

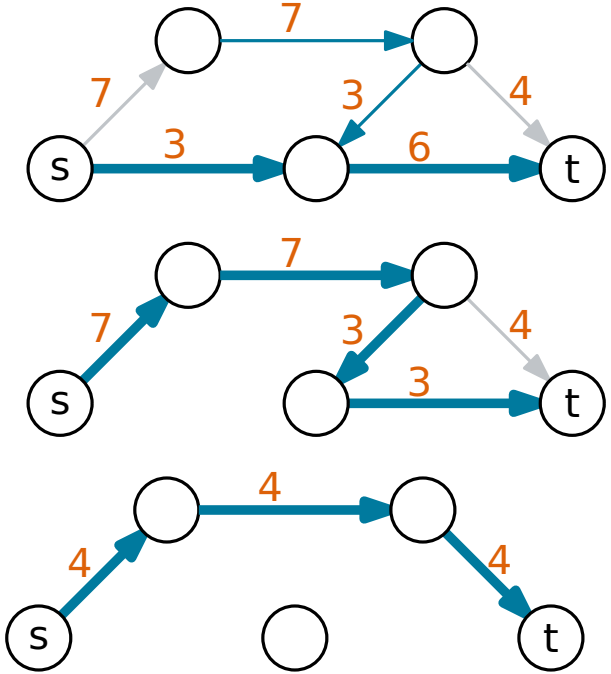


Flow Decomposition

- running time $O(m^2)$
- **Observation**: we don't need edges without flow; why look at them m times?
- for each node, maintain the first outgoing edge with positive flow
- can be done with an index into its ajacency list
- maintaining the next edge for each node takes $O(m)$ time over all graph traversals combined
- each graph traversal (e.g. DFS) considers only one edge per node $\rightarrow O(n)$ instead of $O(m)$
- running time reduced from $O(m^2)$ to $O(nm)$

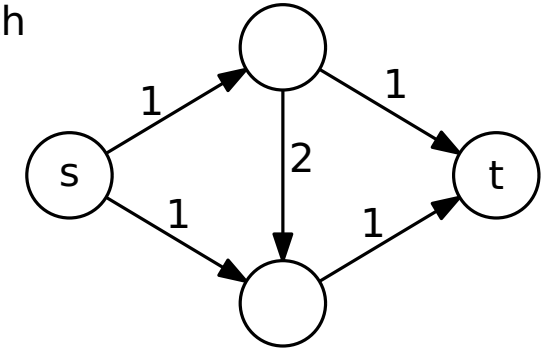
Flow Decomposition Algorithm:

- **while** there is s-t flow **do** — $\leq m$ times
  - find s-t path P with flow — $\Theta(n) \leftarrow \Theta(m)$ * amortized
  - remove flow from P — $O(n)$



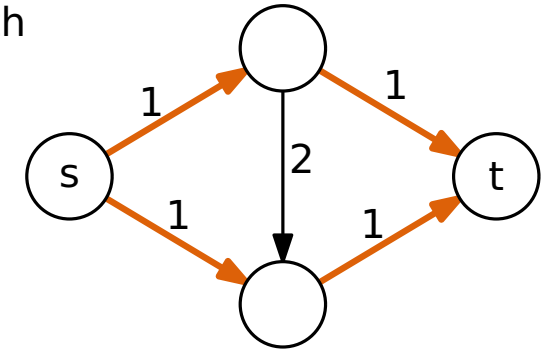
Dinic’s

Blocking Flow: a flow of some network such that every s-t path contains at least one edge which is saturated by this flow



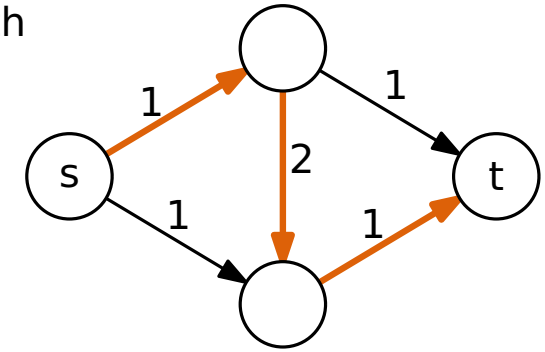
Dinic’s

Blocking Flow: a flow of some network such that every s-t path contains at least one edge which is saturated by this flow



Dinic’s

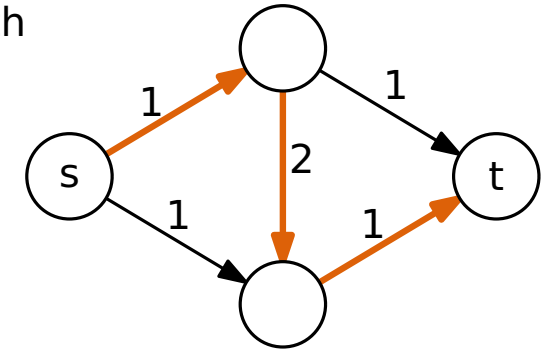
Blocking Flow: a flow of some network such that every s-t path contains at least one edge which is saturated by this flow



Dinic's

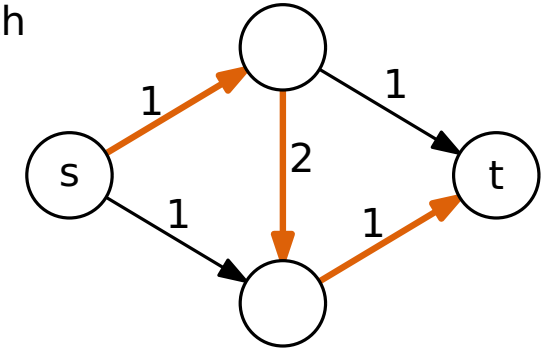
Blocking Flow: a flow of some network such that every s-t path contains at least one edge which is saturated by this flow

Layered Network: Let $d(v)$ be the distance from s to v in residual network. The layered network contains only edges (u, v) of residual network for which $d(u) + 1 = d(v)$

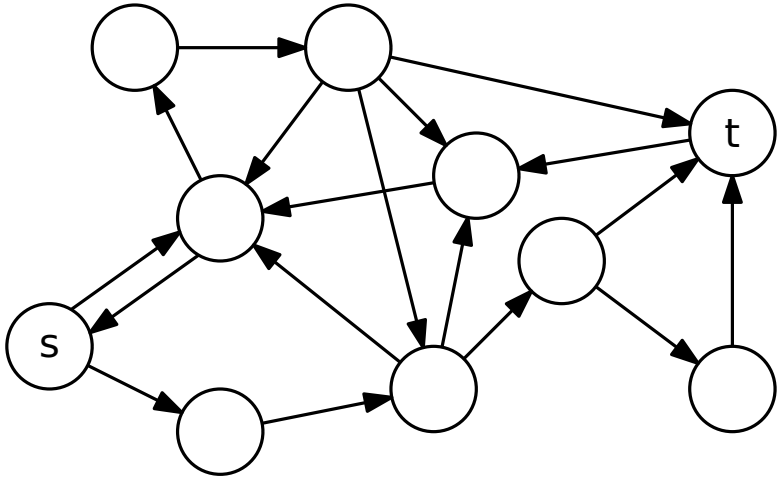


Dinic's

Blocking Flow: a flow of some network such that every s-t path contains at least one edge which is saturated by this flow



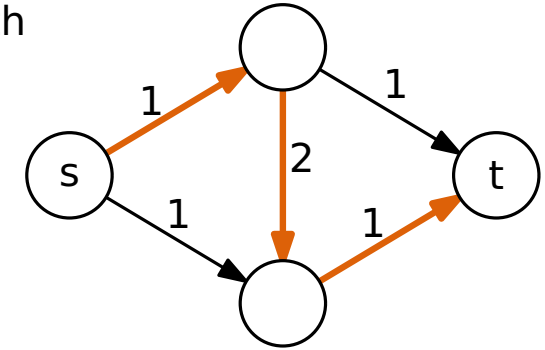
Layered Network: Let $d(v)$ be the distance from s to v in residual network. The layered network contains only edges (u, v) of residual network for which $d(u) + 1 = d(v)$



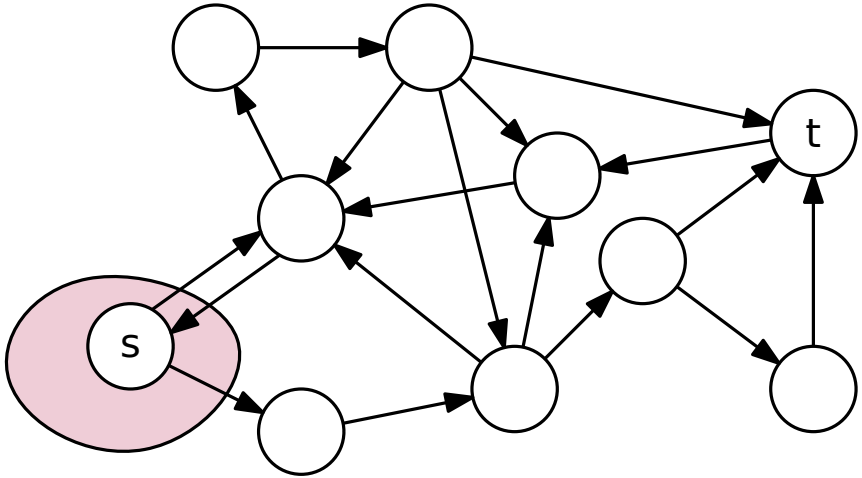
* capacities omitted

Dinic's

Blocking Flow: a flow of some network such that every s-t path contains at least one edge which is saturated by this flow



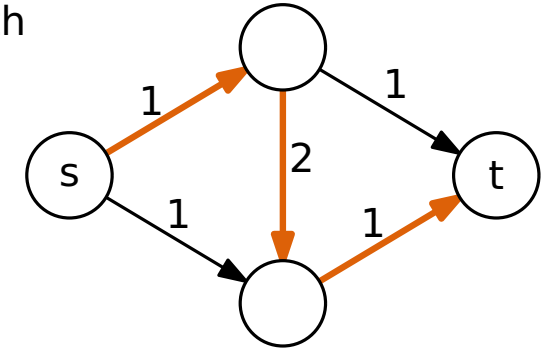
Layered Network: Let $d(v)$ be the distance from s to v in residual network. The layered network contains only edges (u, v) of residual network for which $d(u) + 1 = d(v)$



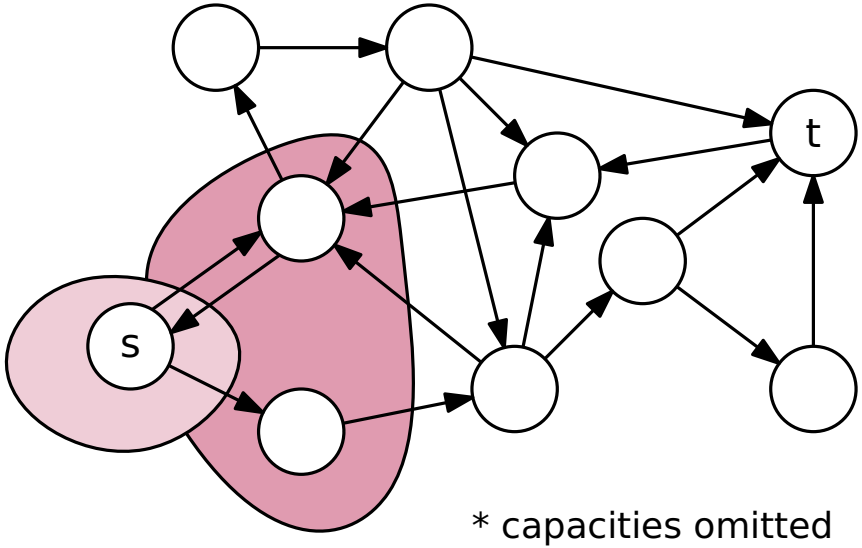
* capacities omitted

Dinic's

Blocking Flow: a flow of some network such that every s-t path contains at least one edge which is saturated by this flow



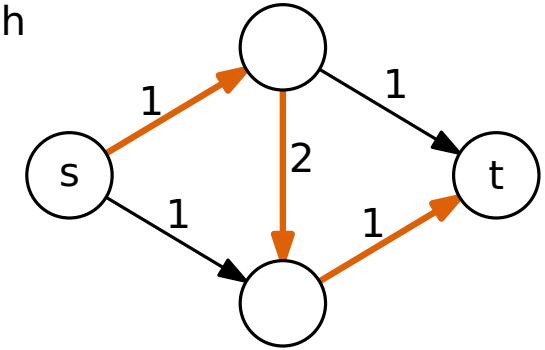
Layered Network: Let $d(v)$ be the distance from s to v in residual network. The layered network contains only edges (u, v) of residual network for which $d(u) + 1 = d(v)$



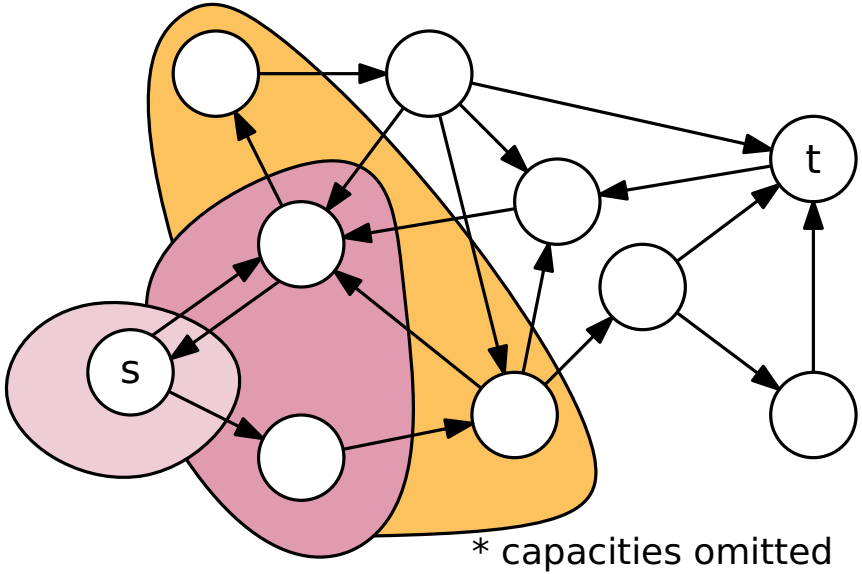
* capacities omitted

Dinic's

Blocking Flow: a flow of some network such that every s-t path contains at least one edge which is saturated by this flow



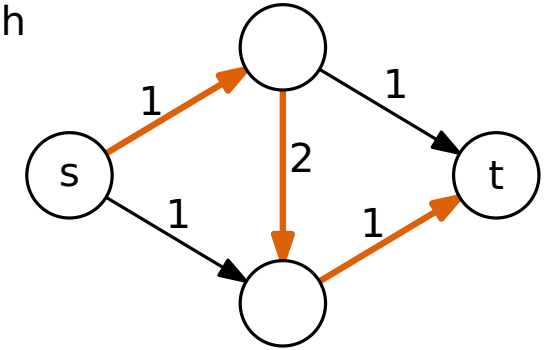
Layered Network: Let $d(v)$ be the distance from s to v in residual network. The layered network contains only edges (u, v) of residual network for which $d(u) + 1 = d(v)$



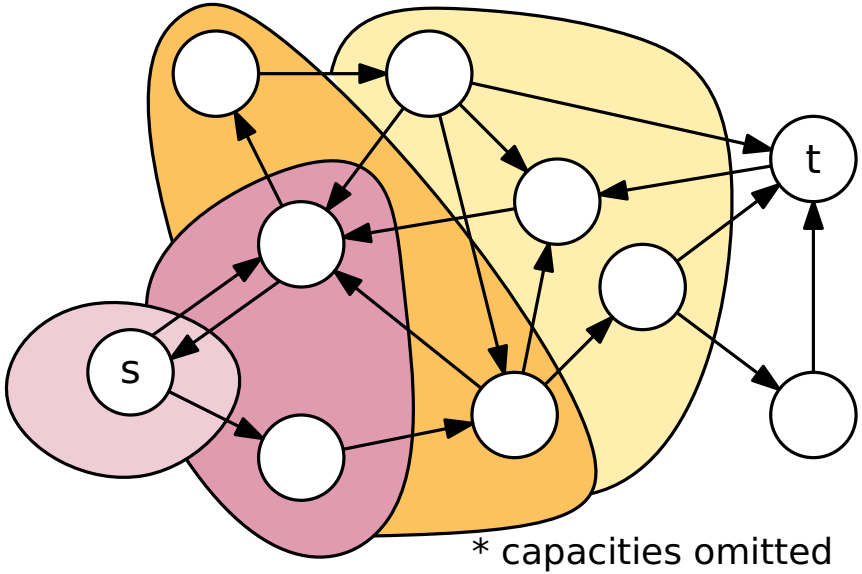
* capacities omitted

Dinic's

Blocking Flow: a flow of some network such that every s-t path contains at least one edge which is saturated by this flow



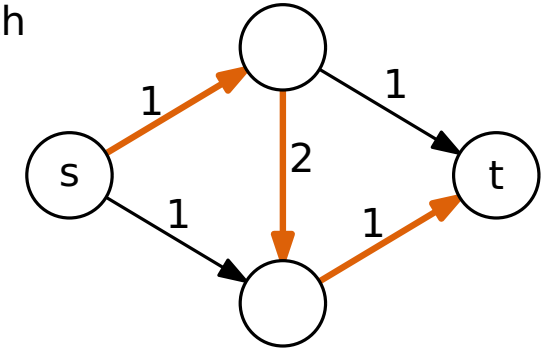
Layered Network: Let $d(v)$ be the distance from s to v in residual network. The layered network contains only edges (u, v) of residual network for which $d(u) + 1 = d(v)$



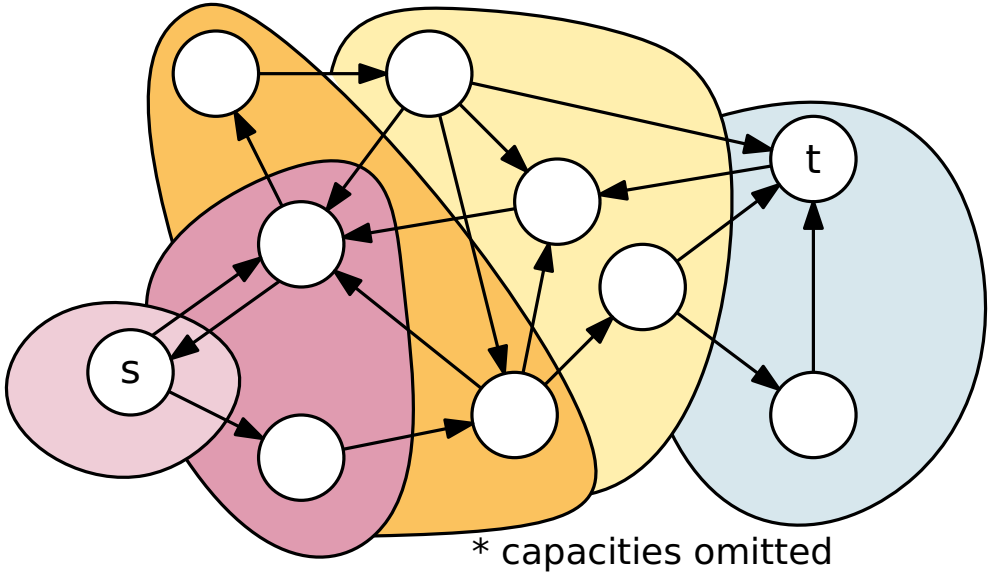
* capacities omitted

Dinic's

Blocking Flow: a flow of some network such that every s-t path contains at least one edge which is saturated by this flow



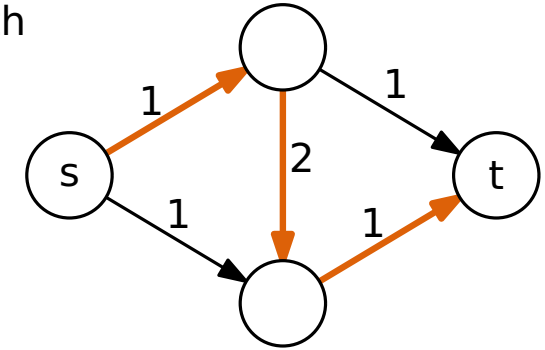
Layered Network: Let $d(v)$ be the distance from s to v in residual network. The layered network contains only edges (u, v) of residual network for which $d(u) + 1 = d(v)$



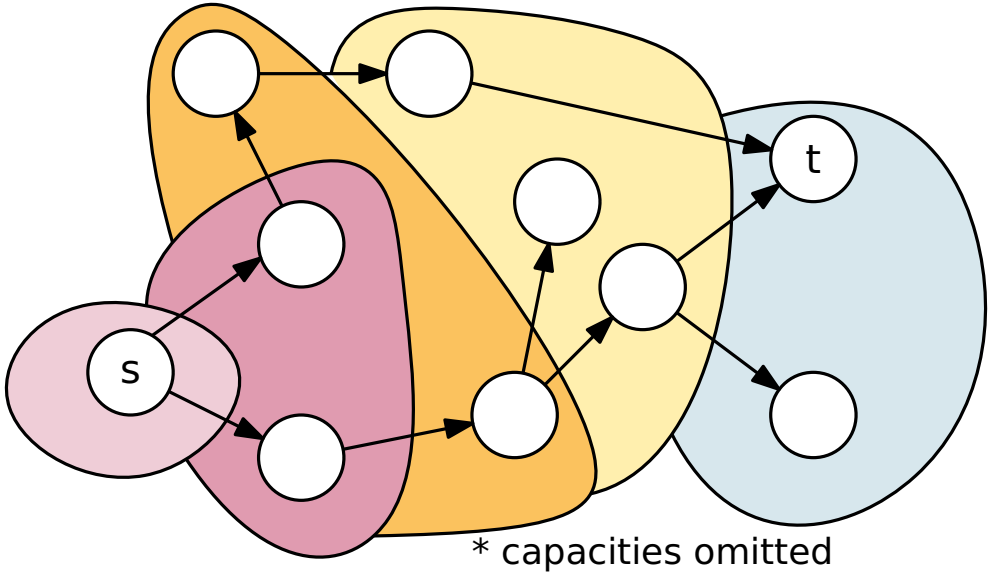
* capacities omitted

Dinic's

Blocking Flow: a flow of some network such that every s-t path contains at least one edge which is saturated by this flow



Layered Network: Let $d(v)$ be the distance from s to v in residual network. The layered network contains only edges (u, v) of residual network for which $d(u) + 1 = d(v)$



* capacities omitted

Dinic's

- same approach as Ford-Fulkerson

Ford-Fulkerson MaxFlow Algorithm

```
while (true) {
    // find augmenting path
    // break if no s-t path found
    // find path bottleneck
    // augment path
}
```

Dinic's

- same approach as Ford-Fulkerson
- instead of augmenting one path, we augment an arbitrary blocking flow in the layered network

Ford-Fulkerson MaxFlow Algorithm

```
while (true) {
    // find augmenting path
    // break if no s-t path found
    // find path bottleneck
    // augment path
}
```

Dinic's MaxFlow Algorithm

```
while (true) {
    // create layered network L
    // break if no s-t path found
    while (/*s-t path in L*/)
        // augment path
}
```

Dinic's

- same approach as Ford-Fulkerson
- instead of augmenting one path, we augment an arbitrary blocking flow in the layered network

Ford-Fulkerson MaxFlow Algorithm

```
while (true) {
    // find augmenting path
    // break if no s-t path found
    // find path bottleneck
    // augment path
}
```

Dinic's MaxFlow Algorithm

	`while (true) {`
$\Theta(m)$	`  // create layered network L`
$\Theta(1)$	`  // break if no s-t path found`
$\Theta(m)$	`  while (/*s-t path in L*/)`
$O(n)$	`  // augment path`
	`}`

Dinic's

- same approach as Ford-Fulkerson
- instead of augmenting one path, we augment an arbitrary blocking flow in the layered network

Ford-Fulkerson MaxFlow Algorithm

```
while (true) {
    // find augmenting path
    // break if no s-t path found
    // find path bottleneck
    // augment path
}
```

Dinic's MaxFlow Algorithm

	`while (true) {`
$\Theta(m)$	`  // create layered network L`
$\Theta(1)$	`  // break if no s-t path found`
$\leq m$ times $\Theta(m)$	`  while (/*s-t path in L*/)`
$O(n)$	`  // augment path`
	`}`

Dinic's

- same approach as Ford-Fulkerson
- instead of augmenting one path, we augment an arbitrary blocking flow in the layered network
- **Lemma**: each time we augment blocking flow in layered network $d(t)$ increases

Ford-Fulkerson MaxFlow Algorithm

```
while (true) {
    // find augmenting path
    // break if no s-t path found
    // find path bottleneck
    // augment path
}
```

Dinic's MaxFlow Algorithm

	`while (true) {`
$\Theta(m)$	`  // create layered network L`
$\Theta(1)$	`  // break if no s-t path found`
$\leq m$ times $\Theta(m)$	`  while (/*s-t path in L*/)`
$O(n)$	`  // augment path`
	`}`

Dinic's

- same approach as Ford-Fulkerson
- instead of augmenting one path, we augment an arbitrary blocking flow in the layered network
- **Lemma**: each time we augment blocking flow in layered network $d(t)$ increases

Ford-Fulkerson MaxFlow Algorithm

```
while (true) {
    // find augmenting path
    // break if no s-t path found
    // find path bottleneck
    // augment path
}
```

Dinic's MaxFlow Algorithm

	`while (true) {`
$\leq n$ times	`    // create layered network L`
$\Theta(m)$	`    // break if no s-t path found`
$\Theta(1)$	`    while (/*s-t path in L*/)`
$\leq m$ times $\Theta(m)$	`        // augment path`
$O(n)$	`}`

Dinic's

- same approach as Ford-Fulkerson
- instead of augmenting one path, we augment an arbitrary blocking flow in the layered network
- **Lemma**: each time we augment blocking flow in layered network $d(t)$ increases
- can employ same 'ptr to next available edge' approach as in decomposition

Ford-Fulkerson MaxFlow Algorithm

```
while (true) {
    // find augmenting path
    // break if no s-t path found
    // find path bottleneck
    // augment path
}
```

Dinic's MaxFlow Algorithm

$\leq n$ times

```
while (true) {
    // create layered network L
    // break if no s-t path found
```

$\Theta(m)$

$\Theta(1)$

$\leq m$ times

```
    while (/*s-t path in L*/)
        // augment path
```

$\Theta(m)$

$O(n)$

```
}
```

Dinic's

- same approach as Ford-Fulkerson
- instead of augmenting one path, we augment an arbitrary blocking flow in the layered network
- **Lemma**: each time we augment blocking flow in layered network $d(t)$ increases
- can employ same 'ptr to next available edge' approach as in decomposition

Ford-Fulkerson MaxFlow Algorithm

```
while (true) {
    // find augmenting path
    // break if no s-t path found
    // find path bottleneck
    // augment path
}
```

Dinic's MaxFlow Algorithm

$\leq n$ times

```
while (true) {
    // create layered network L
    // break if no s-t path found
    while (/*s-t path in L*/)
        // augment path
}
```

$\Theta(m)$

$\Theta(1)$

$\leq m$ times

$\Theta(n)$ * amortized

$O(n)$

Dinic's

- same approach as Ford-Fulkerson
- instead of augmenting one path, we augment an arbitrary blocking flow in the layered network
- **Lemma**: each time we augment blocking flow in layered network $d(t)$ increases
- can employ same 'ptr to next available edge' approach as in decomposition
- running time: $O(mn^2)$
- $O(m\sqrt{n})$ in unit networks
  e.g. max. bipartite matching

  $\rightarrow$ Hopcroft-Karp

Ford-Fulkerson MaxFlow Algorithm

```
while (true) {
    // find augmenting path
    // break if no s-t path found
    // find path bottleneck
    // augment path
}
```

Dinic's MaxFlow Algorithm

```
while (true) {
    // create layered network L          ≤ n times
    // break if no s-t path found        Θ(m)
    while (/*s-t path in L*/)            Θ(1)
        // augment path                  ≤ m times: → Θ(n) * amortized; O(n)
}
```

Dinic's Implementation

```
// create layered network
vector<int> dist(n, INF);
dist[s] = 0;
queue<int> q{{s}};
while (!q.empty()) {
    auto v = q.front(); q.pop();
    for (auto e : adj[v]) {
        if (dist[e->to] == INF &&
            e->flow < e->cap) {
            q.push(e->to);
            dist[e->to] = dist[v] + 1;
        }
    }
}
```

Dinic's Implementation

```
// create layered network
vector<int> dist(n, INF);
dist[s] = 0;
queue<int> q{{s}};
while (!q.empty()) {
    auto v = q.front(); q.pop();
    for (auto e : adj[v]) {
        if (dist[e->to] == INF &&
            e->flow < e->cap) {
            q.push(e->to);
            dist[e->to] = dist[v] + 1;
        }
    }
}
```

- don't actually *create* layered network!
- compute distances
- when traversing graph, ignore all edges that are not in layered network

Dinic's Implementation

```
// create layered network
vector<int> dist(n, INF);
dist[s] = 0;
queue<int> q{{s}};
while (!q.empty()) {
    auto v = q.front(); q.pop();
    for (auto e : adj[v]) {
        if (dist[e->to] == INF &&
            e->flow < e->cap) {
            q.push(e->to);
            dist[e->to] = dist[v] + 1;
        }
    }
}
// break if no s-t path found
if (dist[t]==INF) break;
```

- don't actually *create* layered network!
- compute distances
- when traversing graph, ignore all edges that are not in layered network

Dinic's Implementation

```
// while s-t path in L
    // augment path
vector<int> next(n, 0);
while (true) {
    int aug = dfs(s,INF);
    flow += aug;
    if (aug==0) break;
}
```

* omitted dfs parameters:

`t, adj, next, dist`

Dinic's Implementation

```
// while s-t path in L
    // augment path
vector<int> next(n, 0);
while (true) {
    int aug = dfs(s,INF);
    flow += aug;
    if (aug==0) break;
}
```

- dfs finds any path from s to t in layered network and augments it
- 'next' vector is same idea as in flow decomposition

* omitted dfs parameters:

`t, adj, next, dist`

Dinic's Implementation

```
// while s-t path in L
    // augment path
vector<int> next(n, 0);
while (true) {
    int aug = dfs(s,INF);
    flow += aug;
    if (aug==0) break;
}
```

- dfs finds any path from s to t in layered network and augments it
- 'next' vector is same idea as in flow decomposition

- more concise:

```
int aug;
while (aug = dfs(s,INF)) flow += aug;
```

* omitted dfs parameters:

`t, adj, next, dist`

Dinic's Implementation

- augments some s-t path and returns how much flow reached t

```
int dfs(int v, int aug) {
    if (v == t) return aug;

    return 0;
}
```

Dinic's Implementation

- augments some s-t path and returns how much flow reached t
- v is current node

```
int dfs(int v, int aug) {
    if (v == t) return aug;

    return 0;
}
```

Dinic's Implementation

- augments some s-t path and returns how much flow reached t
- v is current node
- aug is how much flow can be pushed up to this point

```
int dfs(int v, int aug) {
    if (v == t) return aug;
    return 0;
}
```

Dinic's Implementation

- augments some s-t path and returns how much flow reached t
- v is current node
- aug is how much flow can be pushed up to this point

```
int dfs(int v, int aug) {
    if (v == t) return aug;
    for (int& i = next[v]; i<adj[v].size(); ++i) {
        auto e = adj[v][i];

    }
    return 0;
}
```

Dinic's Implementation

- augments some s-t path and returns how much flow reached t
- v is current node
- aug is how much flow can be pushed up to this point
- note that i is a reference

```
int dfs(int v, int aug) {
    if (v == t) return aug;
    for (int& i = next[v]; i<adj[v].size(); ++i) {
        auto e = adj[v][i];



    }
    return 0;
}
```

Dinic's Implementation

- augments some s-t path and returns how much flow reached t
- v is current node
- aug is how much flow can be pushed up to this point
- note that i is a reference

```
int dfs(int v, int aug) {
    if (v == t) return aug;
    for (int& i = next[v]; i<adj[v].size(); ++i) {
        auto e = adj[v][i];
        if (e->flow == e->cap) continue;
        if (dist[e->to] != dist[v] + 1) continue;



    }
    return 0;
}
```

Dinic's Implementation

- augments some s-t path and returns how much flow reached t
- v is current node
- aug is how much flow can be pushed up to this point
- note that i is a reference

```
int dfs(int v, int aug) {
    if (v == t) return aug;
    for (int& i = next[v]; i<adj[v].size(); ++i) {
        auto e = adj[v][i];
        if (e->flow == e->cap) continue;
        if (dist[e->to] != dist[v] + 1) continue;
        int pushed = dfs(e->to,
            min(aug, e->cap - e->flow));




    }
    return 0;
}
```

Dinic's Implementation

- augments some s-t path and returns how much flow reached t
- v is current node
- aug is how much flow can be pushed up to this point
- note that i is a reference

```
int dfs(int v, int aug) {
    if (v == t) return aug;
    for (int& i = next[v]; i<adj[v].size(); ++i) {
        auto e = adj[v][i];
        if (e->flow == e->cap) continue;
        if (dist[e->to] != dist[v] + 1) continue;
        int pushed = dfs(e->to,
            min(aug, e->cap - e->flow));
        if (pushed == 0) continue;



    }
    return 0;
}
```

Dinic's Implementation

- augments some s-t path and returns how much flow reached t
- v is current node
- aug is how much flow can be pushed up to this point
- note that i is a reference

```
int dfs(int v, int aug) {
    if (v == t) return aug;
    for (int& i = next[v]; i<adj[v].size(); ++i) {
        auto e = adj[v][i];
        if (e->flow == e->cap) continue;
        if (dist[e->to] != dist[v] + 1) continue;
        int pushed = dfs(e->to,
            min(aug, e->cap - e->flow));
        if (pushed == 0) continue;
        e->flow += pushed;
        e->twin->flow -= pushed;

    }
    return 0;
}
```

Dinic's Implementation

- augments some s-t path and returns how much flow reached t
- v is current node
- aug is how much flow can be pushed up to this point
- note that i is a reference

```
int dfs(int v, int aug) {
    if (v == t) return aug;
    for (int& i = next[v]; i<adj[v].size(); ++i) {
        auto e = adj[v][i];
        if (e->flow == e->cap) continue;
        if (dist[e->to] != dist[v] + 1) continue;
        int pushed = dfs(e->to,
            min(aug, e->cap - e->flow));
        if (pushed == 0) continue;
        e->flow += pushed;
        e->twin->flow -= pushed;
        return pushed;
    }
    return 0;
}
```

Dinic's Implementation

- augments some s-t path and returns how much flow reached t
- v is current node
- aug is how much flow can be pushed up to this point
- note that i is a reference
- also note that next[v] is not incremented if the edge was augmented

```
int dfs(int v, int aug) {
    if (v == t) return aug;
    for (int& i = next[v]; i<adj[v].size(); ++i) {
        auto e = adj[v][i];
        if (e->flow == e->cap) continue;
        if (dist[e->to] != dist[v] + 1) continue;
        int pushed = dfs(e->to,
            min(aug, e->cap - e->flow));
        if (pushed == 0) continue;
        e->flow += pushed;
        e->twin->flow -= pushed;
        return pushed;
    }
    return 0;
}
```

MaxFlow Algorithms Summary

Ford-Fulkerson	$O(mC)$	
Edmonds-Karp	$O(m^2n)$	
Dinic's	$O(mn^2)$	$O(m\sqrt{n})$ for unit graphs

C is value of maximum flow

MaxFlow Algorithms Summary

Ford-Fulkerson	$O(mC)$	
Edmonds-Karp	$O(m^2n)$	
Dinic's	$O(mn^2)$	$O(m\sqrt{n})$ for unit graphs

When is Ford-Fulkerson better than Dinic's?

C is value of maximum flow

MaxFlow Algorithms Summary

Ford-Fulkerson	$O(mC)$	
Edmonds-Karp	$O(m^2 n)$	
Dinic's	$O(mn^2)$	$O(m\sqrt{n})$ for unit graphs
MPM	$O(n^3)$	
F-F + Capacity Scaling	$O(m^2 \log U)$	
Push-Relabel	$O(mn^2)$	
FIFO Push-Relabel	$O(n^3)$	
highest label P-R	$O(nm + n^2\sqrt{m})$	

When is Ford-Fulkerson better than Dinic's?

C is value of maximum flow

U is highest capacity edge outgoing from source

MaxFlow Algorithms Summary

Ford-Fulkerson	$O(mC)$	
Edmonds-Karp	$O(m^2 n)$	
Dinic's	$O(mn^2)$	$O(m\sqrt{n})$ for unit graphs
MPM	$O(n^3)$	
F-F + Capacity Scaling	$O(m^2 \log U)$	
Push-Relabel	$O(mn^2)$	
FIFO Push-Relabel	$O(n^3)$	
highest label P-R	$O(nm + n^2\sqrt{m})$	

When is Ford-Fulkerson better than Dinic's?

- in contest implement Edmonds-Karp (FF with BFS)

C is value of maximum flow

U is highest capacity edge outgoing from source

22 lines MaxFlow:

```
int flow = 0;
while (true) {
    vector<edge*> inc(n, nullptr);
    queue<int> q{{s}};
    while (!q.empty()) {
        auto v = q.front(); q.pop();
        for (auto e : adj[v])
            if (!inc[e->to] && e->flow< e->cap) {
                q.push(e->to);
                inc[e->to] = e;
            }
    }
    if (!inc[t]) break;
    int aug = 1e9;
    for (int v = t; v != s; v = inc[v]->from)
        aug = min(aug, inc[v]->cap - inc[v]->flow);
    flow += aug;
    for (int v = t; v != s; v = inc[v]->from) {
        inc[v]->flow += aug;
        inc[v]->twin->flow -= aug;
    }
}
```

r unit graphs

When is Ford-Fulkerson better than Dinic's?

- in contest implement Edmonds-Karp (FF with BFS)

imum flow

city edge outgoing from source

MaxFlow Algorithms Summary

Ford-Fulkerson	$O(mC)$	
Edmonds-Karp	$O(m^2 n)$	
Dinic's	$O(mn^2)$	$O(m\sqrt{n})$ for unit graphs
MPM	$O(n^3)$	
F-F + Capacity Scaling	$O(m^2 \log U)$	
Push-Relabel	$O(mn^2)$	
FIFO Push-Relabel	$O(n^3)$	
highest label P-R	$O(nm + n^2\sqrt{m})$	

When is Ford-Fulkerson better than Dinic's?

- in contest implement Edmonds-Karp (FF with BFS)
- if TLE, write one more DFS and you got Dinic's

C is value of maximum flow

U is highest capacity edge outgoing from source

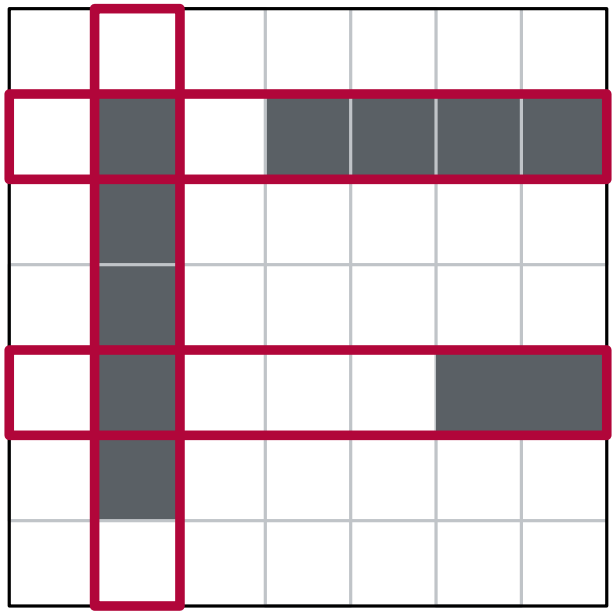
Small Break

There is a square grid of size $n \times n$. There are k black cells, all others are colored in white. In one operation you can select a row or a column and color it white. You are to make all cells white with minimum number of operations.

codeforces.com/problemset/problem/1198/E

$1 \leq n \leq 10^9$

$0 \leq k \leq 50$



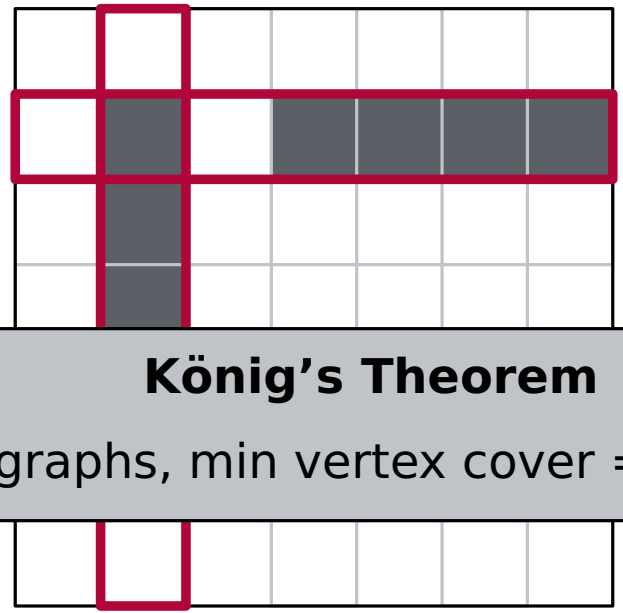
Small Break

There is a square grid of size $n \times n$. There are k black cells, all others are colored in white. In one operation you can select a row or a column and color it white. You are to make all cells white with minimum number of operations.

codeforces.com/problemset/problem/1198/E

$1 \leq n \leq 10^9$

$0 \leq k \leq 50$

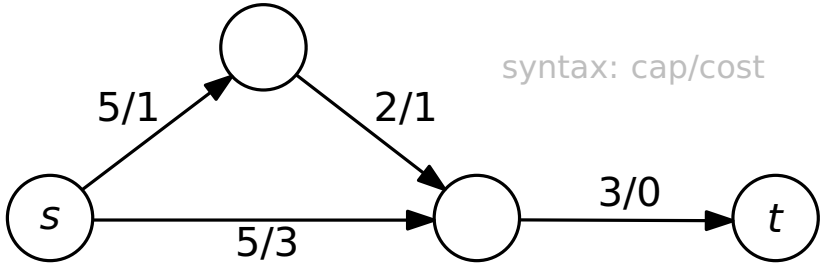


König's Theorem

in bipartite graphs, min vertex cover = max matching

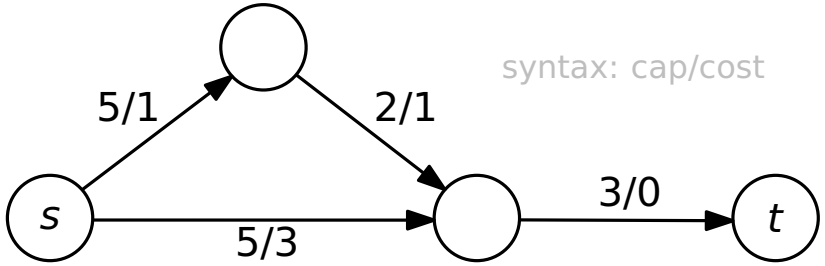
MinCostFlows

- Flow with additional cost function cost : $E \to \mathbb{Z}$



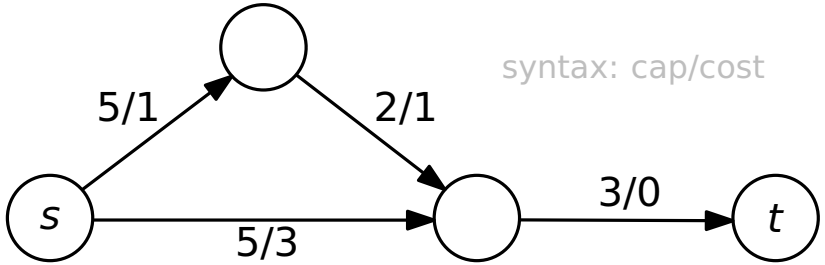
MinCostFlows

- Flow with additional cost function cost : $E \to \mathbb{Z}$
- Directed only (use 2 edges for undirected)



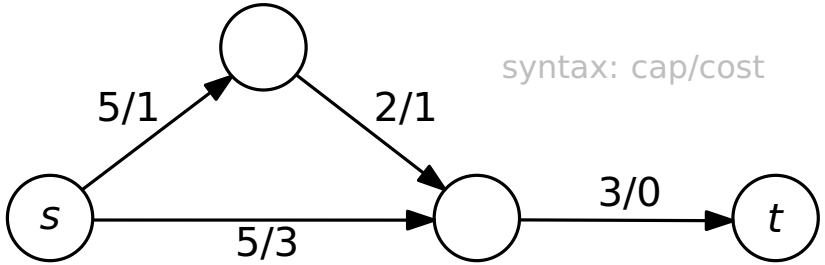
MinCostFlows

- Flow with additional cost function cost : $E \to \mathbb{Z}$
- Directed only (use 2 edges for undirected)
  - G_f is a multi-graph for undirected G



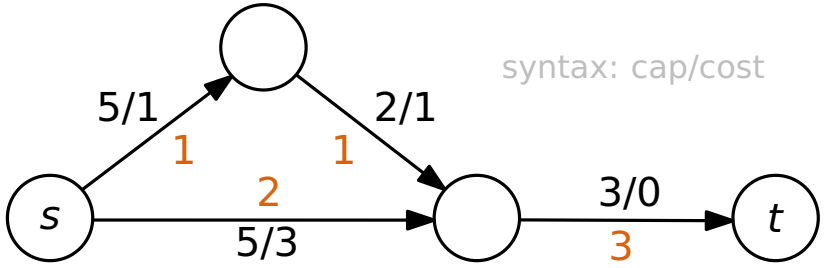
MinCostFlows

- Flow with additional cost function cost : $E \to \mathbb{Z}$
- Directed only (use 2 edges for undirected)
  - G_f is a multi-graph for undirected G
- Cost is per unit of flow!



MinCostFlows

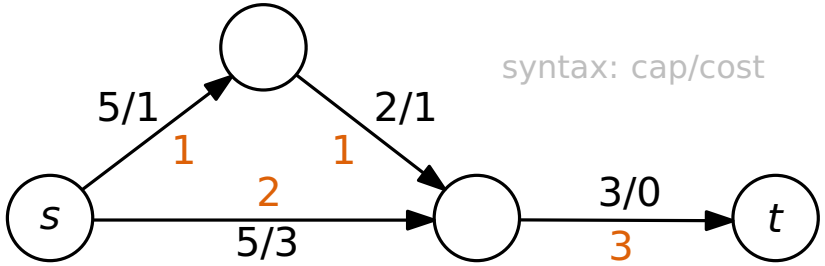
- Flow with additional cost function cost : $E \to \mathbb{Z}$
- Directed only (use 2 edges for undirected)
  - G_f is a multi-graph for undirected G
- Cost is per unit of flow!
  - $\text{cost}(f) = \sum_{(u,v)\in E} f(u,v) \cdot \text{cost}(u,v)$



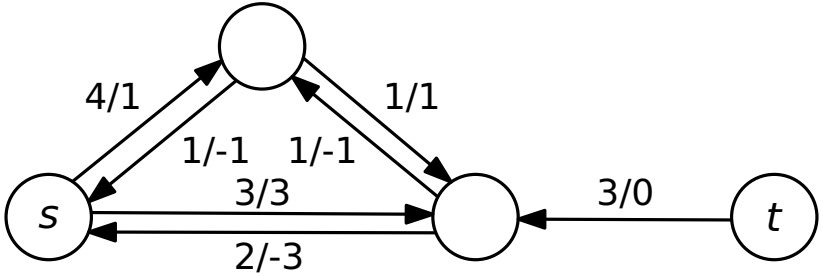
$\text{cost}(f) = 8$

MinCostFlows

- Flow with additional cost function cost : $E \to \mathbb{Z}$
- Directed only (use 2 edges for undirected)
  - G_f is a multi-graph for undirected G
- Cost is per unit of flow!
  - $\text{cost}(f) = \sum_{(u,v)\in E} f(u,v) \cdot \text{cost}(u,v)$
- In G_f: $\text{cost}_{G_f}(v,u) = -\text{cost}_{G_f}(u,v)$

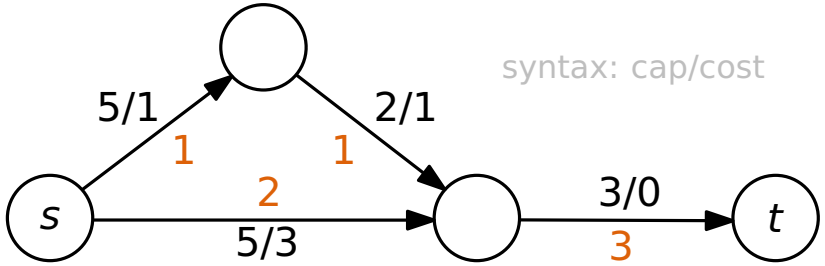


$\text{cost}(f) = 8$

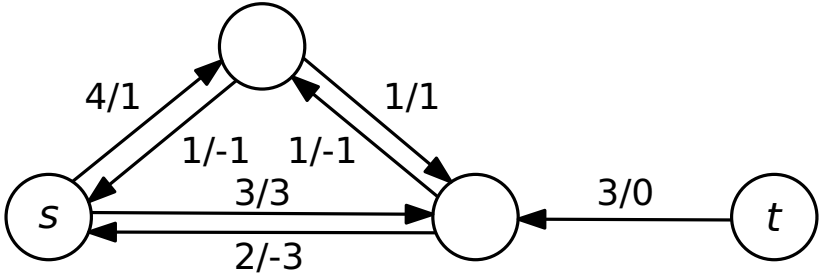


MinCostFlows

- Flow with additional cost function cost : $E \to \mathbb{Z}$
- Directed only (use 2 edges for undirected)
  - G_f is a multi-graph for undirected G
- Cost is per unit of flow!
  - $\text{cost}(f) = \sum_{(u,v)\in E} f(u,v) \cdot \text{cost}(u,v)$
- In G_f: $\text{cost}_{G_f}(v,u) = -\text{cost}_{G_f}(u,v)$
- Variations:

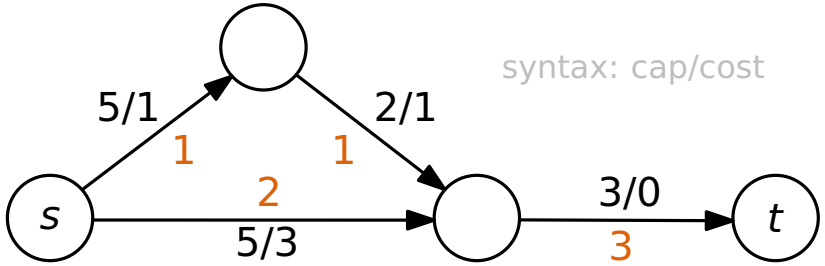


$\text{cost}(f) = 8$

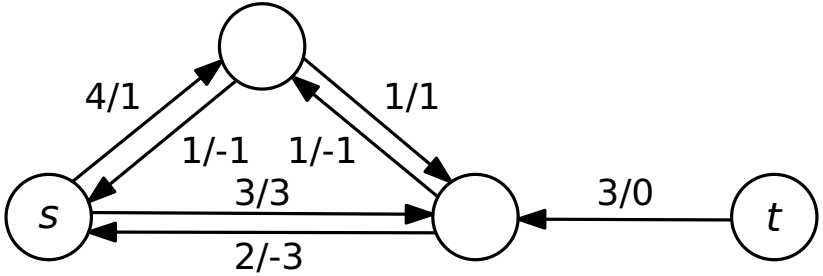


MinCostFlows

- Flow with additional cost function cost : $E \to \mathbb{Z}$
- Directed only (use 2 edges for undirected)
  - G_f is a multi-graph for undirected G
- Cost is per unit of flow!
  - $\text{cost}(f) = \sum_{(u,v)\in E} f(u,v) \cdot \text{cost}(u,v)$
- In G_f: $\text{cost}_{G_f}(v,u) = -\text{cost}_{G_f}(u,v)$
- Variations:
  - MinCostMaxFlow

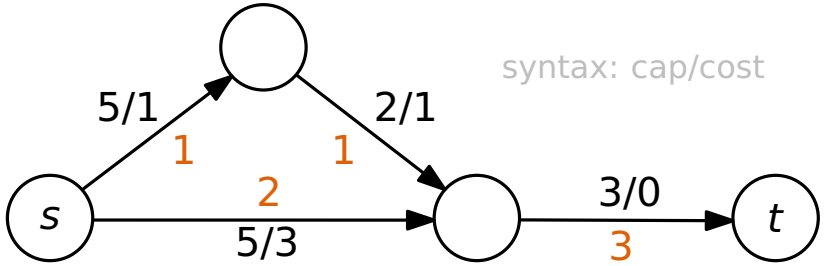


$\text{cost}(f) = 8$

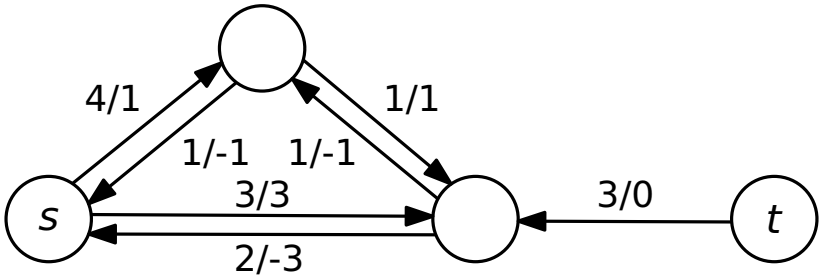


MinCostFlows

- Flow with additional cost function cost : $E \to \mathbb{Z}$
- Directed only (use 2 edges for undirected)
  - G_f is a multi-graph for undirected G
- Cost is per unit of flow!
  - $\text{cost}(f) = \sum_{(u,v)\in E} f(u,v) \cdot \text{cost}(u,v)$
- In G_f: $\text{cost}_{G_f}(v,u) = -\text{cost}_{G_f}(u,v)$
- Variations:
  - MinCostMaxFlow
  - MinCost-k-Flow

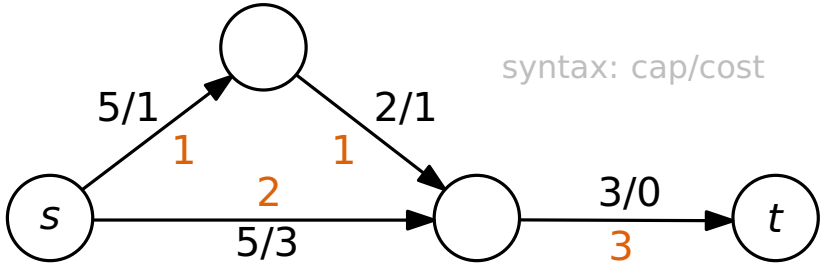


$\text{cost}(f) = 8$

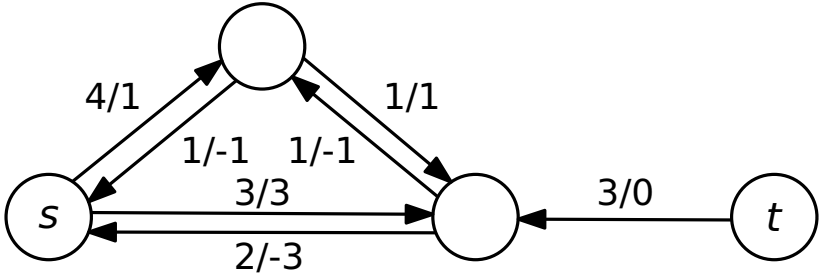


MinCostFlows

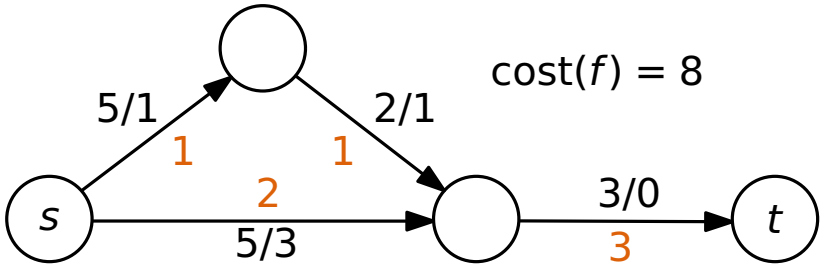
- Flow with additional cost function cost : $E \to \mathbb{Z}$
- Directed only (use 2 edges for undirected)
  - G_f is a multi-graph for undirected G
- Cost is per unit of flow!
  - $\text{cost}(f) = \sum_{(u,v)\in E} f(u,v) \cdot \text{cost}(u,v)$
- In G_f: $\text{cost}_{G_f}(v,u) = -\text{cost}_{G_f}(u,v)$
- Variations:
  - MinCostMaxFlow
  - MinCost-k-Flow
  - MinCostFlow



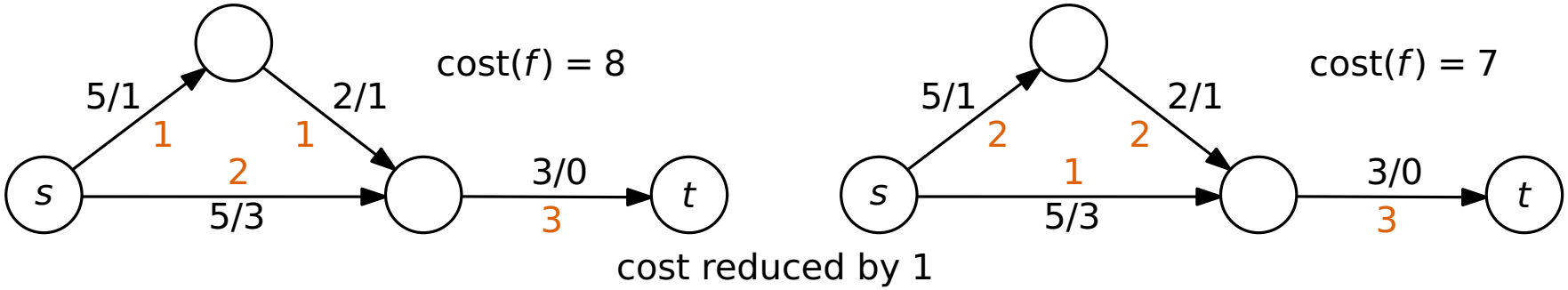
$\text{cost}(f) = 8$



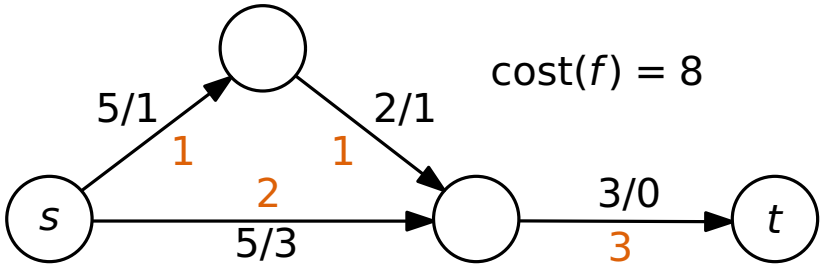
Negative Cost Cycles



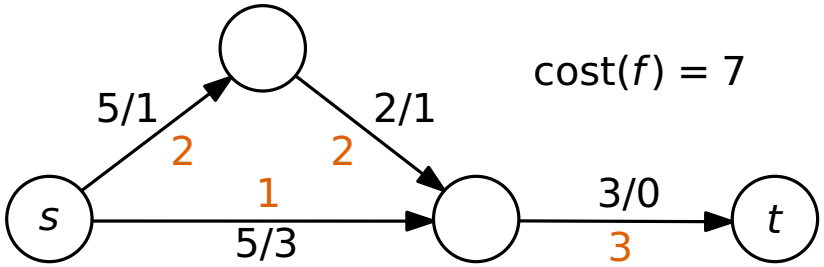
Negative Cost Cycles



Negative Cost Cycles

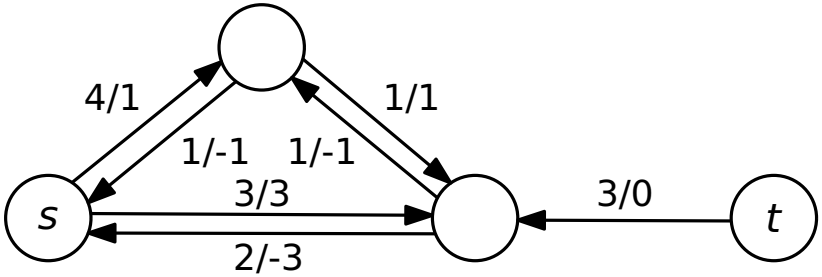


cost(f) = 8

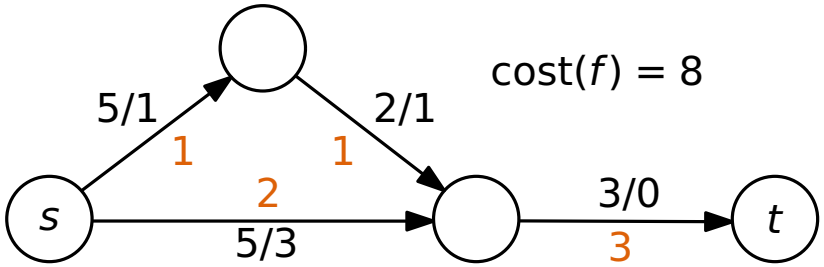


cost(f) = 7

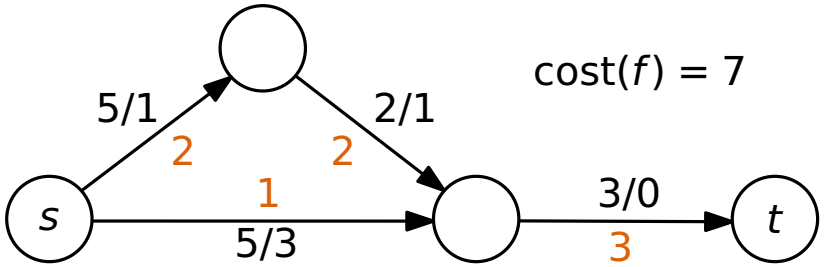
cost reduced by 1



Negative Cost Cycles



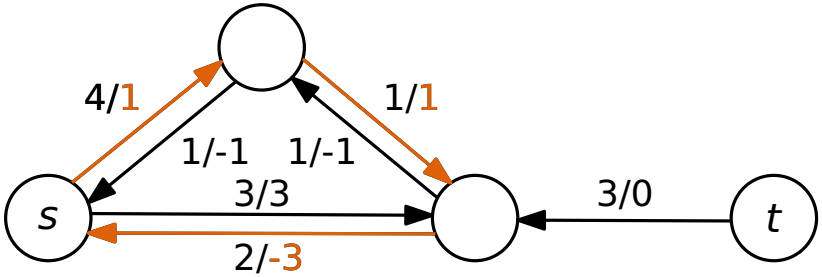
cost(f) = 8



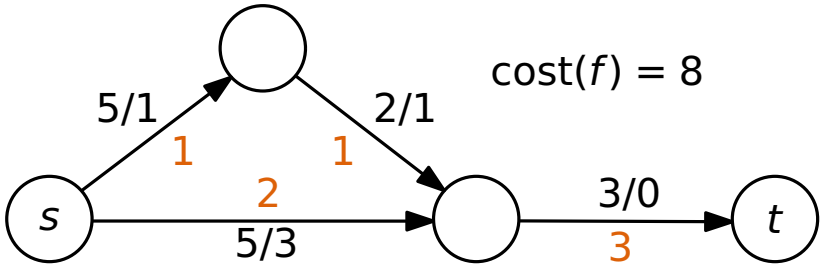
cost(f) = 7

cost reduced by 1

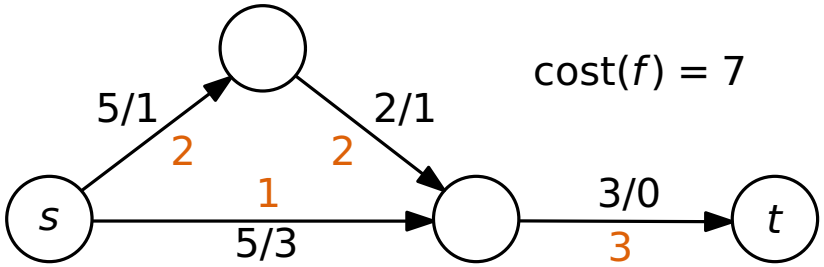
Negative cost cycle in G_f!



Negative Cost Cycles



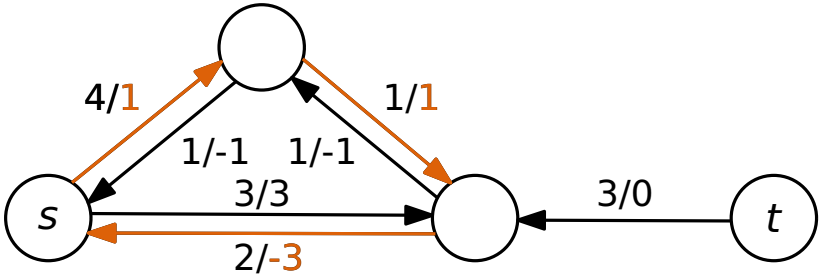
cost(f) = 8



cost(f) = 7

cost reduced by 1

Negative cost cycle in G_f!



Lemma: A k-flow f on some network G is cost minimal iff. G_f does not contain a negative cost cycle C

Cycle Cancelling Algorithm

Cycle Cancelling Algorithm:

- find max flow f
- **while** $\exists$ negative cost cycle C in G_f **do**
  - augment f with C

Cycle Cancelling Algorithm

Cycle Cancelling Algorithm:

- find max flow f
- **while** $\exists$ negative cost cycle C in G_f **do**
  - augment f with C

- Runtime?

Cycle Cancelling Algorithm

Cycle Cancelling Algorithm:

- find max flow f
- **while** $\exists$ negative cost cycle C in G_f **do**
  - augment f with C

- Runtime?
  - Every augmentation removes at least 1 cost

Cycle Cancelling Algorithm

Cycle Cancelling Algorithm:

- find max flow f
- **while** $\exists$ negative cost cycle C in G_f **do**
  - augment f with C

- Runtime?
  - Every augmentation removes at least 1 cost
  - $O(\text{maxflow}(n, m) + \text{findcycle}(n, m) \cdot mCU)$

Cycle Cancelling Algorithm

Cycle Cancelling Algorithm:

- find max flow f
- **while** $\exists$ negative cost cycle C in G_f **do**
  - augment f with C

- Runtime?
  - Every augmentation removes at least 1 cost
  - $O(\text{maxflow}(n, m) + \text{findcycle}(n, m) \cdot mCU)$
    - U: upper bound on edge capacity

Cycle Cancelling Algorithm

Cycle Cancelling Algorithm:

- find max flow f
- **while** $\exists$ negative cost cycle C in G_f **do**
  - augment f with C

- Runtime?
  - Every augmentation removes at least 1 cost
  - $O(\text{maxflow}(n, m) + \text{findcycle}(n, m) \cdot mCU)$
    - U: upper bound on edge capacity
    - C: upper bound on absolute cost

Cycle Cancelling Algorithm

Cycle Cancelling Algorithm:

- find max flow f
- **while** $\exists$ negative cost cycle C in G_f **do**
  - augment f with C

- Runtime?
  - Every augmentation removes at least 1 cost
  - $O(\text{maxflow}(n, m) + \text{findcycle}(n, m) \cdot mCU)$
    - U: upper bound on edge capacity
    - C: upper bound on absolute cost

How do we find negative cycles?

Cycle Cancelling Algorithm

Cycle Cancelling Algorithm:

- find max flow f
- **while** $\exists$ negative cost cycle C in G_f **do**
  - augment f with C

- Runtime?
  - Every augmentation removes at least 1 cost
  - $O(\text{maxflow}(n, m) + \text{findcycle}(n, m) \cdot mCU)$
    - U: upper bound on edge capacity
    - C: upper bound on absolute cost
  - Overall $O(nm^2CU)$

How do we find negative cycles?

Path Augmentation

Lemma: Let f be a cost-minimal k-flow and P a shortest s-t-path in G_f. Then $f' := f + f_P$ is a cost-minimal $(k+1)$-flow.

f_P: one unit of flow along P

Path Augmentation

Lemma: Let f be a cost-minimal k-flow and P a shortest s-t-path in G_f. Then $f' := f + f_P$ is a cost-minimal $(k+1)$-flow.

f_P: one unit of flow along P

obvious

Path Augmentation

Lemma: Let f be a cost-minimal k-flow and P a shortest s-t-path in G_f. Then $f' := f + f_P$ is a cost-minimal $(k+1)$-flow.

needs proof obvious

f_P: one unit of flow along P

Path Augmentation

Lemma: Let f be a cost-minimal k-flow and P a shortest s-t-path in G_f. Then $f' := f + f_P$ is a cost-minimal $(k+1)$-flow.

f_P: one unit of flow along P

needs proof obvious

Proof sketch

Path Augmentation

Lemma: Let f be a cost-minimal k-flow and P a shortest s-t-path in G_f. Then $f' := f + f_P$ is a cost-minimal $(k+1)$-flow.

f_P: one unit of flow along P

needs proof obvious

Proof sketch

Assume f' is not cost-minimal. Then $G_{f'}$ contains a negative cost cycle C.

Path Augmentation

Lemma: Let f be a cost-minimal k-flow and P a shortest s-t-path in G_f. Then $f' := f + f_P$ is a cost-minimal $(k+1)$-flow.

f_P: one unit of flow along P

needs proof obvious

Proof sketch

Assume f' is not cost-minimal. Then $G_{f'}$ contains a negative cost cycle C.

$C \cap P \neq \emptyset$, otherwise C would exist in G_f

Path Augmentation

Lemma: Let f be a cost-minimal k-flow and P a shortest s-t-path in G_f. Then $f' := f + f_P$ is a cost-minimal $(k+1)$-flow.

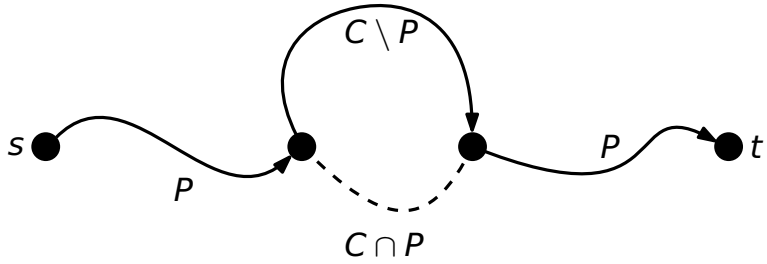
needs proof obvious

f_P: one unit of flow along P

Proof sketch

Assume f' is not cost-minimal. Then $G_{f'}$ contains a negative cost cycle C.

$C \cap P \neq \emptyset$, otherwise C would exist in G_f



Path Augmentation

Lemma: Let f be a cost-minimal k-flow and P a shortest s-t-path in G_f. Then $f' := f + f_P$ is a cost-minimal $(k+1)$-flow.

needs proof obvious

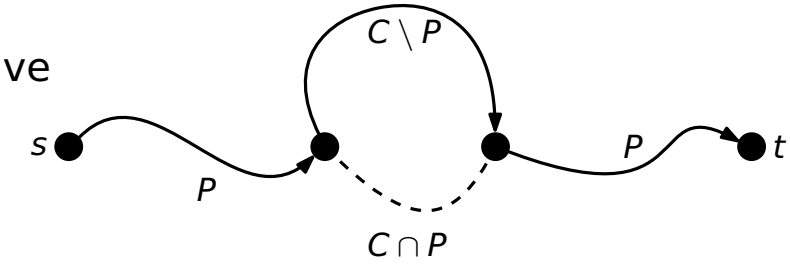
f_P: one unit of flow along P

Proof sketch

Assume f' is not cost-minimal. Then $G_{f'}$ contains a negative cost cycle C.

$C \cap P \neq \emptyset$, otherwise C would exist in G_f

$C \cap P$ has changed direction to make cycle negative



Path Augmentation

Lemma: Let f be a cost-minimal k-flow and P a shortest s-t-path in G_f. Then $f' := f + f_P$ is a cost-minimal $(k+1)$-flow.

needs proof obvious

f_P: one unit of flow along P

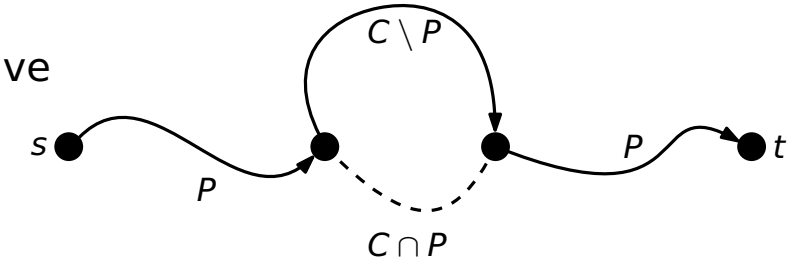
Proof sketch

Assume f' is not cost-minimal. Then $G_{f'}$ contains a negative cost cycle C.

$C \cap P \neq \emptyset$, otherwise C would exist in G_f

$C \cap P$ has changed direction to make cycle negative

Cost of P in G_f was $\text{cost}(P \setminus C) - \text{cost}(C \cap P)$



Path Augmentation

Lemma: Let f be a cost-minimal k-flow and P a shortest s-t-path in G_f. Then $f' := f + f_P$ is a cost-minimal $(k+1)$-flow.

needs proof obvious

f_P: one unit of flow along P

Proof sketch

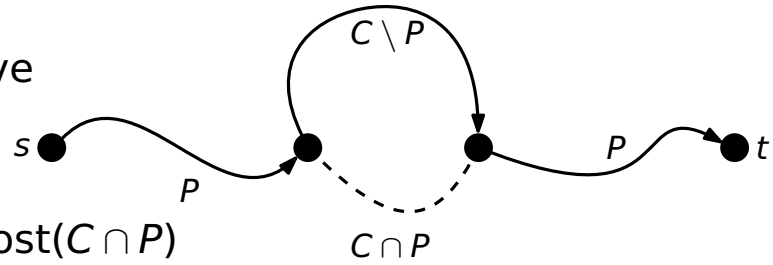
Assume f' is not cost-minimal. Then $G_{f'}$ contains a negative cost cycle C.

$C \cap P \neq \emptyset$, otherwise C would exist in G_f

$C \cap P$ has changed direction to make cycle negative

Cost of P in G_f was $\text{cost}(P \setminus C) - \text{cost}(C \cap P)$

$\text{cost}(C \setminus P) + \text{cost}(C \cap P) < 0$, thus $\text{cost}(C \setminus P) < -\text{cost}(C \cap P)$



Path Augmentation

Lemma: Let f be a cost-minimal k-flow and P a shortest s-t-path in G_f. Then $f' := f + f_P$ is a cost-minimal $(k+1)$-flow.

needs proof obvious

f_P: one unit of flow along P

Proof sketch

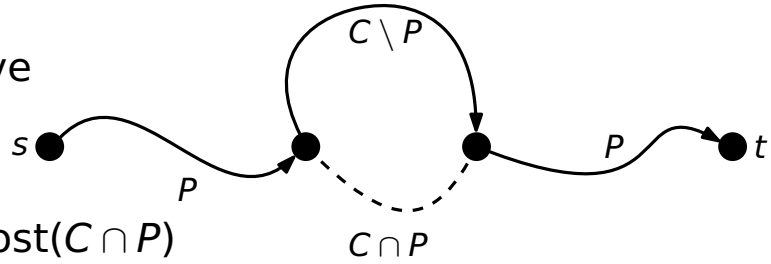
Assume f' is not cost-minimal. Then $G_{f'}$ contains a negative cost cycle C.

$C \cap P \neq \emptyset$, otherwise C would exist in G_f

$C \cap P$ has changed direction to make cycle negative

Cost of P in G_f was $\text{cost}(P \setminus C) - \text{cost}(C \cap P)$

$\text{cost}(C \setminus P) + \text{cost}(C \cap P) < 0$, thus $\text{cost}(C \setminus P) < -\text{cost}(C \cap P)$



↯ $(P \setminus C) \cup (C \setminus P)$ shorter than P in G_f!

Path Augmentation

Lemma: Let f be a cost-minimal k-flow and P a shortest s-t-path in G_f. Then $f' := f + f_P$ is a cost-minimal $(k+1)$-flow.

needs proof obvious

f_P: one unit of flow along P

Proof sketch

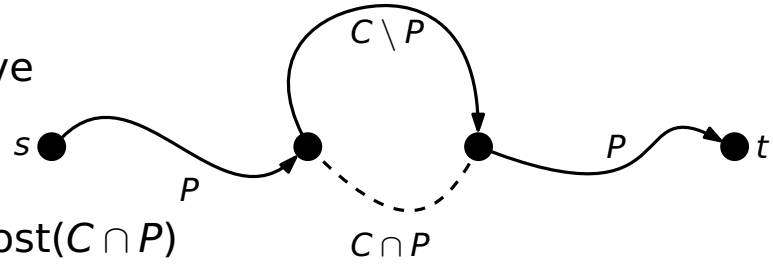
Assume f' is not cost-minimal. Then $G_{f'}$ contains a negative cost cycle C.

$C \cap P \neq \emptyset$, otherwise C would exist in G_f

$C \cap P$ has changed direction to make cycle negative

Cost of P in G_f was $\text{cost}(P \setminus C) - \text{cost}(C \cap P)$

$\text{cost}(C \setminus P) + \text{cost}(C \cap P) < 0$, thus $\text{cost}(C \setminus P) < -\text{cost}(C \cap P)$



↯ $(P \setminus C) \cup (C \setminus P)$ shorter than P in G_f!

Omitted: multiple cycles, multiple intersections of C and P

Successive Shortest Path Algorithm

Successive Shortest Path Algorithm:

- start with null flow f
- **while** s and t connected in G_f **do**
  - find shortest s-t-path P in G_f
  - augment f with P

Successive Shortest Path Algorithm

- Only works if null flow is cost minimal!

Successive Shortest Path Algorithm:

- start with null flow f
- **while** s and t connected in G_f **do**
  - find shortest s-t-path P in G_f
  - augment f with P

Successive Shortest Path Algorithm

- Only works if null flow is cost minimal!
  - e.g. if G does not contain negative cycles

Successive Shortest Path Algorithm:

- start with null flow f
- **while** s and t connected in G_f **do**
  - find shortest s-t-path P in G_f
  - augment f with P

Successive Shortest Path Algorithm

- Only works if null flow is cost minimal!
  - e.g. if G does not contain negative cycles
- G_f may contain negative edges

Successive Shortest Path Algorithm:

- start with null flow f
- **while** s and t connected in G_f **do**
  - find shortest s-t-path P in G_f
  - augment f with P

Successive Shortest Path Algorithm

- Only works if null flow is cost minimal!
  - e.g. if G does not contain negative cycles
- G_f may contain negative edges
  - use Bellman Ford

Successive Shortest Path Algorithm:

- start with null flow f
- **while** s and t connected in G_f **do**
  - find shortest s-t-path P in G_f
  - augment f with P

Successive Shortest Path Algorithm

- Only works if null flow is cost minimal!
  - e.g. if G does not contain negative cycles
- G_f may contain negative edges
  - use Bellman Ford
- Runtime: $O(nmB)$ (B: value of resulting flow)

Successive Shortest Path Algorithm:

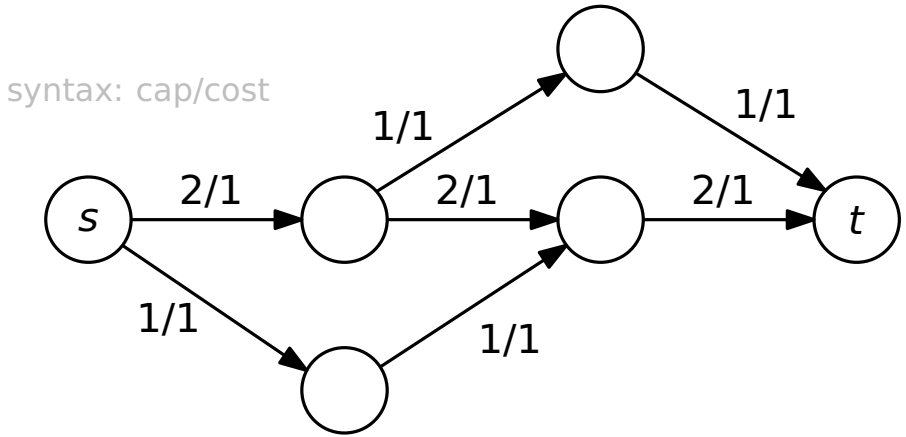
- start with null flow f
- **while** s and t connected in G_f **do**
  - find shortest s-t-path P in G_f
  - augment f with P

Successive Shortest Path Algorithm

- Only works if null flow is cost minimal!
  - e.g. if G does not contain negative cycles
- G_f may contain negative edges
  - use Bellman Ford
- Runtime: $O(nmB)$ (B: value of resulting flow)

Successive Shortest Path Algorithm:

- start with null flow f
- **while** s and t connected in G_f **do**
  - find shortest s-t-path P in G_f
  - augment f with P

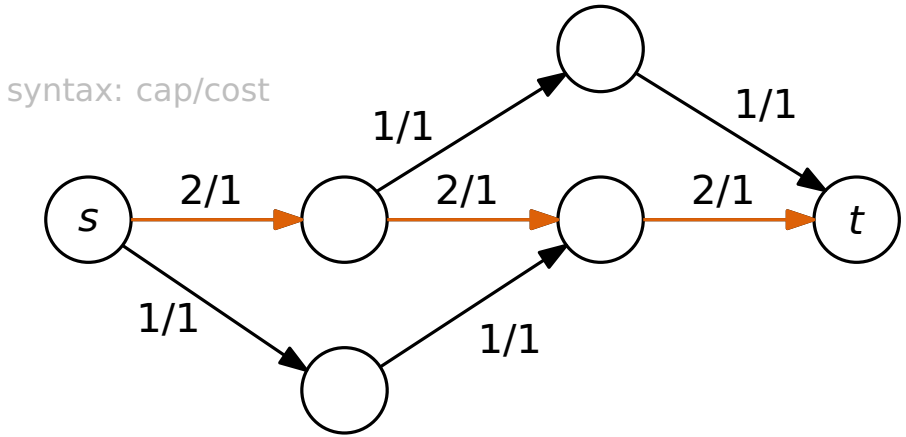


Successive Shortest Path Algorithm

- Only works if null flow is cost minimal!
  - e.g. if G does not contain negative cycles
- G_f may contain negative edges
  - use Bellman Ford
- Runtime: $O(nmB)$ (B: value of resulting flow)

Successive Shortest Path Algorithm:

- start with null flow f
- **while** s and t connected in G_f **do**
  - find shortest s-t-path P in G_f
  - augment f with P

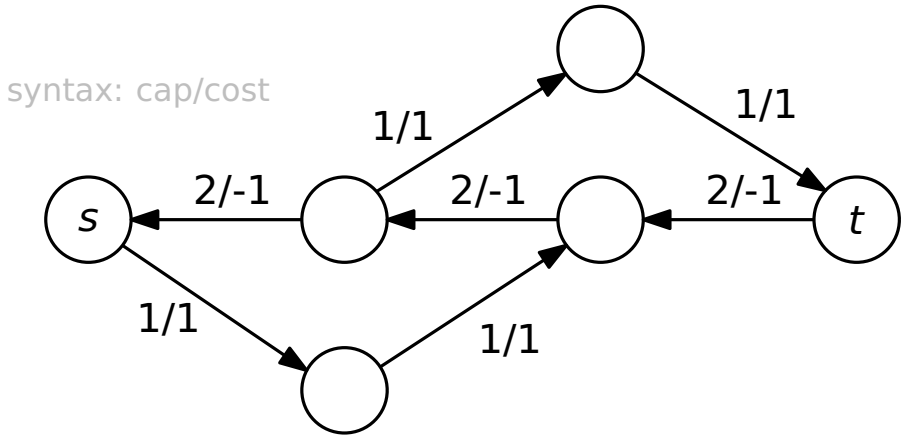


Successive Shortest Path Algorithm

- Only works if null flow is cost minimal!
  - e.g. if G does not contain negative cycles
- G_f may contain negative edges
  - use Bellman Ford
- Runtime: $O(nmB)$ (B: value of resulting flow)

Successive Shortest Path Algorithm:

- start with null flow f
- **while** s and t connected in G_f **do**
  - find shortest s-t-path P in G_f
  - augment f with P

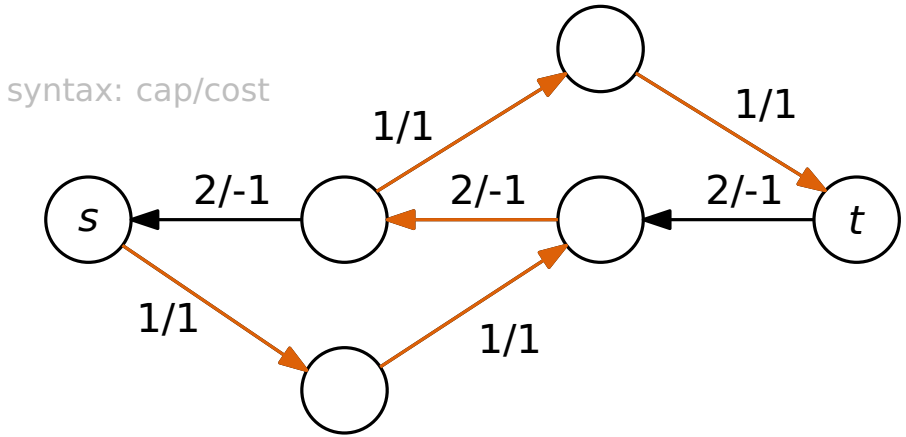


Successive Shortest Path Algorithm

- Only works if null flow is cost minimal!
  - e.g. if G does not contain negative cycles
- G_f may contain negative edges
  - use Bellman Ford
- Runtime: $O(nmB)$ (B: value of resulting flow)

Successive Shortest Path Algorithm:

- start with null flow f
- **while** s and t connected in G_f **do**
  - find shortest s-t-path P in G_f
  - augment f with P

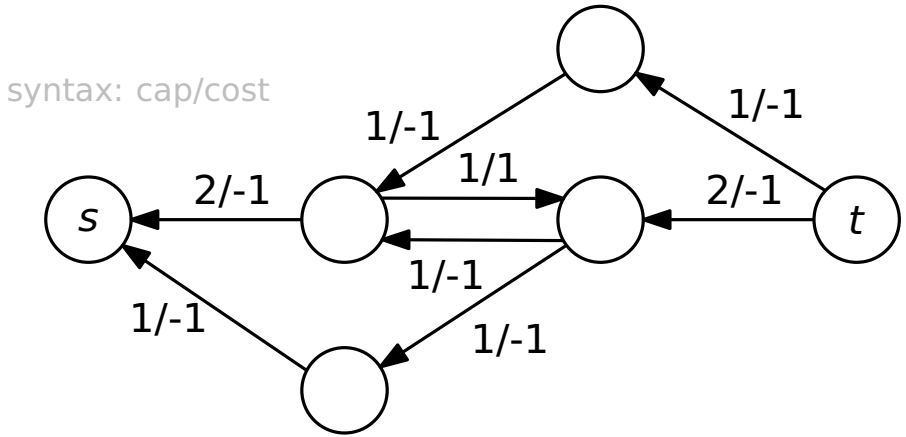


Successive Shortest Path Algorithm

- Only works if null flow is cost minimal!
  - e.g. if G does not contain negative cycles
- G_f may contain negative edges
  - use Bellman Ford
- Runtime: $O(nmB)$ (B: value of resulting flow)

Successive Shortest Path Algorithm:

- start with null flow f
- **while** s and t connected in G_f **do**
  - find shortest s-t-path P in G_f
  - augment f with P

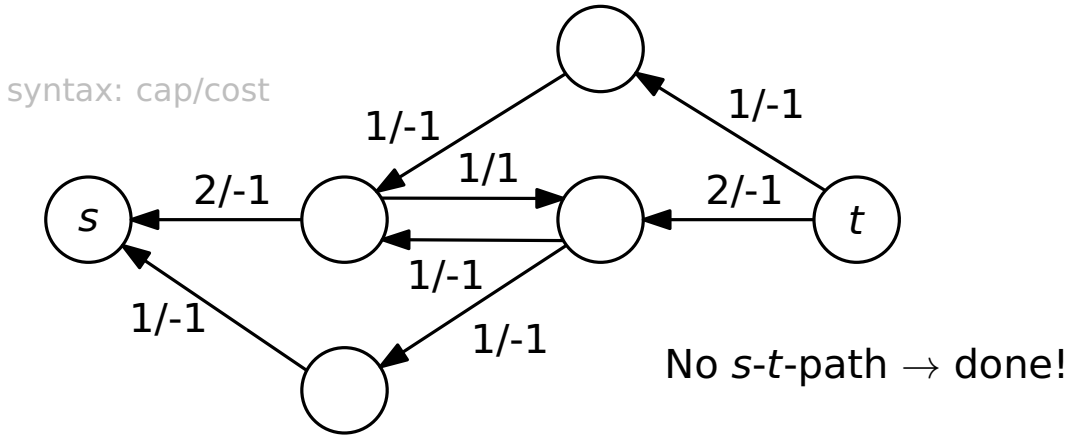


Successive Shortest Path Algorithm

- Only works if null flow is cost minimal!
  - e.g. if G does not contain negative cycles
- G_f may contain negative edges
  - use Bellman Ford
- Runtime: $O(nmB)$ (B: value of resulting flow)

Successive Shortest Path Algorithm:

- start with null flow f
- **while** s and t connected in G_f **do**
  - find shortest s-t-path P in G_f
  - augment f with P



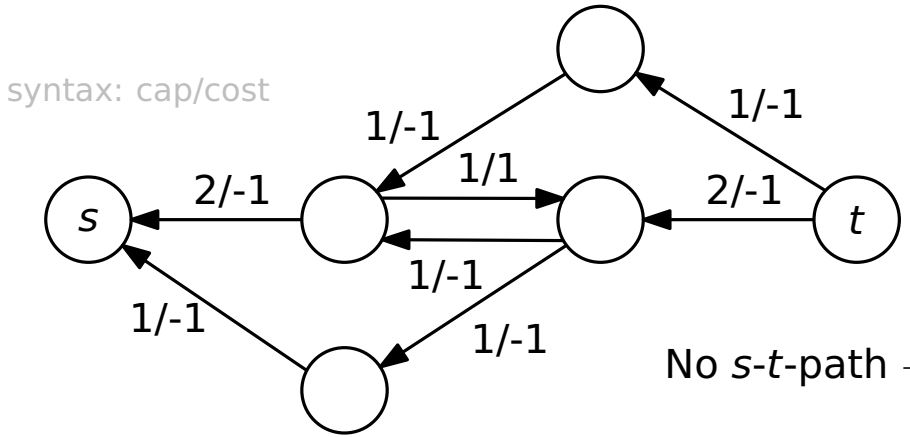
Successive Shortest Path Algorithm

- Only works if null flow is cost minimal!
  - e.g. if G does not contain negative cycles
- G_f may contain negative edges
  - use Bellman Ford
- Runtime: $O(nmB)$ (B: value of resulting flow)

Successive Shortest Path Algorithm:

- start with null flow f
- **while** s and t connected in G_f **do**
  - find shortest s-t-path P in G_f
  - augment f with P

How to solve MinCost-k-Flow?



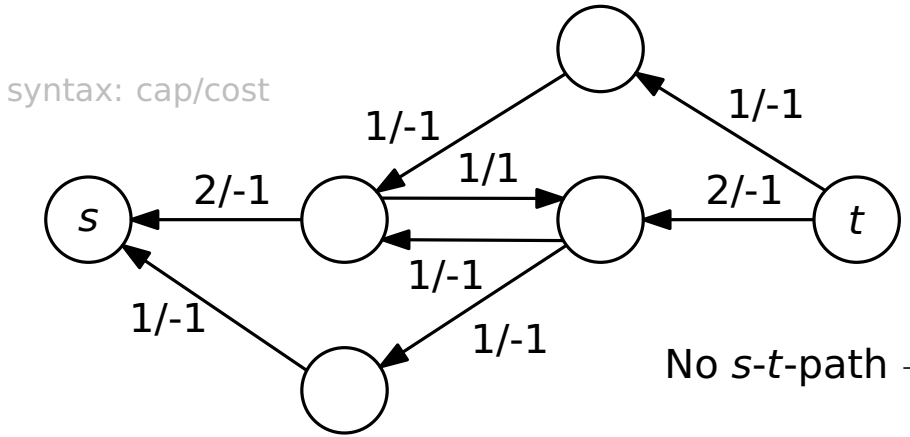
No s-t-path $\rightarrow$ done!

Successive Shortest Path Algorithm

- Only works if null flow is cost minimal!
  - e.g. if G does not contain negative cycles
- G_f may contain negative edges
  - use Bellman Ford
- Runtime: $O(nmB)$ (B: value of resulting flow)

Successive Shortest Path Algorithm:

- start with null flow f
- **while** s and t connected in G_f **do**
  - find shortest s-t-path P in G_f
  - augment f with P



No s-t-path $\rightarrow$ done!

How to solve MinCost-k-Flow?

How to solve MinCostFlow?

SSP with Dijkstra

SSP with Dijkstra

- Use Johnson’s trick:

SSP with Dijkstra

- Use Johnson's trick:
  - Take some mapping $\pi : V \mapsto \mathbb{Z}$

SSP with Dijkstra

- Use Johnson's trick:
  - Take some mapping $\pi : V \mapsto \mathbb{Z}$
  - Use $w(u, v) = \pi(u) + \text{cost}(u, v) - \pi(v)$ as weight for edges $(u, v) \in E$

SSP with Dijkstra

- Use Johnson's trick:
  - Take some mapping $\pi : V \mapsto \mathbb{Z}$
  - Use $w(u, v) = \pi(u) + \text{cost}(u, v) - \pi(v)$ as weight for edges $(u, v) \in E$
  - For cycles:

SSP with Dijkstra

- Use Johnson's trick:
  - Take some mapping $\pi : V \mapsto \mathbb{Z}$
  - Use $w(u, v) = \pi(u) + \text{cost}(u, v) - \pi(v)$ as weight for edges $(u, v) \in E$
  - For cycles:
    - $w(C) = \pi(c_1) + \text{cost}(c_1, c_2) - \pi(c_2) + \pi(c_2) + \text{cost}(c_2, c_3) - \pi(c_3) + \pi(c_3) + \ldots$

SSP with Dijkstra

- Use Johnson's trick:
  - Take some mapping $\pi : V \mapsto \mathbb{Z}$
  - Use $w(u, v) = \pi(u) + \text{cost}(u, v) - \pi(v)$ as weight for edges $(u, v) \in E$
  - For cycles:
    - $w(C) = \pi(c_1) + \text{cost}(c_1, c_2) \cancel{- \pi(c_2)} \cancel{+ \pi(c_2)} + \text{cost}(c_2, c_3) \cancel{- \pi(c_3)} \cancel{+ \pi(c_3)} + \ldots$

SSP with Dijkstra

- Use Johnson's trick:
  - Take some mapping $\pi : V \mapsto \mathbb{Z}$
  - Use $w(u, v) = \pi(u) + \text{cost}(u, v) - \pi(v)$ as weight for edges $(u, v) \in E$
  - For cycles:
    - $w(C) = \cancel{\pi(c_1)} + \text{cost}(c_1, c_2) \cancel{- \pi(c_2)} \cancel{+ \pi(c_2)} + \text{cost}(c_2, c_3) \cancel{- \pi(c_3)} \cancel{+ \pi(c_3)} + \ldots = \text{cost}(C)$

SSP with Dijkstra

- Use Johnson's trick:
  - Take some mapping $\pi : V \mapsto \mathbb{Z}$
  - Use $w(u, v) = \pi(u) + \text{cost}(u, v) - \pi(v)$ as weight for edges $(u, v) \in E$
  - For cycles:
    - $w(C) = \cancel{\pi(c_1)} + \text{cost}(c_1, c_2) \cancel{- \pi(c_2)} \cancel{+ \pi(c_2)} + \text{cost}(c_2, c_3) \cancel{- \pi(c_3)} \cancel{+ \pi(c_3)} + \ldots = \text{cost}(C)$
    - Still no negative cycles!

SSP with Dijkstra

- Use Johnson's trick:
  - Take some mapping $\pi : V \mapsto \mathbb{Z}$
  - Use $w(u,v) = \pi(u) + \text{cost}(u,v) - \pi(v)$ as weight for edges $(u,v) \in E$
  - For cycles:
    - $w(C) = \cancel{\pi(c_1)} + \text{cost}(c_1, c_2) \cancel{- \pi(c_2)} \cancel{+ \pi(c_2)} + \text{cost}(c_2, c_3) \cancel{- \pi(c_3)} \cancel{+ \pi(c_3)} + \ldots = \text{cost}(C)$
    - Still no negative cycles!
  - For s-t-paths: $w(P) = \pi(s) - \pi(t) + \sum_{(u,v) \in P} \text{cost}(u,v)$

SSP with Dijkstra

- Use Johnson's trick:
  - Take some mapping $\pi : V \mapsto \mathbb{Z}$
  - Use $w(u,v) = \pi(u) + \text{cost}(u,v) - \pi(v)$ as weight for edges $(u,v) \in E$
  - For cycles:
    - $w(C) = \cancel{\pi(c_1)} + \text{cost}(c_1,c_2) \cancel{- \pi(c_2)} \cancel{+ \pi(c_2)} + \text{cost}(c_2,c_3) \cancel{- \pi(c_3)} \cancel{+ \pi(c_3)} + \ldots = \text{cost}(C)$
    - Still no negative cycles!
  - For s-t-paths: $w(P) = \pi(s) - \pi(t) + \sum_{(u,v) \in P} \text{cost}(u,v)$
    - Shortest s-t-path based on $w \Leftrightarrow$ shortest s-t-path based on cost

SSP with Dijkstra

- Initially we set $\pi(v) = d(s, v)$. Then $w(u, v) \geq 0$ for every $(u, v) \in E$

SSP with Dijkstra

- Initially we set $\pi(v) = d(s, v)$. Then $w(u, v) \geq 0$ for every $(u, v) \in E$
  - Proof:

$$
\begin{aligned}
& d(s,u) + \text{cost}(u,v) - d(s,v) < 0 \\
\Rightarrow \quad & d(s,u) + \text{cost}(u,v) < d(s,v) \quad \lightning
\end{aligned}
$$

SSP with Dijkstra

- Initially we set $\pi(v) = d(s, v)$. Then $w(u, v) \geq 0$ for every $(u, v) \in E$
  - Proof:

$$\begin{aligned} & d(s,u) + \text{cost}(u,v) - d(s,v) < 0 \\ \Rightarrow \quad & d(s,u) + \text{cost}(u,v) < d(s,v) \quad \lightning \end{aligned}$$

- We can calculate initial $\pi(v)$ using Bellman Ford

SSP with Dijkstra

- Initially we set $\pi(v) = d(s, v)$. Then $w(u, v) \geq 0$ for every $(u, v) \in E$
  - Proof:

$$\begin{aligned} & d(s,u) + \text{cost}(u,v) - d(s,v) < 0 \\ \Rightarrow \quad & d(s,u) + \text{cost}(u,v) < d(s,v) \quad \lightning \end{aligned}$$

- We can calculate initial $\pi(v)$ using Bellman Ford
  - If G contains no negative edges, we can initialize with $\pi(v) = 0$

SSP with Dijkstra

- Initially we set $\pi(v) = d(s, v)$. Then $w(u, v) \geq 0$ for every $(u, v) \in E$
  - Proof:

$$\begin{aligned} & d(s,u) + \text{cost}(u,v) - d(s,v) < 0 \\ \Rightarrow \quad & d(s,u) + \text{cost}(u,v) < d(s,v) \quad \lightning \end{aligned}$$

- We can calculate initial $\pi(v)$ using Bellman Ford
  - If G contains no negative edges, we can initialize with $\pi(v) = 0$
- How to update π?

SSP with Dijkstra

- Initially we set $\pi(v) = d(s, v)$. Then $w(u, v) \geq 0$ for every $(u, v) \in E$
  - Proof:

$$\begin{aligned} & d(s,u) + \text{cost}(u,v) - d(s,v) < 0 \\ \Rightarrow \quad & d(s,u) + \text{cost}(u,v) < d(s,v) \quad \lightning \end{aligned}$$

- We can calculate initial $\pi(v)$ using Bellman Ford
  - If G contains no negative edges, we can initialize with $\pi(v) = 0$
- How to update π?
  - Use distances d' calculated by Dijkstra in current iteration

SSP with Dijkstra

- Initially we set $\pi(v) = d(s, v)$. Then $w(u, v) \geq 0$ for every $(u, v) \in E$
  - Proof:

$$\begin{aligned} & d(s,u) + \text{cost}(u,v) - d(s,v) < 0 \\ \Rightarrow \quad & d(s,u) + \text{cost}(u,v) < d(s,v) \quad \lightning \end{aligned}$$

- We can calculate initial $\pi(v)$ using Bellman Ford
  - If G contains no negative edges, we can initialize with $\pi(v) = 0$
- How to update π?
  - Use distances d' calculated by Dijkstra in current iteration
  - Dijkstra stops once distance to t is known but luckily we only need to update π for nodes that are done by this point

SSP with Dijkstra

- Initially we set $\pi(v) = d(s, v)$. Then $w(u, v) \geq 0$ for every $(u, v) \in E$
  - Proof:

$$\begin{aligned} & d(s,u) + \text{cost}(u,v) - d(s,v) < 0 \\ \Rightarrow \quad & d(s,u) + \text{cost}(u,v) < d(s,v) \quad ↯ \end{aligned}$$

- We can calculate initial $\pi(v)$ using Bellman Ford
  - If G contains no negative edges, we can initialize with $\pi(v) = 0$
- How to update π?
  - Use distances d' calculated by Dijkstra in current iteration
  - Dijkstra stops once distance to t is known but luckily we only need to update π for nodes that are done by this point
  - for nodes permanently labeled by Dijkstra:
    $\pi'(v) = \pi(v) + d'(s, v) - d'(s, t)$

No Proof :)

SSP with Dijkstra

SSP with Dijkstra:

- start with null flow f
- calculate π with Bellman Ford
- **while** s and t connected in G_f **do**
  - run Dijkstra on G_f
  - augment f with shortest s-t-path
  - update π

SSP with Dijkstra

- Runtime: $O((n+m)\log n \cdot B)$

SSP with Dijkstra:

- start with null flow f
- calculate π with Bellman Ford
- **while** s and t connected in G_f **do**
  - run Dijkstra on G_f
  - augment f with shortest s-t-path
  - update π

SSP with Dijkstra

- Runtime: $O((n+m)\log n \cdot B)$
  - (using “normal” heaps)

SSP with Dijkstra:

- start with null flow f
- calculate π with Bellman Ford
- **while** s and t connected in G_f **do**
  - run Dijkstra on G_f
  - augment f with shortest s-t-path
  - update π

SSP with Dijkstra

- Runtime: $O((n+m)\log n \cdot B)$
  - (using “normal” heaps)
  - Still pseudopolynomial

SSP with Dijkstra:

- start with null flow f
- calculate π with Bellman Ford
- **while** s and t connected in G_f **do**
  - run Dijkstra on G_f
  - augment f with shortest s-t-path
  - update π

SSP with Dijkstra

- Runtime: $O((n+m)\log n \cdot B)$
  - (using “normal” heaps)
  - Still pseudopolynomial

SSP with Dijkstra:

- start with null flow f
- calculate π with Bellman Ford
- **while** s and t connected in G_f **do**
  - run Dijkstra on G_f
  - augment f with shortest s-t-path
  - update π

https://arxiv.org/abs/1207.6381

Network Flows: Theory, Algorithms, and Applications

Implementation: Basics

```
struct Edge {
    int from, to, flow, cap, cost;
    Edge* rev = nullptr;
};

vector<vector<Edge*>> adj;

void add_edge(int a, int b, int cap, int cost) {
    auto e1 = new Edge{a, b, 0, cap, cost};
    auto e2 = new Edge{b, a, 0, 0, -cost};
    e1->rev = e2;
    e2->rev = e1;
    adj[a].push_back(e1);
    adj[b].push_back(e2);
}
```

Implementation: Basics

```
struct Edge {
    int from, to, flow, cap, cost;
    Edge* rev = nullptr;
};

vector<vector<Edge*>> adj;

void add_edge(int a, int b, int cap, int cost) {
    auto e1 = new Edge{a, b, 0, cap, cost};
    auto e2 = new Edge{b, a, 0, 0, -cost};
    e1->rev = e2;
    e2->rev = e1;
    adj[a].push_back(e1);
    adj[b].push_back(e2);
}
```

Implementation: Basics

```
struct Edge {
    int from, to, flow, cap, cost;
    Edge* rev = nullptr;
};

vector<vector<Edge*>> adj;

void add_edge(int a, int b, int cap, int cost) {
    auto e1 = new Edge{a, b, 0, cap, cost};
    auto e2 = new Edge{b, a, 0, 0, -cost};
    e1->rev = e2;
    e2->rev = e1;
    adj[a].push_back(e1);
    adj[b].push_back(e2);
}
```

always directed

Implementation: Bellman Ford

DON'T RELAX ALL EDGES $n-1$ TIMES!!!

Implementation: Bellman Ford

DON'T RELAX ALL EDGES $n-1$ TIMES!!!

- use queue to track nodes that had their distance reduced

Implementation: Bellman Ford

DON'T RELAX ALL EDGES $n-1$ TIMES!!!

- use queue to track nodes that had their distance reduced
- don't add nodes into the queue multiple times

Implementation: Bellman Ford

DON'T RELAX ALL EDGES $n-1$ TIMES!!!

- use queue to track nodes that had their distance reduced
- don't add nodes into the queue multiple times
- track previous edge rather than previous node to support multi-graphs

Implementation: Bellman Ford

DON'T RELAX ALL EDGES $n-1$ TIMES!!!

- use queue to track nodes that had their distance reduced
- don't add nodes into the queue multiple times
- track previous edge rather than previous node to support multi-graphs
- implementation should look similar to dijkstra but with queue instead of PQ

Implementation: Bellman Ford

DON'T RELAX ALL EDGES $n-1$ TIMES!!!

- use queue to track nodes that had their distance reduced
- don't add nodes into the queue multiple times
- track previous edge rather than previous node to support multi-graphs
- implementation should look similar to dijkstra but with queue instead of PQ

```
while q not empty:
    v = q.pop(); in_q[v] = false
    for edge in adj[v]:
        if new_dist >= dist[nei]: continue
        dist[nei] = new_dist; prev[nei] = edge
        if not in_q[nei]:
            q.push(nei); in_q[nei] = true
```

Implementation: Bellman Ford

DON'T RELAX ALL EDGES $n-1$ TIMES!!!

- use queue to track nodes that had their distance reduced
- don't add nodes into the queue multiple times
- track previous edge rather than previous node to support multi-graphs
- implementation should look similar to dijkstra but with queue instead of PQ

Shortest Path Faster Algorithm:

```
while q not empty:
    v = q.pop(); in_q[v] = false
    for edge in adj[v]:
        if new_dist >= dist[nei]: continue
        dist[nei] = new_dist; prev[nei] = edge
        if not in_q[nei]:
            q.push(nei); in_q[nei] = true
```

Implementation: Bellman Ford

DON'T RELAX ALL EDGES $n-1$ TIMES!!!

- use queue to track nodes that had their distance reduced
- don't add nodes into the queue multiple times
- track previous edge rather than previous node to support multi-graphs
- implementation should look similar to dijkstra but with queue instead of PQ

Shortest Path Faster Algorithm:

```
while q not empty:
    v = q.pop(); in_q[v] = false
    for edge in adj[v]:
        if new_dist >= dist[nei]: continue
        dist[nei] = new_dist; prev[nei] = edge
        if not in_q[nei]:
            q.push(nei); in_q[nei] = true
```

How to find negative cycles with this approach?

Implementation: Misc.

Cost calculation

```
int cost = 0;
for (auto& row : adj)
    for (auto e : row)
        cost += e->flow * e->cost;
cost /= 2;
```

Implementation: Misc.

Cost calculation

```
int cost = 0;
for (auto& row : adj)
    for (auto e : row)
        cost += e->flow * e->cost;
cost /= 2;
```

Why does this work?

Implementation: Misc.

Cost calculation

```
int cost = 0;
for (auto& row : adj)
    for (auto e : row)
        cost += e->flow * e->cost;
cost /= 2;
```

Why does this work?

Be wary of overflows!

MinCostFlow Algorithm Summary

Cycle Cancelling	$O(nm^2UC)$
Successive Shortest Path (SSP)	$O(nmB)$
SSP with Dijkstra	$O((n+m)\log n \cdot B)$

U: upper bound on capacity, C: upper bound on absolute cost, B: value of resulting flow

For more rigorous proofs, see: https://courses.engr.illinois.edu/cs473/sp2017/notes/G-mincostflow.pdf

MinCostFlow Algorithm Summary

Cycle Cancelling	$O(nm^2UC)$
Successive Shortest Path (SSP)	$O(nmB)$
SSP with Dijkstra	$O((n+m)\log n \cdot B)$

When to use Cycle Cancelling?

U: upper bound on capacity, C: upper bound on absolute cost, B: value of resulting flow

For more rigorous proofs, see: https://courses.engr.illinois.edu/cs473/sp2017/notes/G-mincostflow.pdf

MinCostFlow Algorithm Summary

Cycle Cancelling	$O(nm^2UC)$
Successive Shortest Path (SSP)	$O(nmB)$
SSP with Dijkstra	$O((n+m)\log n \cdot B)$
Minimum-Mean Cycle Cancelling	$O(nm^2 \log n)$
Orlin's Algorithm	$O(m \log n(m + n \log n))$

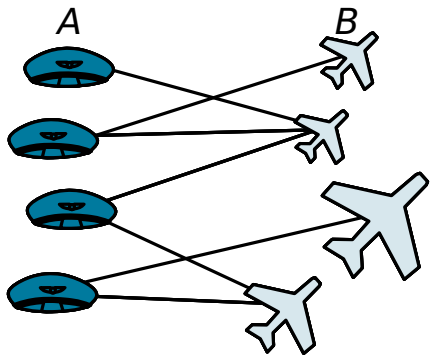
When to use Cycle Cancelling?

U: upper bound on capacity, C: upper bound on absolute cost, B: value of resulting flow

For more rigorous proofs, see: https://courses.engr.illinois.edu/cs473/sp2017/notes/G-mincostflow.pdf

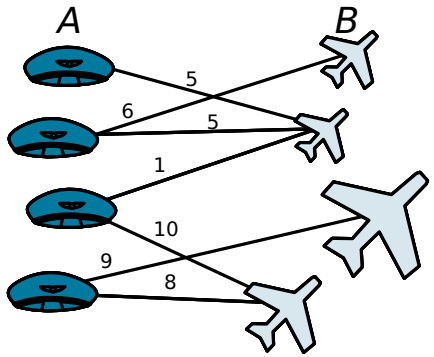
Flow Modelling Techniques

- remember that max-flow solved max. bipartite matching



Flow Modelling Techniques

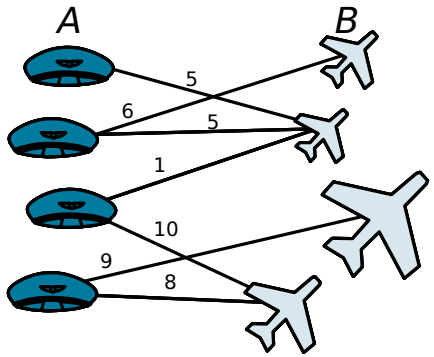
- remember that max-flow solved max. bipartite matching
- min-cost-max-flow solves minimum weight maximum bipartatite matching



Flow Modelling Techniques

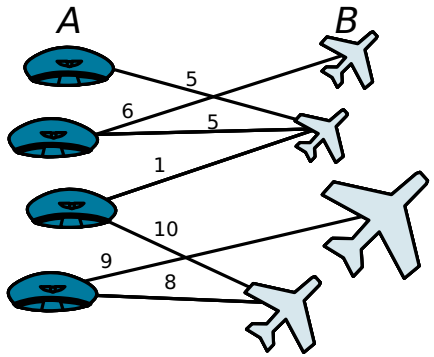
- remember that max-flow solved max. bipartite matching
- min-cost-max-flow solves minimum weight maximum bipartatite matching

What if I want to maximize cost instead?



Flow Modelling Techniques

- remember that max-flow solved max. bipartite matching
- min-cost-max-flow solves minimum weight maximum bipartatite matching

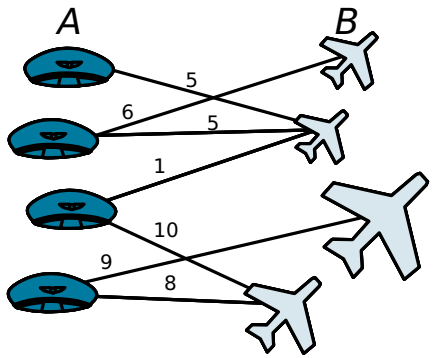


What if I want to maximize cost instead?

What if I don't want to maximize the matching size but have a budget?

Flow Modelling Techniques

- remember that max-flow solved max. bipartite matching
- min-cost-max-flow solves minimum weight maximum bipartatite matching



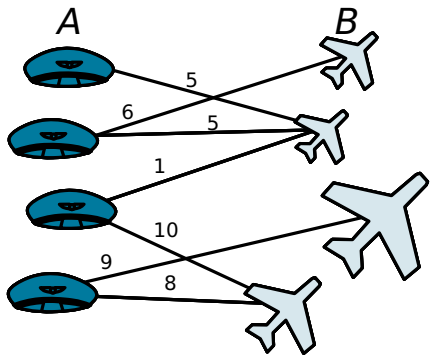
What if I want to maximize cost instead?

What if I don't want to maximize the matching size but have a budget?

What if I can pay pilots overtime to fly one more plane but it gets more expensive each time?

Flow Modelling Techniques

- remember that max-flow solved max. bipartite matching
- min-cost-max-flow solves minimum weight maximum bipartatite matching



What if I want to maximize cost instead?

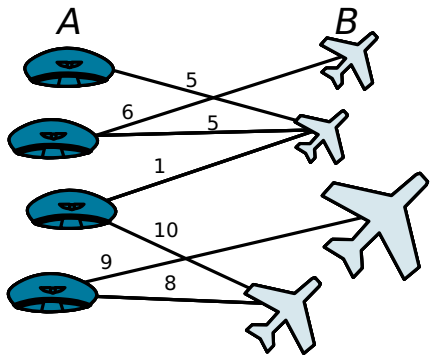
What if I don't want to maximize the matching size but have a budget?

What if I can pay pilots overtime to fly one more plane but it gets more expensive each time?

- Increasing cost on edges (E.g.: first use of edge is free)

Flow Modelling Techniques

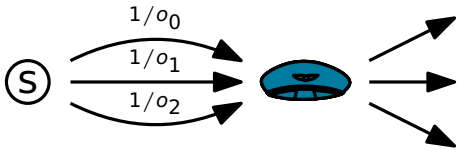
- remember that max-flow solved max. bipartite matching
- min-cost-max-flow solves minimum weight maximum bipartatite matching



What if I want to maximize cost instead?

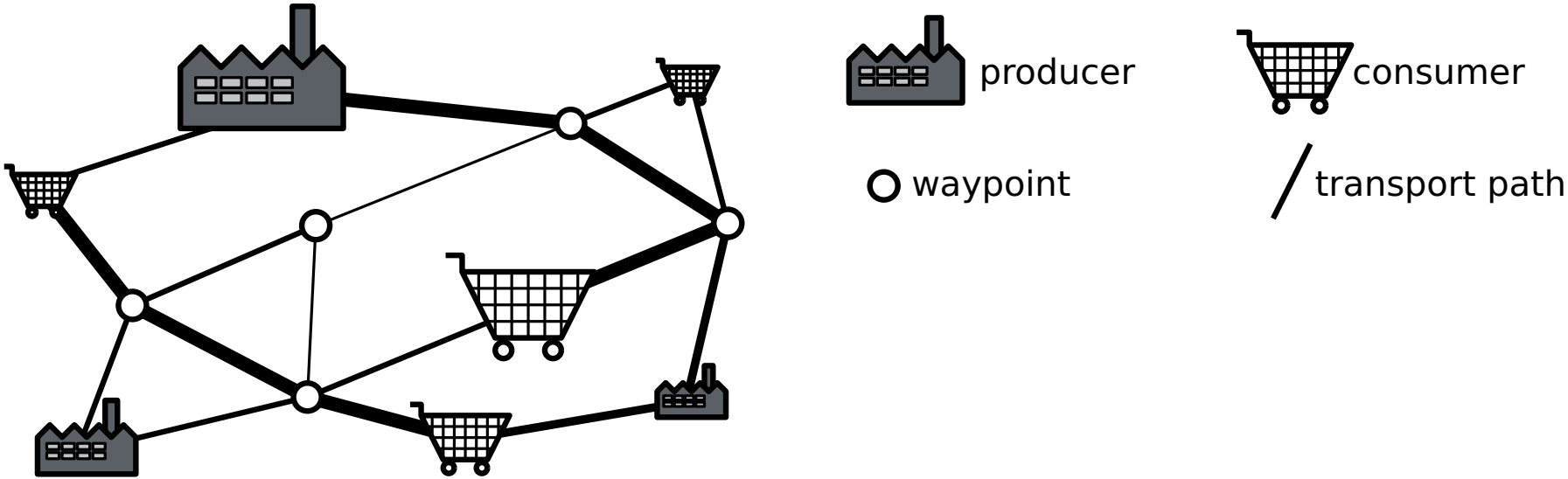
What if I don't want to maximize the matching size but have a budget?

What if I can pay pilots overtime to fly one more plane but it gets more expensive each time?

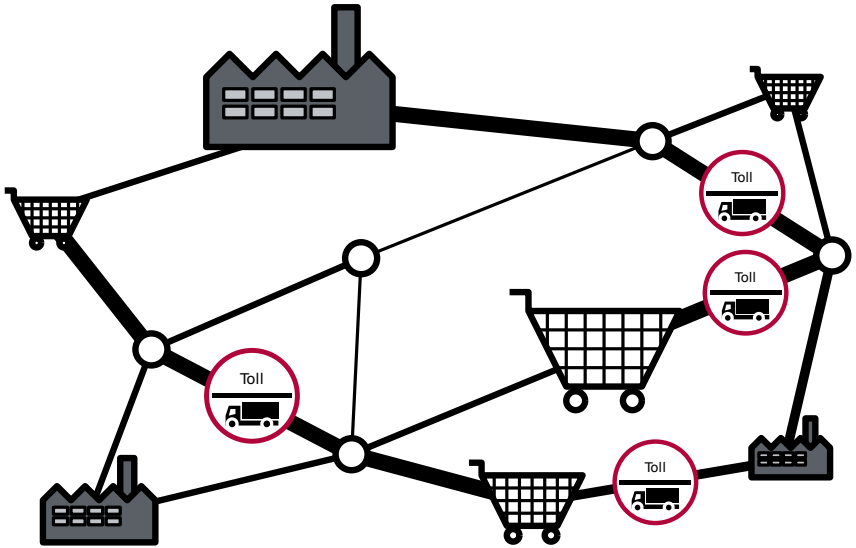


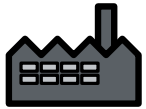
- Increasing cost on edges (E.g.: first use of edge is free)

Flow Modelling Techniques



Flow Modelling Techniques




producer

consumer

waypoint

transport path

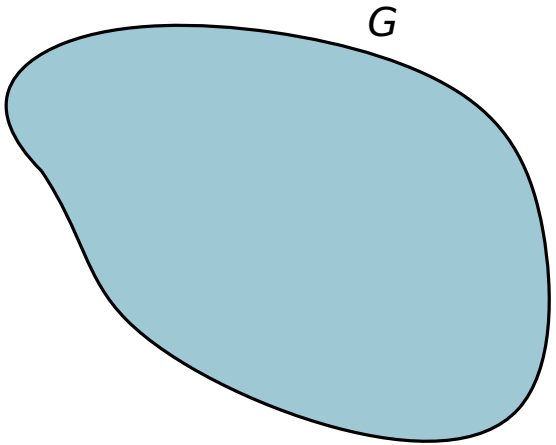
cost for using a transport path

Flow Modelling Techniques

- Handling multiple sources and/or sinks

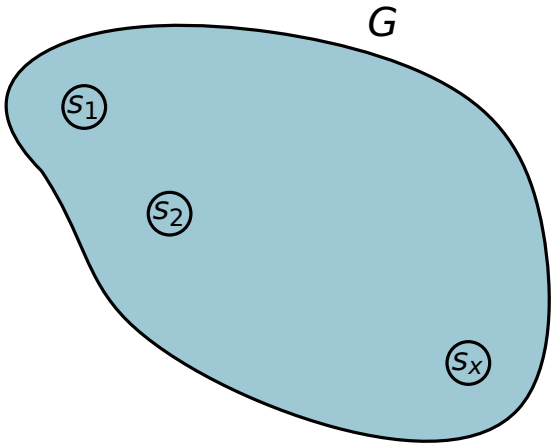
Flow Modelling Techniques

- Handling multiple sources and/or sinks



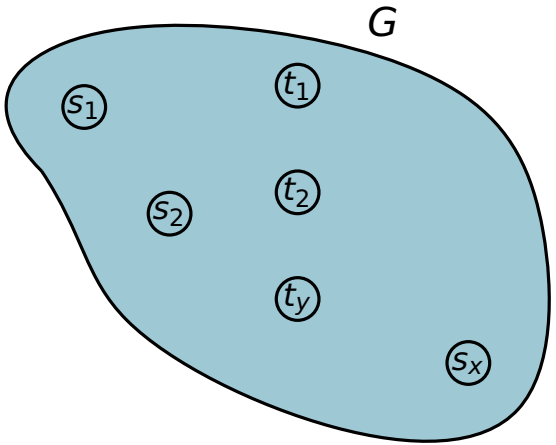
Flow Modelling Techniques

- Handling multiple sources and/or sinks



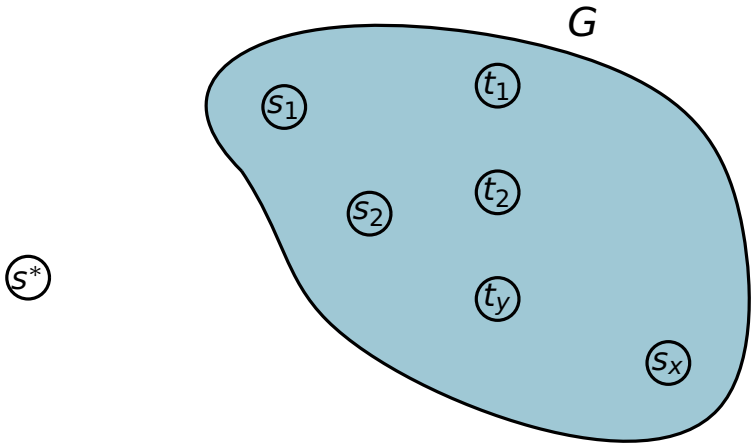
Flow Modelling Techniques

- Handling multiple sources and/or sinks



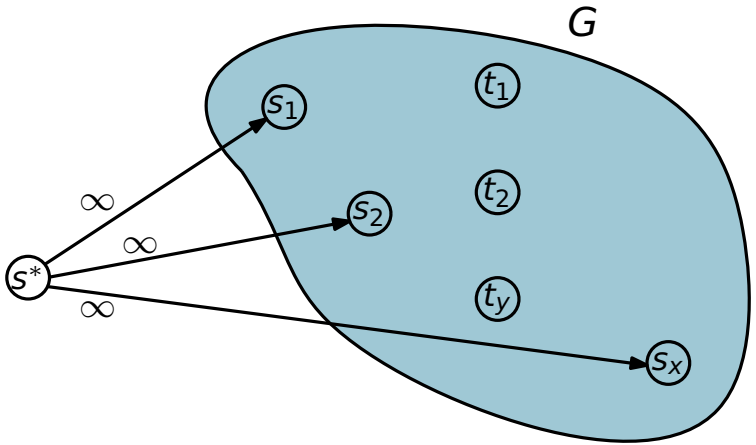
Flow Modelling Techniques

- Handling multiple sources and/or sinks



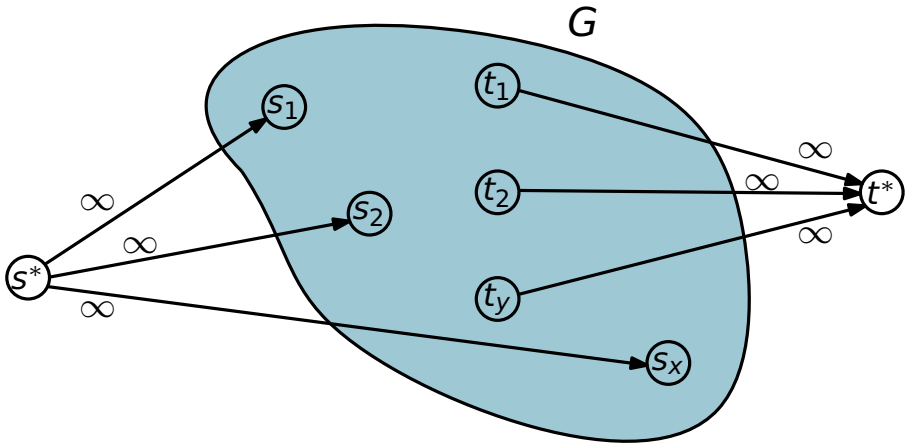
Flow Modelling Techniques

- Handling multiple sources and/or sinks



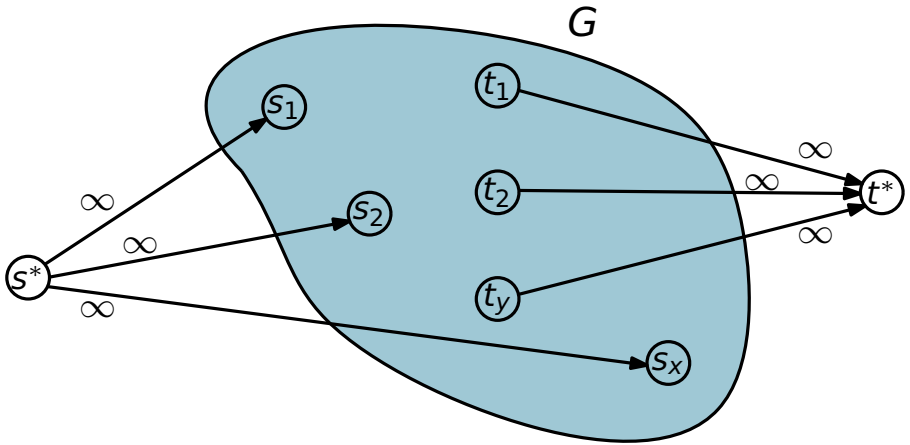
Flow Modelling Techniques

- Handling multiple sources and/or sinks



Flow Modelling Techniques

- Handling multiple sources and/or sinks



Calculate s^*-t^*-flow in adapted graph

Flow Modelling Techniques

- Flow with demand d

Flow Modelling Techniques

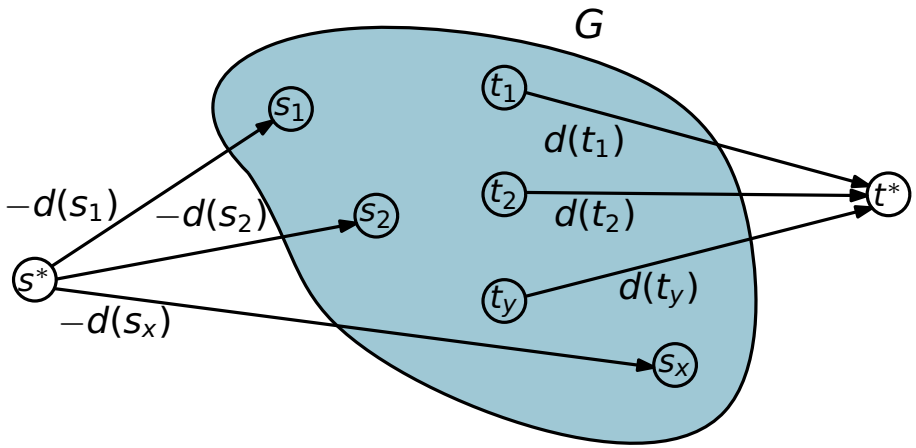
- Flow with demand d
  - Each node either demands flow ($d(v) \geq 0$) or provides it ($d(v) < 0$)

Flow Modelling Techniques

- Flow with demand d
  - Each node either demands flow ($d(v) \geq 0$) or provides it ($d(v) < 0$)
  - Consider each node as either a source with limited supply, or a sink with limited demand

Flow Modelling Techniques

- Flow with demand d
  - Each node either demands flow ($d(v) \geq 0$) or provides it ($d(v) < 0$)
  - Consider each node as either a source with limited supply, or a sink with limited demand
  - Use construction from previous slide:

Flow Modelling Techniques

- Flow with demand d
  - Each node either demands flow ($d(v) \geq 0$) or provides it ($d(v) < 0$)
  - Consider each node as either a source with limited supply, or a sink with limited demand
  - Use construction from previous slide: