算法设计与分析(2025年春季学期)

Network Flow

授课老师: 栗师

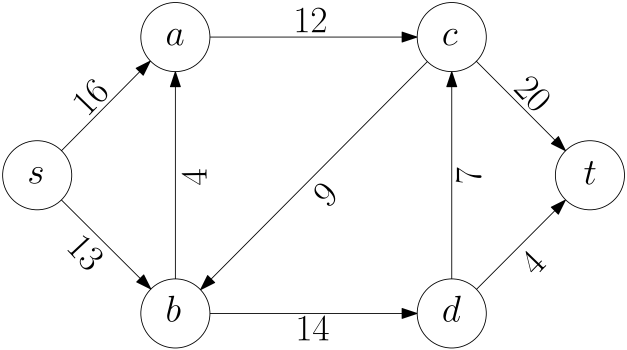
南京大学计算机学院

Outline

1. Network Flow
2. Ford-Fulkerson Method
3. Correctness of Ford-Fulkerson's Method and Maximum Flow Minimum Cut Theorem
4. Running Time of Ford-Fulkerson-Type Algorithm
   - Shortest Augmenting Path Algorithm
   - Capacity-Scaling Algorithm
5. Bipartite Matching Problem
6. s-t Edge-Disjoint Paths Problem
7. More Applications

Flow Network

- Abstraction of fluid flowing through edges
- Digraph $G = (V, E)$ with source $s \in V$ and sink $t \in V$
  - No edges enter s
  - No edges leave t
- Edge capacity $c_e \in \mathbb{R}_{>0}$ for every $e \in E$



Def. An s-t flow is a function $f : E \to \mathbb{R}$ such that

- for every $e \in E$: $0 \leq f(e) \leq c_e$ (capacity conditions)
- for every $v \in V \setminus \{s, t\}$:

$$\sum_{e \in \delta_{\text{in}}(v)} f(e) = \sum_{e \in \delta_{\text{out}}(v)} f(e). \qquad \text{(conservation conditions)}$$

The value of a flow f is

$$\text{val}(f) := \sum_{e \in \delta_{\text{out}}(s)} f(e).$$

Def. An s-t flow is a function $f : E \to \mathbb{R}$ such that

- for every $e \in E$: $0 \le f(e) \le c_e$ (capacity conditions)
- for every $v \in V \setminus \{s, t\}$:

$$\sum_{e \in \delta_{\text{in}}(v)} f(e) = \sum_{e \in \delta_{\text{out}}(v)} f(e). \qquad \text{(conservation conditions)}$$

The value of a flow f is

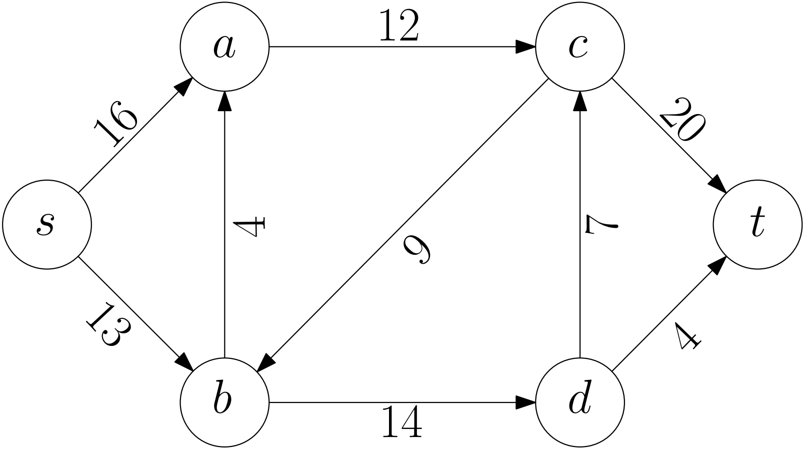
$$\text{val}(f) := \sum_{e \in \delta_{\text{out}}(s)} f(e).$$

Maximum Flow Problem

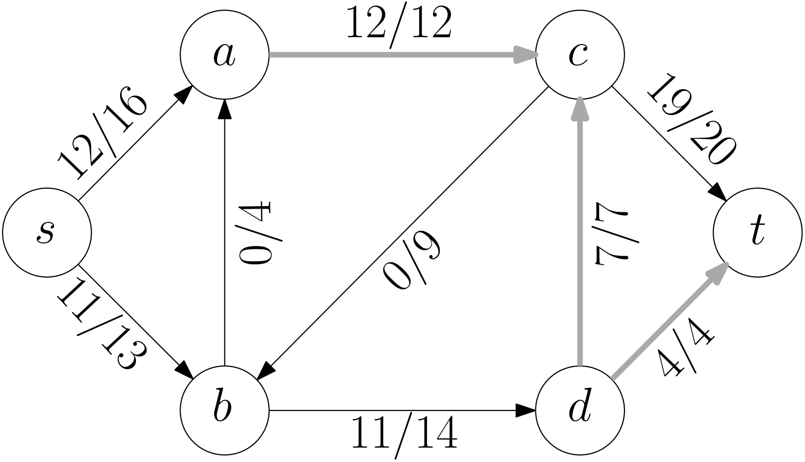
Input: directed network $G = (V, E)$, capacity function $c : E \to \mathbb{R}_{>0}$, source $s \in V$ and sink $t \in V$

Output: an s-t flow f in G with the maximum $\text{val}(f)$

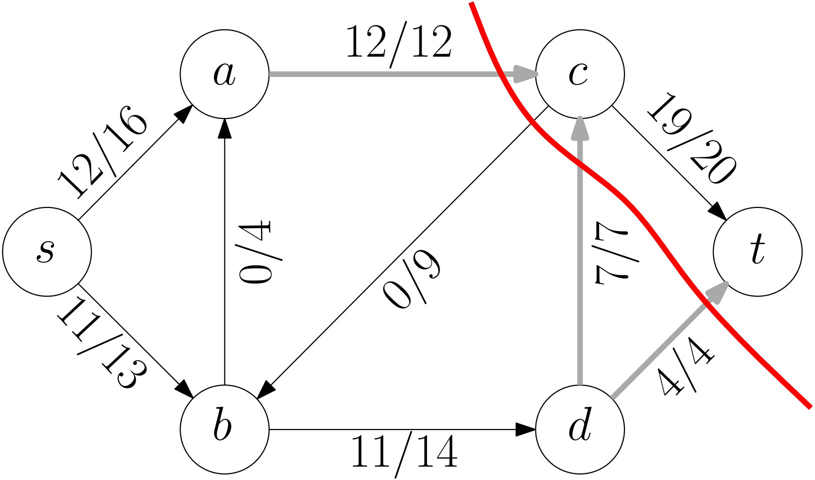
Maximum Flow Problem: Example



Maximum Flow Problem: Example



Maximum Flow Problem: Example



Outline

1. Network Flow
2. Ford-Fulkerson Method
3. Correctness of Ford-Fulkerson's Method and Maximum Flow Minimum Cut Theorem
4. Running Time of Ford-Fulkerson-Type Algorithm
   - Shortest Augmenting Path Algorithm
   - Capacity-Scaling Algorithm
5. Bipartite Matching Problem
6. s-t Edge-Disjoint Paths Problem
7. More Applications

Greedy Algorithm

Greedy Algorithm

- Start with empty flow: $f(e) = 0$ for every $e \in E$

Greedy Algorithm

- Start with empty flow: $f(e) = 0$ for every $e \in E$
- Define the residual capacity of e to be $c_e - f(e)$

Greedy Algorithm

- Start with empty flow: $f(e) = 0$ for every $e \in E$
- Define the residual capacity of e to be $c_e - f(e)$
- Find an augmenting path: a path from s to t, where all edges have positive residual capacity

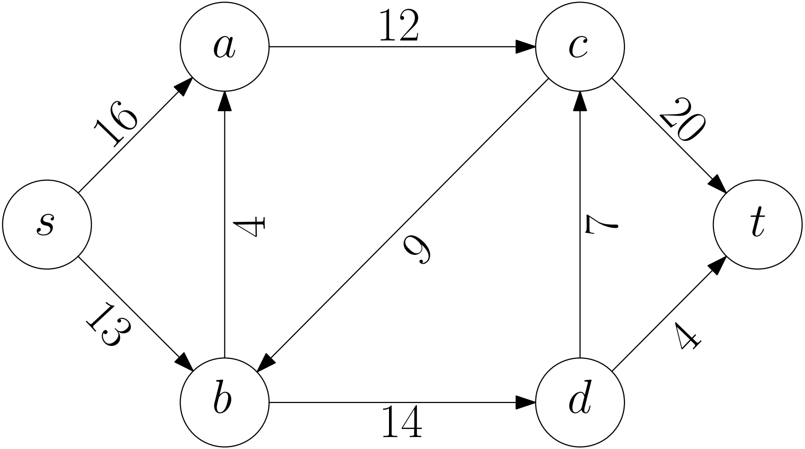
Greedy Algorithm

- Start with empty flow: $f(e) = 0$ for every $e \in E$
- Define the residual capacity of e to be $c_e - f(e)$
- Find an augmenting path: a path from s to t, where all edges have positive residual capacity
- Augment flow along the path as much as possible

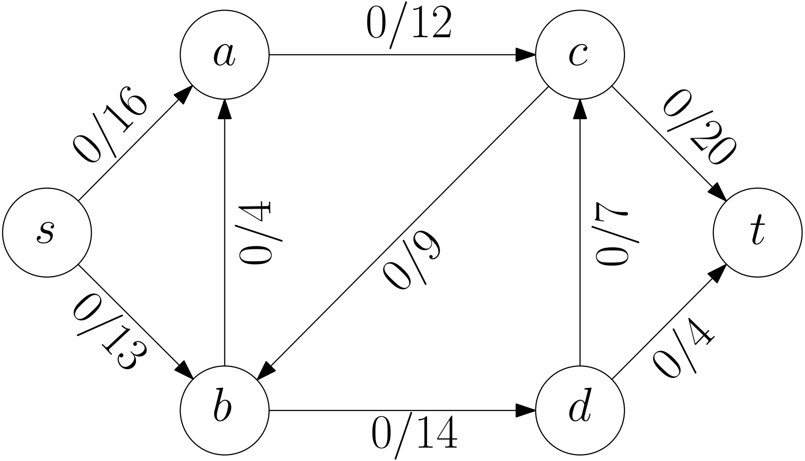
Greedy Algorithm

- Start with empty flow: $f(e) = 0$ for every $e \in E$
- Define the residual capacity of e to be $c_e - f(e)$
- Find an augmenting path: a path from s to t, where all edges have positive residual capacity
- Augment flow along the path as much as possible
- Repeat until we got stuck

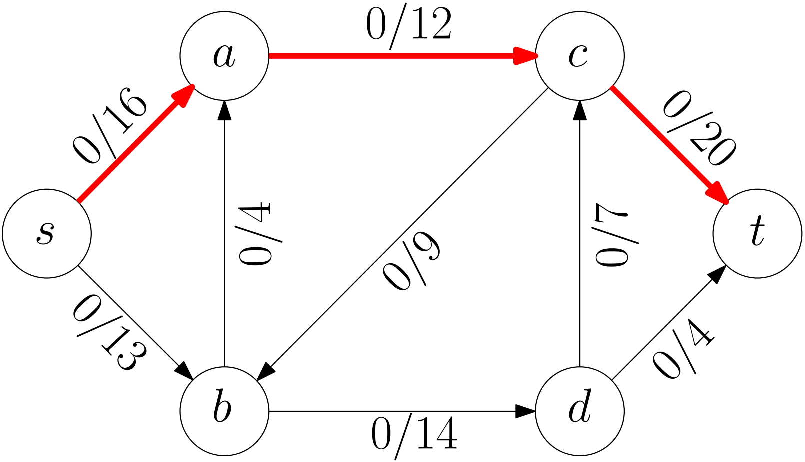
Greedy Algorithm: Example



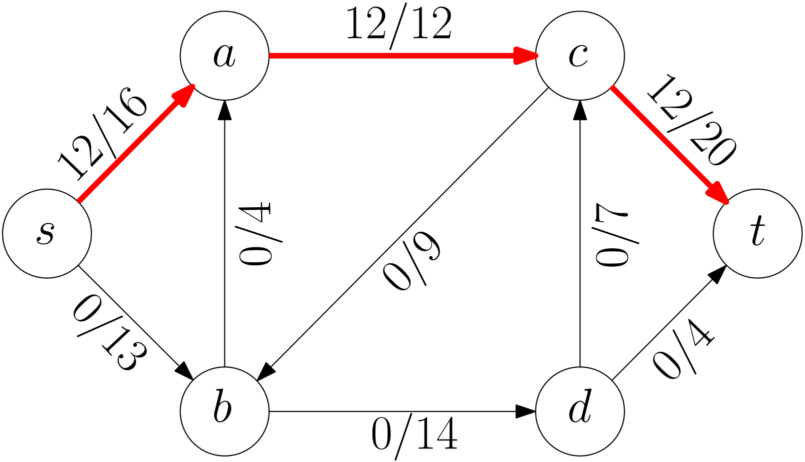
Greedy Algorithm: Example



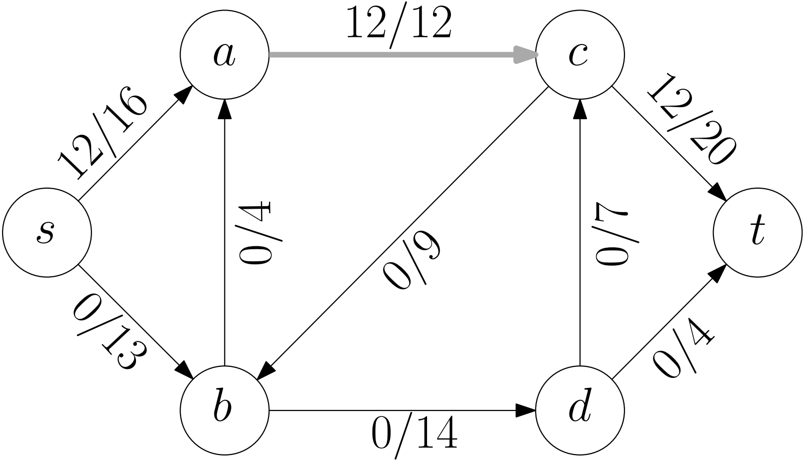
Greedy Algorithm: Example



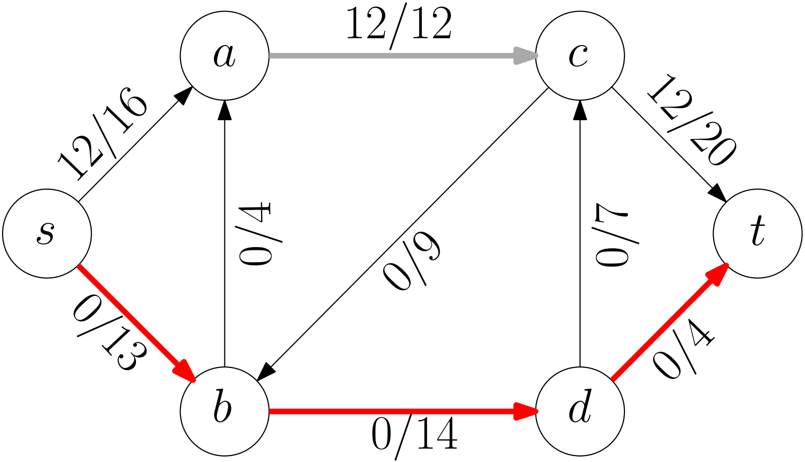
Greedy Algorithm: Example



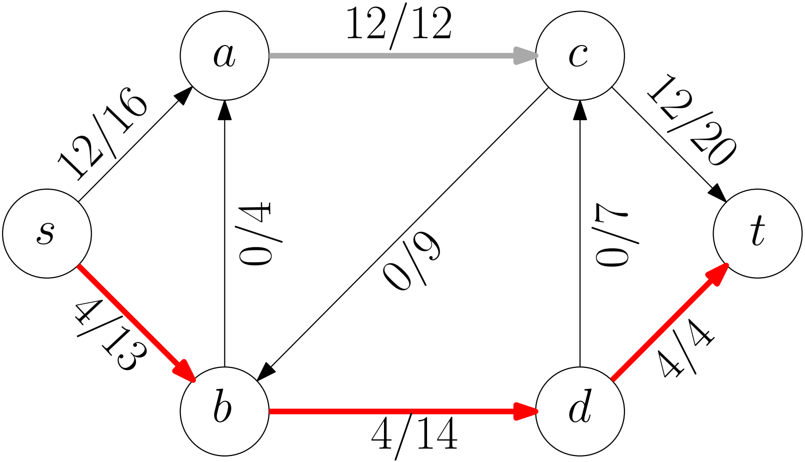
Greedy Algorithm: Example



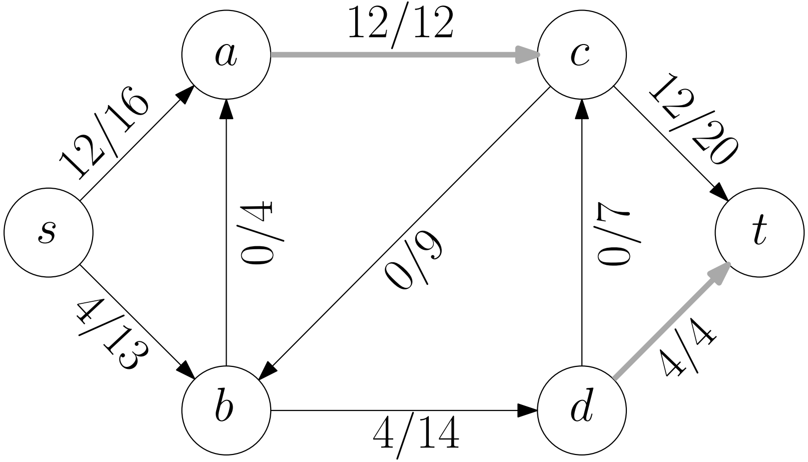
Greedy Algorithm: Example



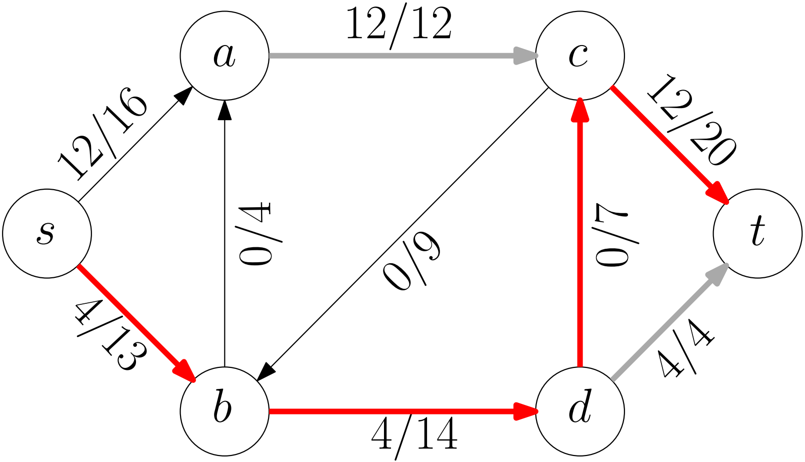
Greedy Algorithm: Example



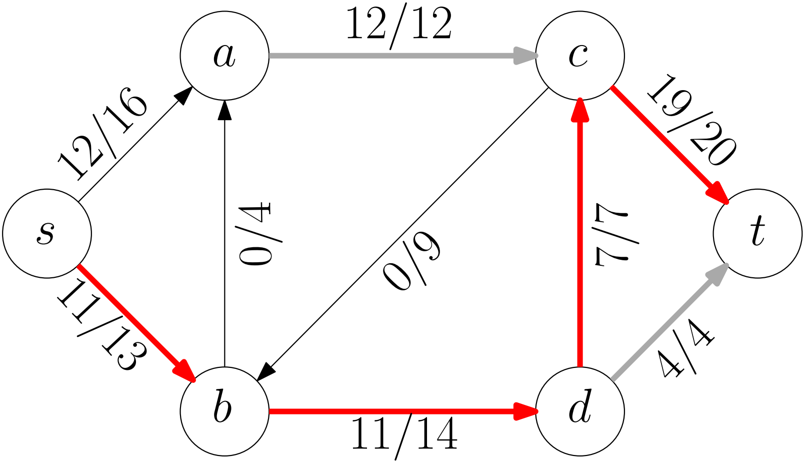
Greedy Algorithm: Example



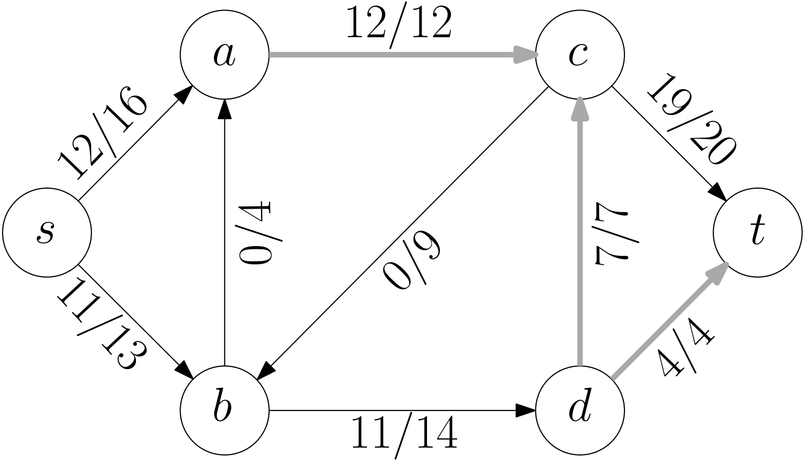
Greedy Algorithm: Example



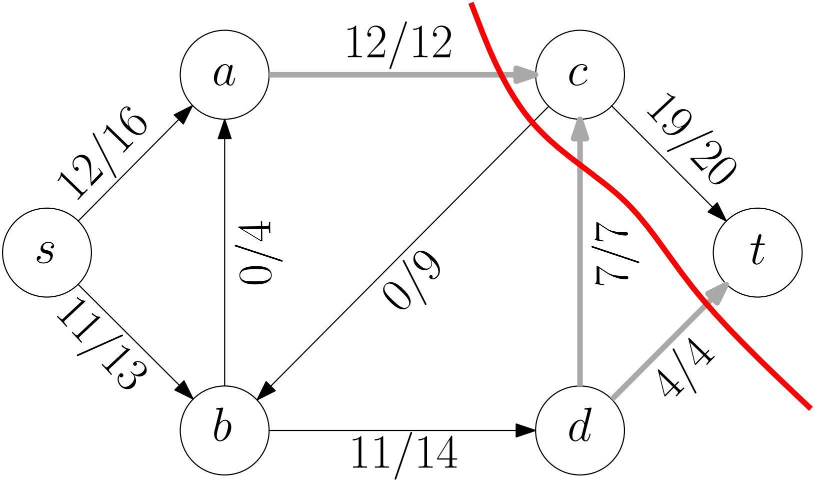
Greedy Algorithm: Example



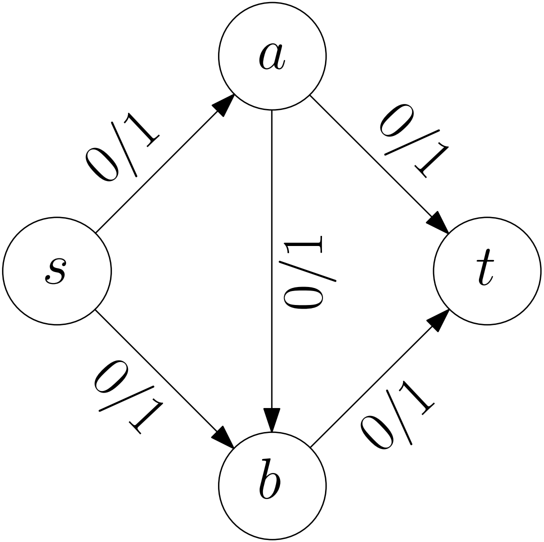
Greedy Algorithm: Example



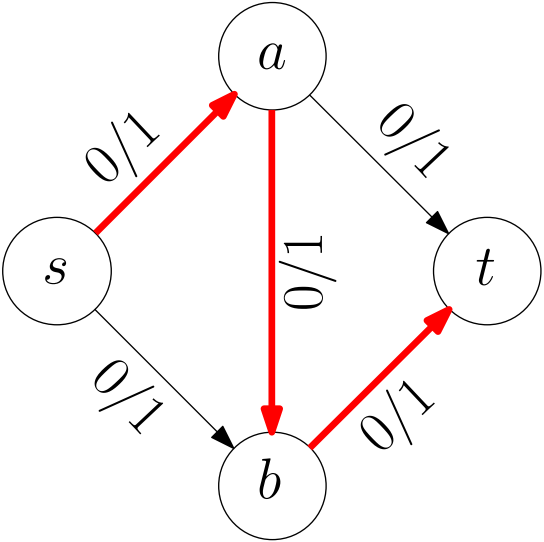
Greedy Algorithm: Example



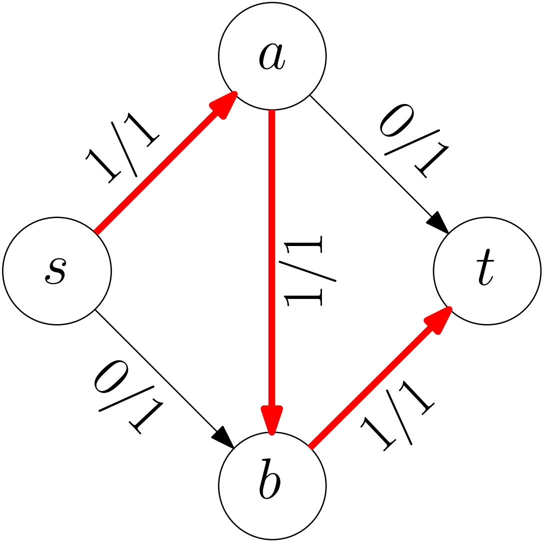
Greedy Algorithm Does Not Always Give a Optimum Solution



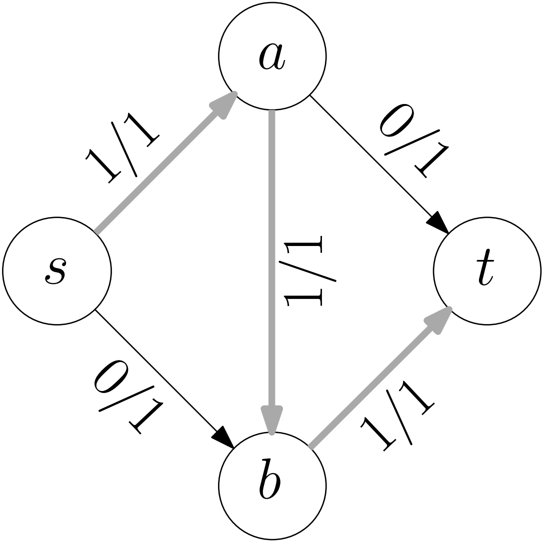
Greedy Algorithm Does Not Always Give a Optimum Solution



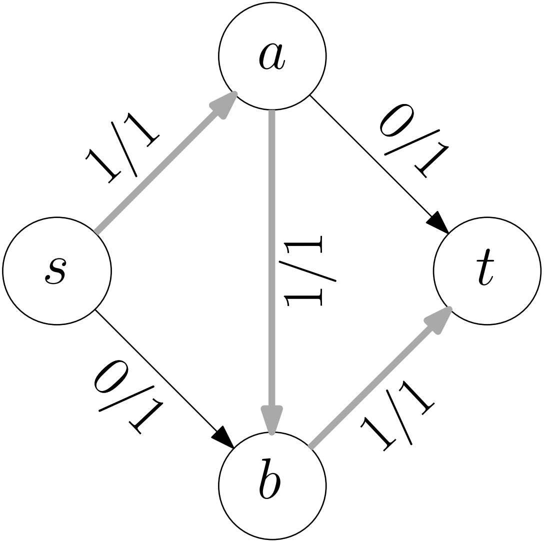
Greedy Algorithm Does Not Always Give a Optimum Solution



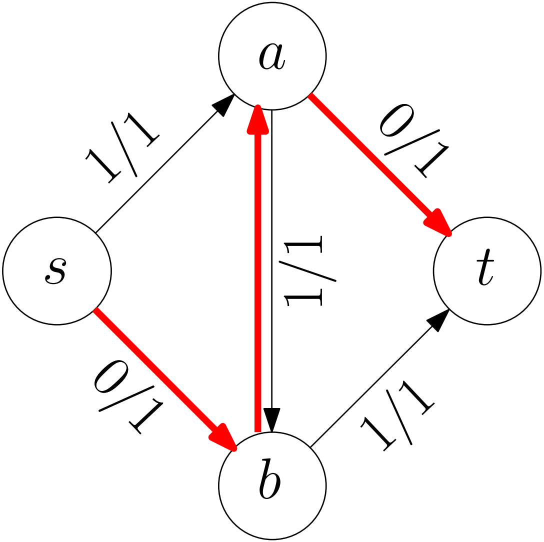
Greedy Algorithm Does Not Always Give a Optimum Solution



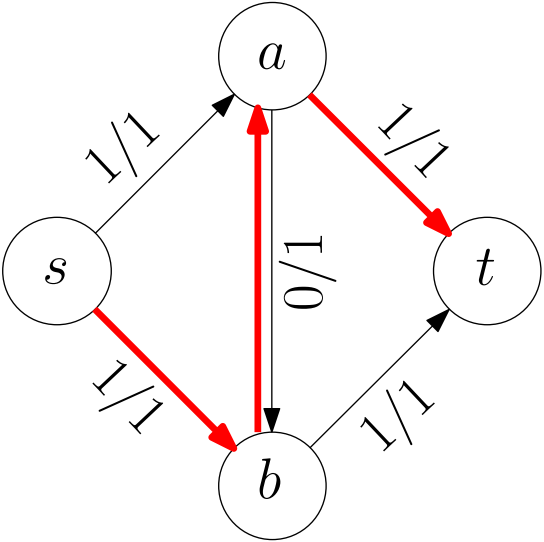
Fix the Issue: Allowing "Undo" Flow Sent



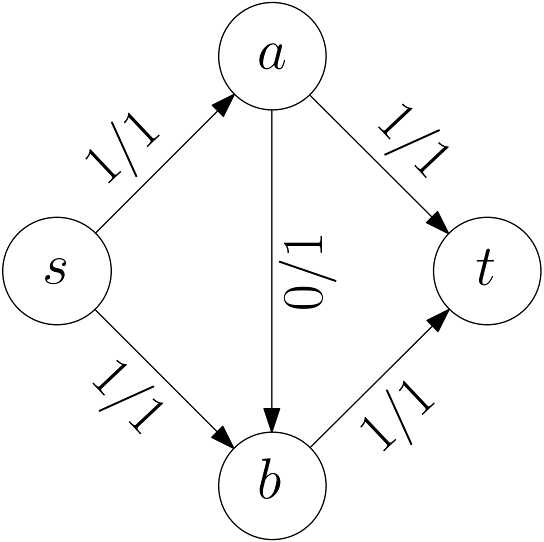
Fix the Issue: Allowing "Undo" Flow Sent



Fix the Issue: Allowing “Undo” Flow Sent



Fix the Issue: Allowing "Undo" Flow Sent



Assumption (u, v) and (v, u) are not both in E

Assumption (u, v) and (v, u) are not both in E

Def. For a s-t flow f, the residual graph G_f of $G = (V, E)$ w.r.t f contains:

Assumption (u, v) and (v, u) are not both in E

Def. For a s-t flow f, the residual graph G_f of $G = (V, E)$ w.r.t f contains:

- the vertex set V,

Assumption (u, v) and (v, u) are not both in E

Def. For a s-t flow f, the residual graph G_f of $G = (V, E)$ w.r.t f contains:

- the vertex set V,
- for every $e = (u, v) \in E$ with $f(e) < c_e$, a forward edge $e = (u, v)$, with residual capacity $c_f(e) = c_e - f(e)$,

Assumption (u, v) and (v, u) are not both in E

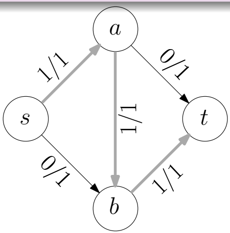
Def. For a s-t flow f, the residual graph G_f of $G = (V, E)$ w.r.t f contains:

- the vertex set V,
- for every $e = (u, v) \in E$ with $f(e) < c_e$, a forward edge $e = (u, v)$, with residual capacity $c_f(e) = c_e - f(e)$,
- for every $e = (u, v) \in E$ with $f(e) > 0$, a backward edge $e' = (v, u)$, with residual capacity $c_f(e') = f(e)$.

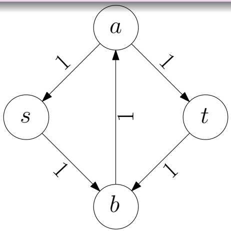
Assumption (u, v) and (v, u) are not both in E

Def. For a s-t flow f, the residual graph G_f of $G = (V, E)$ w.r.t f contains:

- the vertex set V,
- for every $e = (u, v) \in E$ with $f(e) < c_e$, a forward edge $e = (u, v)$, with residual capacity $c_f(e) = c_e - f(e)$,
- for every $e = (u, v) \in E$ with $f(e) > 0$, a backward edge $e' = (v, u)$, with residual capacity $c_f(e') = f(e)$.

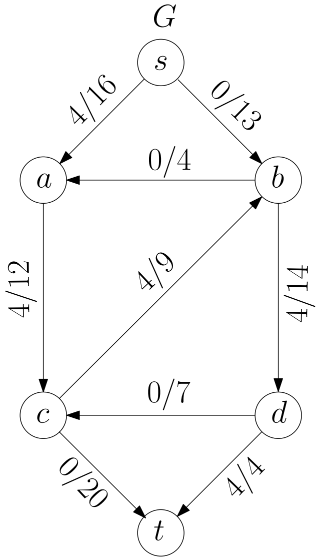


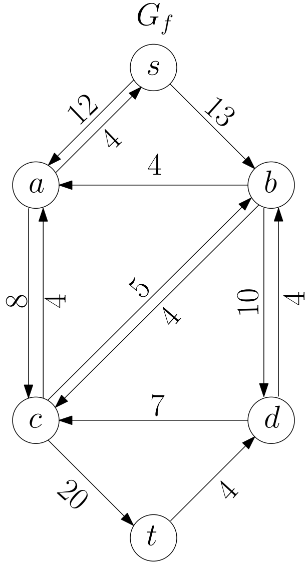
Original graph G and f



Residual Graph G_f

Residual Graph: One More Example





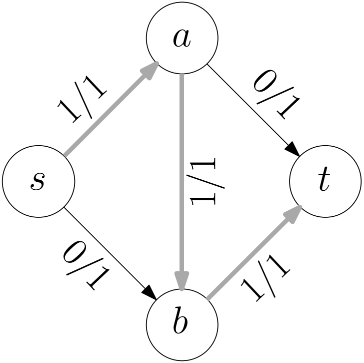
Agumenting Path

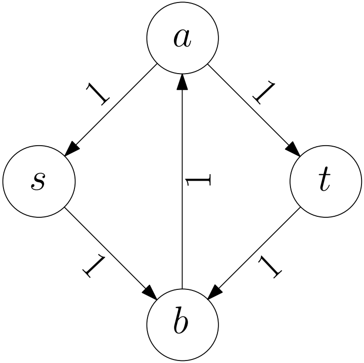
Augmenting the flow along a path P from s to t in G_f

Augment(P)

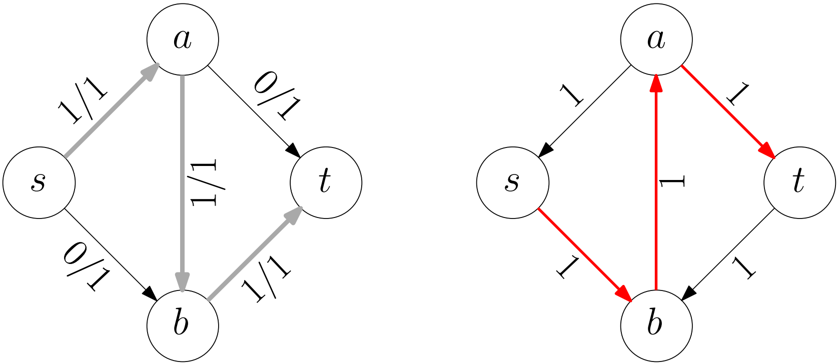
```
b ← $\min_{e \in P} c_f(e)$
for every $(u,v) \in P$ do
    if $(u,v)$ is a forward edge then
        $f(u,v) \leftarrow f(u,v) + b$
    else                                        ▷ $(u,v)$ is a backward edge
        $f(v,u) \leftarrow f(v,u) - b$
return $f$
```

Example for Augmenting Along a Path

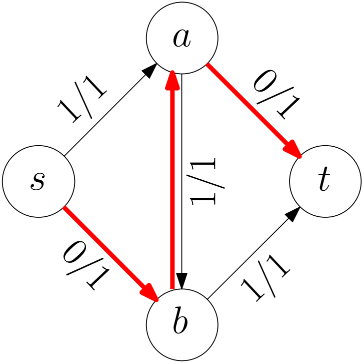


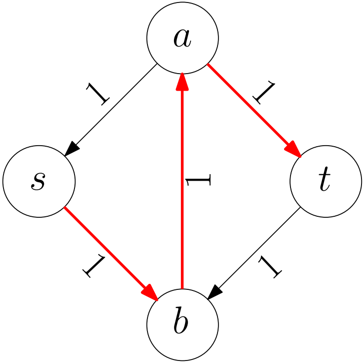


Example for Augmenting Along a Path

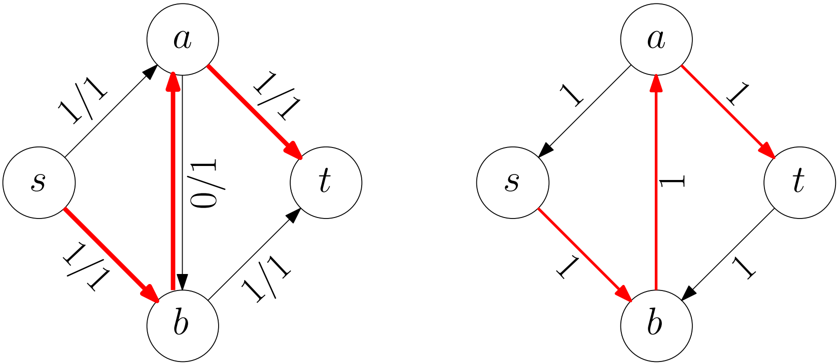


Example for Augmenting Along a Path

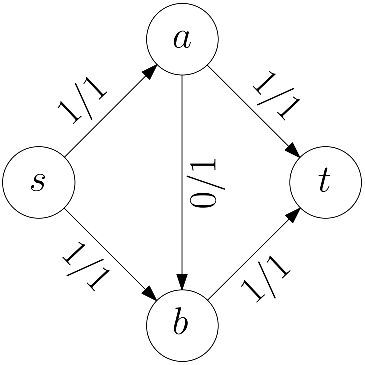


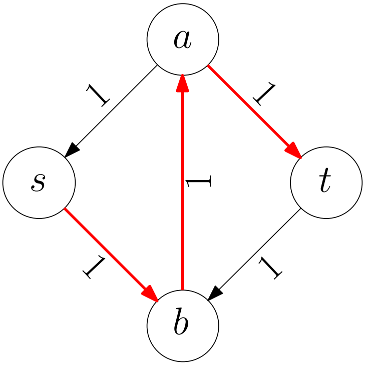


Example for Augmenting Along a Path



Example for Augmenting Along a Path



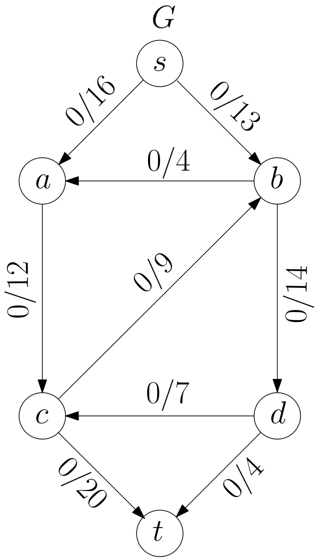


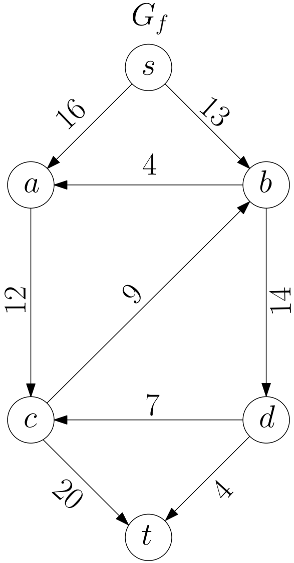
Ford-Fulkerson's Method

Ford-Fulkerson(G, s, t, c)

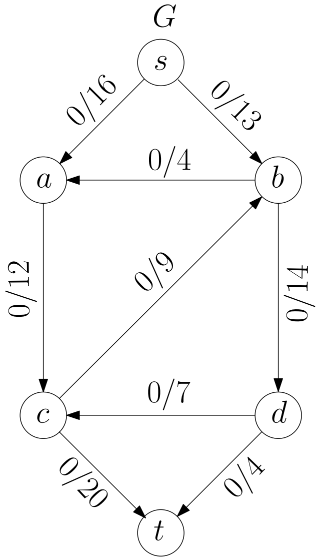
```
let $f(e) \leftarrow 0$ for every $e$ in $G$
while there is a path from $s$ to $t$ in $G_f$ do
    let $P$ be any simple path from $s$ to $t$ in $G_f$
    $f \leftarrow$augment$(f, P)$
return $f$
```

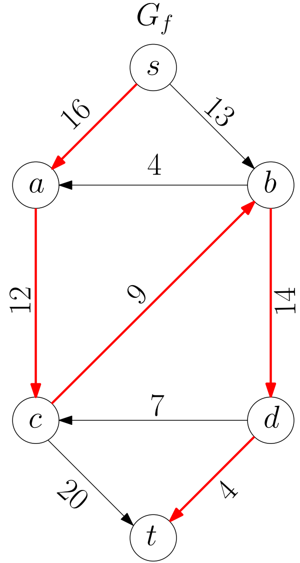
Ford-Fulkerson: Example



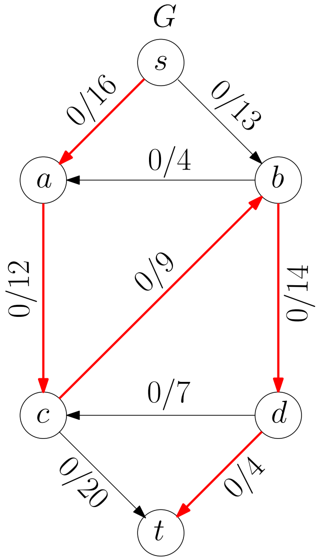


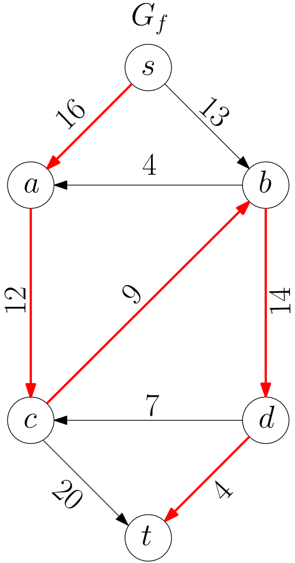
Ford-Fulkerson: Example



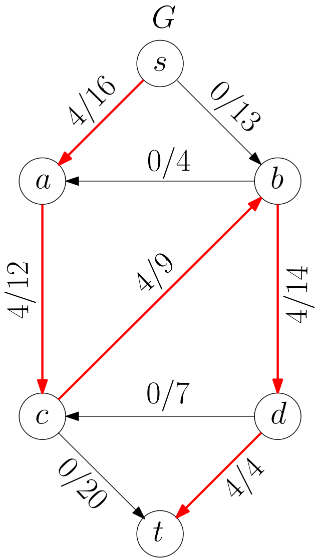


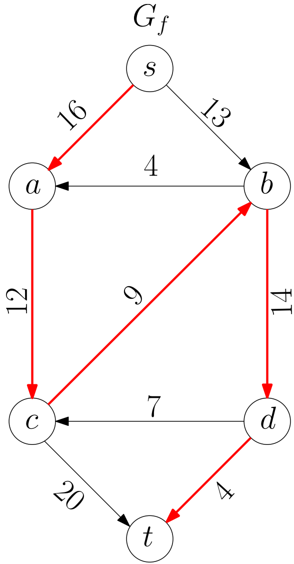
Ford-Fulkerson: Example



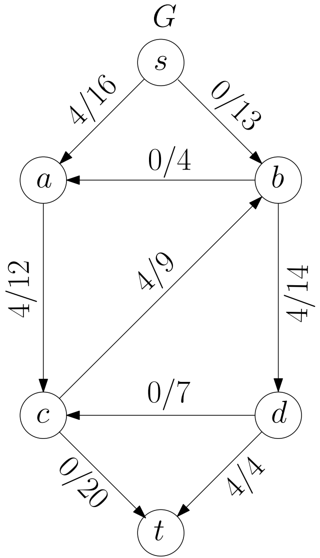


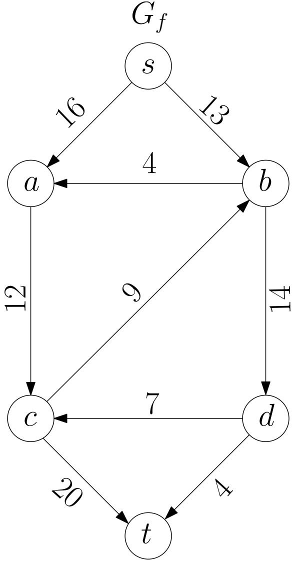
Ford-Fulkerson: Example



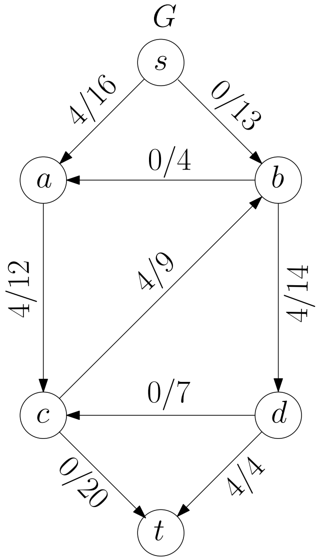


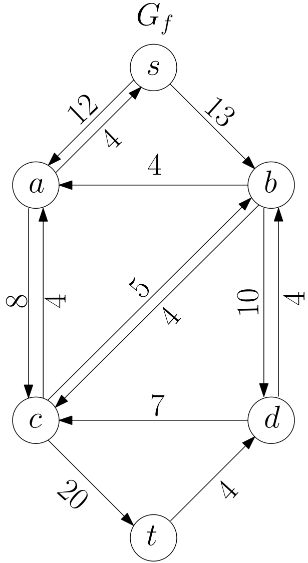
Ford-Fulkerson: Example



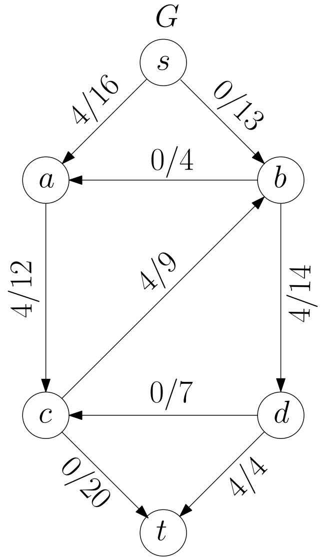


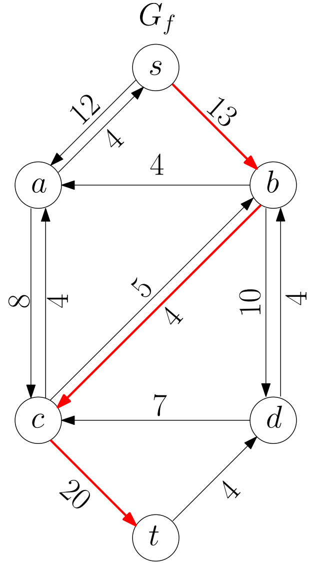
Ford-Fulkerson: Example



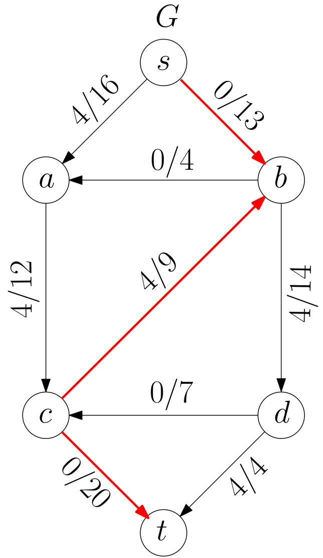


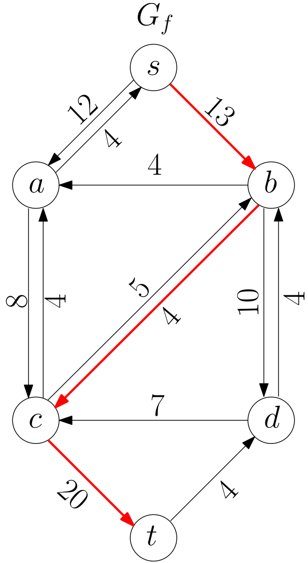
Ford-Fulkerson: Example



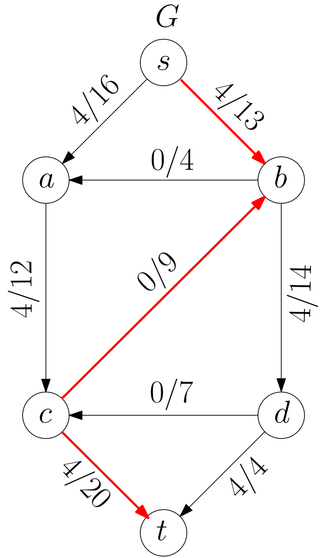


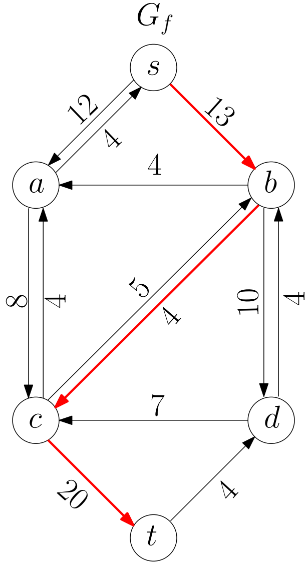
Ford-Fulkerson: Example



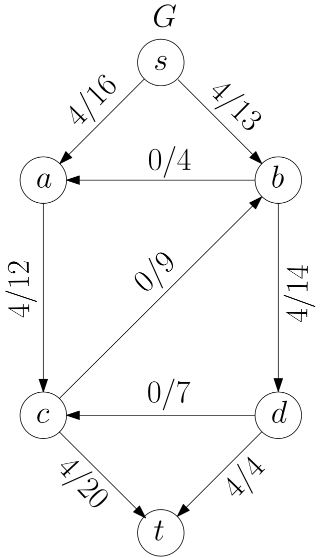


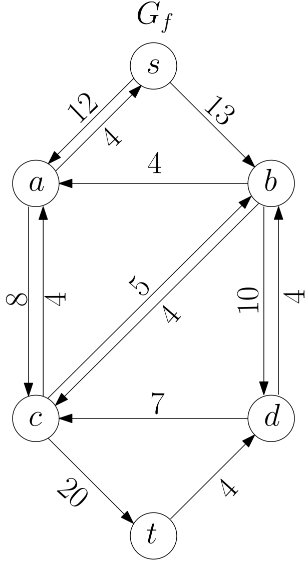
Ford-Fulkerson: Example



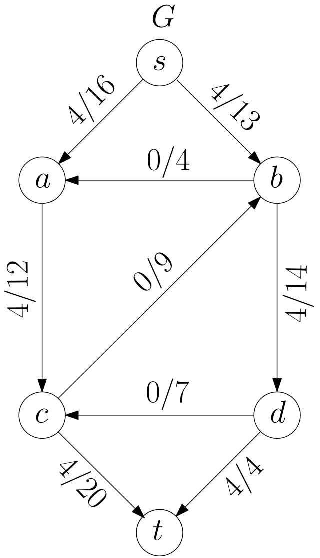


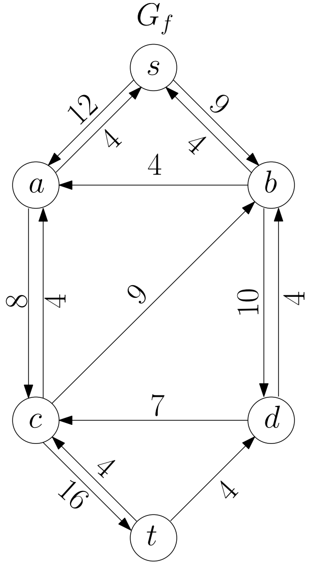
Ford-Fulkerson: Example



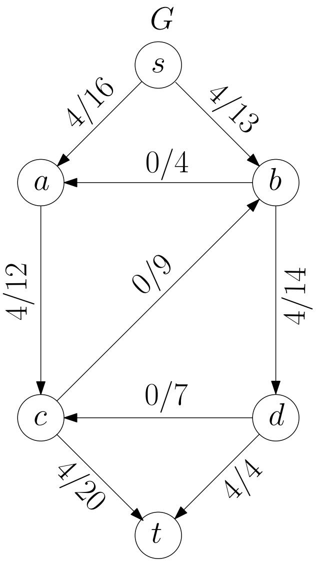


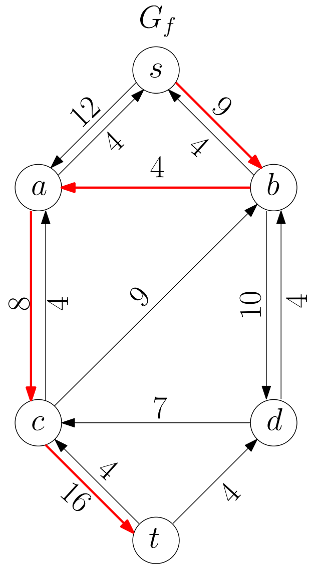
Ford-Fulkerson: Example



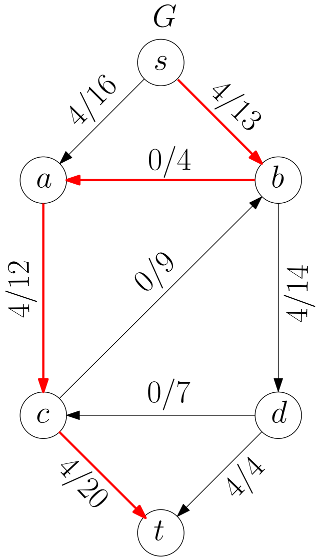


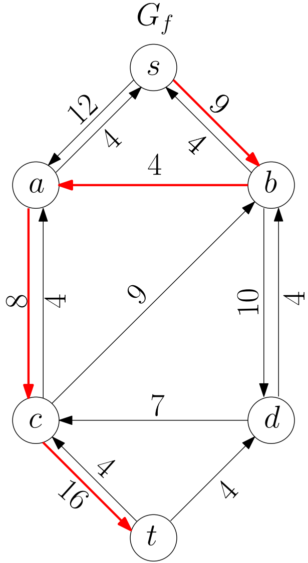
Ford-Fulkerson: Example



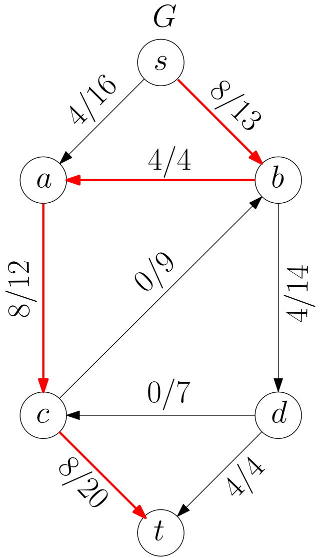


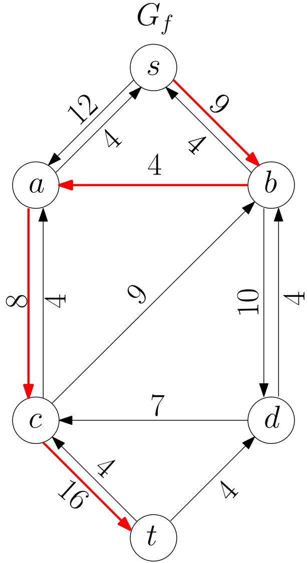
Ford-Fulkerson: Example



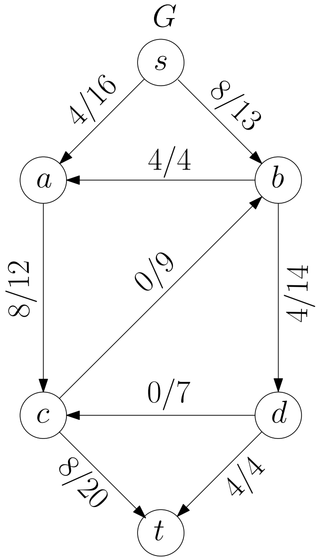


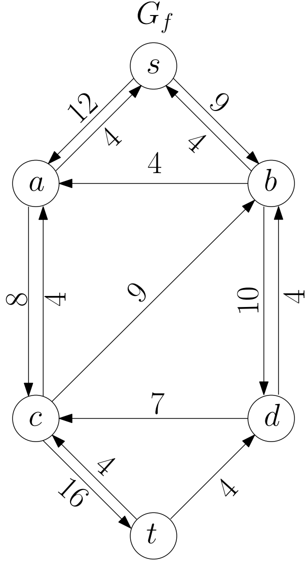
Ford-Fulkerson: Example



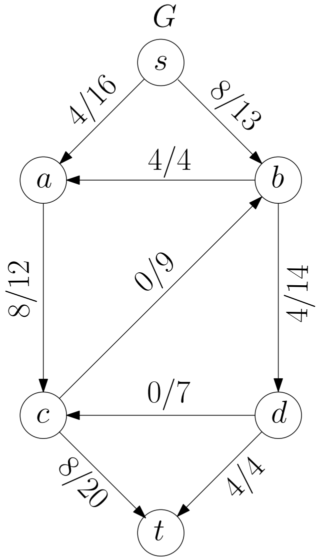


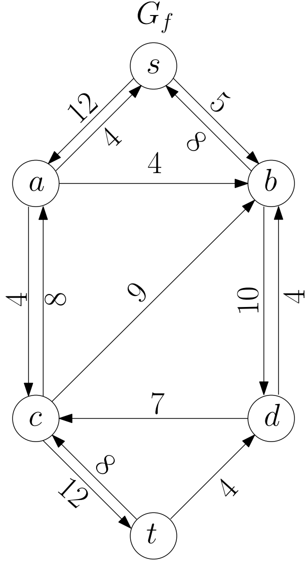
Ford-Fulkerson: Example



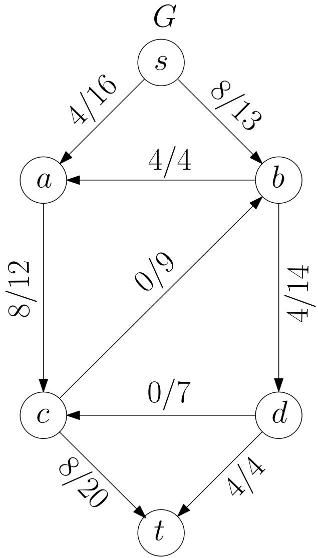


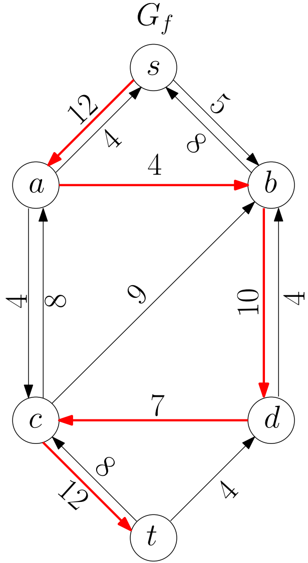
Ford-Fulkerson: Example



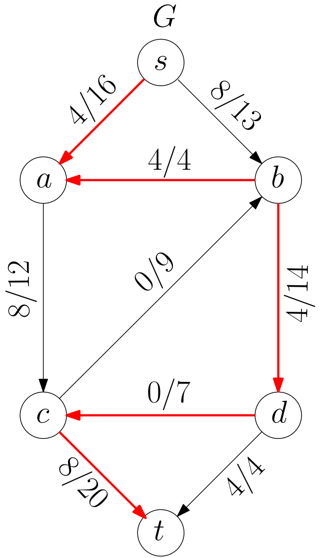


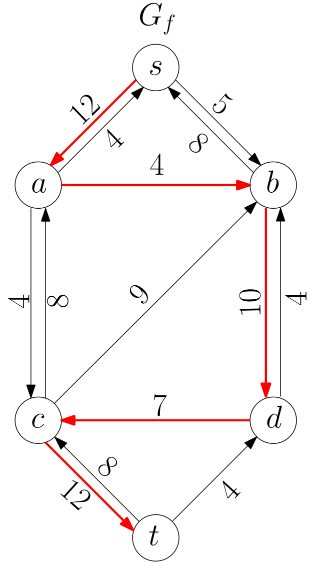
Ford-Fulkerson: Example



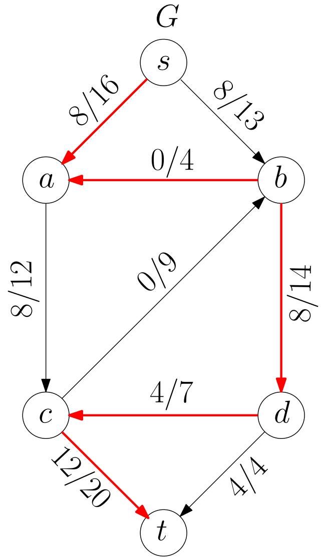


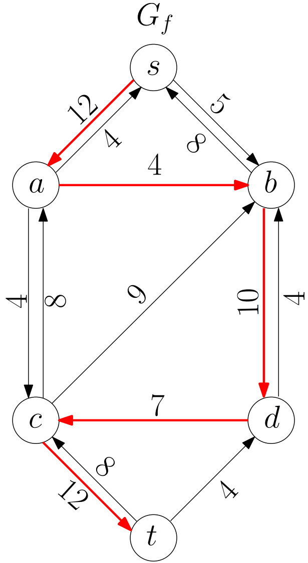
Ford-Fulkerson: Example



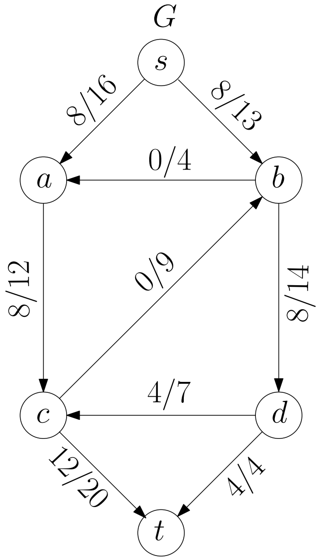


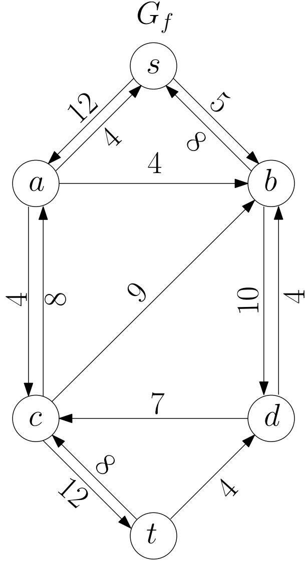
Ford-Fulkerson: Example



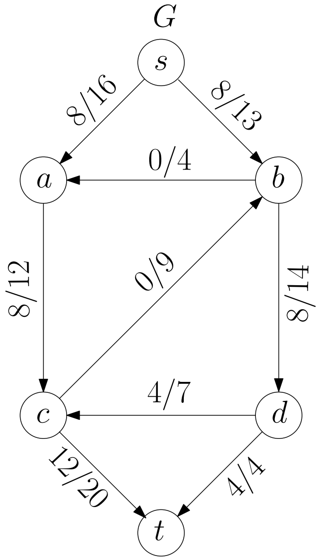


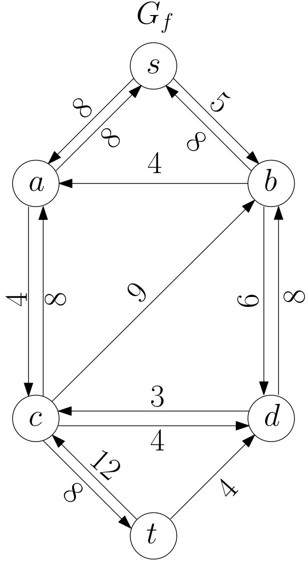
Ford-Fulkerson: Example



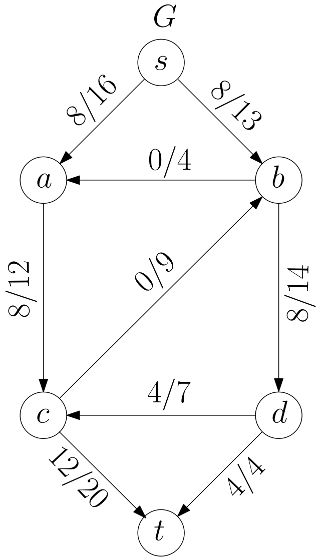


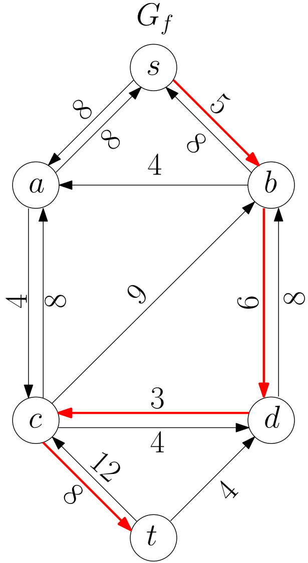
Ford-Fulkerson: Example



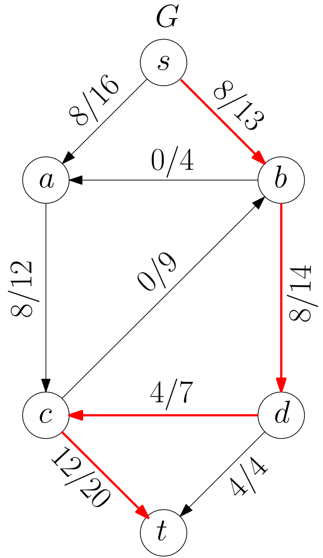


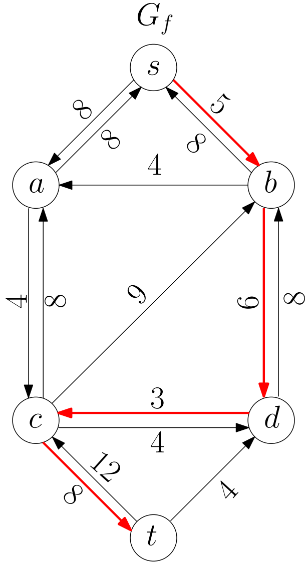
Ford-Fulkerson: Example



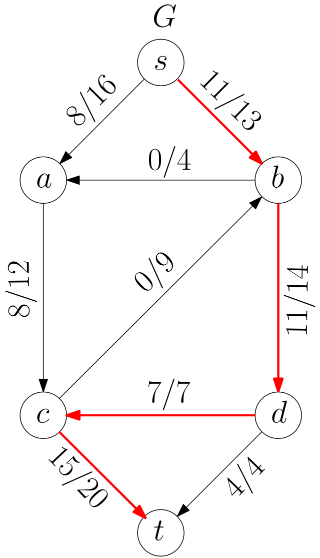


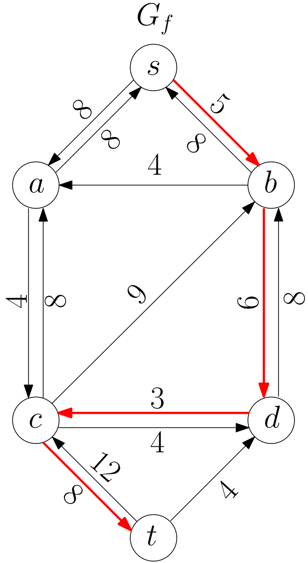
Ford-Fulkerson: Example



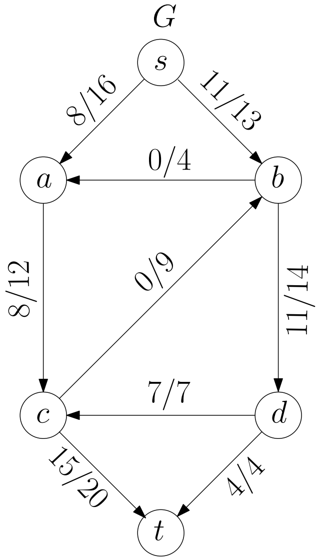


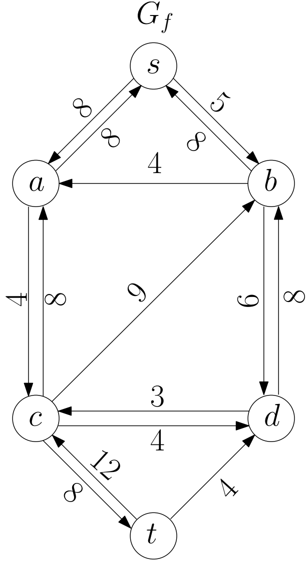
Ford-Fulkerson: Example



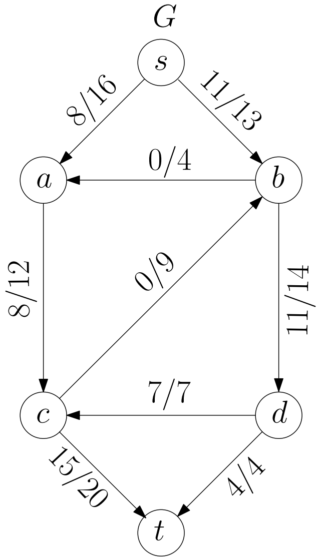


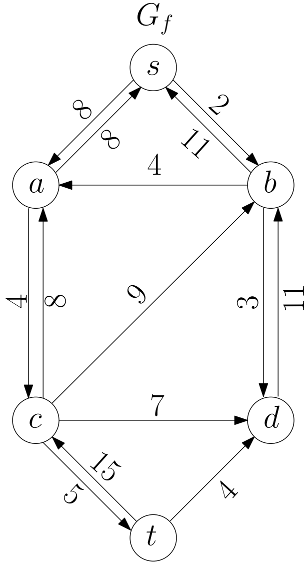
Ford-Fulkerson: Example



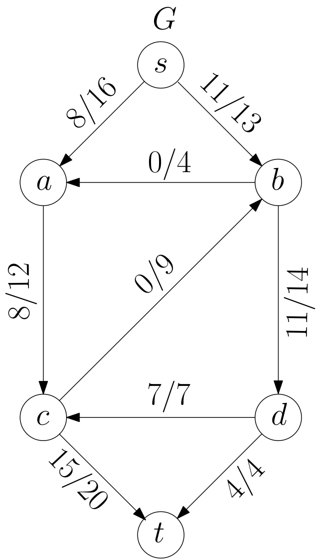


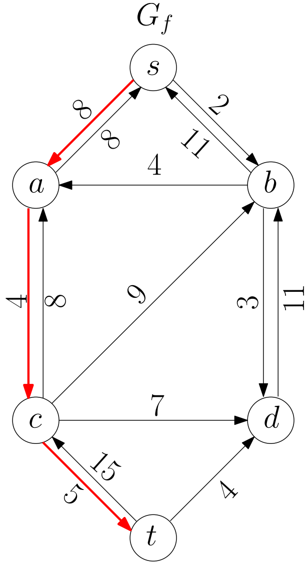
Ford-Fulkerson: Example



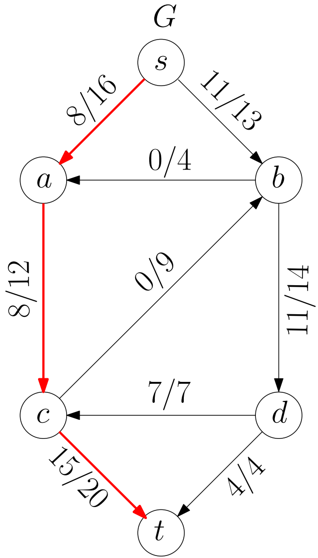


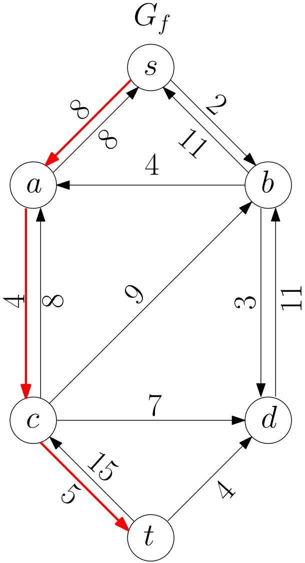
Ford-Fulkerson: Example



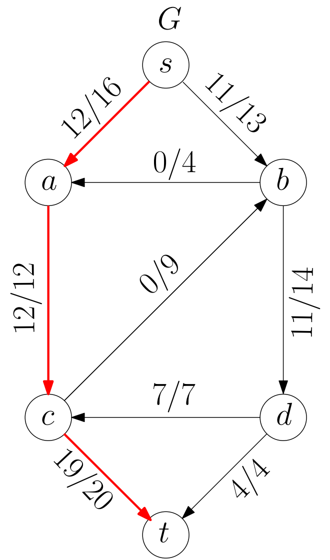


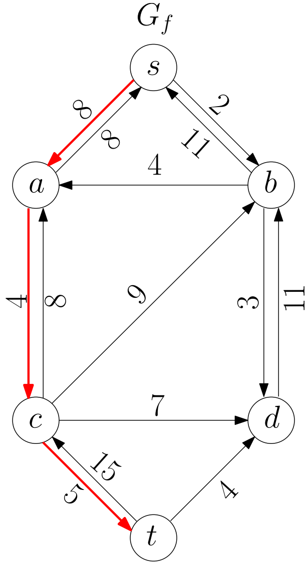
Ford-Fulkerson: Example



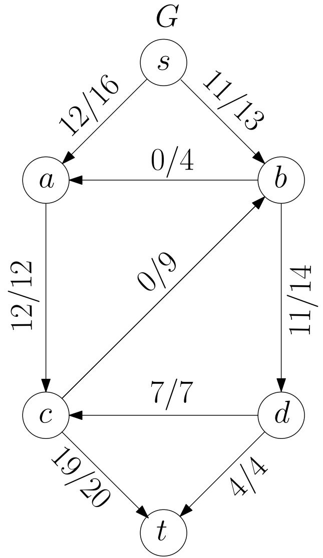


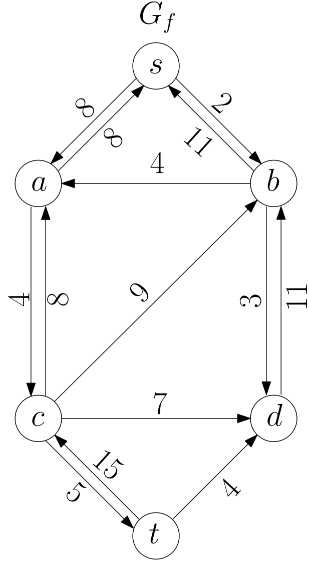
Ford-Fulkerson: Example



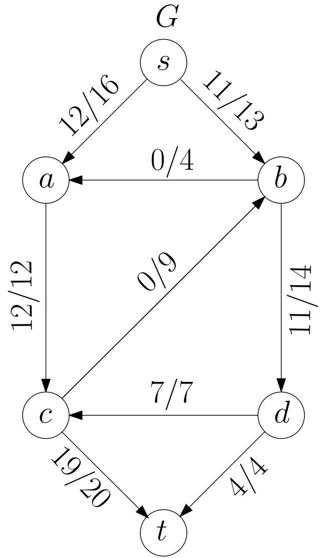


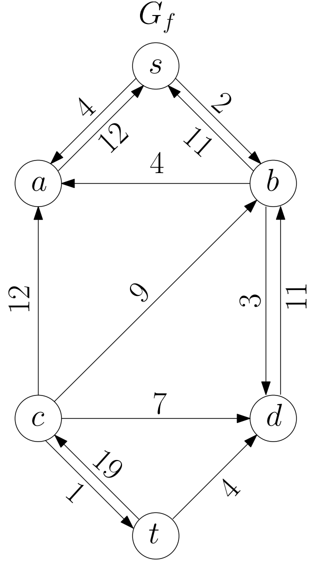
Ford-Fulkerson: Example



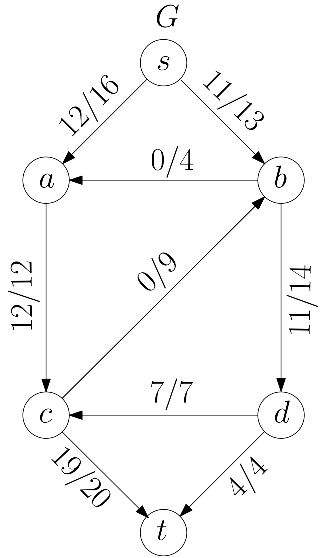


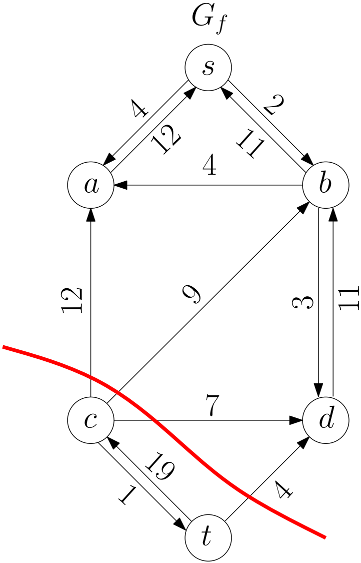
Ford-Fulkerson: Example



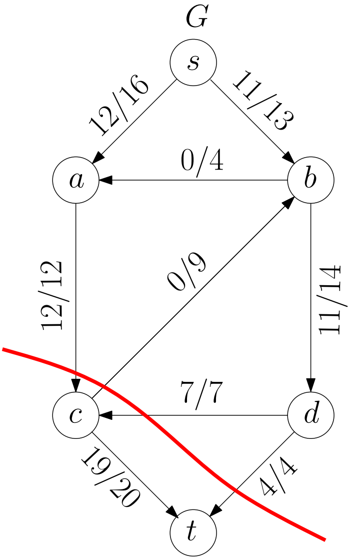


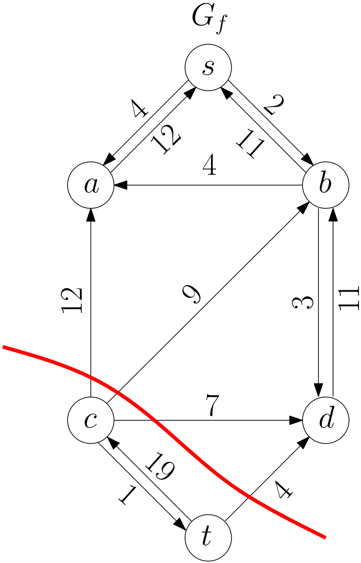
Ford-Fulkerson: Example





Ford-Fulkerson: Example





Outline

1. Network Flow
2. Ford-Fulkerson Method
3. Correctness of Ford-Fulkerson's Method and Maximum Flow Minimum Cut Theorem
4. Running Time of Ford-Fulkerson-Type Algorithm
   - Shortest Augmenting Path Algorithm
   - Capacity-Scaling Algorithm
5. Bipartite Matching Problem
6. s-t Edge-Disjoint Paths Problem
7. More Applications

Correctness of Ford-Fulkerson's Method

1. The procedure augment(f, P) maintains the two conditions:
   - for every $e \in E$: $0 \leq f(e) \leq c_e$ (capacity conditions)
   - for every $v \in V \setminus \{s, t\}$:

$$\sum_{e \in \delta_{\text{in}}(v)} f(e) = \sum_{e \in \delta_{\text{out}}(v)} f(e). \qquad \text{(conservation conditions)}$$

2. When Ford-Fulkerson's Method terminates, val(f) is maximized
3. Ford-Fulkerson's Method will terminate

Correctness of Ford-Fulkerson's Method

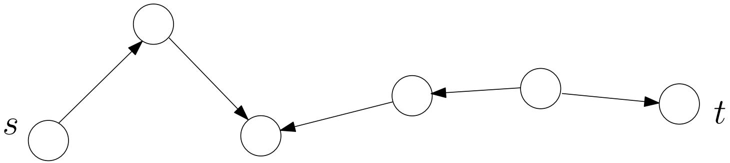
1. The procedure augment(f, P) maintains the two conditions:
   - for every $e \in E$: $0 \le f(e) \le c_e$ (capacity conditions)
   - for every $v \in V \setminus \{s, t\}$:

$$\sum_{e \in \delta_{\mathsf{in}}(v)} f(e) = \sum_{e \in \delta_{\mathsf{out}}(v)} f(e). \qquad \text{(conservation conditions)}$$

2. When Ford-Fulkerson's Method terminates, val(f) is maximized
3. Ford-Fulkerson's Method will terminate

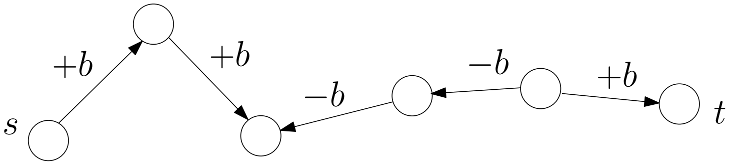
- for every $e \in E$: $0 \leq f(e) \leq c_e$ (capacity conditions)
- for every $v \in V \setminus \{s, t\}$:

$$\sum_{e \text{ into } v} f(e) = \sum_{e \text{ out of } v} f(e). \qquad \text{(conservation conditions)}$$



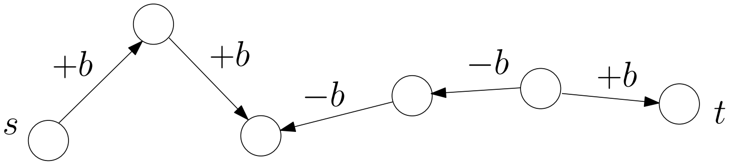
- for every $e \in E$: $0 \leq f(e) \leq c_e$ (capacity conditions)
- for every $v \in V \setminus \{s,t\}$:

$$\sum_{e \text{ into } v} f(e) = \sum_{e \text{ out of } v} f(e). \qquad \text{(conservation conditions)}$$



- for every $e \in E$: $0 \leq f(e) \leq c_e$ (capacity conditions)
- for every $v \in V \setminus \{s, t\}$:

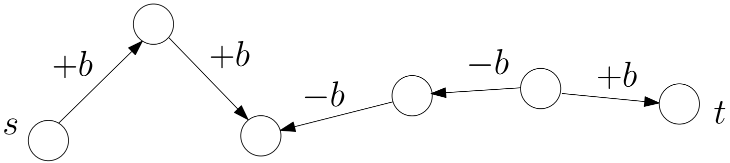
$$\sum_{e \text{ into } v} f(e) = \sum_{e \text{ out of } v} f(e). \qquad \text{(conservation conditions)}$$



- for an edge e correspondent to a forward edge :
  $b \leq c_e - f(e) \implies f(e) + b \leq c_e$

- for every $e \in E$: $0 \leq f(e) \leq c_e$ (capacity conditions)
- for every $v \in V \setminus \{s,t\}$:

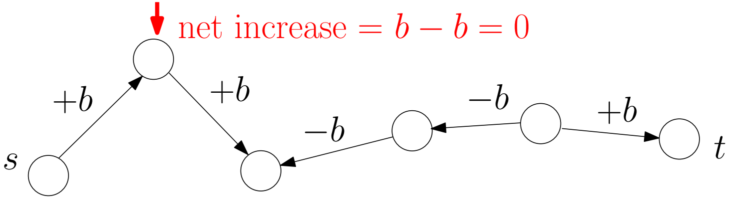
$$\sum_{e \text{ into } v} f(e) = \sum_{e \text{ out of } v} f(e). \qquad \text{(conservation conditions)}$$



- for an edge e correspondent to a forward edge :
  $b \leq c_e - f(e) \implies f(e) + b \leq c_e$
- for an edge e correspondent to a backward edge :
  $b \leq f(e) \implies f(e) - b \geq 0$

- for every $e \in E$: $0 \le f(e) \le c_e$ (capacity conditions)
- for every $v \in V \setminus \{s, t\}$:

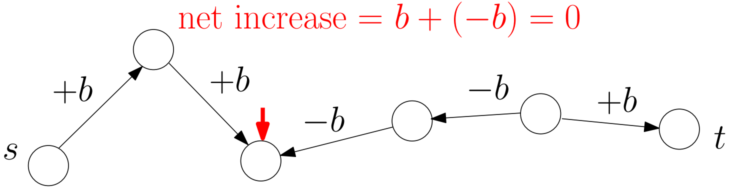
$$\sum_{e \text{ into } v} f(e) = \sum_{e \text{ out of } v} f(e). \qquad \text{(conservation conditions)}$$



- for an edge e correspondent to a forward edge :
  $b \le c_e - f(e) \implies f(e) + b \le c_e$
- for an edge e correspondent to a backward edge :
  $b \le f(e) \implies f(e) - b \ge 0$

- for every $e \in E$: $0 \leq f(e) \leq c_e$ (capacity conditions)
- for every $v \in V \setminus \{s, t\}$:

$$\sum_{e \text{ into } v} f(e) = \sum_{e \text{ out of } v} f(e). \qquad \text{(conservation conditions)}$$

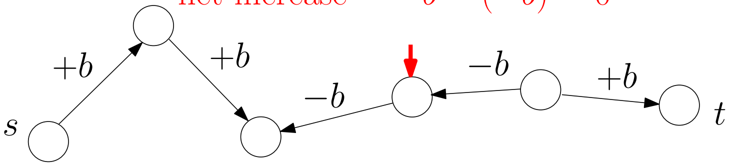


- for an edge e correspondent to a forward edge :
  $b \leq c_e - f(e) \implies f(e) + b \leq c_e$
- for an edge e correspondent to a backward edge :
  $b \leq f(e) \implies f(e) - b \geq 0$

- for every $e \in E$: $0 \leq f(e) \leq c_e$ (capacity conditions)
- for every $v \in V \setminus \{s, t\}$:

$$\sum_{e \text{ into } v} f(e) = \sum_{e \text{ out of } v} f(e). \qquad \text{(conservation conditions)}$$

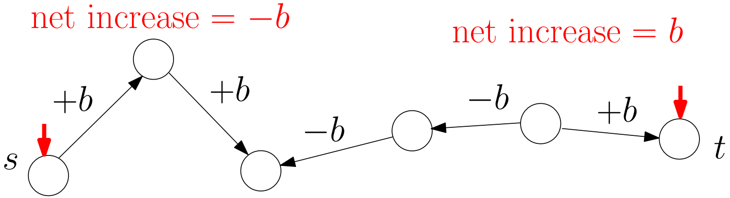
$\text{net increase} = -b - (-b) = 0$



- for an edge e correspondent to a forward edge :
  $b \leq c_e - f(e) \implies f(e) + b \leq c_e$
- for an edge e correspondent to a backward edge :
  $b \leq f(e) \implies f(e) - b \geq 0$

- for every $e \in E$: $0 \leq f(e) \leq c_e$ (capacity conditions)
- for every $v \in V \setminus \{s, t\}$:

$$\sum_{e \text{ into } v} f(e) = \sum_{e \text{ out of } v} f(e). \qquad \text{(conservation conditions)}$$



- for an edge e correspondent to a forward edge :
  $b \leq c_e - f(e) \implies f(e) + b \leq c_e$
- for an edge e correspondent to a backward edge :
  $b \leq f(e) \implies f(e) - b \geq 0$

Correctness of Ford-Fulkerson's Method

1. The procedure augment(f, P) maintains the two conditions:
   - for every $e \in E$: $0 \le f(e) \le c_e$ (capacity conditions)
   - for every $v \in V \setminus \{s, t\}$:

$$\sum_{e \in \delta_{\mathsf{in}}(v)} f(e) = \sum_{e \in \delta_{\mathsf{out}}(v)} f(e). \qquad \text{(conservation conditions)}$$

2. When Ford-Fulkerson's Method terminates, val(f) is maximized
3. Ford-Fulkerson's Method will terminate

Def. An s-t cut of $G = (V, E)$ is a pair $(S \subseteq V, T = V \setminus S)$ such that $s \in S$ and $t \in T$.

Def. An s-t cut of $G = (V, E)$ is a pair $(S \subseteq V, T = V \setminus S)$ such that $s \in S$ and $t \in T$.

Def. The cut value of an s-t cut is

$$c(S, T) := \sum_{e=(u,v)\in E: u\in S, v\in T} c_e.$$

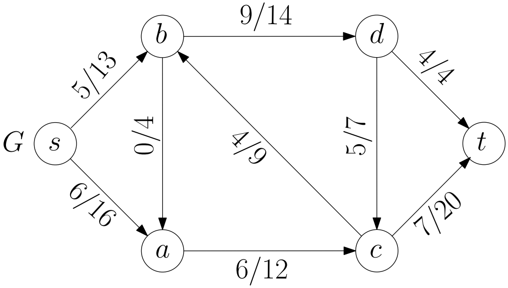
Def. An s-t cut of $G = (V, E)$ is a pair $(S \subseteq V, T = V \setminus S)$ such that $s \in S$ and $t \in T$.

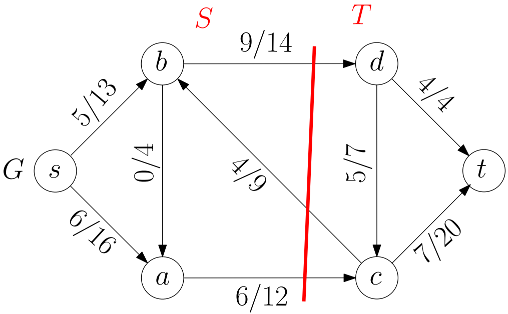
Def. The cut value of an s-t cut is

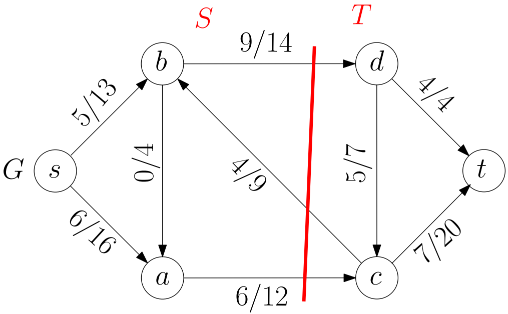
$$c(S, T) := \sum_{e=(u,v) \in E : u \in S, v \in T} c_e.$$

Def. Given an s-t flow f and an s-t cut (S, T), the net flow sent from S to T is

$$f(S, T) := \sum_{e=(u,v) \in E : u \in S, v \in T} f(e) - \sum_{e=(u,v) \in E : u \in T, v \in S} f(e).$$

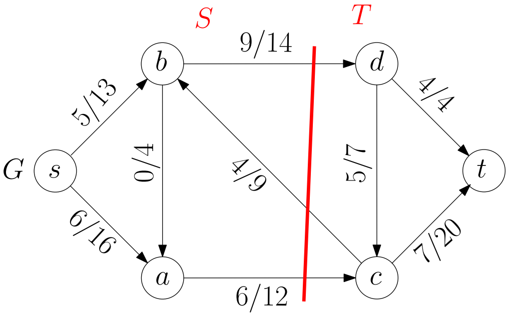

G
s
b
a
d
c
t
5/13
6/16
0/4
9/14
4/9
6/12
5/7
4/4
7/20

S
T
G
s
b
a
d
c
t
5/13
6/16
0/4
9/14
4/9
6/12
5/7
4/4
7/20



$$c(S, T) = 14 + 12 = 26$$

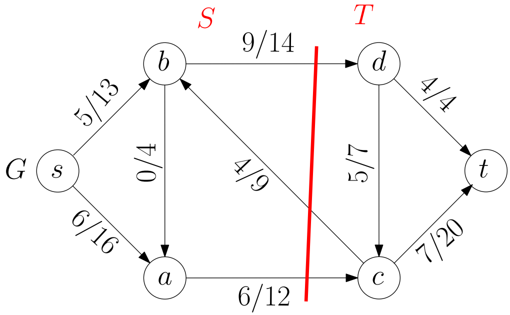
$$f(S, T) = 9 + 6 - 4 = 11$$



$c(S,T) = 14 + 12 = 26$

$f(S,T) = 9 + 6 - 4 = 11$

Obs. $f(S,T) \le c(S,T)$ s-t cut (S,T).

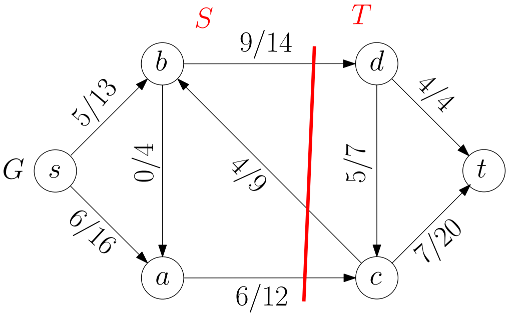


$c(S, T) = 14 + 12 = 26$

$f(S, T) = 9 + 6 - 4 = 11$

Obs. $f(S, T) \leq c(S, T)$ s-t cut (S, T).

Obs. $f(S, T) = \text{val}(f)$ for any s-t flow f and any s-t cut (S, T).



$c(S,T) = 14 + 12 = 26$

$f(S,T) = 9 + 6 - 4 = 11$

Obs. $f(S,T) \leq c(S,T)$ s-t cut (S,T).

Obs. $f(S,T) = \text{val}(f)$ for any s-t flow f and any s-t cut (S,T).

Coro.

$$\text{val}(f) \leq \min_{s\text{-}t \text{ cut } (S,T)} c(S,T) \text{ for every } s\text{-}t \text{ flow} f.$$

Coro. $\mathsf{val}(f) \leq \min_{s\text{-}t\ \mathsf{cut}\ (S,T)} c(S,T)$ for every s-t flow$f$.

Coro. $$\text{val}(f) \le \min_{s\text{-}t \text{ cut } (S,T)} c(S,T) \text{ for every } s\text{-}t \text{ flow} f.$$

We will prove

Main Lemma The flow f found by the Ford-Fulkerson's Method satisfies

$$\text{val}(f) = c(S,T) \text{ for some } s\text{-}t \text{ cut } (S,T).$$

Coro. $$\text{val}(f) \le \min_{s\text{-}t \text{ cut } (S,T)} c(S,T) \text{ for every } s\text{-}t \text{ flow} f.$$

We will prove

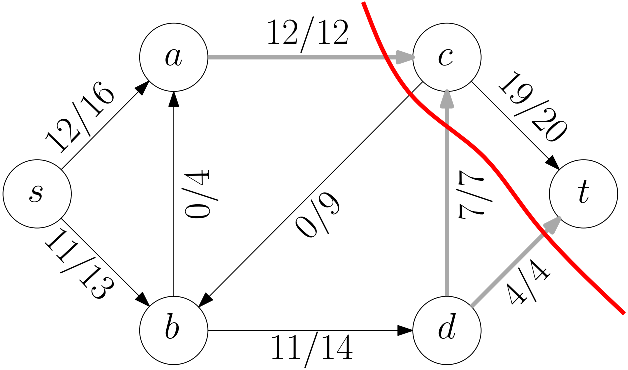
Main Lemma The flow f found by the Ford-Fulkerson's Method satisfies
$$\text{val}(f) = c(S,T) \text{ for some } s\text{-}t \text{ cut } (S,T).$$

Corollary and Main Lemma implies

Maximum Flow Minimum Cut Theorem
$$\sup_{s\text{-}t \text{ flow } f} \text{val}(f) = \min_{s\text{-}t \text{ cut } (S,T)} c(S,T).$$

Maximum Flow Minimum Cut Theorem

$$\sup_{s\text{-}t \text{ flow } f} \mathsf{val}(f) = \min_{s\text{-}t \text{ cut } (S,T)} c(S,T).$$



Main Lemma The flow f found by the Ford-Fulkerson's Method satisfies

$$\text{val}(f) = c(S,T) \text{ for some } s\text{-}t \text{ cut } (S,T).$$

Proof of Main Lemma.

Main Lemma The flow f found by the Ford-Fulkerson's Method satisfies

$$\text{val}(f) = c(S,T) \text{ for some } s\text{-}t \text{ cut } (S,T).$$

Proof of Main Lemma.

- When algorithm terminates, no path from s to t in G_f,

Main Lemma The flow f found by the Ford-Fulkerson's Method satisfies

$$\text{val}(f) = c(S,T) \text{ for some } s\text{-}t \text{ cut } (S,T).$$

Proof of Main Lemma.

- When algorithm terminates, no path from s to t in G_f,
- What can we say about G_f?

Main Lemma The flow f found by the Ford-Fulkerson's Method satisfies

$$\text{val}(f) = c(S, T) \text{ for some } s\text{-}t \text{ cut } (S, T).$$

Proof of Main Lemma.

- When algorithm terminates, no path from s to t in G_f,
- What can we say about G_f?
- There is a s-t cut (S, T), such that there are no edges from S to T

Main Lemma The flow f found by the Ford-Fulkerson's Method satisfies

$$\text{val}(f) = c(S,T) \text{ for some } s\text{-}t \text{ cut } (S,T).$$

Proof of Main Lemma.

- When algorithm terminates, no path from s to t in G_f,
- What can we say about G_f?
- There is a s-t cut (S,T), such that there are no edges from S to T
- For every $e = (u,v) \in E, u \in S, v \in T$, we have $f(e) = c_e$

Main Lemma The flow f found by the Ford-Fulkerson's Method satisfies

$$\text{val}(f) = c(S,T) \text{ for some } s\text{-}t \text{ cut } (S,T).$$

Proof of Main Lemma.

- When algorithm terminates, no path from s to t in G_f,
- What can we say about G_f?
- There is a s-t cut (S,T), such that there are no edges from S to T
- For every $e = (u,v) \in E, u \in S, v \in T$, we have $f(e) = c_e$
- For every $e = (u,v) \in E, u \in T, v \in S$, we have $f(e) = 0$

Main Lemma The flow f found by the Ford-Fulkerson's Method satisfies

$$\text{val}(f) = c(S,T) \text{ for some } s\text{-}t \text{ cut } (S,T).$$

Proof of Main Lemma.

- When algorithm terminates, no path from s to t in G_f,
- What can we say about G_f?
- There is a s-t cut (S,T), such that there are no edges from S to T
- For every $e = (u,v) \in E, u \in S, v \in T$, we have $f(e) = c_e$
- For every $e = (u,v) \in E, u \in T, v \in S$, we have $f(e) = 0$
- Thus,
  $$\text{val}(f) = f(S,T) = \sum_{e=(u,v)\in E, u\in S, v\in T} f(e) - \sum_{e=(u,v)\in E, u\in T, v\in S} f(e) = \sum_{e=(u,v)\in E, u\in S, v\in T} c_e = c(S,T).$$

□

Correctness of Ford-Fulkerson's Method

1. The procedure augment(f, P) maintains the two conditions:
   - for every $e \in E$: $0 \le f(e) \le c_e$ (capacity conditions)
   - for every $v \in V \setminus \{s, t\}$:

$$\sum_{e \in \delta_{\text{in}}(v)} f(e) = \sum_{e \in \delta_{\text{out}}(v)} f(e). \qquad \text{(conservation conditions)}$$

2. When Ford-Fulkerson's Method terminates, val(f) is maximized
3. Ford-Fulkerson's Method will terminate

Ford-Fulkerson's Method will Terminate

Intuition:

- In every iteration, we increase the flow value by some amount

Ford-Fulkerson's Method will Terminate

Intuition:

- In every iteration, we increase the flow value by some amount
- There is a maximum flow value

Ford-Fulkerson's Method will Terminate

Intuition:

- In every iteration, we increase the flow value by some amount
- There is a maximum flow value
- So the algorithm will finally reach the maximum value

Ford-Fulkerson's Method will Terminate

Intuition:

- In every iteration, we increase the flow value by some amount
- There is a maximum flow value
- So the algorithm will finally reach the maximum value

However, the algorithm may not terminate if some capacities are irrational numbers. ("Pathological cases")

Lemma Ford-Fulkerson's Method will terminate if all capacities are integers.

Proof.

Lemma Ford-Fulkerson's Method will terminate if all capacities are integers.

Proof.

- The maximum flow value is finite (not ∞).
- In every iteration, we increase the flow value by at least 1.
- So the algorithm will terminate. □

Lemma Ford-Fulkerson's Method will terminate if all capacities are integers.

Proof.

- The maximum flow value is finite (not ∞).
- In every iteration, we increase the flow value by at least 1.
- So the algorithm will terminate. □

- Integers can be replaced by rational numbers.

Correctness of Ford-Fulkerson's Method

1. The procedure augment(f, P) maintains the two conditions:
   - for every $e \in E$: $0 \leq f(e) \leq c_e$ (capacity conditions)
   - for every $v \in V \setminus \{s, t\}$:

$$\sum_{e \in \delta_{\mathsf{in}}(v)} f(e) = \sum_{e \in \delta_{\mathsf{out}}(v)} f(e). \qquad \text{(conservation conditions)}$$

2. When Ford-Fulkerson's Method terminates, val(f) is maximized
3. Ford-Fulkerson's Method will terminate

Outline

1. Network Flow
2. Ford-Fulkerson Method
3. Correctness of Ford-Fulkerson's Method and Maximum Flow Minimum Cut Theorem
4. Running Time of Ford-Fulkerson-Type Algorithm
    - Shortest Augmenting Path Algorithm
    - Capacity-Scaling Algorithm
5. Bipartite Matching Problem
6. s-t Edge-Disjoint Paths Problem
7. More Applications

Running time of the Generic Ford-Fulkerson's Algorithm

Ford-Fulkerson(G, s, t, c)

```
1: let $f(e) \leftarrow 0$ for every $e$ in $G$
2: **while** there is a path from $s$ to $t$ in $G_f$ **do**
3:     let $P$ be any simple path from $s$ to $t$ in $G_f$
4:     $f \leftarrow$augment$(f, P)$
5: **return** $f$
```

- $O(m)$-time for Steps 3 and 4 in each iteration
- Total time $= O(m) \times$ number of iterations

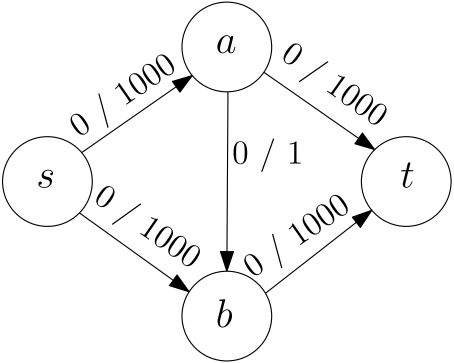
Running time of the Generic Ford-Fulkerson's Algorithm

Ford-Fulkerson(G, s, t, c)

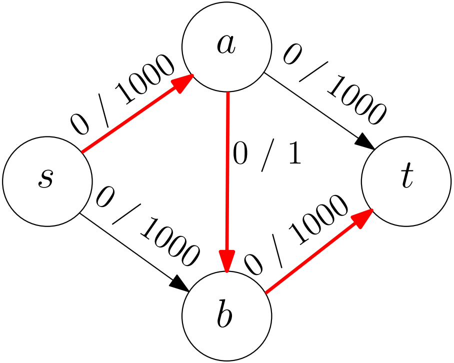
```
1: let $f(e) \leftarrow 0$ for every $e$ in $G$
2: while there is a path from $s$ to $t$ in $G_f$ do
3:     let $P$ be any simple path from $s$ to $t$ in $G_f$
4:     $f \leftarrow$augment$(f, P)$
5: return $f$
```

- $O(m)$-time for Steps 3 and 4 in each iteration
- Total time $= O(m) \times$ number of iterations
- Assume all capacities are integers, then algorithm may run up to val(f^*) iterations, where f^* is the optimum flow
- Total time $= O(m \cdot \text{val}(f^*))$
- Running time is "Pseudo-polynomial"

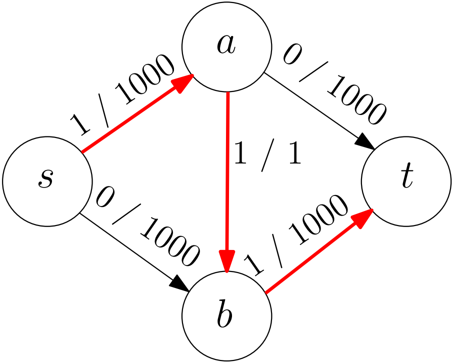
The Upper Bound on Running Time Is Tight!



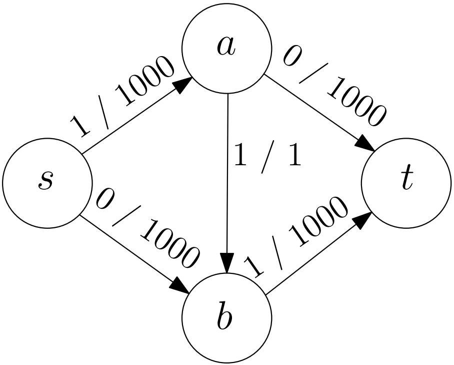
The Upper Bound on Running Time Is Tight!



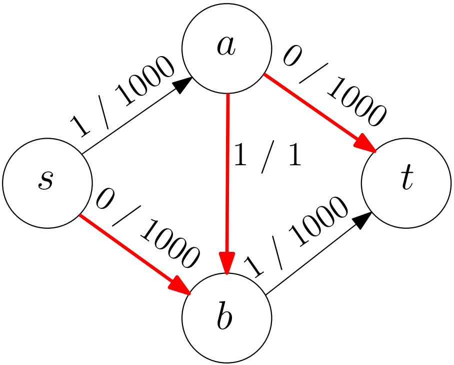
The Upper Bound on Running Time Is Tight!



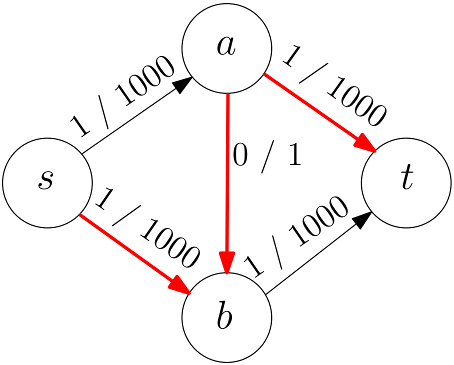
The Upper Bound on Running Time Is Tight!



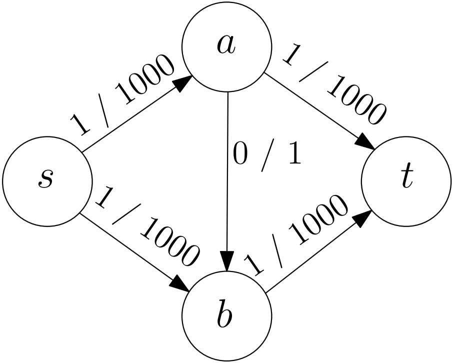
The Upper Bound on Running Time Is Tight!



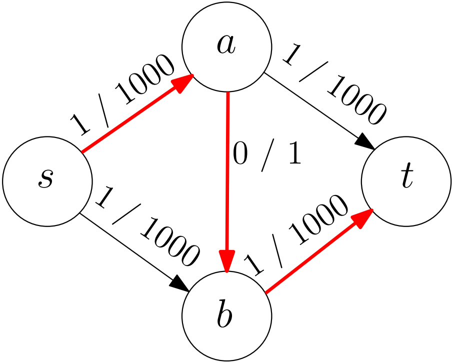
The Upper Bound on Running Time Is Tight!



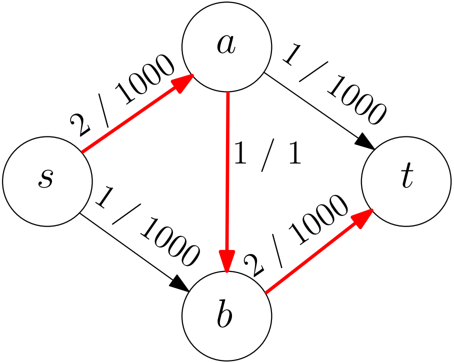
The Upper Bound on Running Time Is Tight!



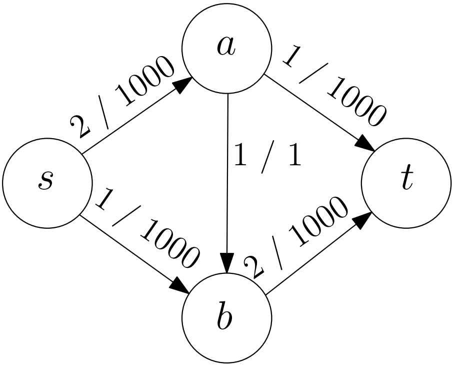
The Upper Bound on Running Time Is Tight!



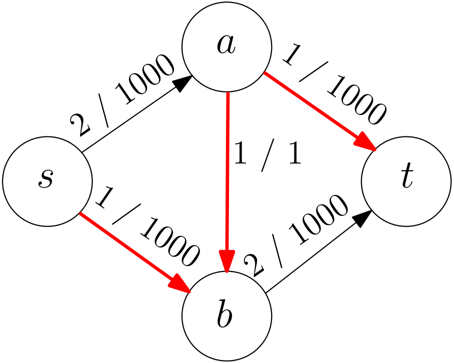
The Upper Bound on Running Time Is Tight!



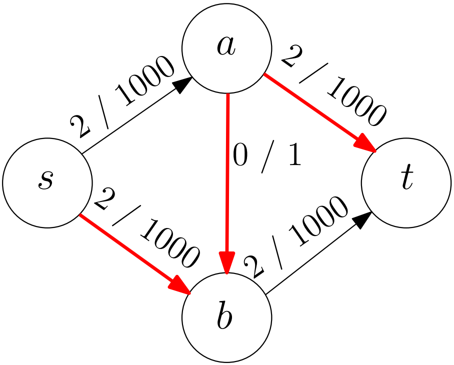
The Upper Bound on Running Time Is Tight!



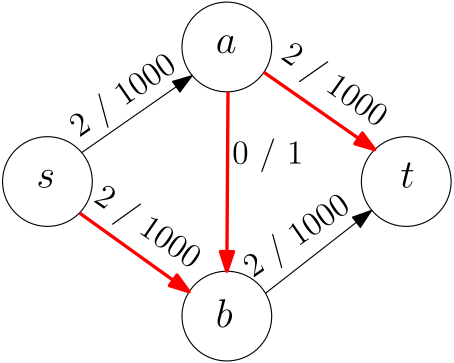
The Upper Bound on Running Time Is Tight!



The Upper Bound on Running Time Is Tight!



The Upper Bound on Running Time Is Tight!



Better choices for choosing augmentation paths:

- Choose the shortest augmentation path
- Choose the augmentation path with the largest bottleneck capacity

Outline

1. Network Flow

2. Ford-Fulkerson Method

3. Correctness of Ford-Fulkerson's Method and Maximum Flow Minimum Cut Theorem

4. Running Time of Ford-Fulkerson-Type Algorithm
   - Shortest Augmenting Path Algorithm
   - Capacity-Scaling Algorithm

5. Bipartite Matching Problem

6. s-t Edge-Disjoint Paths Problem

7. More Applications

Shortest Augmenting Path

```
shortest-augmenting-path$(G, s, t, c)$
let $f(e) \leftarrow 0$ for every $e$ in $G$
while there is a path from $s$ to $t$ in $G_f$ do
    $P \leftarrow$ breadth-first-search$(G_f, s, t)$
    $f \leftarrow$augment$(f, P)$
return $f$
```

Due to [Dinitz 1970] and [Edmonds-Karp, 1970]

Running Time of Shortest Augmenting Path Algorithm

Lemma 1. Throughout the algorithm, length of shortest path from s to t in G_f never decreases.
2. After at most m shortest path augmentations, the length of the shortest path from s to t in G_f strictly increases.

Running Time of Shortest Augmenting Path Algorithm

Lemma 1. Throughout the algorithm, length of shortest path from s to t in G_f never decreases.
2. After at most m shortest path augmentations, the length of the shortest path from s to t in G_f strictly increases.

- Length of shortest path is between 1 and $n - 1$

Running Time of Shortest Augmenting Path Algorithm

Lemma 1. Throughout the algorithm, length of shortest path from s to t in G_f never decreases.
2. After at most m shortest path augmentations, the length of the shortest path from s to t in G_f strictly increases.

- Length of shortest path is between 1 and $n-1$
- Algorithm takes at most $O(mn)$ iterations

Running Time of Shortest Augmenting Path Algorithm

Lemma 1. Throughout the algorithm, length of shortest path from s to t in G_f never decreases.
2. After at most m shortest path augmentations, the length of the shortest path from s to t in G_f strictly increases.

- Length of shortest path is between 1 and $n - 1$
- Algorithm takes at most $O(mn)$ iterations
- Shortest path from s to t can be found in $O(m)$ time using BFS

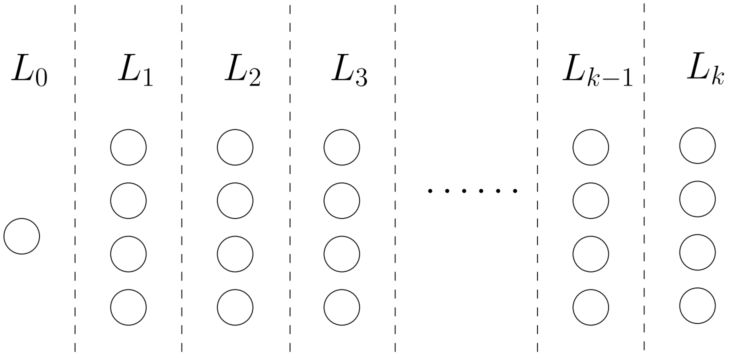
Running Time of Shortest Augmenting Path Algorithm

Lemma 1. Throughout the algorithm, length of shortest path from s to t in G_f never decreases.
2. After at most m shortest path augmentations, the length of the shortest path from s to t in G_f strictly increases.

- Length of shortest path is between 1 and $n-1$
- Algorithm takes at most $O(mn)$ iterations
- Shortest path from s to t can be found in $O(m)$ time using BFS

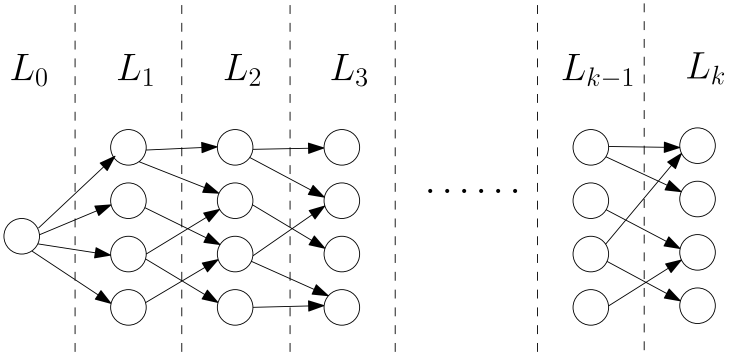
Theorem The shortest-augmenting-path algorithm runs in time $O(m^2n)$.

Proof of Lemma: Focus on G_f



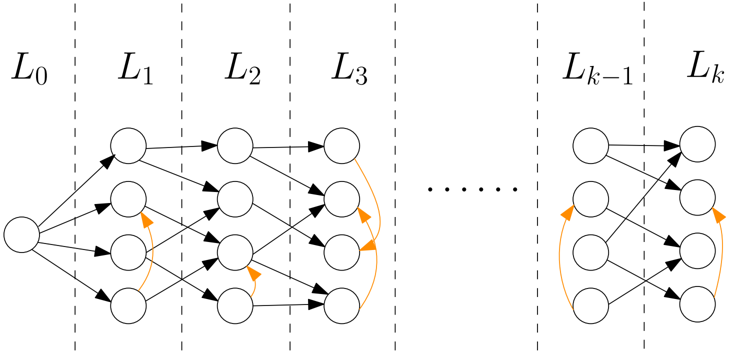
- Divide V into levels: L_i contains the set of vertices v such that the length of shortest path from s to v in G_f is i

Proof of Lemma: Focus on G_f



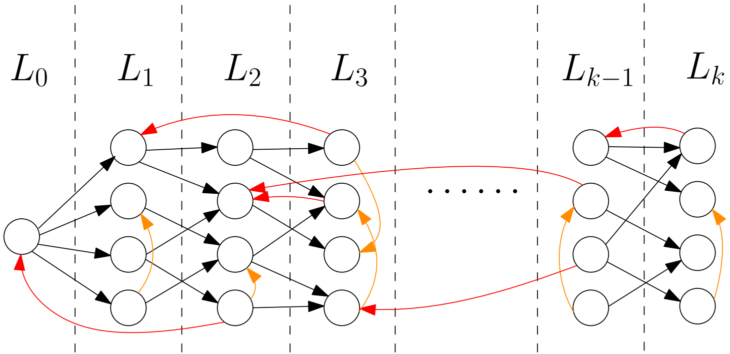
- Divide V into levels: L_i contains the set of vertices v such that the length of shortest path from s to v in G_f is i
- Forth edges : edges from L_i to L_{i+1} for some i

Proof of Lemma: Focus on G_f



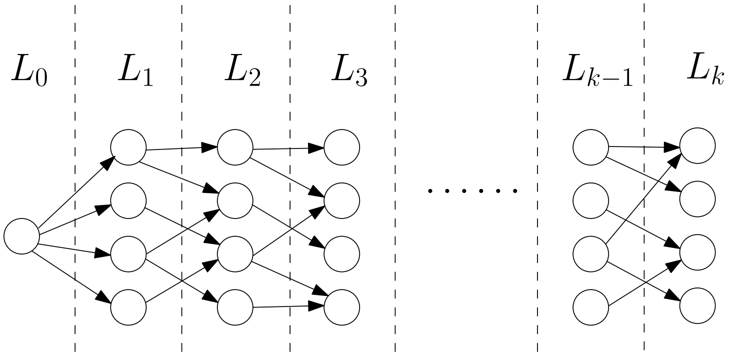
- Divide V into levels: L_i contains the set of vertices v such that the length of shortest path from s to v in G_f is i
- Forth edges : edges from L_i to L_{i+1} for some i
- Side edges : edges from L_i to L_i for some i

Proof of Lemma: Focus on G_f



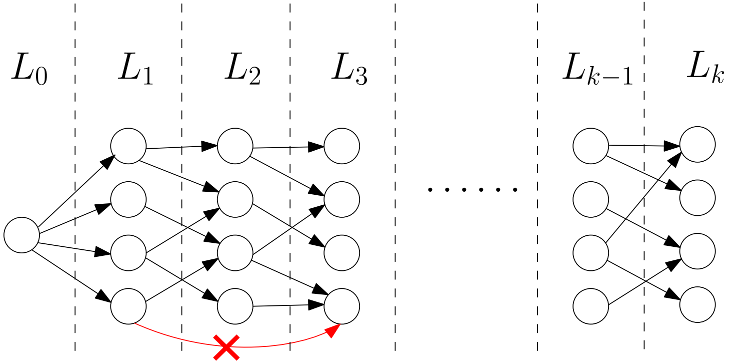
- Divide V into levels: L_i contains the set of vertices v such that the length of shortest path from s to v in G_f is i
- Forth edges : edges from L_i to L_{i+1} for some i
- Side edges : edges from L_i to L_i for some i
- Back edges: edges from L_i to L_j for some $i > j$

Proof of Lemma: Focus on G_f



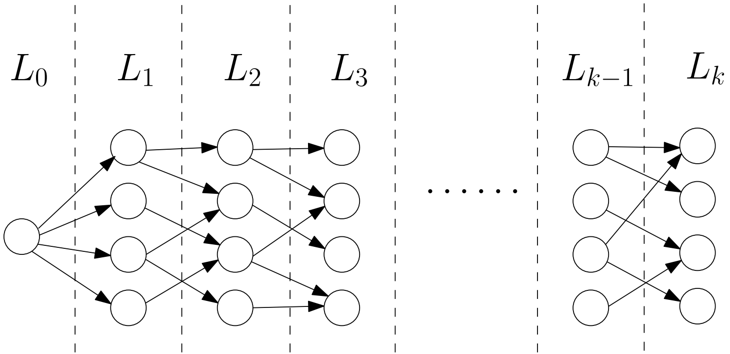
- Divide V into levels: L_i contains the set of vertices v such that the length of shortest path from s to v in G_f is i
- Forth edges : edges from L_i to L_{i+1} for some i
- Side edges : edges from L_i to L_i for some i
- Back edges: edges from L_i to L_j for some $i > j$

Proof of Lemma: Focus on G_f



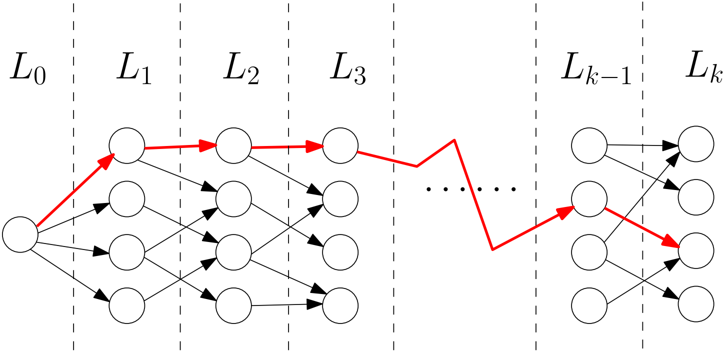
- Divide V into levels: L_i contains the set of vertices v such that the length of shortest path from s to v in G_f is i
- Forth edges : edges from L_i to L_{i+1} for some i
- Side edges : edges from L_i to L_i for some i
- Back edges: edges from L_i to L_j for some $i > j$
- No jump edges: edges from L_i to L_j for $j \geq i + 2$

Proof of Lemma: Focus on G_f



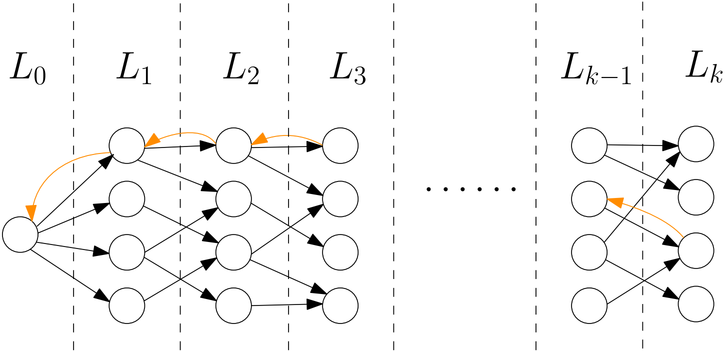
- Divide V into levels: L_i contains the set of vertices v such that the length of shortest path from s to v in G_f is i
- Forth edges : edges from L_i to L_{i+1} for some i
- Side edges : edges from L_i to L_i for some i
- Back edges: edges from L_i to L_j for some $i > j$
- No jump edges: edges from L_i to L_j for $j \geq i + 2$

Proof of Lemma: Focus on G_f



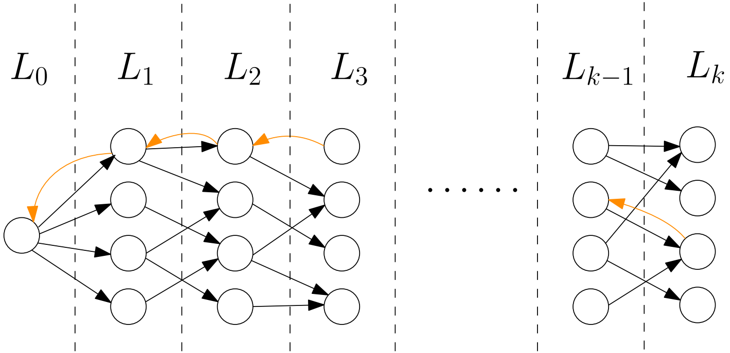
- Assuming $t \in L_k$, shortest $s \to t$ path uses k forth edges

Proof of Lemma: Focus on G_f



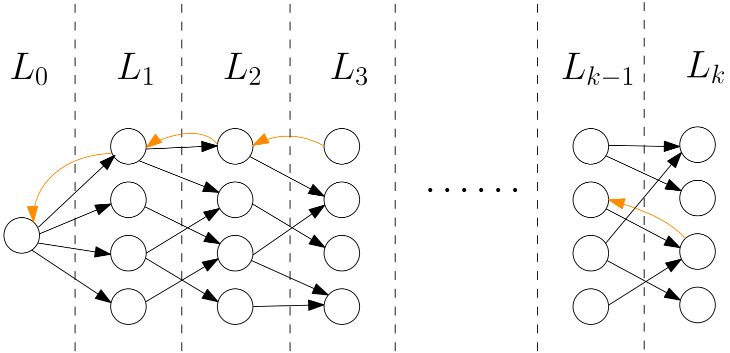
- Assuming $t \in L_k$, shortest $s \to t$ path uses k forth edges
- After augmenting along the path, back edges will be added to G_f

Proof of Lemma: Focus on G_f



- Assuming $t \in L_k$, shortest $s \to t$ path uses k forth edges
- After augmenting along the path, back edges will be added to G_f
- One forth edge will be removed from G_f

Proof of Lemma: Focus on G_f



- Assuming $t \in L_k$, shortest $s \to t$ path uses k forth edges
- After augmenting along the path, back edges will be added to G_f
- One forth edge will be removed from G_f
- In $O(m)$ iterations, there will be no paths from s to t of length k in G_f.

Improving the $O(m^2n)$ Running Time for Shortest Path Augmentation Algorithm

Improving the $O(m^2n)$ Running Time for Shortest Path Augmentation Algorithm

- For some networks, $O(mn)$-augmentations are necessary

Improving the $O(m^2n)$ Running Time for Shortest Path Augmentation Algorithm

- For some networks, $O(mn)$-augmentations are necessary
- Idea for improved running time: reduce running time for each iteration

Improving the $O(m^2n)$ Running Time for Shortest Path Augmentation Algorithm

- For some networks, $O(mn)$-augmentations are necessary
- Idea for improved running time: reduce running time for each iteration
- Simple idea $\Rightarrow O(mn^2)$ [Dinic 1970]

Improving the $O(m^2n)$ Running Time for Shortest Path Augmentation Algorithm

- For some networks, $O(mn)$-augmentations are necessary
- Idea for improved running time: reduce running time for each iteration
- Simple idea $\Rightarrow O(mn^2)$ [Dinic 1970]
- Dynamic Trees $\Rightarrow O(mn\log n)$ [Sleator-Tarjan 1983]

Outline

1. Network Flow
2. Ford-Fulkerson Method
3. Correctness of Ford-Fulkerson's Method and Maximum Flow Minimum Cut Theorem
4. Running Time of Ford-Fulkerson-Type Algorithm
   - Shortest Augmenting Path Algorithm
   - Capacity-Scaling Algorithm
5. Bipartite Matching Problem
6. s-t Edge-Disjoint Paths Problem
7. More Applications

Capacity-Scaling Algorithm

- Idea: find the augment path from s to t with the largest bottleneck capacity

Capacity-Scaling Algorithm

- Idea: find the augment path from s to t with the sufficiently large bottleneck capacity

Capacity-Scaling Algorithm

- Idea: find the augment path from s to t with the sufficiently large bottleneck capacity
- Assumption: Capacities are integers between 1 and C

Capacity-Scaling Algorithm

- Idea: find the augment path from s to t with the sufficiently large bottleneck capacity
- Assumption: Capacities are integers between 1 and C

capacity-scaling(G, s, t, c)

```
let $f(e) \leftarrow 0$ for every $e$ in $G$
$\Delta \leftarrow$ largest power of $2$ which is at most $C$
while $\Delta \geq 1$ do do
    while there exists an augmenting path $P$ with bottleneck capacity at least $\Delta$ do
        $f \leftarrow$augment$(f, P)$
    $\Delta \leftarrow \Delta/2$
return $f$
```

Obs. The outer while loop repeats $1 + \lfloor \log_2 C \rfloor$ times.

Obs. The outer while loop repeats $1 + \lfloor \log_2 C \rfloor$ times.

Lemma At the beginning of Δ-scale phase, the value of the max-flow is at most val$(f) + 2m\Delta$.

Obs. The outer while loop repeats $1 + \lfloor \log_2 C \rfloor$ times.

Lemma At the beginning of Δ-scale phase, the value of the max-flow is at most val$(f) + 2m\Delta$.

- Each augmentation increases the flow value by at least Δ

Obs. The outer while loop repeats $1 + \lfloor \log_2 C \rfloor$ times.

Lemma At the beginning of Δ-scale phase, the value of the max-flow is at most val$(f) + 2m\Delta$.

- Each augmentation increases the flow value by at least Δ
- Thus, there are at most $2m$ augmentations for Δ-scale phase.

Obs. The outer while loop repeats $1 + \lfloor \log_2 C \rfloor$ times.

Lemma At the beginning of Δ-scale phase, the value of the max-flow is at most val$(f) + 2m\Delta$.

- Each augmentation increases the flow value by at least Δ
- Thus, there are at most $2m$ augmentations for Δ-scale phase.

Theorem The number of augmentations in the scaling max-flow algorithm is at most $O(m \log C)$. The running time of the algorithm is $O(m^2 \log C)$.

Polynomial Time

Assume all capacities are integers between 1 and C.

Ford-Fulkerson	$O(m^2C)$	pseudo-polynomial
Capacity-scaling:	$O(m^2 \log C)$	weakly-polynomial
Shortest-Path-Augmenting:	$O(m^2n)$	strongly-polynomial

- Polynomial : weakly-polynomial and strongly-polynomial

Polynomial Time

Assume all capacities are integers between 1 and C.

Ford-Fulkerson	$O(m^2C)$	pseudo-polynomial
Capacity-scaling:	$O(m^2 \log C)$	weakly-polynomial
Shortest-Path-Augmenting:	$O(m^2n)$	strongly-polynomial

- Polynomial : weakly-polynomial and strongly-polynomial

Brief History

Algorithm	**Year**	**Time**	**Description**
Ford-Fulkerson	1956	$O(mf)$	Ford-Fulkerson Method.
Edmonds-Karp	1972	$O(nm^2)$	Shortest Augmenting Paths
Dinic	1970	$O(n^2m)$	SAP with blocking Flows
Goldberg-Tarjan	1988	$O(n^3)$	Generic Push-Relabel
Goldberg-Tarjan	1988	$O(n^2\sqrt{m})$	PR using highest-label nodes
Chen et al.	2022	$O(m^{1+o(1)})$	LP-solver, dynamic algorithms

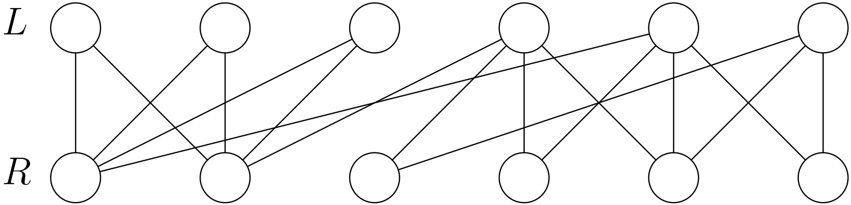
- Chen et al. [Chen-Kyng-Liu-Peng-Gutenberg-Sachdeva, 2022].

Outline

1. Network Flow
2. Ford-Fulkerson Method
3. Correctness of Ford-Fulkerson's Method and Maximum Flow Minimum Cut Theorem
4. Running Time of Ford-Fulkerson-Type Algorithm
   - Shortest Augmenting Path Algorithm
   - Capacity-Scaling Algorithm
5. Bipartite Matching Problem
6. s-t Edge-Disjoint Paths Problem
7. More Applications

Bipartite Graphs

Def. A graph $G = (V, E)$ is bipartite if the vertices V can be partitioned into two subsets L and R such that every edge in E is between a vertex in L and a vertex in R.



Def. Given a bipartite graph $G = (L \cup R, E)$, a matching in G is a set $M \subseteq E$ of edges such that every vertex in V is an endpoint of at most one edge in M.

Def. Given a bipartite graph $G = (L \cup R, E)$, a matching in G is a set $M \subseteq E$ of edges such that every vertex in V is an endpoint of at most one edge in M.

Maximum Bipartite Matching Problem

Input: bipartite graph $G = (L \cup R, E)$

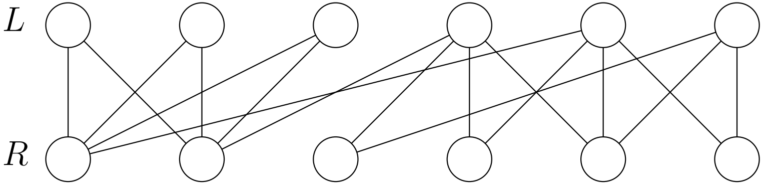
Output: a matching M in G of the maximum size

Def. Given a bipartite graph $G = (L \cup R, E)$, a matching in G is a set $M \subseteq E$ of edges such that every vertex in V is an endpoint of at most one edge in M.

Maximum Bipartite Matching Problem

Input: bipartite graph $G = (L \cup R, E)$

Output: a matching M in G of the maximum size

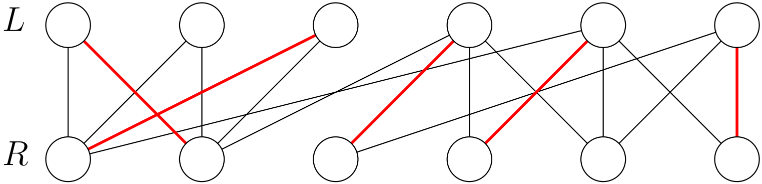


Def. Given a bipartite graph $G = (L \cup R, E)$, a matching in G is a set $M \subseteq E$ of edges such that every vertex in V is an endpoint of at most one edge in M.

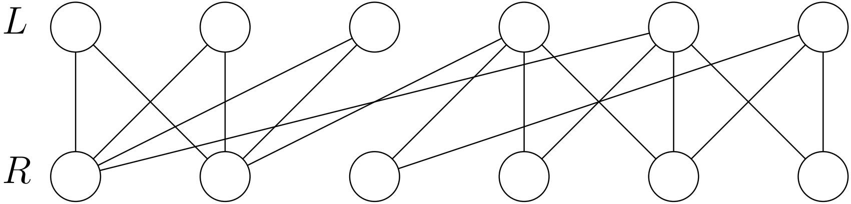
Maximum Bipartite Matching Problem

Input: bipartite graph $G = (L \cup R, E)$

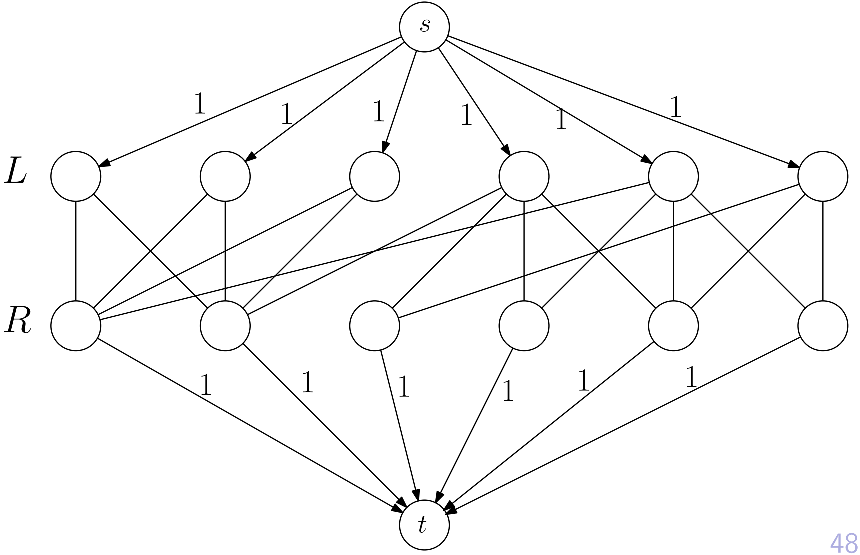
Output: a matching M in G of the maximum size



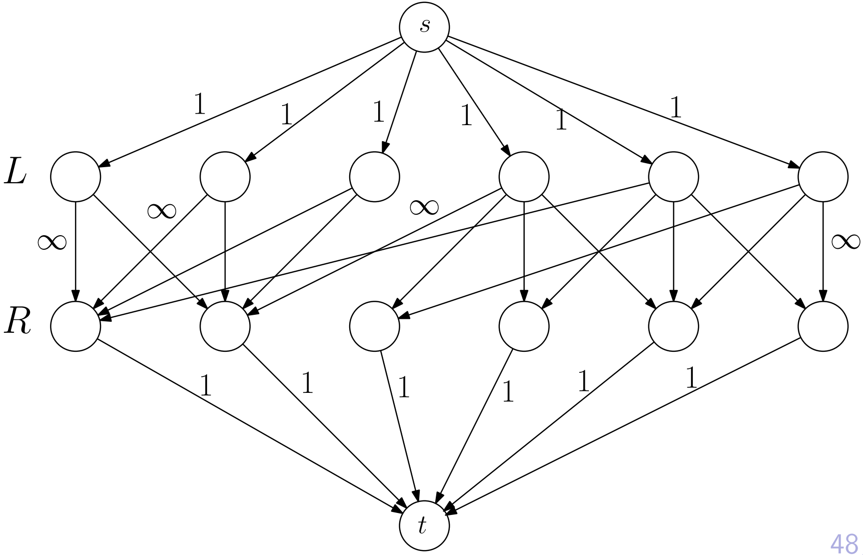
Reduce Maximum Bipartite Matching to Maximum Flow Problem



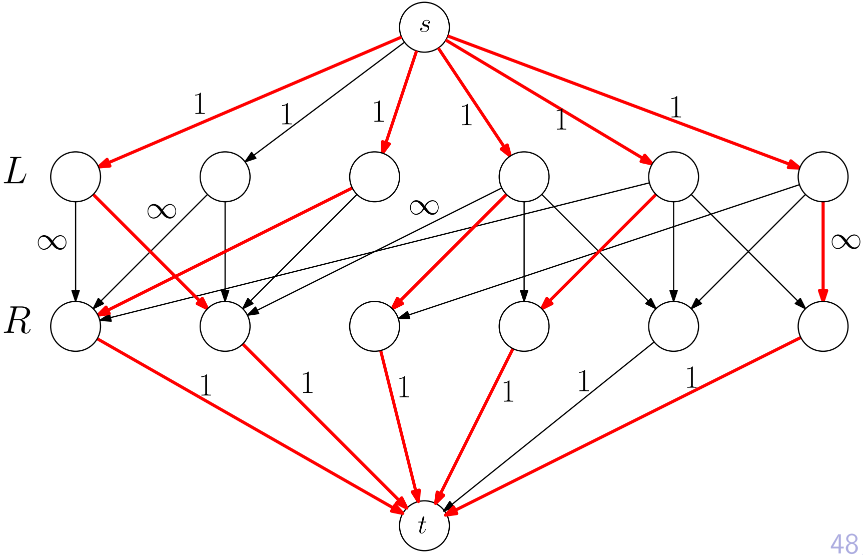
Reduce Maximum Bipartite Matching to Maximum Flow Problem



Reduce Maximum Bipartite Matching to Maximum Flow Problem



Reduce Maximum Bipartite Matching to Maximum Flow Problem



Reduce Maximum Bipartite Matching to Maximum Flow Problem

- Create a digraph $G' = (L \cup R \cup \{s, t\}, E')$ with capacity $c : E' \to \mathbb{R}_{\geq 0}$:
  - Add a source s and a sink t
  - Add an edge from s to each vertex $u \in L$ of capacity 1
  - Add an edge from each vertex $v \in R$ to t of capacity 1
  - Direct all edges in E from L to R, and assign ∞ capacity (or capacity 1) to them

Reduce Maximum Bipartite Matching to Maximum Flow Problem

- Create a digraph $G' = (L \cup R \cup \{s, t\}, E')$ with capacity $c : E' \to \mathbb{R}_{\geq 0}$:
  - Add a source s and a sink t
  - Add an edge from s to each vertex $u \in L$ of capacity 1
  - Add an edge from each vertex $v \in R$ to t of capacity 1
  - Direct all edges in E from L to R, and assign ∞ capacity (or capacity 1) to them
- Compute the maximum flow from s to t in G'

Reduce Maximum Bipartite Matching to Maximum Flow Problem

- Create a digraph $G' = (L \cup R \cup \{s, t\}, E')$ with capacity $c : E' \to \mathbb{R}_{\geq 0}$:
  - Add a source s and a sink t
  - Add an edge from s to each vertex $u \in L$ of capacity 1
  - Add an edge from each vertex $v \in R$ to t of capacity 1
  - Direct all edges in E from L to R, and assign ∞ capacity (or capacity 1) to them
- Compute the maximum flow from s to t in G'
- The maximum flow gives a matching

Reduce Maximum Bipartite Matching to Maximum Flow Problem

- Create a digraph $G' = (L \cup R \cup \{s, t\}, E')$ with capacity $c : E' \to \mathbb{R}_{\geq 0}$:
  - Add a source s and a sink t
  - Add an edge from s to each vertex $u \in L$ of capacity 1
  - Add an edge from each vertex $v \in R$ to t of capacity 1
  - Direct all edges in E from L to R, and assign ∞ capacity (or capacity 1) to them
- Compute the maximum flow from s to t in G'
- The maximum flow gives a matching
- Running time:

Reduce Maximum Bipartite Matching to Maximum Flow Problem

- Create a digraph $G' = (L \cup R \cup \{s, t\}, E')$ with capacity $c : E' \to \mathbb{R}_{\geq 0}$:
  - Add a source s and a sink t
  - Add an edge from s to each vertex $u \in L$ of capacity 1
  - Add an edge from each vertex $v \in R$ to t of capacity 1
  - Direct all edges in E from L to R, and assign ∞ capacity (or capacity 1) to them
- Compute the maximum flow from s to t in G'
- The maximum flow gives a matching
- Running time:
  - Ford-Fulkerson: $O(m \times \text{max flow value}) = O(mn)$.

Reduce Maximum Bipartite Matching to Maximum Flow Problem

- Create a digraph $G' = (L \cup R \cup \{s, t\}, E')$ with capacity $c : E' \to \mathbb{R}_{\geq 0}$:
  - Add a source s and a sink t
  - Add an edge from s to each vertex $u \in L$ of capacity 1
  - Add an edge from each vertex $v \in R$ to t of capacity 1
  - Direct all edges in E from L to R, and assign ∞ capacity (or capacity 1) to them
- Compute the maximum flow from s to t in G'
- The maximum flow gives a matching
- Running time:
  - Ford-Fulkerson: $O(m \times \text{max flow value}) = O(mn)$.
  - Hopcroft-Karp: $O(mn^{1/2})$ time

Lemma Size of max matching = value of max flow in G'

Lemma Size of max matching = value of max flow in G'

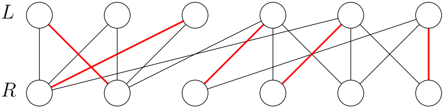
Proof. $\leq$.

Given a maximum matching $M \subseteq E$, send a flow along each edge $e \in M$ and thus we have a flow of value $|M|$. □

Lemma Size of max matching = value of max flow in G'

Proof. $\leq$.

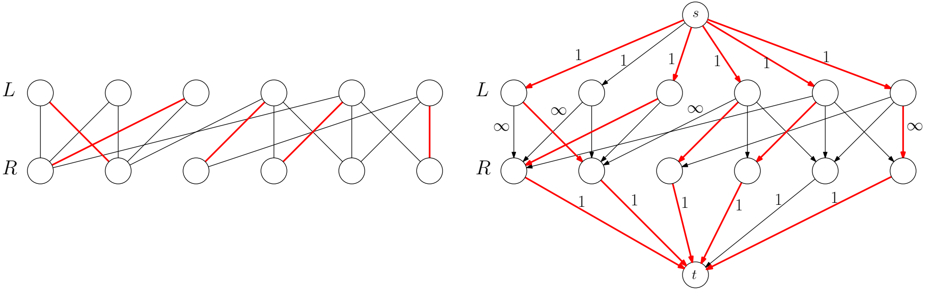
Given a maximum matching $M \subseteq E$, send a flow along each edge $e \in M$ and thus we have a flow of value $|M|$. □



Lemma Size of max matching = value of max flow in G'

Proof. $\leq$.

Given a maximum matching $M \subseteq E$, send a flow along each edge $e \in M$ and thus we have a flow of value $|M|$. □



Lemma Size of max matching = value of max flow in G'

Lemma Size of max matching = value of max flow in G'

Proof. $\geq$.

Lemma Size of max matching $=$ value of max flow in G'

Proof. $\geq$.

- The maximum flow f in G' is integral since all capacities are integral

Lemma Size of max matching = value of max flow in G'

Proof. $\geq$.

- The maximum flow f in G' is integral since all capacities are integral
- Let M to be the set of edges e from L to R with $f(e) = 1$

Lemma Size of max matching = value of max flow in G'

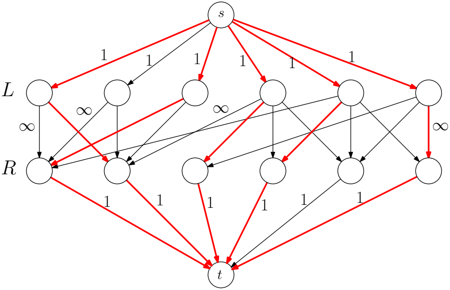
Proof. $\geq$.

- The maximum flow f in G' is integral since all capacities are integral
- Let M to be the set of edges e from L to R with $f(e) = 1$
- M is a matching of size that equals to the flow value □

Lemma Size of max matching = value of max flow in G'

Proof. $\geq$.

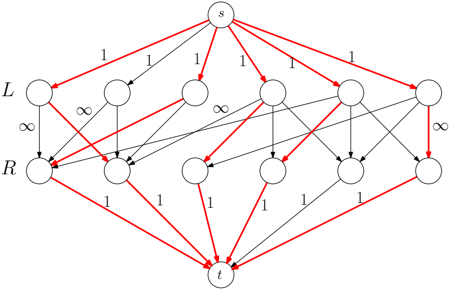
- The maximum flow f in G' is integral since all capacities are integral
- Let M to be the set of edges e from L to R with $f(e) = 1$
- M is a matching of size that equals to the flow value □

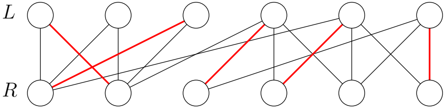


Lemma Size of max matching $=$ value of max flow in G'

Proof. $\geq$.

- The maximum flow f in G' is integral since all capacities are integral
- Let M to be the set of edges e from L to R with $f(e) = 1$
- M is a matching of size that equals to the flow value □





Perfect Matching

Def. Given a bipartite graph $G = (L \cup R, E)$ with $|L| = |R|$, a perfect matching M of G is a matching such that every vertex $v \in L \cup R$ participates in exactly one edge in M.

Perfect Matching

Def. Given a bipartite graph $G = (L \cup R, E)$ with $|L| = |R|$, a perfect matching M of G is a matching such that every vertex $v \in L \cup R$ participates in exactly one edge in M.

Assuming $|L| = |R| = n$, when does $G = (L \cup R, E)$ have a perfect matching?

Perfect Matching

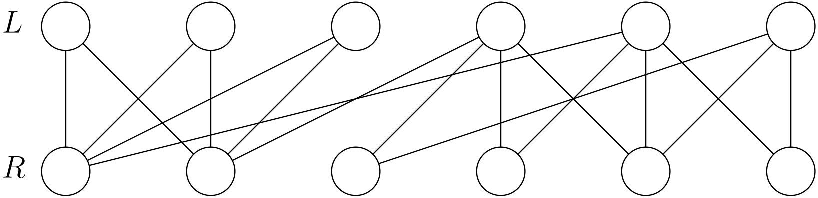
Def. Given a bipartite graph $G = (L \cup R, E)$ with $|L| = |R|$, a perfect matching M of G is a matching such that every vertex $v \in L \cup R$ participates in exactly one edge in M.

Assuming $|L| = |R| = n$, when does $G = (L \cup R, E)$ not have a perfect matching?

Perfect Matching

Def. Given a bipartite graph $G = (L \cup R, E)$ with $|L| = |R|$, a perfect matching M of G is a matching such that every vertex $v \in L \cup R$ participates in exactly one edge in M.

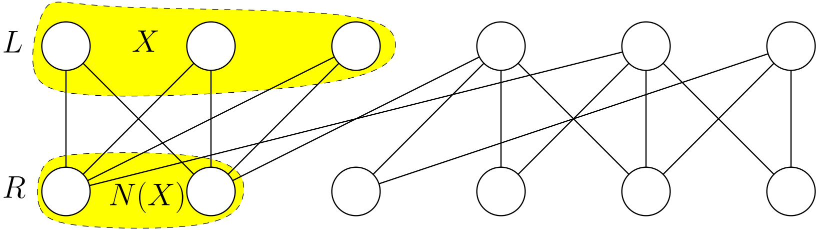
Assuming $|L| = |R| = n$, when does $G = (L \cup R, E)$ not have a perfect matching?



Perfect Matching

Def. Given a bipartite graph $G = (L \cup R, E)$ with $|L| = |R|$, a perfect matching M of G is a matching such that every vertex $v \in L \cup R$ participates in exactly one edge in M.

Assuming $|L| = |R| = n$, when does $G = (L \cup R, E)$ not have a perfect matching?

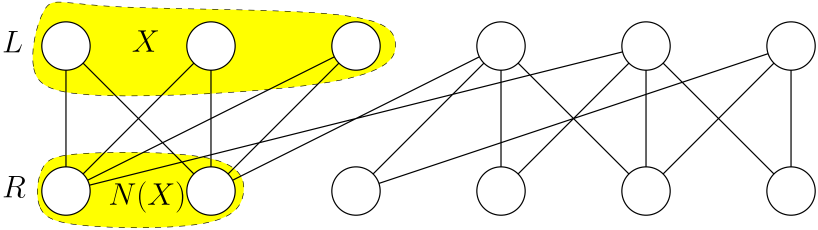


- For $X \subseteq L$, define $N(X) = \{v \in R : \exists u \in X, (u, v) \in E\}$

Perfect Matching

Def. Given a bipartite graph $G = (L \cup R, E)$ with $|L| = |R|$, a perfect matching M of G is a matching such that every vertex $v \in L \cup R$ participates in exactly one edge in M.

Assuming $|L| = |R| = n$, when does $G = (L \cup R, E)$ not have a perfect matching?

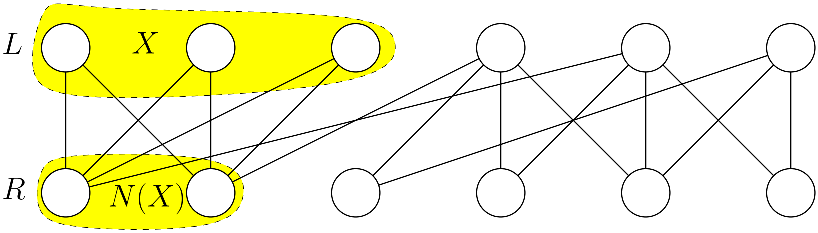


- For $X \subseteq L$, define $N(X) = \{v \in R : \exists u \in X, (u, v) \in E\}$
- $|N(X)| < X$ for some $X \subseteq L \implies$ no perfect matching

Perfect Matching

Def. Given a bipartite graph $G = (L \cup R, E)$ with $|L| = |R|$, a perfect matching M of G is a matching such that every vertex $v \in L \cup R$ participates in exactly one edge in M.

Assuming $|L| = |R| = n$, when does $G = (L \cup R, E)$ not have a perfect matching?



- For $X \subseteq L$, define $N(X) = \{v \in R : \exists u \in X, (u, v) \in E\}$
- $|N(X)| < X$ for some $X \subseteq L$ $\Longleftrightarrow$ no perfect matching

Hall's Theorem Let $G = (L \cup R, E)$ be a bipartite graph with $|L| = |R|$. Then G has a perfect matching if and only if $|N(X)| \geq |X|$ for every $X \subseteq L$.

Hall's Theorem Let $G = (L \cup R, E)$ be a bipartite graph with $|L| = |R|$. Then G has a perfect matching if and only if $|N(X)| \geq |X|$ for every $X \subseteq L$.

Proof. $\Longrightarrow$.

If G has a perfect matching, then vertices matched to $X \subseteq N(X)$; thus $|N(X)| \geq |X|$. □

Hall's Theorem Let $G = (L \cup R, E)$ be a bipartite graph with $|L| = |R|$. Then G has a perfect matching if and only if $|N(X)| \geq |X|$ for every $X \subseteq L$.

Hall's Theorem Let $G = (L \cup R, E)$ be a bipartite graph with $|L| = |R|$. Then G has a perfect matching if and only if $|N(X)| \geq |X|$ for every $X \subseteq L$.

Proof. $\Longleftarrow$.

Hall's Theorem Let $G = (L \cup R, E)$ be a bipartite graph with $|L| = |R|$. Then G has a perfect matching if and only if $|N(X)| \geq |X|$ for every $X \subseteq L$.

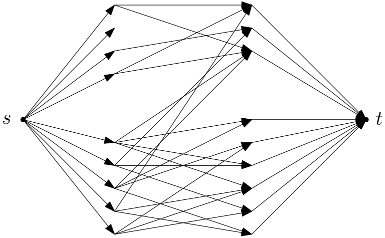
Proof. $\Longleftarrow$.

- Contrapositive: if no perfect matching, then $\exists X \subseteq L, |N(X)| < |X|$

Hall's Theorem Let $G = (L \cup R, E)$ be a bipartite graph with $|L| = |R|$. Then G has a perfect matching if and only if $|N(X)| \geq |X|$ for every $X \subseteq L$.

Proof. $\Longleftarrow$.

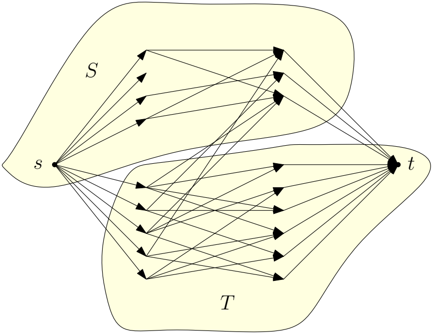
- Contrapositive: if no perfect matching, then $\exists X \subseteq L, |N(X)| < |X|$
- Consider the network flow instance



Hall's Theorem Let $G = (L \cup R, E)$ be a bipartite graph with $|L| = |R|$. Then G has a perfect matching if and only if $|N(X)| \geq |X|$ for every $X \subseteq L$.

Proof. $\Longleftarrow$.

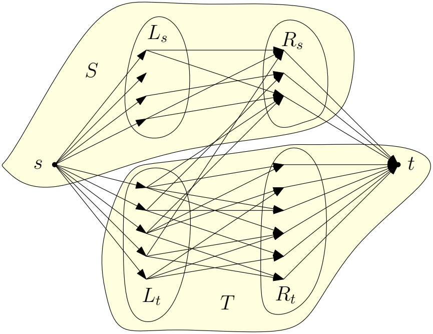
- Contrapositive: if no perfect matching, then $\exists X \subseteq L, |N(X)| < |X|$
- Consider the network flow instance
- There is a s-t cut (S, T) of value at most $n - 1$



Hall's Theorem Let $G = (L \cup R, E)$ be a bipartite graph with $|L| = |R|$. Then G has a perfect matching if and only if $|N(X)| \geq |X|$ for every $X \subseteq L$.

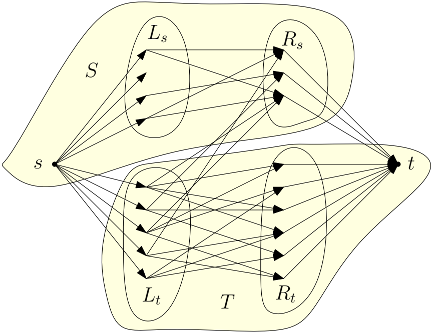
Proof. $\Longleftarrow$.

- Contrapositive: if no perfect matching, then $\exists X \subseteq L, |N(X)| < |X|$
- Consider the network flow instance
- There is a s-t cut (S, T) of value at most $n - 1$
- Define L_s, L_t, R_s, R_t as in figure □



Hall's Theorem Let $G = (L \cup R, E)$ be a bipartite graph with $|L| = |R|$. Then G has a perfect matching if and only if $|N(X)| \geq |X|$ for every $X \subseteq L$.

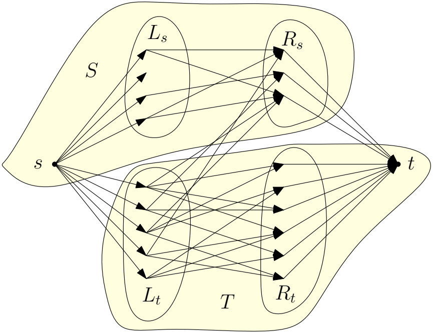
Proof. $\Longleftarrow$.



Hall's Theorem Let $G = (L \cup R, E)$ be a bipartite graph with $|L| = |R|$. Then G has a perfect matching if and only if $|N(X)| \geq |X|$ for every $X \subseteq L$.

Proof. $\Longleftarrow$.

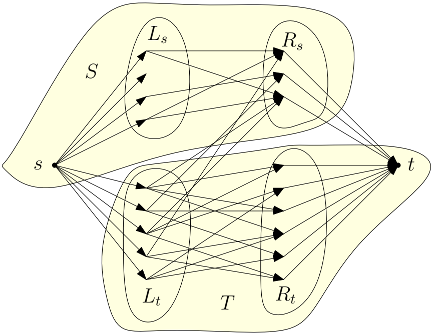
- Contrapositive: if no perfect matching, then $\exists X \subseteq L, |N(X)| < |X|$



Hall's Theorem Let $G = (L \cup R, E)$ be a bipartite graph with $|L| = |R|$. Then G has a perfect matching if and only if $|N(X)| \geq |X|$ for every $X \subseteq L$.

Proof. $\Longleftarrow$.

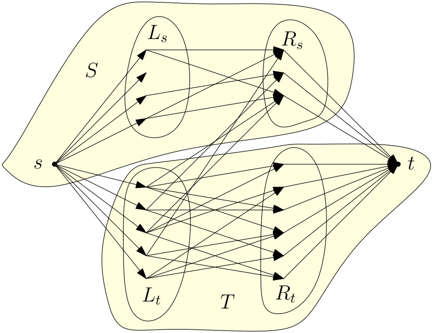
- Contrapositive: if no perfect matching, then $\exists X \subseteq L, |N(X)| < |X|$
- No edges from L_s to R_t, since their capacities are ∞



Hall's Theorem Let $G = (L \cup R, E)$ be a bipartite graph with $|L| = |R|$. Then G has a perfect matching if and only if $|N(X)| \geq |X|$ for every $X \subseteq L$.

Proof. $\Longleftarrow$.

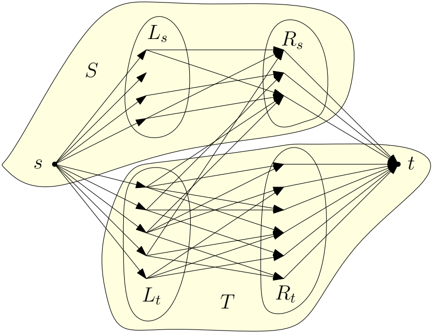
- Contrapositive: if no perfect matching, then $\exists X \subseteq L, |N(X)| < |X|$
- No edges from L_s to R_t, since their capacities are ∞
- $c(S, T) = |L_t| + |R_s| < n$



Hall's Theorem Let $G = (L \cup R, E)$ be a bipartite graph with $|L| = |R|$. Then G has a perfect matching if and only if $|N(X)| \geq |X|$ for every $X \subseteq L$.

Proof. $\Longleftarrow$.

- Contrapositive: if no perfect matching, then $\exists X \subseteq L, |N(X)| < |X|$
- No edges from L_s to R_t, since their capacities are ∞
- $c(S, T) = |L_t| + |R_s| < n$
- $|N(L_s)| \leq |R_s| < n - |L_t| = |L_s|$. □



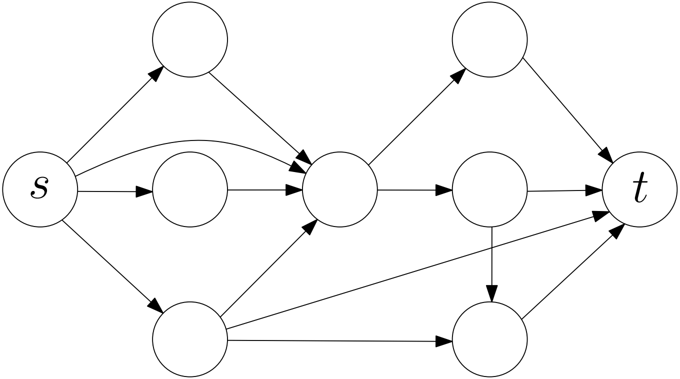
Outline

1. Network Flow
2. Ford-Fulkerson Method
3. Correctness of Ford-Fulkerson's Method and Maximum Flow Minimum Cut Theorem
4. Running Time of Ford-Fulkerson-Type Algorithm
   - Shortest Augmenting Path Algorithm
   - Capacity-Scaling Algorithm
5. Bipartite Matching Problem
6. s-t Edge-Disjoint Paths Problem
7. More Applications

s-t Edge Disjoint Paths

Input: a directed (or undirected) graph $G = (V, E)$ and $s, t \in V$

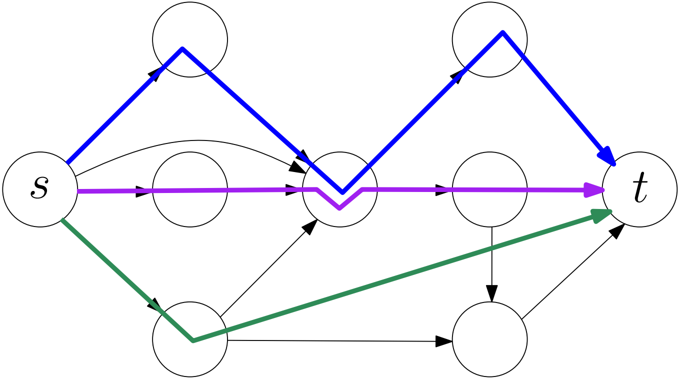
Output: the maximum number of edge-disjoint paths from s to t in G



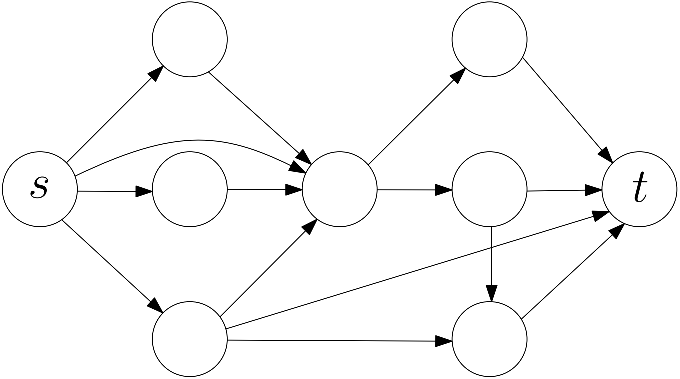
s-t Edge Disjoint Paths

Input: a directed (or undirected) graph $G = (V, E)$ and $s, t \in V$

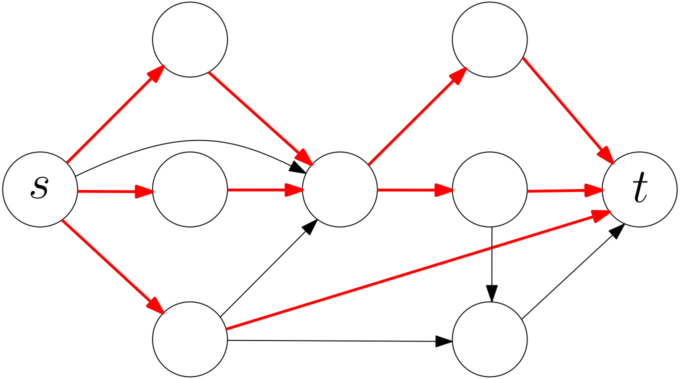
Output: the maximum number of edge-disjoint paths from s to t in G



- Solving the maximum flow problem, where all capacities are 1
- All flow values are integral (i.e, either 0 or 1)

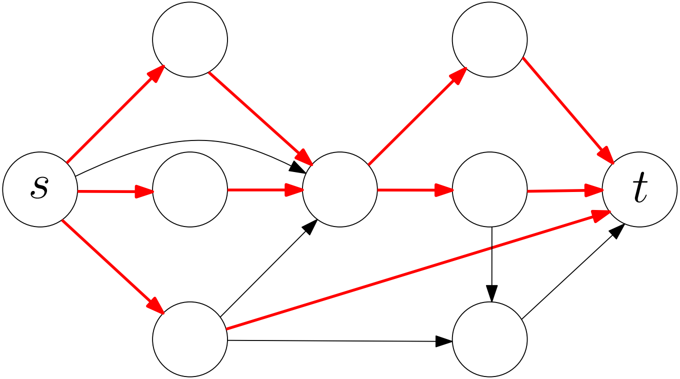


- Solving the maximum flow problem, where all capacities are 1
- All flow values are integral (i.e, either 0 or 1)



- Solving the maximum flow problem, where all capacities are 1
- All flow values are integral (i.e, either 0 or 1)

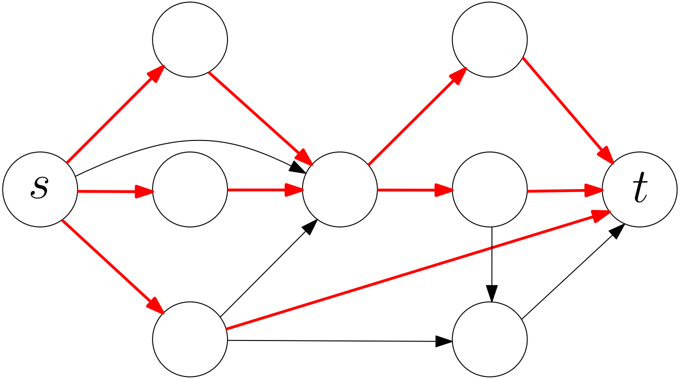
From flow to disjoint paths



- Solving the maximum flow problem, where all capacities are 1
- All flow values are integral (i.e, either 0 or 1)

From flow to disjoint paths

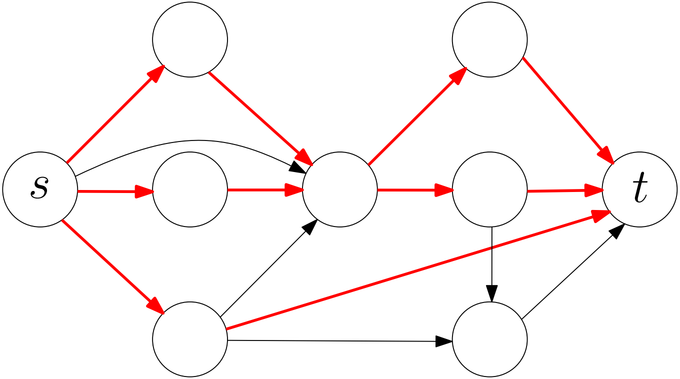
- find an arbitrary $s \to t$ path where all edges have flow value 1



- Solving the maximum flow problem, where all capacities are 1
- All flow values are integral (i.e, either 0 or 1)

From flow to disjoint paths

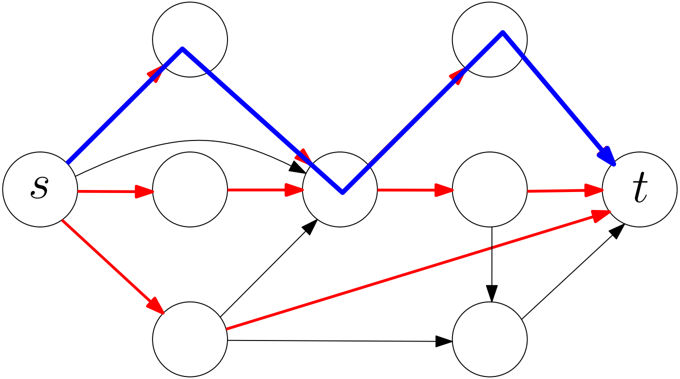
- find an arbitrary $s \to t$ path where all edges have flow value 1
- change the flow values of the path to 0 and repeat



- Solving the maximum flow problem, where all capacities are 1
- All flow values are integral (i.e, either 0 or 1)

From flow to disjoint paths

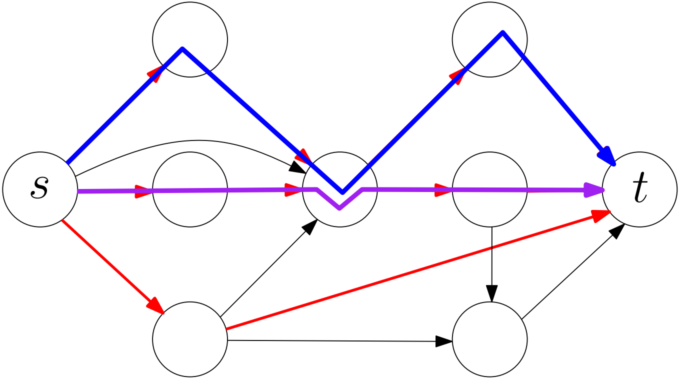
- find an arbitrary $s \to t$ path where all edges have flow value 1
- change the flow values of the path to 0 and repeat



- Solving the maximum flow problem, where all capacities are 1
- All flow values are integral (i.e, either 0 or 1)

From flow to disjoint paths

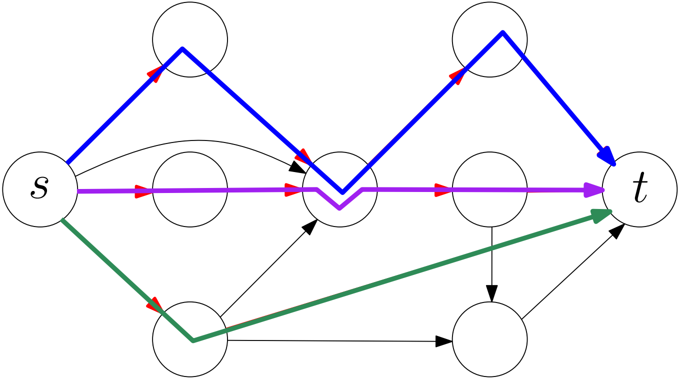
- find an arbitrary $s \to t$ path where all edges have flow value 1
- change the flow values of the path to 0 and repeat



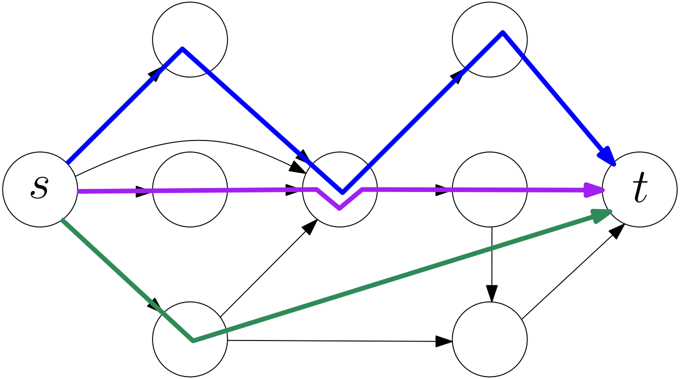
- Solving the maximum flow problem, where all capacities are 1
- All flow values are integral (i.e, either 0 or 1)

From flow to disjoint paths

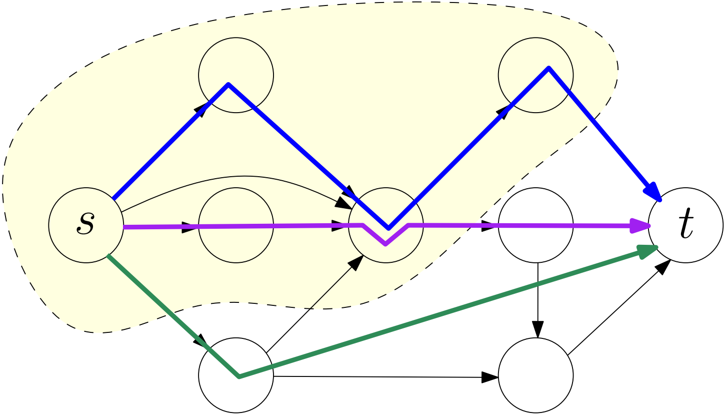
- find an arbitrary $s \to t$ path where all edges have flow value 1
- change the flow values of the path to 0 and repeat



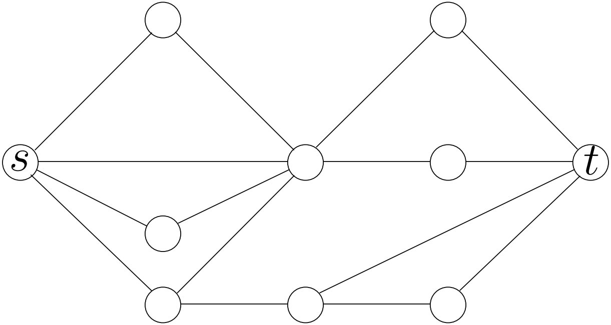
Theorem The maximum number of edge disjoint paths from s to t equals the minimum value of an s-t cut (S, T).



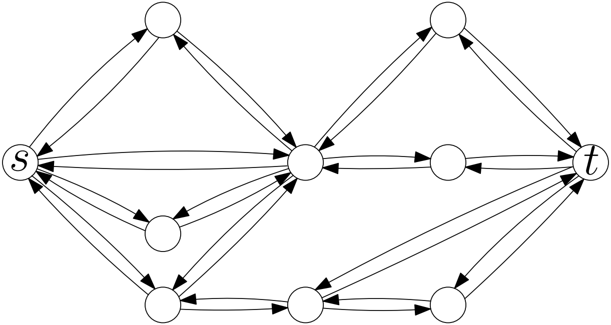
Theorem The maximum number of edge disjoint paths from s to t equals the minimum value of an s-t cut (S,T).



s-t Edge Disjoint Paths in Undirected Graphs

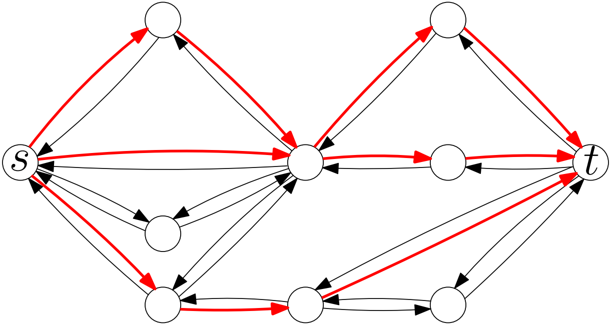


s-t Edge Disjoint Paths in Undirected Graphs



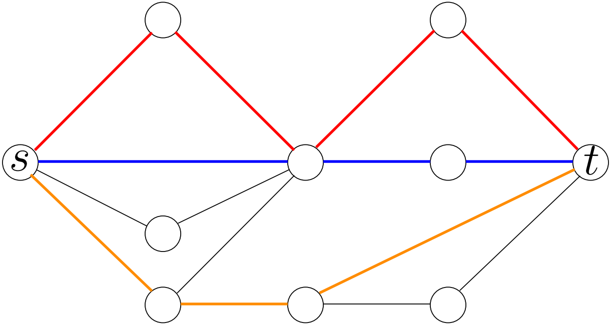
- an undirected edge → two anti-parallel directed edges.

s-t Edge Disjoint Paths in Undirected Graphs



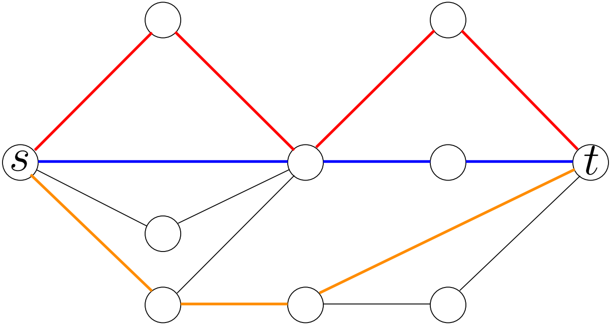
- an undirected edge → two anti-parallel directed edges.
- Solving the s-t maximum flow problem in the directed graph

s-t Edge Disjoint Paths in Undirected Graphs



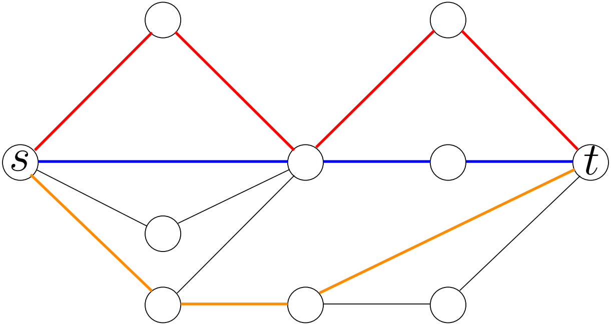
- an undirected edge → two anti-parallel directed edges.
- Solving the s-t maximum flow problem in the directed graph
- Convert the flow to paths

s-t Edge Disjoint Paths in Undirected Graphs



- an undirected edge → two anti-parallel directed edges.
- Solving the s-t maximum flow problem in the directed graph
- Convert the flow to paths
- Issue: both $e = (u, v)$ and $e' = (v, u)$ are used

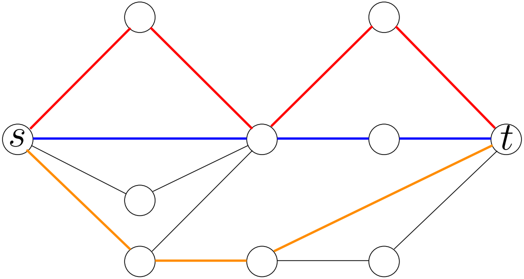
s-t Edge Disjoint Paths in Undirected Graphs



- an undirected edge → two anti-parallel directed edges.
- Solving the s-t maximum flow problem in the directed graph
- Convert the flow to paths
- Issue: both $e = (u, v)$ and $e' = (v, u)$ are used
- Fix: if this happens we change $f(e) = f(e') = 0$

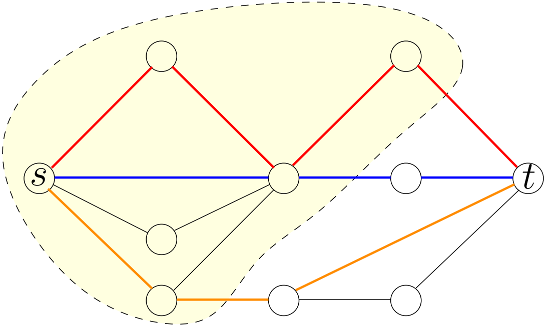
Menger's Theorem

Menger's Theorem In an undirected graph, the maximum number of edge-disjoint paths between s to t is equal to the minimum number of edges whose removal disconnects s and t.



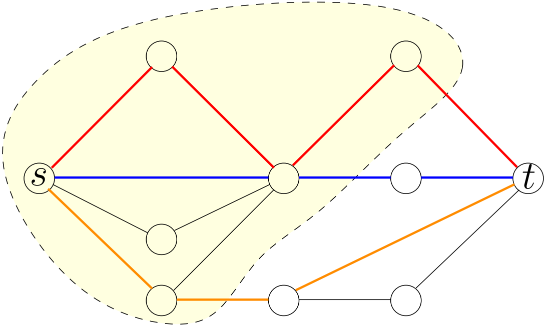
Menger's Theorem

Menger's Theorem In an undirected graph, the maximum number of edge-disjoint paths between s to t is equal to the minimum number of edges whose removal disconnects s and t.



Menger's Theorem

Menger's Theorem In an undirected graph, the maximum number of edge-disjoint paths between s to t is equal to the minimum number of edges whose removal disconnects s and t.

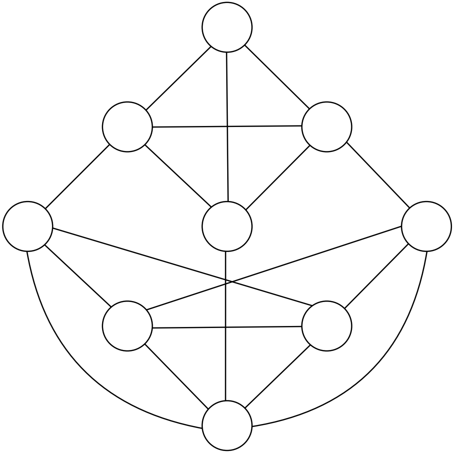


s-t connectivity measures how well s and t are connected.

Global Min-Cut Problem

Input: a connected graph $G = (V, E)$

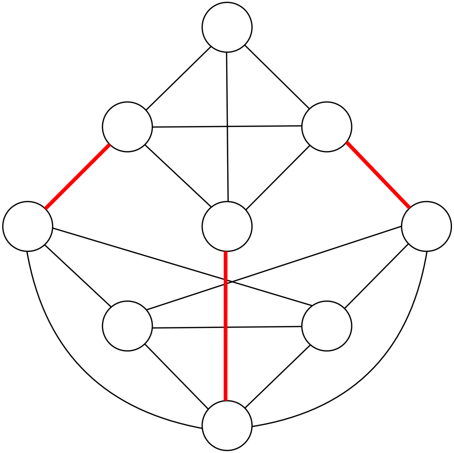
Output: the minimum number of edges whose removal will disconnect G



Global Min-Cut Problem

Input: a connected graph $G = (V, E)$

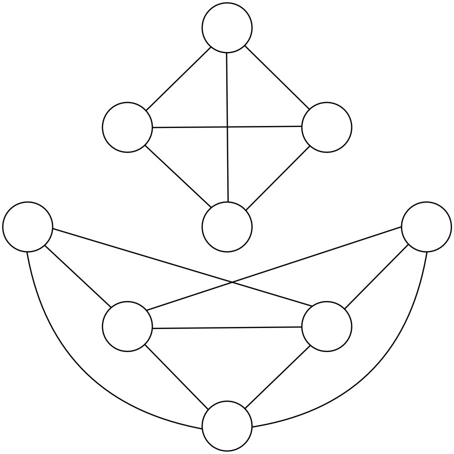
Output: the minimum number of edges whose removal will disconnect G



Global Min-Cut Problem

Input: a connected graph $G = (V, E)$

Output: the minimum number of edges whose removal will disconnect G



Solving Global Min-Cut Using Maximum Flow

```
let $G'$ be the directed graph obtained from $G$ by replacing every edge with two anti-parallel edges
for every pair $s \neq t$ of vertices do
    obtain the minimum cut separating $s$ and $t$ in $G$, by solving the maximum flow instance with graph $G'$,source $s$ and sink $t$
output the smallest minimum cut we found
```

- Need to solve $\Theta(n^2)$ maximum flow instances

Solving Global Min-Cut Using Maximum Flow

```
let $G'$ be the directed graph obtained from $G$ by replacing every edge with two anti-parallel edges
for every pair $s \neq t$ of vertices do
    obtain the minimum cut separating $s$ and $t$ in $G$, by solving the maximum flow instance with graph $G'$,source $s$ and sink $t$
output the smallest minimum cut we found
```

- Need to solve $\Theta(n^2)$ maximum flow instances
- Can we do better?

Solving Global Min-Cut Using Maximum Flow

```
let $G'$ be the directed graph obtained from $G$ by replacing every edge with two anti-parallel edges
for every pair $s \neq t$ of vertices do
    obtain the minimum cut separating $s$ and $t$ in $G$, by solving the maximum flow instance with graph $G'$,source $s$ and sink $t$
output the smallest minimum cut we found
```

- Need to solve $\Theta(n^2)$ maximum flow instances
- Can we do better?
- Yes. We can fix s. We only need to enumerate t

Outline

1. Network Flow

2. Ford-Fulkerson Method

3. Correctness of Ford-Fulkerson's Method and Maximum Flow Minimum Cut Theorem

4. Running Time of Ford-Fulkerson-Type Algorithm
   - Shortest Augmenting Path Algorithm
   - Capacity-Scaling Algorithm

5. Bipartite Matching Problem

6. s-t Edge-Disjoint Paths Problem

7. More Applications

Extension of Network Flow: Circulation Problem

Input: A digraph $G = (V, E)$

capacities $c \in \mathbb{Z}_{\geq 0}^E$

supply vector $d \in \mathbb{Z}^V$ with $\sum_{v \in V} d_v = 0$

Output: whether there exists $f : E \to \mathbb{Z}_{\geq 0}$ s.t.

$$\sum_{e \in \delta^{\mathsf{out}}(v)} f(e) - \sum_{e \in \delta^{\mathsf{in}}(v)} f(e) = d_v \qquad \forall v \in V$$

$$0 \leq f(e) \leq c_e \qquad \forall e \in E$$

Extension of Network Flow: Circulation Problem

Input: A digraph $G = (V, E)$

capacities $c \in \mathbb{Z}_{\geq 0}^E$

supply vector $d \in \mathbb{Z}^V$ with $\sum_{v \in V} d_v = 0$

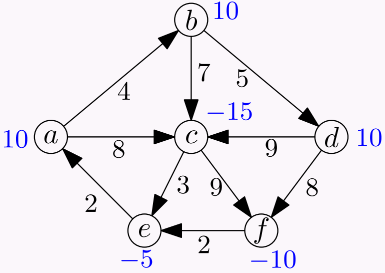
Output: whether there exists $f : E \to \mathbb{Z}_{\geq 0}$ s.t.

$$\sum_{e \in \delta^{\mathsf{out}}(v)} f(e) - \sum_{e \in \delta^{\mathsf{in}}(v)} f(e) = d_v \qquad \forall v \in V$$

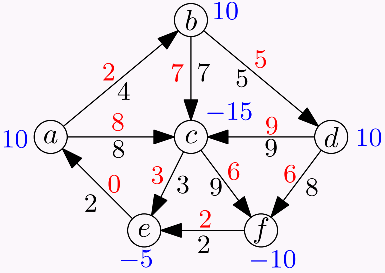
$$0 \leq f(e) \leq c_e \qquad \forall e \in E$$

- d_v denotes the net supply of a good
- $d_v > 0$: there is a supply of d_v at v
- $d_v < 0$: there is a demand of $-d_v$ at v
- problem: whether we can match the supplies and demands without violating capacity constraints

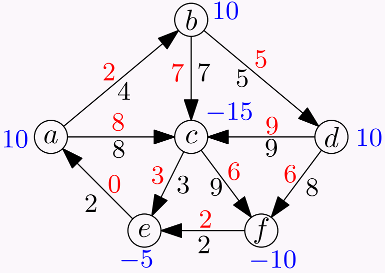
Example



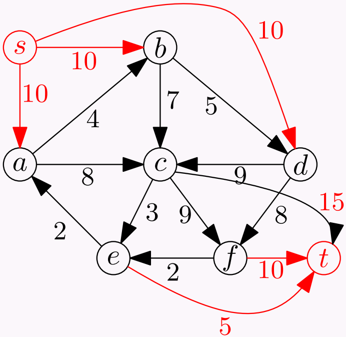
Example



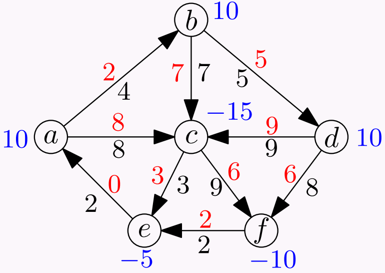
Example



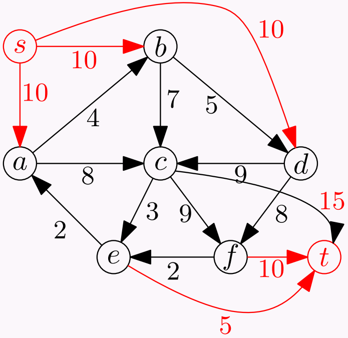
Reduction



Example



Reduction



Reduction to maximum flow

- add a super-source s and a super-sink t to network
- for every $v \in V$ with $d_v > 0$: add edge (s, v) of capacity d_v
- for every $v \in V$ with $d_v < 0$: add edge (v, t) of capacity $-d_v$
- check if maximum flow has value $\sum_{v:d_v>0} d_v$

- $d(S) := \sum_{v \in S} d_v, \forall S \subseteq V$.
- $c(S, V \setminus S) := \sum_{(u,v) \in E: u \in S, v \notin S} c_{(u,v)}$.

Lemma The instance is feasible if and only if for every $S \subseteq V$, $d(S) \le c(S, V \setminus S)$.

- $d(S) := \sum_{v \in S} d_v, \forall S \subseteq V$.
- $c(S, V \setminus S) := \sum_{(u,v) \in E: u \in S, v \notin S} c_{(u,v)}$.

Lemma The instance is feasible if and only if for every $S \subseteq V$, $d(S) \leq c(S, V \setminus S)$.

Proof of "only if" direction.

- if for some $S \subseteq V$, $c(S, V \setminus S) < d(S)$, then the demand in S can not be sent out of S. □

- $d(S) := \sum_{v \in S} d_v, \forall S \subseteq V$.
- $c(S, V \setminus S) := \sum_{(u,v) \in E : u \in S, v \notin S} c_{(u,v)}$.

Lemma The instance is feasible if and only if for every $S \subseteq V$, $d(S) \leq c(S, V \setminus S)$.

Proof of "only if" direction.

- if for some $S \subseteq V$, $c(S, V \setminus S) < d(S)$, then the demand in S can not be sent out of S. □

- It remains to consider the "if" direction

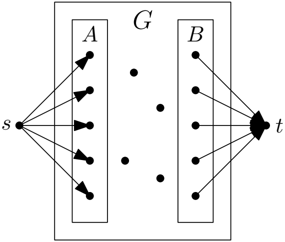
Proof of “if” Direction

Lemma The instance is feasible if and only if for every $S \subseteq V$, $d(S) \leq c(S, V \setminus S)$.

Proof of "if" Direction

Lemma The instance is feasible if and only if for every $S \subseteq V$, $d(S) \leq c(S, V \setminus S)$.

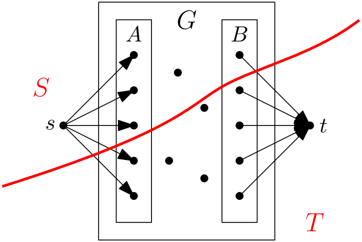
- assume instance is infeasible: max-flow $< d(A)$
- $A := \{v \in V : d_v > 0\}$
- $B := \{v \in V : d_v < 0\}$



Proof of "if" Direction

Lemma The instance is feasible if and only if for every $S \subseteq V$, $d(S) \leq c(S, V \setminus S)$.

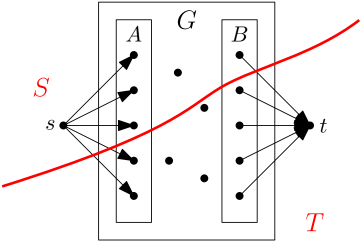
- assume instance is infeasible: max-flow $< d(A)$
- $A := \{v \in V : d_v > 0\}$
- $B := \{v \in V : d_v < 0\}$
- $(S \ni s, T \ni t)$: min-cut



Proof of "if" Direction

Lemma The instance is feasible if and only if for every $S \subseteq V$, $d(S) \leq c(S, V \setminus S)$.

- assume instance is infeasible: max-flow $< d(A)$
- $A := \{v \in V : d_v > 0\}$
- $B := \{v \in V : d_v < 0\}$
- $(S \ni s, T \ni t)$: min-cut

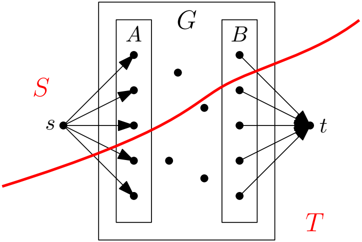


$$d(T \cap A) + |d(S \cap B)| + c(S \setminus \{s\}, T \setminus \{t\}) < d(A)$$
$$d(T \cap A) - d(S \cap B) + c(S \setminus \{s\}, T \setminus \{t\}) < d(A)$$
$$c(S \setminus \{s\}, T \setminus \{t\}) < d(S \cap A) + d(S \cap B) = d(S \setminus \{s\})$$

Proof of "if" Direction

Lemma The instance is feasible if and only if for every $S \subseteq V$, $d(S) \leq c(S, V \setminus S)$.

- assume instance is infeasible: max-flow $< d(A)$
- $A := \{v \in V : d_v > 0\}$
- $B := \{v \in V : d_v < 0\}$
- $(S \ni s, T \ni t)$: min-cut



$$d(T \cap A) + |d(S \cap B)| + c(S \setminus \{s\}, T \setminus \{t\}) < d(A)$$
$$d(T \cap A) - d(S \cap B) + c(S \setminus \{s\}, T \setminus \{t\}) < d(A)$$
$$c(S \setminus \{s\}, T \setminus \{t\}) < d(S \cap A) + d(S \cap B) = d(S \setminus \{s\})$$

- Define $S' = S \setminus \{s\}$: $d(S') > c(S', V \setminus S')$.

Circulation Problem with Capacity Lower Bounds

Input: A digraph $G = (V, E)$

capacities $c \in \mathbb{Z}_{\geq 0}^{E}$

capacity lower bounds $l \in \mathbb{Z}_{\geq 0}^{E}$, $0 \leq l_e \leq c_e$

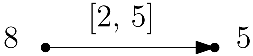
supply vector $d \in \mathbb{Z}^{V}$ with $\sum_{v \in V} d_v = 0$

Output: whether there exists $f : E \to \mathbb{Z}_{\geq 0}$ s.t.

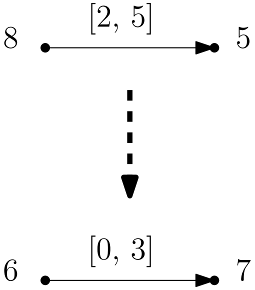
$$\sum_{e \in \delta^{\mathrm{out}}(v)} f(e) - \sum_{e \in \delta^{\mathrm{in}}(v)} f(e) = d_v \qquad \forall v \in V$$

$$l_e \leq f(e) \leq c_e \qquad \forall e \in E$$

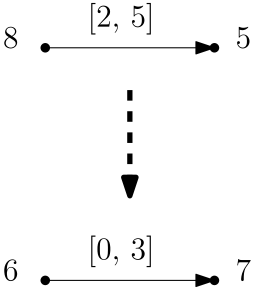
Removing Capacity Lower Bounds



Removing Capacity Lower Bounds



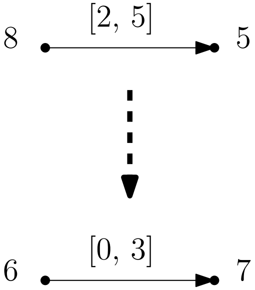
Removing Capacity Lower Bounds



handling $e = (u, v)$ with $l_e > 0$

- $d'_u \leftarrow d_u - l_e$
- $d'_v \leftarrow d_v + l_e$
- $c'_e \leftarrow c_e - l_e$
- $l'_e \leftarrow 0$

Removing Capacity Lower Bounds



handling $e = (u, v)$ with $l_e > 0$

- $d'_u \leftarrow d_u - l_e$
- $d'_v \leftarrow d_v + l_e$
- $c'_e \leftarrow c_e - l_e$
- $l'_e \leftarrow 0$

- in old instance: flow is $f(e) \in [l_e, c_e] \implies f(e) - l_e \in [0, c_e - l_e]$
- in new instance: flow is $f(e) - l_e \in [0, c'_e]$

Survey Design

Input: integers $n, k \geq 1$ and $E \subseteq [n] \times [k]$

integers $0 \leq c_i \leq c'_i, \forall i \in [n]$

integers $0 \leq p_j \leq p'_j, \forall j \in [k]$

Survey Design

Input: integers $n, k \geq 1$ and $E \subseteq [n] \times [k]$

integers $0 \leq c_i \leq c_i', \forall i \in [n]$

integers $0 \leq p_j \leq p_j', \forall j \in [k]$

Output: $E' \subseteq E$ s.t.

$$\begin{aligned} c_i \leq |\{j \in [k] : (i,j) \in E'\}| \leq c_i', & \qquad \forall i \in [n] \\ p_j \leq |\{i \in [m] : (i,j) \in E'\}| \leq p_j', & \qquad \forall j \in [k] \end{aligned}$$

Survey Design

Input: integers $n, k \geq 1$ and $E \subseteq [n] \times [k]$

integers $0 \leq c_i \leq c'_i, \forall i \in [n]$

integers $0 \leq p_j \leq p'_j, \forall j \in [k]$

Output: $E' \subseteq E$ s.t.

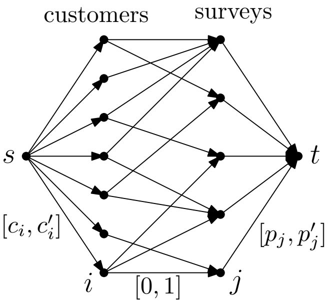
$$\begin{aligned} c_i \leq |\{j \in [k] : (i,j) \in E'\}| &\leq c'_i, & \forall i \in [n] \\ p_j \leq |\{i \in [m] : (i,j) \in E'\}| &\leq p'_j, & \forall j \in [k] \end{aligned}$$

Background

- $[n]$: customers, $[k]$:products
- $ij \in E$: customer i purchased product j and can do a survey
- every customer i needs to do between c_i and c'_i surveys
- every product j needs to collect between p_j and p'_j surveys

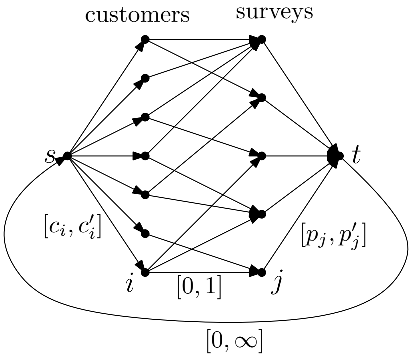
Reduction to Circulation

- vertices $\{s,t\} \uplus [n] \uplus [k]$,
- $(i,j) \in E$: (i,j) with bounds $[0,1]$
- $\forall i$: (s,i) with bounds $[c_i, c'_i]$
- $\forall j$: (j,t) with bounds $[p_j, p'_i]$



Reduction to Circulation

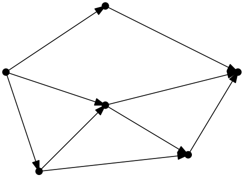
- vertices $\{s, t\} \uplus [n] \uplus [k]$,
- $(i, j) \in E$: (i, j) with bounds $[0, 1]$
- $\forall i$: (s, i) with bounds $[c_i, c_i']$
- $\forall j$: (j, t) with bounds $[p_j, p_i']$
- (t, s) with bounds $[0, \infty]$



Airline Scheduling

Input: a DAG $G = (V, E)$

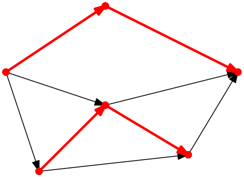
Output: the minimum number of disjoint paths in G to cover all vertices



Airline Scheduling

Input: a DAG $G = (V, E)$

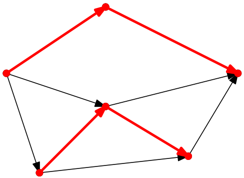
Output: the minimum number of disjoint paths in G to cover all vertices



Airline Scheduling

Input: a DAG $G = (V, E)$

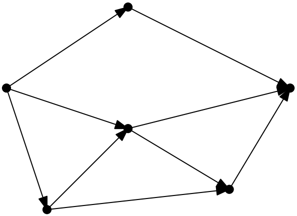
Output: the minimum number of disjoint paths in G to cover all vertices



Background

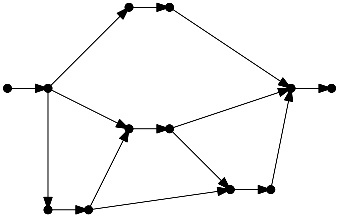
- vertex : a flight
- edge (u, v): an aircraft that serves u can serve v immediately
- goal: minimize the number of aircrafts

Reduction to the Circulation Problem



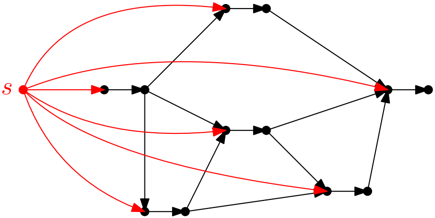
Reduction to the Circulation Problem

- split v into $(v_{\text{in}}, v_{\text{out}})$



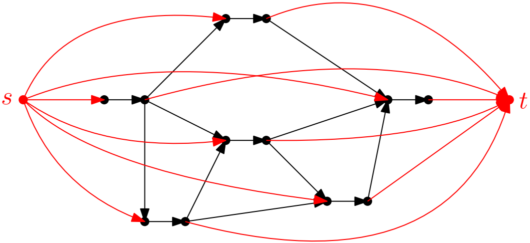
Reduction to the Circulation Problem

- split v into $(v_{\textsf{in}}, v_{\textsf{out}})$
- add s, and $(s, v_{\textsf{in}}), \forall v$



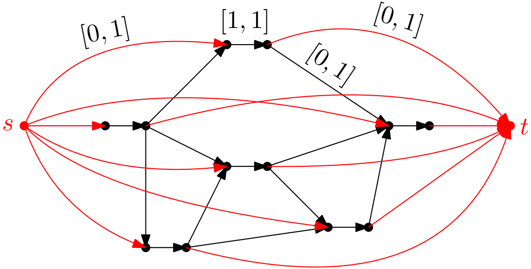
Reduction to the Circulation Problem

- split v into $(v_{\text{in}}, v_{\text{out}})$
- add s, and $(s, v_{\text{in}}), \forall v$
- add t, and $(v_{\text{out}}, t), \forall v$



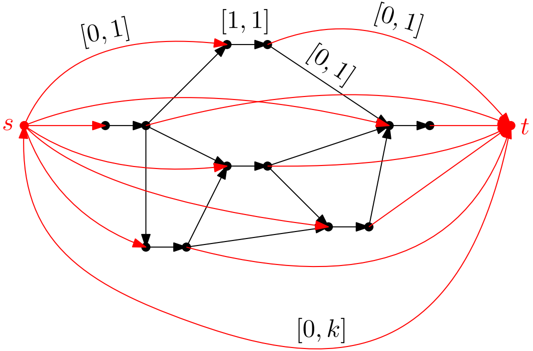
Reduction to the Circulation Problem

- split v into $(v_{\text{in}}, v_{\text{out}})$
- add s, and $(s, v_{\text{in}}), \forall v$
- add t, and $(v_{\text{out}}, t), \forall v$
- set lower and upper bounds



Reduction to the Circulation Problem

- split v into $(v_{\mathsf{in}}, v_{\mathsf{out}})$
- add s, and $(s, v_{\mathsf{in}}), \forall v$
- add t, and $(v_{\mathsf{out}}, t), \forall v$
- set lower and upper bounds
- add $t \rightarrow s$ of capacity k



Reduction to the Circulation Problem

- split v into $(v_{\text{in}}, v_{\text{out}})$
- add s, and $(s, v_{\text{in}}), \forall v$
- add t, and $(v_{\text{out}}, t), \forall v$
- set lower and upper bounds
- add $t \to s$ of capacity k
- find minimum k s.t. instance is feasible

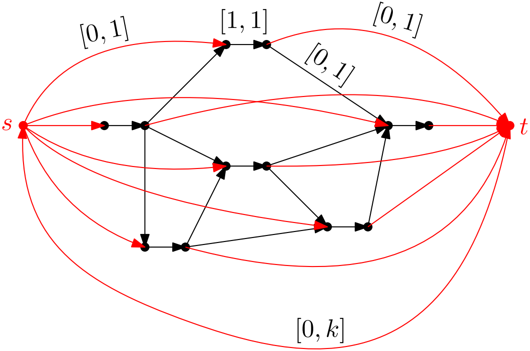


Image Segmentation

Input: A graph $G = (V, E)$, with edge costs $c \in \mathbb{Z}^E_{\geq 0}$
two reward vectors $a, b \in \mathbb{Z}^V_{\geq 0}$

Image Segmentation

Input: A graph $G = (V, E)$, with edge costs $c \in \mathbb{Z}_{\geq 0}^E$
two reward vectors $a, b \in \mathbb{Z}_{\geq 0}^V$

Output: a cut (A, B) of G so as to maximize

$$\sum_{v \in A} a_v + \sum_{v \in B} b_v - \sum_{(u,v) \in E : |\{u,v\} \cap A| = 1} c_{(u,v)}$$

Image Segmentation

Input: A graph $G = (V, E)$, with edge costs $c \in \mathbb{Z}_{\geq 0}^{E}$
two reward vectors $a, b \in \mathbb{Z}_{\geq 0}^{V}$

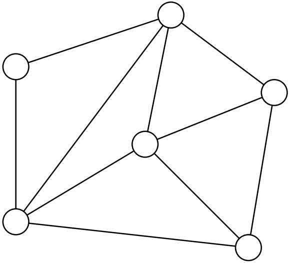
Output: a cut (A, B) of G so as to maximize

$$\sum_{v \in A} a_v + \sum_{v \in B} b_v - \sum_{(u,v) \in E : |\{u,v\} \cap A| = 1} c_{(u,v)}$$

Background

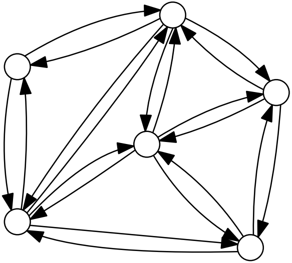
- a_v: the likelihood of v being a foreground pixel
- b_v: the likelihood of v being a background pixel
- $c_{(u,v)}$: the penalty for separating u and v
- need to maximize total reward - total penalty

Reduction to Network Flow



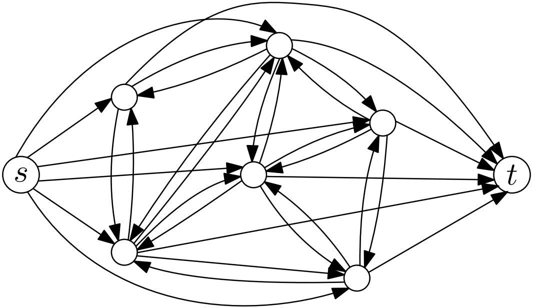
Reduction to Network Flow

- replace (u, v) with two anti-parallel arcs



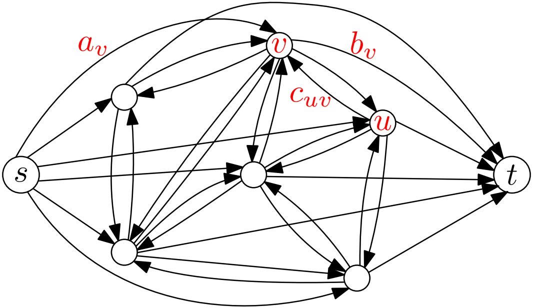
Reduction to Network Flow

- replace (u, v) with two anti-parallel arcs
- add source s and arcs $(s, v), \forall v$
- add sink t and arcs $(v, t), \forall v$



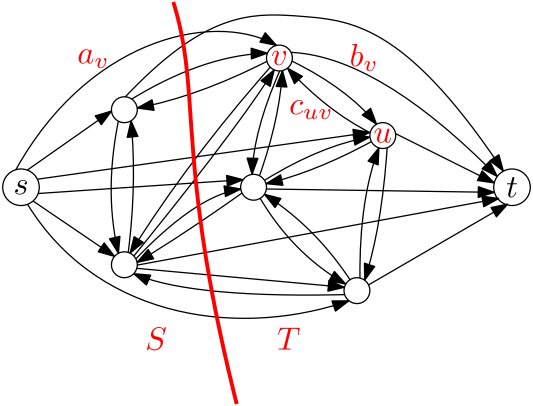
Reduction to Network Flow

- replace (u, v) with two anti-parallel arcs
- add source s and arcs $(s, v), \forall v$
- add sink t and arcs $(v, t), \forall v$
- set capacities



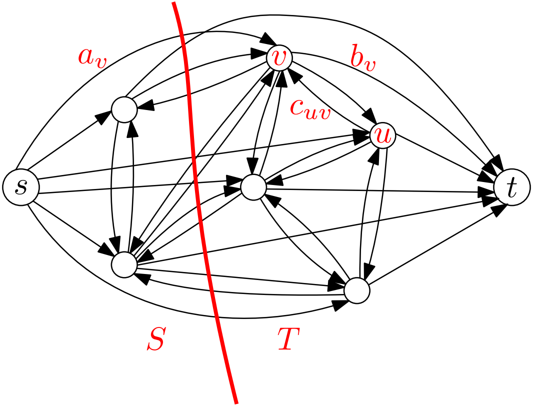
Reduction to Network Flow

- replace (u, v) with two anti-parallel arcs
- add source s and arcs $(s, v), \forall v$
- add sink t and arcs $(v, t), \forall v$
- set capacities



Reduction to Network Flow

- replace (u, v) with two anti-parallel arcs
- add source s and arcs $(s, v), \forall v$
- add sink t and arcs $(v, t), \forall v$
- set capacities



- The cut value of $(S = \{s\} \cup A, \{t\} \cup B)$ is

$$\sum_{v\in B} a_v + \sum_{v\in A} b_v + \sum_{(u,v)\in E:|\{u,v\}\cap A|=1} c_{(u,v)}$$

$$= \sum_{v\in V}(a_v + b_v) - \Bigg(\sum_{v\in A} a_v + \sum_{v\in B} b_v - \sum_{(u,v)\in E:|\{u,v\}\cap A|=1} c_{(u,v)}\Bigg)$$

- The cut value of $(S = \{s\} \cup A, \{t\} \cup B)$ is

$$\sum_{v \in V}(a_v + b_v) - \Big(\sum_{v \in A} a_v + \sum_{v \in B} b_v - \sum_{(u,v) \in E : |\{u,v\} \cap A| = 1} c_{(u,v)} \Big)$$

$$= \sum_{v \in V}(a_v + b_v) - \big(\text{objective of } (A, B)\big)$$

- The cut value of $(S = \{s\} \cup A, \{t\} \cup B)$ is

$$\begin{aligned}
&\sum_{v \in V}(a_v + b_v) - \Big(\sum_{v \in A} a_v + \sum_{v \in B} b_v - \sum_{(u,v) \in E : |\{u,v\} \cap A| = 1} c_{(u,v)} \Big) \\
= &\sum_{v \in V}(a_v + b_v) - \big(\text{objective of } (A, B)\big)
\end{aligned}$$

- So, maximizing the objective of (A, B) is equivalent to minimizing the cut value.

Project Selection

Input: A DAG $G = (V, E)$

revenue on vertices: $p \in \mathbb{Z}^V$; p_v's could be negative.

Project Selection

Input: A DAG $G = (V, E)$

revenue on vertices: $p \in \mathbb{Z}^V$; p_v's could be negative.

Output: A set $B \subseteq V$ satisfying the precedence constraints:

$$v \in B \implies u \in B, \quad \forall (u, v) \in E$$

Project Selection

Input: A DAG $G = (V, E)$

revenue on vertices: $p \in \mathbb{Z}^V$; p_v's could be negative.

Output: A set $B \subseteq V$ satisfying the precedence constraints:

$$v \in B \implies u \in B, \quad \forall (u, v) \in E$$

Motivation

- Motivation: $(u, v) \in E$: u is a prerequisite of v, to select v, we must select u
- Goal: maximize the revenue subject to the precedence constraint.

Project Selection

Input: A DAG $G = (V, E)$

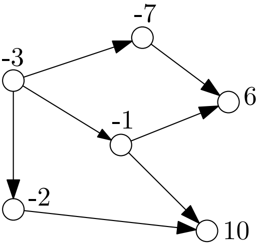
revenue on vertices: $p \in \mathbb{Z}^V$; p_v's could be negative.

Output: A set $B \subseteq V$ satisfying the precedence constraints:

$$v \in B \implies u \in B, \quad \forall (u, v) \in E$$

Motivation

- Motivation: $(u, v) \in E$: u is a prerequisite of v, to select v, we must select u
- Goal: maximize the revenue subject to the precedence constraint.



Project Selection

Input: A DAG $G = (V, E)$

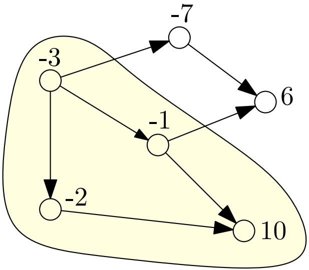
revenue on vertices: $p \in \mathbb{Z}^V$; p_v's could be negative.

Output: A set $B \subseteq V$ satisfying the precedence constraints:

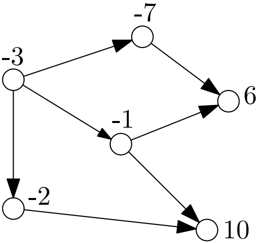
$$v \in B \implies u \in B, \quad \forall (u, v) \in E$$

Motivation

- Motivation: $(u, v) \in E$: u is a prerequisite of v, to select v, we must select u
- Goal: maximize the revenue subject to the precedence constraint.

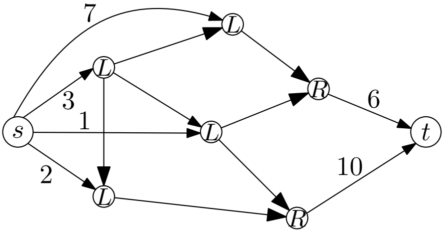


Reduction



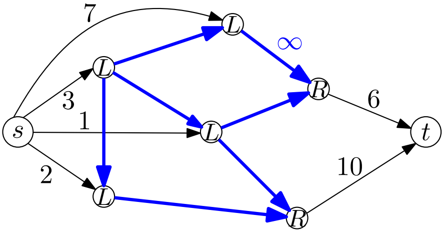
Reduction

- add source s and sink t
- $p_v < 0$: (s, v) of capacity $-p_v$
- $p_v > 0$: (v, t) of capacity p_v
- $L = \{v : p_v < 0\}$
- $R = \{v : p_v > 0\}$.



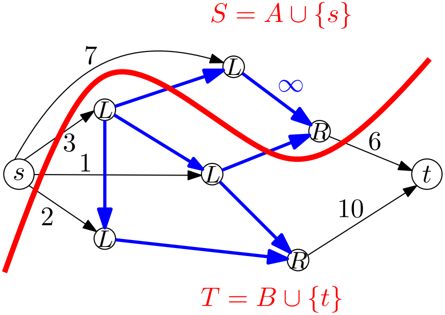
Reduction

- add source s and sink t
- $p_v < 0$: (s, v) of capacity $-p_v$
- $p_v > 0$: (v, t) of capacity p_v
- $L = \{v : p_v < 0\}$
- $R = \{v : p_v > 0\}$.
- precedence edges: ∞ capacity



Reduction

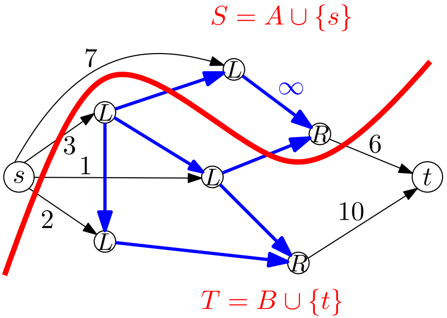
- add source s and sink t
- $p_v < 0$: (s, v) of capacity $-p_v$
- $p_v > 0$: (v, t) of capacity p_v
- $L = \{v : p_v < 0\}$
- $R = \{v : p_v > 0\}$.
- precedence edges: ∞ capacity



- min-cut $(S = \{s\} \cup A, T = \{t\} \cup B)$

Reduction

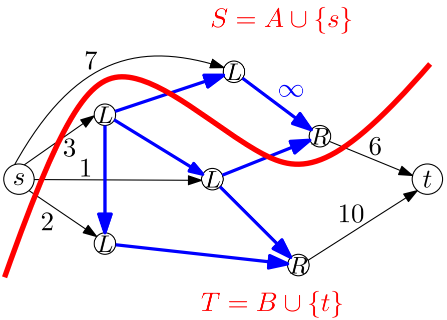
- add source s and sink t
- $p_v < 0$: (s, v) of capacity $-p_v$
- $p_v > 0$: (v, t) of capacity p_v
- $L = \{v : p_v < 0\}$
- $R = \{v : p_v > 0\}$.
- precedence edges: ∞ capacity



- min-cut $(S = \{s\} \cup A, T = \{t\} \cup B)$
- no ∞-capacity edges from A to B

Reduction

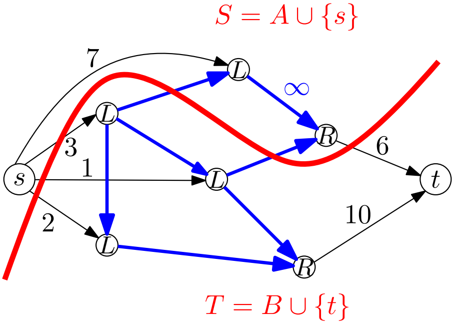
- add source s and sink t
- $p_v < 0$: (s, v) of capacity $-p_v$
- $p_v > 0$: (v, t) of capacity p_v
- $L = \{v : p_v < 0\}$
- $R = \{v : p_v > 0\}$.
- precedence edges: ∞ capacity



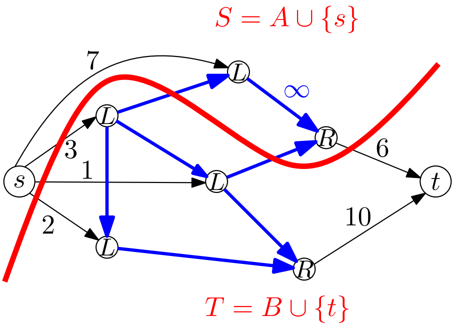
- min-cut $(S = \{s\} \cup A, T = \{t\} \cup B)$
- no ∞-capacity edges from A to B
- cut value is

$$\sum_{v \in B \cap L} (-p_v) + \sum_{v \in A \cap R} p_v = -\sum_{v \in B \cap L} p_v - \sum_{v \in B \cap R} p_v + \sum_{v \in R} p_v$$

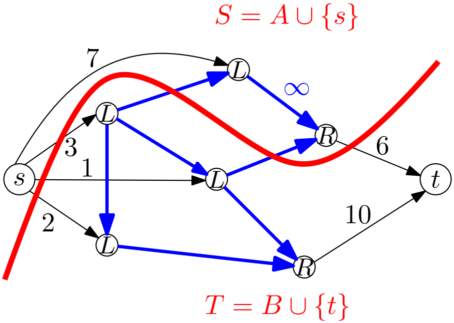
$$= \sum_{v \in R} p_v - \sum_{v \in B} p_v$$



- B is a valid solution $\iff c(S,T) \neq \infty$



- B is a valid solution $\iff c(S,T) \neq \infty$
- when B is valid, $c(S,T) = \sum_{v \in R} p_v - \sum_{v \in B} p_v$



- B is a valid solution $\iff c(S,T) \neq \infty$
- when B is valid, $c(S,T) = \sum_{v \in R} p_v - \sum_{v \in B} p_v$
- so, to maximize $\sum_{v \in B} p_v$, we need to minimize $c(S,T)$.

More Applications

- Graph orientation
- maximum independent set (and minimum vertex cover) in a bipartite graph
- $\cdots$