COL 351 : Analysis & Design of Algorithms

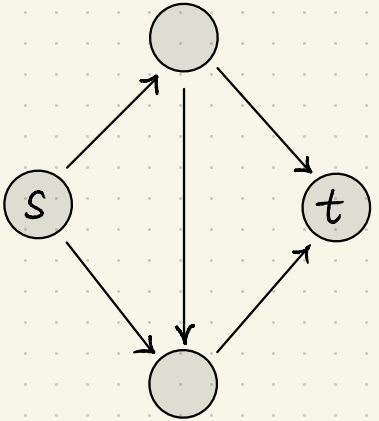
LECTURE 29

NETWORK FLOW I :

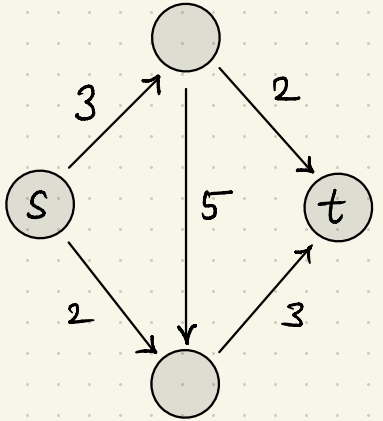
INTRODUCTION TO NETWORK FLOW

OCT 16, 2026 | ROHIT VAISH

Maximum Flow Problem

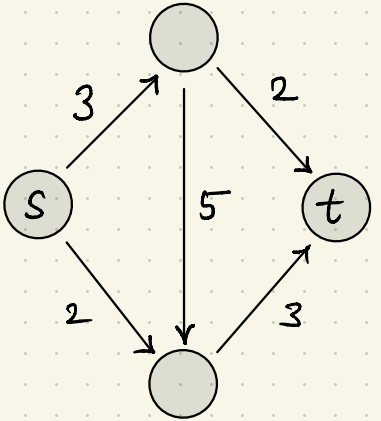


Maximum Flow Problem



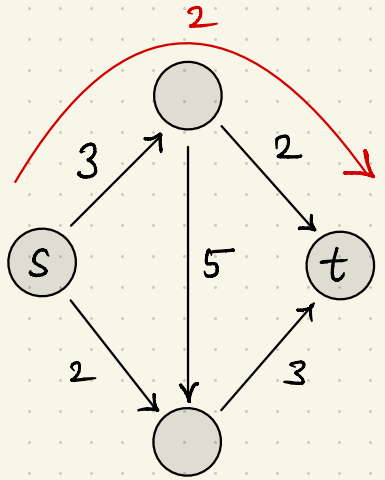
Capacity = maximum amount of stuff an edge can carry

Maximum Flow Problem



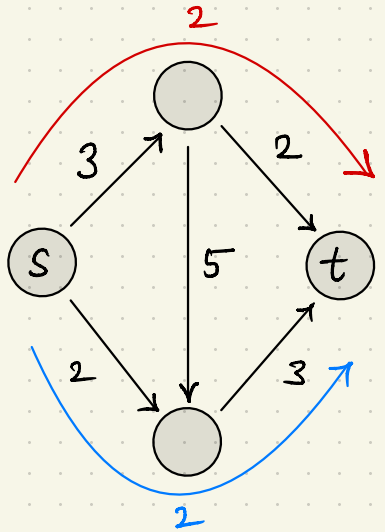
How much stuff can be pushed from s to t ?

Maximum Flow Problem



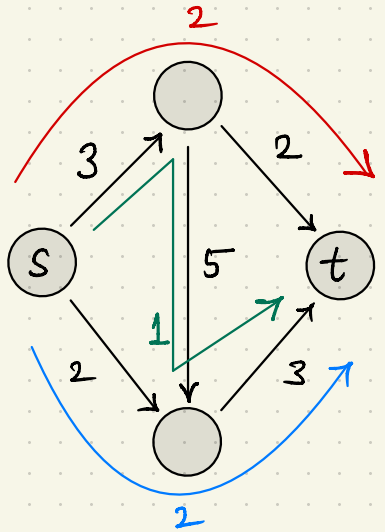
How much stuff can be pushed from s to t?

Maximum Flow Problem



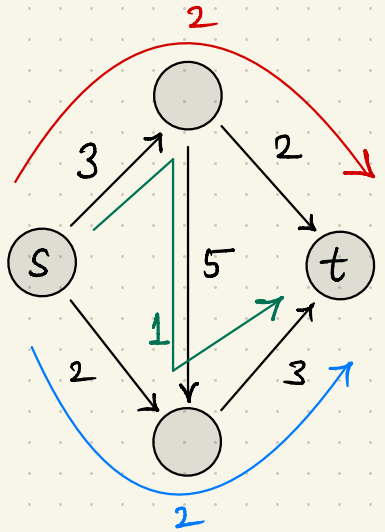
How much stuff can be pushed from s to t ?

Maximum Flow Problem



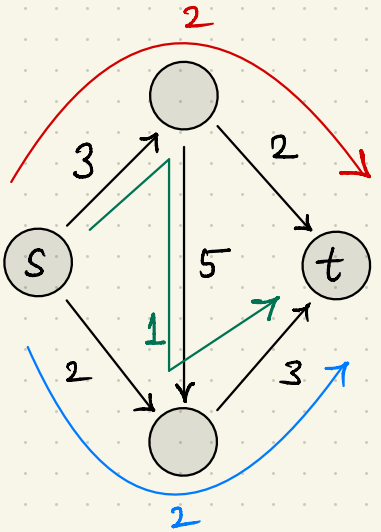
How much stuff can be pushed from s to t ?

Maximum Flow Problem



How much stuff can be pushed from s to t? 5 units.

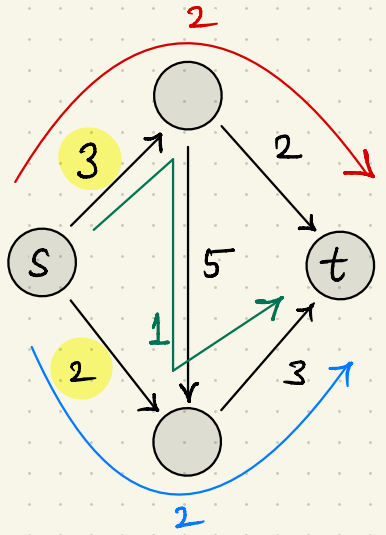
Maximum Flow Problem



How much stuff can be pushed from s to t? 5 units.

Can we do better?

Maximum Flow Problem



How much stuff can be pushed from s to t? 5 units.

Can we do better? NO!

Maximum Flow Problem

an instance of max flow problem is defined by:

Maximum Flow Problem

an instance of max flow problem is defined by:

a directed graph $G = (V, E)$

Maximum Flow Problem

an instance of max flow problem is defined by:

- a directed graph $G = (V, E)$
- a source vertex $s \in V$
- a sink vertex $t \in V$

Maximum Flow Problem

an instance of max flow problem is defined by:

- a directed graph $G = (V, E)$
- a source vertex $s \in V$
- a sink vertex $t \in V$
- a nonnegative and integral capacity u_e for each edge $e \in E$

Maximum Flow Problem

an instance of max flow problem is defined by:

- a directed graph $G = (V, E)$
- a source vertex $s \in V$
- a sink vertex $t \in V$
- a nonnegative and integral capacity u_e for each edge $e \in E$

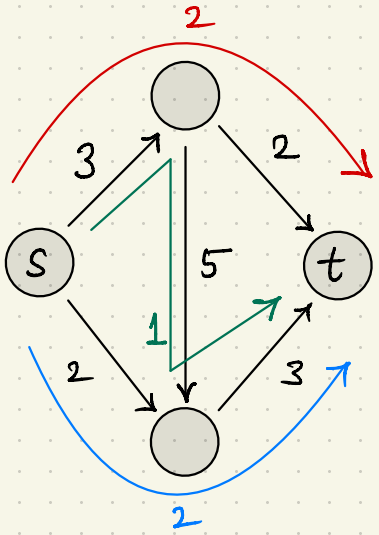
Why not c_e?

Maximum Flow Problem

Feasible Solutions : Flows

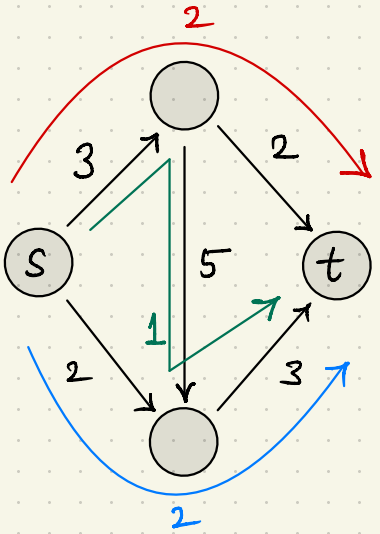
Maximum Flow Problem

Feasible Solutions : Flows

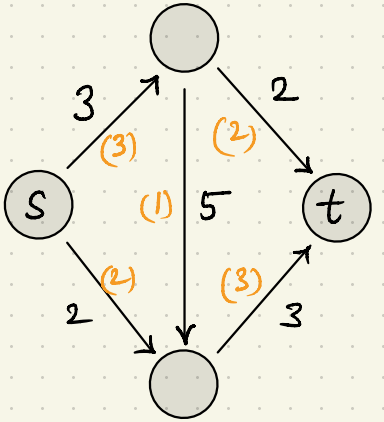


Maximum Flow Problem

Feasible Solutions : Flows



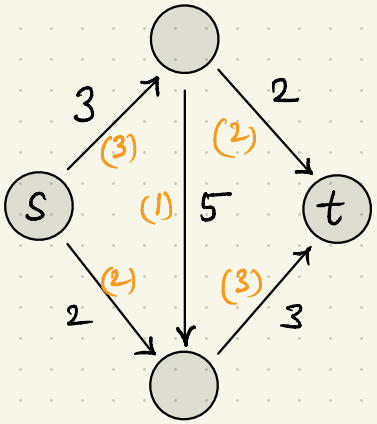
≡



Superposition of paths

Maximum Flow Problem

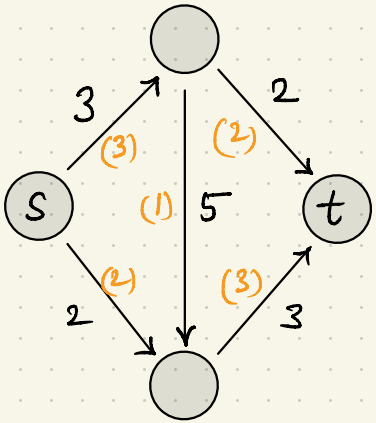
Feasible Solutions : Flows



Maximum Flow Problem

Feasible Solutions : Flows

A nonnegative vector $\{f_e\}_{e \in E}$

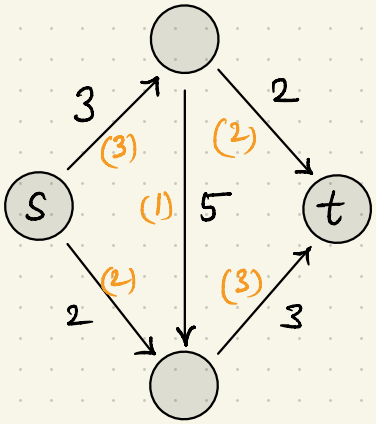


Maximum Flow Problem

Feasible Solutions : Flows

A nonnegative vector $\{f_e\}_{e \in E}$ satisfying

1. Capacity constraints



2. Conservation constraints

Maximum Flow Problem

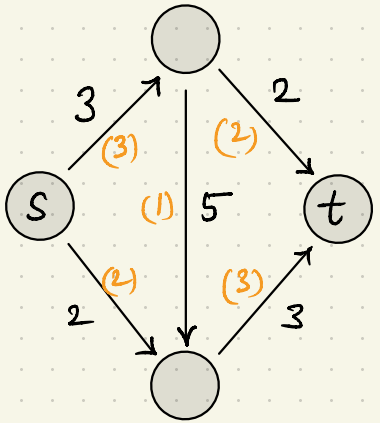
Feasible Solutions : Flows

A nonnegative vector $\{f_e\}_{e \in E}$ satisfying

1. Capacity constraints

$$f_e \leq u_e \quad \forall e \in E$$

2. Conservation constraints



Maximum Flow Problem

Feasible Solutions : Flows

A nonnegative vector $\{f_e\}_{e \in E}$ satisfying

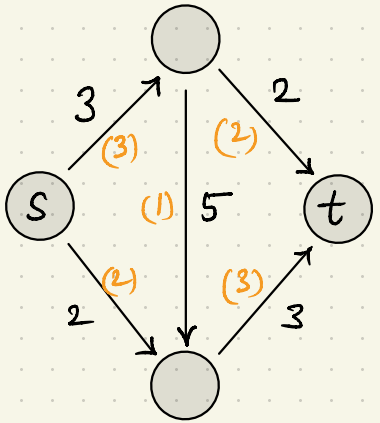
1. Capacity constraints

$$f_e \leq u_e \quad \forall e \in E$$

2. Conservation constraints

$$\text{flow in} = \text{flow out} \quad \forall v \neq s, t$$





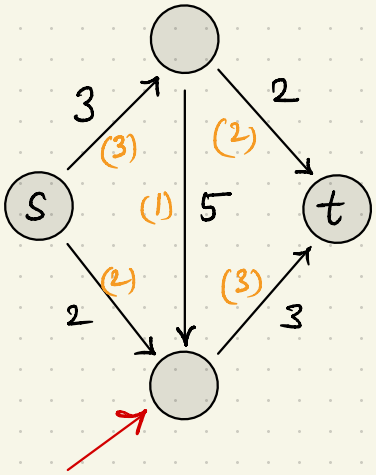
Maximum Flow Problem

Feasible Solutions : Flows

A nonnegative vector $\{f_e\}_{e \in E}$ satisfying

1. Capacity constraints

$$f_e \leq u_e \quad \forall e \in E$$



2. Conservation constraints

$$\text{flow in} = \text{flow out} \quad \forall v \neq s, t$$



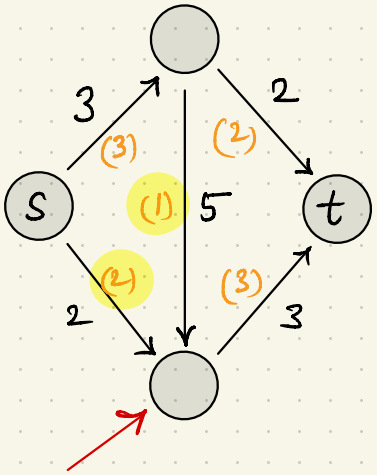
Maximum Flow Problem

Feasible Solutions : Flows

A nonnegative vector $\{f_e\}_{e \in E}$ satisfying

1. Capacity constraints

$$f_e \leq u_e \quad \forall e \in E$$



2. Conservation constraints

$$\text{flow in} = \text{flow out} \quad \forall v \neq s, t$$



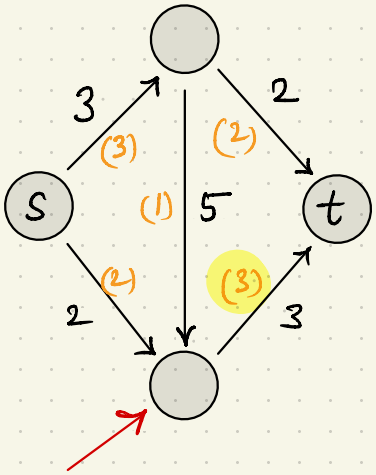
Maximum Flow Problem

Feasible Solutions : Flows

A nonnegative vector $\{f_e\}_{e \in E}$ satisfying

1. Capacity constraints

$$f_e \leq u_e \quad \forall e \in E$$



2. Conservation constraints

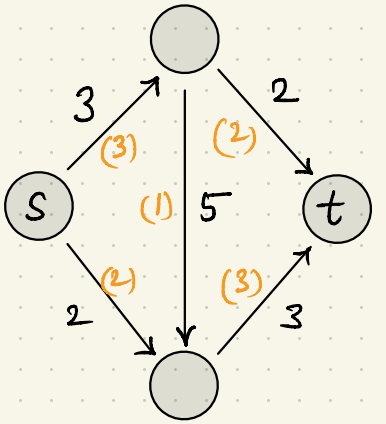
$$\text{flow in} = \text{flow out} \quad \forall v \neq s, t$$



Maximum Flow Problem

Feasible Solutions : Flows

A nonnegative vector $\{f_e\}_{e \in E}$ satisfying



1. Capacity constraints

$$f_e \leq u_e \quad \forall e \in E$$

2. Conservation constraints

$$\text{flow in} = \text{flow out} \quad \forall v \neq s, t$$



$f_e = 0 \ \forall e$ is feasible

Maximum Flow Problem

Claim: Flow out of s = Flow into t

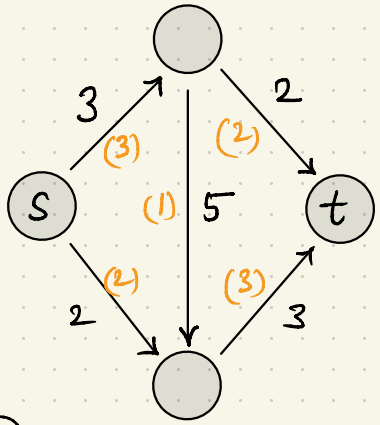
Maximum Flow Problem

Claim: Flow out of s = Flow into t

Proof: Define Type I edges : (s, v) for $v \notin \{s, t\}$

Type II edges : (v, t) for $v \notin \{s, t\}$

Type III edges : (u, v) for $u, v \notin \{s, t\}$



$$\sum_{v \neq s,t} \text{flow-in}(v) = \sum_{\text{Type I}} f_e + \sum_{\text{Type III}} f_e \quad \text{——} \quad (1)$$

$$\sum_{v \neq s,t} \text{flow-out}(v) = \sum_{\text{Type II}} f_e + \sum_{\text{Type III}} f_e \quad \text{——} \quad (2)$$

Capacity constraints $\Rightarrow$ (1) = (2) $\Rightarrow$ flow out of s = flow into t.

(Type I) (Type II)

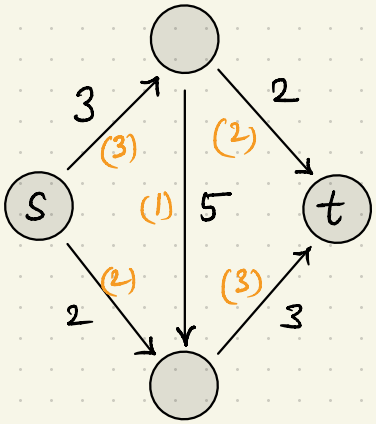
∎

Maximum Flow Problem

Feasible Solutions : Flows

A nonnegative vector $\{f_e\}_{e \in E}$ satisfying

1. Capacity constraints
2. Conservation constraints

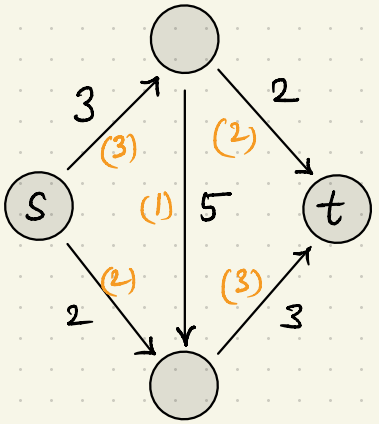


Maximum Flow Problem

Feasible Solutions : Flows

A nonnegative vector $\{f_e\}_{e \in E}$ satisfying

1. Capacity constraints
2. Conservation constraints



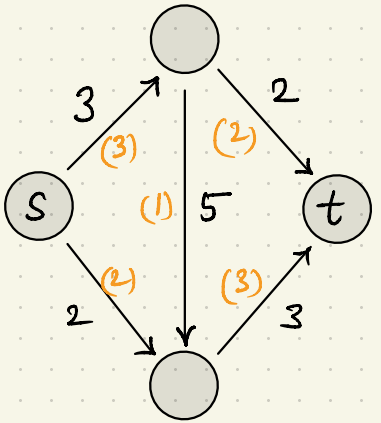
Objective : maximize flow value := flow going out of s

Maximum Flow Problem

Feasible Solutions : Flows

A nonnegative vector $\{f_e\}_{e \in E}$ satisfying

1. Capacity constraints
2. Conservation constraints



Objective : maximize flow value := flow going out of s

Assume : no edges going into s or coming out of t. (WOLOG)

ALGORITHMS

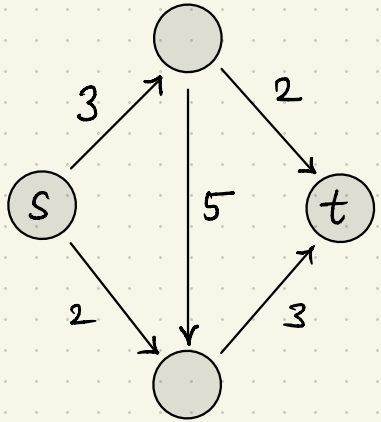
Idea #1 : Naive Greedy Approach

Idea #1: Naive Greedy Approach

Compute a sequence of flows with progressively higher values

IDEA # 1 : NAIVE GREEDY APPROACH

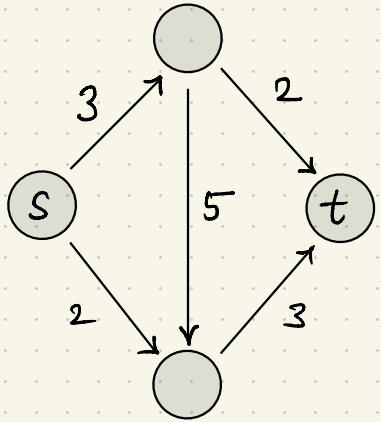
Compute a sequence of flows with progressively higher values



IDEA # 1 : NAIVE GREEDY APPROACH

Compute a sequence of flows with progressively higher values

① initialize $f_e = 0 \quad \forall e$

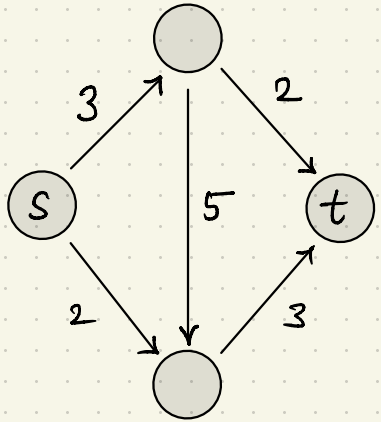


IDEA # 1 : NAIVE GREEDY APPROACH

Compute a sequence of flows with progressively higher values

① initialize $f_e = 0 \quad \forall e$

② repeat :



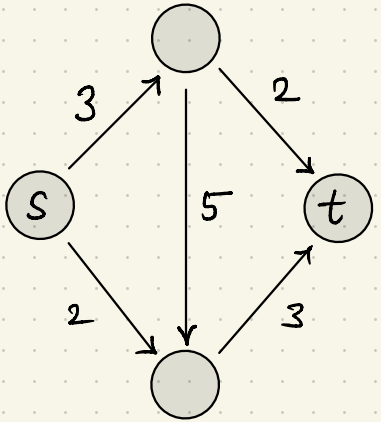
IDEA # 1 : NAIVE GREEDY APPROACH

Compute a sequence of flows with progressively higher values

① initialize $f_e = 0 \quad \forall e$

② repeat :

* search for an $s-t$ path P such that $f_e < u_e \; \forall e \in P$.



IDEA # 1 : NAIVE GREEDY APPROACH

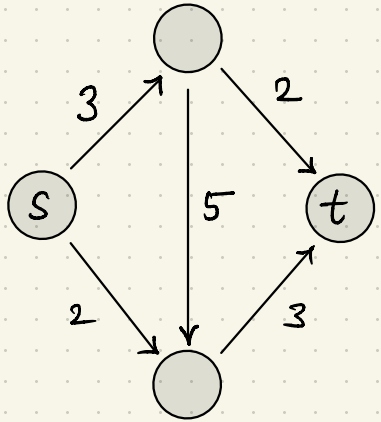
Compute a sequence of flows with progressively higher values

① initialize $f_e = 0 \quad \forall e$

② repeat:

* search for an $s-t$ path P such that $f_e < u_e \ \forall e \in P$.

// possible in $O(|E|)$ time via BFS / DFS



IDEA # 1 : NAIVE GREEDY APPROACH

Compute a sequence of flows with progressively higher values

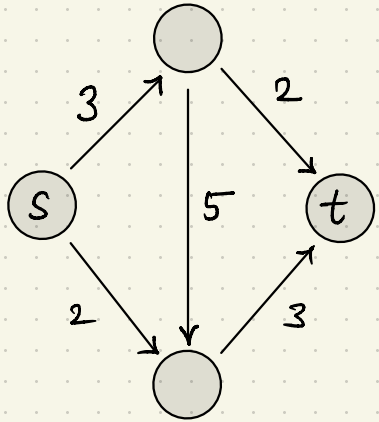
① initialize $f_e = 0 \quad \forall e$

② repeat:

* search for an $s-t$ path P such that $f_e < u_e \ \forall e \in P$.

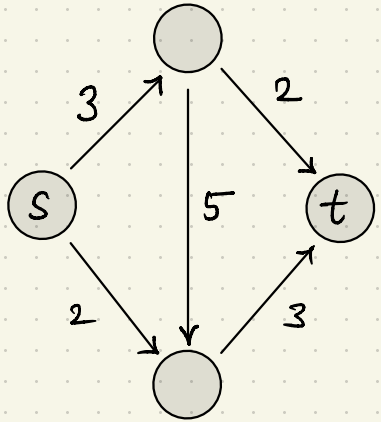
// possible in $O(|E|)$ time via BFS / DFS

* if no such path, return current flow $\{f_e\}_{e \in E}$



IDEA # 1 : NAIVE GREEDY APPROACH

Compute a sequence of flows with progressively higher values



① initialize $f_e = 0 \quad \forall e$

② repeat :

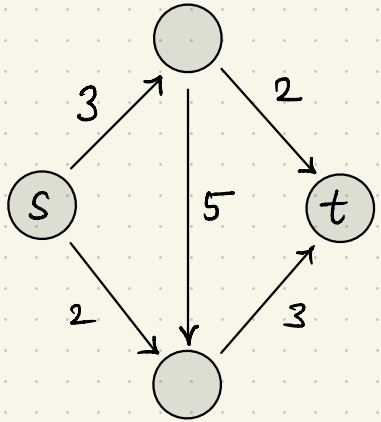
* search for an $s-t$ path P such that $f_e < u_e \ \forall e \in P$.

// possible in $O(|E|)$ time via BFS / DFS

* if no such path, return current flow $\{f_e\}_{e \in E}$
* else,

IDEA # 1 : NAIVE GREEDY APPROACH

Compute a sequence of flows with progressively higher values



① initialize $f_e = 0 \quad \forall e$

② repeat:

* search for an $s-t$ path P such that $f_e < u_e \ \forall e \in P$.

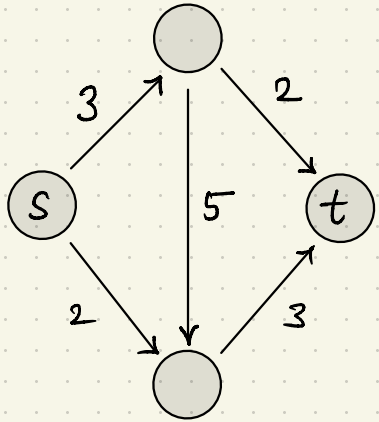
// possible in $O(|E|)$ time via BFS / DFS

* if no such path, return current flow $\{f_e\}_{e \in E}$
* else, let $\Delta := \min_{e \in P} (u_e - f_e)$

└

IDEA # 1 : NAIVE GREEDY APPROACH

Compute a sequence of flows with progressively higher values



① initialize $f_e = 0 \quad \forall e$

② repeat:

* search for an $s-t$ path P such that $f_e < u_e \ \forall e \in P$.

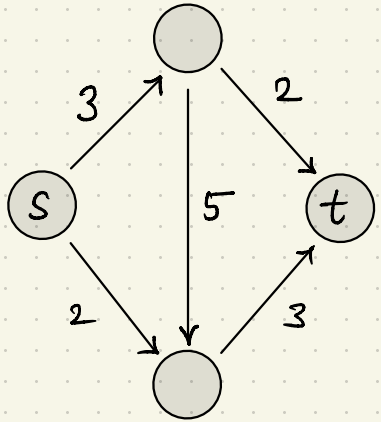
// possible in $O(|E|)$ time via BFS / DFS

* if no such path, return current flow $\{f_e\}_{e \in E}$
* else, let $\Delta := \min_{e \in P} \underbrace{(u_e - f_e)}_{\text{room on } e}$

└

IDEA # 1 : NAIVE GREEDY APPROACH

Compute a sequence of flows with progressively higher values



① initialize $f_e = 0 \quad \forall e$

② repeat:

* search for an $s-t$ path P such that $f_e < u_e \ \forall e \in P$.

// possible in $O(|E|)$ time via BFS / DFS

* if no such path, return current flow $\{f_e\}_{e \in E}$

* else, let $\Delta := \min_{e \in P} (u_e - f_e)$

$(u_e - f_e)$: room on e

$\min_{e \in P}$: room on P

└

Idea #1 : Naive Greedy Approach

Compute a sequence of flows with progressively higher values

① initialize $f_e = 0 \quad \forall e$

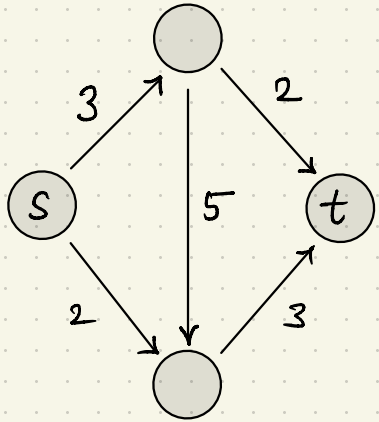
② repeat:

* search for an $s-t$ path P such that $f_e < u_e \ \forall e \in P$.

// possible in $O(|E|)$ time via BFS / DFS

* if no such path, return current flow $\{f_e\}_{e \in E}$
* else, let $\Delta := \min_{e \in P} (u_e - f_e)$

  for all edges $e \in P$, increase f_e by Δ.



IDEA # 1 : NAIVE GREEDY APPROACH

Compute a sequence of flows with progressively higher values

① initialize $f_e = 0 \quad \forall e$

② repeat:

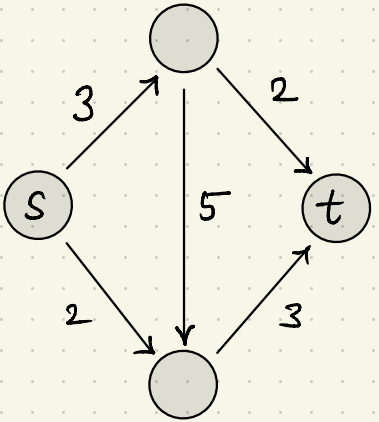
* search for an $s-t$ path P such that $f_e < u_e \ \forall e \in P$.

// possible in $O(|E|)$ time via BFS / DFS

* if no such path, return current flow $\{f_e\}_{e \in E}$

* else, let $\Delta := \min_{e \in P} (u_e - f_e)$ ——— maintains capacity and conservation constraints

  for all edges $e \in P$, increase f_e by Δ.



IDEA #1 : NAIVE GREEDY APPROACH

Compute a sequence of flows with progressively higher values

① initialize $f_e = 0 \quad \forall e$

② repeat:

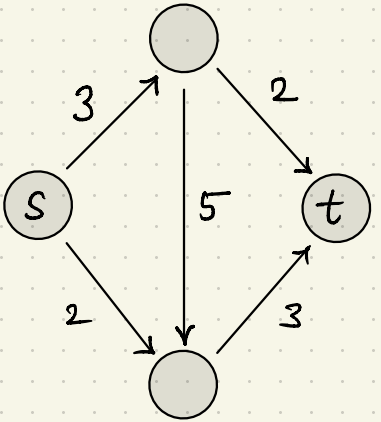
* search for an $s-t$ path P such that $f_e < u_e \ \forall e \in P$.

// possible in $O(|E|)$ time via BFS / DFS

* if no such path, return current flow $\{f_e\}_{e \in E}$

* else, let $\Delta := \min_{e \in P} (u_e - f_e)$

  for all edges $e \in P$, increase f_e by Δ.



Is the output a max flow?

Idea #1 : Naive Greedy Approach

Compute a sequence of flows with progressively higher values

① initialize $f_e = 0 \quad \forall e$

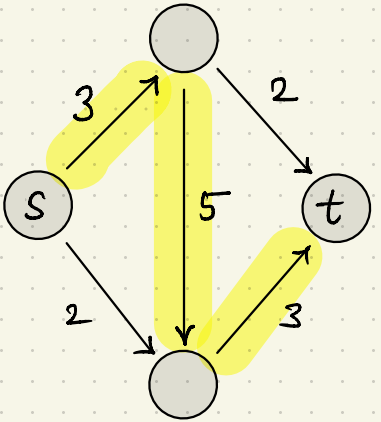
② repeat:

* search for an s–t path P such that $f_e < u_e \ \forall e \in P$.

// possible in $O(|E|)$ time via BFS / DFS

* if no such path, return current flow $\{f_e\}_{e \in E}$
* else, let $\Delta := \min_{e \in P} (u_e - f_e)$

  for all edges $e \in P$, increase f_e by Δ.



Is the output a max flow?

IDEA # 1 : NAIVE GREEDY APPROACH

Compute a sequence of flows with progressively higher values

① initialize $f_e = 0 \quad \forall e$

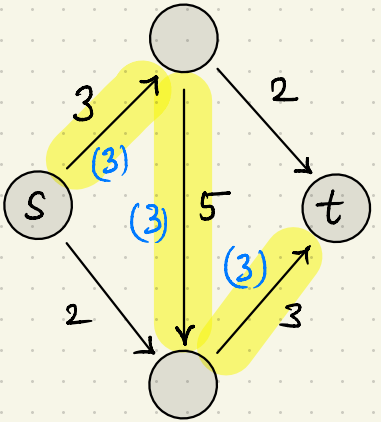
② repeat:

* search for an $s-t$ path P such that $f_e < u_e \ \forall e \in P$.

// possible in $O(|E|)$ time via BFS / DFS

* if no such path, return current flow $\{f_e\}_{e \in E}$
* else, let $\Delta := \min_{e \in P} (u_e - f_e)$

  for all edges $e \in P$, increase f_e by Δ.



Is the output a max flow?

IDEA # 1 : NAIVE GREEDY APPROACH

Compute a sequence of flows with progressively higher values

① initialize $f_e = 0 \quad \forall e$

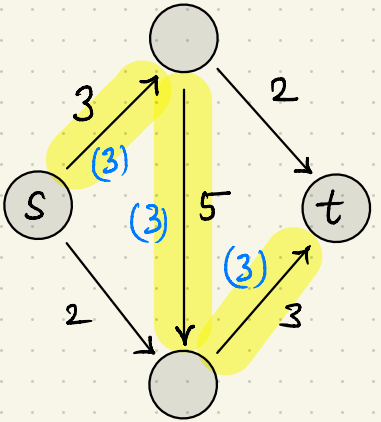
② repeat:

* search for an $s-t$ path P such that $f_e < u_e \ \forall e \in P$.

// possible in $O(|E|)$ time via BFS / DFS

* if no such path, return current flow $\{f_e\}_{e \in E}$
* else, let $\Delta := \min_{e \in P} (u_e - f_e)$

  for all edges $e \in P$, increase f_e by Δ.

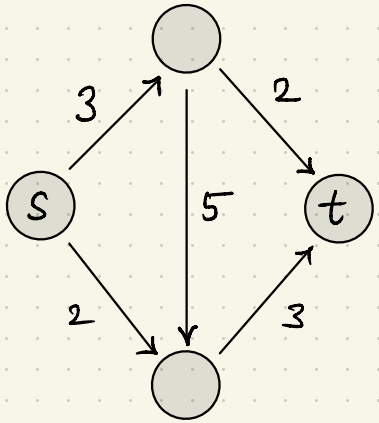


Is the output a max flow?

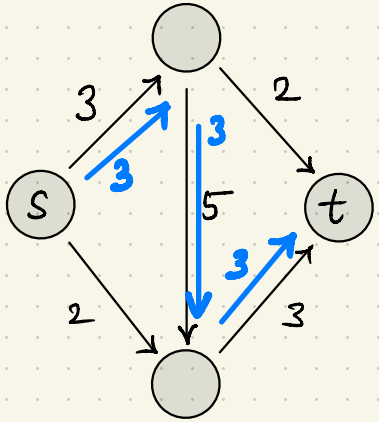
NO!

Idea #2: Naive Greedy + "undo"

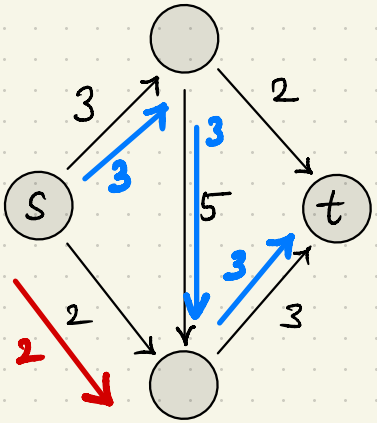
Idea #2: Naive Greedy + "undo"



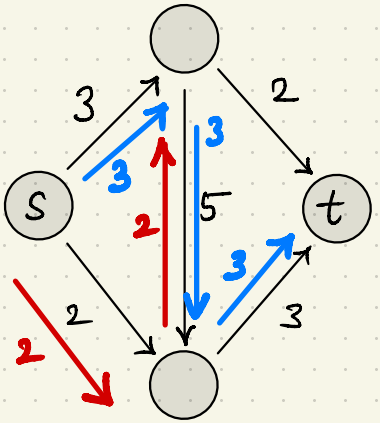
IDEA #2 : NAIVE GREEDY + "UNDO"



IDEA #2 : NAIVE GREEDY + "UNDO"

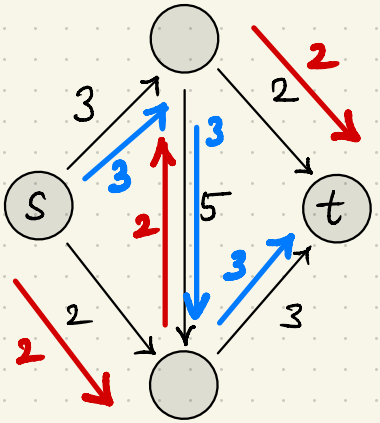


IDEA #2 : NAIVE GREEDY + "UNDO"

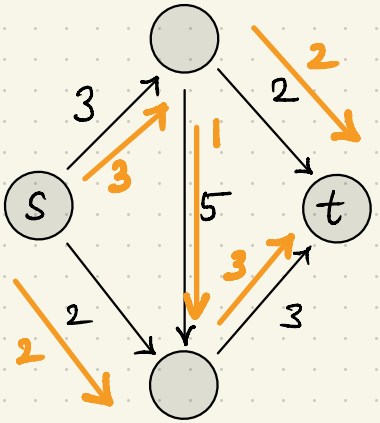


Undo previous flow

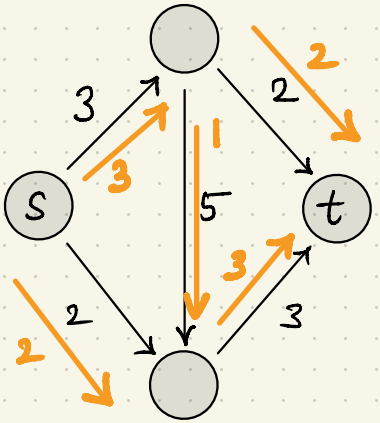
IDEA #2 : NAIVE GREEDY + "UNDO"



Idea #2: Naive Greedy + "undo"



IDEA #2: NAIVE GREEDY + "UNDO"



max flow !

IDEA #2: NAIVE GREEDY + "UNDO"

Let f be a feasible flow in G.

IDEA #2 : NAIVE GREEDY + "UNDO"

Let f be a feasible flow in G.

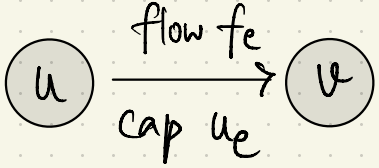
Residual graph G_f

IDEA #2 : NAIVE GREEDY + "UNDO"

Let f be a feasible flow in G.

Residual graph G_f

in G

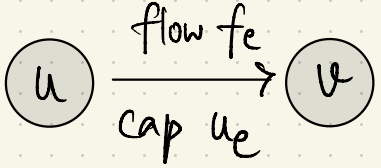


IDEA #2: NAIVE GREEDY + "UNDO"

Let f be a feasible flow in G.

Residual graph G_f

in G in G_f

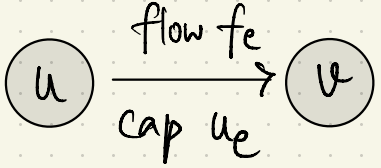


IDEA # 2 : NAIVE GREEDY + "UNDO"

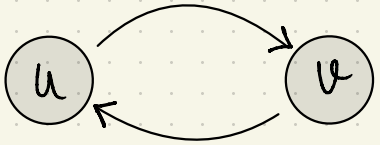
Let f be a feasible flow in G.

Residual graph G_f

in G



in G_f

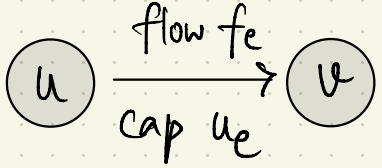


IDEA # 2 : NAIVE GREEDY + "UNDO"

Let f be a feasible flow in G.

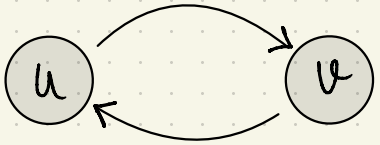
Residual graph G_f

in G



in G_f

residual cap = $u_e - f_e$

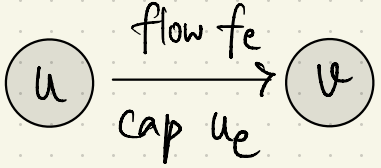


IDEA # 2 : NAIVE GREEDY + "UNDO"

Let f be a feasible flow in G.

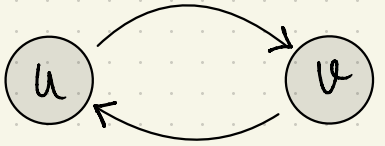
Residual graph G_f

in G



in G_f

residual cap = $u_e - f_e$



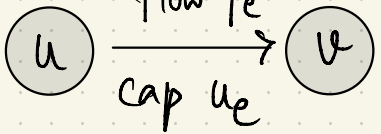
residual cap = f_e

IDEA # 2 : NAIVE GREEDY + "UNDO"

Let f be a feasible flow in G.

Residual graph G_f

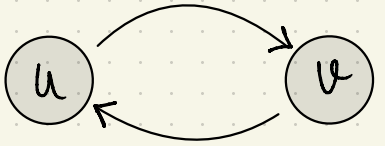
in G



in G_f

scope for pushing further

residual cap = $u_e - f_e$



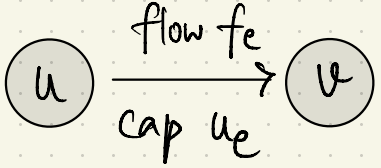
residual cap = f_e

IDEA # 2 : NAIVE GREEDY + "UNDO"

Let f be a feasible flow in G.

Residual graph G_f

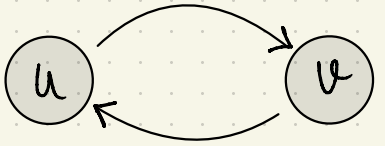
in G



in G_f

scope for pushing further

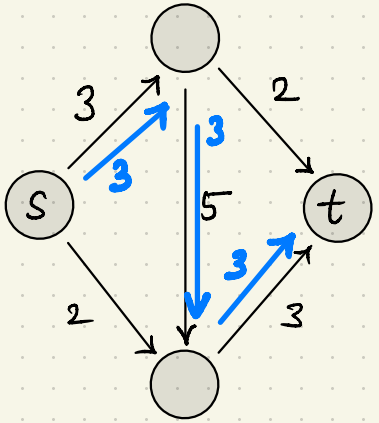
residual cap = $u_e - f_e$



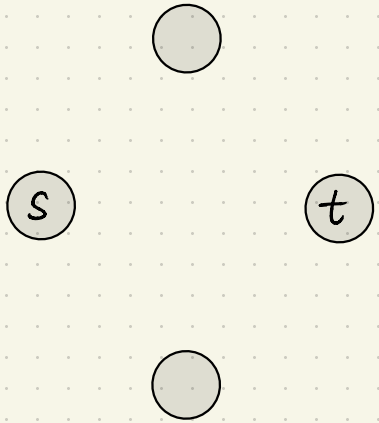
residual cap = f_e

scope for "undo"

IDEA #2 : NAIVE GREEDY + "UNDO"

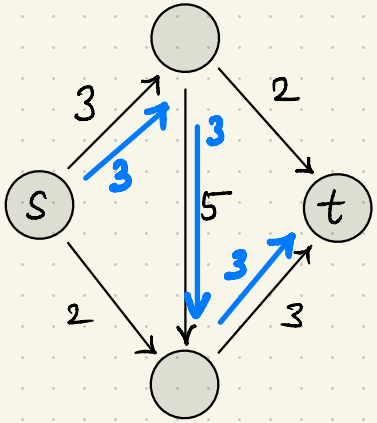


G

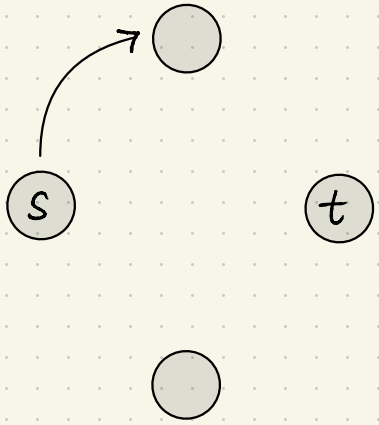


G_f

IDEA #2 : NAIVE GREEDY + "UNDO"

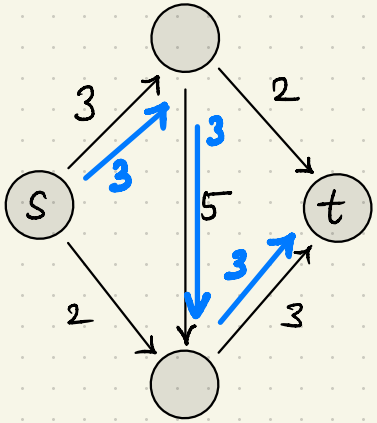


G

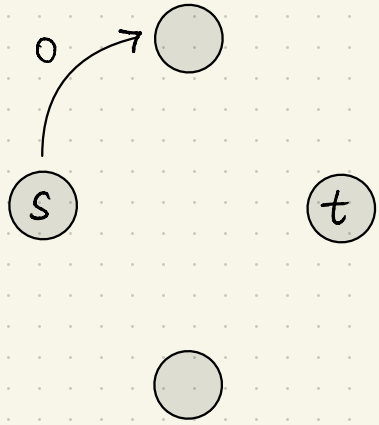


G_f

IDEA #2 : NAIVE GREEDY + "UNDO"

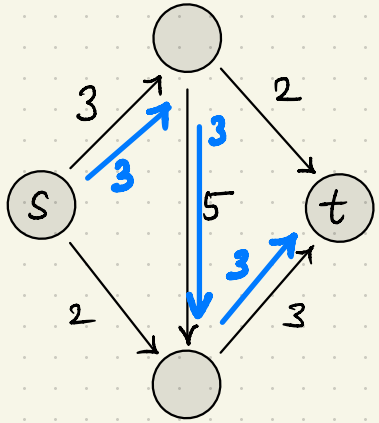


G

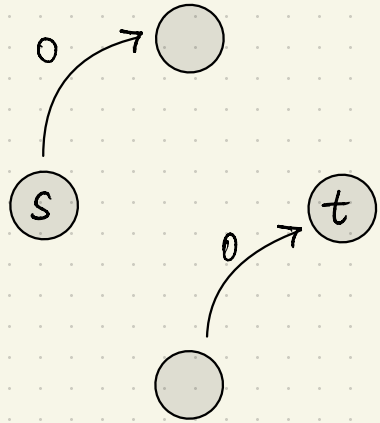


G_f

IDEA #2 : NAIVE GREEDY + "UNDO"

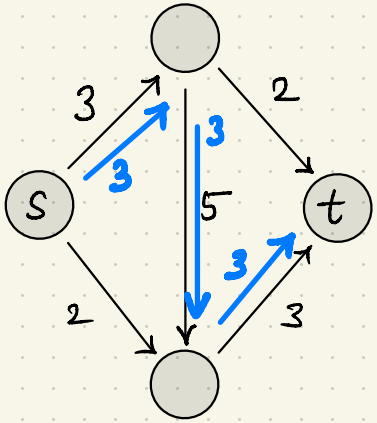


G

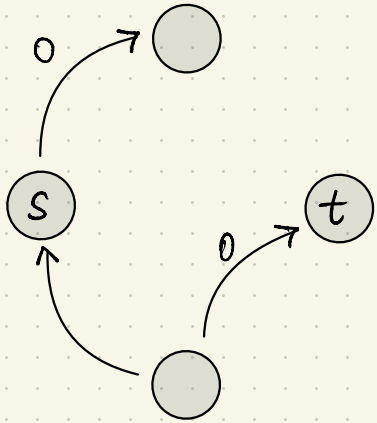


G_f

IDEA #2 : NAIVE GREEDY + "UNDO"

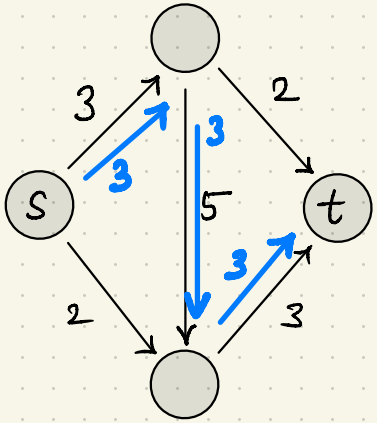


G

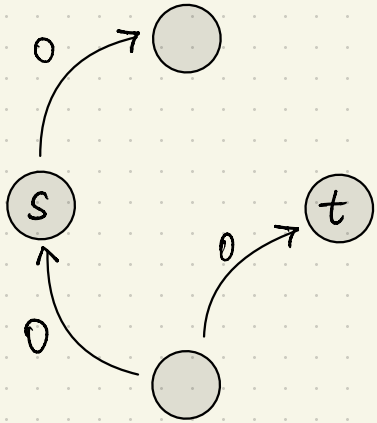


G_f

IDEA #2 : NAIVE GREEDY + "UNDO"

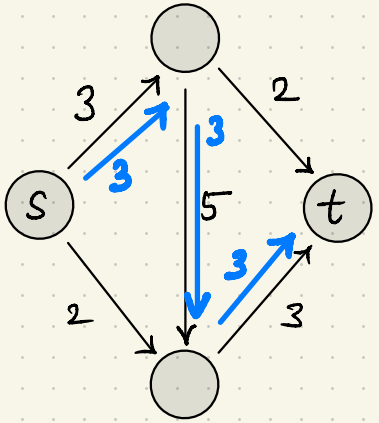


G

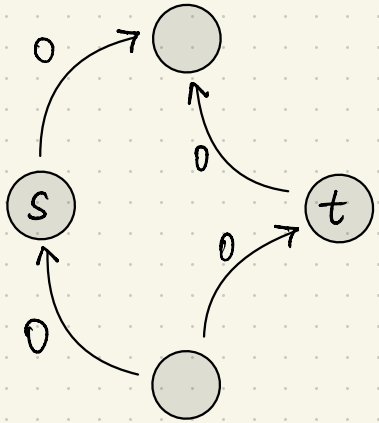


G_f

IDEA #2 : NAIVE GREEDY + "UNDO"

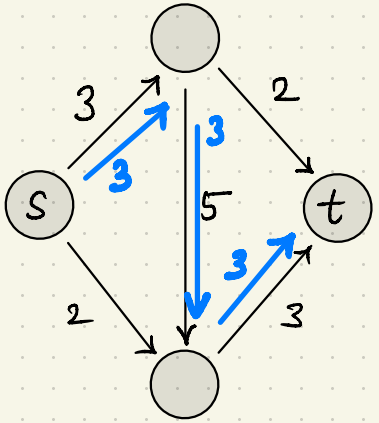


G

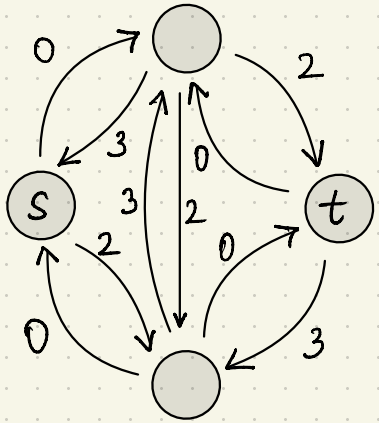


G_f

IDEA #2 : NAIVE GREEDY + "UNDO"

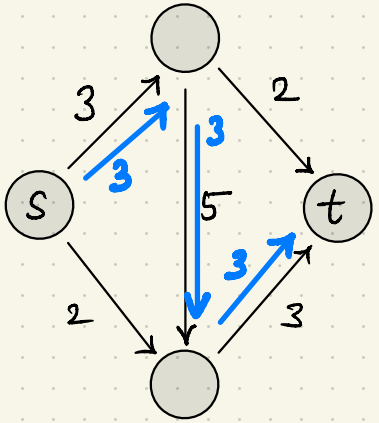


G

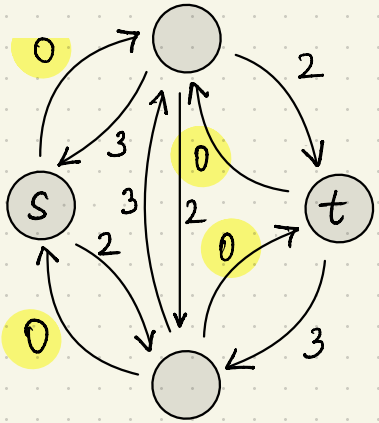


G_f

IDEA #2 : NAIVE GREEDY + "UNDO"

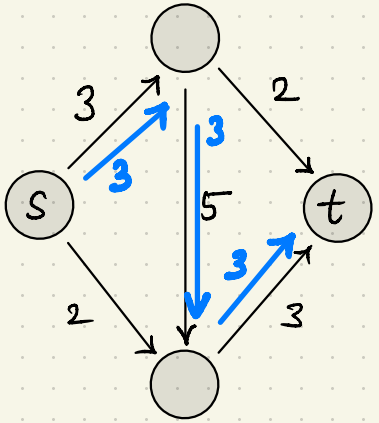


G

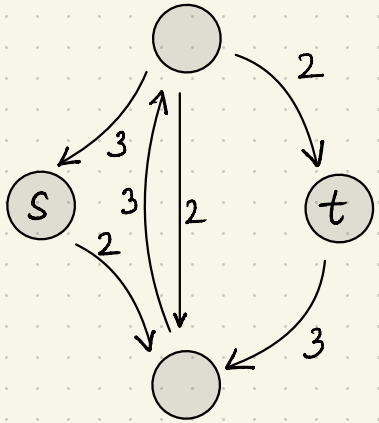


G_f

IDEA #2 : NAIVE GREEDY + "UNDO"

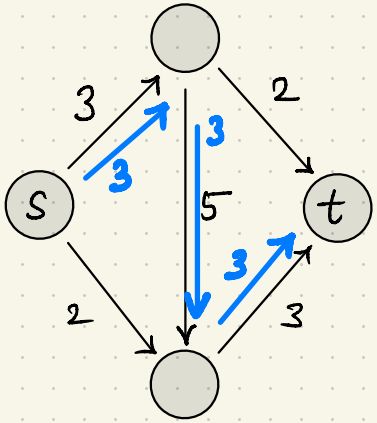


G

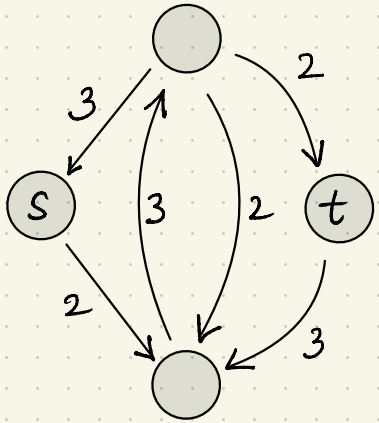


G_f

IDEA #2 : NAIVE GREEDY + "UNDO"

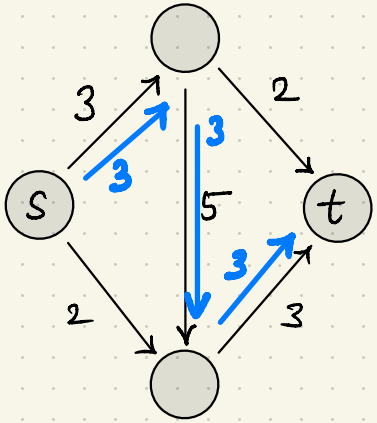


G

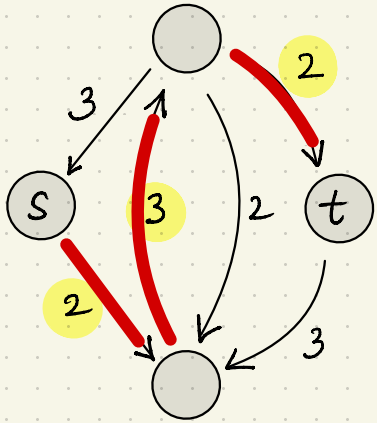


G_f

IDEA #2 : NAIVE GREEDY + "UNDO"

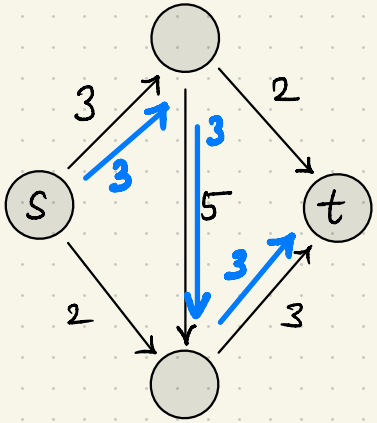


G



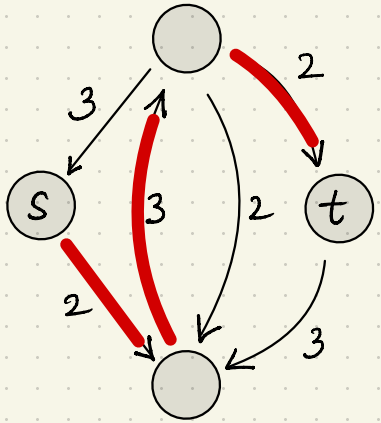
G_f

IDEA #2 : NAIVE GREEDY + "UNDO"



G

Undo



G_f



Run naive greedy in residual graph G_f

FORD - FULKERSON ALGORITHM

FORD - FULKERSON ALGORITHM

Naive greedy

① initialize $f_e = 0 \quad \forall e$

② repeat:

* search for an $s-t$ path P such that $f_e < u_e \ \forall e \in P$.

// possible in $O(|E|)$ time via BFS / DFS

* if no such path, return current flow $\{f_e\}_{e \in E}$
* else, let $\Delta := \min_{e \in P} (u_e - f_e)$

  for all edges $e \in P$, increase f_e by Δ.

FORD - FULKERSON ALGORITHM

① initialize $f_e = 0 \quad \forall e$

② repeat:

* search for an $s-t$ path P such that $f_e < u_e \ \forall e \in P$.

// possible in $O(|E|)$ time via BFS / DFS

* if no such path, return current flow $\{f_e\}_{e \in E}$
* else, let $\Delta := \min_{e \in P} (u_e - f_e)$

  for all edges $e \in P$, increase f_e by Δ.

FORD - FULKERSON ALGORITHM

① initialize $f_e = 0 \quad \forall e$

② repeat:

* search for an $s-t$ path P in G_f such that ~~$f_e < u_e \ \forall e \in P$~~. every edge of P has positive residual capacity

// possible in $O(|E|)$ time via BFS / DFS

* if no such path, return current flow $\{f_e\}_{e \in E}$
* else, let $\Delta := \min_{e \in P} (u_e - f_e)$

  for all edges $e \in P$, increase f_e by Δ.

FORD - FULKERSON ALGORITHM

① initialize $f_e = 0 \quad \forall e$

② repeat:

* search for an $s-t$ path P in G_f
  such that ~~$f_e < u_e \ \forall e \in P$~~. every edge of P has positive residual capacity

// possible in $O(|E|)$ time via BFS / DFS

* if no such path, return current flow $\{f_e\}_{e \in E}$
* else, let $\Delta := \min_{e \in P} (u_e - f_e)$

  for all edges $e \in P$, increase f_e by Δ.

FORD - FULKERSON ALGORITHM

① initialize $f_e = 0 \quad \forall e$

② repeat:

* search for an $s-t$ path P in G_f such that ~~$f_e < u_e \ \forall e \in P$~~. every edge of P has positive residual capacity

// possible in $O(|E|)$ time via BFS / DFS

* if no such path, return current flow $\{f_e\}_{e \in E}$
* else, let $\Delta := \min_{e \in P} (u_e - f_e)$ e's residual capacity in G_f

  for all edges $e \in P$, increase f_e by Δ.

FORD - FULKERSON ALGORITHM

① initialize $f_e = 0 \quad \forall e$

② repeat:

* search for an $s-t$ path P in G_f
  such that ~~$f_e < u_e \ \forall e \in P$~~. every edge of P has positive residual capacity

// possible in $O(|E|)$ time via BFS / DFS

* if no such path, return current flow $\{f_e\}_{e \in E}$
* else, let $\Delta := \min_{e \in P} (u_e - f_e)$ e's residual capacity in G_f

  for all edges $e \in P$, increase f_e by Δ.

FORD - FULKERSON ALGORITHM

① initialize $f_e = 0 \quad \forall e$

② repeat:

* search for an $s-t$ path P in G_f such that ~~$f_e < u_e \ \forall e \in P$~~. every edge of P has positive residual capacity

// possible in $O(|E|)$ time via BFS / DFS

* if no such path, return current flow $\{f_e\}_{e \in E}$

* else, let $\Delta := \min_{e \in P} (u_e - f_e)$ e's residual capacity in G_f

  ~~for all edges $e \in P$, increase f_e by Δ.~~

for all $e \in G$ whose forward edge is in P, increase f_e by Δ

" " " " reverse " " , decrease " " .

FORD - FULKERSON ALGORITHM

① initialize $f_e = 0 \quad \forall e$

② repeat:

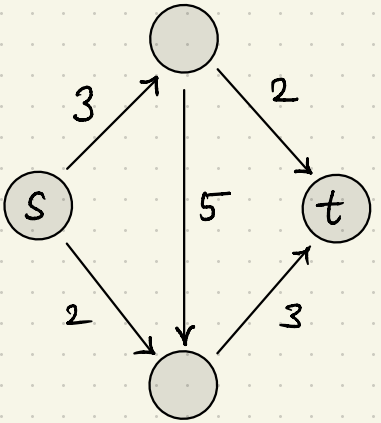
* search for an $s-t$ path P in G_f
  such that every edge in P has positive residual capacity

// possible in $O(|E|)$ time via BFS / DFS

* if no such path, return current flow $\{f_e\}_{e \in E}$
* else, let $\Delta := \min_{e \in P}$ e's residual capacity in G_f

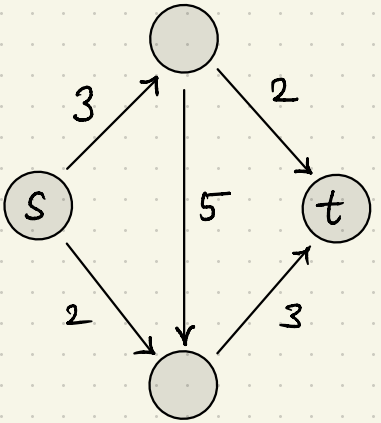
  for all $e \in G$ with forward edges in P, add Δ to f_e.
  for all $e \in G$ with reverse edges in P, subtract Δ from f_e.

FORD - FULKERSON ALGORITHM

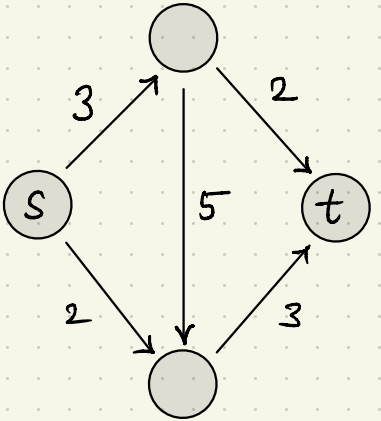


G

FORD - FULKERSON ALGORITHM



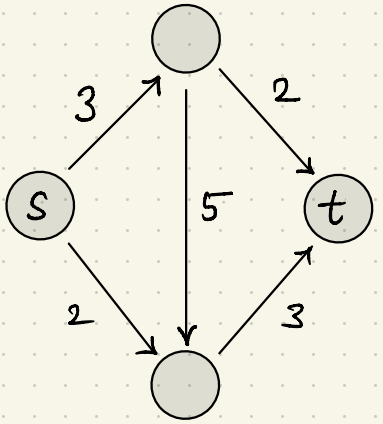
G



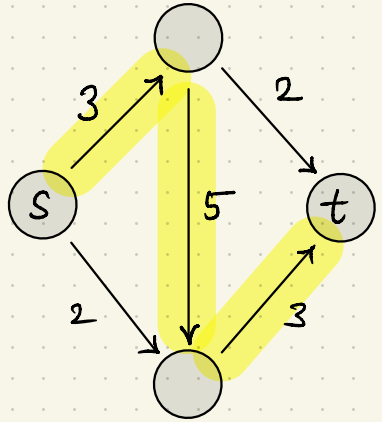
G_f

(Omitting zero capacity edges)

FORD - FULKERSON ALGORITHM

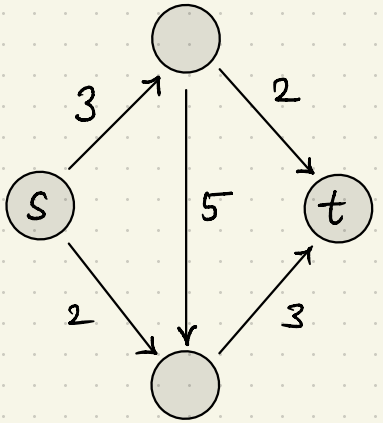


G

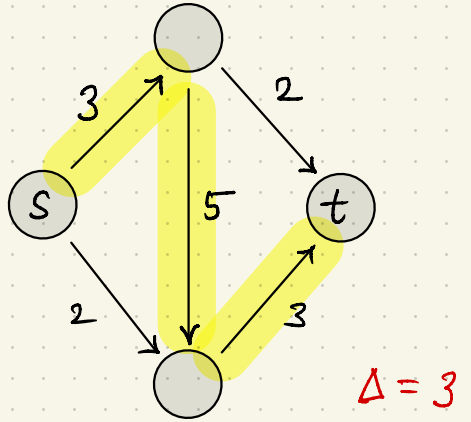


G_f

FORD - FULKERSON ALGORITHM



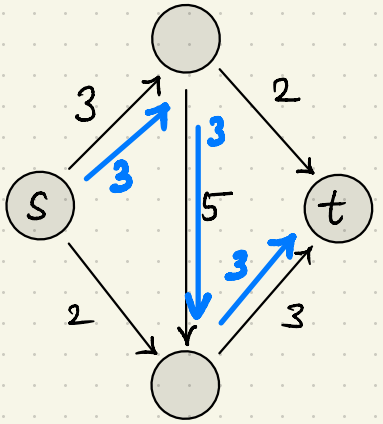
G



$\Delta = 3$

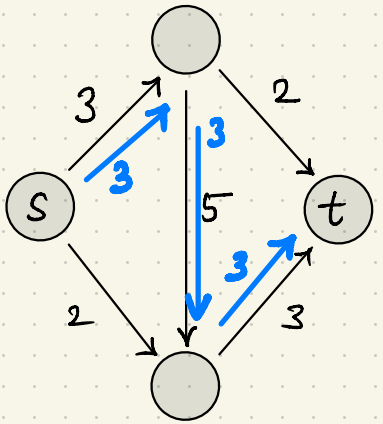
G_f

FORD - FULKERSON ALGORITHM

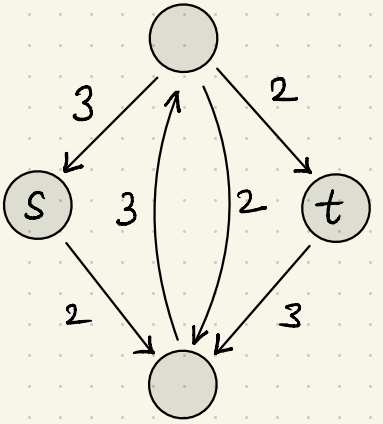


G

FORD - FULKERSON ALGORITHM

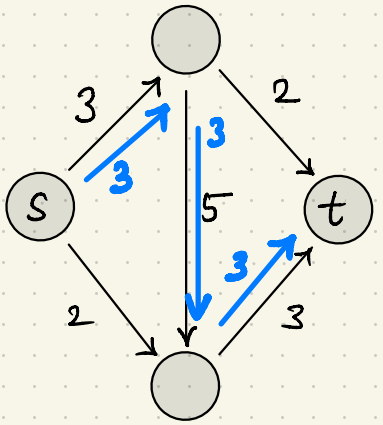


G

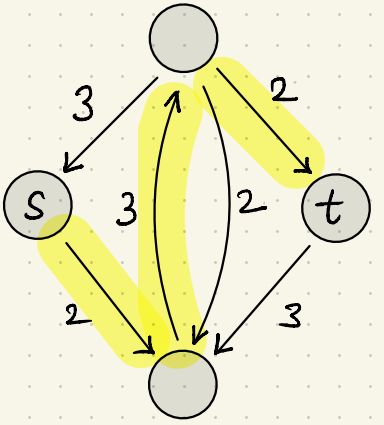


G_f

FORD - FULKERSON ALGORITHM

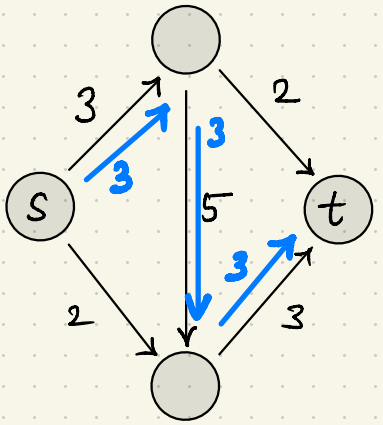


G

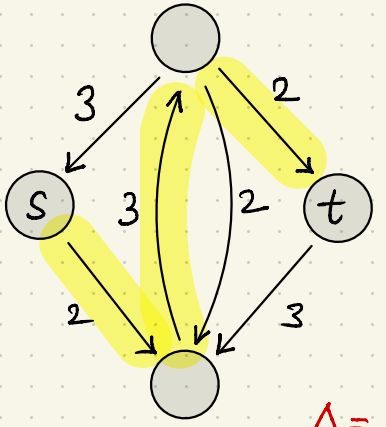


G_f

FORD - FULKERSON ALGORITHM



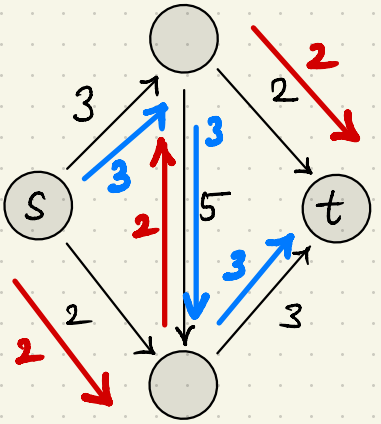
G



$\Delta = 2$

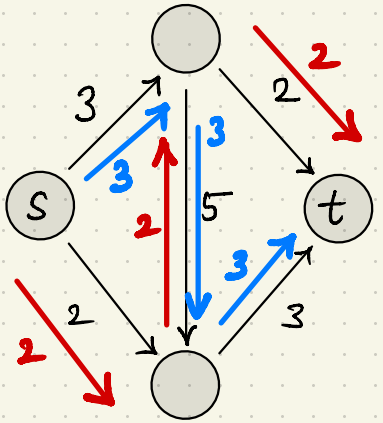
G_f

FORD - FULKERSON ALGORITHM

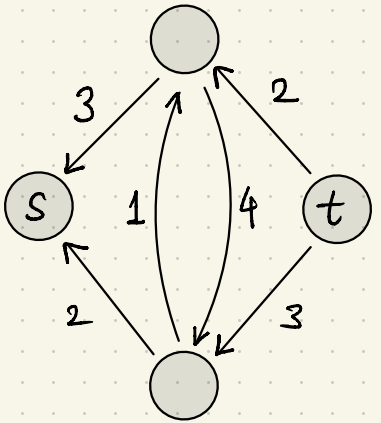


G

FORD - FULKERSON ALGORITHM

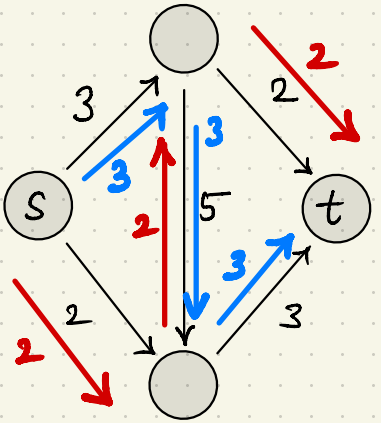


G

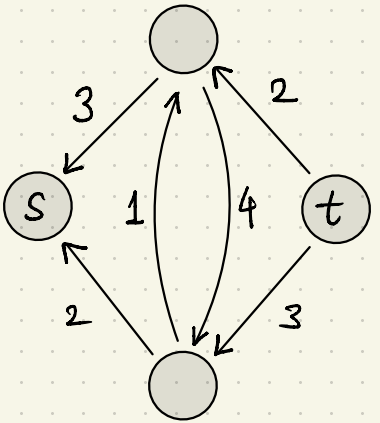


G_f

Ford - Fulkerson Algorithm



G



G_f

No s-t path in G_f

$\Rightarrow$ done!

Termination of Ford-Fulkerson

Termination of Ford-Fulkerson

Claim 1: The algorithm maintains the invariant that $\{f_e\}_{e \in E}$ is a flow.

TERMINATION OF FORD-FULKERSON

Claim 1: The algorithm maintains the invariant that $\{f_e\}_{e \in E}$ is a flow.

Proof: True initially for $f_e = 0 \quad \forall e \in E$.

TERMINATION OF FORD-FULKERSON

Claim 1: The algorithm maintains the invariant that $\{f_e\}_{e \in E}$ is a flow.

Proof: True initially for $f_e = 0 \quad \forall e \in E$.

At each step:

(1) Δ is chosen to maintain non-negativity and capacity constraints.

Termination of Ford-Fulkerson

Claim 1: The algorithm maintains the invariant that $\{f_e\}_{e \in E}$ is a flow.

Proof: True initially for $f_e = 0 \quad \forall e \in E$.

At each step:

(1) Δ is chosen to maintain non-negativity and capacity constraints.

(2) For conservation constraints, consider a vertex v.

TERMINATION OF FORD-FULKERSON

Claim 1: The algorithm maintains the invariant that $\{f_e\}_{e \in E}$ is a flow.

Proof: True initially for $f_e = 0 \quad \forall e \in E$.

At each step:

(1) Δ is chosen to maintain non-negativity and capacity constraints.

(2) For conservation constraints, consider a vertex v.

- If v is not on the path P, then flow in and out of v is unchanged.

TERMINATION OF FORD-FULKERSON

Claim 1: The algorithm maintains the invariant that $\{f_e\}_{e \in E}$ is a flow.

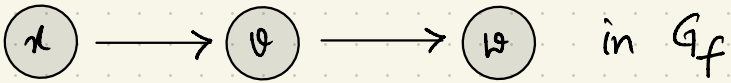
Proof: True initially for $f_e = 0 \quad \forall e \in E$.

At each step:

(1) Δ is chosen to maintain non-negativity and capacity constraints.

(2) For conservation constraints, consider a vertex v.

- If v is not on the path P, then flow in and out of v is unchanged.
- If v is on the path P, say via the edges (u, v) and (v, w).

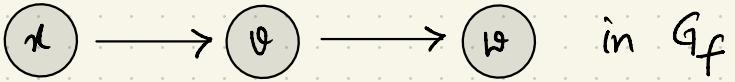


in G_f

TERMINATION OF FORD-FULKERSON

Claim 1: The algorithm maintains the invariant that $\{f_e\}_{e \in E}$ is a flow.

Proof: If v is on the path P, say via the edges (u, v) and (v, w).

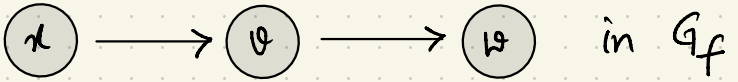


in G_f

Termination of Ford-Fulkerson

Claim 1: The algorithm maintains the invariant that $\{f_e\}_{e \in E}$ is a flow.

Proof: If v is on the path P, say via the edges (u, v) and (v, w).



in G_f

Case I:

Case II:

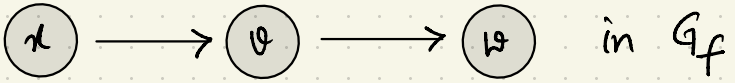
Case III:

Case IV:

TERMINATION OF FORD-FULKERSON

Claim 1: The algorithm maintains the invariant that $\{f_e\}_{e \in E}$ is a flow.

Proof: If v is on the path P, say via the edges (u, v) and (v, w).



in G_f

Case I: Both (u, v) and (v, w) are forward edges.

Case II: Both (u, v) and (v, w) are reverse edges.

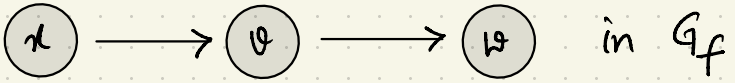
Case III: (u, v) is a forward edge and (v, w) is a reverse edge

Case IV: (u, v) is a reverse edge and (v, w) is a forward edge

TERMINATION OF FORD-FULKERSON

Claim 1: The algorithm maintains the invariant that $\{f_e\}_{e \in E}$ is a flow.

Proof: If v is on the path P, say via the edges (u, v) and (v, w).



in G_f

Case I: Both (u, v) and (v, w) are forward edges.

$\Rightarrow$ flow-in $\uparrow$ by Δ, flow-out $\uparrow$ by Δ.

Case II: Both (u, v) and (v, w) are reverse edges.

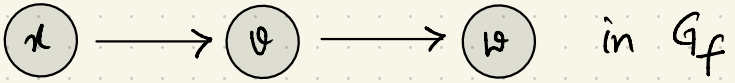
Case III: (u, v) is a forward edge and (v, w) is a reverse edge

Case IV: (u, v) is a reverse edge and (v, w) is a forward edge

TERMINATION OF FORD-FULKERSON

Claim 1: The algorithm maintains the invariant that $\{f_e\}_{e \in E}$ is a flow.

Proof: If v is on the path P, say via the edges (u, v) and (v, w).



in G_f

Case I: Both (u, v) and (v, w) are forward edges.

$\Rightarrow$ flow-in $\uparrow$ by Δ, flow-out $\uparrow$ by Δ.

Case II: Both (u, v) and (v, w) are reverse edges.

$\Rightarrow$ flow-out $\downarrow$ by Δ, flow-in $\downarrow$ by Δ.

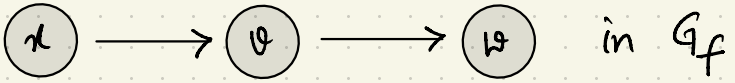
Case III: (u, v) is a forward edge and (v, w) is a reverse edge

Case IV: (u, v) is a reverse edge and (v, w) is a forward edge

TERMINATION OF FORD-FULKERSON

Claim 1: The algorithm maintains the invariant that $\{f_e\}_{e \in E}$ is a flow.

Proof: If v is on the path P, say via the edges (u, v) and (v, w).



in G_f

Case I: Both (u, v) and (v, w) are forward edges.

$\Rightarrow$ flow-in $\uparrow$ by Δ, flow-out $\uparrow$ by Δ.

Case II: Both (u, v) and (v, w) are reverse edges.

$\Rightarrow$ flow-out $\downarrow$ by Δ, flow-in $\downarrow$ by Δ.

Case III: (u, v) is a forward edge and (v, w) is a reverse edge

$\Rightarrow$ flow-in $\uparrow$ by Δ and $\downarrow$ by Δ.

Case IV: (u, v) is a reverse edge and (v, w) is a forward edge

TERMINATION OF FORD-FULKERSON

Claim 1: The algorithm maintains the invariant that $\{f_e\}_{e \in E}$ is a flow.

Proof: If v is on the path P, say via the edges (u, v) and (v, w).



in G_f

Case I: Both (u, v) and (v, w) are forward edges.

$\Rightarrow$ flow-in $\uparrow$ by Δ, flow-out $\uparrow$ by Δ.

Case II: Both (u, v) and (v, w) are reverse edges.

$\Rightarrow$ flow-out $\downarrow$ by Δ, flow-in $\downarrow$ by Δ.

Case III: (u, v) is a forward edge and (v, w) is a reverse edge

$\Rightarrow$ flow-in $\uparrow$ by Δ and $\downarrow$ by Δ.

Case IV: (u, v) is a reverse edge and (v, w) is a forward edge

$\Rightarrow$ flow-out $\downarrow$ by Δ and $\uparrow$ by Δ.

TERMINATION OF FORD-FULKERSON

Claim 1: The algorithm maintains the invariant that $\{f_e\}_{e \in E}$ is a flow.

Proof: If v is on the path P, say via the edges (u, v) and (v, w).



in G_f

Case I: Both (u, v) and (v, w) are forward edges.

$\Rightarrow$ flow-in $\uparrow$ by Δ, flow-out $\uparrow$ by Δ.

Case II: Both (u, v) and (v, w) are reverse edges.

$\Rightarrow$ flow-out $\downarrow$ by Δ, flow-in $\downarrow$ by Δ.

Case III: (u, v) is a forward edge and (v, w) is a reverse edge

$\Rightarrow$ flow-in $\uparrow$ by Δ and $\downarrow$ by Δ.

Case IV: (u, v) is a reverse edge and (v, w) is a forward edge

$\Rightarrow$ flow-out $\downarrow$ by Δ and $\uparrow$ by Δ.

TERMINATION OF FORD-FULKERSON

Claim 2: The algorithm maintains the invariant that $\{f_e\}_{e \in E}$ is integral.

Termination of Ford-Fulkerson

Claim 2: The algorithm maintains the invariant that $\{f_e\}_{e \in E}$ is integral.

Proof: By induction. All capacities and residual capacities are integers.

So, Δ is integral, and so are f_e's. ∎

TERMINATION OF FORD-FULKERSON

Claim 2: The algorithm maintains the invariant that $\{f_e\}_{e \in E}$ is integral.

Proof: By induction. All capacities and residual capacities are integers. So, Δ is integral, and so are f_e's. ∎

Claim 3: $\Delta \geq 1$.

TERMINATION OF FORD-FULKERSON

Claim 2: The algorithm maintains the invariant that $\{f_e\}_{e \in E}$ is integral.

Proof: By induction. All capacities and residual capacities are integers. So, Δ is integral, and so are f_e's. ∎

Claim 3: $\Delta \geq 1$.

Proof: Claim 2 + positive residual capacity path. ∎

Termination of Ford-Fulkerson

Claim 1: The algorithm maintains the invariant that $\{f_e\}_{e \in E}$ is a flow.

Claim 2: The algorithm maintains the invariant that $\{f_e\}_{e \in E}$ is integral.

Claim 3: $\Delta \geq 1$.

TERMINATION OF FORD-FULKERSON

Claim 1: The algorithm maintains the invariant that $\{f_e\}_{e \in E}$ is a flow.

Claim 2: The algorithm maintains the invariant that $\{f_e\}_{e \in E}$ is integral.

Claim 3: $\Delta \geq 1$.

Claims 1, 2, 3 and the fact that finite amount of capacity exiting s

$\Rightarrow$ termination in a finite number of iterations.

TERMINATION OF FORD-FULKERSON

Claim 1: The algorithm maintains the invariant that $\{f_e\}_{e \in E}$ is a flow.

Claim 2: The algorithm maintains the invariant that $\{f_e\}_{e \in E}$ is integral.

Claim 3: $\Delta \geq 1$.

Claims 1, 2, 3 and the fact that finite amount of capacity exiting s

$\Rightarrow$ termination in a finite number of iterations.



What if capacities are rational-valued?

What if they are irrational?