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 $f : E \mapsto \mathbb{Z}$ with...
Reminder: Flows

undirected network $G = (V, E)$
a source and a sink $s, t \in V$
a capacity function $c : E \rightarrow \mathbb{N}$
a flow $f : E \rightarrow \mathbb{Z}$ 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 \( f : E \mapsto \mathbb{Z} \) 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 $c : E \rightarrow \mathbb{N}$
a flow $f : E \rightarrow \mathbb{Z}$ with...

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

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

- conservation of flow
  
  $$\forall v \in V \setminus \{s, t\} :$$
  $$\sum_{\{u, v\} \in E} f(u, v) - f(v, u) = 0$$

\[ \begin{array}{ccc}
\text{s} & \overset{2/8}{\xrightarrow{3/3}} & \overset{2/3}{\xrightarrow{4/4}} \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & 
\end{array} \]
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 $f : E \rightarrow \mathbb{Z}$ 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 $f : E \mapsto \mathbb{Z}$ with...

- consistency
  $f(u, v) = -f(v, u)$
- capacity constraint
  $f(u, v) \leq c(\{u, v\})$
- conservation of flow $\forall v \in V \setminus \{s, t\}$:
  $\sum_{\{u,v\} \in E} f(u, v) - f(v, u) = 0$

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

$f(s, v) - f(v, s) = 5 - (-5) \neq 0$
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 $f : E \rightarrow \mathbb{Z}$ with...

- consistency
  \[ f(u, v) = -f(v, u) \]
- capacity constraint
  \[ f(u, v) \leq c({u, v}) \]
- conservation of flow \( \forall v \in V \setminus \{s, t\} : \)
  \[ \sum_{\{u, v\} \in E} f(u, v) - f(v, u) = 0 \]
- 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

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

Ford-Fulkerson MaxFlow Algorithm

```c
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

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

**Ford-Fulkerson MaxFlow Algorithm**

- $\Theta(1)$ times
- $\Theta(m)$
- $\Theta(1)$
- $O(m)$
- $O(m)$

$C$ is value of MaxFlow
Ford-Fulkerson

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

Ford-Fulkerson MaxFlow Algorithm

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

$s$, $t$, $A$, $B$
Ford-Fulkerson

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

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

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

Ford-Fulkerson MaxFlow Algorithm

- $\Theta(1)$
- $\leq C$ times
- $\Theta(m)$
- $O(m)$
- $O(m)$

$C$ is value of MaxFlow

$\rightarrow$ max-flow $\geq$ min-cut
Ford-Fulkerson

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

```
int flow = 0;
while (true) {
    // find augmenting path
    // break if no s-t path found
    // find path bottleneck
    // 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) $\rightarrow \text{max-flow} \geq \text{min-cut}$
- every s-t cut limits the maximum flow $\rightarrow \text{max-flow} \leq \text{min-cut}$
Ford-Fulkerson

- Idea: find augmenting path, augment, repeat
- Running time: $O(mC)$
- $C$ is value of MaxFlow
- Correctness:
  - 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 MaxFlow Algorithm

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

$\Theta(1)$ times

$O(mC)$
Ford-Fulkerson Implementation

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

```cpp
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

```c
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;
        }
    }
}
if (!inc[t]) break;

// 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;
        }
    }
}
if (!inc[t]) break;
/ 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;

// 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;
// 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

\[ O(nm^2) \]
// 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);

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

```c
// 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

```c
// 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
**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
**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

Diagram:

- Nodes $s$, $t$, $A$, and $B$
- $s$ is reachable from $t$ in the residual network
- $t$ is not in the partition of $s$
**Problem** extract edges in MinCut from given MaxFlow

- reachability from s in residual network partitions nodes
- t is in other partition than s
  \[ \rightarrow \text{otherwise flow not maximal} \]
- 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
**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 Diagram](image-url)
**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
**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?
```

![Graph Diagram]

```
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:

![Graph Diagram]

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 $\infty$

![Diagram of a directed graph with vertex weights and edges]

-1 → 8
5 → -2
3 → -4

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 $\infty$
- new node $s$ connects to vertices with $w_i > 0$ using $w_i$ as edge weight

![Graph](image.png)
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 $\infty$
- 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

![Graph Diagram]
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 $\infty$
- 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 $C$. 

![Graph with nodes and edges](image)
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 \( \infty \)
- 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 $\infty$.
- 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?

Reduction to 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.

**Reduction** to MinCut:

- all old edges weight $\infty$
- 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 $\in C$
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.

![Diagram showing flow decomposition](image-url)
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

```
s  7/8  3/3  6/6
  ▸  ▸  ▸
  7/7 ▸  ▸
    ▸  ▸
  t
```

```
s
  ▸  ▸
  4/4
  ▸  ▸
  t
```

```
s
  ▸  ▸
  3/3
  ▸  ▸
  t
```

```
s
  ▸  ▸
  3/3
  ▸  ▸
  t
```

```
s
  ▸  ▸
  3/3
  ▸  ▸
  t
```
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:

- while there is s-t flow do
  - $\leq m$ times
    - $\Theta(m)$
    - $O(n)$
  - 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?

Flow Decomposition Algorithm:

- **while** there is s-t flow do
  - $\leq m$ times
  - $\Theta(m)$
  - $O(n)$
  - 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 adjacency list

**Flow Decomposition Algorithm**:

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

$\leq m$ times

$\Theta(m)$

$O(n)$
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 adjacency list

**Flow Decomposition Algorithm:**

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

$\leq m$ times

$\Theta(m)$

$O(n)$
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 adjacency list

Flow Decomposition Algorithm:

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

---

Flow Decomposition Algorithm:

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

---

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 adjacency list

- maintaining the next edge for each node takes $O(m)$ time over all graph traversals combined

**Flow Decomposition Algorithm:**

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

Running time $\leq m$ times $\Theta(m) \in \Theta(n)$
Flow Decomposition

- **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 adjacency 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 → \( O(n) \) instead of \( O(m) \)

**Flow Decomposition Algorithm**:

- while there is s-t flow do
  - find s-t path \( P \) with flow
  - remove flow from \( P \)

\[
\text{running time } O(m^2)
\]

\( \leq m \text{ times} \)

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

* amortized

\( \Theta(n) \)

\( O(n) \)
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 adjacency 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
  - find s-t path $P$ with flow
  - remove flow from $P$

$\Theta(n) \leq m$ times

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

* amortized

$O(n)$
**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 $d(u) + 1 = d(v)$.
**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 $d(u) + 1 = d(v)$.
**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 $d(u) + 1 = d(v)$.
**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 $d(u) + 1 = d(v)$.
**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

- same approach as Ford-Fulkerson

```plaintext
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

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

Ford-Fulkerson 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 MaxFlow Algorithm
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

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

Dinic’s MaxFlow Algorithm

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

\[\Theta(m)\]

\[\Theta(1)\]

\[\Theta(m)\]

\[O(n)\]
Dinic’s

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

Ford-Fulkerson MaxFlow Algorithm

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

Dinic’s MaxFlow Algorithm

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

\[
\Theta(m) \leq m \times \Theta(m) = O(n)
\]
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

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

Ford-Fulkerson MaxFlow Algorithm

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

Dinic’s MaxFlow Algorithm

```c
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
- **Lemma**: each time we augment blocking flow in layered network $d(t)$ increases

### Ford-Fulkerson MaxFlow Algorithm

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

### Dinic’s MaxFlow Algorithm

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

\[
\begin{align*}
\text{while} \quad \text{true} & \quad \{ \\
\text{\quad} & \text{// find augmenting path} \\
\text{\quad} & \text{// break if no s-t path found} \\
\text{\quad} & \text{// find path bottleneck} \\
\text{\quad} & \text{// augment path} \\
\text{\} }
\end{align*}
\]

\[
\begin{align*}
\leq n \text{ times} \\
\Theta(m) \\
\Theta(1) \\
\leq m \text{ times} \\
\Theta(m) \\
O(n)
\end{align*}
\]
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**

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

**Dinic’s MaxFlow Algorithm**

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

$\leq n$ times

$\Theta(m)$

$\Theta(1)$

$\leq m$ times

$\Theta(m)$

$O(n)$
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

* amortized

### Ford-Fulkerson MaxFlow Algorithm

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

### Dinic’s MaxFlow Algorithm

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

$\leq n$ times

$\Theta(m)$

$\Theta(1)$

$\leq m$ times (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
  → Hopcroft-Karp

Ford-Fulkerson MaxFlow Algorithm

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

Dinic’s MaxFlow Algorithm

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

* amortized

$\leq n$ times
$\Theta(m)$
$\Theta(1)$
$\leq m$ times
$O(n)$
Dinic’s Implementation

```cpp
// 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

```cpp
// 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

```cpp
// 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

```cpp
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;
}
```

augments some s-t path
and returns how much flow reached t
Dinic’s Implementation

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

```c++
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;
}
```

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

```c
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;
}
```

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

```cpp
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;
}
```

v is current node
aug is how much flow can be pushed up to this point
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

```cpp
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;
}
```

`v` is current node `aug` is how much flow can be pushed up to this point

Note that `i` is a reference
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

```cpp
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;
}
```

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

```cpp
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;
}
```

v is current node
aug is how much flow can be pushed up to this point
note that i is a reference
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

```cpp
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;
}
```

v is current node
aug is how much flow can be pushed up to this point
note that i is a reference
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

```cpp
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;
}
```

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

```cpp
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;
}
```

v is current node
aug is how much flow can be pushed up to this point
note that i is a reference
Dinic’s Implementation

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

```cpp
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

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ford-Fulkerson</td>
<td>$O(mC)$</td>
</tr>
<tr>
<td>Edmonds-Karp</td>
<td>$O(m^2n)$</td>
</tr>
<tr>
<td>Dinic’s</td>
<td>$O(mn^2)$</td>
</tr>
</tbody>
</table>

$C$ is value of maximum flow
MaxFlow Algorithms Summary

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ford-Fulkerson</td>
<td>$O(mC)$</td>
</tr>
<tr>
<td>Edmonds-Karp</td>
<td>$O(m^2n)$</td>
</tr>
<tr>
<td>Dinic’s</td>
<td>$O(mn^2)$ $O(m\sqrt{n})$ for unit graphs</td>
</tr>
</tbody>
</table>

When is Ford-Fulkerson better than Dinic’s?

$C$ is value of maximum flow
## MaxFlow Algorithms Summary

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ford-Fulkerson</td>
<td>$O(mC)$</td>
</tr>
<tr>
<td>Edmonds-Karp</td>
<td>$O(m^2n)$</td>
</tr>
<tr>
<td>Dinic’s</td>
<td>$O(mn^2)$</td>
</tr>
<tr>
<td>MPM</td>
<td>$O(n^3)$</td>
</tr>
<tr>
<td>F-F + Capacity Scaling</td>
<td>$O(m^2 \log U)$</td>
</tr>
<tr>
<td>Push-Relabel</td>
<td>$O(mn^2)$</td>
</tr>
<tr>
<td>FIFO Push-Relabel</td>
<td>$O(n^3)$</td>
</tr>
<tr>
<td>highest label P-R</td>
<td>$O(nm + n^2 \sqrt{m})$</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ford-Fulkerson</td>
<td>$O(mC)$</td>
</tr>
<tr>
<td>Edmonds-Karp</td>
<td>$O(m^2n)$</td>
</tr>
<tr>
<td>Dinic’s</td>
<td>$O(mn^2)$</td>
</tr>
<tr>
<td></td>
<td>$O(m\sqrt{n})$ for unit graphs</td>
</tr>
<tr>
<td>MPM</td>
<td>$O(n^3)$</td>
</tr>
<tr>
<td>F-F + Capacity Scaling</td>
<td>$O(m^2 \log U)$</td>
</tr>
<tr>
<td>Push-Relabel</td>
<td>$O(mn^2)$</td>
</tr>
<tr>
<td>FIFO Push-Relabel</td>
<td>$O(n^3)$</td>
</tr>
<tr>
<td>highest label P-R</td>
<td>$O(nm + n^2 \sqrt{m})$</td>
</tr>
</tbody>
</table>

- **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:

```c
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;
    }
}
```

O(m√n) for unit graphs

- Ford-Fulkerson
- Edmonds-Karp
- F-F + Capacity Scaling
- Dinic's
- MPM

O(m2n)

- U is highest capacity edge outgoing from source
- C is value of maximum flow

O(m2 log U)

- U is highest capacity edge outgoing from source

O(mC)

O(mn2)

O(n3)

When is Ford-Fulkerson better than Dinic’s?

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

Edmonds-Karp

Minimum flow

Highest label P-R

Maximum flow

Highest capacity edge outgoing from source
# MaxFlow Algorithms Summary

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ford-Fulkerson</td>
<td>$O(mC)$</td>
</tr>
<tr>
<td>Edmonds-Karp</td>
<td>$O(m^2n)$</td>
</tr>
<tr>
<td>Dinic’s</td>
<td>$O(mn^2)$</td>
</tr>
<tr>
<td></td>
<td>$O(m\sqrt{n})$ for unit graphs</td>
</tr>
<tr>
<td>MPM</td>
<td>$O(n^3)$</td>
</tr>
<tr>
<td>F-F + Capacity Scaling</td>
<td>$O(m^2 \log U)$</td>
</tr>
<tr>
<td>Push-Relabel</td>
<td>$O(mn^2)$</td>
</tr>
<tr>
<td>FIFO Push-Relabel</td>
<td>$O(n^3)$</td>
</tr>
<tr>
<td>highest label P-R</td>
<td>$O(nm + n^2 \sqrt{m})$</td>
</tr>
</tbody>
</table>

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
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

$1 \leq n \leq 10^9$

$0 \leq k \leq 50$
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.

$1 \leq n \leq 10^9$

$0 \leq k \leq 50$

König’s Theorem

in bipartite graphs, min vertex cover = max matching
MinCostFlows

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

![Diagram of a graph with labels for flow and cost](image)
MinCostFlows

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

Syntax: cap/cost
MinCostFlows

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

```
s   5/3   t
    ^     |
   5/1   2/1
       3/0
```
MinCostFlows

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

![Diagram]

Syntax: cap/cost

Cost is per unit of flow!

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

- Flow with additional cost function $\text{cost} : E \to \mathbb{Z}$
- 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 $G_f$: $\text{cost}_{G_f}(v,u) = -\text{cost}_{G_f}(u,v)$

Cost is per unit of flow!

In $G_f$: $\text{cost}_{G_f}(v,u) = -\text{cost}_{G_f}(u,v)$

Syntax: cap/cost
MinCostFlows

- Flow with additional cost function cost : \( E \rightarrow \mathbb{Z} \)
- 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:
MinCostFlows

- Flow with additional cost function \( \text{cost} : E \rightarrow \mathbb{Z} \)
- 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

Syntax: cap/cost

\[
\text{cost}(f) = \sum_{(u,v) \in E} f(u,v) \cdot \text{cost}(u,v)
\]

Example diagrams with flow and costs.
**MinCostFlows**

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

Syntax: cap/cost

Example graph:

- $s$ to $t$ with costs:
  - $s$ to $v$: $5/3$
  - $v$ to $t$: $2/1$
  - $s$ to $t$: $3/0$

Cost of flow $f$:

$$\text{cost}(f) = \sum_{(u,v) \in E} f(u, v) \cdot \text{cost}(u, v) = 8$$
MinCostFlows

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

![Diagram](syntax: cap/cost)

Cost($f$) = 8
Negative Cost Cycles

cost(f) = 8
Negative Cost Cycles

\[
\begin{align*}
\text{cost}(f) &= 8 \\
\text{cost reduced by } 1
\end{align*}
\]

\[
\begin{align*}
\text{cost}(f) &= 7 \\
\text{cost reduced by } 1
\end{align*}
\]
Negative Cost Cycles

![Diagram of negative cost cycles]

- Initial graph with cost $f = 8$
- $s \rightarrow t$ with edges $s \rightarrow t: 5/1$, $t \rightarrow s: 2/1$
- $s \rightarrow t$: cost reduced by 1

- Final graph with cost $f = 7$
- $s \rightarrow t$ with edges $s \rightarrow t: 2/1$, $t \rightarrow s: 5/1$
- $s \rightarrow t$: cost reduced by 1

- Graph with self-loops showing negative cost cycles
- $s \rightarrow s: 4/1$ and $t \rightarrow t: 1/1$
- $s \rightarrow t$ with edges $s \rightarrow t: 3/3$, $t \rightarrow s: 2/-3$
Negative Cost Cycles

Negative cost cycle in $G_f$!
Negative Cost Cycles

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:

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

$n$ needs proof, $f_P$ obvious

**Proof sketch**
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.

Proof sketch

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

$f_p$: 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.

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 $G_f$.
**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.

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.

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.
**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.

*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 $\text{cost}(P \setminus C) - \text{cost}(C \cap P)$.

$f_p$: one unit of flow along $P$. 

$G_{f'}$: Graph with cost-minimal $(k + 1)$-flow $f'$. 

$C \setminus P$: Path $P$ excluding vertices and edges of cycle $C$. 

$C \cap P$: Intersection of path $P$ and cycle $C$. 

$s$: Source vertex. 

$t$: Sink vertex. 

Path $P$: Shortest $s$-$t$-path in $G_f$. 

Cycle $C$: Negative cost cycle in $G_{f'}$. 

Diagram: Graph $G_{f'}$ with nodes $s$, $t$, and edge $P$ connecting them.
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.

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

**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 \( G_f \)!
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.

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 \( G_f \)!
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 $P$

Syntax: cap/cost

![Graph](image)
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)

Syntax: cap/cost

```
s    2/1    1/1    1/1    2/1    1/1
  v   2/1
  v   1/1
  v   1/1
    t
```

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)

Syntax: cap/cost

```
s     2/-1
2/-1  1/1
1/1   1/1
     2/-1
     1/1
     1/1
  t
```

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 $P$
```

Syntax: cap/cost

```
s 2/-1 2/-1 2/-1 t
 1/1 1/1 1/1
```

Runtime: $O(nmB)$ (B: value of resulting flow)
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)

Syntax: cap/cost

```
while $s$ and $t$ connected in $G_f$ do
    find shortest $s$-$t$-path $P$ in $G_f$
    augment $f$ with $P$
```

E.g. if $G$ does not contain negative cycles $G_f$ may contain negative edges

Runtime: $O(nmB)$ ($B$: value of resulting flow)

Use Bellman Ford
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 $P$

Syntax: cap/cost

![Diagram of a directed graph showing the successive shortest path algorithm.](image-url)
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)

### Syntax:
- cap/cost

### Example Graph:

```
s
```
```
\[ \text{2/-1} \]
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```

No $s$-$t$-path → done!

### 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?

- syntax: cap/cost
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 $P$

**Syntax:** cap/cost

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:
    - $w(C) = \pi(c_1) + \text{cost}(c_1, c_2) - \pi(c_2) + \pi(c_2) + \text{cost}(c_2, c_3) - \pi(c_3) + \pi(c_3) + \ldots$
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:
  - $w(C) = \pi(c_1) + \text{cost}(c_1, c_2) - \pi(c_2) + \pi(c_2) + \text{cost}(c_2, c_3) - \pi(c_3) + \pi(c_3) + \ldots = \text{cost}(C)$
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:
    - $w(C) = \pi(c_1) + \text{cost}(c_1, c_2) - \pi(c_2) + \text{cost}(c_2, c_3) - \pi(c_3) + \ldots = \text{cost}(C)$
    - Still no negative cycles!
**SSP with Dijkstra**

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

\[
\begin{align*}
&d(s, u) + \text{cost}(u, v) - d(s, v) < 0 \\
\Rightarrow &d(s, u) + \text{cost}(u, v) < d(s, v)
\end{align*}
\]
SSP with Dijkstra

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

  - Proof:
    
    $$d(s, u) + \text{cost}(u, v) - d(s, v) < 0$$
    
    $$\Rightarrow d(s, u) + \text{cost}(u, v) < d(s, v)$$

- 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:
    
    $$d(s, u) + \text{cost}(u, v) - d(s, v) < 0$$
    $$\Rightarrow d(s, u) + \text{cost}(u, v) < d(s, v)$$

- 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:
    
    $d(s, u) + \text{cost}(u, v) - d(s, v) < 0$
    
    $\Rightarrow d(s, u) + \text{cost}(u, v) < d(s, v)$

- 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 $\pi$?
SSP with Dijkstra

- Initially we set $\pi(v) = d(s, v)$. Then $w(u, v) \geq 0$ for every $(u, v) \in E$
  - Proof:
    $$d(s, u) + \text{cost}(u, v) - d(s, v) < 0$$
    $$\Rightarrow \quad d(s, u) + \text{cost}(u, v) < d(s, v)$$

- 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 $\pi$?
  - 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:
    $$d(s, u) + \text{cost}(u, v) - d(s, v) < 0$$
    $$\Rightarrow d(s, u) + \text{cost}(u, v) < d(s, v)$$

- 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 $\pi$?
  - Use distances $d'$ calculated by Dijkstra in current iteration
  - Dijkstra stops once distance to $t$ is known but luckily we only need to update $\pi$ 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:
    \[
    d(s, u) + \text{cost}(u, v) - d(s, v) < 0
    \Rightarrow
    d(s, u) + \text{cost}(u, v) < d(s, v)
    \]

- 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 $\pi$?
  - Use distances $d'$ calculated by Dijkstra in current iteration.
  - Dijkstra stops once distance to $t$ is known but luckily we only need to update $\pi$ 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 $\pi$ 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 $\pi$
SSP with Dijkstra

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

SSP with Dijkstra:

- start with null flow $f$
- calculate $\pi$ 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 $\pi$
SSP with Dijkstra

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

SSP with Dijkstra:
- start with null flow $f$
- calculate $\pi$ 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 $\pi$
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 $\pi$ 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 $\pi$
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 $\pi$ 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 $\pi$

https://arxiv.org/abs/1207.6381

Network Flows: Theory, Algorithms, and Applications
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

```cpp
template 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

```cpp
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
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

```python
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:**

```python
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
```

DON’T RELAX ALL EDGES \( n - 1 \) TIMES!!!
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:

```python
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?
Cost calculation

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

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

Why does this work?
**Cost calculation**

```cpp
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

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycle Cancelling</td>
<td>$O(nm^2UC)$</td>
</tr>
<tr>
<td>Successive Shortest Path (SSP)</td>
<td>$O(nmB)$</td>
</tr>
<tr>
<td>SSP with Dijkstra</td>
<td>$O((n + m) \log n \cdot B)$</td>
</tr>
</tbody>
</table>

$U$: 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](https://courses.engr.illinois.edu/cs473/sp2017/notes/G-mincostflow.pdf)
MinCostFlow Algorithm Summary

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycle Cancelling</td>
<td>$O(nm^2UC)$</td>
</tr>
<tr>
<td>Successive Shortest Path (SSP)</td>
<td>$O(nmB)$</td>
</tr>
<tr>
<td>SSP with Dijkstra</td>
<td>$O((n + m) \log n \cdot B)$</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycle Cancelling</td>
<td>$O(nm^2 UC)$</td>
</tr>
<tr>
<td>Successive Shortest Path (SSP)</td>
<td>$O(nmB)$</td>
</tr>
<tr>
<td>SSP with Dijkstra</td>
<td>$O((n + m) \log n \cdot B)$</td>
</tr>
<tr>
<td>Minimum-Mean Cycle Cancelling</td>
<td>$O(nm^2 \log n)$</td>
</tr>
<tr>
<td>Orlin’s Algorithm</td>
<td>$O(m \log n(m + n \log n))$</td>
</tr>
</tbody>
</table>

### 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](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 bipartite matching
Flow Modelling Techniques

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

\[ u \xrightarrow{c/w} v \] becomes \[ u \xrightarrow{c - 1/w} v \]
Flow Modelling Techniques

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

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

producer

consumer

waypoint

transport path
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

![Diagram showing network with multiple sources and/or sinks](image-url)
Flow Modelling Techniques

- Handling multiple sources and/or sinks
Flow Modelling Techniques

- Handling multiple sources and/or sinks

![Diagram showing a network with multiple sources and sinks, labeled S1, S2, S*, S, T1, T2, T, Sx, Sy, and G.]
Flow Modelling Techniques

- Handling multiple sources and/or sinks
Flow Modelling Techniques

- Handling multiple sources and/or sinks

![Diagram of a graph G with nodes and edges showing multiple sources and sinks](image)
Flow Modelling Techniques

- Handling multiple sources and/or sinks

Calculate $s^*-t^*$-flow in adapted graph
Flow Modelling Techniques

- Flow with demand $d$
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: