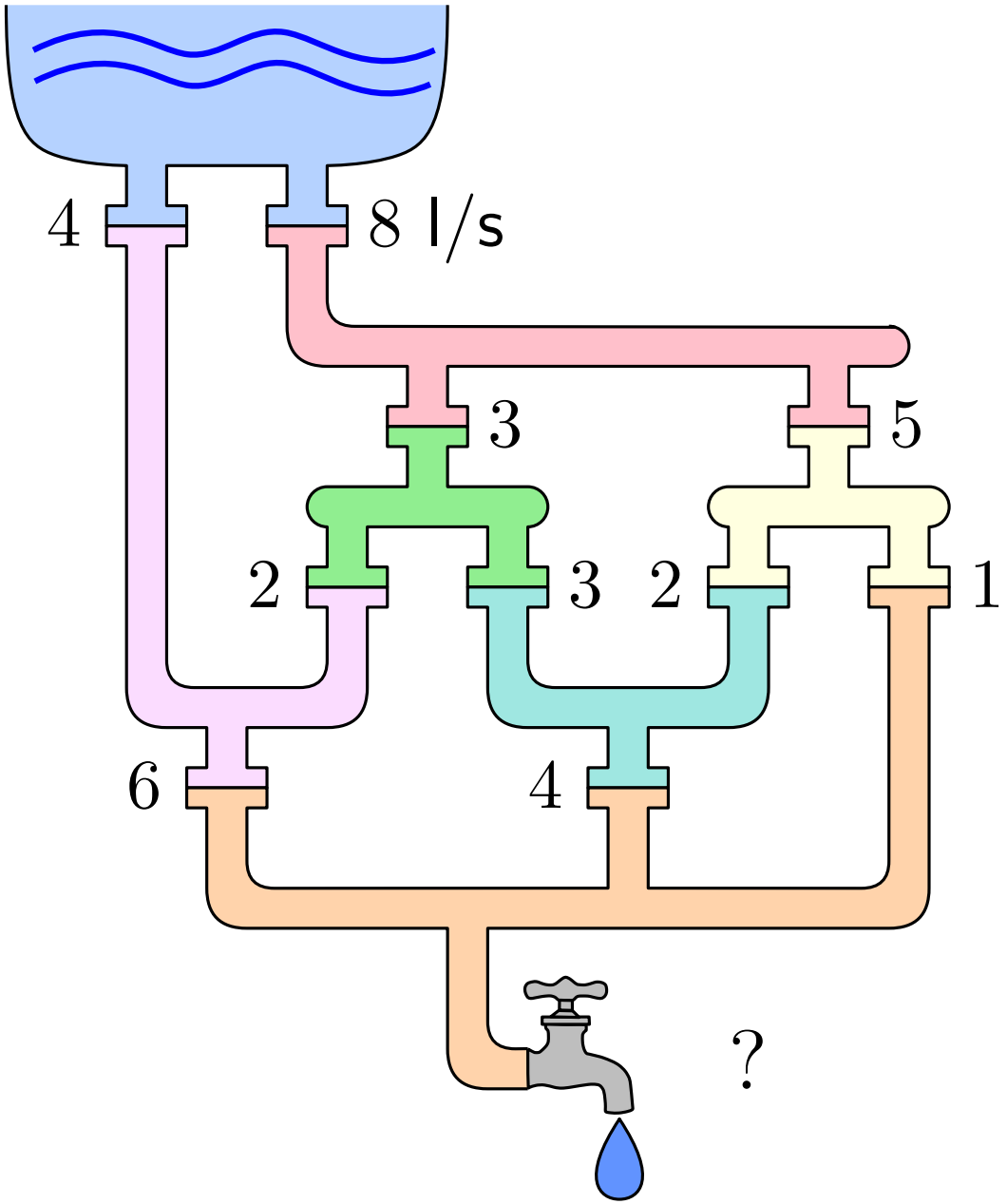
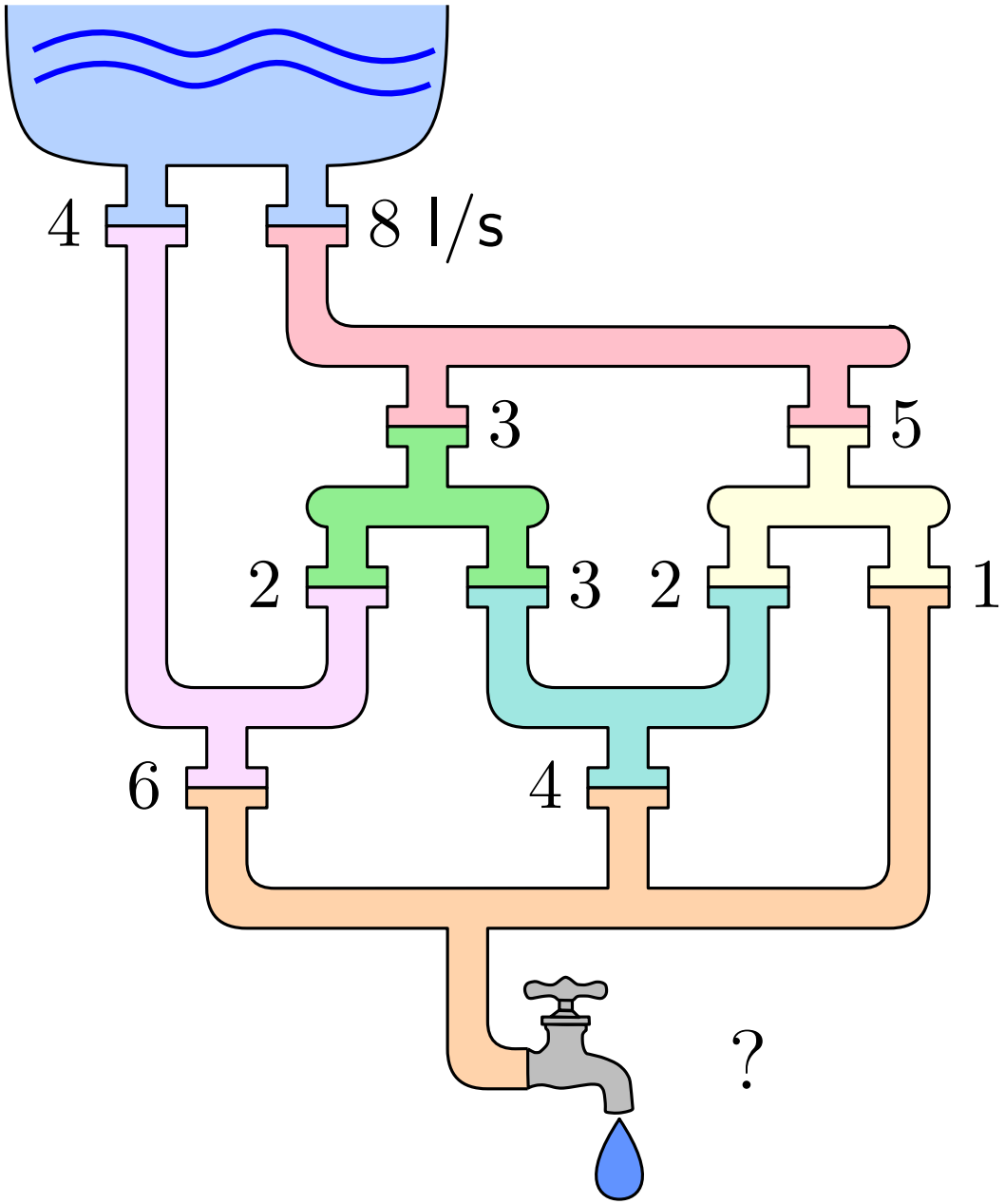
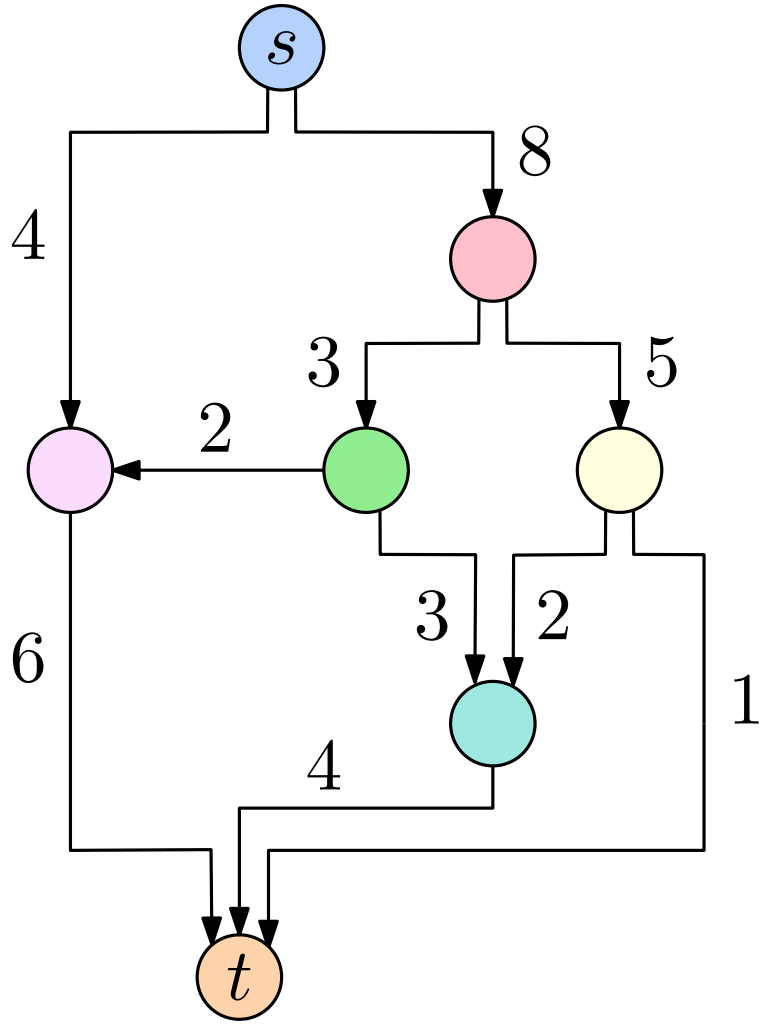
Network Flow

Network Flow



Network Flow





Network Flow: Problem Definition

Input:

- A directed graph $G = (V, E)$
- A source vertex $s \in V$, with no incoming edges
- A target vertex $t \in V$, with no outgoing edges
- A function $c : E \to \mathbb{N}$ that maps each edge to its *capacity*

Output:

A function $f : E \to \mathbb{R}$ that associates each edge e to the *flow* $f(e) \geq 0$ *across* e and satisfies:

- **Capacity constraints:** $\forall e \in E,\ f(e) \leq c(e)$
- **Flow conservation:** $\forall v \in V \setminus \{s, t\}, \sum_{(u,v) \in E} f(u, v) = \sum_{(v,w) \in E} f(v, w)$

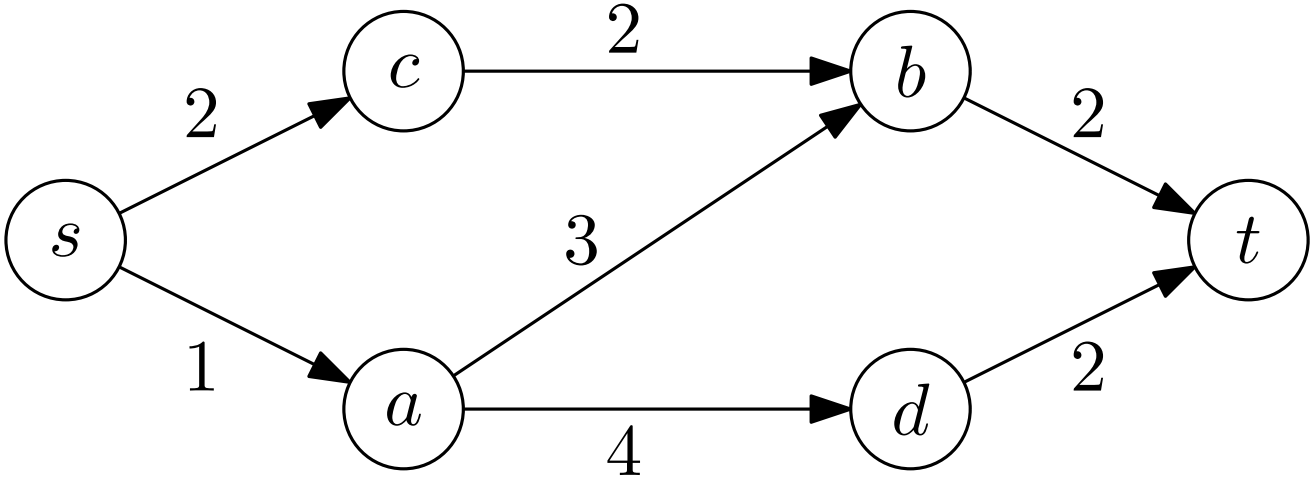
Network Flow: Problem Definition

Measure (to maximize):

- The amount of flow leaving s (equivalently, reaching t).

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

Example:



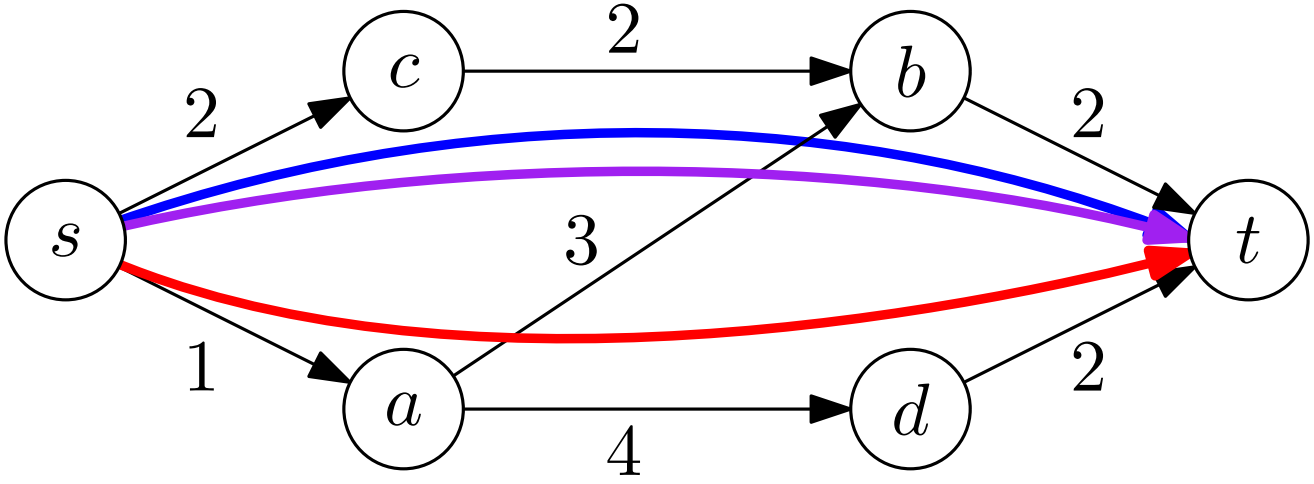
Network Flow: Problem Definition

Measure (to maximize):

- The amount of flow leaving s (equivalently, reaching t).

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

Example:



Maximum flow $= 3$

Linear Programming Formulation

$$\max \sum_{(s,v)\in E} f_{s,v} \qquad \text{s.t.}$$

$|f|$

Capacity

$$c(u,v) - f_{u,v} \geq 0 \qquad \forall (u,v) \in E$$

$$\sum_{(u,v)\in E} f_{u,v} - \sum_{(v,w)\in E} f_{v,w} = 0 \qquad \forall v \in V \setminus \{s,t\}$$

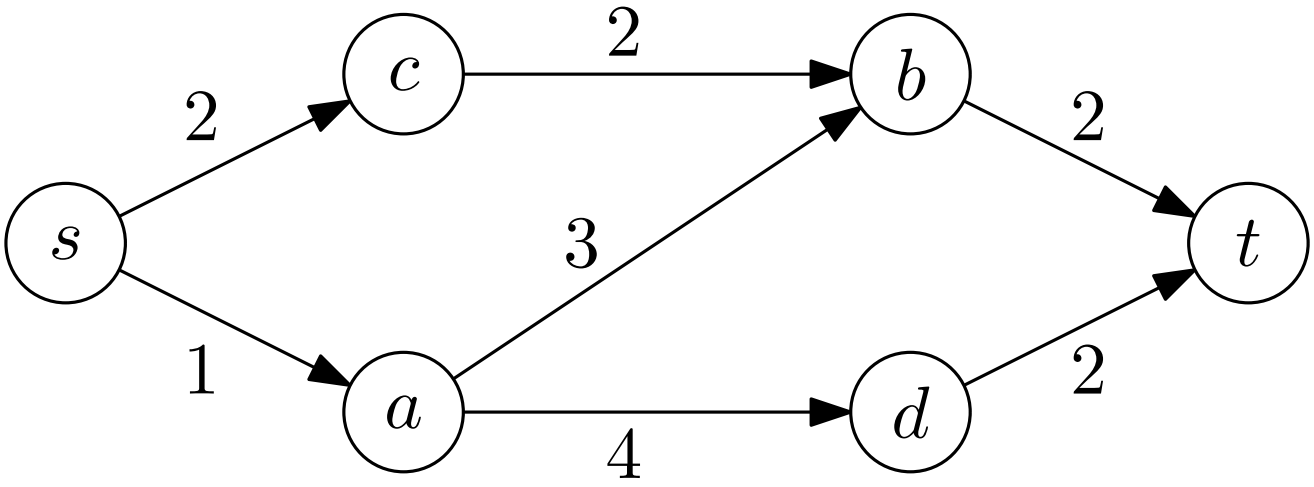
Conservation

$$f_{u,v} \geq 0 \qquad \forall (u,v) \in E$$

Non-negative flow

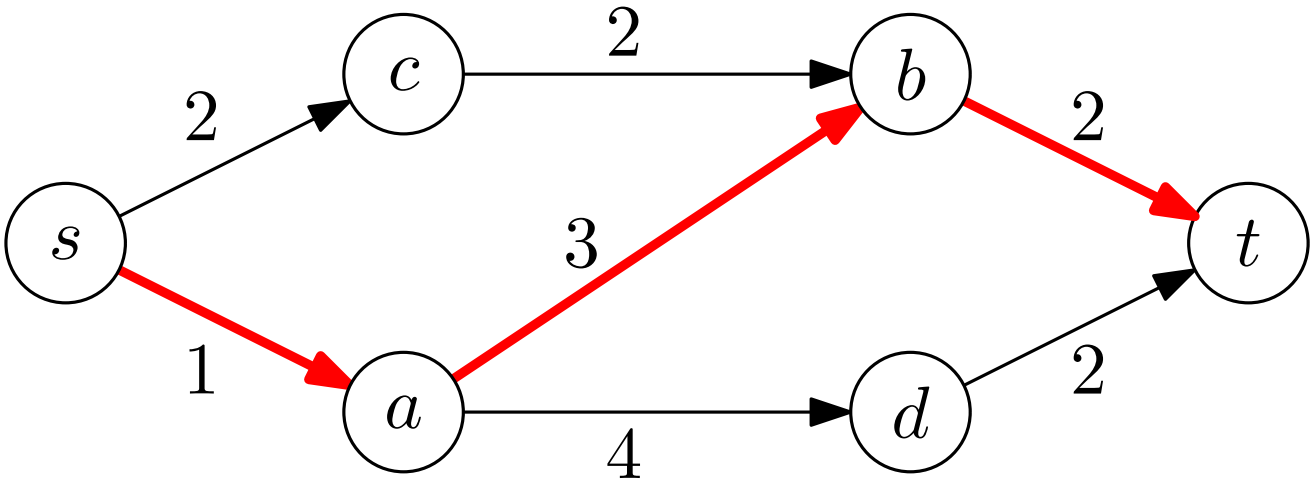
A Solution Attempt

- Find a path P from s to t in G



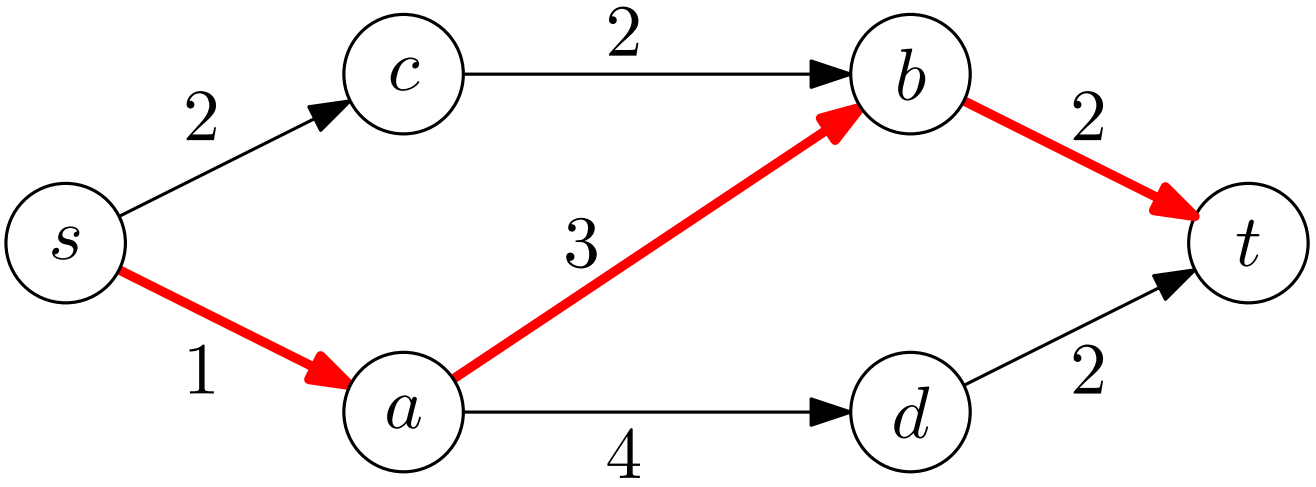
A Solution Attempt

- Find a path P from s to t in G



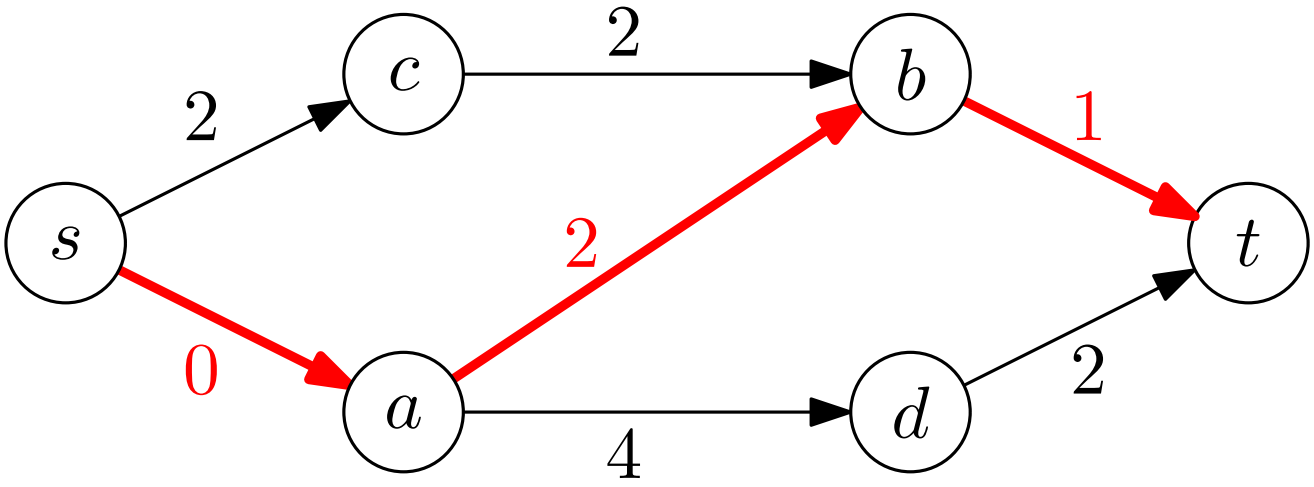
A Solution Attempt

- Find a path P from s to t in G
- Send one unit of flow along P



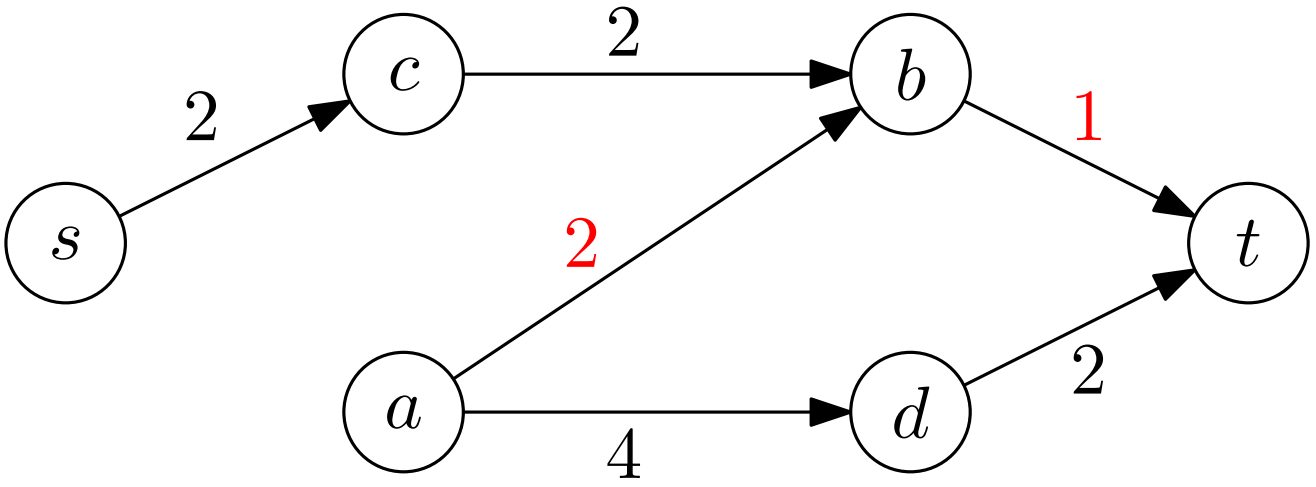
A Solution Attempt

- Find a path P from s to t in G
- Send one unit of flow along P
- Update capacities



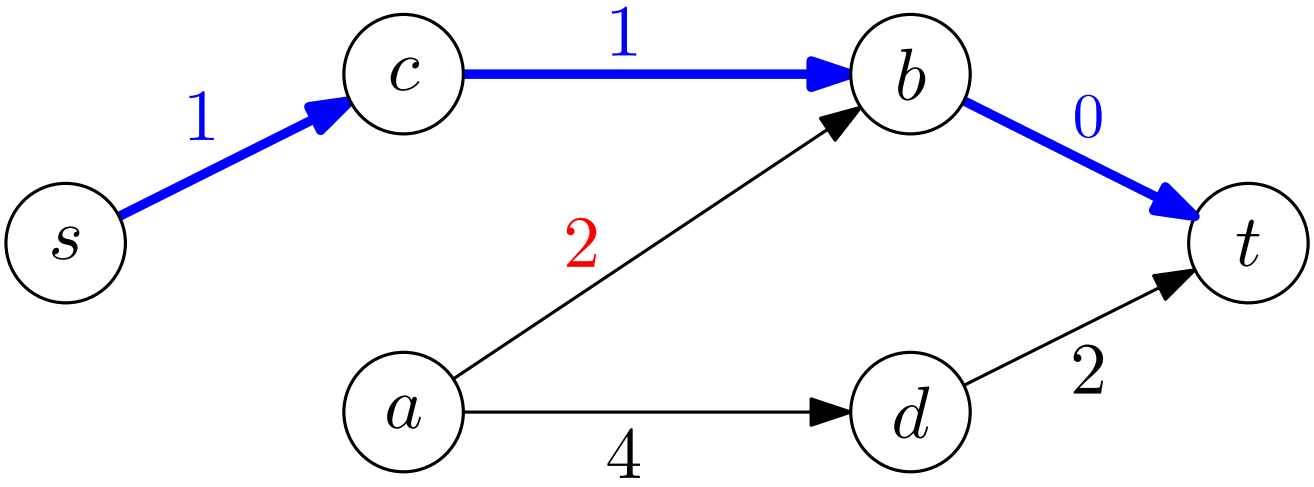
A Solution Attempt

- Find a path P from s to t in G
- Send one unit of flow along P
- Update capacities
- Repeat



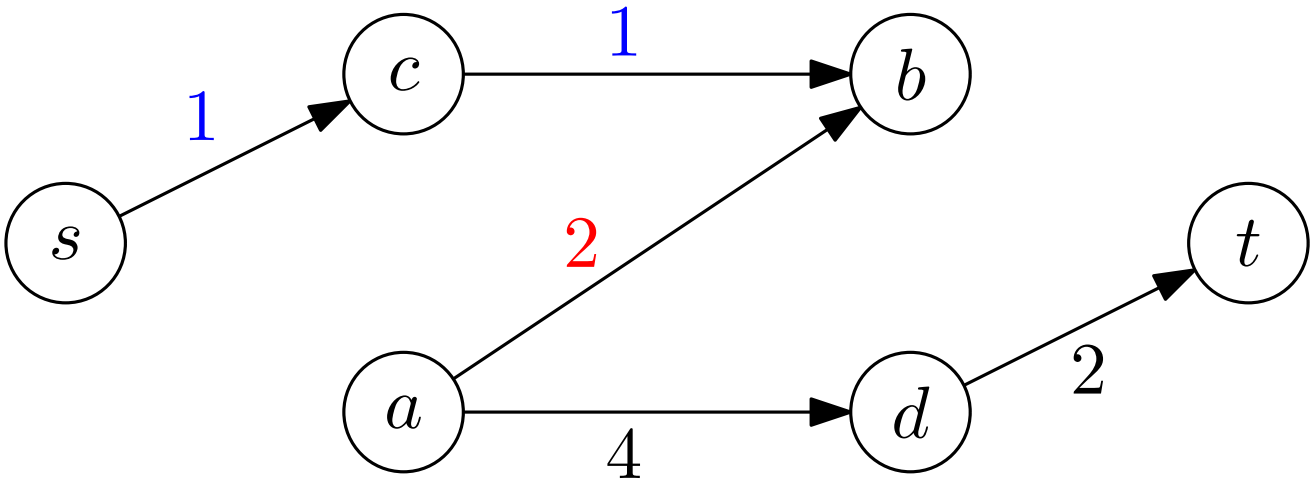
A Solution Attempt

- Find a path P from s to t in G
- Send one unit of flow along P
- Update capacities
- Repeat



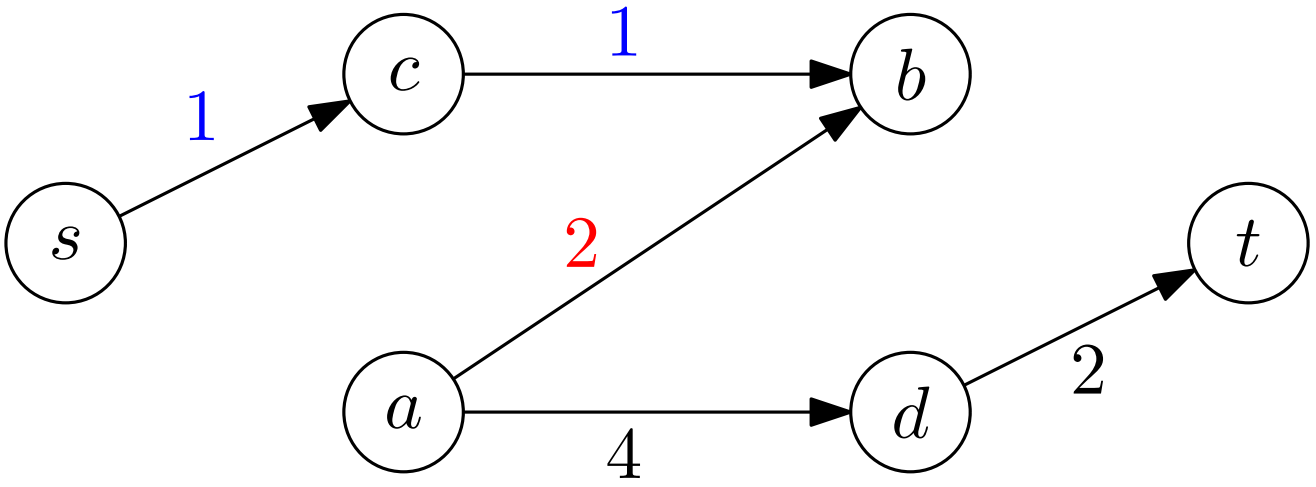
A Solution Attempt

- Find a path P from s to t in G
- Send one unit of flow along P
- Update capacities
- Repeat until no more paths from s to t exist



A Solution Attempt

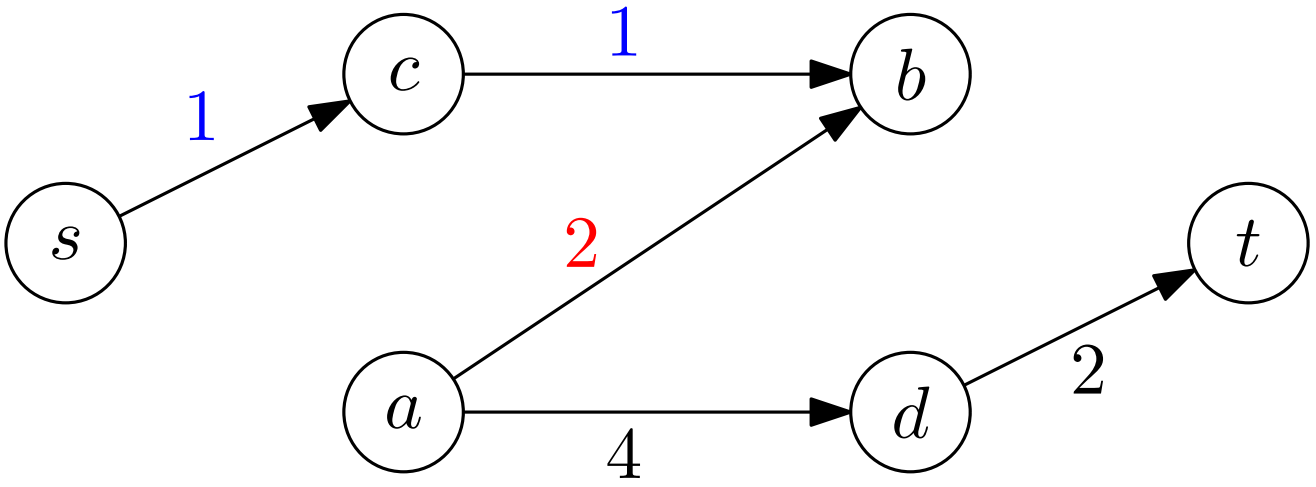
- Find a path P from s to t in G
- Send one unit of flow along P
- Update capacities
- Repeat until no more paths from s to t exist



Computed flow $= 2$

A Solution Attempt

- Find a path P from s to t in G
- Send one unit of flow along P
- Update capacities
- Repeat until no more paths from s to t exist



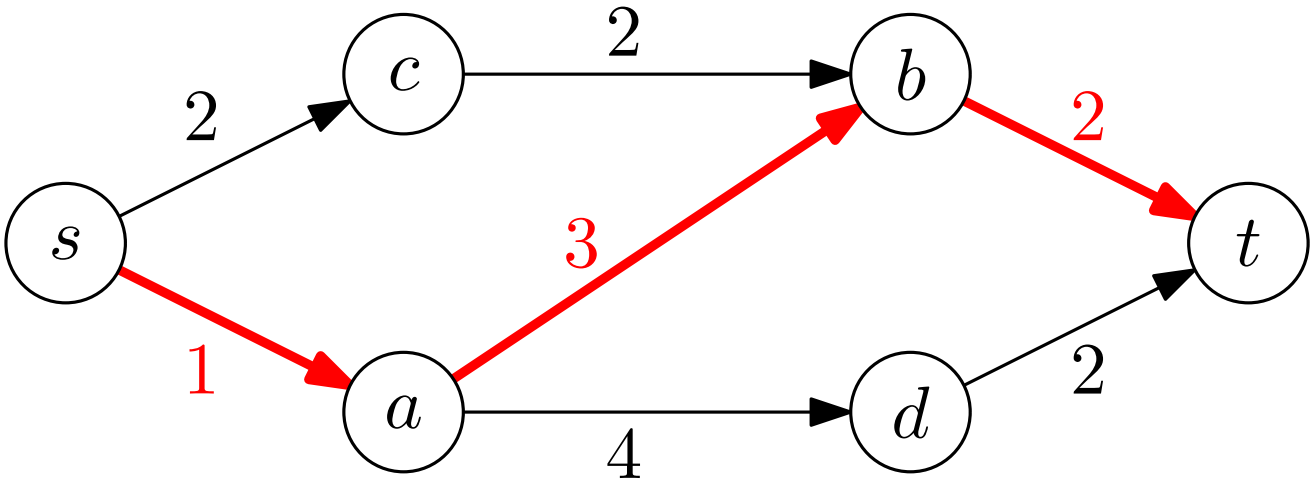
Computed flow $= 2$

Maximum flow $= 3$

Might get stuck in a local maximum.

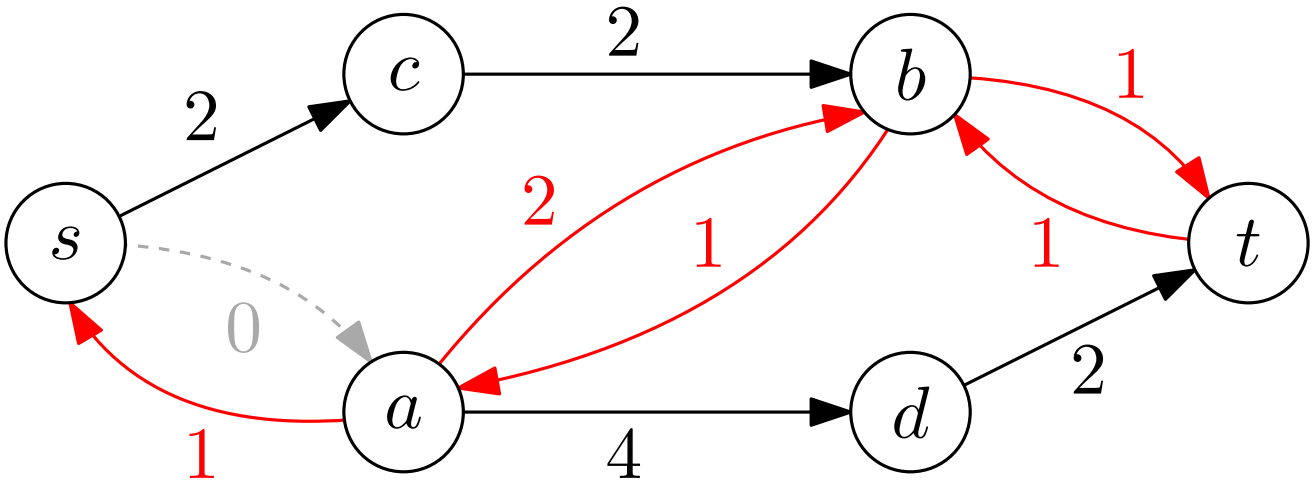
A Second Attempt

- Find an *augmenting path* P from s to t in G
- Send one unit of flow along P
- Compute the *residual graph*



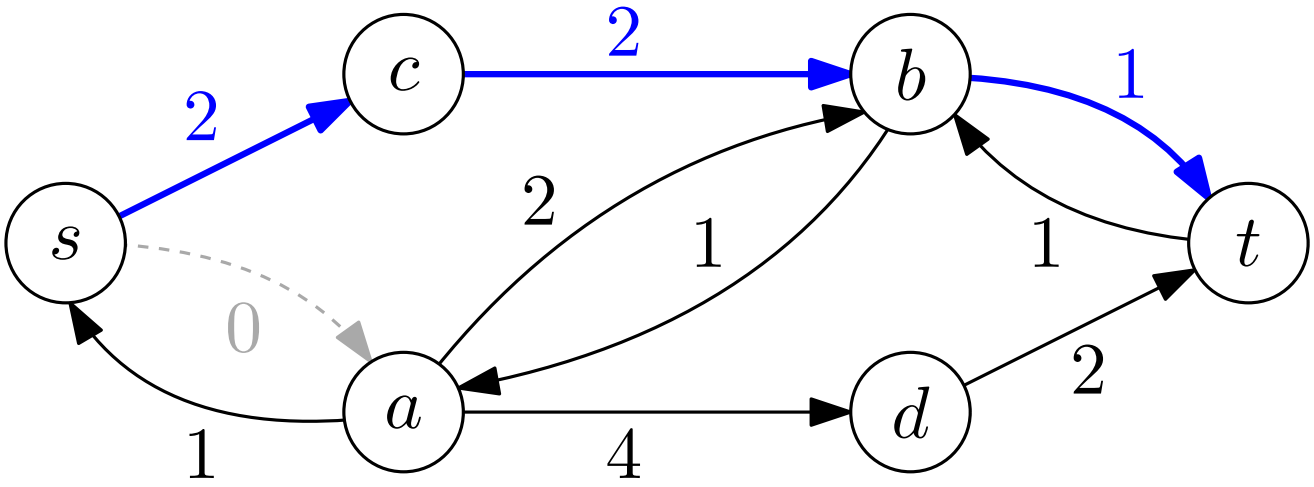
A Second Attempt

- Find an *augmenting path* P from s to t in G
- Send one unit of flow along P
- Compute the *residual graph*



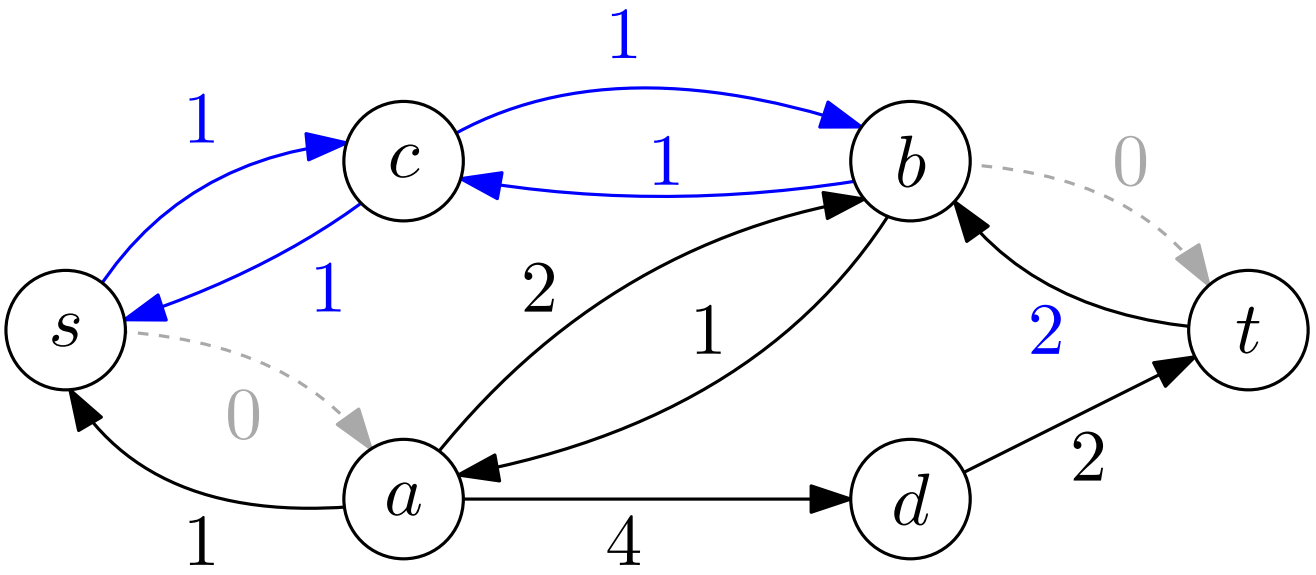
A Second Attempt

- Find an *augmenting path* P from s to t in G
- Send one unit of flow along P
- Compute the *residual graph*
- Repeat



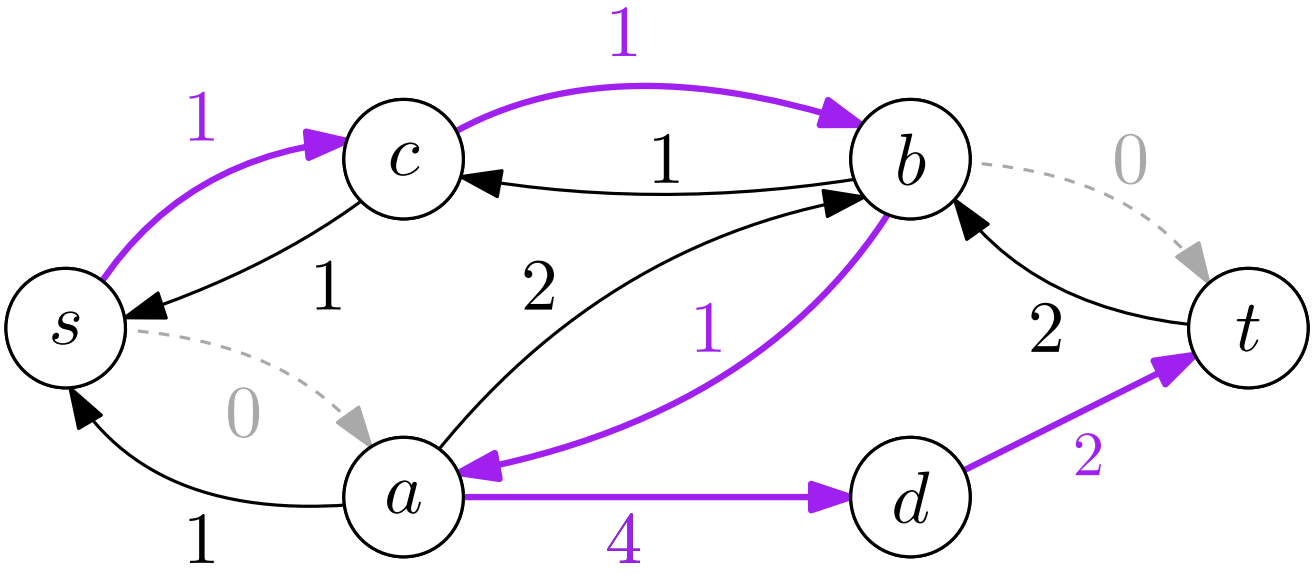
A Second Attempt

- Find an *augmenting path* P from s to t in G
- Send one unit of flow along P
- Compute the *residual graph*
- Repeat



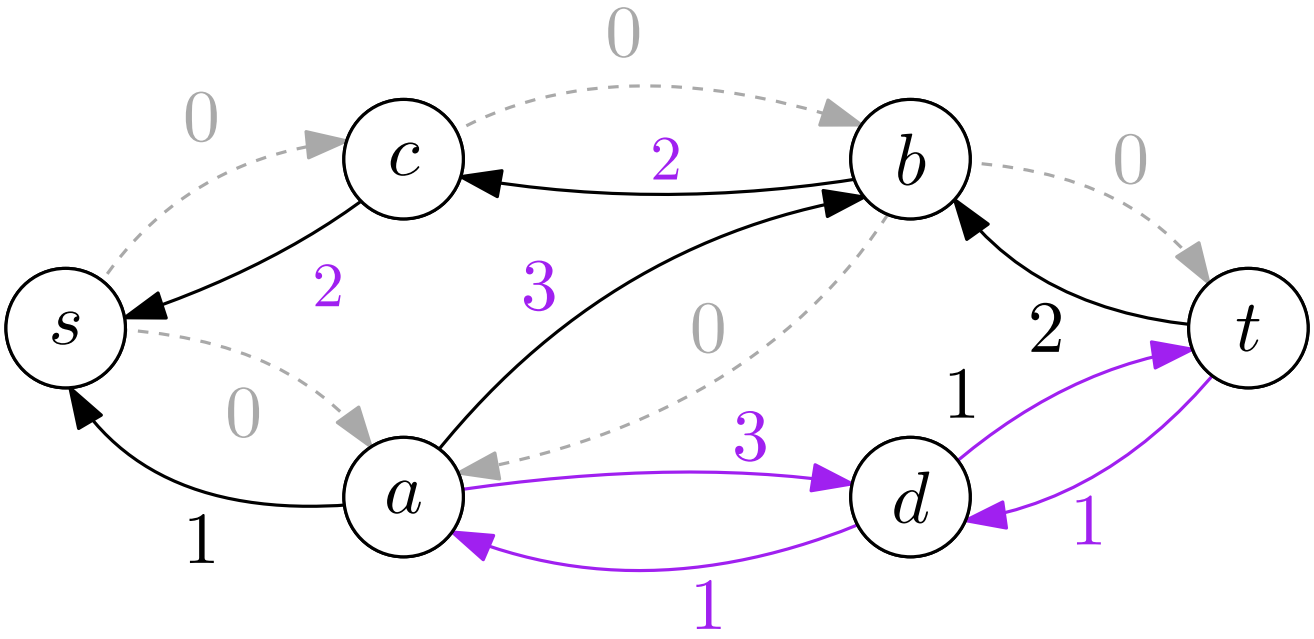
A Second Attempt

- Find an *augmenting path* P from s to t in G
- Send one unit of flow along P
- Compute the *residual graph*
- Repeat



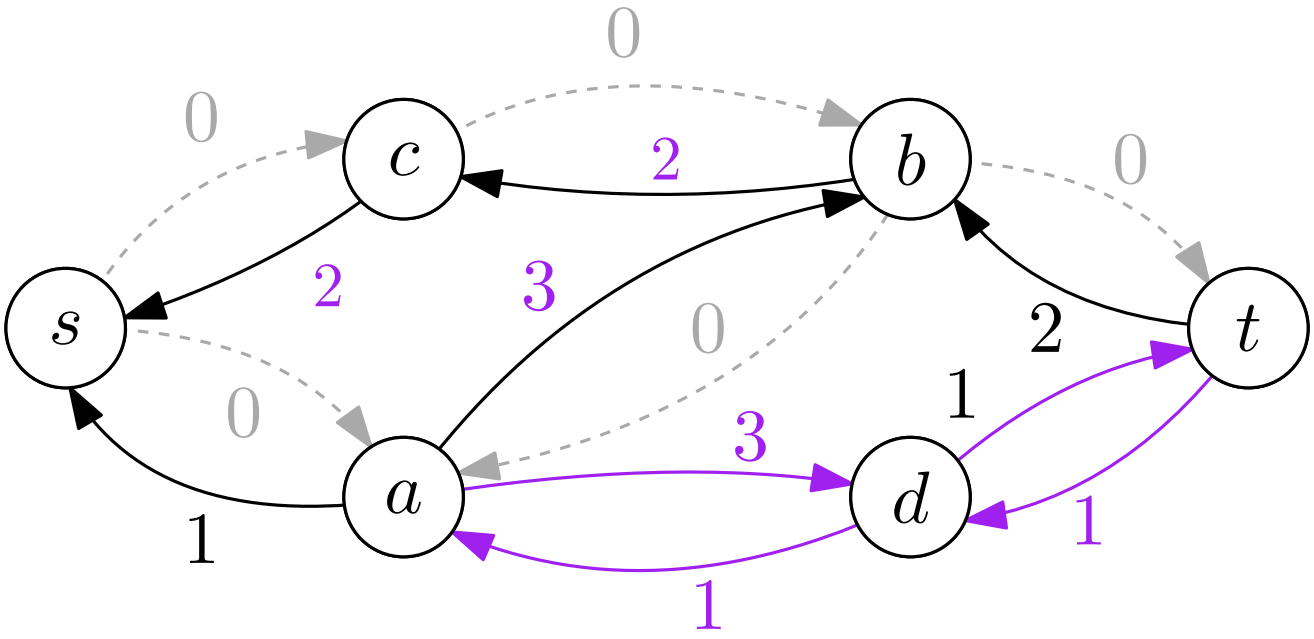
A Second Attempt

- Find an *augmenting path* P from s to t in G
- Send one unit of flow along P
- Compute the *residual graph*
- Repeat



A Second Attempt

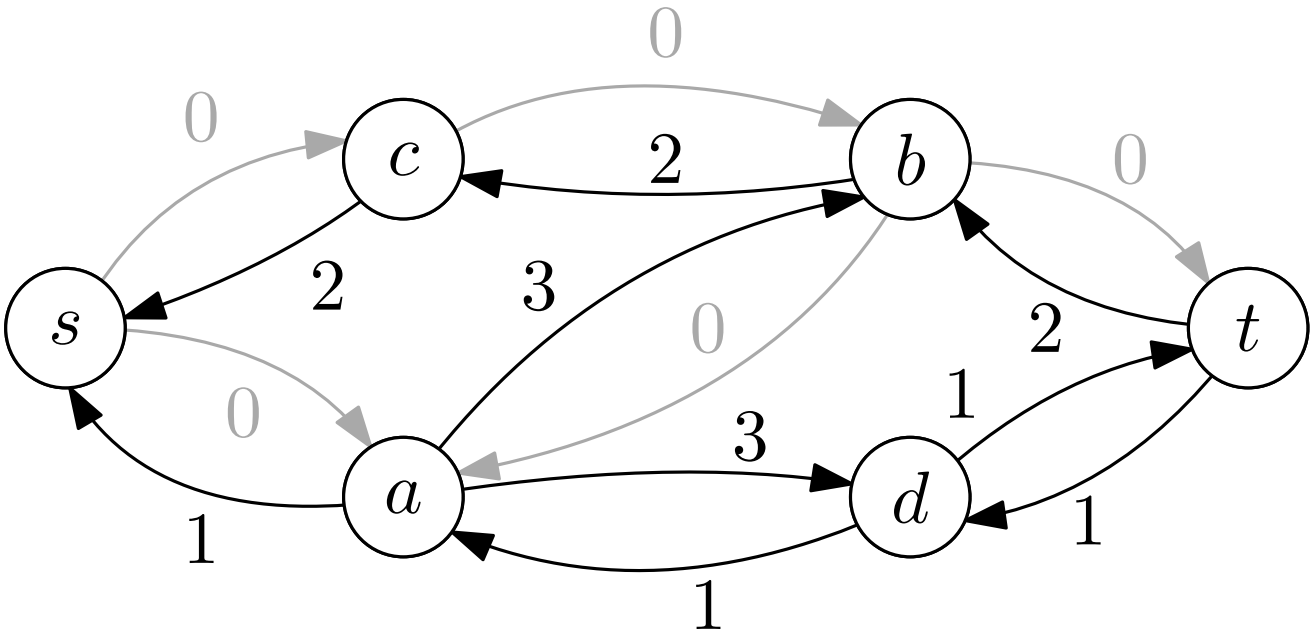
- Find an *augmenting path* P from s to t in G
- Send one unit of flow along P
- Compute the *residual graph*
- Repeat



No more paths from s to t. Computed flow $= 3$.

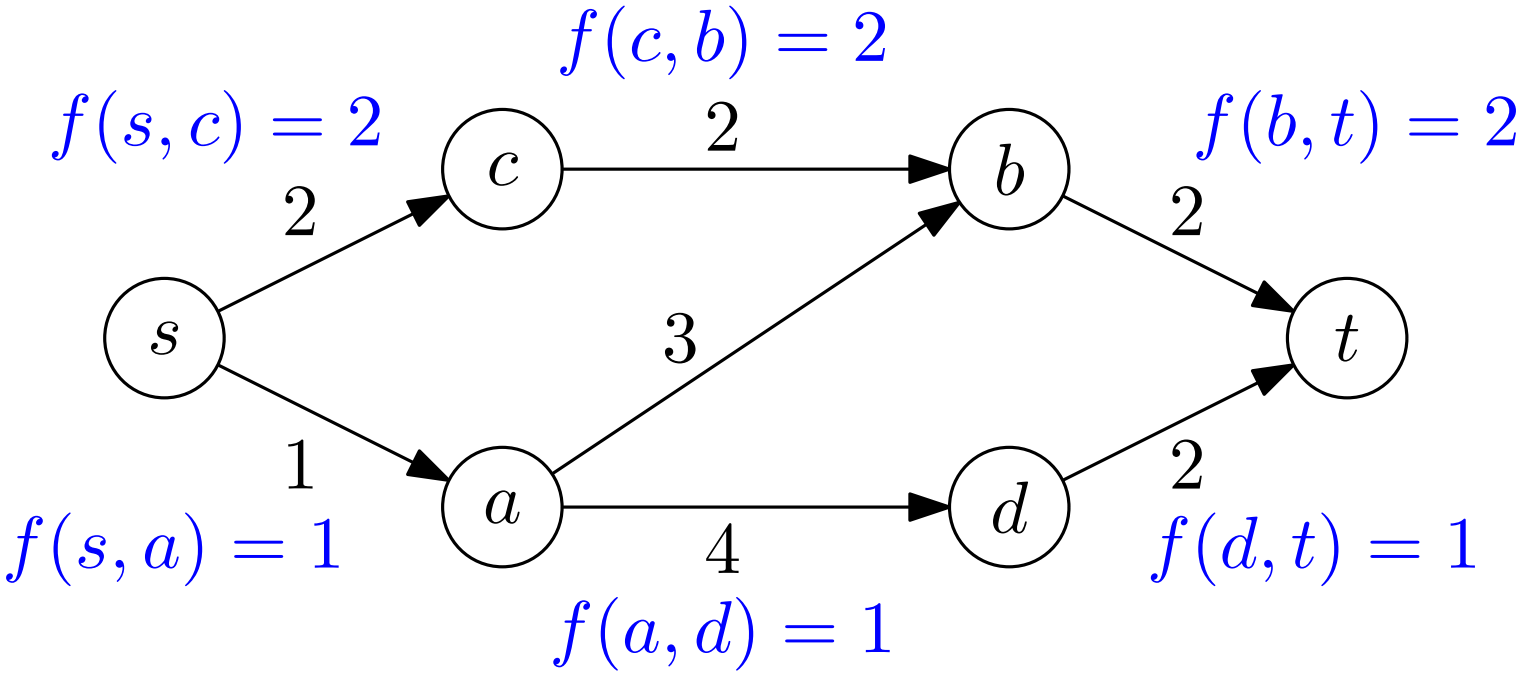
A Second Attempt

- The flow $f(e)$ on e is the original capacity $c(e)$ of e minus the capacity of e in the residual graph G_f.



A Second Attempt

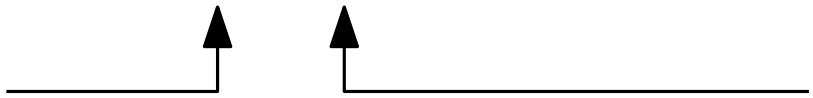
- The flow $f(e)$ on e is the original capacity $c(e)$ of e minus the capacity of e in the residual graph G_f.



Flow Algorithms

- Ford-Fulkerson (1955): Choose any augmenting path P

  Time: $O(m \cdot f^*) = O(m \cdot n \cdot \max_e c(e))$

Time to find P  Value of max flow

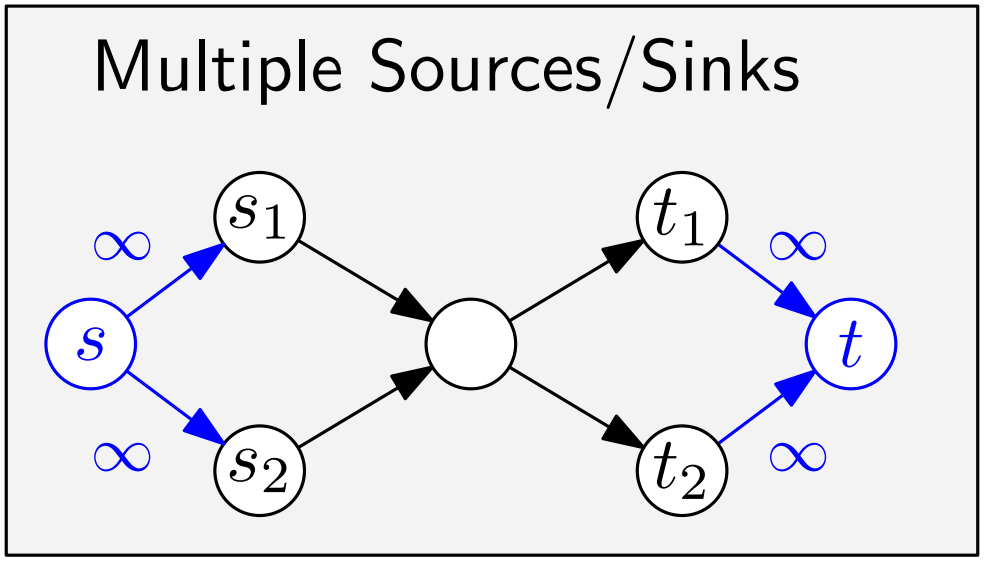
- Edmonds-Karp (1972): Choose an augmenting path P with the fewest number of edges

  Time: $O(\min\{m \cdot f^*, m^2 \cdot n\})$

- Push-Relabel (1986):

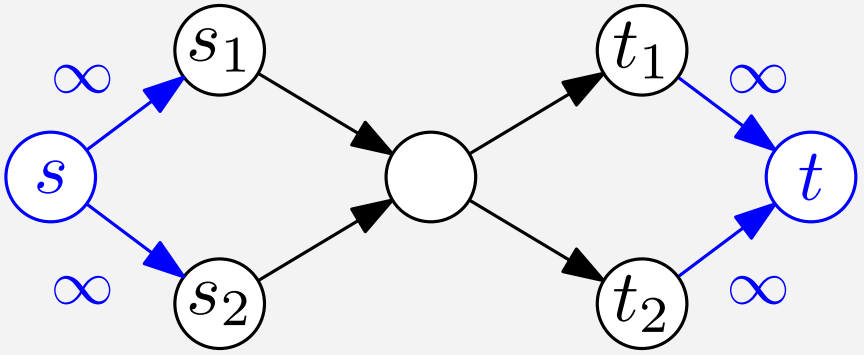
  Time: $O(n^3)$

Flow Tricks

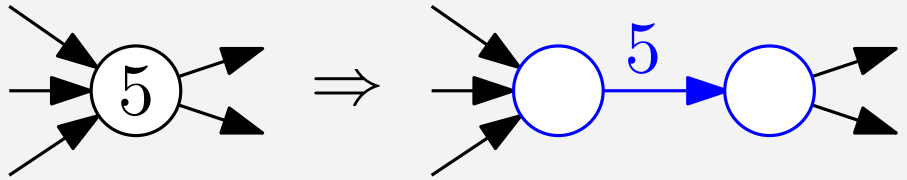

Multiple Sources/Sinks
s
s_1
s_2
t_1
t_2
t
∞
∞
∞
∞

Flow Tricks

Multiple Sources/Sinks

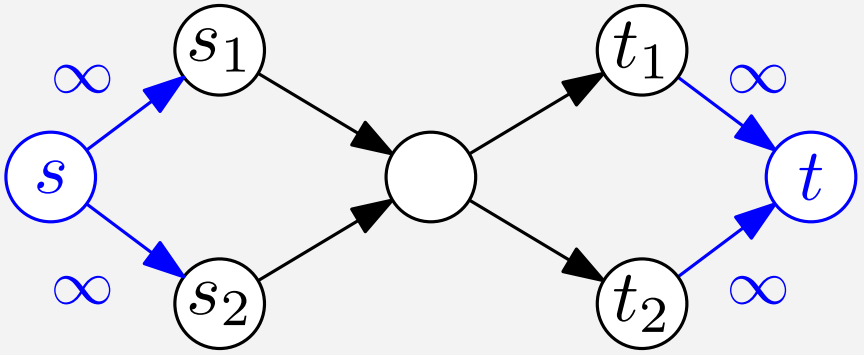


Vertex capacities

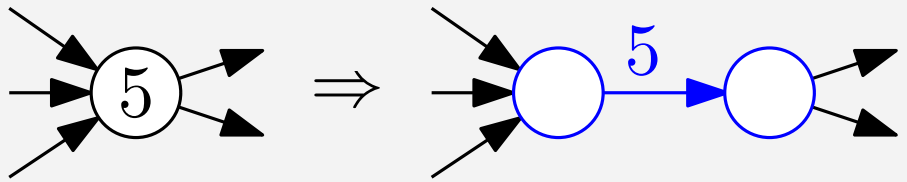


Flow Tricks

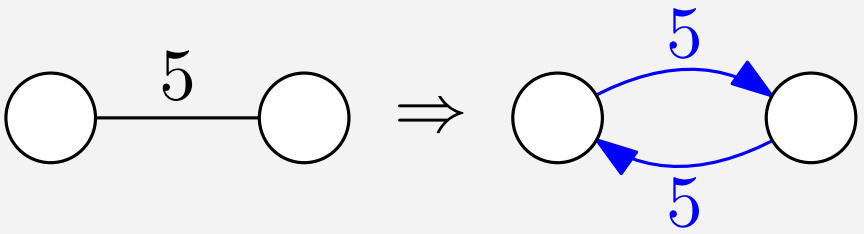
Multiple Sources/Sinks



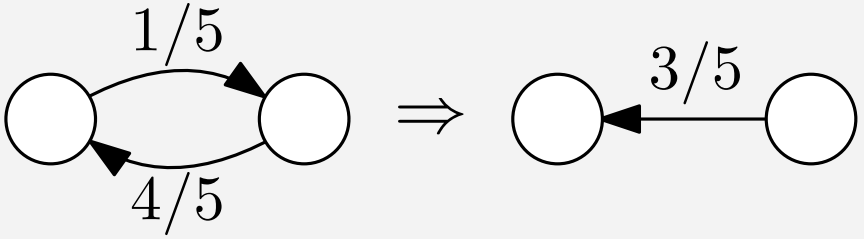
Vertex capacities



Undirected Edges

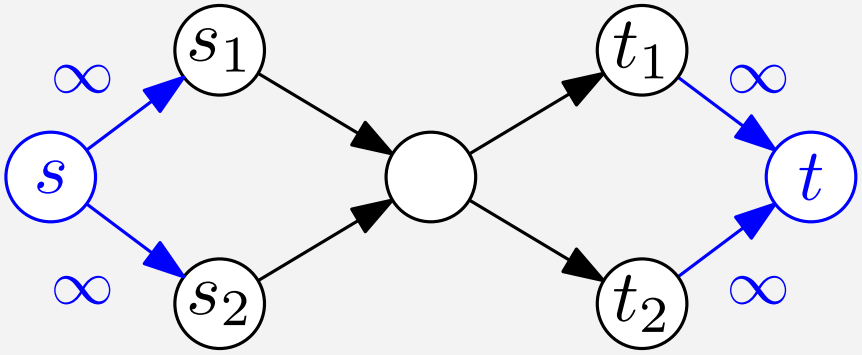


+ re-transform flow to use only 1 direction

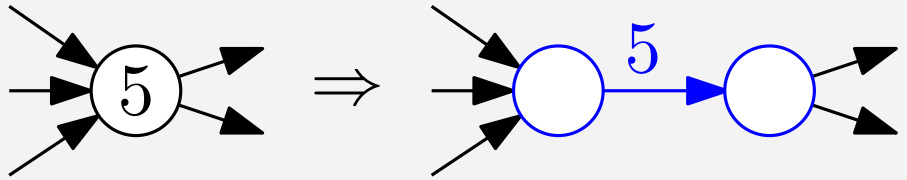


Flow Tricks

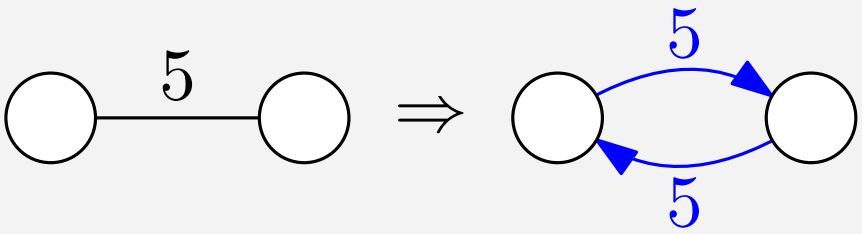
Multiple Sources/Sinks



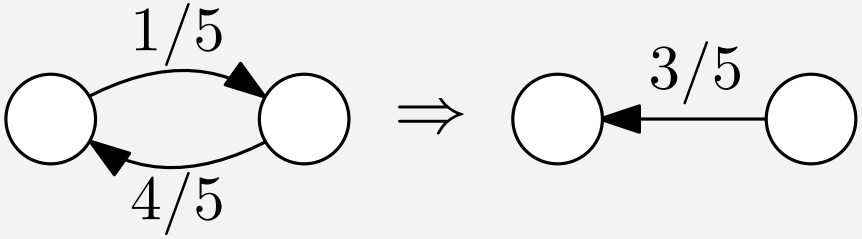
Vertex capacities



Undirected Edges

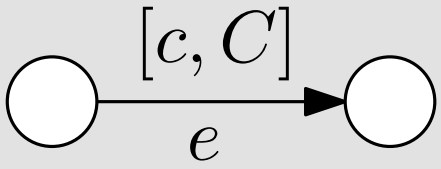


+ re-transform flow to use only 1 direction



Minimum flow across edges

$$f(e) \in [c, C]$$



Solution $f_1 + f_2$ can be found as a feasible flow f_1 plus a max-flow f_2

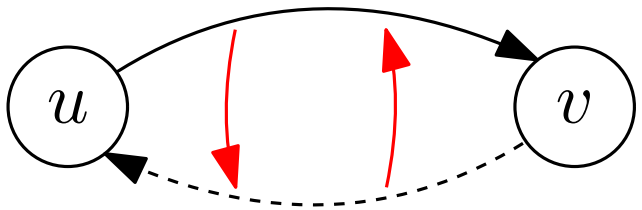
(See provided references if interested in the details)

Network Flow in BGL

Network Flow in BGL

We will need three property maps:

- `boost::edge_capacity_t`: maps each edge e to its capacity $c(e)$.
- `boost::edge_residual_capacity_t`: maps each edge e to its capacity in the residual network.
- `boost::edge_reverse_t`: we need to map each edge $e = (u, v)$ to its corresponding reverse edge $e' = (v, u)$, *and vice-versa*.

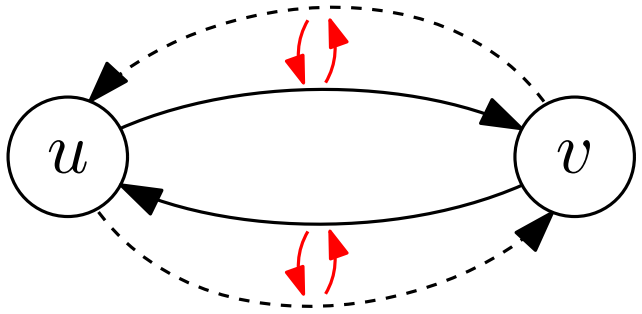


Network Flow in BGL

We will need three property maps:

- `boost::edge_capacity_t`: maps each edge e to its capacity $c(e)$.
- `boost::edge_residual_capacity_t`: maps each edge e to its capacity in the residual network.
- `boost::edge_reverse_t`: we need to map each edge $e = (u, v)$ to its corresponding reverse edge $e' = (v, u)$, *and vice-versa*.

If both (u, v) and (v, u) are in the input graph, then the boost graph will have parallel edges.



Network Flow in BGL

```
#include <boost/graph/adjacency_list.hpp>

typedef boost::adjacency_list_traits<boost::vecS, boost::vecS,
    boost::directedS> Traits;

typedef boost::adjacency_list<boost::vecS, boost::vecS,
   boost::directedS, boost::no_property,
      boost::property<boost::edge_capacity_t, long,
         boost::property<boost::edge_residual_capacity_t, long,
            boost::property<boost::edge_reverse_t, Traits::edge_descriptor>
   > > > Graph;

typedef boost::property_map<Graph, boost::edge_capacity_t>::type
capacity_map;

typedef boost::property_map<Graph, boost::edge_residual_capacity_t>::type
residual_map;

typedef boost::property_map<Graph, boost::edge_reverse_t>::type
reverse_map;
```

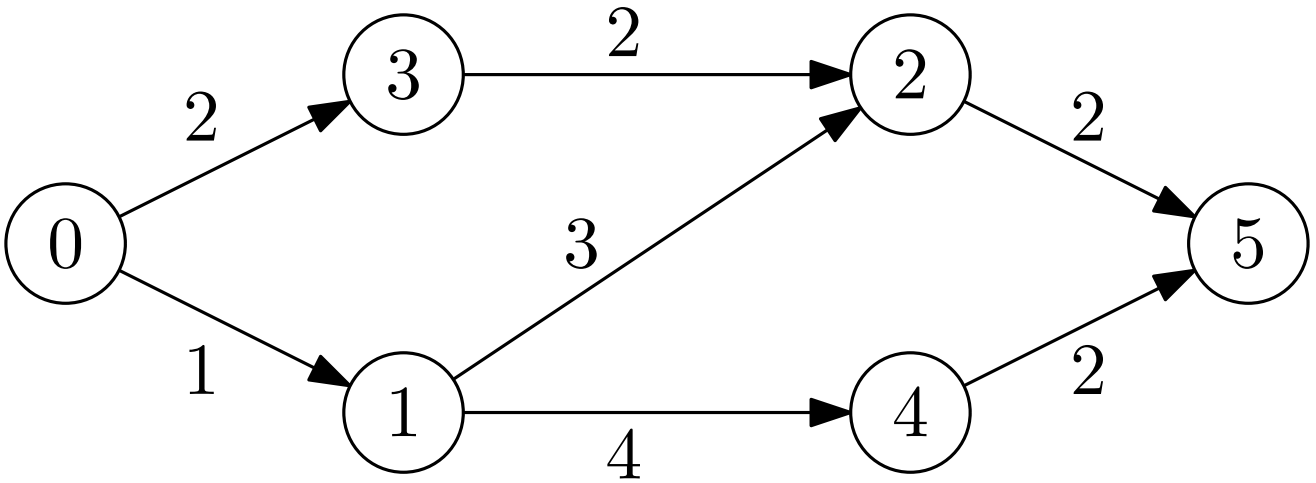
Network Flow in BGL

Simplify Graph Construction with a Helper Class

```
class EdgeAdder
{
    Graph &G; capacity_map capacity; reverse_map reverse;

public:
    explicit EdgeAdder(Graph &G) : G(G)
    {
        capacity = boost::get(boost::edge_capacity, G);
        reverse = boost::get(boost::edge_reverse, G);
    }

    void add_edge(long u, long v, long c)
    {
        auto [e, added] = boost::add_edge(u, v, G);
        auto [rev, rev_added] = boost::add_edge(v, u, G);

        capacity[e] = c; capacity[rev] = 0;
        reverse[e] = rev; reverse[rev] = e;
    }
};
```

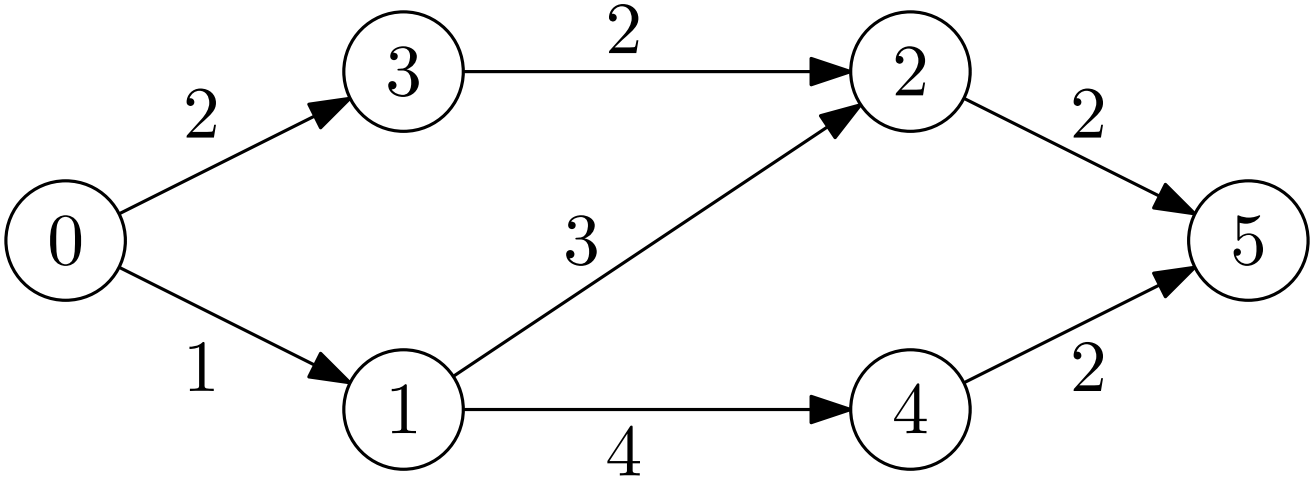
Network Flow in BGL

```
int main()
{
    Graph G(6);
    EdgeAdder edge_adder(G);

    edge_adder.add_edge(0, 1, 1);
    edge_adder.add_edge(0, 3, 2);
    edge_adder.add_edge(1, 2, 3);
    edge_adder.add_edge(1, 4, 4);
    edge_adder.add_edge(2, 5, 2);
    edge_adder.add_edge(3, 2, 2);
    edge_adder.add_edge(4, 5, 2);

    [...]
```



Network Flow in BGL

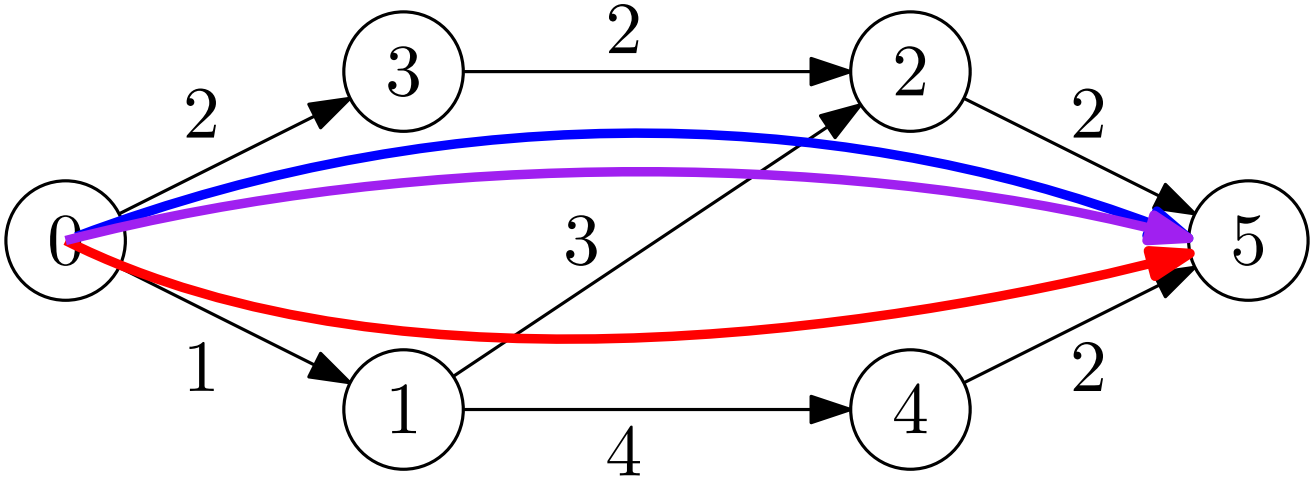
```
#include <boost/graph/edmonds_karp_max_flow.hpp>

[...]

long flow = boost::edmonds_karp_max_flow(G, 0, 5);

std::cout << "The␣maximum␣flow␣from␣0␣to␣5␣is␣" << flow << "\n";
capacity_map capacity = boost::get(boost::edge_capacity, G);
residual_map residual_capacity =
                            boost::get(boost::edge_residual_capacity, G);

auto [e, found] = boost::edge(1, 4, G);
std::cout << "The␣flow␣across␣edge␣(1,␣4)␣is:␣"
          << capacity[e] - residual_capacity[e] << "\n";
```



Network Flow in BGL

```
#include <boost/graph/edmonds_karp_max_flow.hpp>
[..
lon
std
cap
res
                                                    G);
aut
std
        << capacity[e] << " " << residual_capacity[e] << "\n";
```

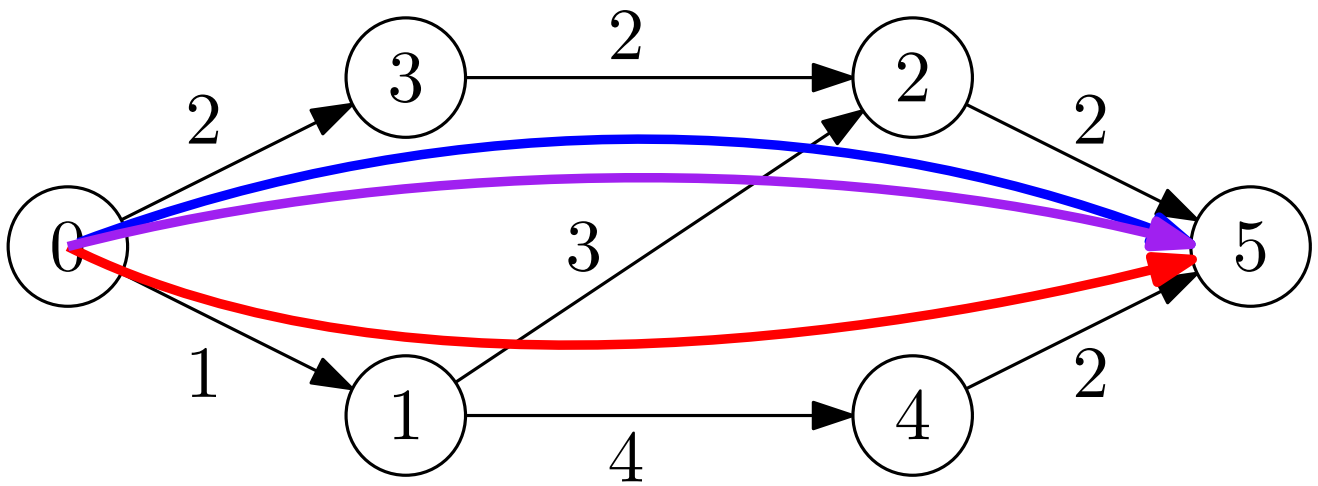
```
$ g++ -std=c++17 flow.cpp -o flow
$
$ ./flow
The maximum flow from 0 to 5 is 3
The flow across edge (1, 4) is: 1
$
```



Push-Relabel in BGL

```
#include <boost/graph/edmonds_karp_max_flow.hpp>

long flow = boost::edmonds_karp_max_flow(G, 0, 5);
```

```
#include <boost/graph/push_relabel_max_flow.hpp>

long flow = boost::push_relabel_max_flow(G, 0, 5);
```



Flow Applications

Minimum s–t Cut

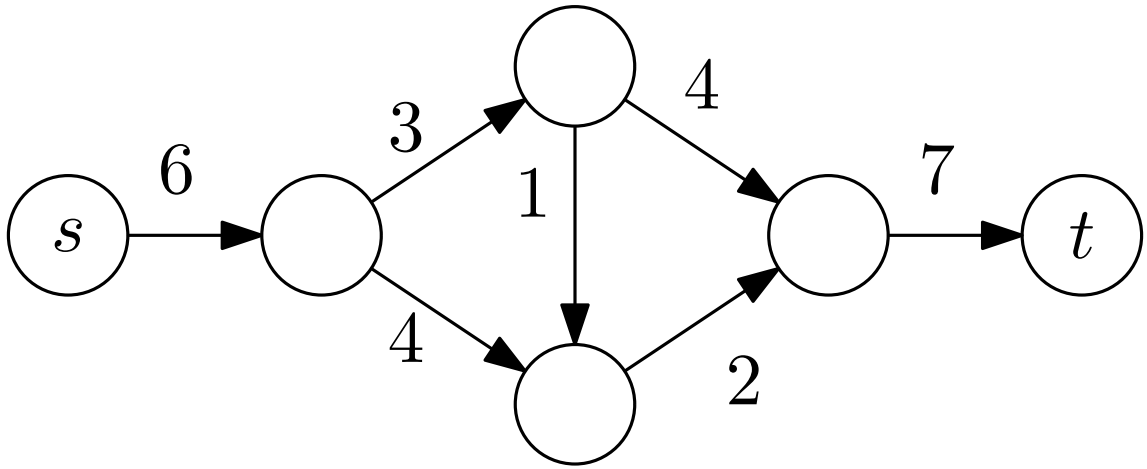
Input:

- A directed graph $G = (V, E)$ with non-negative edge weights $c : E \to \mathbb{N}$.
- Two distinguished vertices $s, t \in V$.
- An s–t cut is a partition A, B of V with $s \in A$ and $t \in B$.

Output:

- An s–t cut of minimum *capacity* $\text{cap}(A, B) = \sum_{\substack{e=(u,v)\in E \\ u\in A, v\in B}} c(e)$.

Example:



Minimum s–t Cut

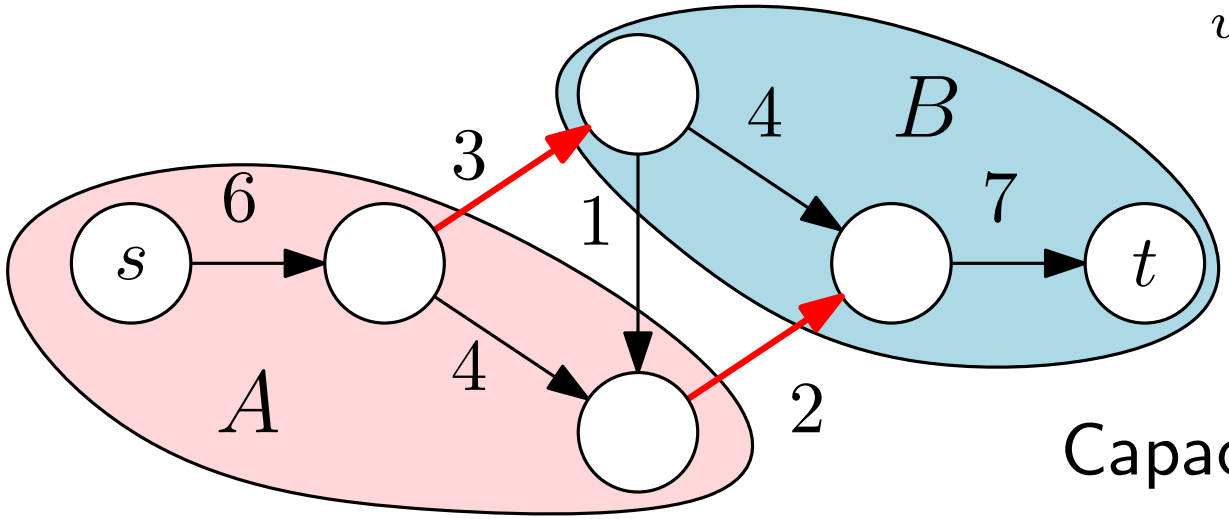
Input:

- A directed graph $G = (V, E)$ with non-negative edge weights $c : E \to \mathbb{N}$.
- Two distinguished vertices $s, t \in V$.
- An s–t cut is a partition A, B of V with $s \in A$ and $t \in B$.

Output:

- An s–t cut of minimum *capacity* $\text{cap}(A, B) = \sum_{\substack{e=(u,v)\in E \\ u\in A, v\in B}} c(e)$.

Example:



Capacity: 5

Minimum s–t Cut

Theorem: Let f be a maximum flow between s and t in G. Let (A^*, B^*) be an s–t cut of minimum capacity in G.

$$|f| = \text{cap}(A^*, B^*).$$

Minimum s–t Cut

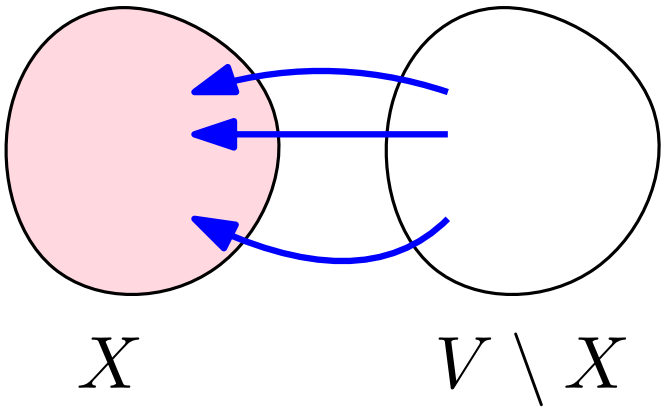
Theorem: Let f be a maximum flow between s and t in G. Let (A^*, B^*) be an s–t cut of minimum capacity in G.

$$|f| = \text{cap}(A^*, B^*).$$

Proof: Given $X \subseteq V$, define:

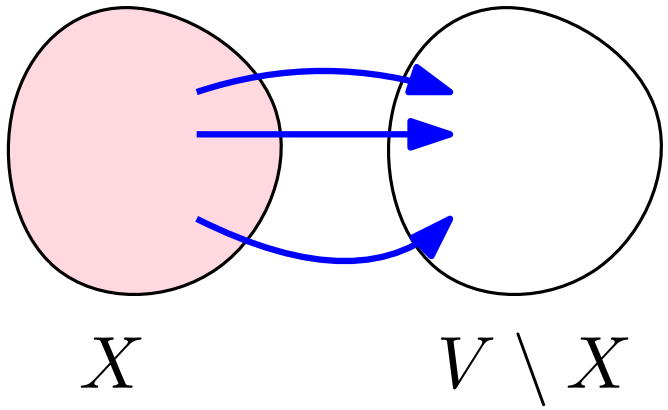
Flow going into X

$$f^{\text{in}}(X) = \sum_{\substack{e=(u,v)\in E \\ u\in V\setminus X, v\in X}} f(e)$$



Flow leaving X

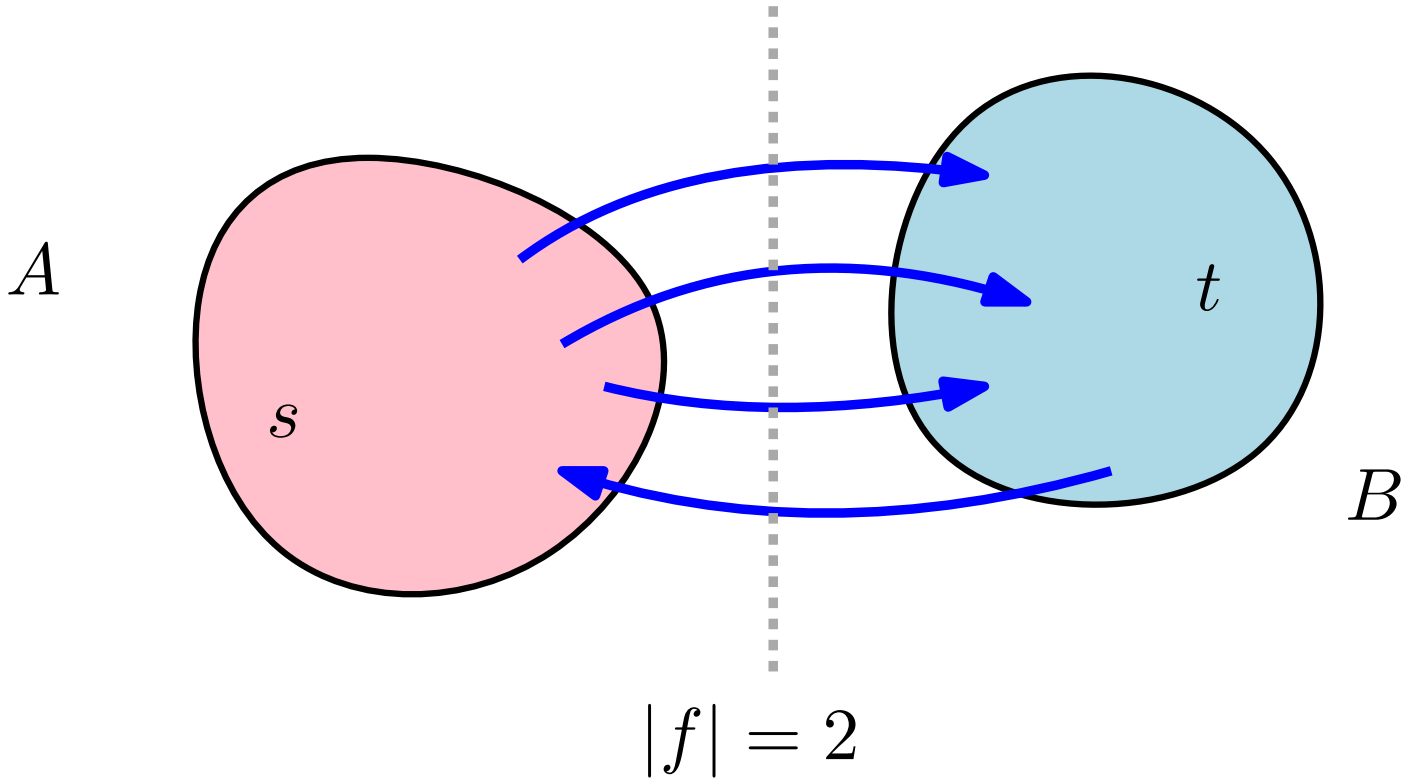
$$f^{\text{out}}(X) = \sum_{\substack{e=(u,v)\in E \\ u\in X, v\in V\setminus X}} f(e)$$



Minimum s–t Cut

Lemma: Let (A, B) be an s–t cut. $|f| = f^{\text{out}}(A) - f^{\text{in}}(A)$.

Intuitively: the net flow leaving any s–t cut is $|f|$.



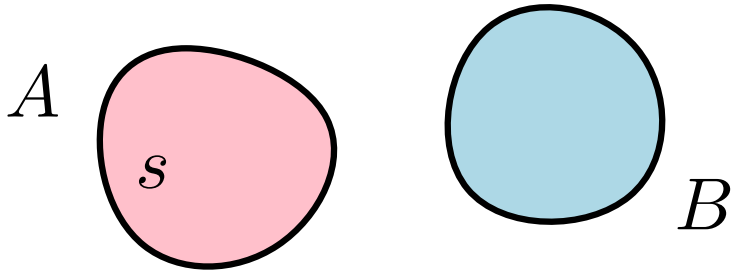
Minimum s–t Cut

Lemma: Let (A, B) be an s–t cut. $|f| = f^{\text{out}}(A) - f^{\text{in}}(A)$.

Proof:

$$|f| = f^{\text{out}}(s) = \sum_{u \in A} \left(f^{\text{out}}(u) - f^{\text{in}}(u) \right)$$

Consider the contribution of edge $e = (u, v) \in E$ to the sum:



Minimum s–t Cut

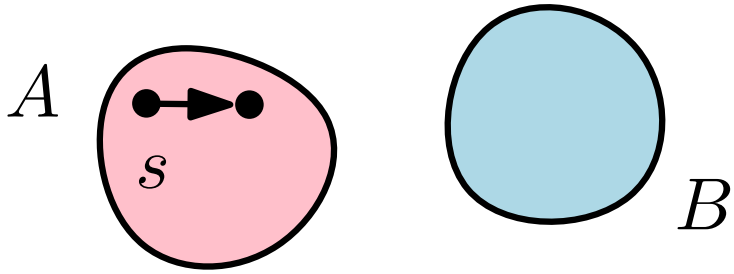
Lemma: Let (A, B) be an s–t cut. $|f| = f^{\text{out}}(A) - f^{\text{in}}(A)$.

Proof:

$$|f| = f^{\text{out}}(s) = \sum_{u \in A} \left(f^{\text{out}}(u) - f^{\text{in}}(u) \right)$$

Consider the contribution of edge $e = (u, v) \in E$ to the sum:

- If $u, v \in A$, e contributes 0



Minimum s–t Cut

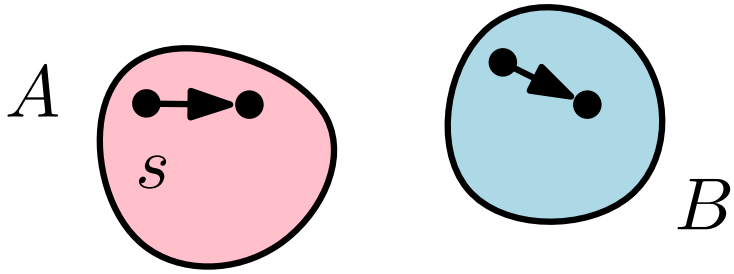
Lemma: Let (A, B) be an s–t cut. $|f| = f^{\text{out}}(A) - f^{\text{in}}(A)$.

Proof:

$$|f| = f^{\text{out}}(s) = \sum_{u \in A} \left(f^{\text{out}}(u) - f^{\text{in}}(u) \right)$$

Consider the contribution of edge $e = (u, v) \in E$ to the sum:

- If $u, v \in A$, e contributes 0
- If $u, v \in B$, e contributes 0



Minimum s–t Cut

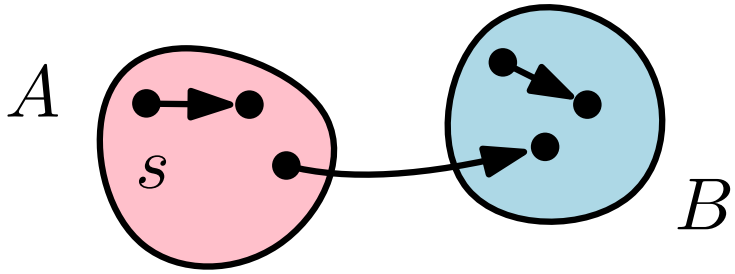
Lemma: Let (A, B) be an s–t cut. $|f| = f^{\text{out}}(A) - f^{\text{in}}(A)$.

Proof:

$$|f| = f^{\text{out}}(s) = \sum_{u \in A} \left(f^{\text{out}}(u) - f^{\text{in}}(u) \right)$$

Consider the contribution of edge $e = (u, v) \in E$ to the sum:

- If $u, v \in A$, e contributes 0
- If $u, v \in B$, e contributes 0
- If $u \in A$ and $v \in B$, e contributes $f(e)$



Minimum s–t Cut

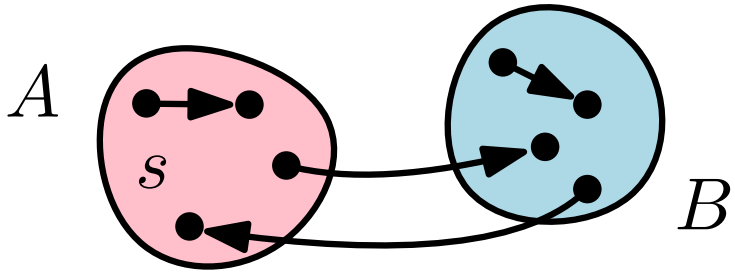
Lemma: Let (A, B) be an s–t cut. $|f| = f^{\text{out}}(A) - f^{\text{in}}(A)$.

Proof:

$$|f| = f^{\text{out}}(s) = \sum_{u \in A} \left(f^{\text{out}}(u) - f^{\text{in}}(u) \right)$$

Consider the contribution of edge $e = (u, v) \in E$ to the sum:

- If $u, v \in A$, e contributes 0
- If $u, v \in B$, e contributes 0
- If $u \in A$ and $v \in B$, e contributes $f(e)$
- If $u \in B$ and $v \in A$, e contributes $-f(e)$



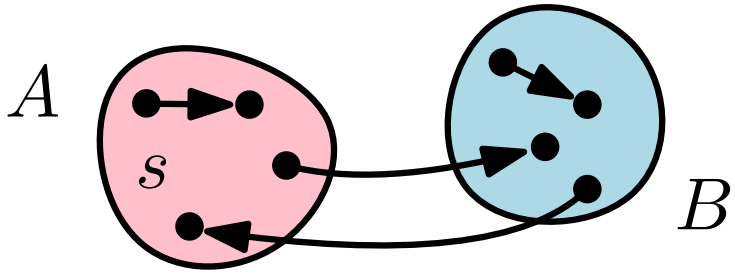
Minimum s–t Cut

Lemma: Let (A, B) be an s–t cut. $|f| = f^{\text{out}}(A) - f^{\text{in}}(A)$.

Proof:

$$|f| = f^{\text{out}}(s) = \sum_{u \in A} \left(f^{\text{out}}(u) - f^{\text{in}}(u) \right)$$

Consider the contribution of edge $e = (u, v) \in E$ to the sum:



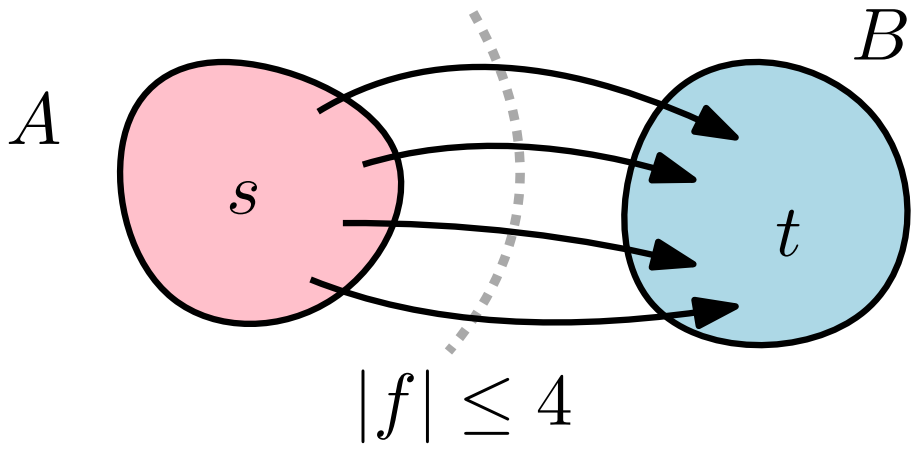
- If $u, v \in A$, e contributes 0
- If $u, v \in B$, e contributes 0
- If $u \in A$ and $v \in B$, e contributes $f(e)$
- If $u \in B$ and $v \in A$, e contributes $-f(e)$

$$|f| = \sum_{\substack{e=(u,v) \in E \\ u \in A, v \in V \setminus A}} f(e) - \sum_{\substack{e=(u,v) \in E \\ u \in V \setminus A, v \in A}} f(e) = f^{\text{out}}(A) - f^{\text{in}}(A)$$

□

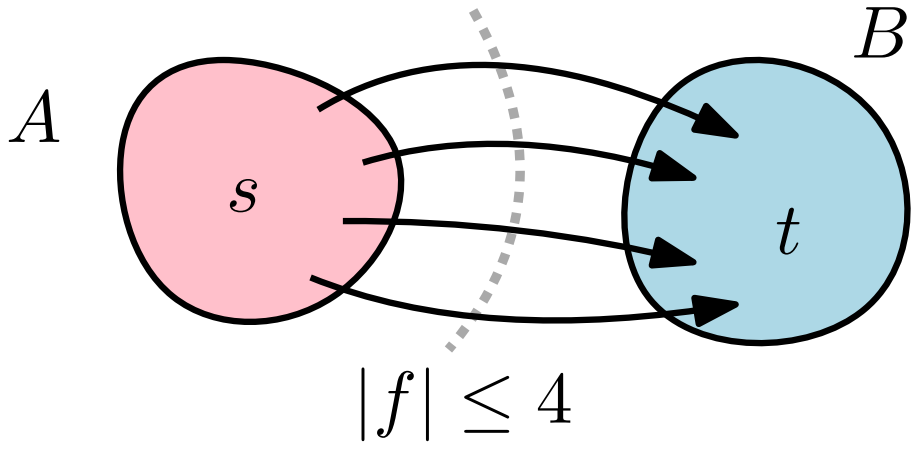
Minimum s–t Cut

Claim: Let (A, B) be an s–t cut. $|f| \leq \mathsf{cap}(A, B)$.



Minimum s–t Cut

Claim: Let (A, B) be an s–t cut. $|f| \leq \mathsf{cap}(A, B)$.



Proof:

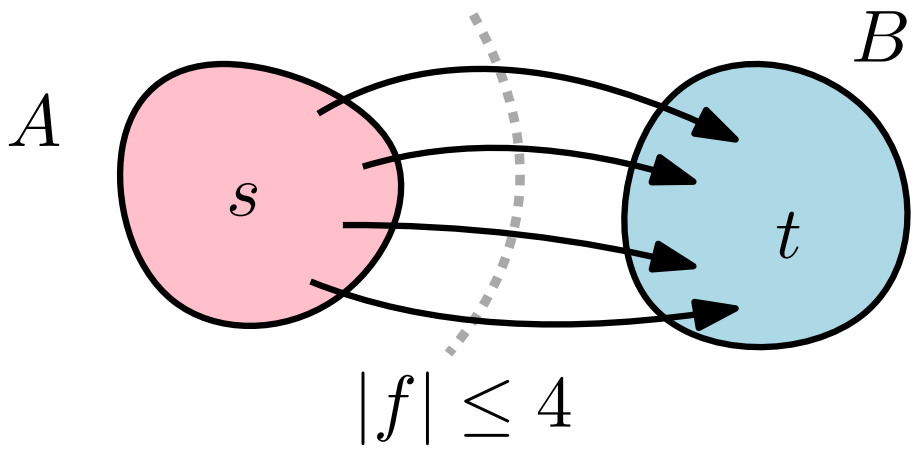
$$|f| = f^{\mathsf{out}}(A) - f^{\mathsf{in}}(A) \leq f^{\mathsf{out}}(A) =$$

$$= \sum_{\substack{e=(u,v)\in E \\ u\in A, v\in V\setminus A}} f(e) \leq \sum_{\substack{e=(u,v)\in E \\ u\in A, v\in V\setminus A}} c(e) = \mathsf{cap}(A, B).$$

□

Minimum s–t Cut

Claim: Let (A, B) be an s–t cut. $|f| \leq \mathsf{cap}(A, B)$.



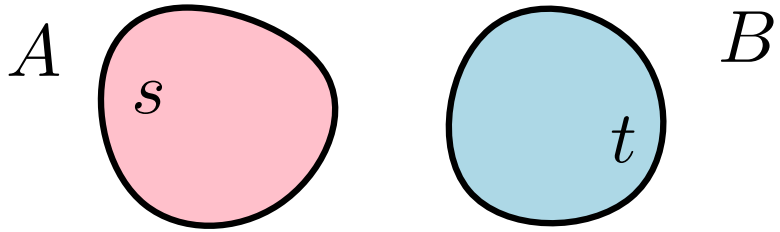
Corollary: $$|f| \leq \min_{s\text{–}t \text{ cut } (A,B)} \mathsf{cap}(A, B) = \mathsf{cap}(A^*, B^*).$$

Minimum s–t Cut

Claim: There is an s–t cut (A, B) such that $|f| \geq \text{cap}(A, B)$.

Proof:

- The residual graph G_f for f has no augmenting path.
- Let A be the vertices reachable from s in G_f and $B = V \setminus A$.
- Clearly $s \in A$ and $t \in B \implies (A, B)$ is an s–t cut.

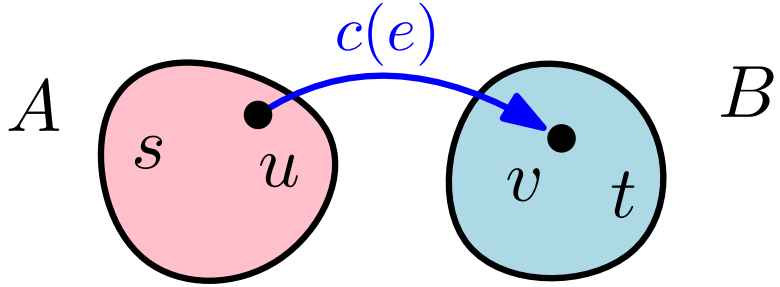


Minimum s–t Cut

Claim: There is an s–t cut (A, B) such that $|f| \geq \text{cap}(A, B)$.

Proof:

- The residual graph G_f for f has no augmenting path.
- Let A be the vertices reachable from s in G_f and $B = V \setminus A$.
- Clearly $s \in A$ and $t \in B \implies (A, B)$ is an s–t cut.



- If $e = (u, v) \in E$ with $u \in A$ and $v \in B$, $f(e) = c(e)$.

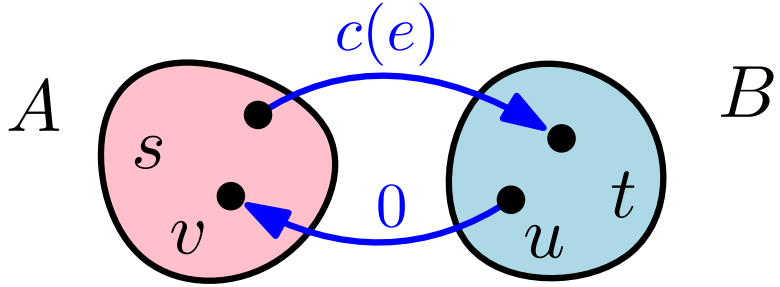
(otherwise (u, v) is in G_f and $v \in A$).

Minimum s–t Cut

Claim: There is an s–t cut (A, B) such that $|f| \geq \text{cap}(A, B)$.

Proof:

- The residual graph G_f for f has no augmenting path.
- Let A be the vertices reachable from s in G_f and $B = V \setminus A$.
- Clearly $s \in A$ and $t \in B \implies (A, B)$ is an s–t cut.



- If $e = (u, v) \in E$ with $u \in A$ and $v \in B$, $f(e) = c(e)$.
- If $e = (u, v) \in E$ with $u \in B$ and $v \in A$, $f(e) = 0$.

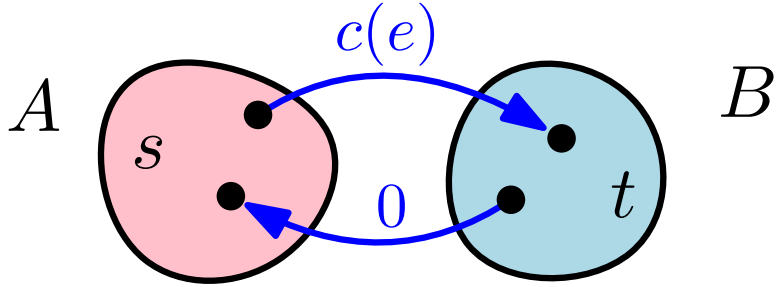
(otherwise (v, u) is in G_f and $u \in A$).

Minimum s–t Cut

Claim: There is an s–t cut (A, B) such that $|f| \geq \mathsf{cap}(A, B)$.

Proof:

- The residual graph G_f for f has no augmenting path.
- Let A be the vertices reachable from s in G_f and $B = V \setminus A$.
- Clearly $s \in A$ and $t \in B \implies (A, B)$ is an s–t cut.



- If $e = (u, v) \in E$ with $u \in A$ and $v \in B$, $f(e) = c(e)$.
- If $e = (u, v) \in E$ with $u \in B$ and $v \in A$, $f(e) = 0$.

$$|f| = f^{\mathsf{out}}(A) - f^{\mathsf{in}}(A) = \sum_{\substack{e=(u,v)\in E \\ u\in A, v\in B}} c(e) - 0 = \mathsf{cap}(A, B).$$

□

Minimum s–t Cut

We proved:

Claim: There is an s–t cut (A, B) such that $|f| \geq \mathsf{cap}(A, B)$.

Minimum s–t Cut

We proved:

Claim: There is an s–t cut (A, B) such that $|f| \geq \mathsf{cap}(A, B)$.

Corollary: $$|f| \geq \min_{s\text{–}t \text{ cut } (A,B)} \mathsf{cap}(A, B) = \mathsf{cap}(A^*, B^*).$$

Minimum s–t Cut

We proved:

Claim: There is an s–t cut (A, B) such that $|f| \geq \mathsf{cap}(A, B)$.

Corollary: $$|f| \geq \min_{s\text{–}t \text{ cut } (A,B)} \mathsf{cap}(A, B) = \mathsf{cap}(A^*, B^*).$$

$+$

Corollary: $$|f| \leq \min_{s\text{–}t \text{ cut } (A,B)} \mathsf{cap}(A, B) = \mathsf{cap}(A^*, B^*).$$

Minimum s–t Cut

We proved:

Claim: There is an s–t cut (A, B) such that $|f| \geq \mathsf{cap}(A, B)$.

Corollary: $$|f| \geq \min_{s\text{–}t \text{ cut } (A,B)} \mathsf{cap}(A, B) = \mathsf{cap}(A^*, B^*).$$

$+$

Corollary: $$|f| \leq \min_{s\text{–}t \text{ cut } (A,B)} \mathsf{cap}(A, B) = \mathsf{cap}(A^*, B^*).$$

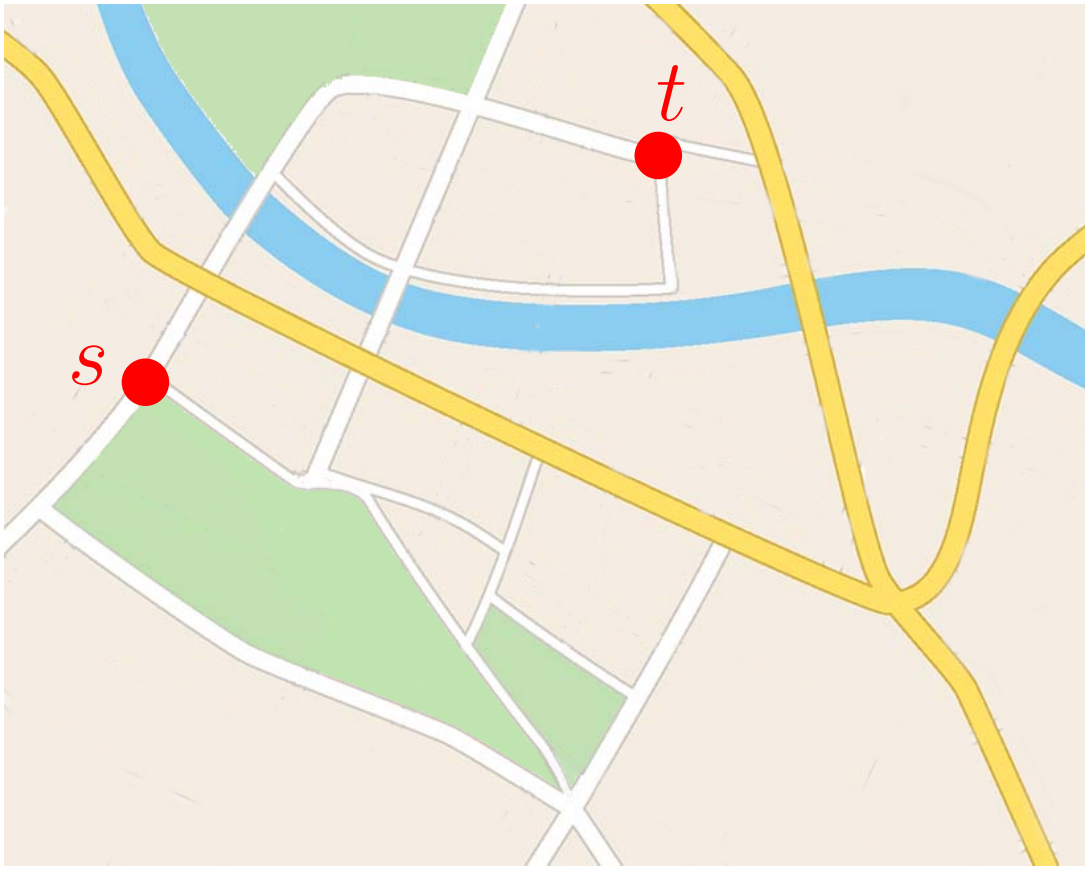
$\Downarrow$

$$|f| = \mathsf{cap}(A^*, B^*).$$

$\square$

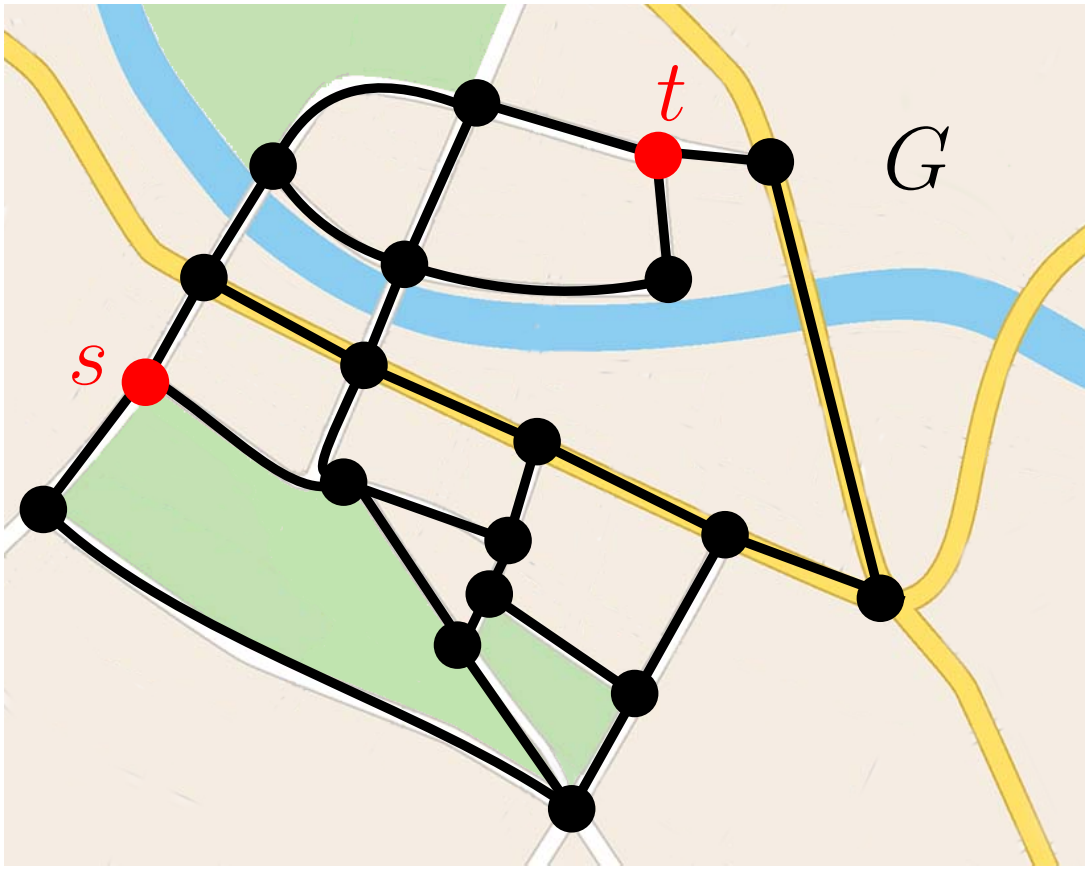
Edge Disjoint Paths

Goal: Find the maximum number of ways to go from s to t using each street at most once



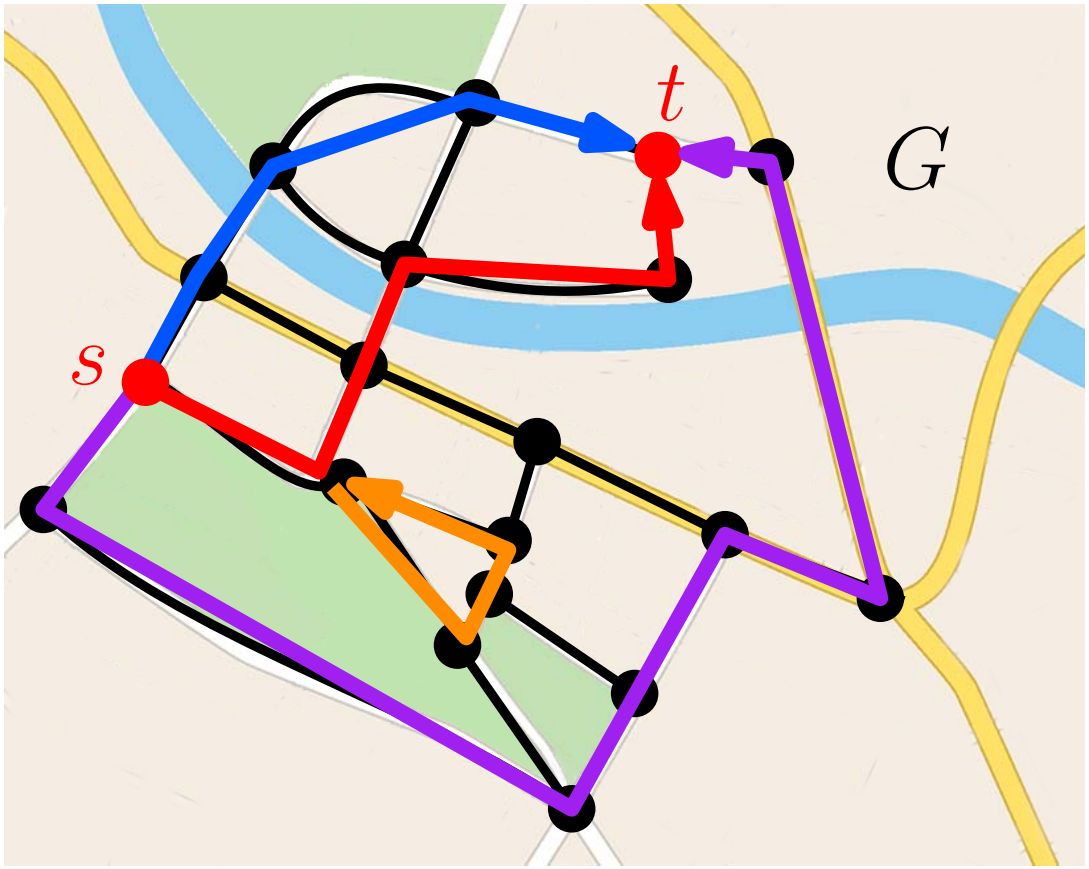
Edge Disjoint Paths

Goal: Find the maximum number η of *edge-disjoint* paths from s to t in G.



Edge Disjoint Paths

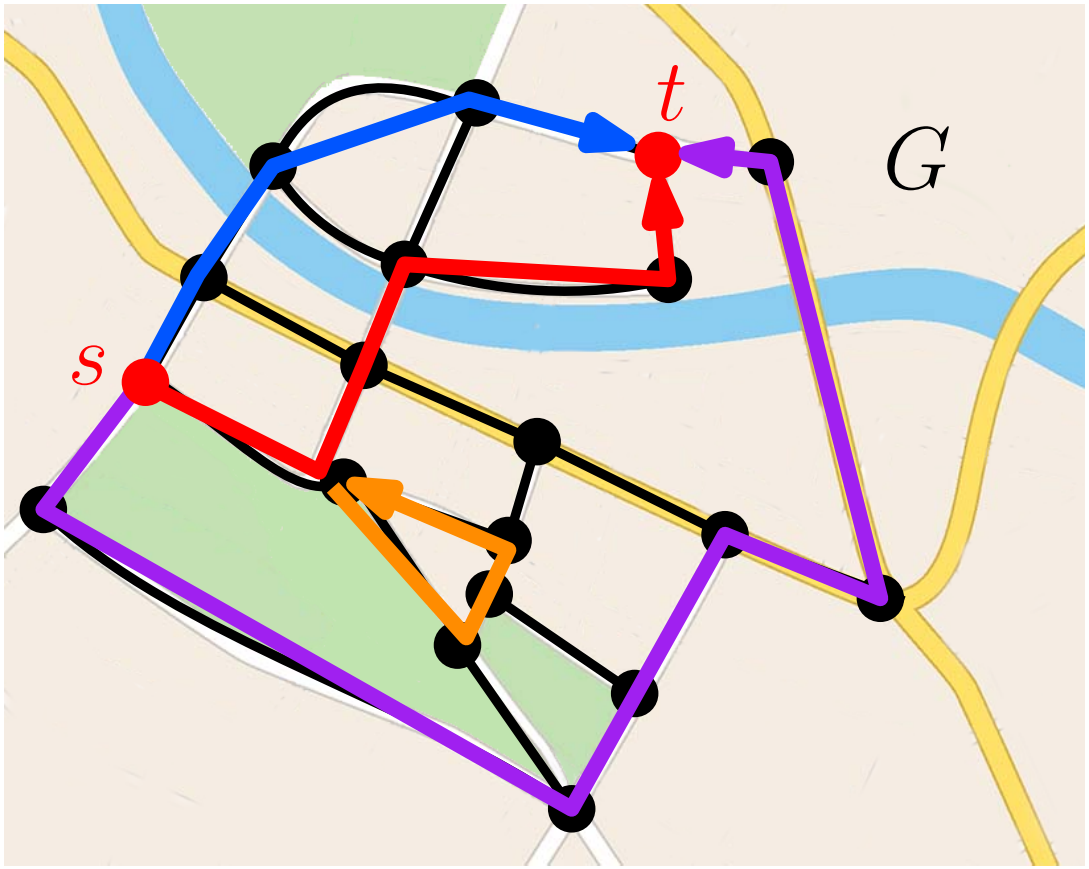
Goal: Find the maximum number η of *edge-disjoint* paths from s to t in G.



Let f be a maximum flow from s to t in G. Then, $\eta = |f|$.

Edge Disjoint Paths

Goal: Find the maximum number η of *edge-disjoint* paths from s to t in G.



Let f be a maximum flow from s to t in G. Then, $\eta = |f|$.

$|f| \leq n \implies$ Time complexity: $O(mn)$

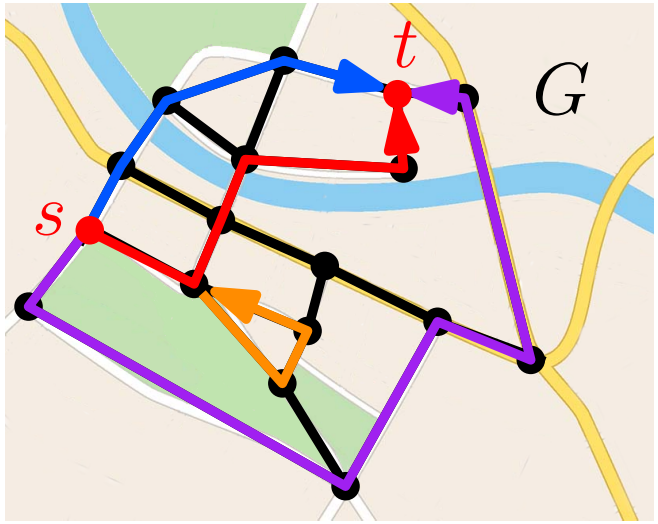
Edge Disjoint Paths

Flow decomposition: A flow f can be decomposed into

- $|f|$ paths from s to t, and

- A collection of cycles,

that are all edge-disjoint.



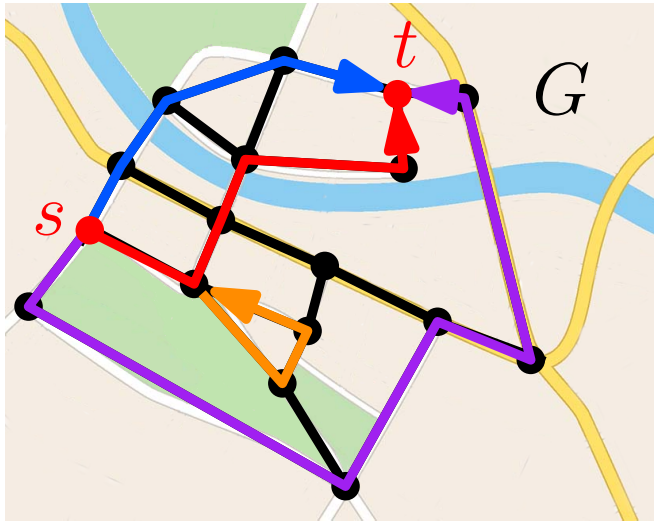
Edge Disjoint Paths

Flow decomposition: A flow f can be decomposed into

- $|f|$ paths from s to t, and

- A collection of cycles,

that are all edge-disjoint.



Finding the decomposition:

- Start from a graph G' with edges along the flow direction.
- Pick any edge (u, v) from G'
- Walk backwards from u and forward from v until a cycle or an s-t-path is found. Remove its edges. Repeat.

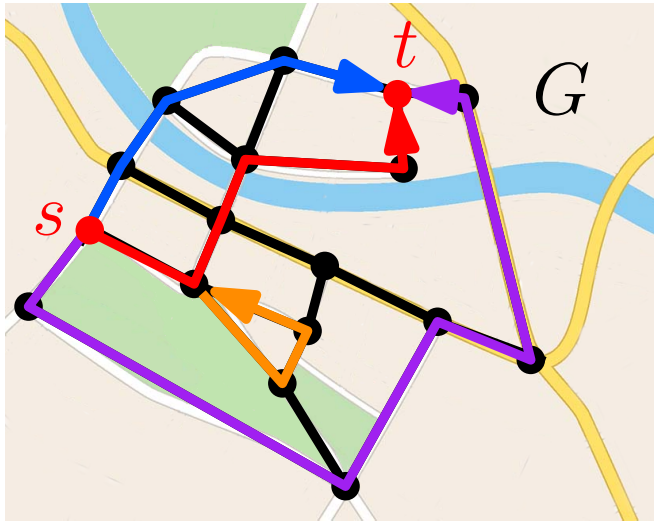
Edge Disjoint Paths

Flow decomposition: A flow f can be decomposed into

- $|f|$ paths from s to t, and

- A collection of cycles,

that are all edge-disjoint.



Time: $O(mn)$

Finding the decomposition: $O(m)$ iterations $O(n)$ time per it.

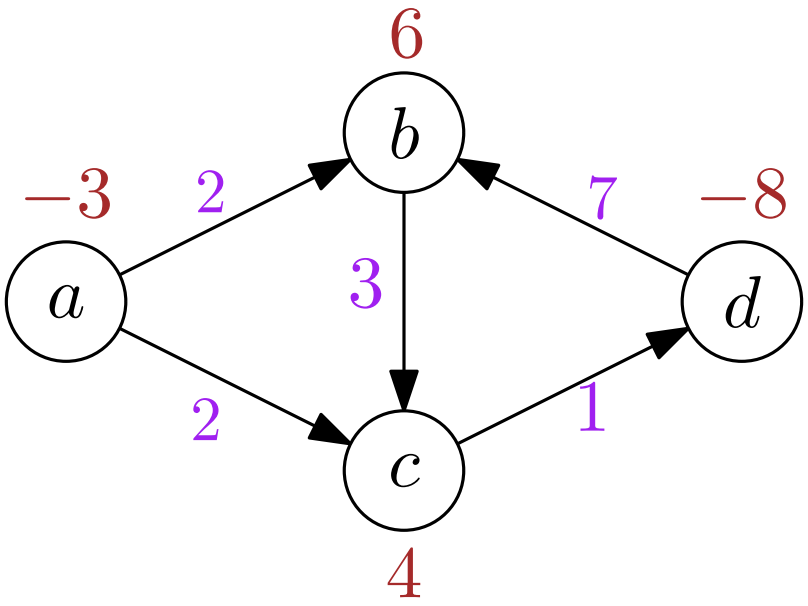
- Start from a graph G' with edges along the flow direction.
- Pick any edge (u, v) from G'
- Walk backwards from u and forward from v until a cycle or an s-t-path is found. Remove its edges. Repeat.

Circulation

In addition to edge capacities $c(e) \in \mathbb{N}$, each vertex v has an associated value $d_v \in \mathbb{Z}$

- If $d_v > 0$, vertex v has a *demand* of d_v units of flow.
- If $d_v < 0$, vertex v has a *supply* of $-d_v$ units of flow.

Is it possible to circulate flow so that all demands are met?

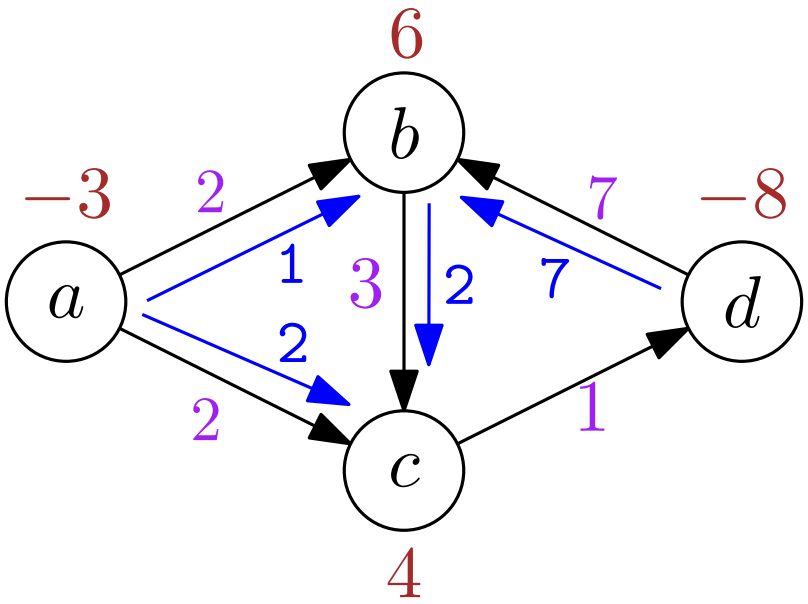


Circulation

In addition to edge capacities $c(e) \in \mathbb{N}$, each vertex v has an associated value $d_v \in \mathbb{Z}$

- If $d_v > 0$, vertex v has a *demand* of d_v units of flow.
- If $d_v < 0$, vertex v has a *supply* of $-d_v$ units of flow.

Is it possible to circulate flow so that all demands are met?

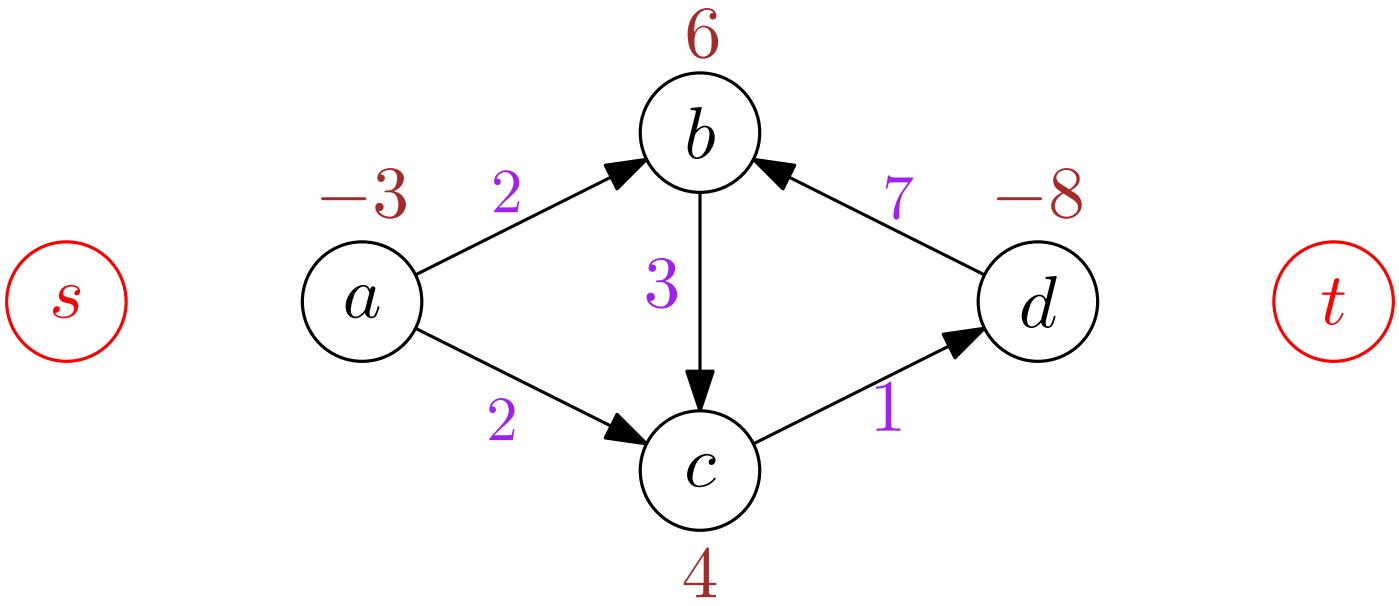


Circulation

In addition to edge capacities $c(e) \in \mathbb{N}$, each vertex v has an associated value $d_v \in \mathbb{Z}$

- If $d_v > 0$, vertex v has a *demand* of d_v units of flow.
- If $d_v < 0$, vertex v has a *supply* of $-d_v$ units of flow.

Is it possible to circulate flow so that all demands are met?

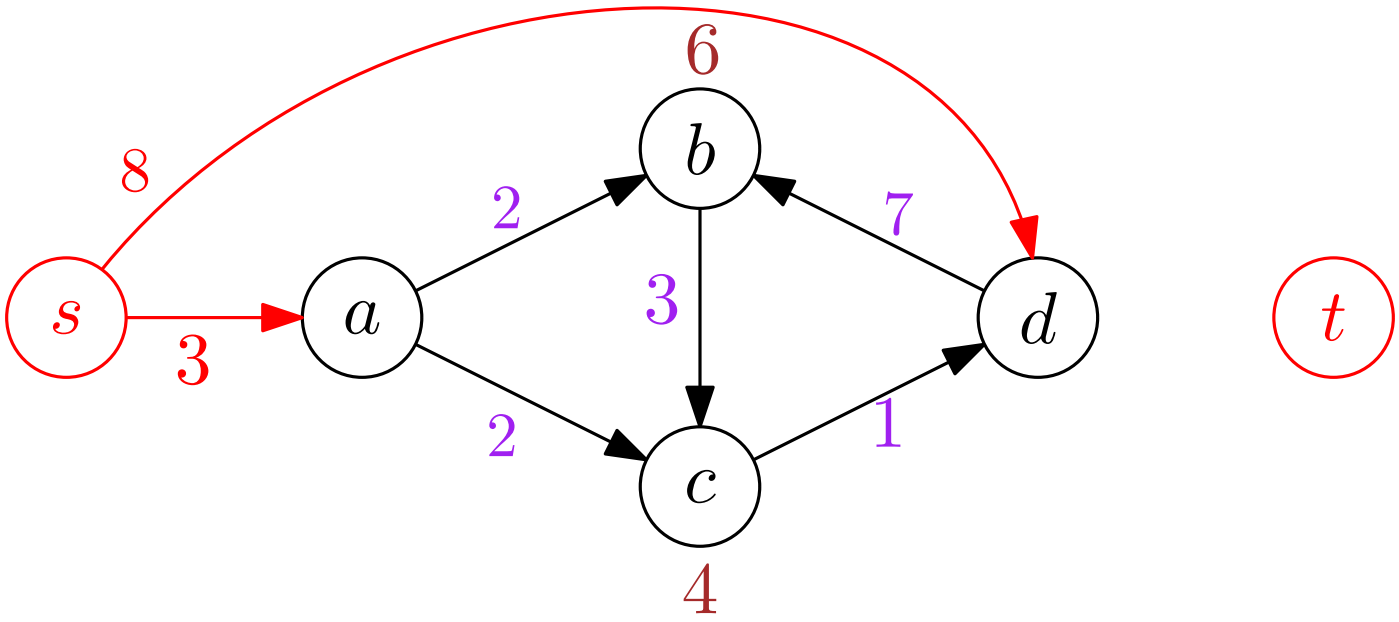


Circulation

In addition to edge capacities $c(e) \in \mathbb{N}$, each vertex v has an associated value $d_v \in \mathbb{Z}$

- If $d_v > 0$, vertex v has a *demand* of d_v units of flow.
- If $d_v < 0$, vertex v has a *supply* of $-d_v$ units of flow.

Is it possible to circulate flow so that all demands are met?

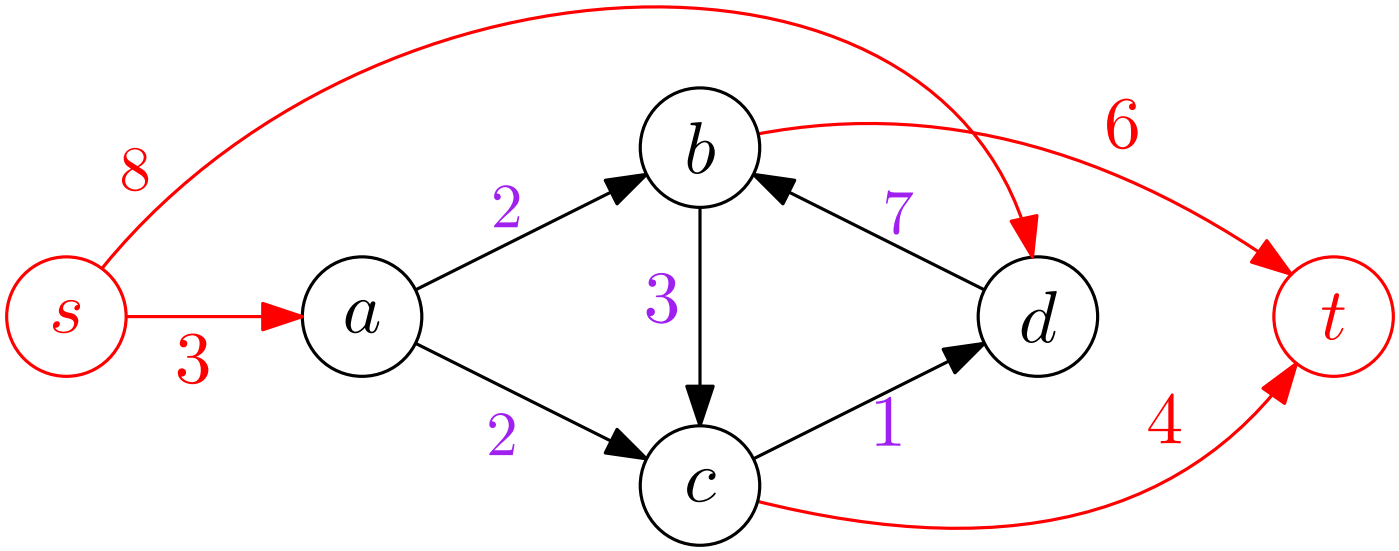


Circulation

In addition to edge capacities $c(e) \in \mathbb{N}$, each vertex v has an associated value $d_v \in \mathbb{Z}$

- If $d_v > 0$, vertex v has a *demand* of d_v units of flow.
- If $d_v < 0$, vertex v has a *supply* of $-d_v$ units of flow.

Is it possible to circulate flow so that all demands are met?

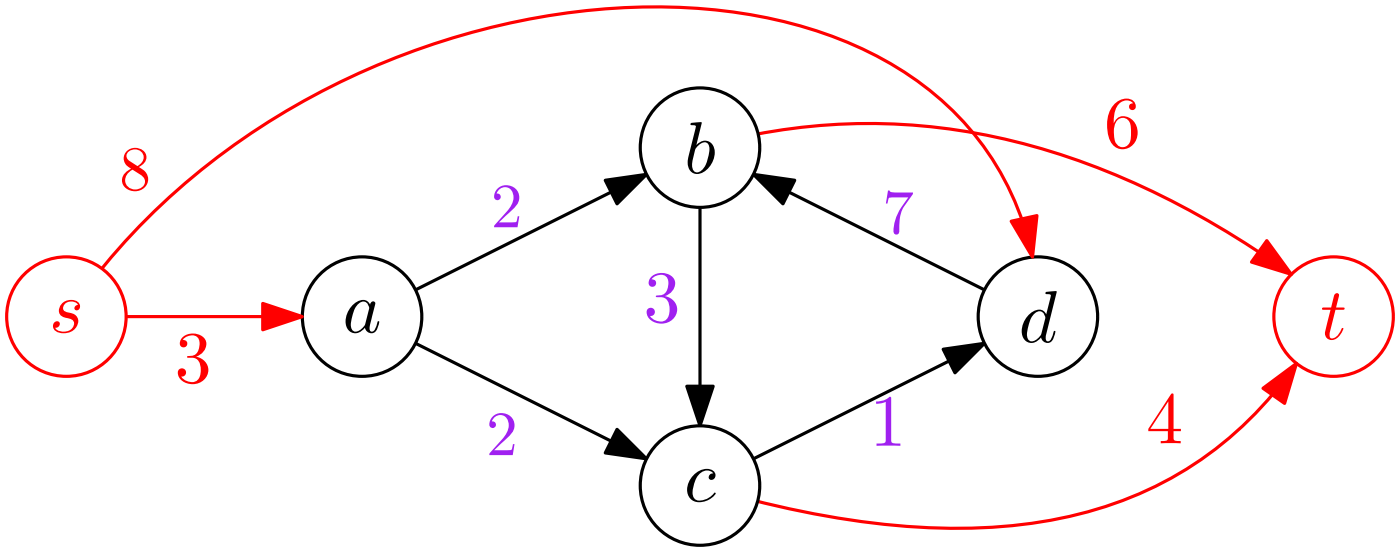


Circulation

In addition to edge capacities $c(e) \in \mathbb{N}$, each vertex v has an associated value $d_v \in \mathbb{Z}$

- If $d_v > 0$, vertex v has a *demand* of d_v units of flow.
- If $d_v < 0$, vertex v has a *supply* of $-d_v$ units of flow.

Is it possible to circulate flow so that all demands are met?

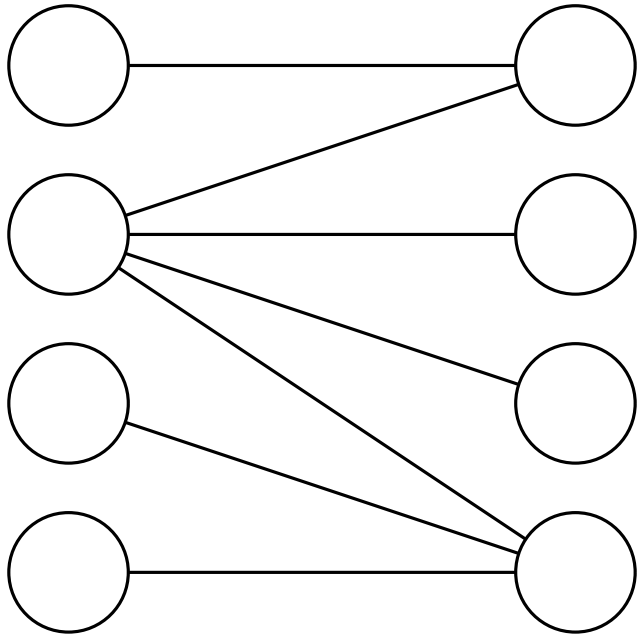


Compute maximum flow f and check if $|f| = \sum_{(v,t)} c(v,t)$.

Maximum Bipartite Matching

$M \subseteq E$ is a matching if no two edges in M share an endvertex

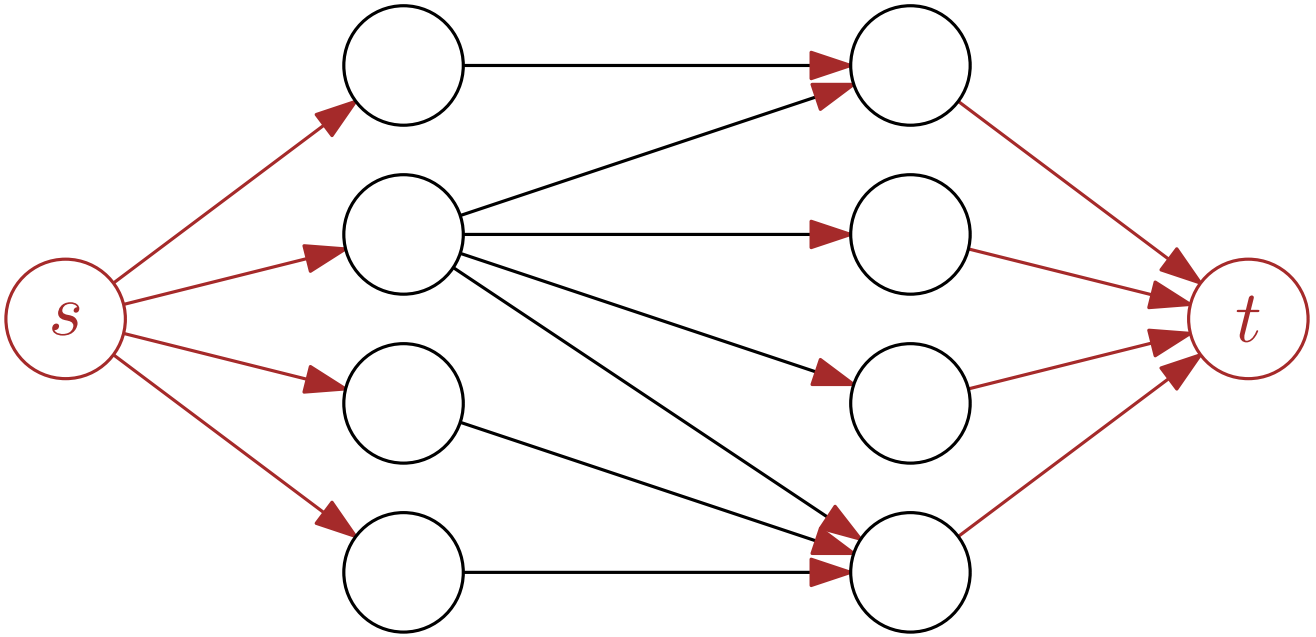
Goal: find a maximum-cardinality matching.



Maximum Bipartite Matching

$M \subseteq E$ is a matching if no two edges in M share an endvertex

Goal: find a maximum-cardinality matching.

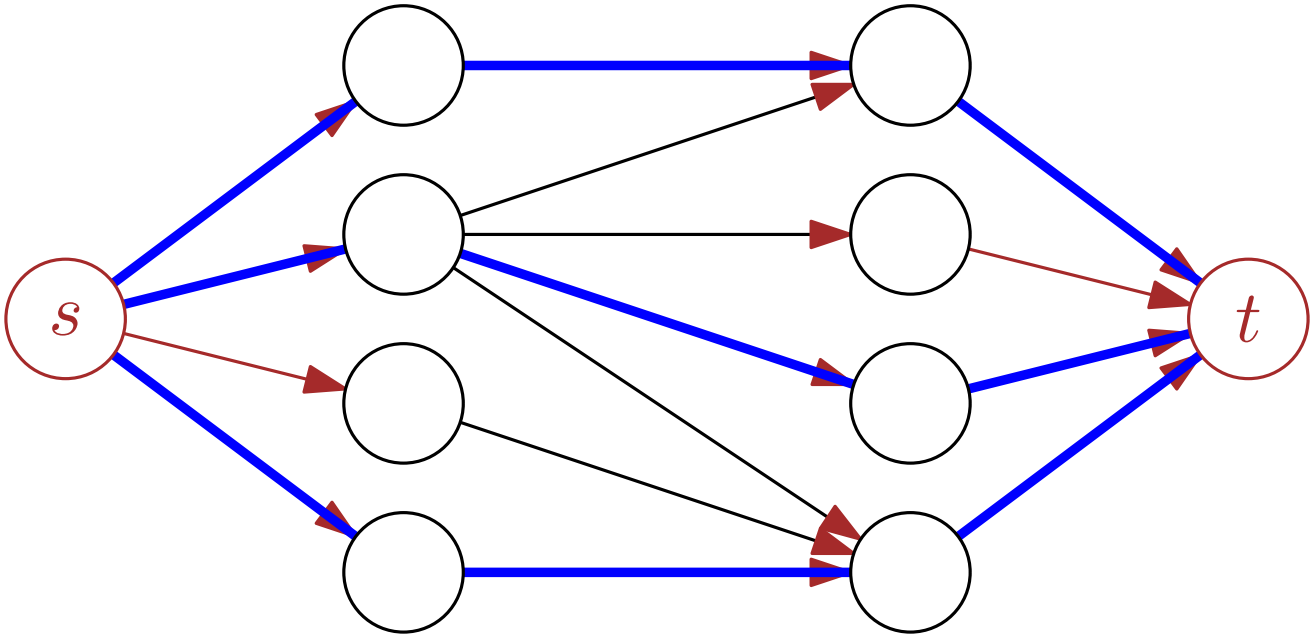


All capacities are 1

Maximum Bipartite Matching

$M \subseteq E$ is a matching if no two edges in M share an endvertex

Goal: find a maximum-cardinality matching.

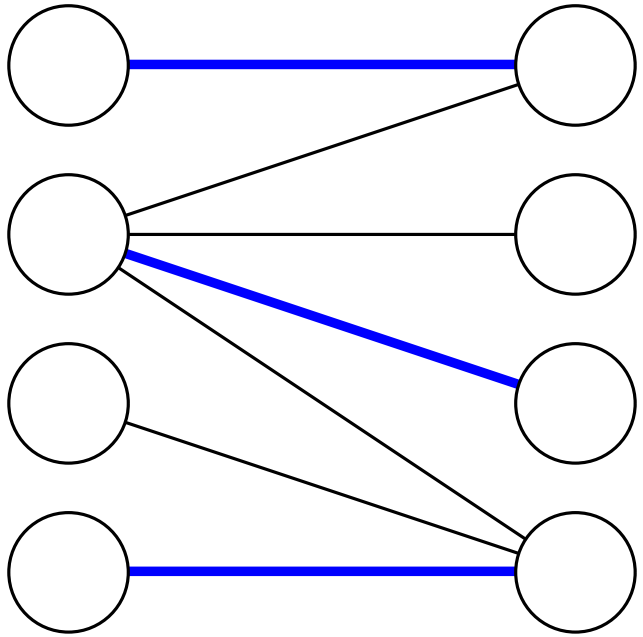


All capacities are 1

Maximum Bipartite Matching

$M \subseteq E$ is a matching if no two edges in M share an endvertex

Goal: find a maximum-cardinality matching.

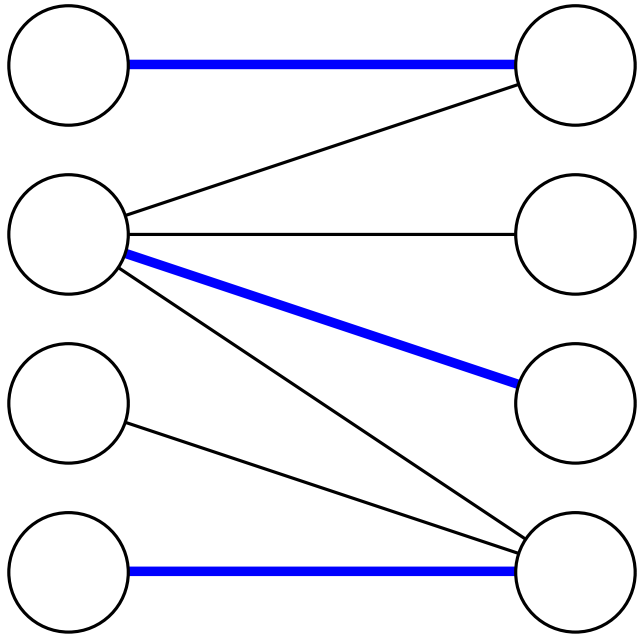


Size of a maximum-cardinality matching: 3

Maximum Bipartite Matching

$M \subseteq E$ is a matching if no two edges in M share an endvertex

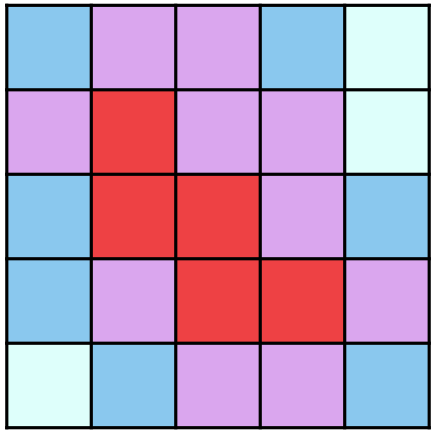
Goal: find a maximum-cardinality matching.



Size of a maximum-cardinality matching: 3

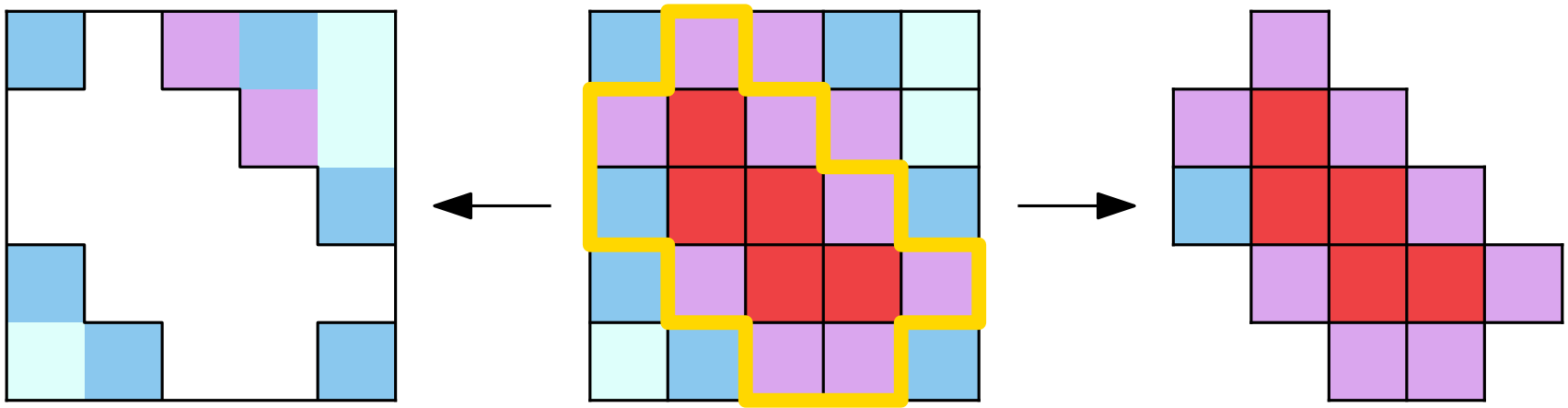
Kőnig's theorem: on bipartite graphs, the cardinality of a maximum matching is the size of a minimum vertex cover.

Image Segmentation



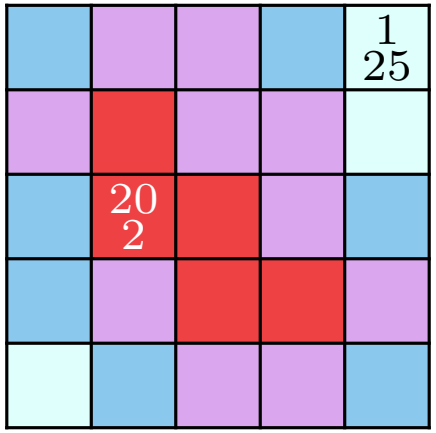
Goal (inf): segment an image into *background* and *foreground*

Image Segmentation



Goal (inf): segment an image into *background* and *foreground*

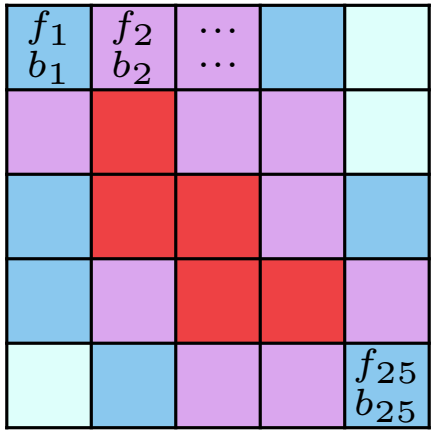
Image Segmentation



Goal (inf): segment an image into *background* and *foreground*

- Each pixel i has an associated likelihood f_i (resp. b_i) to be in the foreground (resp. background)

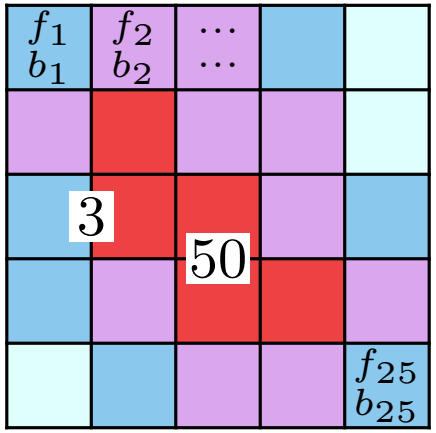
Image Segmentation



Goal (inf): segment an image into *background* and *foreground*

- Each pixel i has an associated likelihood f_i (resp. b_i) to be in the foreground (resp. background)

Image Segmentation

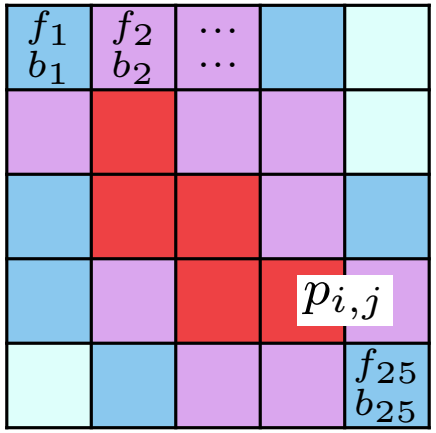


Goal (inf): segment an image into *background* and *foreground*

- Each pixel i has an associated likelihood f_i (resp. b_i) to be in the foreground (resp. background)
- Each pair i, j of adjacent pixels have an associated *separation penalty* $p_{i,j}$

  This penalty is incurred if one pixel is in the background and the other is in the foreground

Image Segmentation

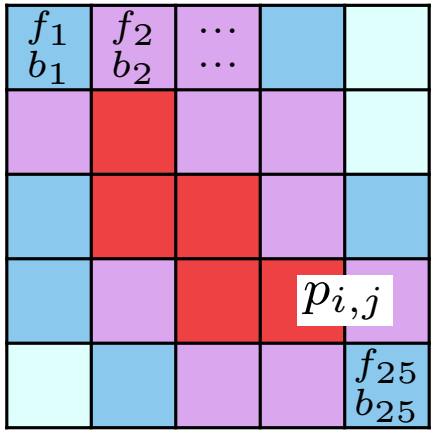


Goal (inf): segment an image into *background* and *foreground*

- Each pixel i has an associated likelihood f_i (resp. b_i) to be in the foreground (resp. background)
- Each pair i, j of adjacent pixels have an associated *separation penalty* $p_{i,j}$

  This penalty is incurred if one pixel is in the background and the other is in the foreground

Image Segmentation



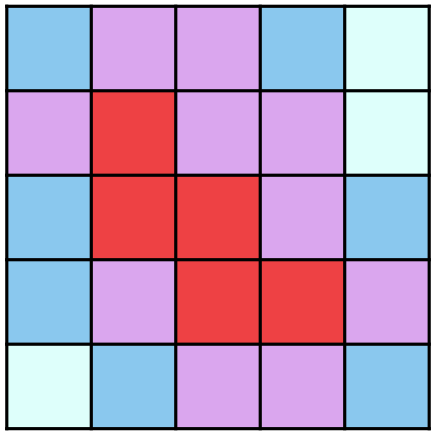
Goal (inf): segment an image into *background* and *foreground*

- Each pixel i has an associated likelihood f_i (resp. b_i) to be in the foreground (resp. background)
- Each pair i, j of adjacent pixels have an associated *separation penalty* $p_{i,j}$

Goal: find a partition F, B of the pixels maximizing

$$\sum_{i \in F} f_i + \sum_{i \in B} b_i - \sum_{\substack{\text{adjacent } i,j \\ |F \cap \{i,j\}|=1}} p_{i,j}$$

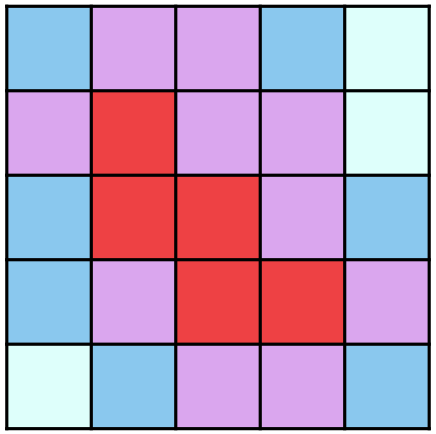
Image Segmentation



Goal: find a partition F, B of the pixels maximizing

$$\sum_{i \in F} f_i + \sum_{i \in B} b_i - \sum_{\substack{\text{adjacent } i,j \\ |F \cap \{i,j\}|=1}} p_{i,j}$$

Image Segmentation

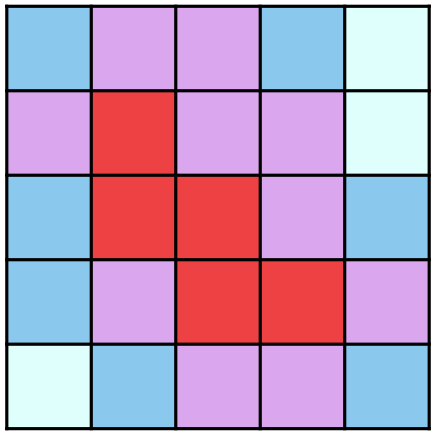


Goal: find a partition F, B of the pixels maximizing

$$\sum_{i \in F} f_i + \sum_{i \in B} b_i - \sum_{\substack{\text{adjacent } i,j \\ |F \cap \{i,j\}|=1}} p_{i,j}$$

$$\sum_i f_i - \sum_{i \in B} f_i$$

Image Segmentation

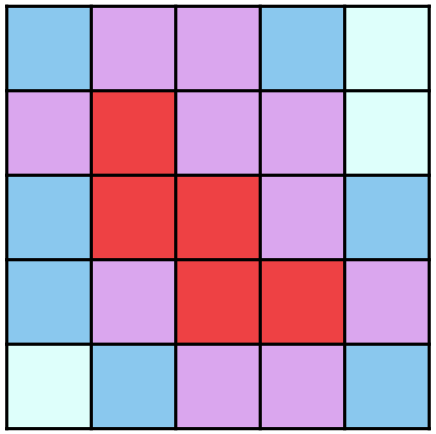


Goal: find a partition F, B of the pixels maximizing

$$\sum_{i \in F} f_i + \sum_{i \in B} b_i - \sum_{\substack{\text{adjacent } i,j \\ |F \cap \{i,j\}|=1}} p_{i,j}$$

$\sum_i f_i - \sum_{i \in B} f_i$ $\qquad$ $\sum_i b_i - \sum_{i \in F} b_i$

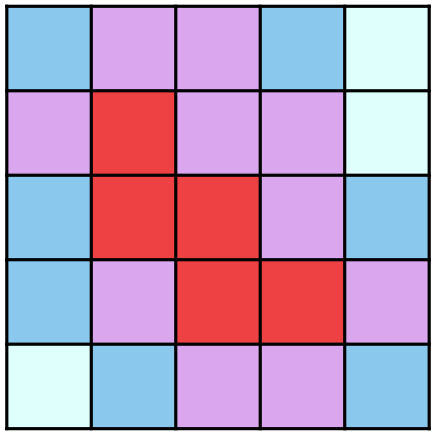
Image Segmentation



Goal: find a partition F, B of the pixels maximizing

$$\sum_i (f_i + b_i) - \sum_{i \in B} f_i - \sum_{i \in F} b_i - \sum_{\substack{\text{adjacent } i,j \\ |F \cap \{i,j\}|=1}} p_{i,j}$$

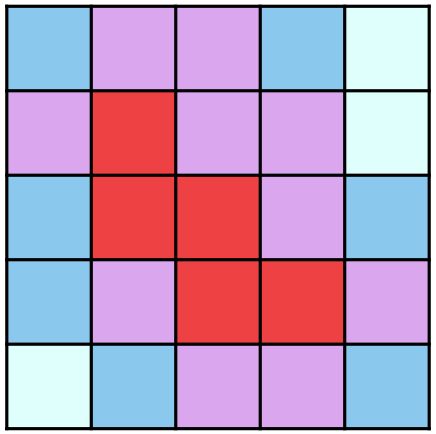
Image Segmentation



Goal: find a partition F, B of the pixels maximizing

$$\cancel{\sum_i (f_i + b_i)} - \sum_{i \in B} f_i - \sum_{i \in F} b_i - \sum_{\substack{\text{adjacent } i,j \\ |F \cap \{i,j\}|=1}} p_{i,j}$$

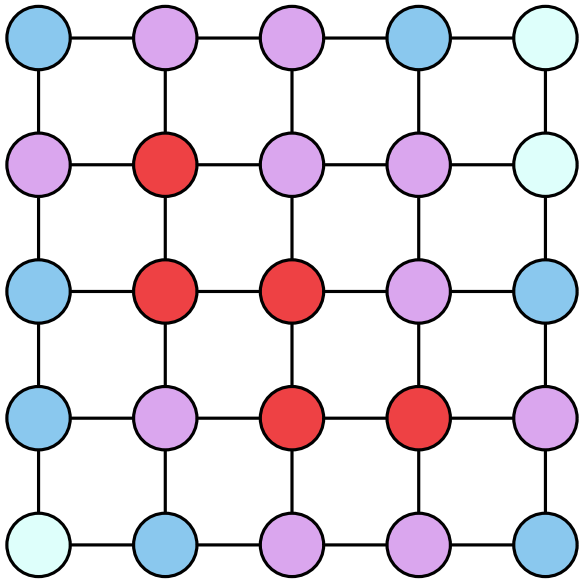
Image Segmentation



Equivalent goal: find a partition F, B of the pixels **minimizing**

$$\sum_{i \in B} f_i + \sum_{i \in F} b_i + \sum_{\substack{\text{adjacent } i,j \\ |F \cap \{i,j\}|=1}} p_{i,j}$$

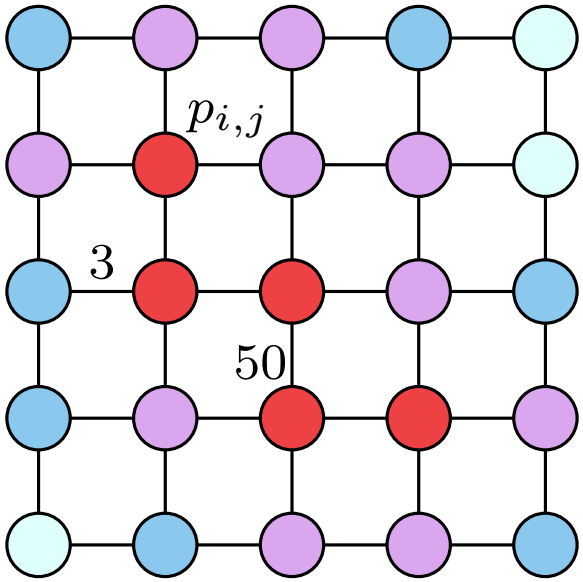
Image Segmentation



Equivalent goal: find a partition F, B of the pixels **minimizing**

$$\sum_{i \in B} f_i + \sum_{i \in F} b_i + \sum_{\substack{\text{adjacent } i,j \\ |F \cap \{i,j\}|=1}} p_{i,j}$$

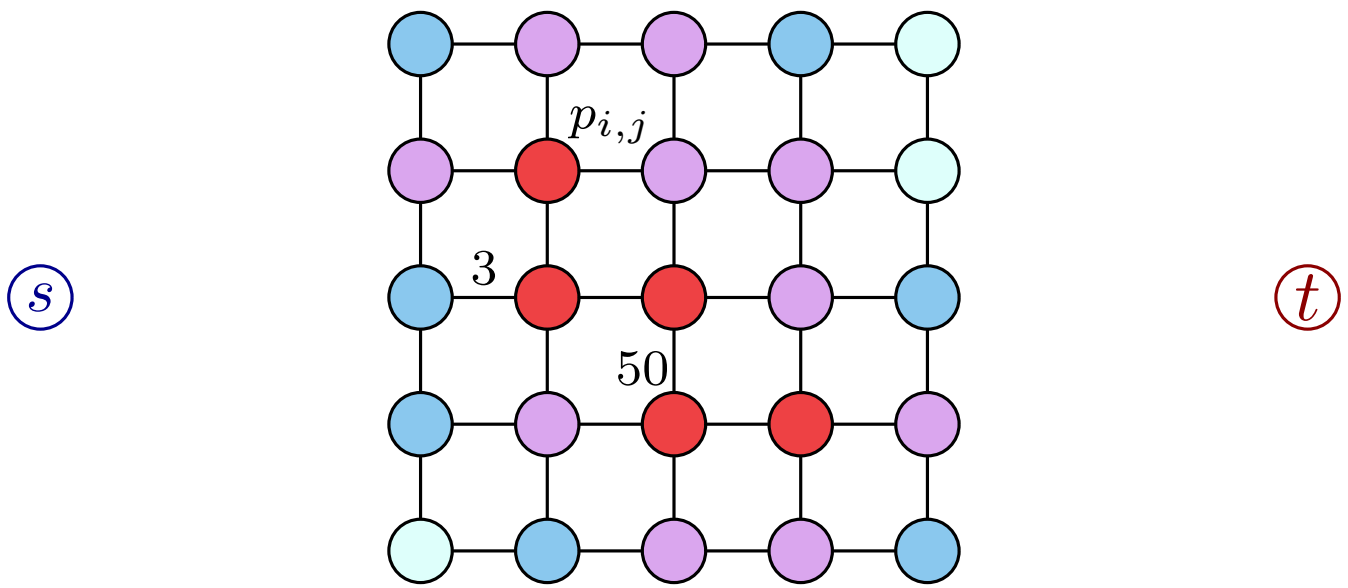
Image Segmentation



Equivalent goal: find a partition F, B of the pixels **minimizing**

$$\sum_{i \in B} f_i + \sum_{i \in F} b_i + \sum_{\substack{\text{adjacent } i,j \\ |F \cap \{i,j\}|=1}} p_{i,j}$$

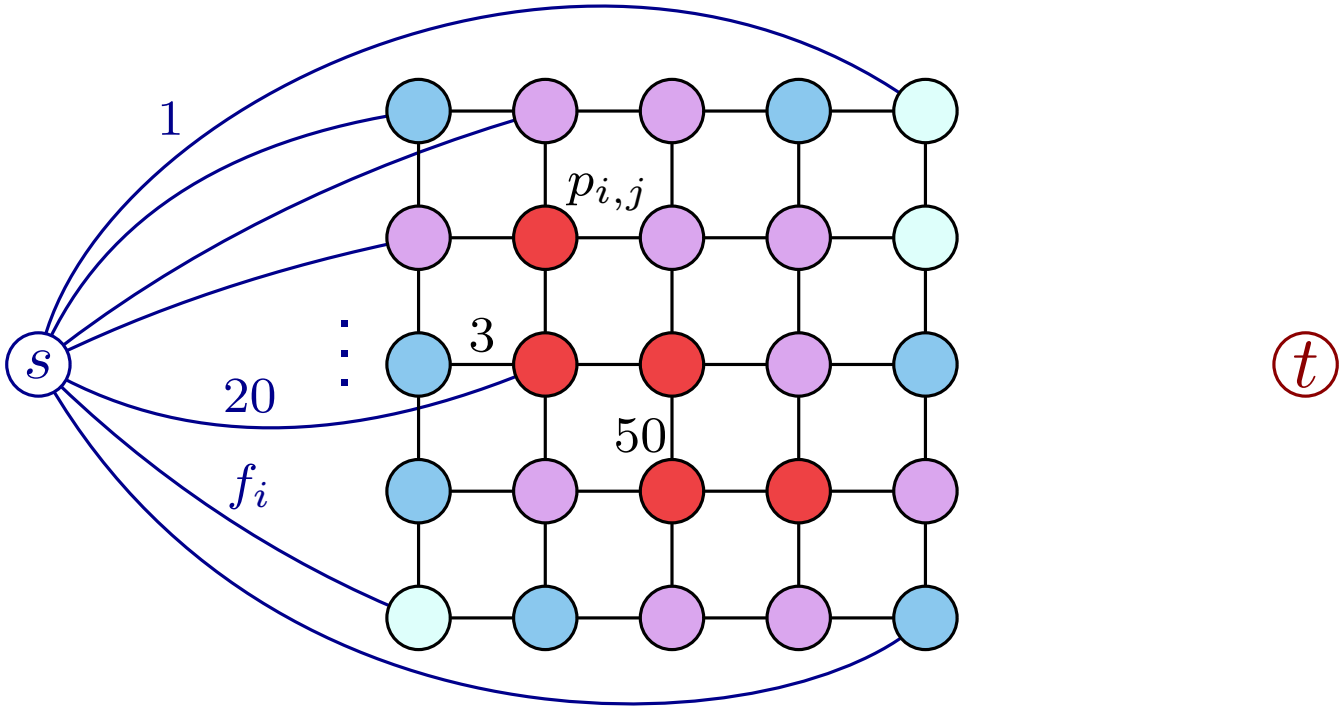
Image Segmentation



Equivalent goal: find a partition F, B of the pixels **minimizing**

$$\sum_{i \in B} f_i + \sum_{i \in F} b_i + \sum_{\substack{\text{adjacent } i,j \\ |F \cap \{i,j\}|=1}} p_{i,j}$$

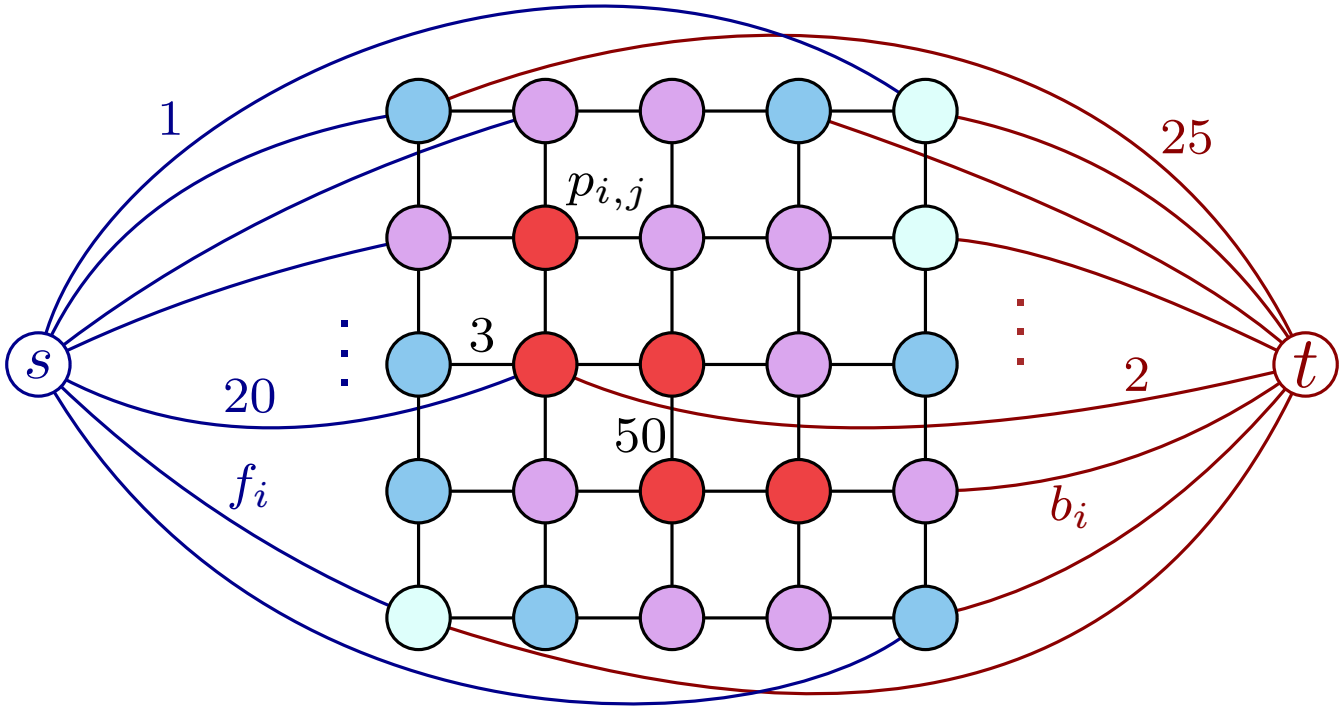
Image Segmentation



Equivalent goal: find a partition F, B of the pixels **minimizing**

$$\sum_{i \in B} f_i + \sum_{i \in F} b_i + \sum_{\substack{\text{adjacent } i,j \\ |F \cap \{i,j\}|=1}} p_{i,j}$$

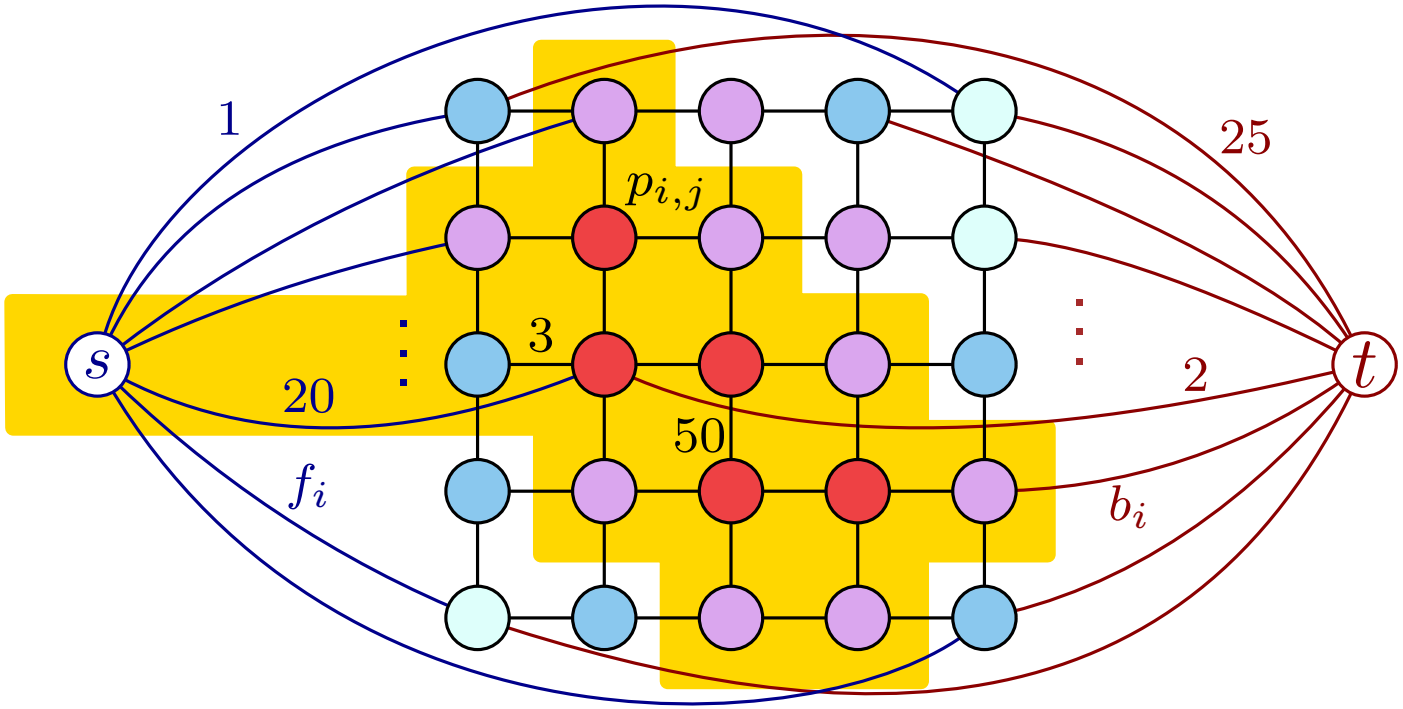
Image Segmentation



Equivalent goal: find a partition F, B of the pixels **minimizing**

$$\sum_{i \in B} f_i + \sum_{i \in F} b_i + \sum_{\substack{\text{adjacent } i,j \\ |F \cap \{i,j\}|=1}} p_{i,j}$$

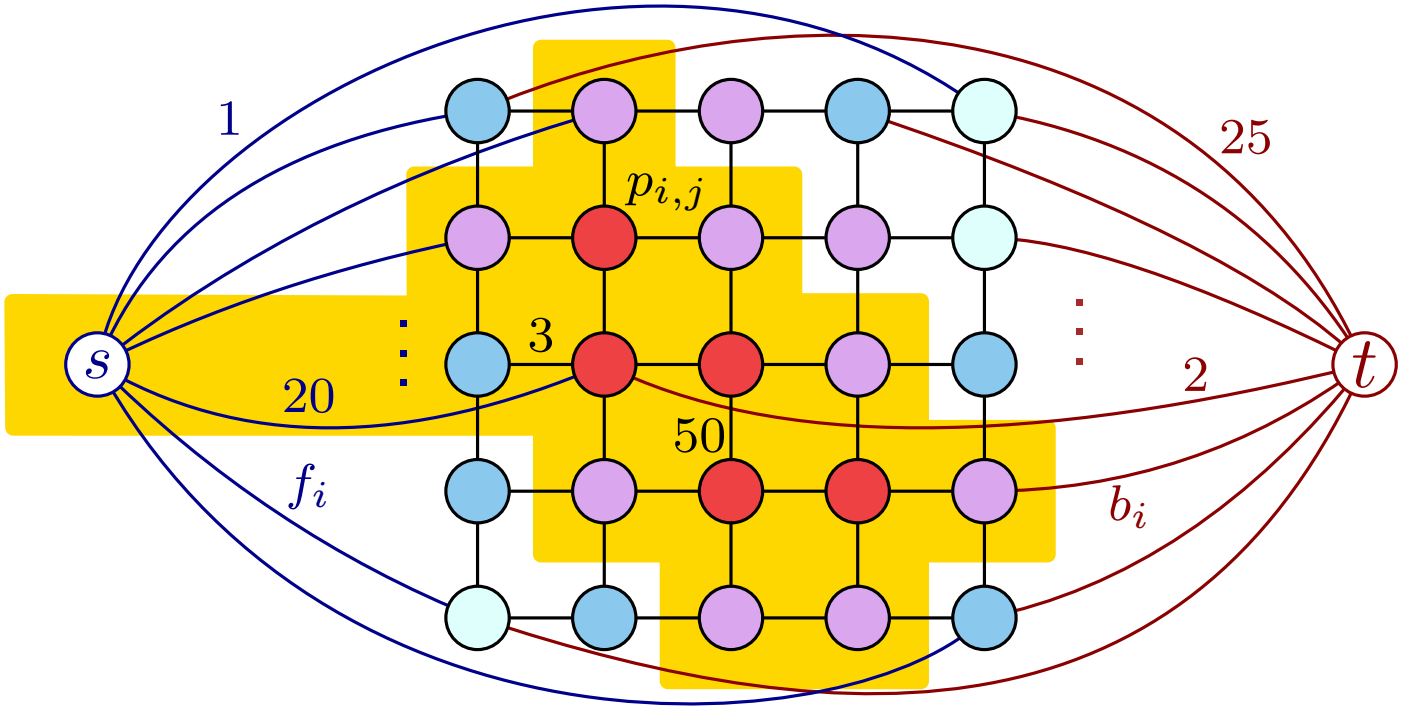
Image Segmentation



Equivalent goal: find a partition F, B of the pixels **minimizing**

$$\sum_{i \in B} f_i + \sum_{i \in F} b_i + \sum_{\substack{\text{adjacent } i,j \\ |F \cap \{i,j\}|=1}} p_{i,j}$$

Image Segmentation



Equivalent goal: find a partition F, B of the pixels **minimizing**

$$\sum_{i \in B} f_i + \sum_{i \in F} b_i + \sum_{\substack{\text{adjacent } i,j \\ |F \cap \{i,j\}|=1}} p_{i,j}$$

Solution: Compute a minimum s-t-cut (F', B'),
pick $F = F' \setminus \{s\}$ and $B = B' \setminus \{t\}$

Min-Cost Max-Flow

Min-Cost Max-Flow

Input:

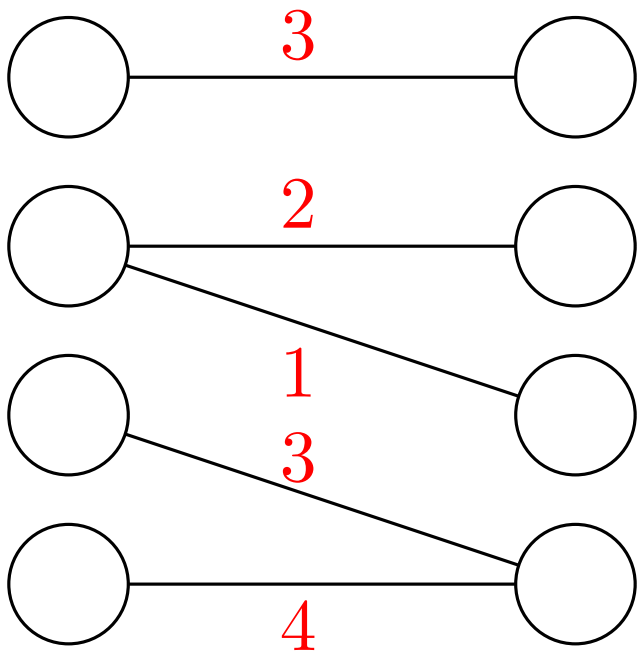
- A directed graph $G = (V, E)$
- A source vertex $s \in V$, with no incoming edges
- A target vertex $t \in V$, with no outgoing edges
- A function cap : $E \to \mathbb{N}$ that maps each edge to its *capacity*
- A function cost : $E \to \mathbb{Z}$ that maps each edge to its *cost*

Output:

A flow f that minimizes $cost(f) = \sum_{e \in E} f(e) \cdot \texttt{cost}(e)$ chosen among all s–t flows that maximize $|f|$.

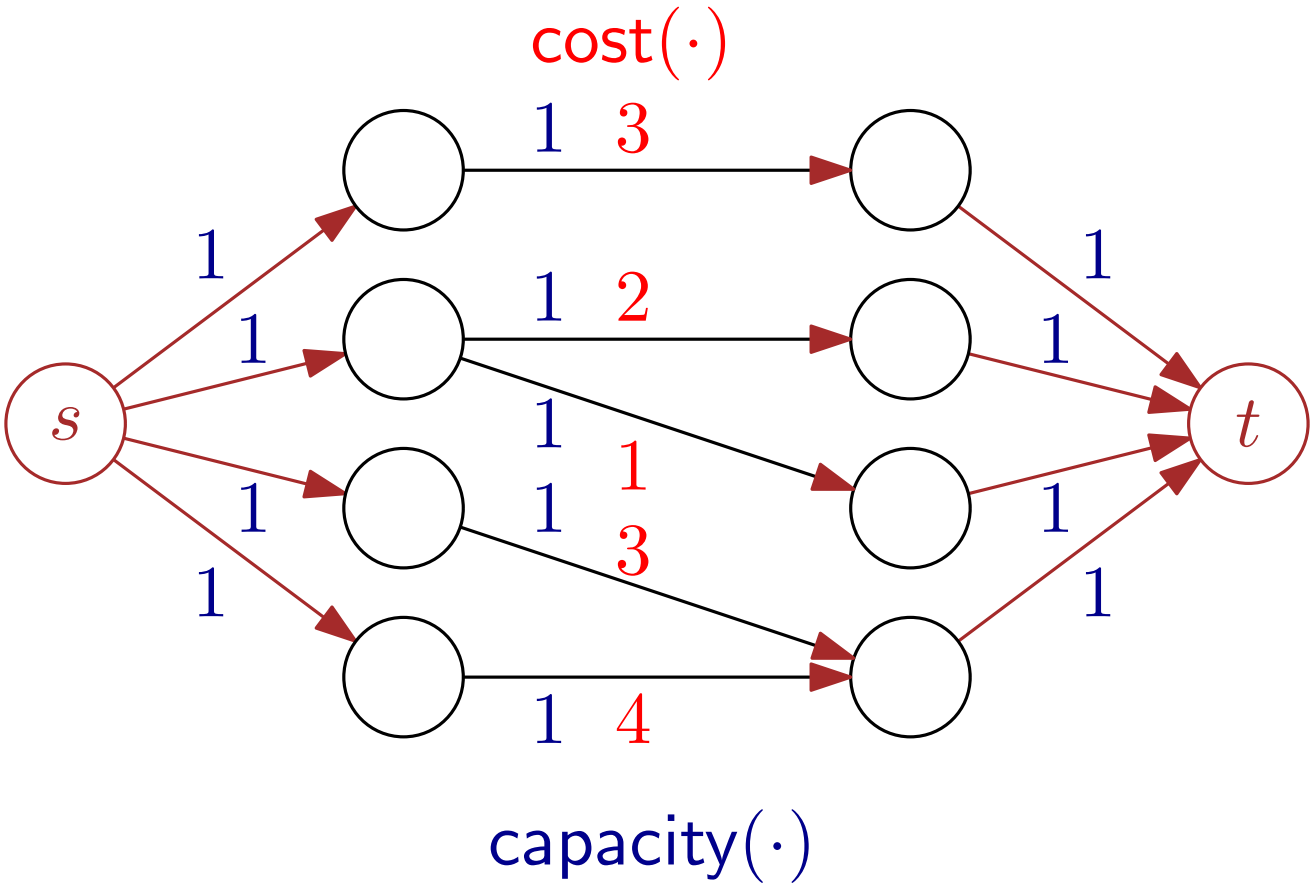
Minimum Cost Bipartite Matching

Goal: find a maximum-cardinality matching of minimum cost.



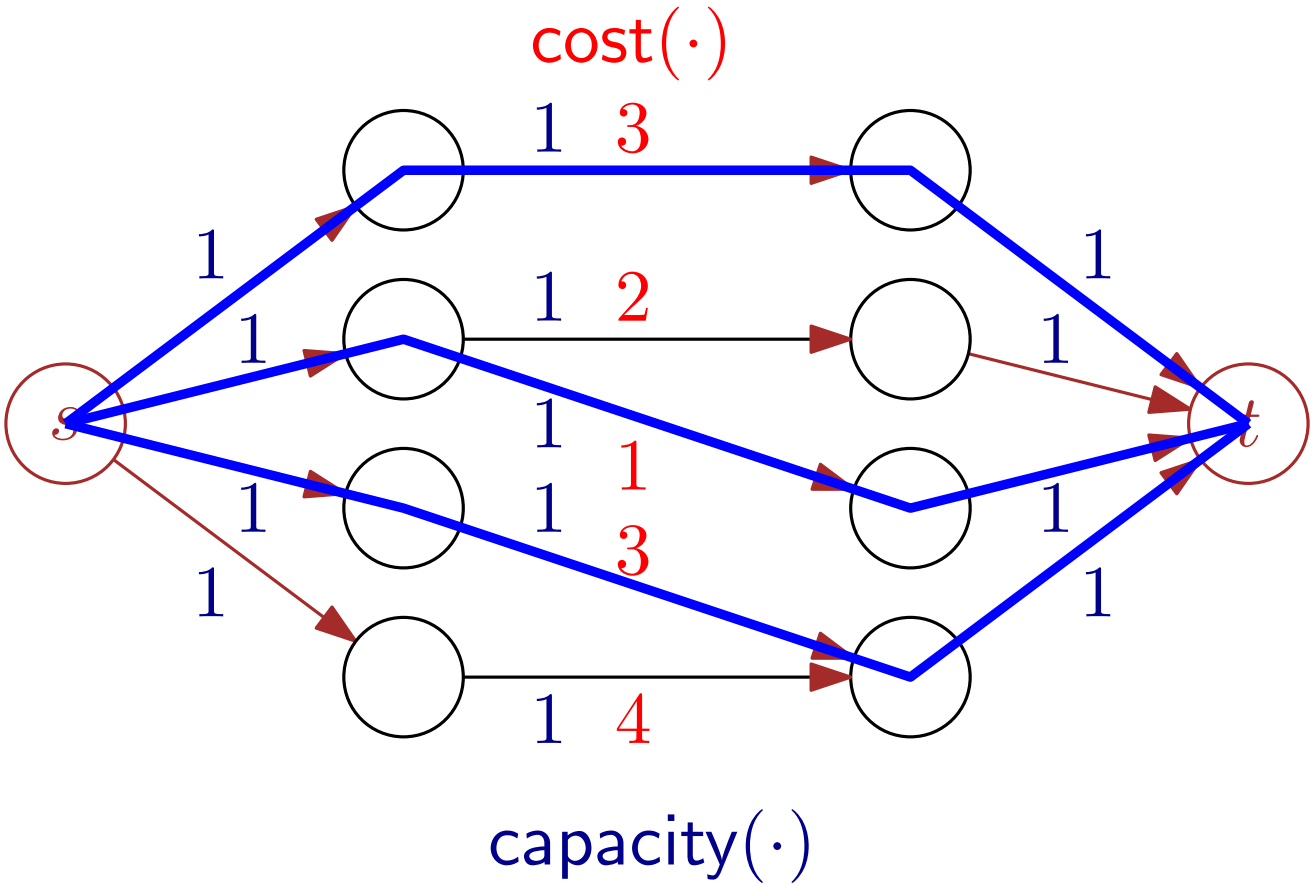
Minimum Cost Bipartite Matching

Goal: find a maximum-cardinality matching of minimum cost.



Minimum Cost Bipartite Matching

Goal: find a maximum-cardinality matching of minimum cost.



$|f| = 3$

$\text{cost}(f) = 3 + 1 + 3 = 7$

Oil Delivery

Oil needs to be delivered from refineries to gas stations.

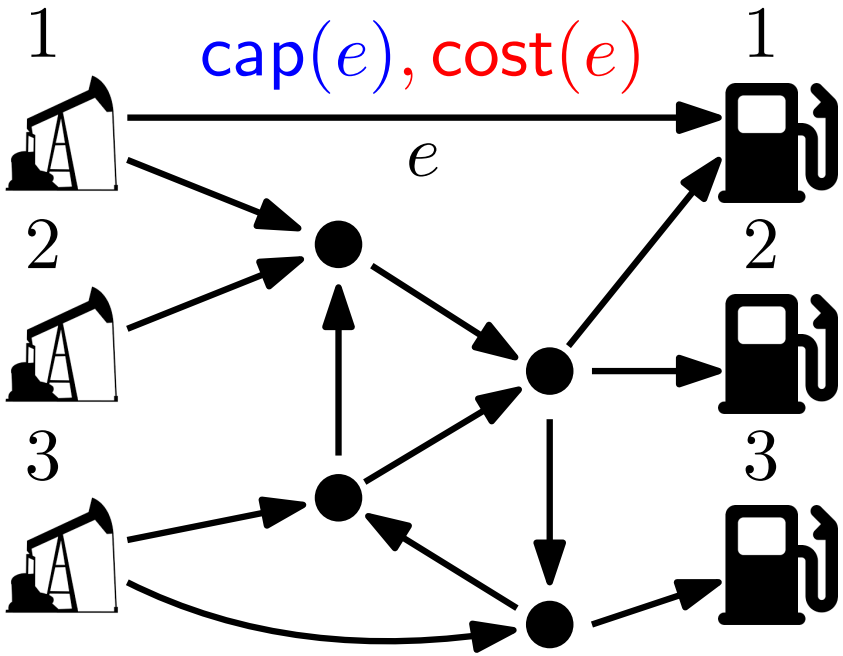
Refinery i produces s_i units of oil.

Gas station j needs d_j units of oil.

Using a truck to transport 1 unit oil across road e costs $\mathsf{cost}(e)$.

At most $\mathsf{cap}(e)$ trucks per day can traverse road e.

Goal: satisfy all demands with minimum cost.



Oil Delivery

Oil needs to be delivered from refineries to gas stations.

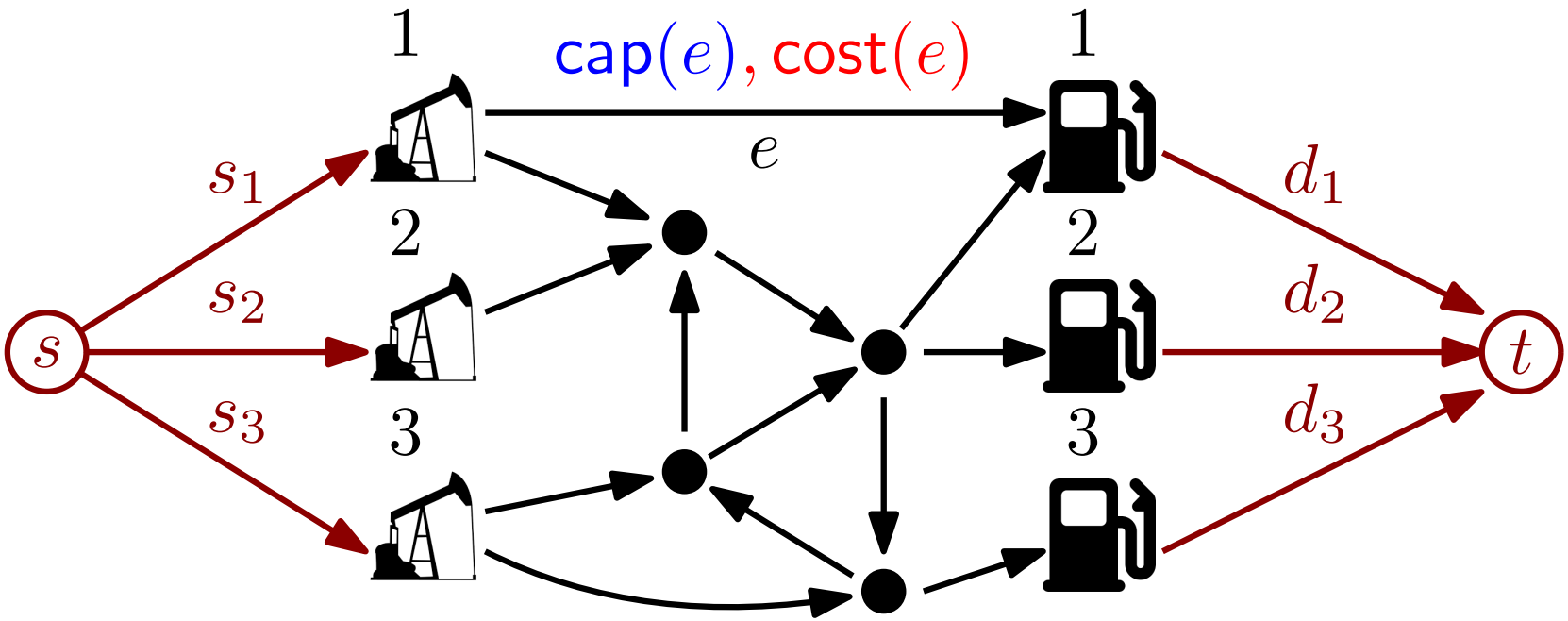
Refinery i produces s_i units of oil.

Gas station j needs d_j units of oil.

Using a truck to transport 1 unit oil across road e costs $\mathsf{cost}(e)$.

At most $\mathsf{cap}(e)$ trucks per day can traverse road e.

Goal: satisfy all demands with minimum cost.



Min-Cost Max-Flow in BGL

Costs are encoded as edge weights (`boost::edge_weight_t`).

Edge e has weight cost(e). The reverse edge has weight $-$cost(e)

Cycle Canceling

- Needs an initial maximum flow f to work.
- Time $O(C \cdot n \cdot m)$, where C is the cost difference between f and a MCMF.
- Can handle negative costs

Min-Cost Max-Flow in BGL

Costs are encoded as edge weights (`boost::edge_weight_t`).

Edge e has weight cost(e). The reverse edge has weight $-$cost(e)

Cycle Canceling

- Needs an initial maximum flow f to work.
- Time $O(C \cdot n \cdot m)$, where C is the cost difference between f and a MCMF.
- Can handle negative costs

Successive Shortest Paths

- Does not need an initial flow.
- Time $O(|f| \cdot (m + n \log n))$, where $|f|$ is the maximum flow.
- **Cannot** handle negative costs

Cycle Canceling: Example

```
int main()
{
    Graph G(6);
    EdgeAdder edge_adder(G);

    edge_adder.add_edge(0, 1, 1, 2);
    edge_adder.add_edge(0, 3, 2, 3);
    edge_adder.add_edge(1, 2, 3, 5);
    edge_adder.add_edge(1, 4, 4, 1);
    edge_adder.add_edge(2, 5, 2, 3);
    edge_adder.add_edge(3, 1, 1, 1);
    edge_adder.add_edge(3, 2, 2, 6);
    edge_adder.add_edge(4, 5, 2, 4);

    long flow = boost::push_relabel_max_flow(G, 0, 5);
    std::cout << "The␣maximum␣flow␣from␣0␣to␣5␣is␣" << flow << "\n";

    boost::cycle_canceling(G);
    std::cout << "The␣minimum␣cost␣of␣a␣max␣flow␣from␣0␣to␣15␣is␣"
              << boost::find_flow_cost(G) << "\n";

    return EXIT_SUCCESS;
}
```

Cycle Canceling: Example

```
int main()
{
    Graph G(6);
    EdgeAdder edge_adder(G);

    edge_adder.add_edge(0, 1, 1, 2);
    edge_adder.add_edge(0, 3, 2, 3);
    edge_adder.add_edge(1, 2, 3, 5);
    edge_adder.add_edge(1, 4, 4, 1);
    edge_adder.add_edge(2, 5, 2, 3);
    edge_adder.add_edge(3, 1, 1, 1);
    edge_adder.add_edge(3, 2, 2, 6);
    edge_adder.add_edge(4, 5, 2, 4);

    long flow = boost::push_relabel_max_flow(G, 0, 5);
    std::cout << "The maximum flow from 0 to 5 is " << flow << "\n";

    boost::cycle_canceling(G);
    std::cout << "The minimum cost of a max flow from 0 to 15 is "
              << boost::find_flow_cost(G) << "\n";

    return EXIT_SUCCESS;
}
```

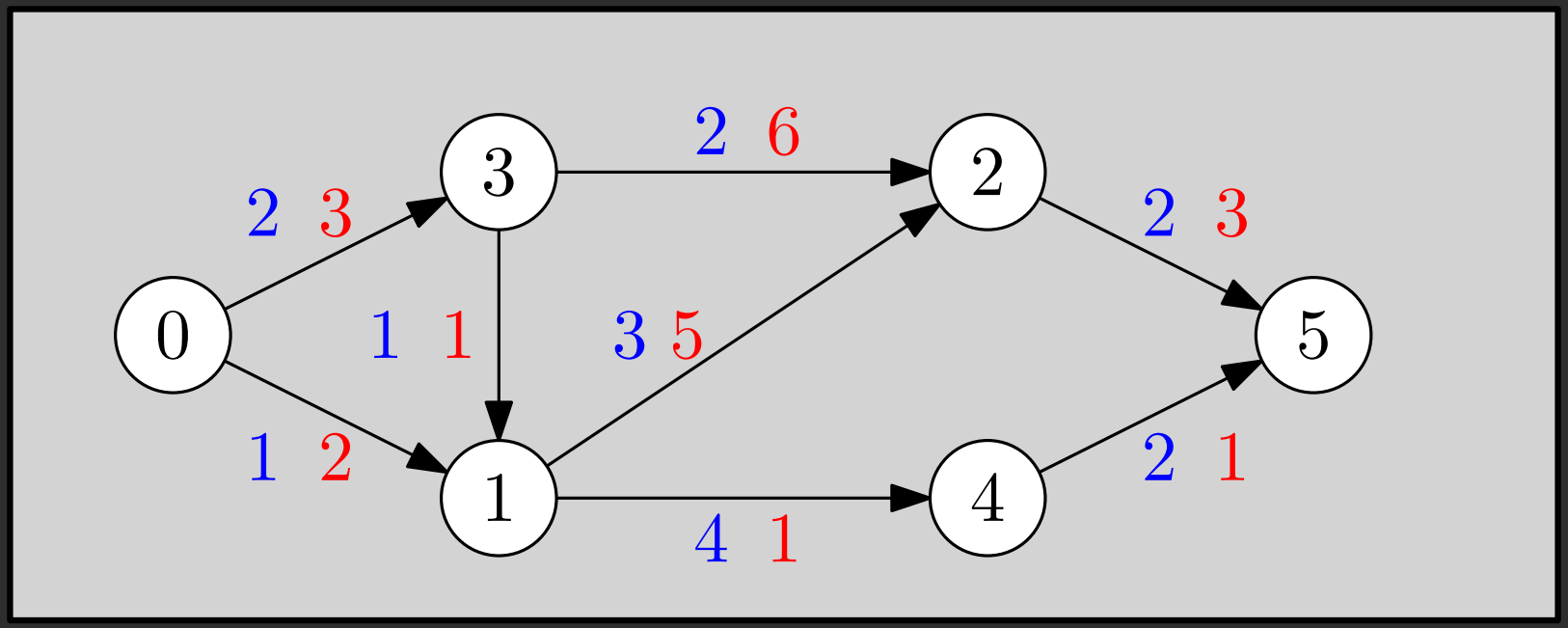
costs

Alternatively,
edmonds_karp_max_flow

returns the cost of the flow

Cycle Canceling: Example

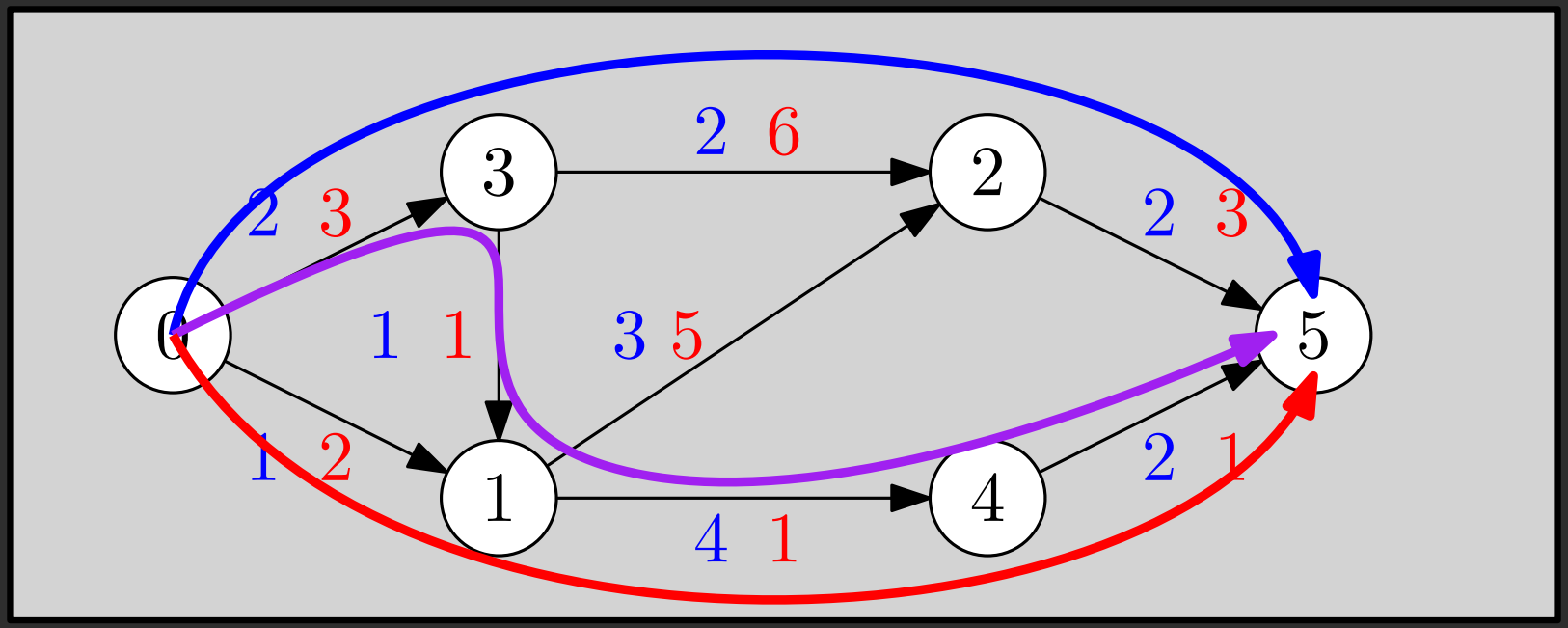
```
int main()
{
    $ g++ -std=c++17 mcmf_cc.cpp -o mcmf_cc
    $
    $ ./mcmf_cc
    The maximum flow from 0 to 5 is 3
    The minimum cost of a max flow from 0 to 5 is 22
    $
```




```
    return EXIT_SUCCESS;
}
```

Cycle Canceling: Example

```
int main()
{
    $ g++ -std=c++17 mcmf_cc.cpp -o mcmf_cc
    $
    $ ./mcmf_cc
    The maximum flow from 0 to 5 is 3
    The minimum cost of a max flow from 0 to 5 is 22
    $
```




```
    return EXIT_SUCCESS;
}
```

Successive Shortest Paths: Example

```
int main()
{
    [...]

    boost::successive_shortest_path_nonnegative_weights(G, 0, 5);

    std::cout << "The␣minimum␣cost␣of␣a␣max␣flow␣from␣0␣to␣5␣is␣"
              << boost::find_flow_cost(G) << "\n";

    return EXIT_SUCCESS;
}
```

Successive Shortest Paths: Example

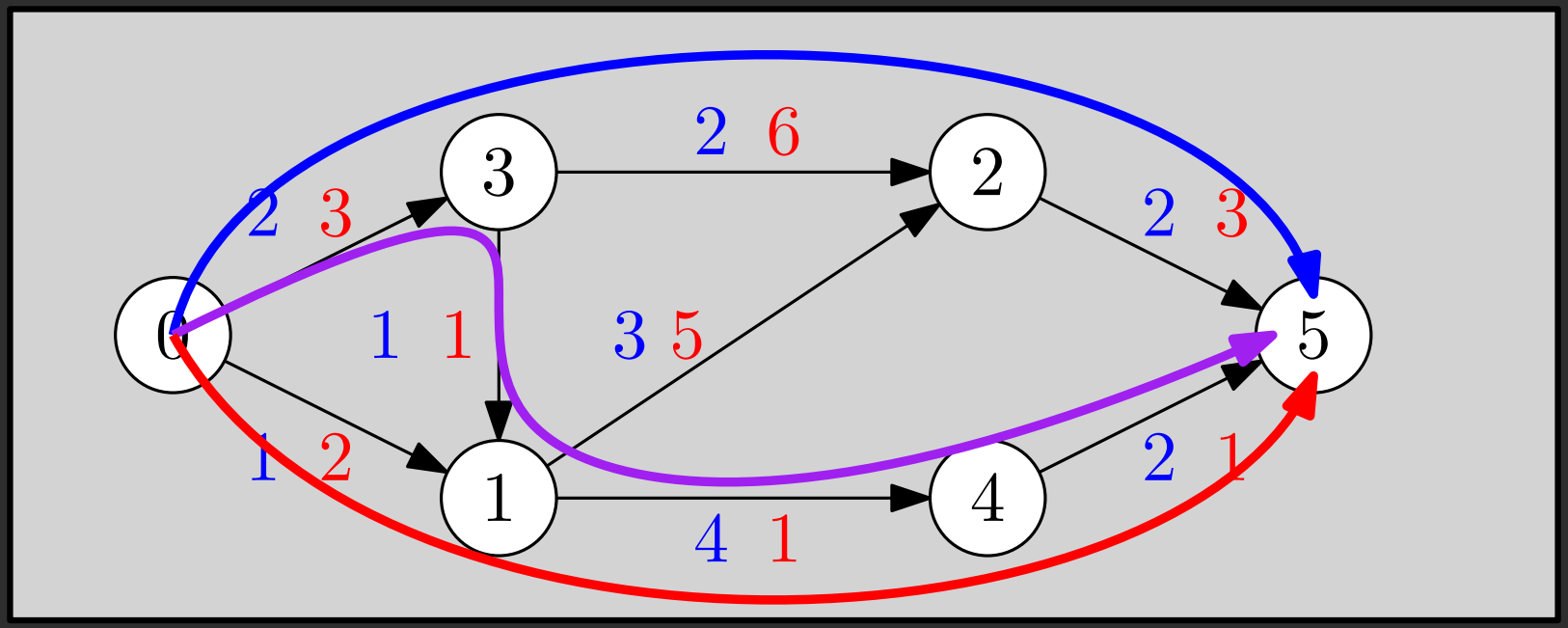
```
int main()
{
    [...]

    boost::successive_shortest_path_nonnegative_weights(G, 0, 5);

    std::cout << "The␣minimum␣cost␣of␣a␣max␣flow␣from␣0␣to␣5␣is␣"
              << boost::find_flow_cost(G) << "\n";

    //Compute flow value by summing over out-edges of the source vertex 0
    capacity_map capacity = boost::get(boost::edge_capacity, G);
    residual_map residual_capacity =
                              boost::get(boost::edge_residual_capacity, G);
```

Compute $|f| = \sum_{e=(s,v)} f(e)$

```
    long flow = 0;
    for(auto [eit, eend]= boost::out_edges(0, G); eit!=eend; eit++)
        flow += capacity[*eit] - residual_capacity[*eit];

    std::cout << "The␣maximum␣flow␣from␣0␣to␣5␣is␣" << flow << "\n";

    return EXIT_SUCCESS;
}
```

Successive Shortest Paths: Example

```
int main()
{
    $ g++ -std=c++17 mcmf_ssp.cpp -o mcmf_ssp
    $
    $ ./mcmf_ssp
    The minimum cost of a max flow from 0 to 5 is 22
    The maximum flow from 0 to 5 is 3
    $
```




```
    return EXIT_SUCCESS;
}
```