COL 351 : Analysis & Design of Algorithms

LECTURE 12

GRAPH ALGORITHMS V :

STRONGLY CONNECTED COMPONENTS & DIJKSTRA's ALGORITHM

AUG 21, 2024 | ROHIT VAISH

APPLICATIONS OF DFS

Topological ordering

Strongly Connected Components

APPLICATIONS OF DFS

Topological ordering
- Remove sink iteratively
- via DFS

Strongly Connected Components

APPLICATIONS OF DFS

Topological ordering
- Remove sink iteratively $O(n^2)$
- via DFS $O(m+n)$

Strongly Connected Components

APPLICATIONS OF DFS

Topological ordering
- Remove sink iteratively $O(n^2)$
- via DFS $O(m+n)$

Strongly Connected Components

TOPOLOGICAL ORDERING via DFS

DFS pseudocode

```
DFS (G, s)   // all vertices initially unexplored
mark s as explored
for each (s, v) in adj. list of s
    if v is unexplored
        DFS (G, v)
```

TOPOLOGICAL ORDERING via DFS

DFS-Loop (G)

DFS (G, s) // all vertices initially unexplored

mark s as explored

for each (s, v) in adj. list of s

    if v is unexplored

        DFS (G, v)

TOPOLOGICAL ORDERING via DFS

```
DFS-Loop(G)
mark all vertices unexplored
current_label := |V| // labeling f
```

```
DFS(G, s) // all vertices initially unexplored
mark s as explored
for each (s, v) in adj. list of s
    if v is unexplored
        DFS(G, v)
```

TOPOLOGICAL ORDERING via DFS

```
DFS-Loop(G)
mark all vertices unexplored
current_label := |V|  // labeling f
for each v ∈ V
    if v is unexplored
        DFS(G, v)
```

```
DFS(G, s)  // all vertices initially unexplored
mark s as explored
for each (s, v) in adj. list of s
    if v is unexplored
        DFS(G, v)
```

TOPOLOGICAL ORDERING via DFS

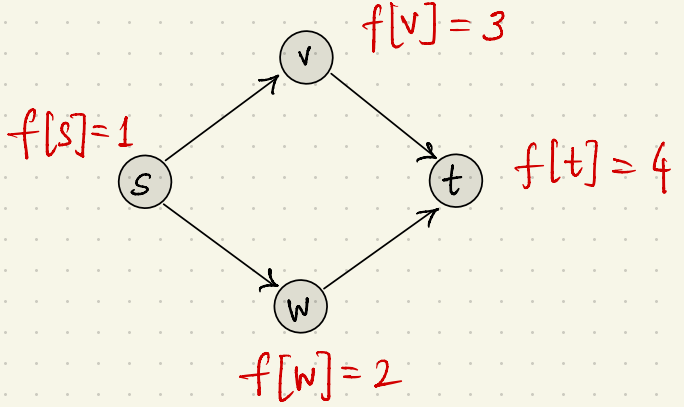
```
DFS-Loop(G)
mark all vertices unexplored
current_label := |V|  // labeling f
for each v ∈ V
    if v is unexplored
        DFS(G, v)
```

```
DFS(G, s)  // all vertices initially unexplored
mark s as explored
for each (s, v) in adj. list of s
    if v is unexplored
        DFS(G, v)
Set f[s] = current_label
decrease current_label by 1
```

TOPOLOGICAL ORDERING via DFS

Claim : For any DAG $G = (V, E)$, the labeling f generated by DFS-Loop(G) has the following property :

if $(u, v) \in E$ then $f[u] < f[v]$.



TOPOLOGICAL ORDERING via DFS

Claim : For any DAG $G = (V, E)$, the labeling f generated by DFS-Loop(G) has the following property :

$$\text{if } (u, v) \in E \quad \text{then} \quad f[u] < f[v].$$

NOTE : DFS-Loop algorithm terminates on any input graph and returns some labeling f.

- any → possibly cyclic
- some → not necessarily topological ordering

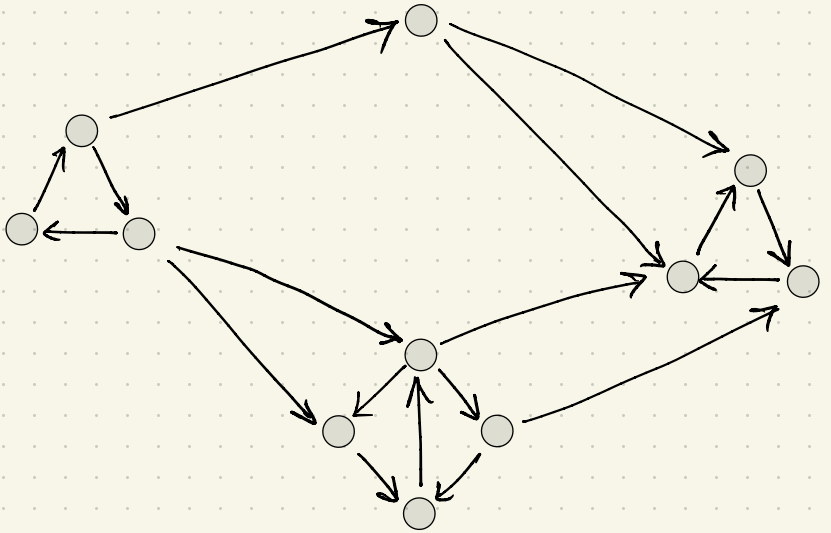
APPLICATIONS OF DFS

Topological ordering

Strongly Connected Components

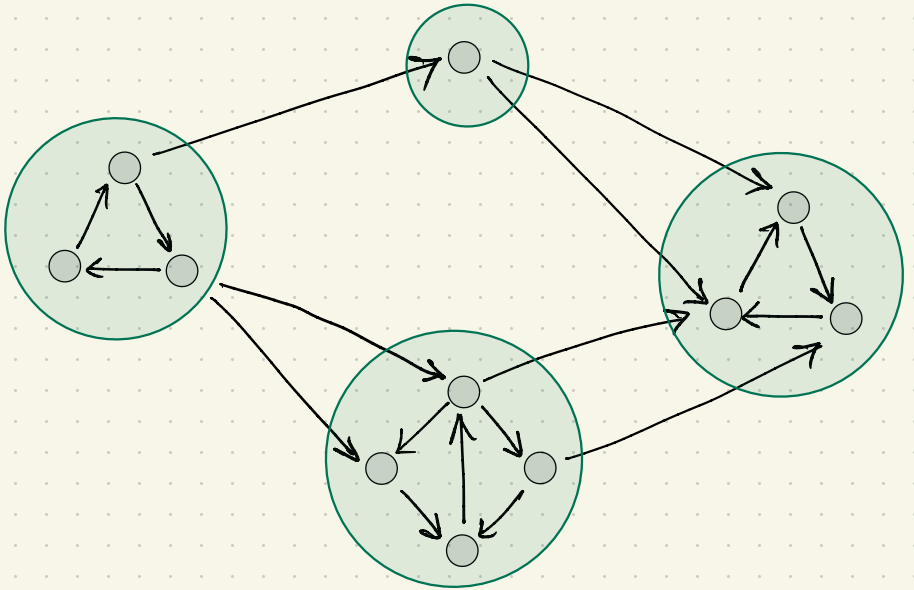
STRONGLY CONNECTED COMPONENTS

A strongly connected component of a directed graph G is a maximal subgraph of G that is strongly connected.



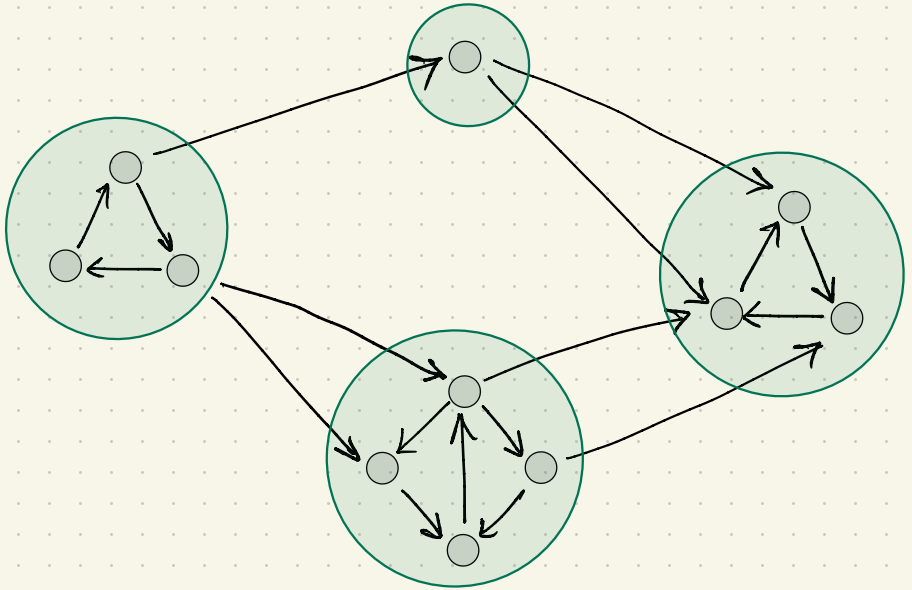
STRONGLY CONNECTED COMPONENTS

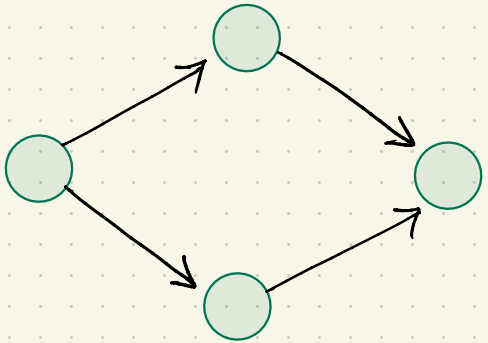
A strongly connected component of a directed graph G is a maximal subgraph of G that is strongly connected.



STRONGLY CONNECTED COMPONENTS

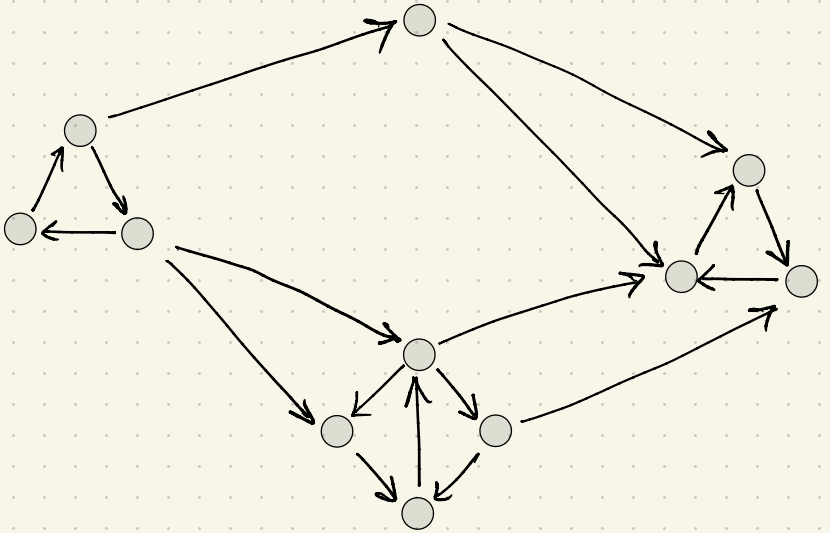
A strongly connected component of a directed graph G is a maximal subgraph of G that is strongly connected.



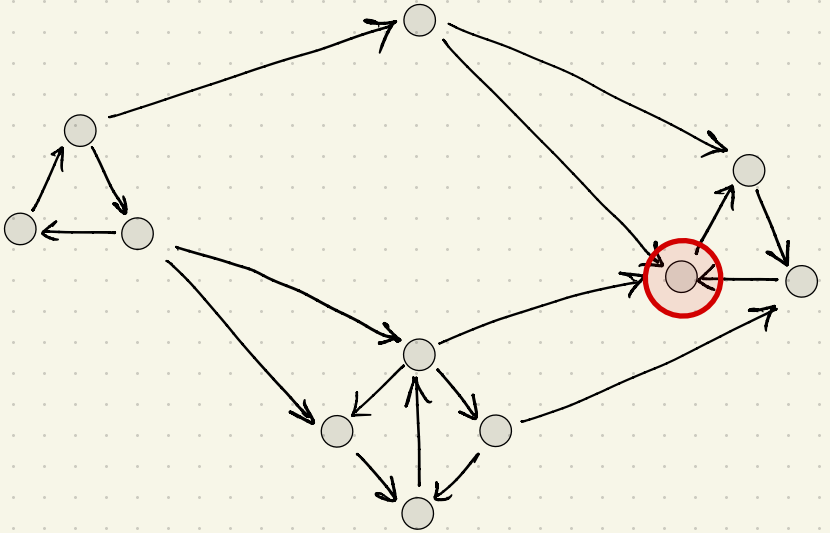


Meta graph is a DAG!

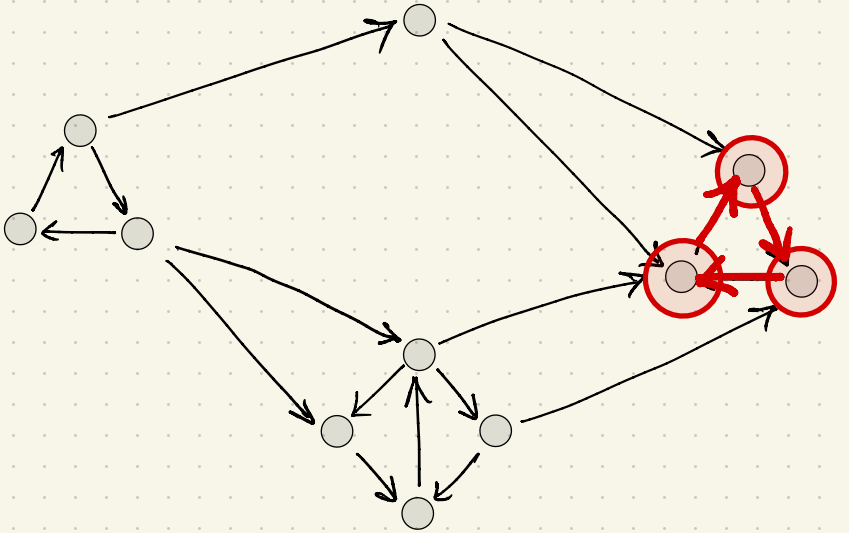
STRONGLY CONNECTED COMPONENTS via DFS



STRONGLY CONNECTED COMPONENTS via DFS

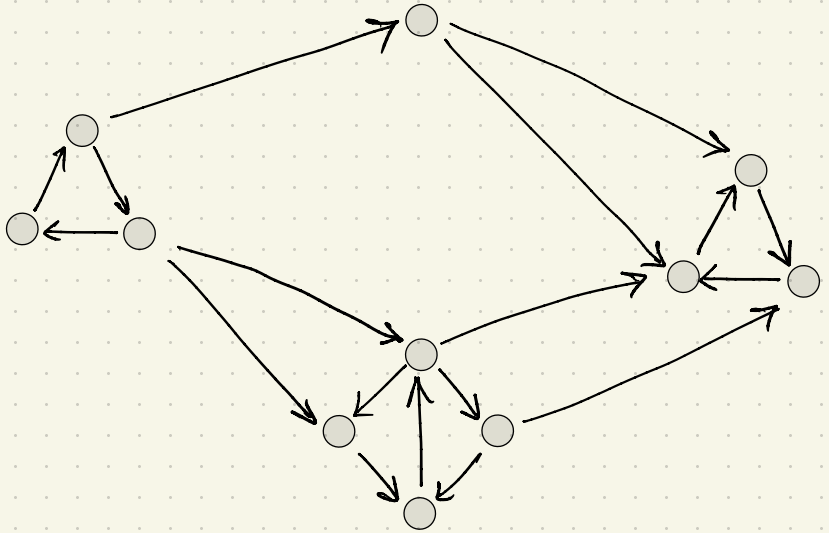


STRONGLY CONNECTED COMPONENTS via DFS



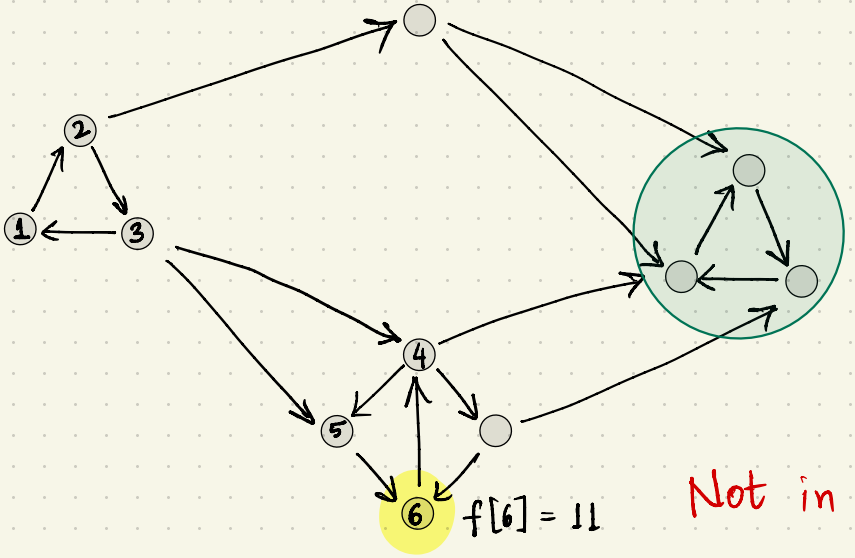
an SCC!

STRONGLY CONNECTED COMPONENTS via DFS



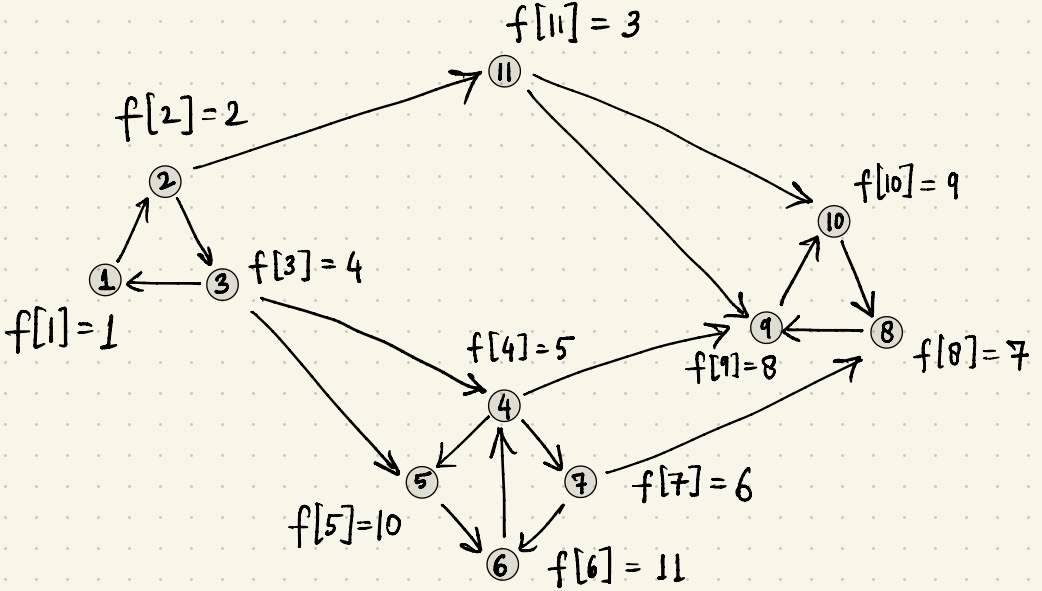
🤔 Maybe the "largest label" vertex always lies in the sink SCC?
(as per topological ordering algo)

First Attempt

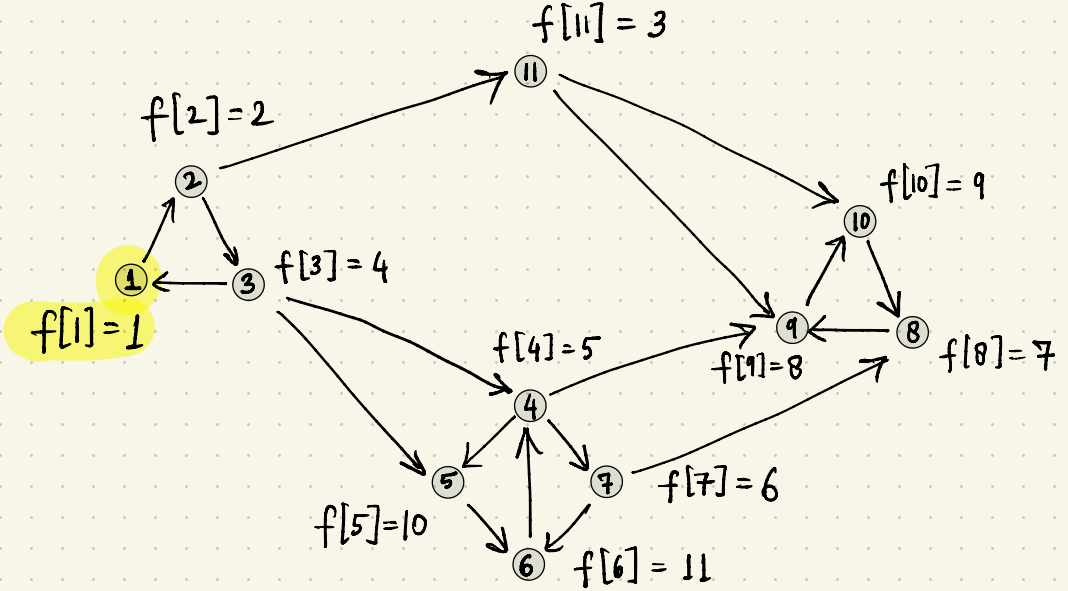


Not in "sink" SCC ☹

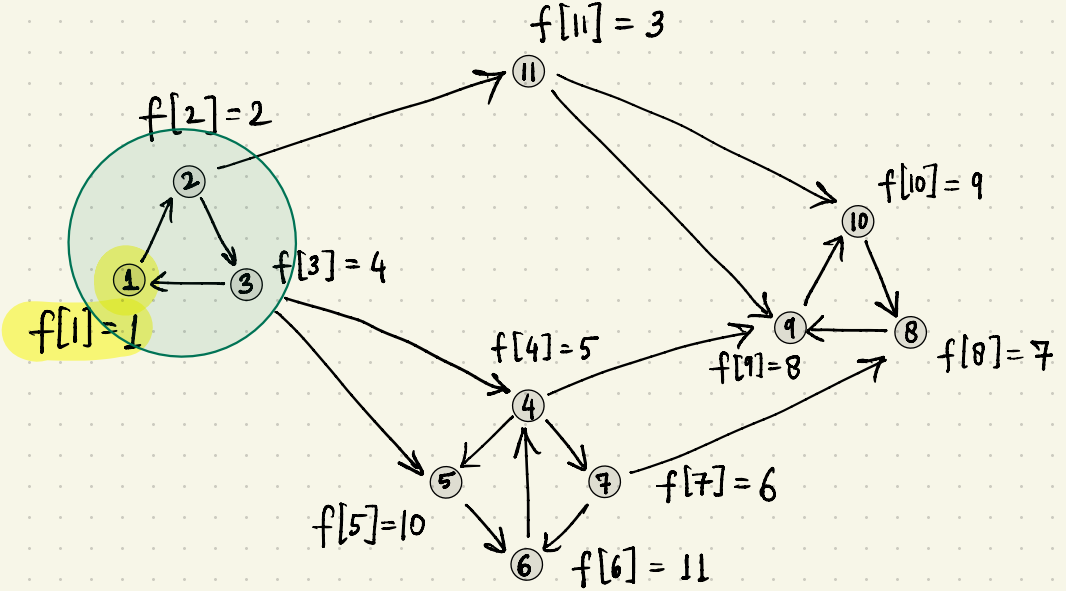
First Attempt



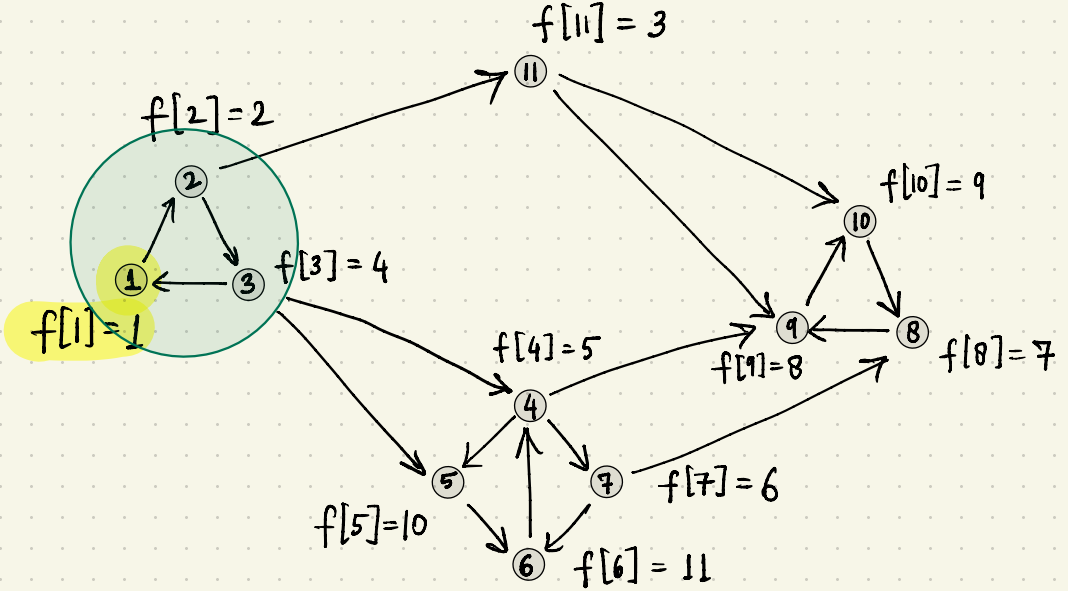
First Attempt



First Attempt



First Attempt

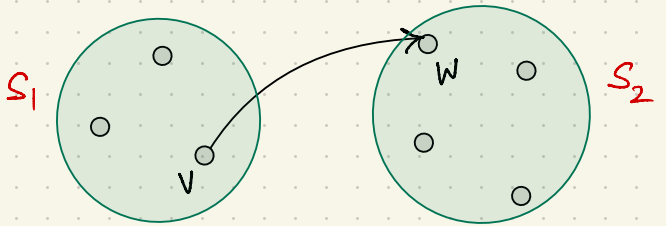


Perhaps the "smallest-label" vertex is always in "source" SCC?
(as per topological ordering)

KEY OBSERVATION

Theorem: Let f be the labeling of directed graph G generated by the topological ordering algorithm on G (arbitrary ordering of vertices). Let S_1, S_2 be two "adjacent" SCCs of G, i.e., there is an edge (v, w) with $v \in S_1$ and $w \in S_2$. Then,

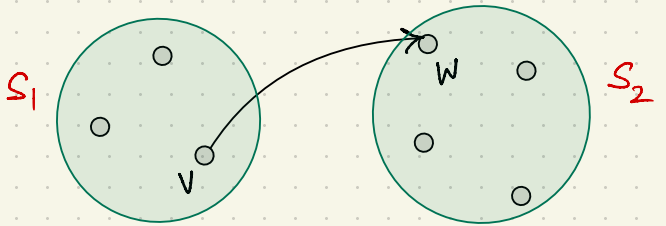
$$\min_{x \in S_1} f(x) < \min_{y \in S_2} f(y).$$



KEY OBSERVATION

Theorem: Let f be the labeling of directed graph G generated by the topological ordering algorithm on G (arbitrary ordering of vertices). Let S_1, S_2 be two "adjacent" SCCs of G, i.e., there is an edge (v, w) with $v \in S_1$ and $w \in S_2$. Then,

$$\min_{x \in S_1} f(x) < \min_{y \in S_2} f(y).$$



recursive call of some vertex in S_1 finishes after that of all vertices in S_2.

KEY OBSERVATION

Theorem: Let f be the labeling of directed graph G generated by the topological ordering algorithm on G (arbitrary ordering of vertices). Let S_1, S_2 be two "adjacent" SCCs of G, i.e., there is an edge (v, w) with $v \in S_1$ and $w \in S_2$. Then,

$$\min_{x \in S_1} f(x) < \min_{y \in S_2} f(y).$$

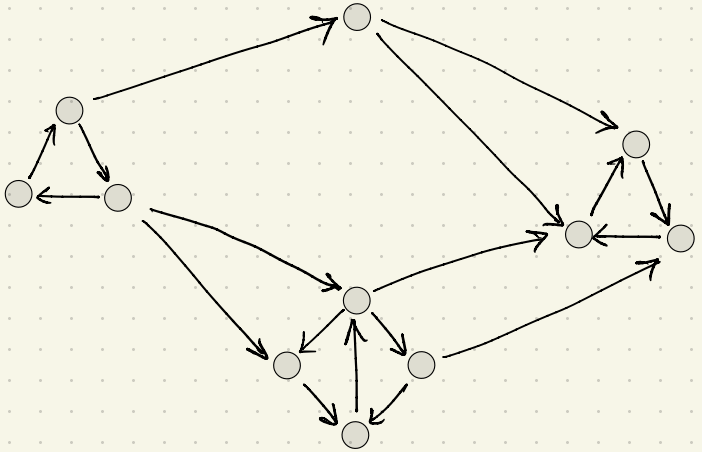
Corollary: The vertex v with $f[v] = 1$ must lie in source SCC.

What we have: A way of identifying a vertex in source SCC.

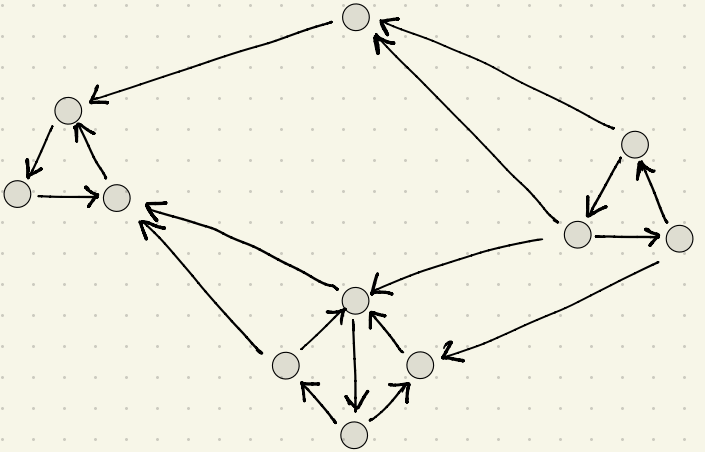
What we want: A way of identifying a vertex in sink SCC.

What's the fix: Reverse the graph!

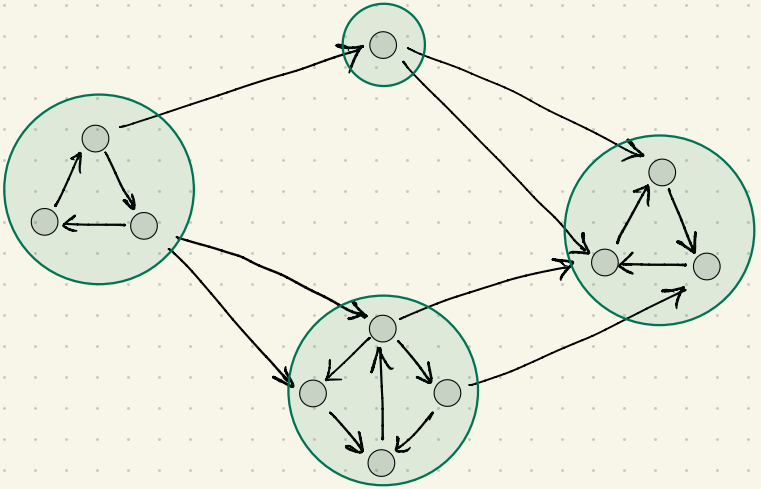
G



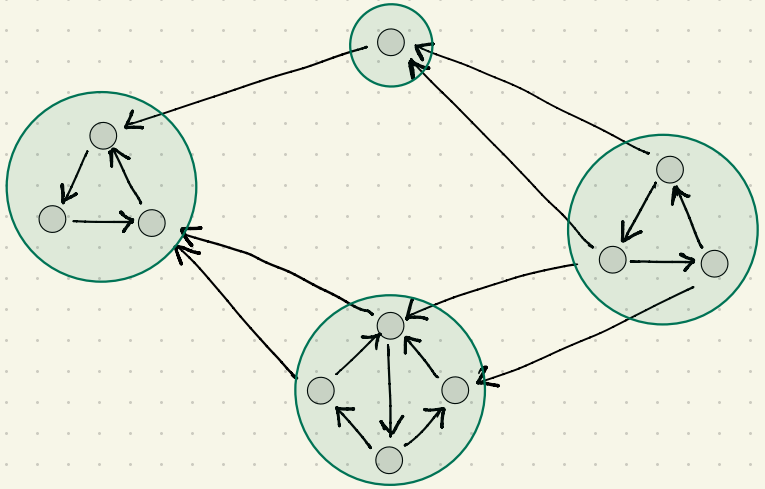
G^{rev}



G



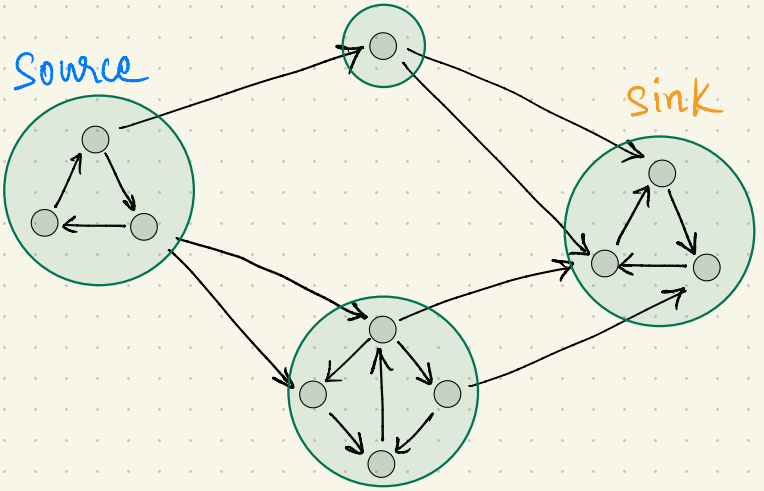
G^{rev}

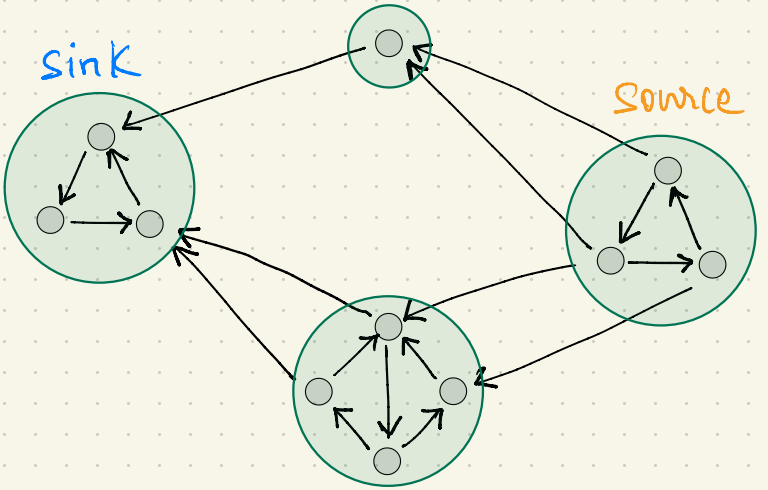


SCCs stay the same!

G

G^{rev}

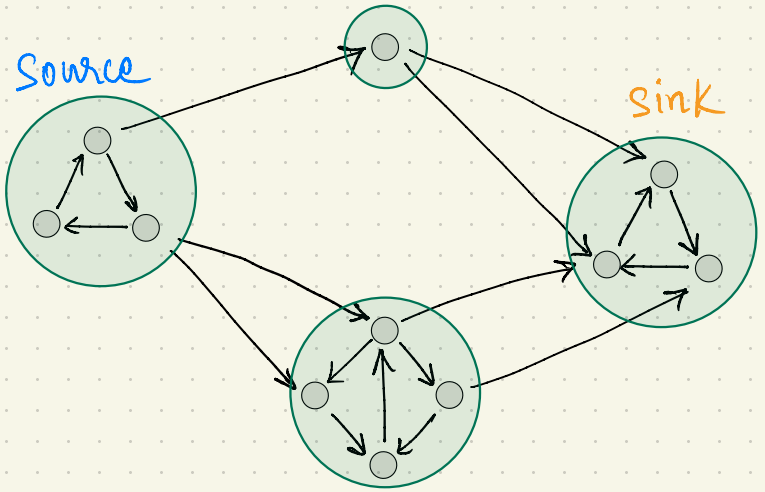




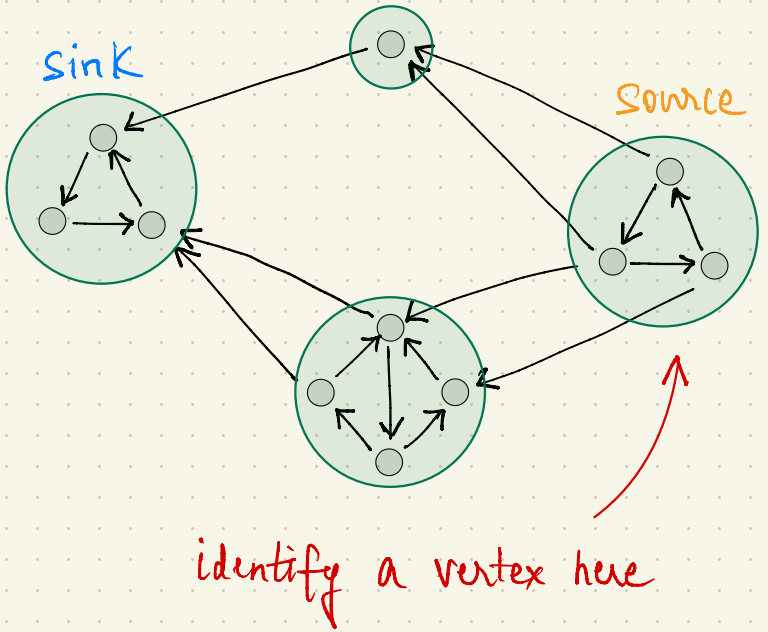
Source / sink in meta graph of G

Sink / Source " " " G^{rev}

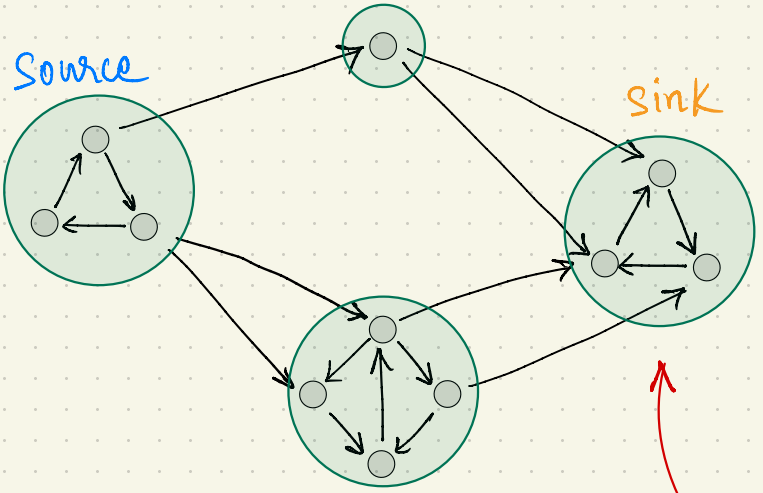
G



G^{rev}

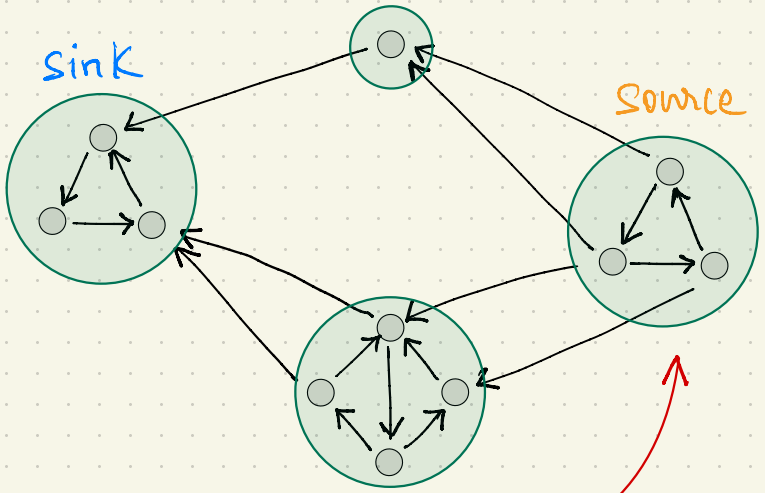


G



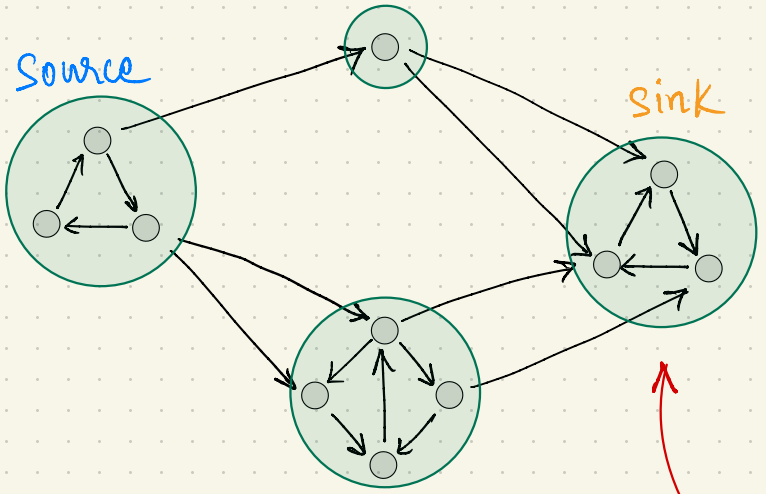
equivalently, here

G^{rev}



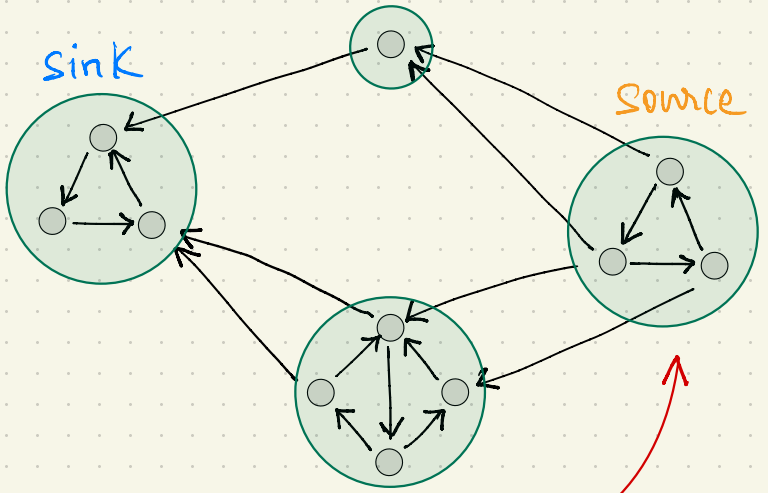
identify a vertex here

G



equivalently, here

G^{rev}

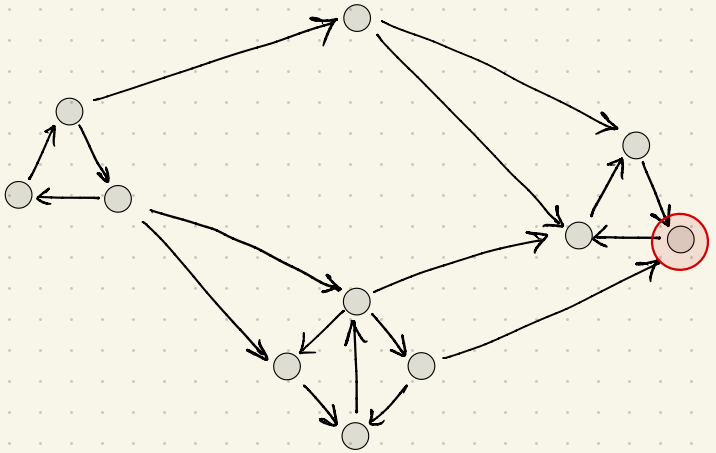


identify a vertex here

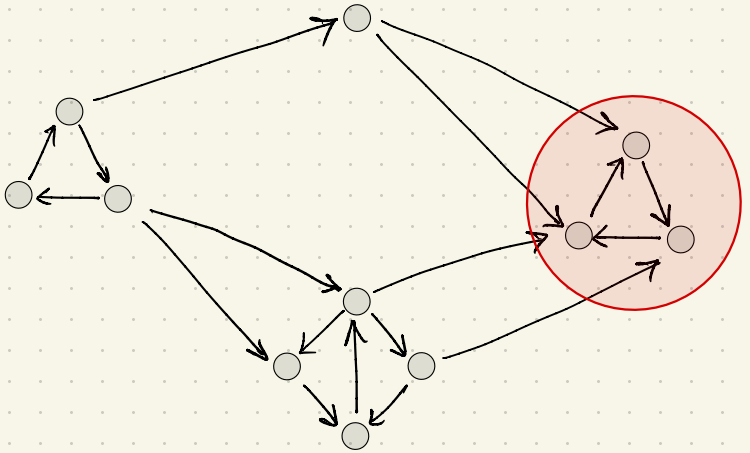
"Pluck out" a sink SCC of G and work our way backward.

(similar to topological ordering algo.)

G

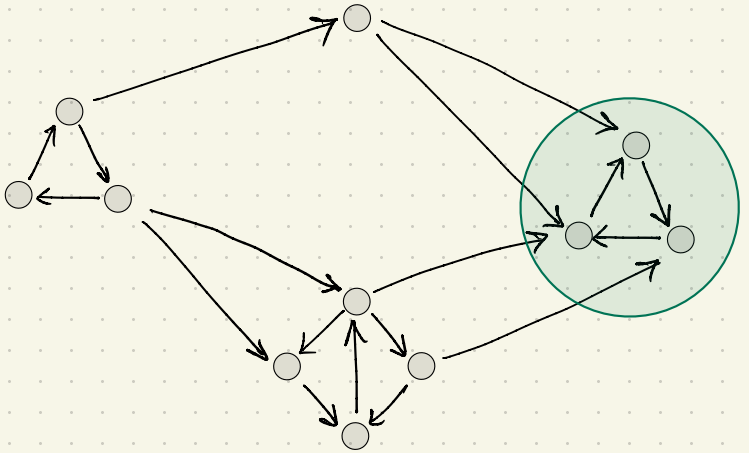


G



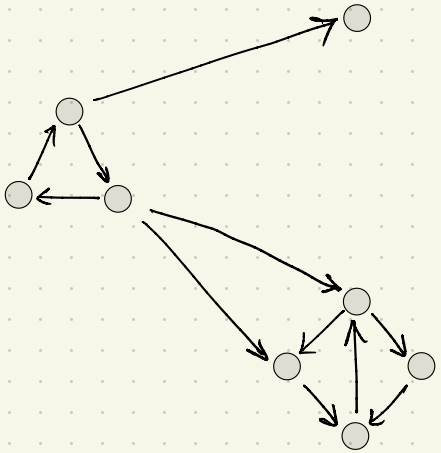
discover sink SCC via DFS

G

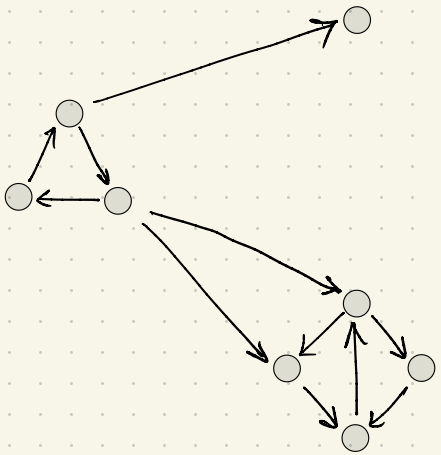


mark sink SCC as explored

G



G

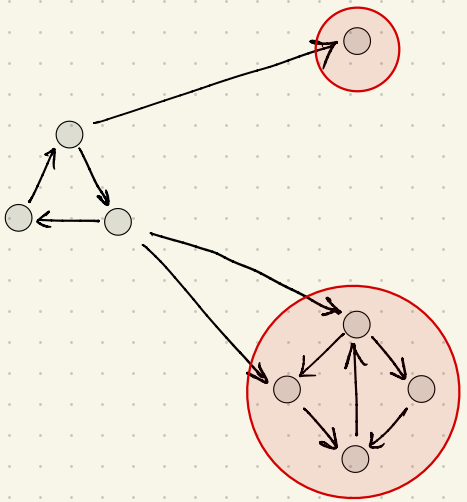


Key observation ⇒

among unexplored vertices, the "min f" label

should lie in one of the new sink SCCs

G

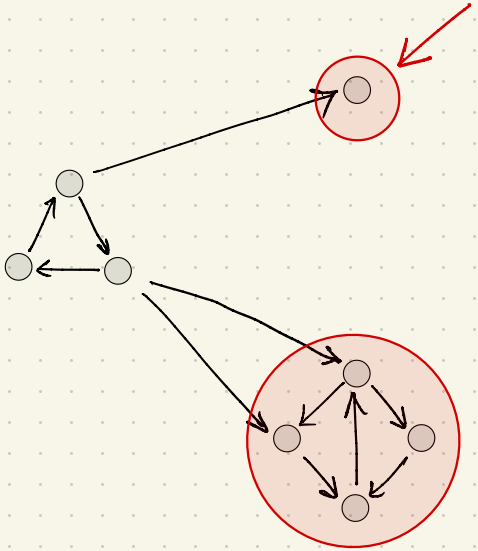


Key observation ⇒

among unexplored vertices, the "min f" label

should lie in one of the new sink SCCs

G

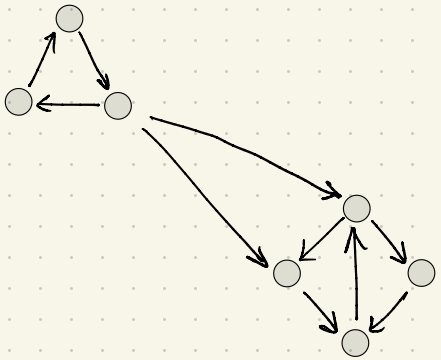


Key observation ⇒

among unexplored vertices, the "min f" label should lie in one of the new sink SCCs

"Pluck out" the sink SCC and repeat!

G

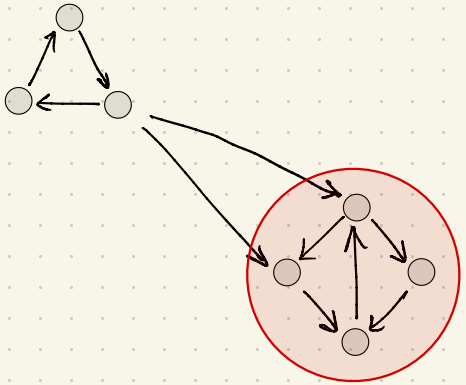


Key observation ⇒

among unexplored vertices, the "min f" label should lie in one of the new sink SCCs

"Pluck out" the sink SCC and repeat!

G

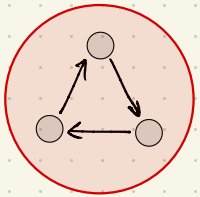


Key observation ⇒

among unexplored vertices, the "min f" label should lie in one of the new sink SCCs

"Pluck out" the sink SCC and repeat!

G



Key observation ⇒

among unexplored vertices, the "min f" label should lie in one of the new sink SCCs

"Pluck out" the sink SCC and repeat!

KOSARAJU's ALGORITHM

KOSARAJU's ALGORITHM

A two-pass algorithm to compute SCCs in $O(m+n)$ time.

KOSARAJU's ALGORITHM

A two-pass algorithm to compute SCCs in $O(m+n)$ time.

* Once on G^{rev} to find the "magical" labeling f

* Once on G to find the SCCs using f.

KOSARAJU's ALGORITHM

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

Output: all SCCs of G

KOSARAJU's ALGORITHM

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

Output: all SCCs of G

① G^{rev} := reverse edges of G

mark all vertices of G^{rev} as unexplored

KOSARAJU's ALGORITHM

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

Output: all SCCs of G

(1) G^{rev} := reverse edges of G
mark all vertices of G^{rev} as unexplored

(2) Compute labeling f via topological ordering algo on G^{rev}

KOSARAJU's ALGORITHM

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

output: all SCCs of G

(1) G^{rev} := reverse edges of G
mark all vertices of G^{rev} as unexplored

(2) Compute labeling f via topological ordering algo on G^{rev}

(3) mark all vertices of G as unexplored
global numSCC := 0

KOSARAJU's ALGORITHM

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

output: all SCCs of G

(1) G^{rev} := reverse edges of G

mark all vertices of G^{rev} as unexplored

(2) Compute labeling f via topological ordering algo on G^{rev}

(3) mark all vertices of G as unexplored

global numSCC := 0

for each v in increasing order of $f[v]$

|

KOSARAJU's ALGORITHM

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

output: all SCCs of G

(1) G^{rev} := reverse edges of G

mark all vertices of G^{rev} as unexplored

(2) Compute labeling f via topological ordering algo on G^{rev}

(3) mark all vertices of G as unexplored

global numSCC := 0

for each v in increasing order of $f[v]$

| if v is unexplored

| |

KOSARAJU's ALGORITHM

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

output: all SCCs of G

(1) G^{rev} := reverse edges of G
mark all vertices of G^{rev} as unexplored

(2) Compute labeling f via topological ordering algo on G^{rev}

(3) mark all vertices of G as unexplored
global numSCC := 0
for each v in increasing order of $f[v]$
    if v is unexplored
        increment numSCC by 1

KOSARAJU's ALGORITHM

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

output: all SCCs of G

(1) G^{rev} := reverse edges of G
mark all vertices of G^{rev} as unexplored

(2) Compute labeling f via topological ordering algo on G^{rev}

(3) mark all vertices of G as unexplored
global numSCC := 0
for each v in increasing order of $f[v]$
    if v is unexplored
        increment numSCC by 1
        DFS-SCC(G, v)

KOSARAJU's ALGORITHM

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

output: all SCCs of G

(1) G^{rev} := reverse edges of G
mark all vertices of G^{rev} as unexplored

(2) Compute labeling f via topological ordering algo on G^{rev}

(3) mark all vertices of G as unexplored
global numSCC := 0
for each v in increasing order of $f[v]$
    if v is unexplored
        increment numSCC by 1
        DFS-SCC(G, v)

DFS-SCC(G, v)

KOSARAJU's ALGORITHM

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

output: all SCCs of G

(1) G^{rev} := reverse edges of G
mark all vertices of G^{rev} as unexplored

(2) Compute labeling f via topological ordering algo on G^{rev}

(3) mark all vertices of G as unexplored
global numSCC := 0
for each v in increasing order of $f[v]$
    if v is unexplored
        increment numSCC by 1
        DFS-SCC(G, v)

DFS-SCC (G, v)

// performs DFS and assigns SCC values

KOSARAJU's ALGORITHM

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

output: all SCCs of G

(1) G^{rev} := reverse edges of G

mark all vertices of G^{rev} as unexplored

(2) Compute labeling f via topological ordering algo on G^{rev}

(3) mark all vertices of G as unexplored

global numSCC := 0

for each v in increasing order of $f[v]$

    if v is unexplored

        increment numSCC by 1

        DFS-SCC(G, v)

DFS-SCC (G, v)

// performs DFS and assigns SCC values

mark v as explored

KOSARAJU's ALGORITHM

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

output: all SCCs of G

(1) G^{rev} := reverse edges of G
mark all vertices of G^{rev} as unexplored

(2) Compute labeling f via topological ordering algo on G^{rev}

(3) mark all vertices of G as unexplored
global numSCC := 0
for each v in increasing order of $f[v]$
    if v is unexplored
        increment numSCC by 1
        DFS-SCC(G, v)

DFS-SCC (G, v)

// performs DFS and assigns SCC values

mark v as explored

scc(v) := numSCC

KOSARAJU's ALGORITHM

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

output: all SCCs of G

(1) G^{rev} := reverse edges of G
mark all vertices of G^{rev} as unexplored

(2) Compute labeling f via topological ordering algo on G^{rev}

(3) mark all vertices of G as unexplored
global numSCC := 0
for each v in increasing order of $f[v]$
    if v is unexplored
        increment numSCC by 1
        DFS-SCC(G, v)

DFS-SCC (G, v)

// performs DFS and assigns SCC values

mark v as explored

scc(v) := numSCC

for each edge (v, w)

KOSARAJU's ALGORITHM

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

output: all SCCs of G

(1) G^{rev} := reverse edges of G
mark all vertices of G^{rev} as unexplored

(2) Compute labeling f via topological ordering algo on G^{rev}

(3) mark all vertices of G as unexplored
global numSCC := 0
for each v in increasing order of $f[v]$
    if v is unexplored
        increment numSCC by 1
        DFS-SCC(G, v)

DFS-SCC(G, v)

// performs DFS and assigns SCC values

mark v as explored

scc(v) := numSCC

for each edge (v, w)
    if w is unexplored
        DFS-SCC(G, w)

KOSARAJU's ALGORITHM

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

output: all SCCs of G

(1) G^{rev} := reverse edges of G

mark all vertices of G^{rev} as unexplored

(2) Compute labeling f via topological ordering algo on G^{rev}

(3) mark all vertices of G as unexplored

global numSCC := 0

for each v in increasing order of $f[v]$

  if v is unexplored

    increment numSCC by 1

    DFS-SCC(G, v)

DFS-SCC(G, v)

// performs DFS and assigns SCC values

mark v as explored

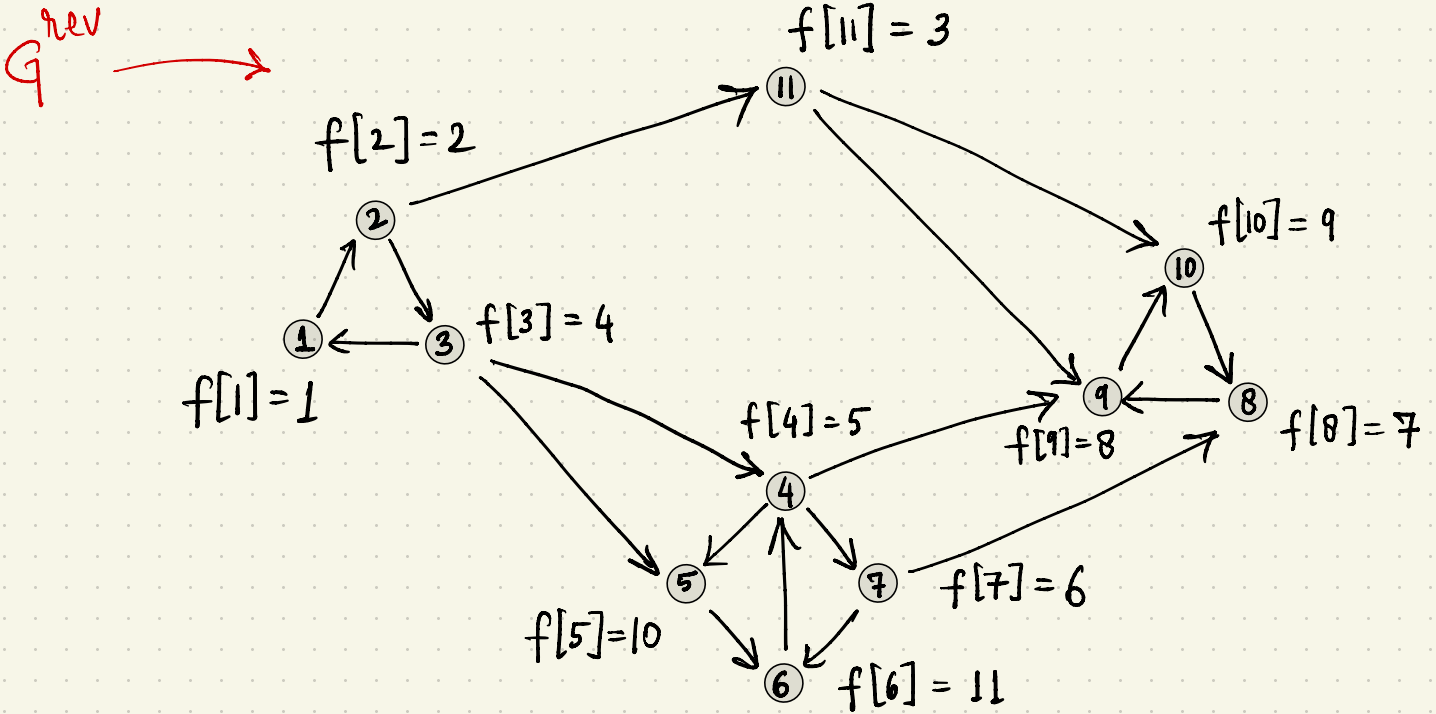
scc(v) := numSCC

for each edge (v, w)

  if w is unexplored

    DFS-SCC(G, w)

KOSARAJU's ALGORITHM



KOSARAJU's ALGORITHM

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

output: all SCCs of G

(1) G^{rev} := reverse edges of G
mark all vertices of G^{rev} as unexplored

(2) Compute labeling f via topological ordering algo on G^{rev}

(3) mark all vertices of G as unexplored
global numSCC := 0
for each v in increasing order of $f[v]$
    if v is unexplored
        increment numSCC by 1
        DFS-SCC(G, v)

DFS-SCC (G, v)

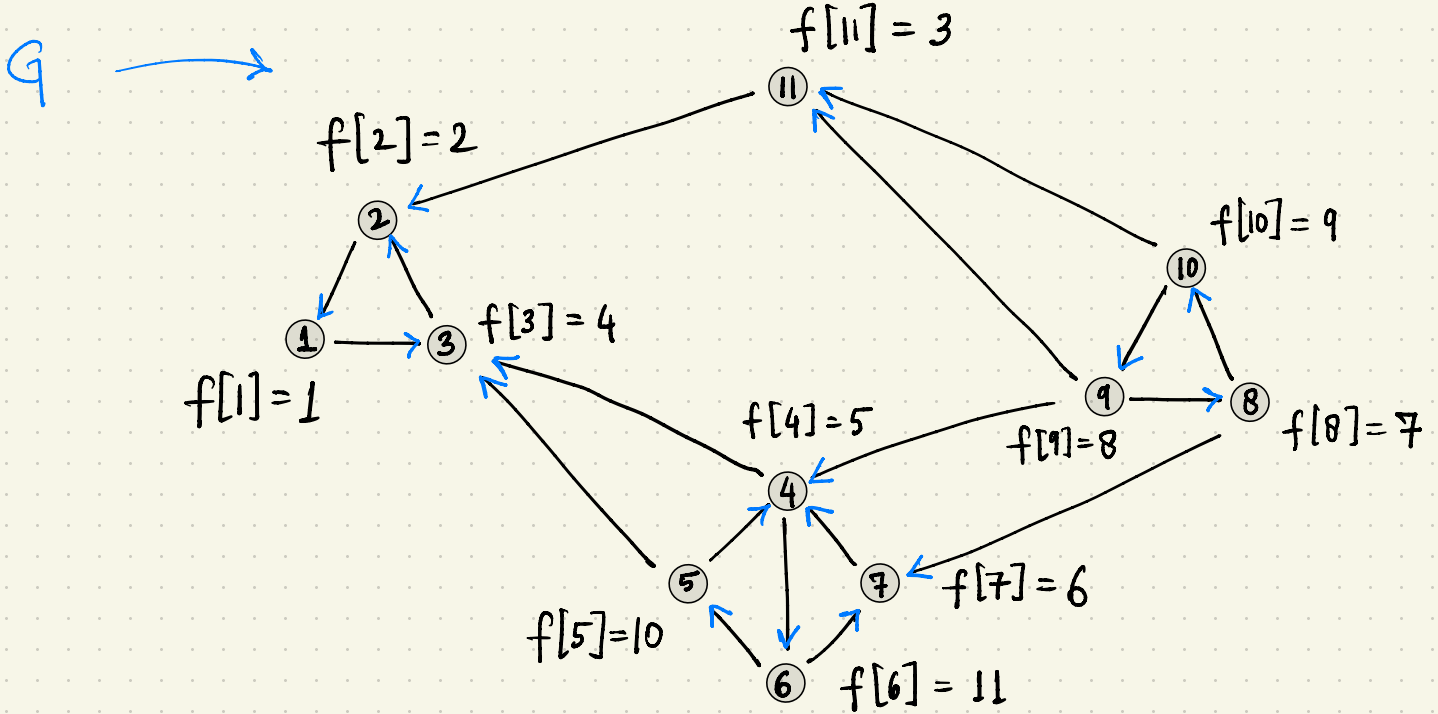
// performs DFS and assigns SCC values

mark v as explored

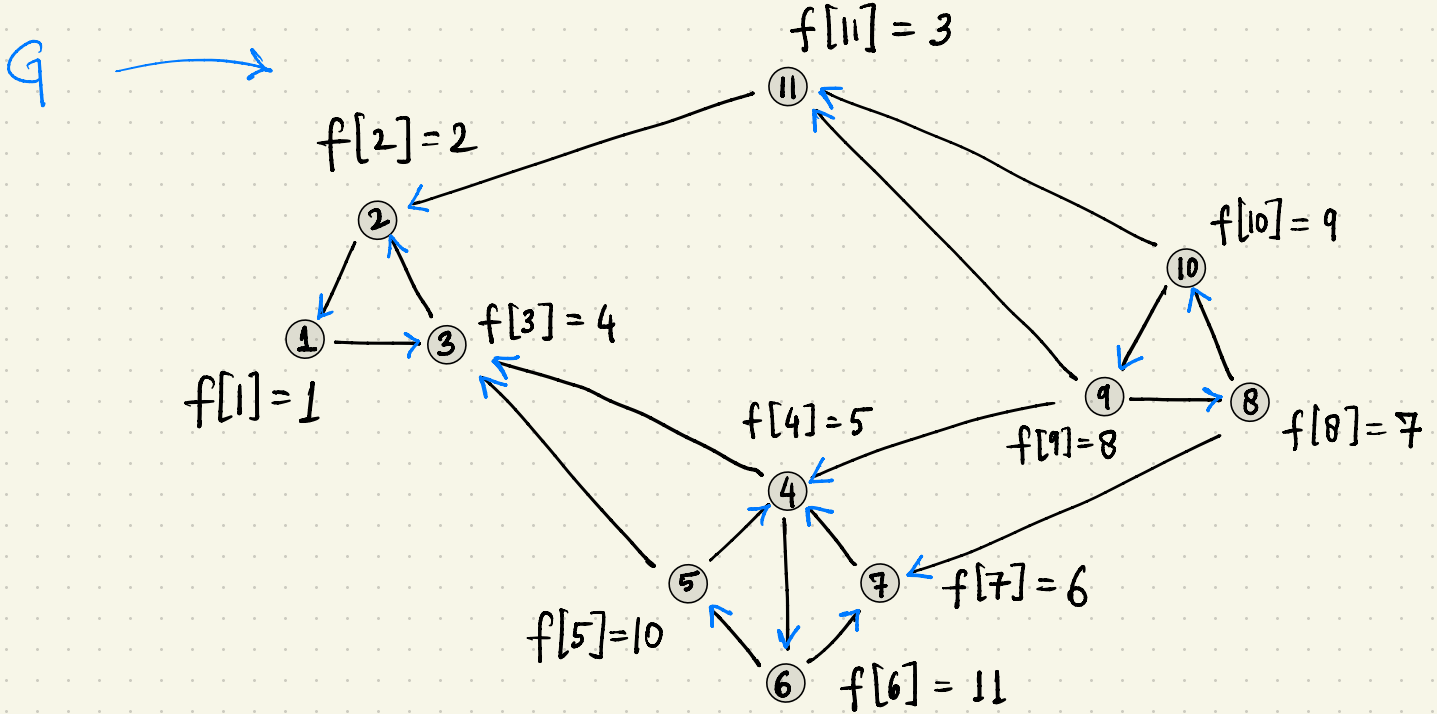
scc(v) := numSCC

for each edge (v, w)
    if w is unexplored
        DFS-SCC (G, w)

KOSARAJU's ALGORITHM



KOSARAJU's ALGORITHM



Recall: f is computed with respect to G^{rev}.

KOSARAJU's ALGORITHM

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

output: all SCCs of G

(1) G^{rev} := reverse edges of G
mark all vertices of G^{rev} as unexplored

(2) Compute labeling f via topological ordering algo on G^{rev}

(3) mark all vertices of G as unexplored
global numSCC := 0
for each v in increasing order of $f[v]$
    if v is unexplored
        increment numSCC by 1
        DFS-SCC(G, v)

DFS-SCC(G, v)

// performs DFS and assigns SCC values

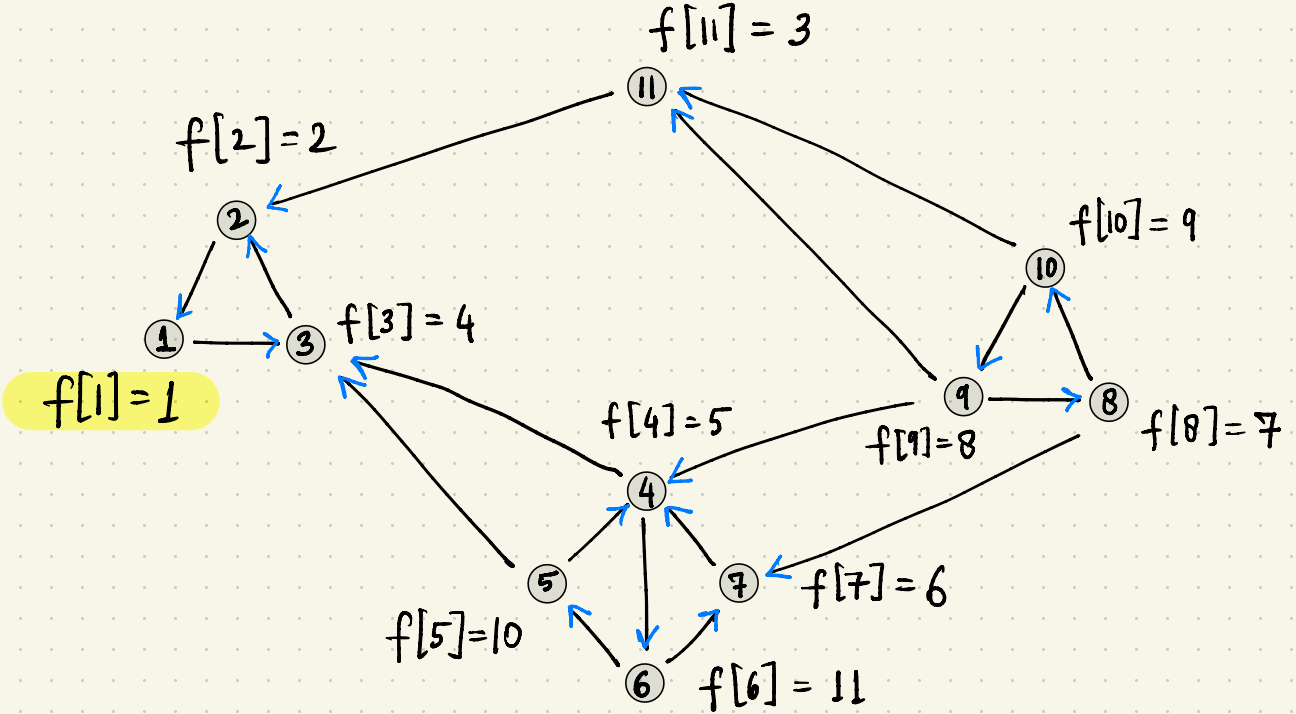
mark v as explored

scc(v) := numSCC

for each edge (v, w)
    if w is unexplored
        DFS-SCC(G, w)

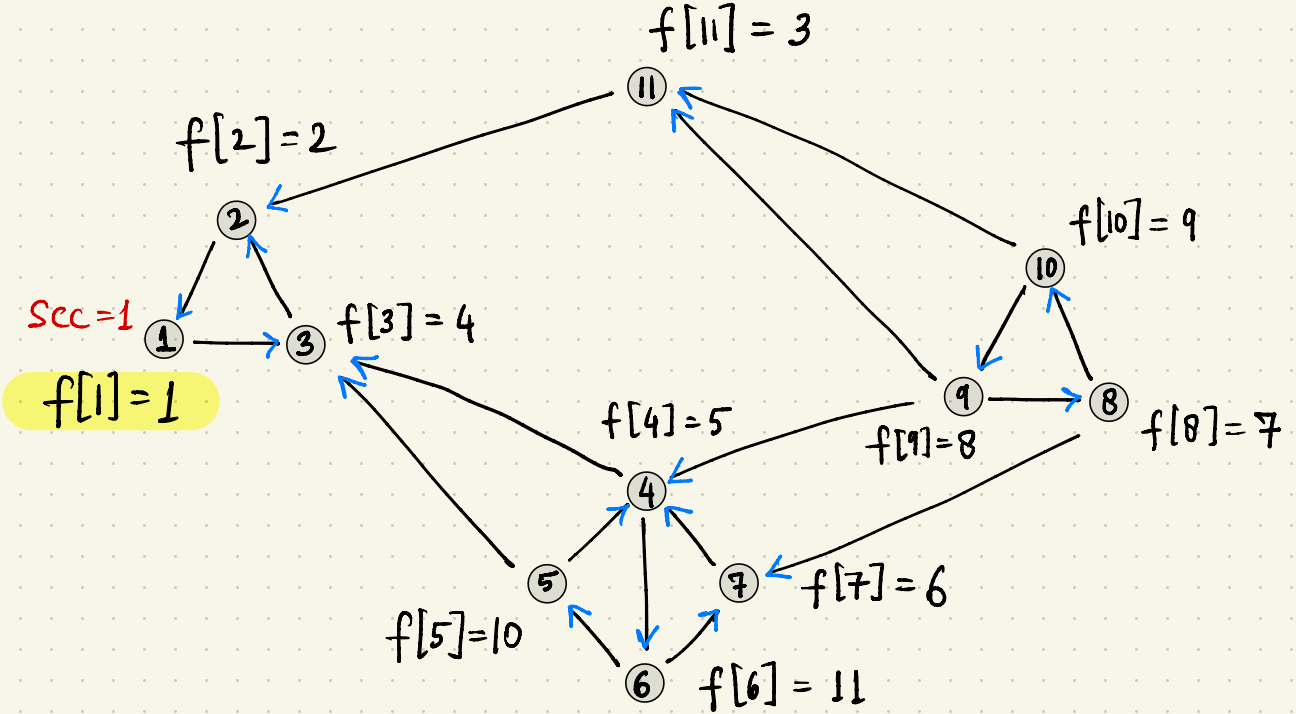
KOSARAJU's ALGORITHM

G



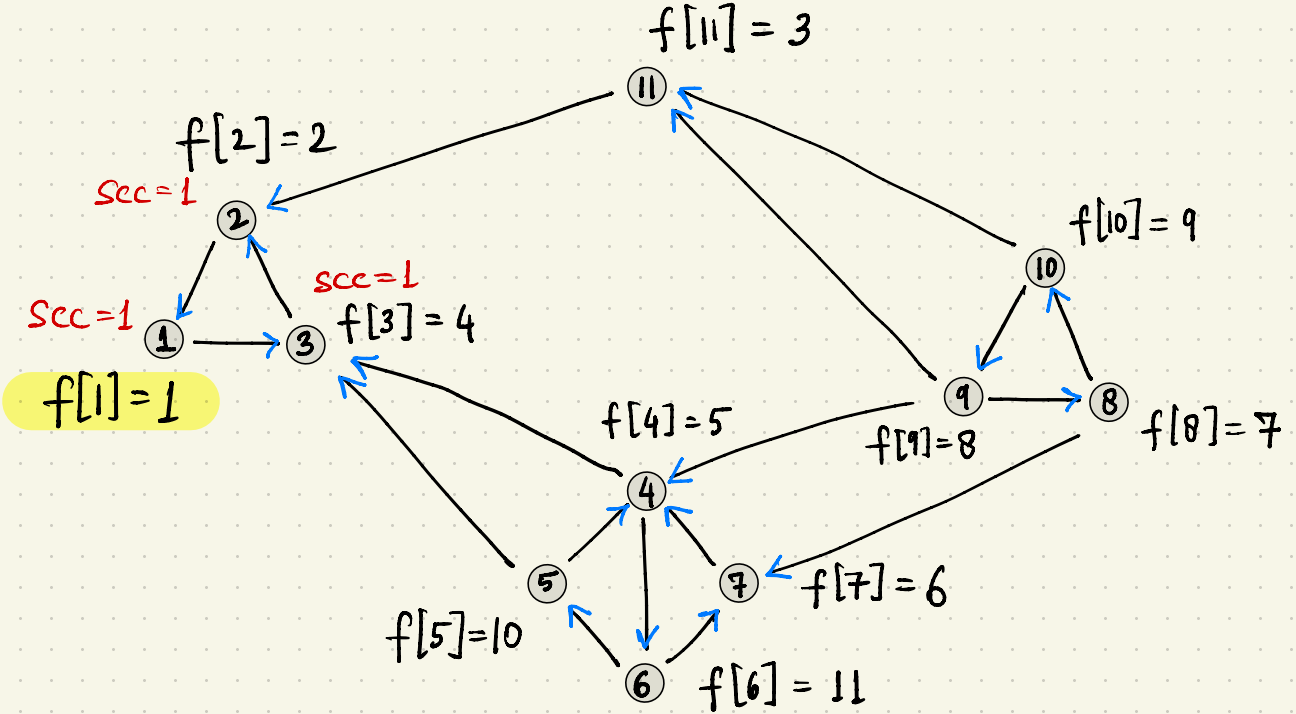
KOSARAJU's ALGORITHM

G



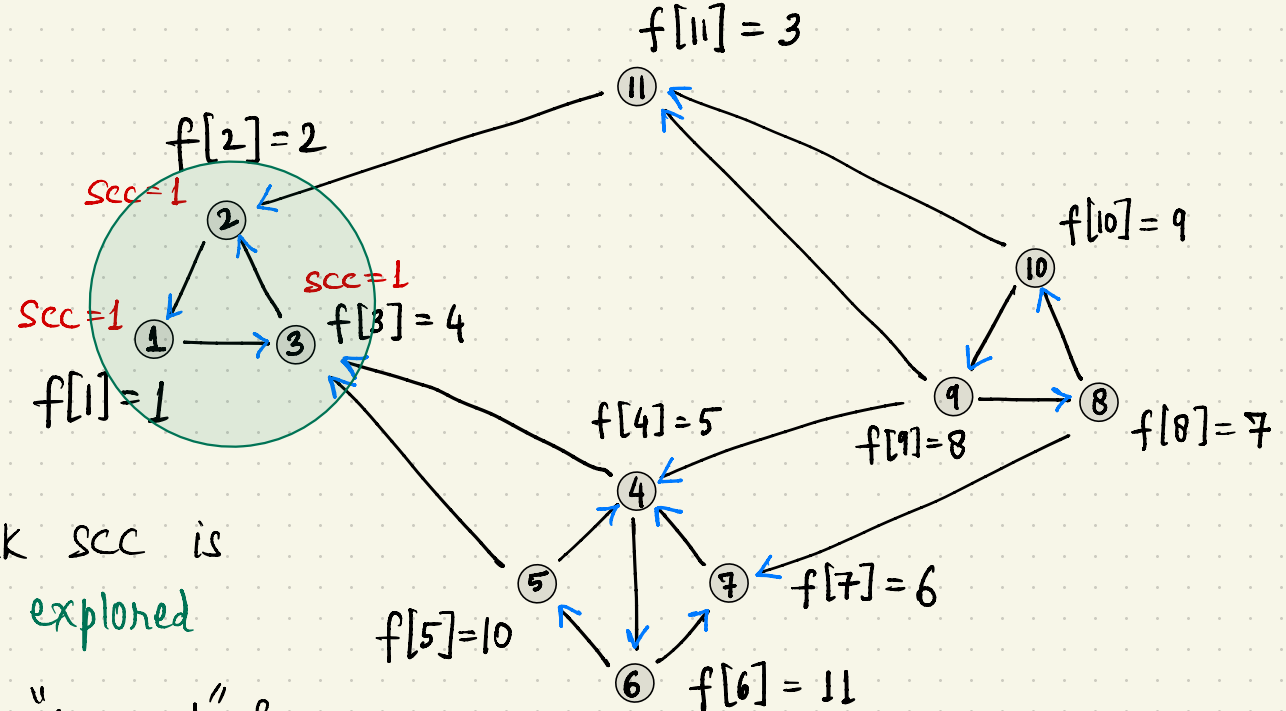
KOSARAJU's ALGORITHM

G



KOSARAJU's ALGORITHM

G



The sink SCC is now explored

(effectively "removed" for the rest of the algorithm)

KOSARAJU's ALGORITHM

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

output: all SCCs of G

(1) G^{rev} := reverse edges of G
mark all vertices of G^{rev} as unexplored

(2) Compute labeling f via topological ordering algo on G^{rev}

(3) mark all vertices of G as unexplored
global numSCC := 0
for each v in increasing order of $f[v]$
    if v is unexplored
        increment numSCC by 1
        DFS-SCC(G, v)

DFS-SCC(G, v)

// performs DFS and assigns SCC values

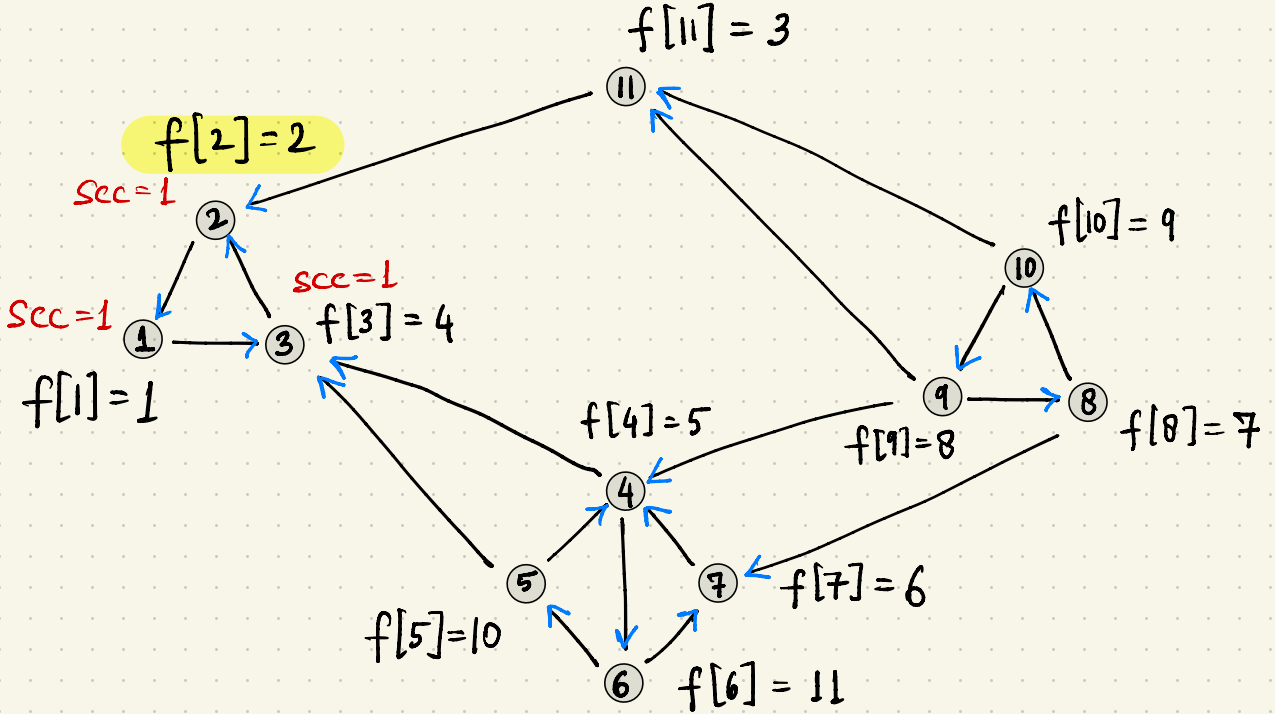
mark v as explored

scc(v) := numSCC

for each edge (v, w)
    if w is unexplored
        DFS-SCC(G, w)

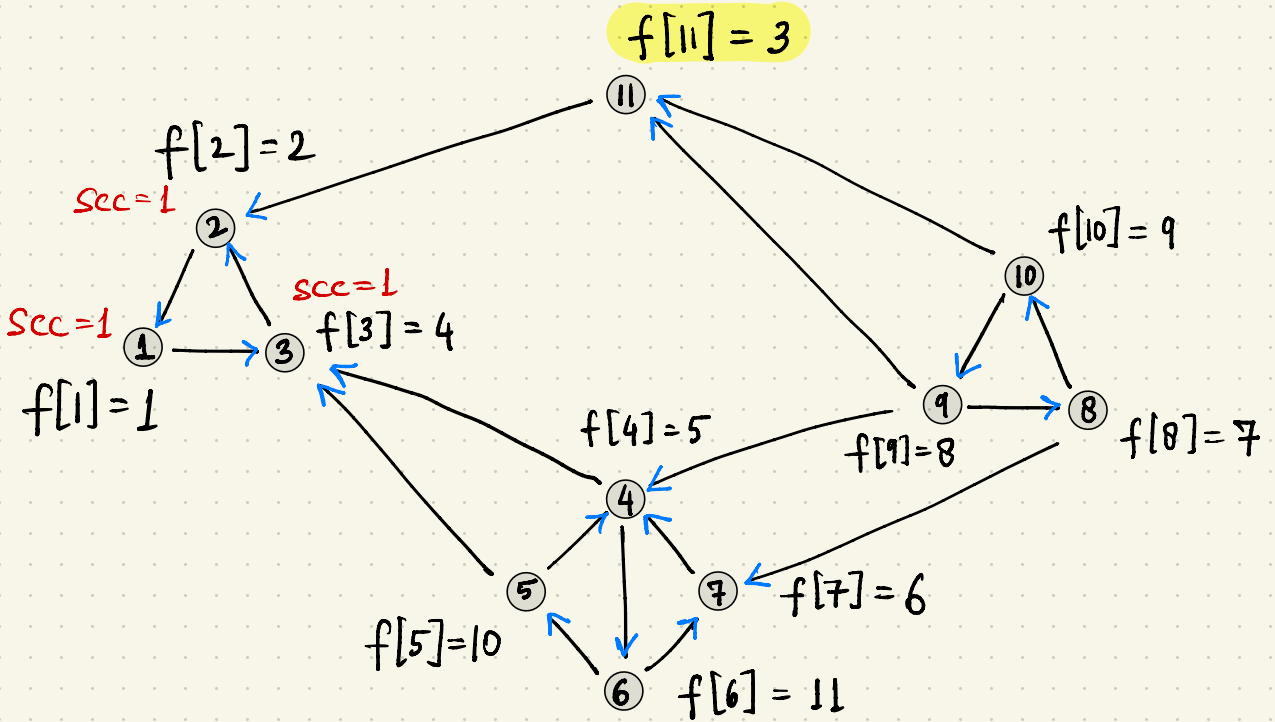
KOSARAJU's ALGORITHM

G



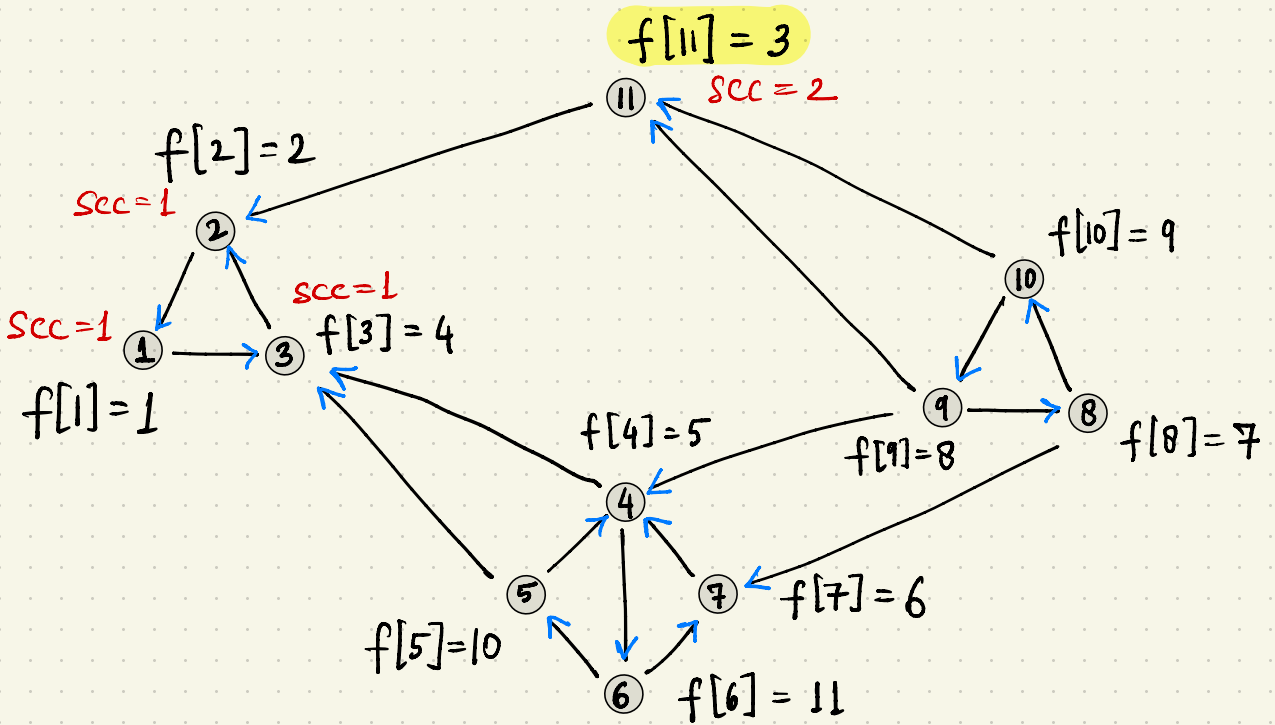
KOSARAJU's ALGORITHM

G



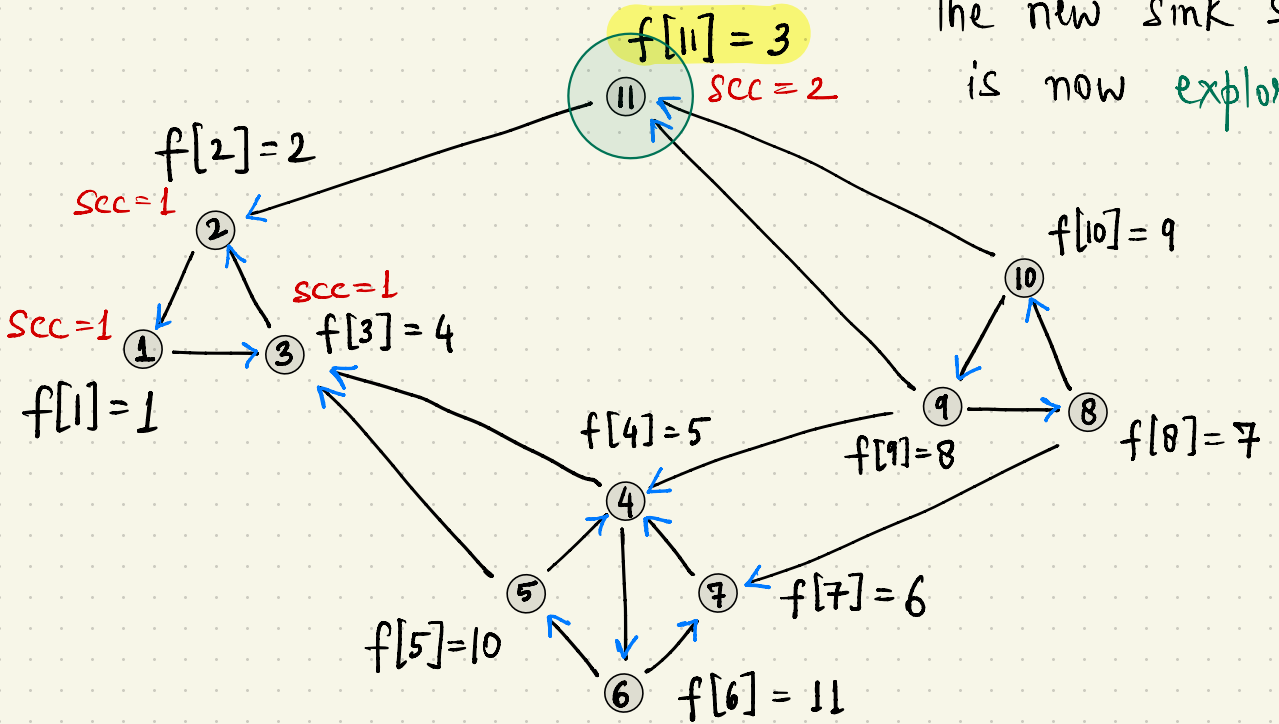
KOSARAJU's ALGORITHM

G



KOSARAJU's ALGORITHM

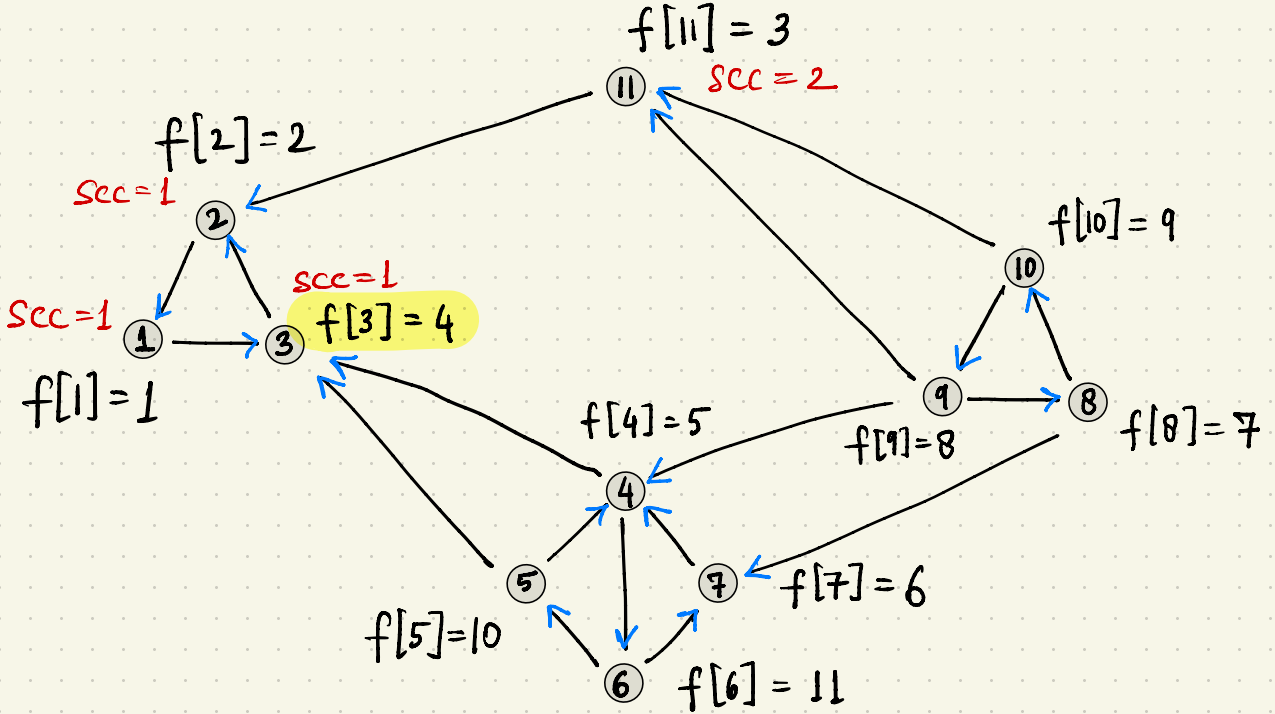
G



The "new" sink SCC is now explored

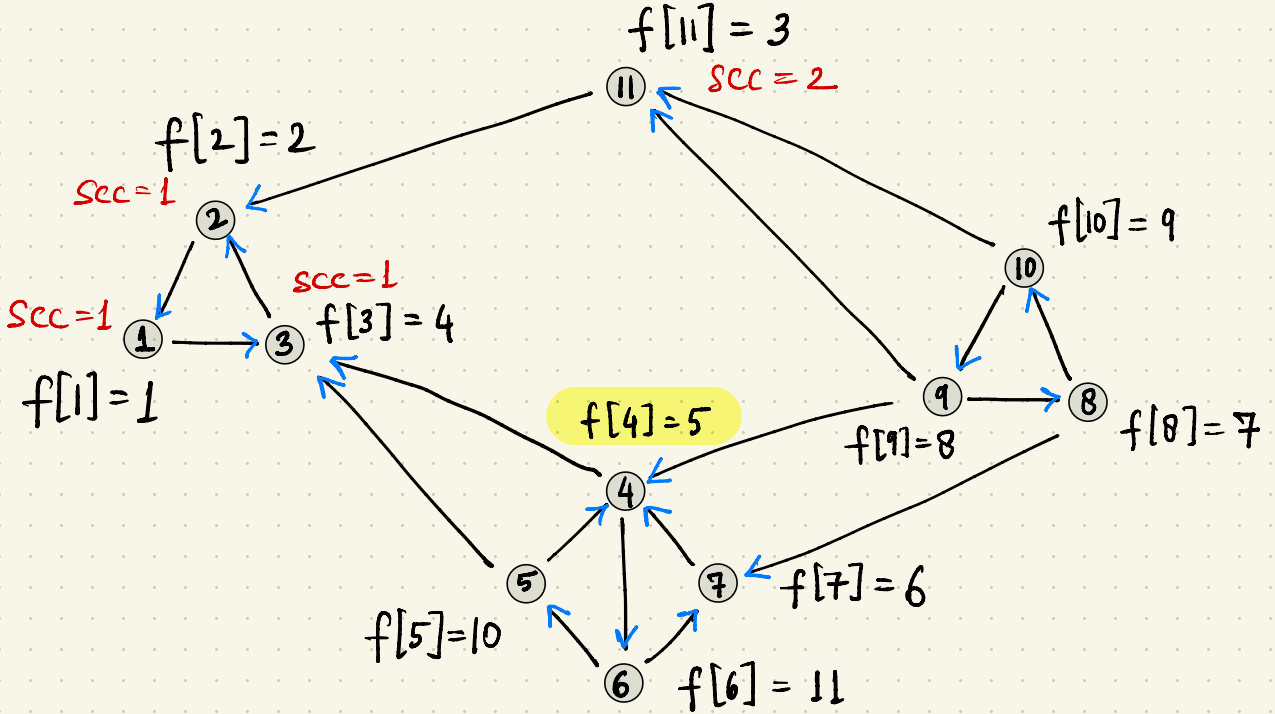
KOSARAJU's ALGORITHM

G



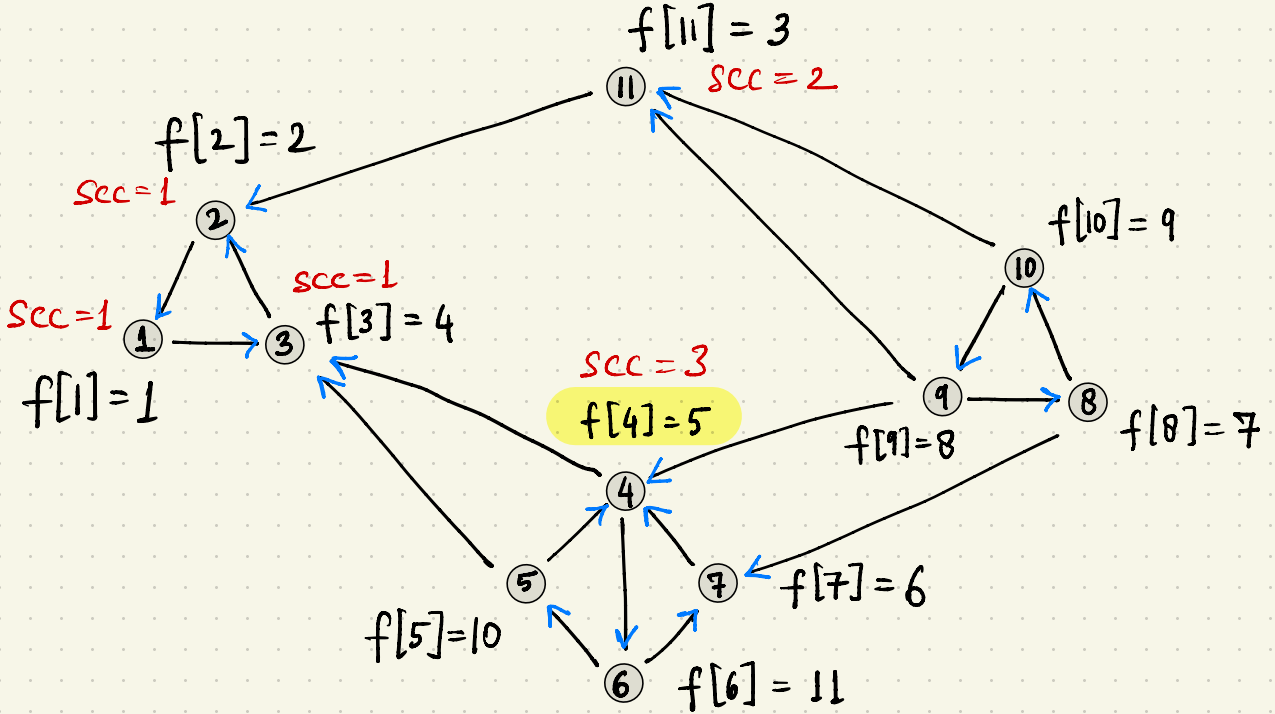
KOSARAJU's ALGORITHM

G



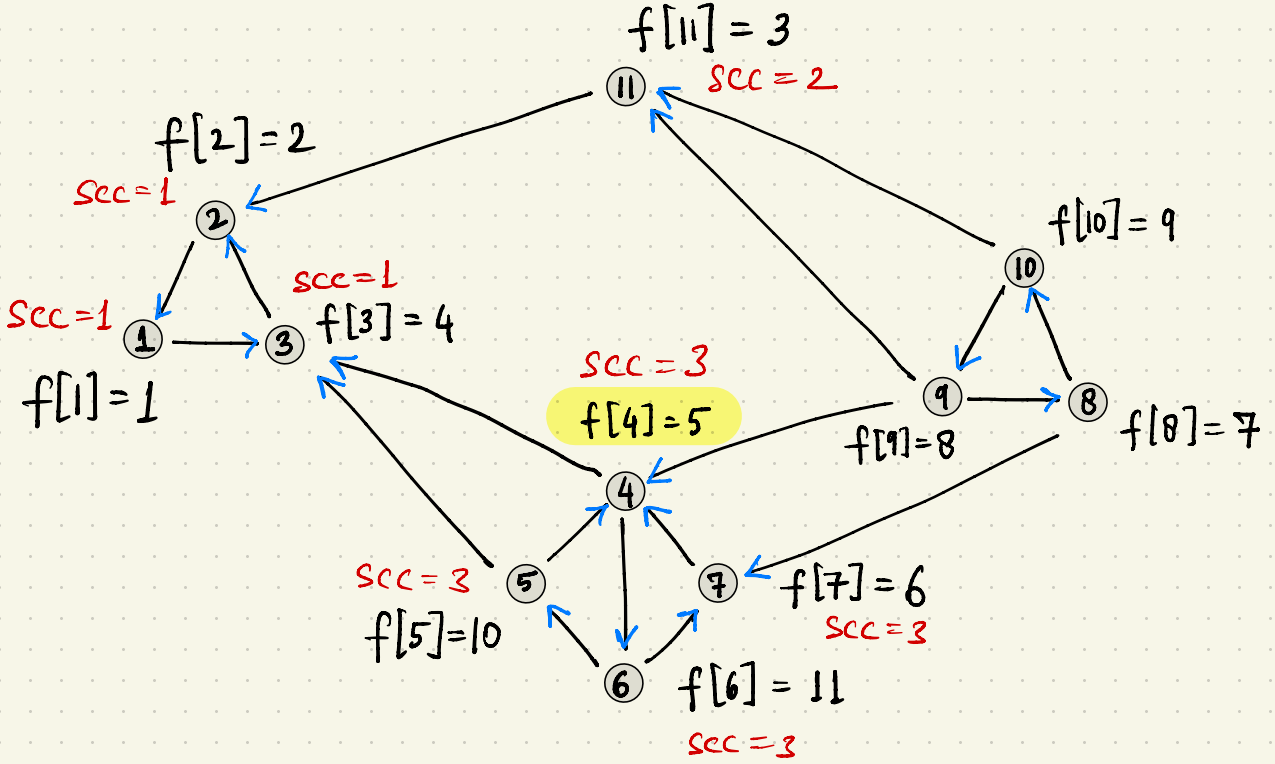
KOSARAJU's ALGORITHM

G



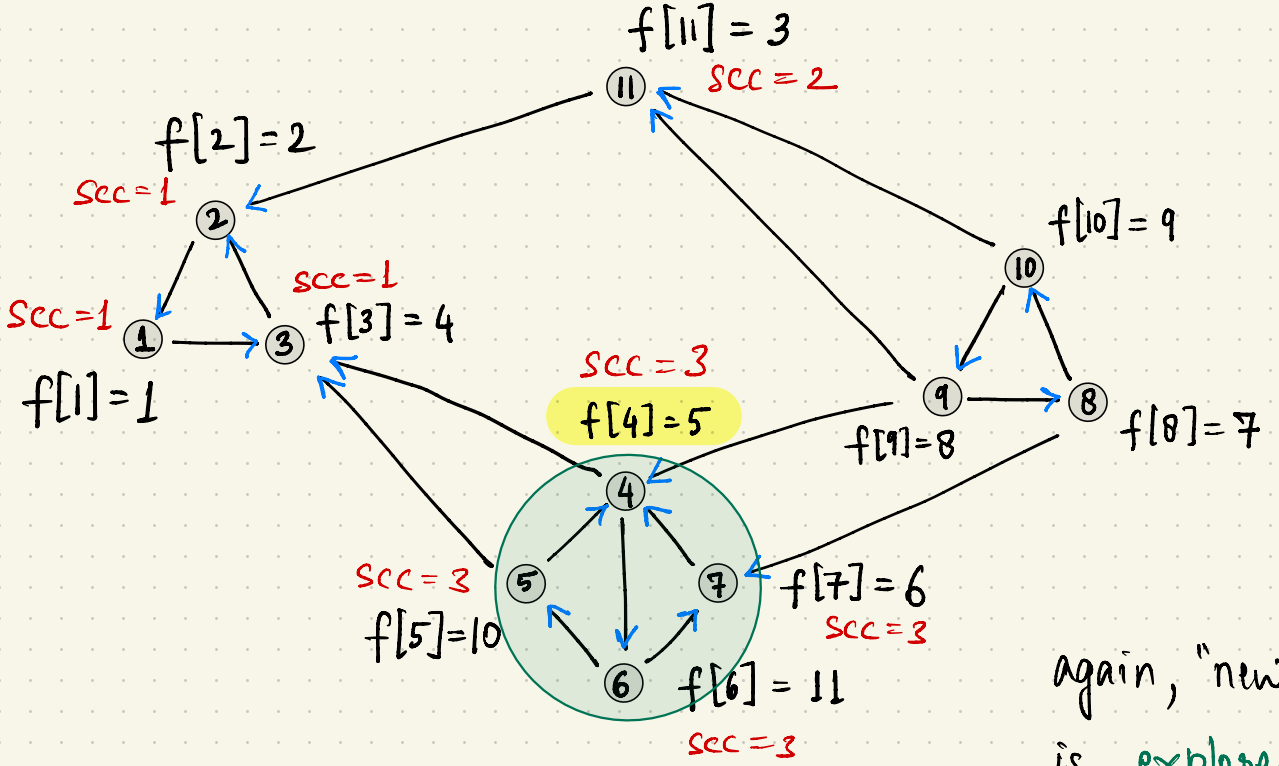
KOSARAJU's ALGORITHM

G



KOSARAJU's ALGORITHM

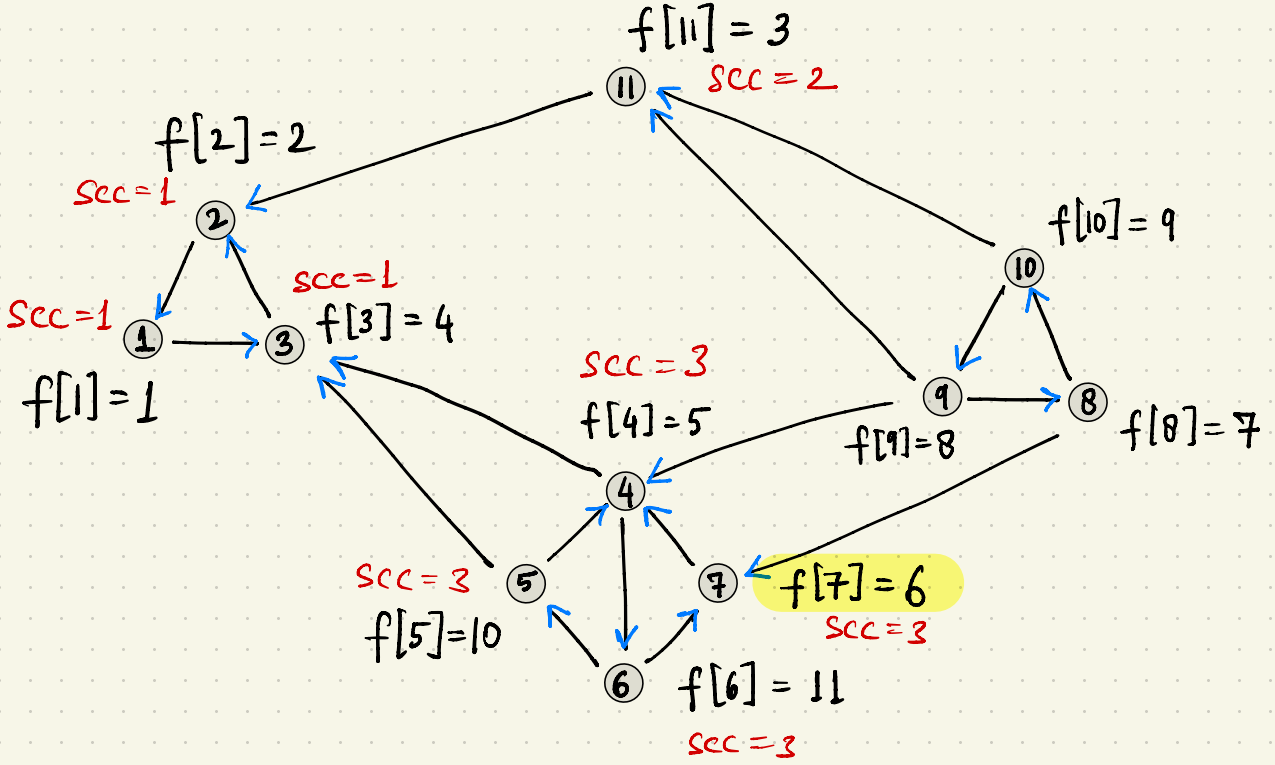
G



again, "new" sink scc is explored

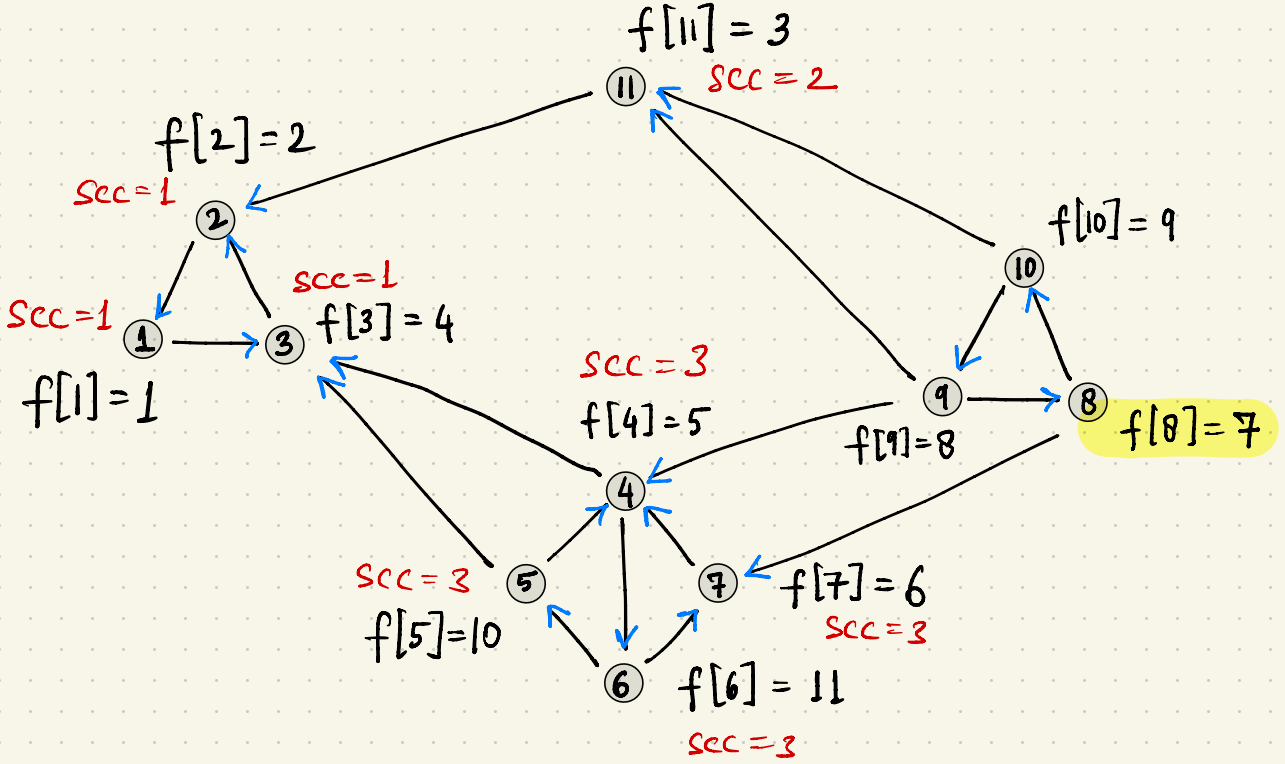
KOSARAJU's ALGORITHM

G



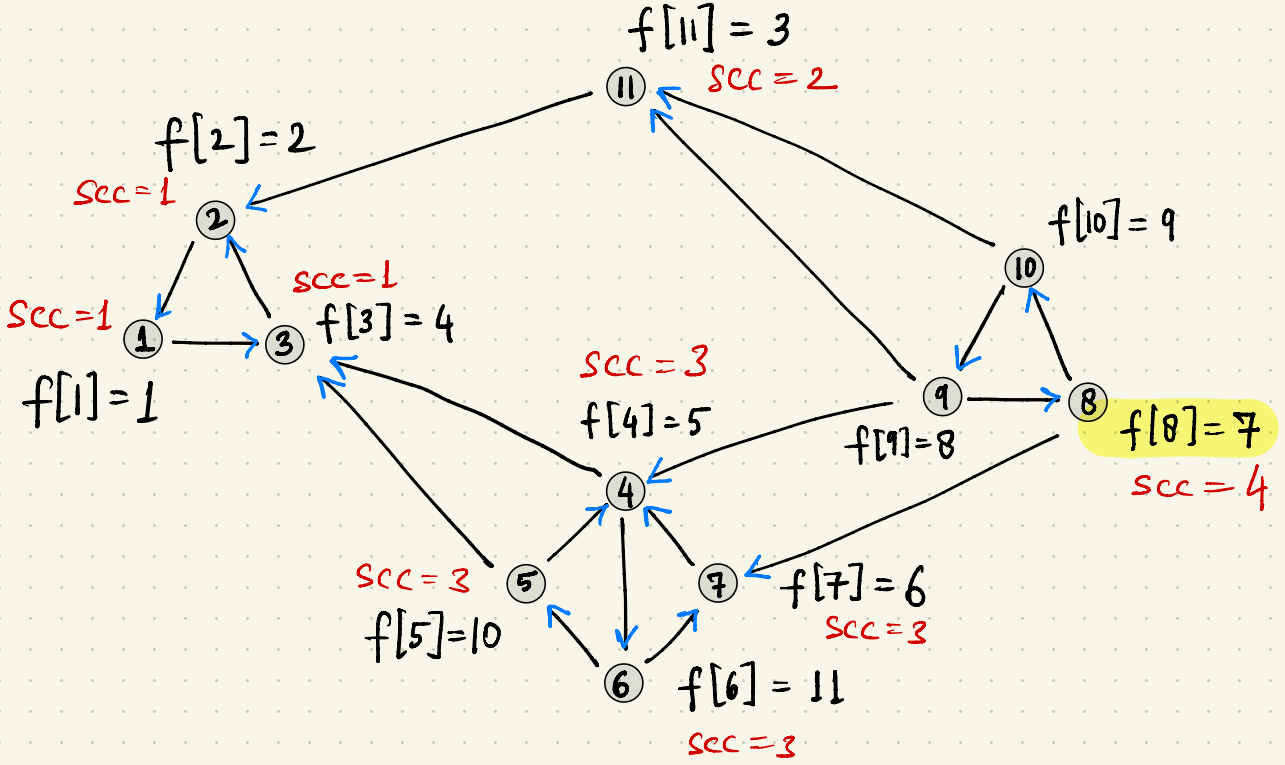
KOSARAJU's ALGORITHM

G



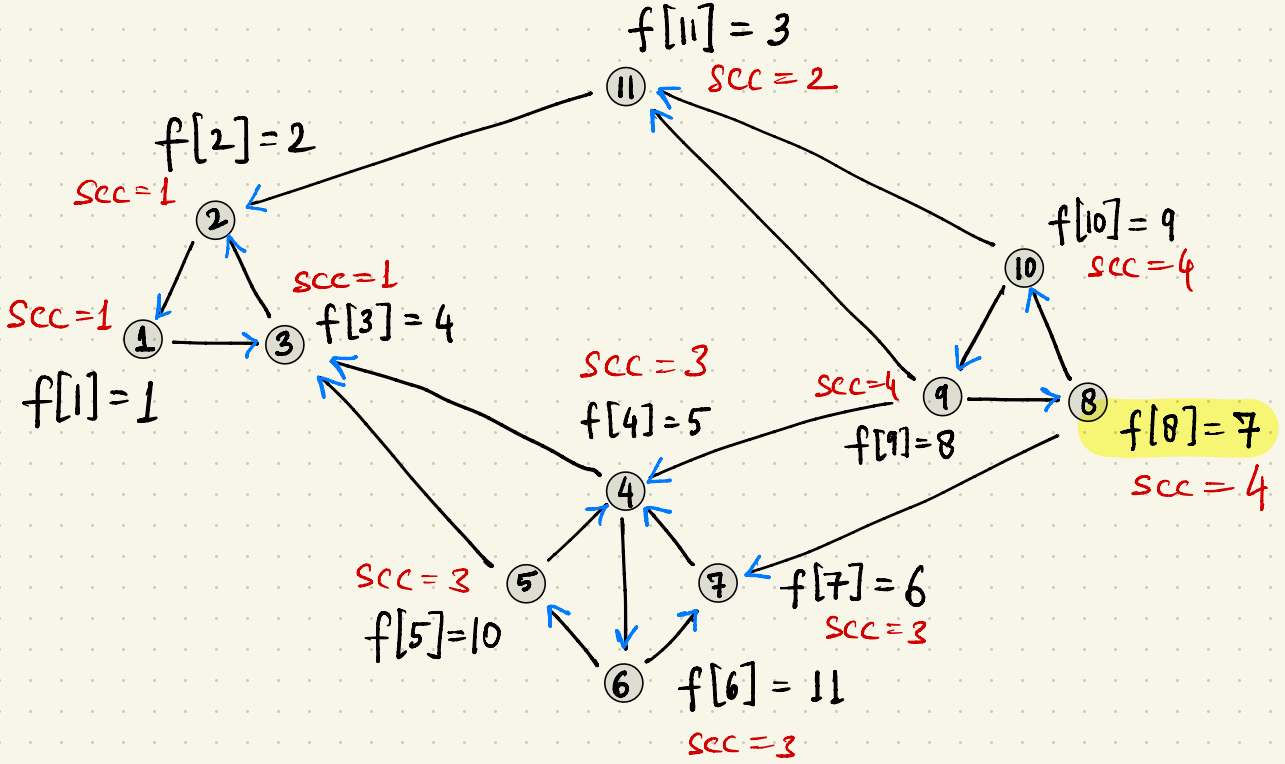
KOSARAJU's ALGORITHM

G



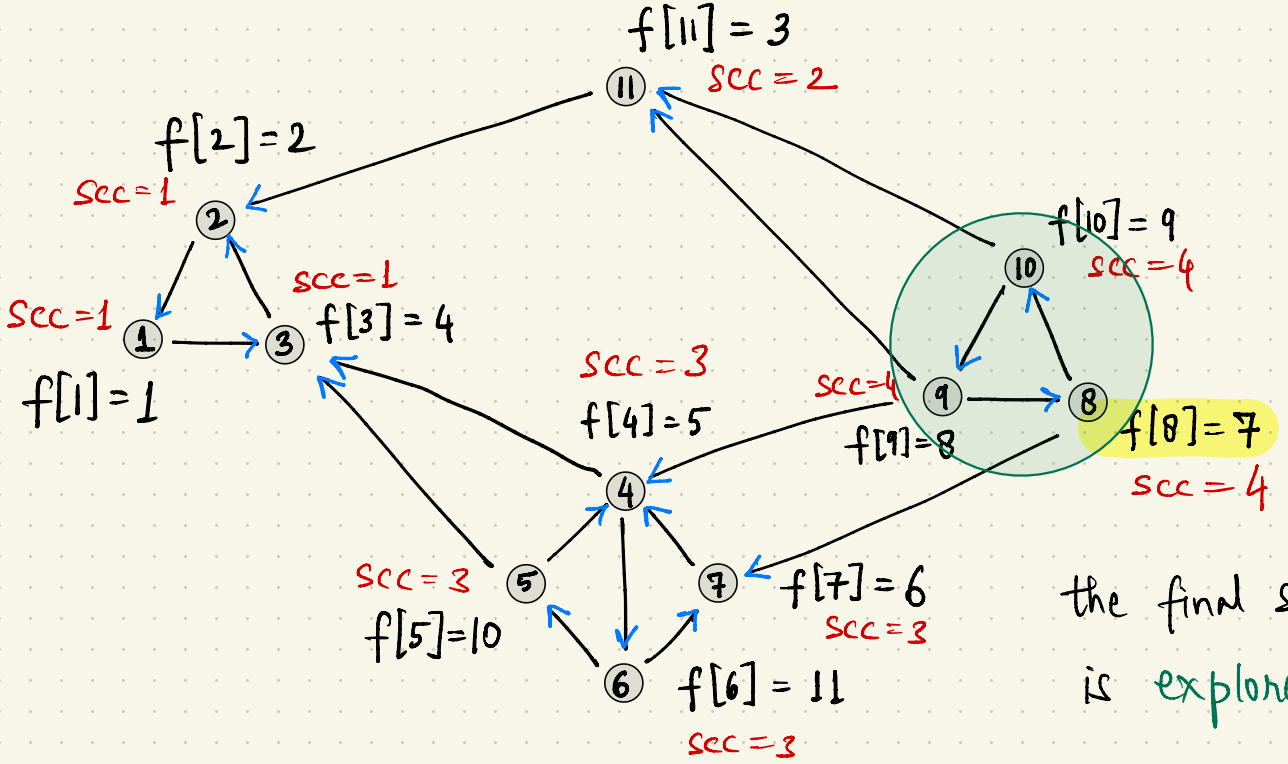
KOSARAJU's ALGORITHM

G



KOSARAJU's ALGORITHM

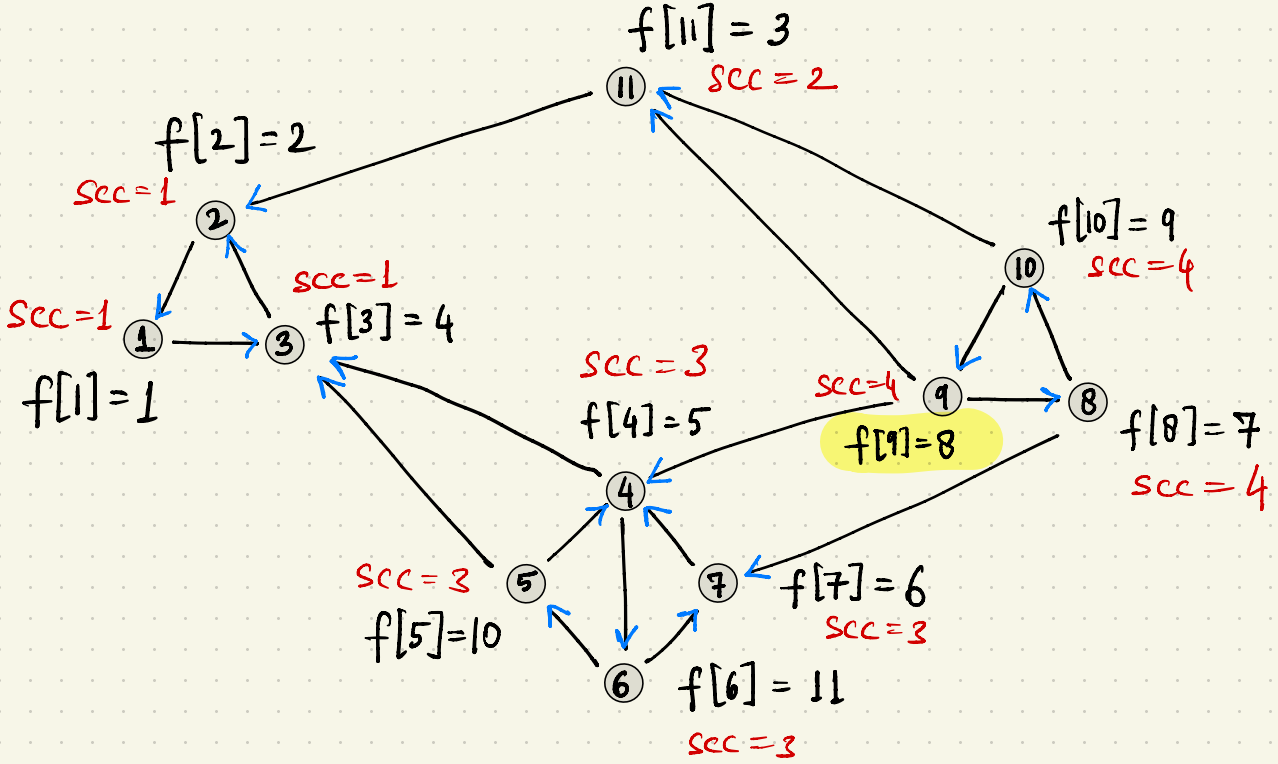
G



the final sink SCC is explored

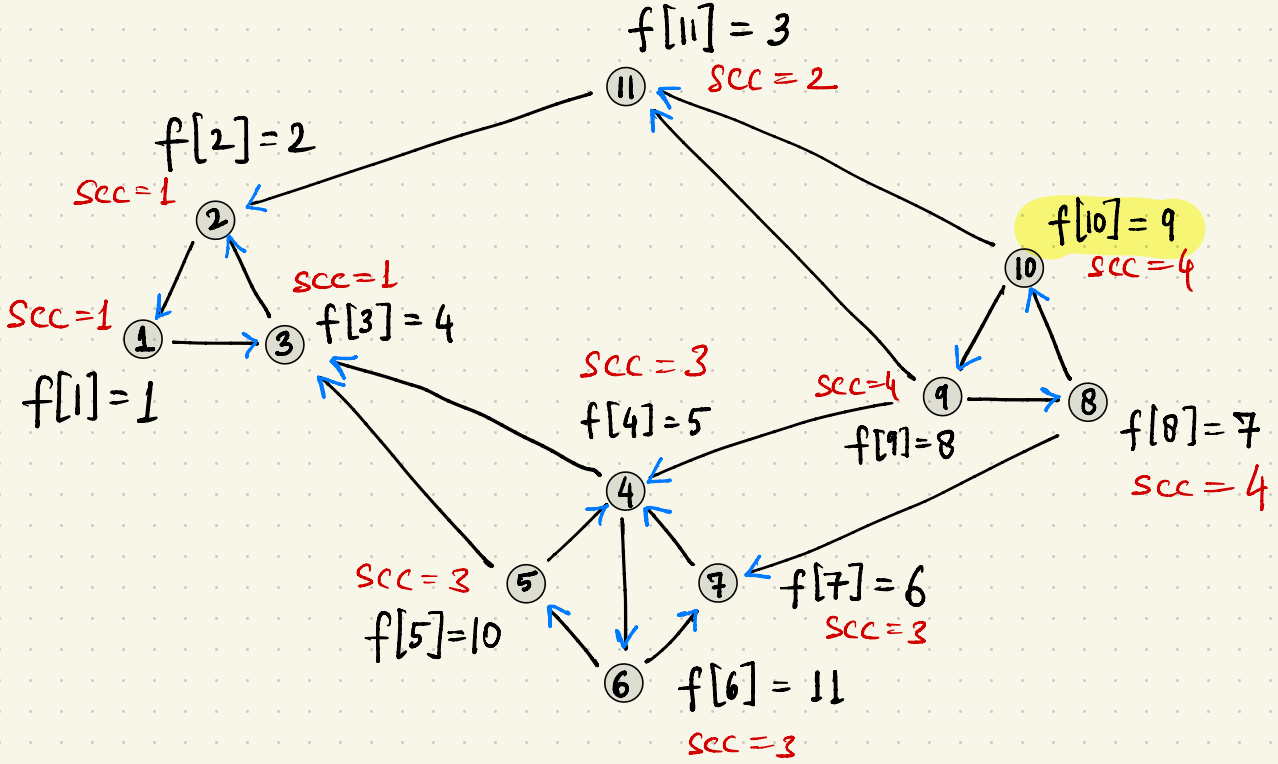
KOSARAJU's ALGORITHM

G



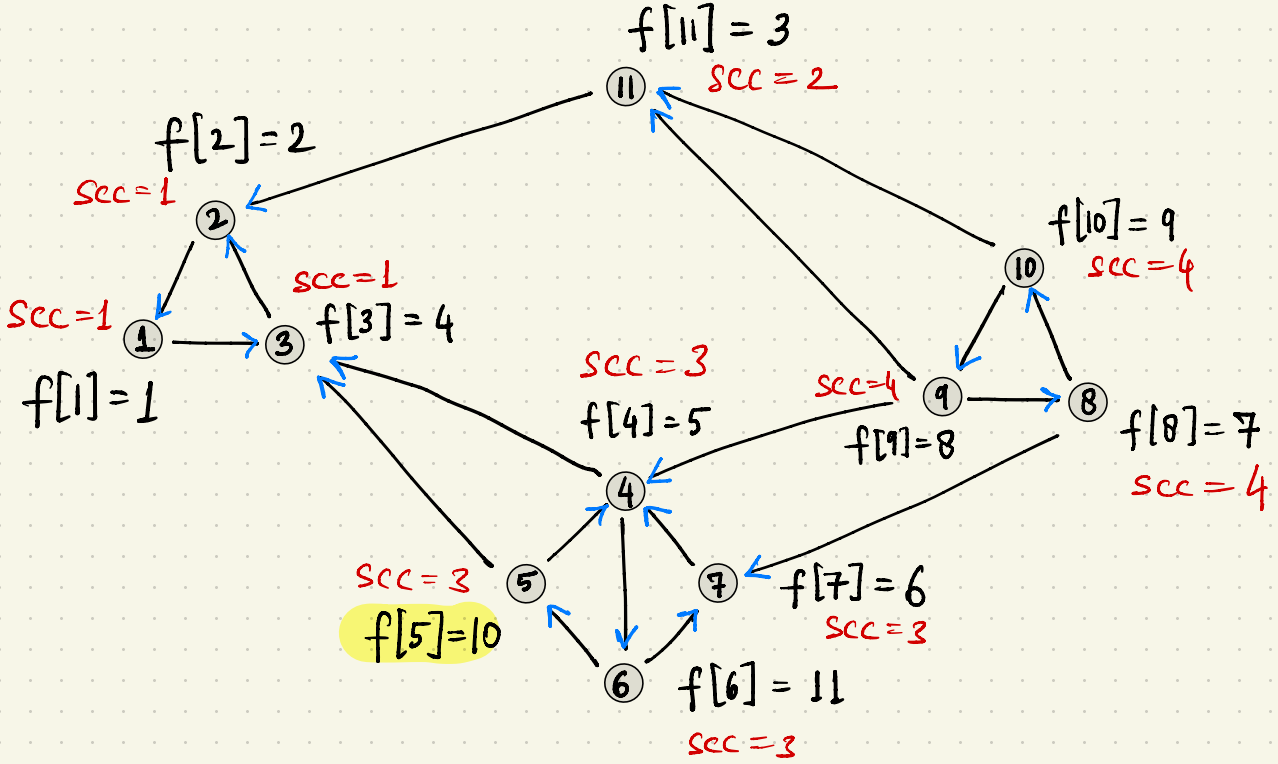
KOSARAJU's ALGORITHM

G



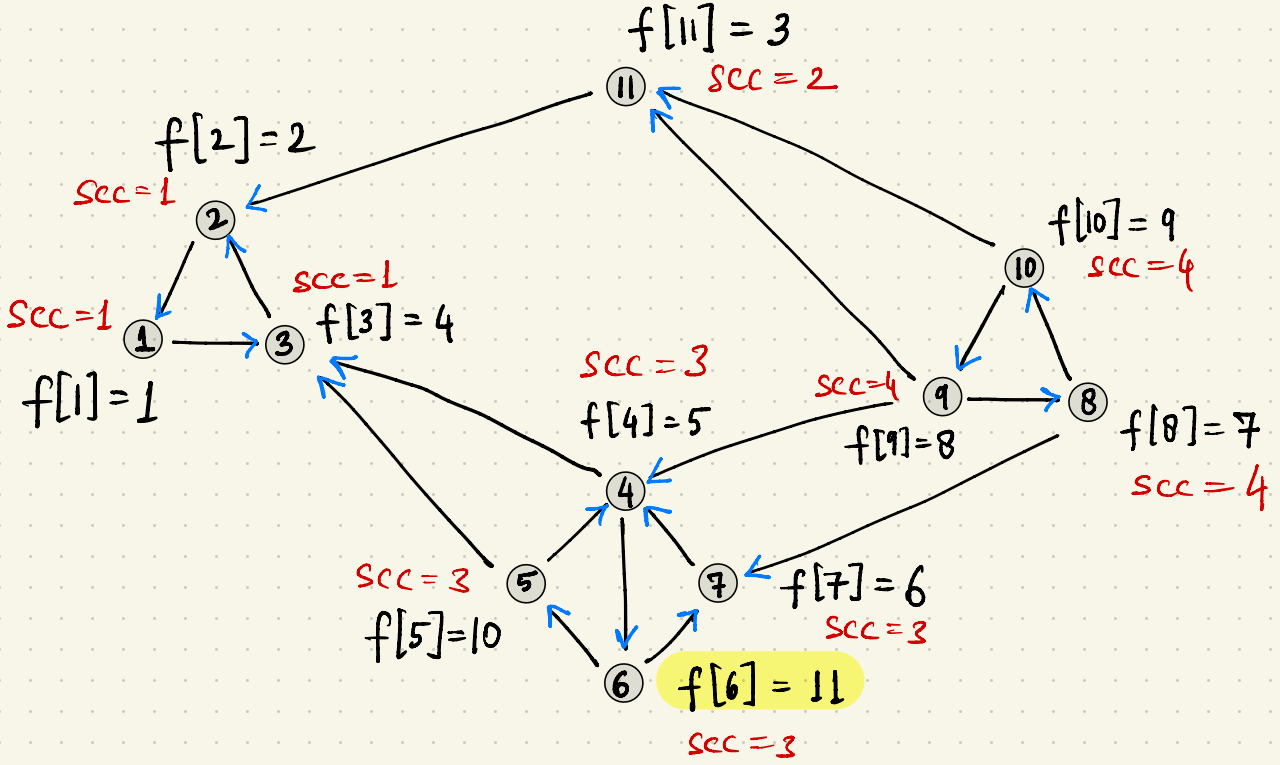
KOSARAJU's ALGORITHM

G



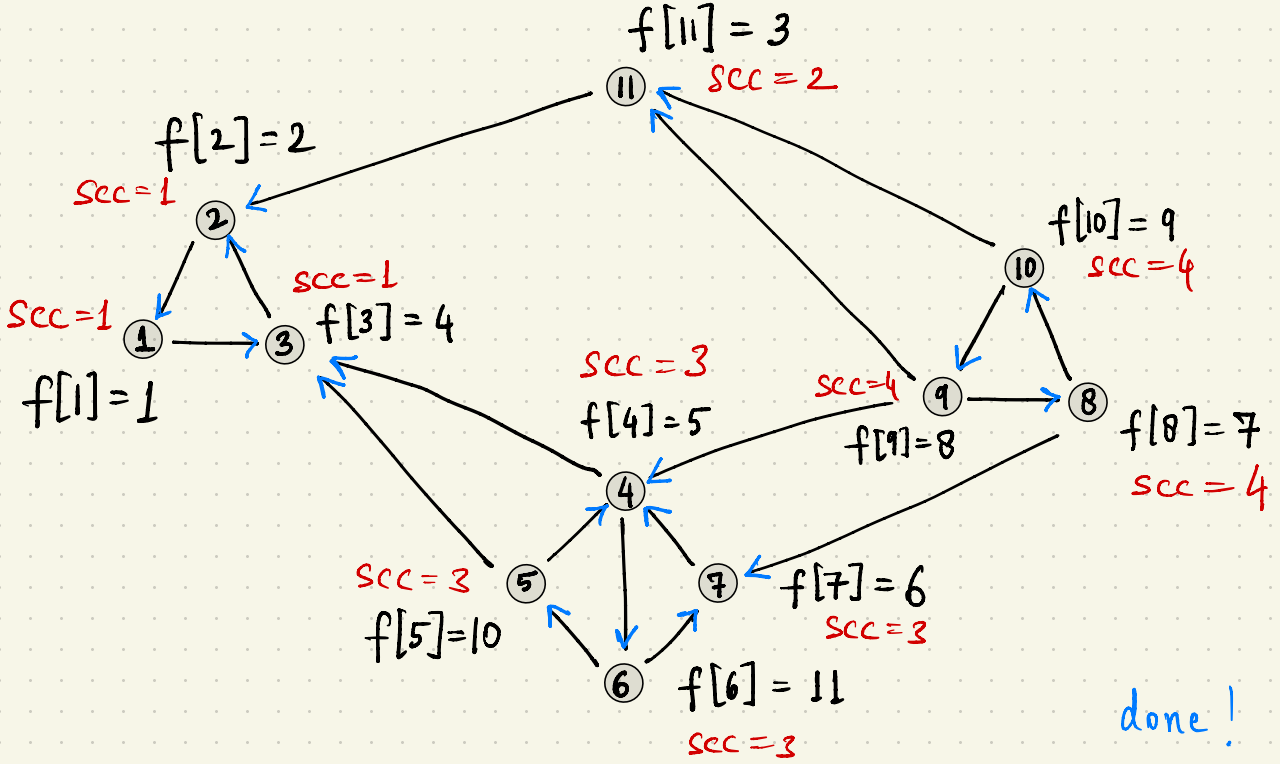
KOSARAJU's ALGORITHM

G



KOSARAJU's ALGORITHM

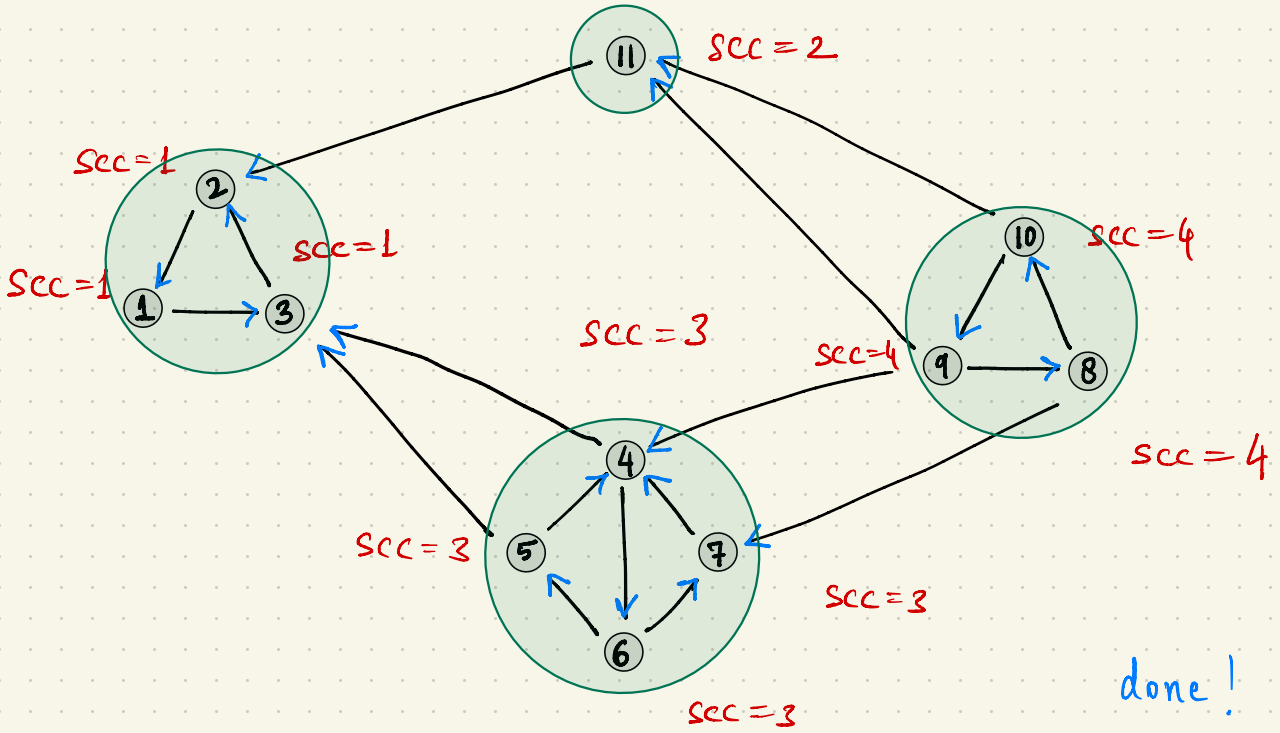
G



done!

KOSARAJU's ALGORITHM

G



KOSARAJU's ALGORITHM

Theorem: Kosaraju's algorithm terminates in $O(m+n)$ time.

KOSARAJU's ALGORITHM

Theorem: Kosaraju's algorithm terminates in $O(m+n)$ time.

Proof: Exercise

Idea: Two runs of DFS.

KOSARAJU's ALGORITHM

Theorem: Kosaraju's algorithm terminates in $O(m+n)$ time.

Proof: Exercise

Idea: Two runs of DFS.

🤔 To go through f[v] in ascending order, do we need to sort?

KOSARAJU's ALGORITHM

Theorem: At termination of kosaraju's algorithm, for any $u, v \in V$

$$scc[u] = scc[v] \iff u \text{ and } v \text{ are in the same strongly connected component}$$

KOSARAJU's ALGORITHM

Theorem: At termination of kosaraju's algorithm, for any $u, v \in V$

$$scc[u] = scc[v] \iff u \text{ and } v \text{ are in the same strongly connected component}$$

Proof: ($\Leftarrow$)

KOSARAJU's ALGORITHM

Theorem: At termination of kosaraju's algorithm, for any $u, v \in V$

$$scc[u] = scc[v] \iff u \text{ and } v \text{ are in the same strongly connected component}$$

Proof: ($\Leftarrow$) Suppose u and v in the same SCC, say S.

KOSARAJU's ALGORITHM

Theorem: At termination of kosaraju's algorithm, for any $u, v \in V$

$$scc[u] = scc[v] \iff u \text{ and } v \text{ are in the same strongly connected component}$$

Proof: ($\Leftarrow$) Suppose u and v in the same SCC, say S.

Let $x \in S$ be the first explored vertex in S ("leader" vertex)

KOSARAJU's ALGORITHM

Theorem: At termination of kosaraju's algorithm, for any $u, v \in V$

$$scc[u] = scc[v] \iff u \text{ and } v \text{ are in the same strongly connected component}$$

Proof: ($\Leftarrow$) Suppose u and v in the same SCC, say S.

Let $x \in S$ be the first explored vertex in S ("leader" vertex)

By correctness of DFS, $scc[u] = scc[v] = scc[x]$.

KOSARAJU's ALGORITHM

Theorem: At termination of kosaraju's algorithm, for any $u, v \in V$

$$scc[u] = scc[v] \iff u \text{ and } v \text{ are in the same strongly connected component}$$

Proof: ($\Rightarrow$)

KOSARAJU's ALGORITHM

Theorem: At termination of kosaraju's algorithm, for any $u, v \in V$

$$scc[u] = scc[v] \iff u \text{ and } v \text{ are in the same strongly connected component}$$

Proof: ($\Rightarrow$) (by strong induction)

$P(i)$: $scc[u] = scc[v] = i \Rightarrow u, v$ in same SCC

KOSARAJU's ALGORITHM

Theorem: At termination of kosaraju's algorithm, for any $u, v \in V$

$scc[u] = scc[v] \iff$ u and v are in the same strongly connected component

Proof: ($\Rightarrow$) (by strong induction)

$P(i)$: $scc[u] = scc[v] = i \Rightarrow u, v$ in same SCC

Base case $P(1)$: $scc[u] = scc[v] = 1$

$\Rightarrow$ must belong to sink SCC of G (by key observation).

KOSARAJU's ALGORITHM

Theorem: At termination of kosaraju's algorithm, for any $u, v \in V$

$$scc[u] = scc[v] \iff u \text{ and } v \text{ are in the same strongly connected component}$$

Proof: ($\Rightarrow$) (by strong induction)

$P(i+1)$: $scc[u] = scc[v] = i+1$

KOSARAJU's ALGORITHM

Theorem: At termination of kosaraju's algorithm, for any $u, v \in V$

$$scc[u] = scc[v] \iff u \text{ and } v \text{ are in the same strongly connected component}$$

Proof: ($\Rightarrow$) (by strong induction)

$P(i+1)$: $scc[u] = scc[v] = i+1$

Among vertices with $scc[\cdot] = i+1$, let x be the first to be marked as explored.

KOSARAJU's ALGORITHM

Theorem: At termination of kosaraju's algorithm, for any $u, v \in V$

$scc[u] = scc[v] \iff$ u and v are in the same strongly connected component

Proof: ($\Rightarrow$) (by strong induction)

$P(i+1)$: $scc[u] = scc[v] = i+1$

Then, all vertices in SCC of x get $scc[\cdot] = i+1$.

KOSARAJU's ALGORITHM

Theorem: At termination of kosaraju's algorithm, for any $u, v \in V$

$scc[u] = scc[v] \iff$ u and v are in the same strongly connected component

Proof: ($\Rightarrow$) (by strong induction)

$P(i+1)$: $scc[u] = scc[v] = i+1$

Then, all vertices in SCC of x get $scc[\cdot] = i+1$.

No other vertex gets $scc[\cdot] = i+1$. Why?

KOSARAJU's ALGORITHM

Theorem: At termination of kosaraju's algorithm, for any $u, v \in V$

$scc[u] = scc[v] \iff u$ and v are in the same strongly connected component

Proof: ($\Rightarrow$) (by strong induction)

$P(i+1)$: $scc[u] = scc[v] = i+1$

Then, all vertices in SCC of x get $scc[\cdot] = i+1$.

No other vertex gets $scc[\cdot] = i+1$. Why?

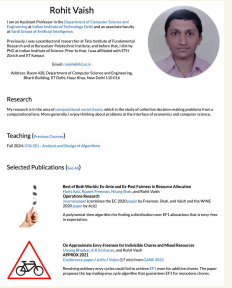
SCC $i+1$ can only have edges to SCC $1, 2, \ldots, i$ — explored!

∎

Application of SCCs : Understanding the web graph

The Web Graph

The Web Graph



Rohit Vaish

I am an Assistant Professor in the Department of Computer Science and Engineering at Indian Institute of Technology Delhi and an associate faculty at Yardi School of Artificial Intelligence.

Previously, I was a postdoctoral research fellow at Tata Institute of Fundamental Research and at Rensselaer Polytechnic Institute, and before that, I did my PhD in Computer Science at Technion–Israel Institute of Technology. I was also affiliated with ETH Zürich and IST Vienna.

Email: rvaish@iitd.ac.in

Address: Room 430, Department of Computer Science and Engineering, Bharti Building, IIT Delhi, Hauz Khas, New Delhi 110016

Research

My research is in the area of computational social choice, which is the study of collective decision-making problems from a computational lens. More generally, I enjoy thinking about problems at the interface of economics and computer science.

Teaching [Previous Courses]

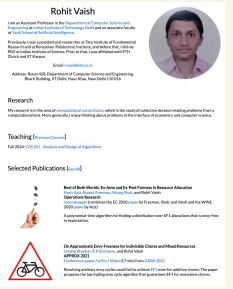
Fall 2024: COL351 - Analysis and Design of Algorithms

Selected Publications [See All]

Best of Both Worlds: Ex-Ante and Ex-Post Fairness in Resource Allocation
Operations Research
2024 paper by Hannaneh Akrami, Jugal Garg, Pieter Kleer, Mengqi Zhao, and Rohit Vaish
A polynomial-time algorithm for finding a distribution over EF1 allocations that is envy-free in expectation.

On Approximate Envy-Freeness for Indivisible Chores and Mixed Resources
APPROX 2021
Breaking a three-decade envy-cycle could help us achieve 1/2 + ε envy-free for additive chores. The paper proposes the bar-trading envy-cycle algorithm that guarantees EF1/3 for monotone chores.

The Web Graph



Rohit Vaish

I am an Assistant Professor in the Department of Computer Science and Engineering at Indian Institute of Technology Delhi and an associate faculty at Yardi School of Artificial Intelligence.

Previously, I was a postdoctoral research fellow at Tata Institute of Fundamental Research and at Rensselaer Polytechnic Institute, and before that, I did my PhD in Computer Science and Business Price-in-Hat. I was affiliated with ETH Zürich and IIT Kanpur.

Email: rvaish@iitd.ac.in

Address: Room 430, Department of Computer Science and Engineering, Bharti Building, IIT Delhi, Hauz Khas, New Delhi 110016

Research

My research is in the area of computational social choice, which is the study of collective decision-making problems from a computational lens. More generally, I enjoy thinking about problems at the interface of economics and computer science.

Teaching [Previous Courses]

Fall 2024: COL351 - Analysis and Design of Algorithms

Selected Publications [See All]

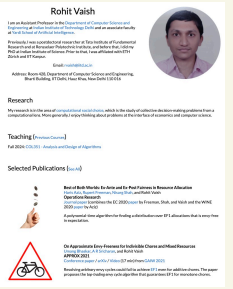
Best of Both Worlds: Ex-Ante and Ex-Post Fairness in Resource Allocation
Operations Research
2024 version by Arxiv

A polynomial-time algorithm for finding a distribution over EF1 allocations that is envy-free in expectation.

On Approximate Envy-Freeness for Indivisible Chores and Mixed Resources
APPROX 2021
2021 version / arXiv / Video [67 min] from GAMM 2021

Breaking a three-envy cycle could fail to achieve EF1 even for additive chores. The paper proposes the bar-trading envy cycle algorithm that guarantees EF1 for monotone chores.

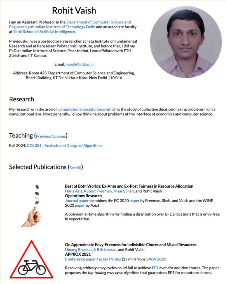
The Web Graph



COL351, Fall Semester 2024

Analysis and Design of Algorithms

The Web Graph



Rohit Vaish

I am an Assistant Professor in the Department of Computer Science and Engineering at Indian Institute of Technology Delhi and an associate faculty at Yardi School of Artificial Intelligence.

Previously, I was a postdoctoral researcher at Tata Institute of Fundamental Research and at Rensselaer Polytechnic Institute, and before that, I did my PhD at Indian Institute of Science. Prior to that, I was affiliated with ETH Zürich and IIT Kanpur.

Email: rvaish@iitd.ac.in

Address: Room 428, Department of Computer Science and Engineering, Bharti Building, IIT Delhi, Hauz Khas, New Delhi 110 016

Research

My research is in the area of computational social choice, which is the study of collective decision-making problems from a computational lens. More generally, I enjoy thinking about problems at the interface of economics and computer science.

Teaching (Previous Courses)

Fall 2024: COL351 - Analysis and Design of Algorithms

Selected Publications (See All)

Best of Both Worlds: Ex-Ante and Ex-Post Fairness in Resource Allocation
Haris Aziz, Rupert Freeman, Nisarg Shah, and Rohit Vaish
Operations Research
Journal paper (combines the EC 2020 paper by Freeman, Shah, and Vaish and the WINE 2020 paper by Aziz)

A polynomial-time algorithm for finding a distribution over EF1 allocations that is envy-free in expectation.

On Approximate Envy-Freeness for Indivisible Chores and Mixed Resources
Umang Bhaskar, A R Sricharan, and Rohit Vaish
APPROX 2021
Conference paper / arXiv / Video (17 min) from GAIW 2021

Resolving *arbitrary* envy cycles could fail to achieve EF1 even for additive chores. The paper proposes the *top-trading envy cycle algorithm* that guarantees EF1 for monotone chores.

COL351, Fall Semester 2024

Analysis and Design of Algorithms

awesome collaborators!

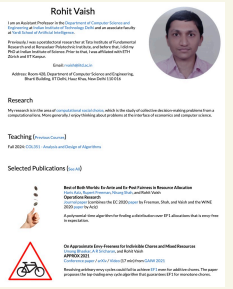
The Web Graph

Department of Computer Science and Engineering
Indian Institute of Technology Delhi

Rohit Vaish
Assistant Professor and Pankaj Gupta Faculty Fellow
Ph.D. (IISc)

Algorithms and Economics, Artificial Intelligence, Computational Social Choice, Game Theory

COMSOC



COL351, Fall Semester 2024
Analysis and Design of Algorithms

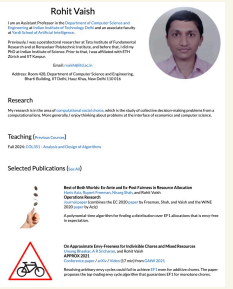
- **Computational Social Choice** – Rohit Vaish (Indian Institute of Technology Delhi)
  2023 · 2022-Fall · 2022-Spring



. . .

awesome collaborators!

The Web Graph



- **Computational Social Choice** – Rohit Vaish (Indian Institute of Technology Delhi)
  2023 · 2022-Fall · 2022-Spring

...

What does the web graph look like?

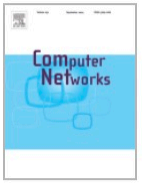
awesome collaborators!

The Web Graph

ELSEVIER

Computer Networks

Volume 33, Issues 1–6, June 2000, Pages 309-320



Graph structure in the Web

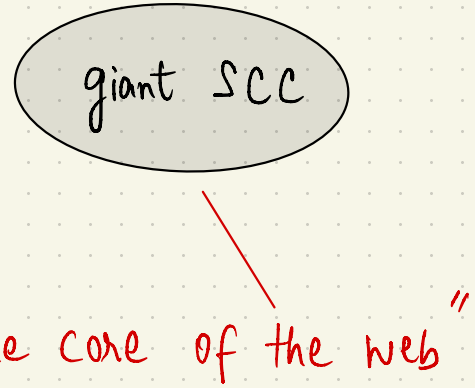
Andrei Broder a, Ravi Kumar b, Farzin Maghoul a, Prabhakar Raghavan b, Sridhar Rajagopalan b, Raymie Stata c, Andrew Tomkins b, Janet Wiener c

Size : ~ 200 million vertices

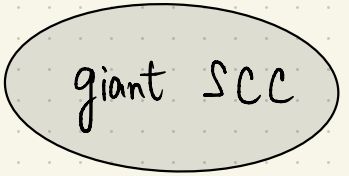
~ 1.5 billion edges

The Bow Tie

The Bow Tie



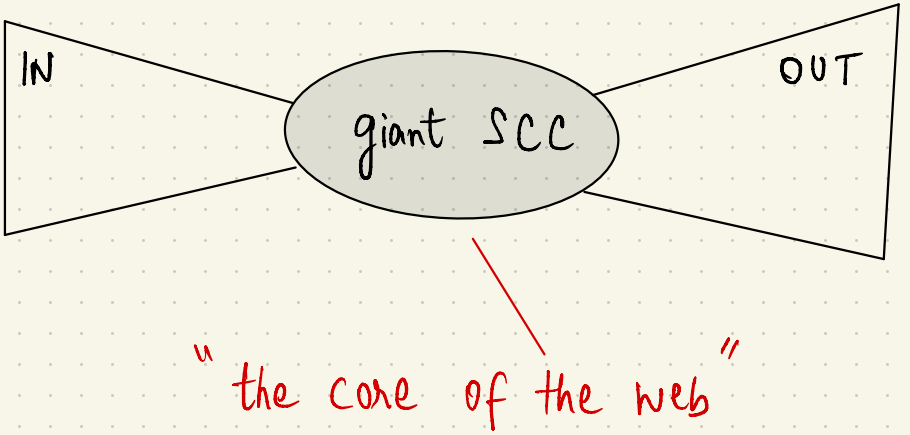
THE BOW TIE



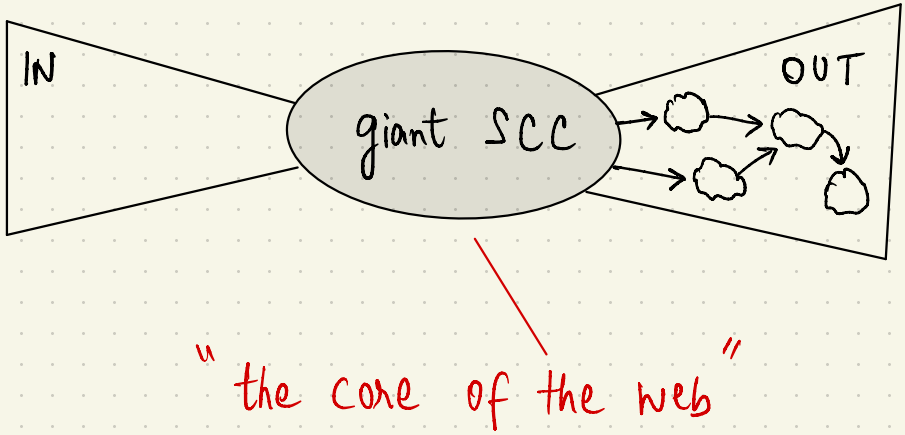
"the core of the web"

Intuitively, there should be only one giant SCC.

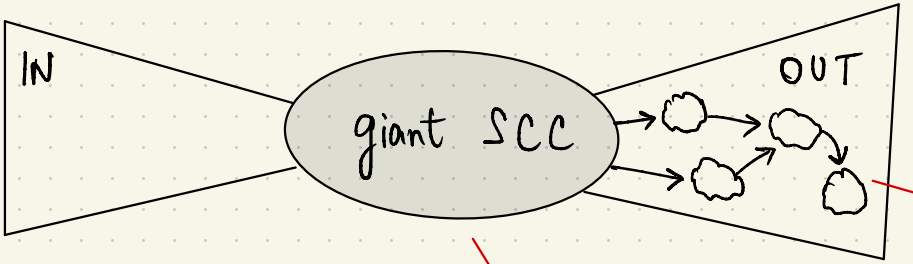
THE BOW TIE



THE BOW TIE



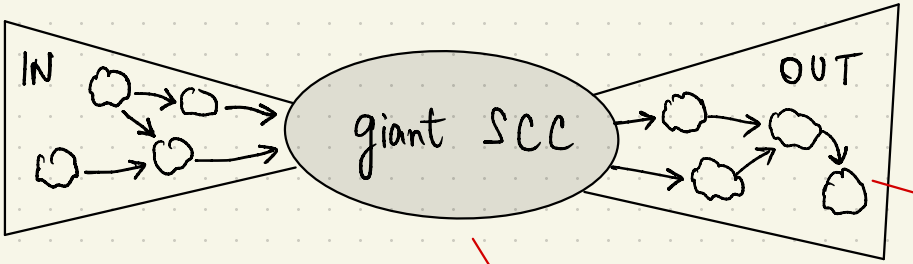
THE BOW TIE



e.g., corporate websites

"the core of the web"

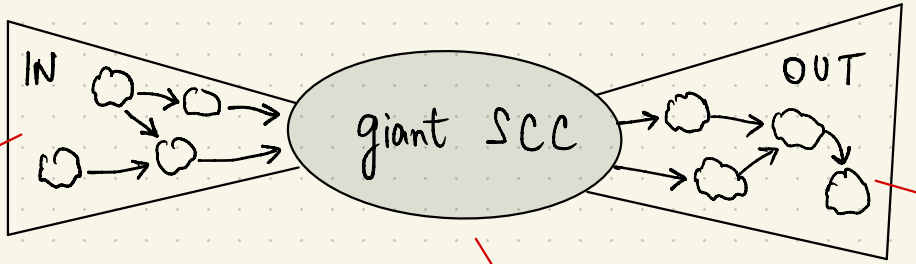
THE BOW TIE



e.g., corporate websites

"the core of the web"

THE BOW TIE

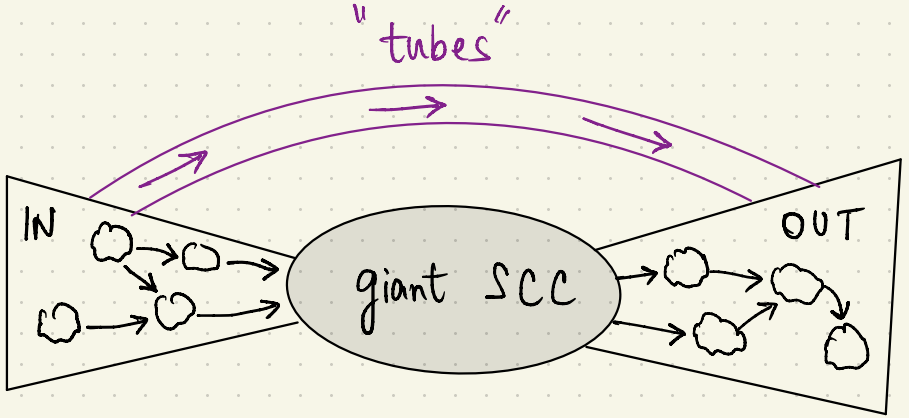


e.g., new webpages

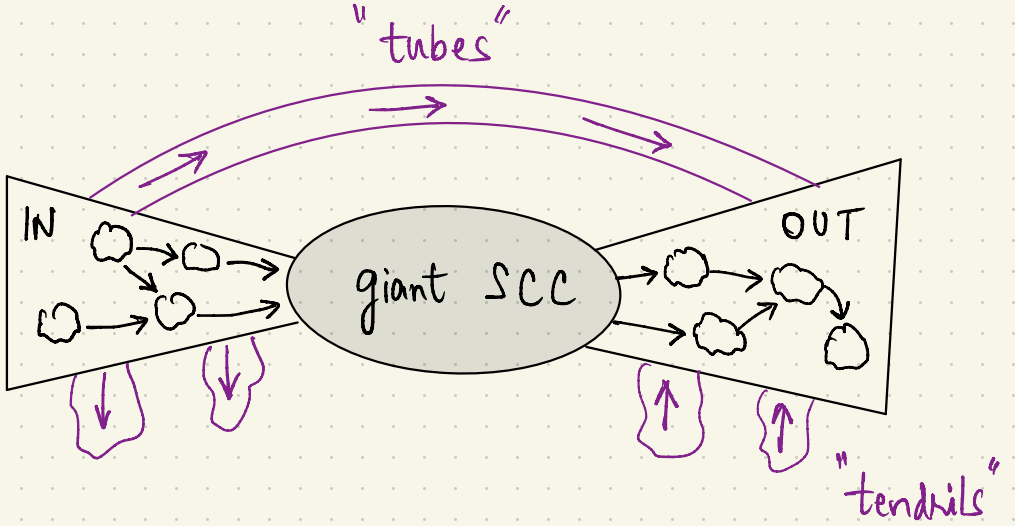
e.g., corporate websites

"the core of the web"

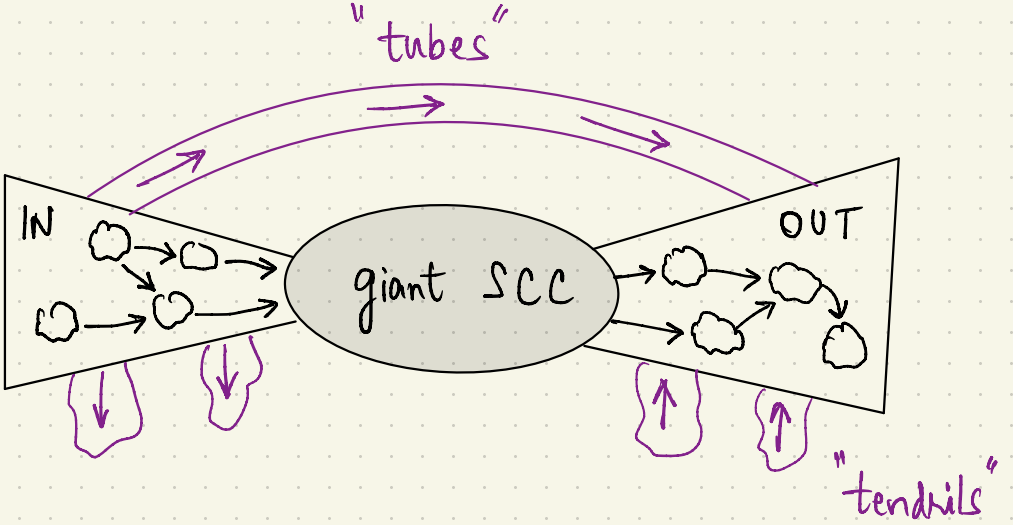
The Bow Tie



THE BOW TIE

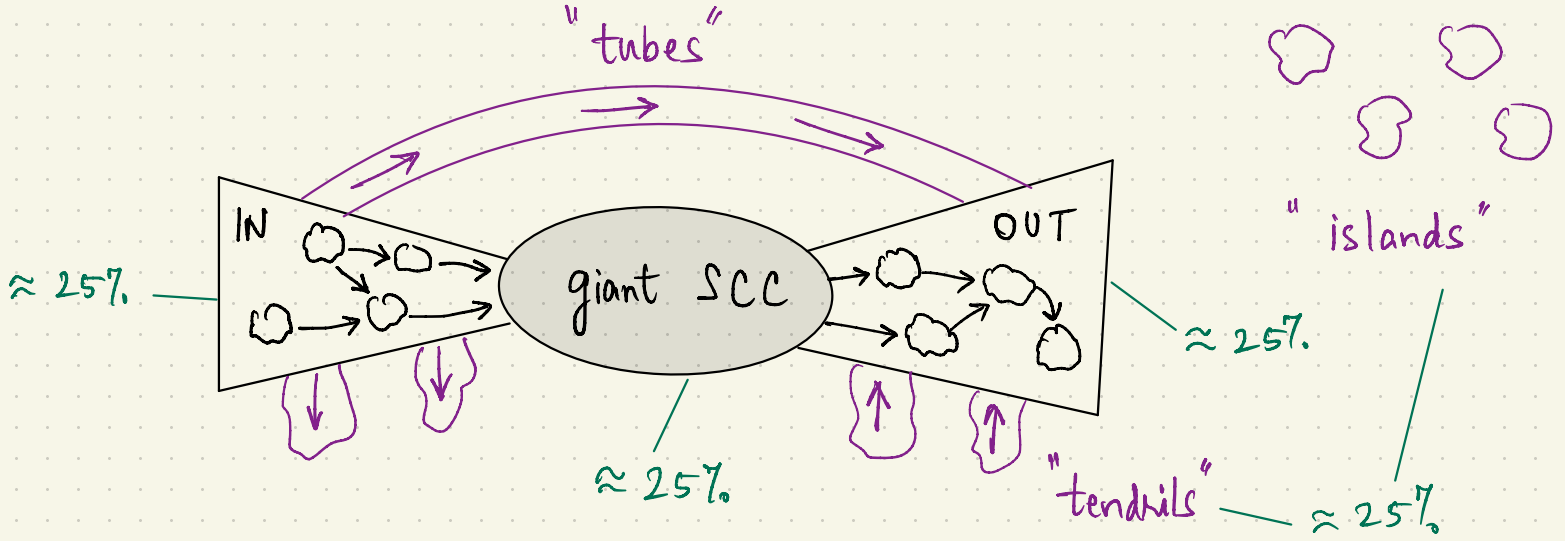


THE BOW TIE

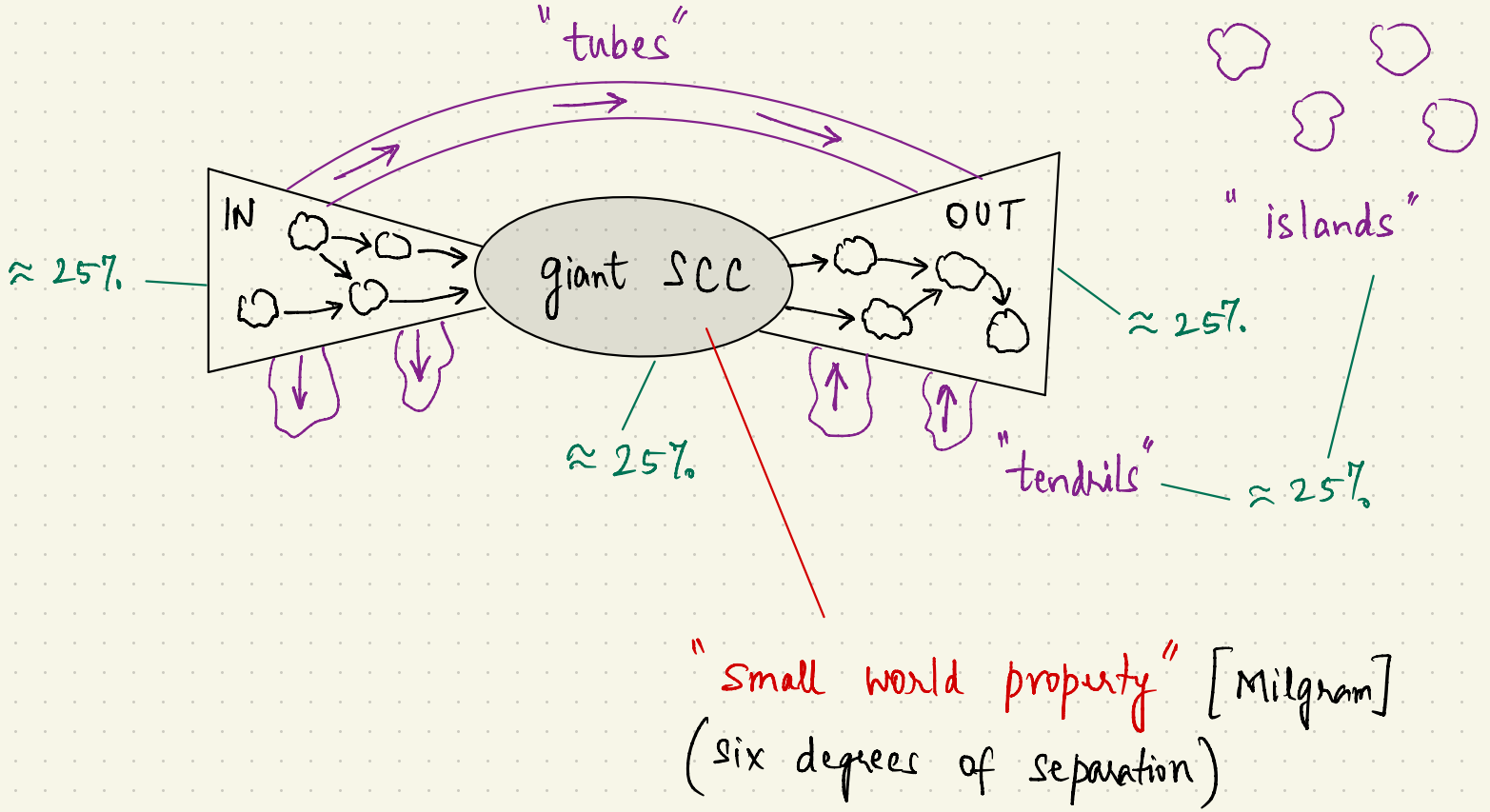


"islands"

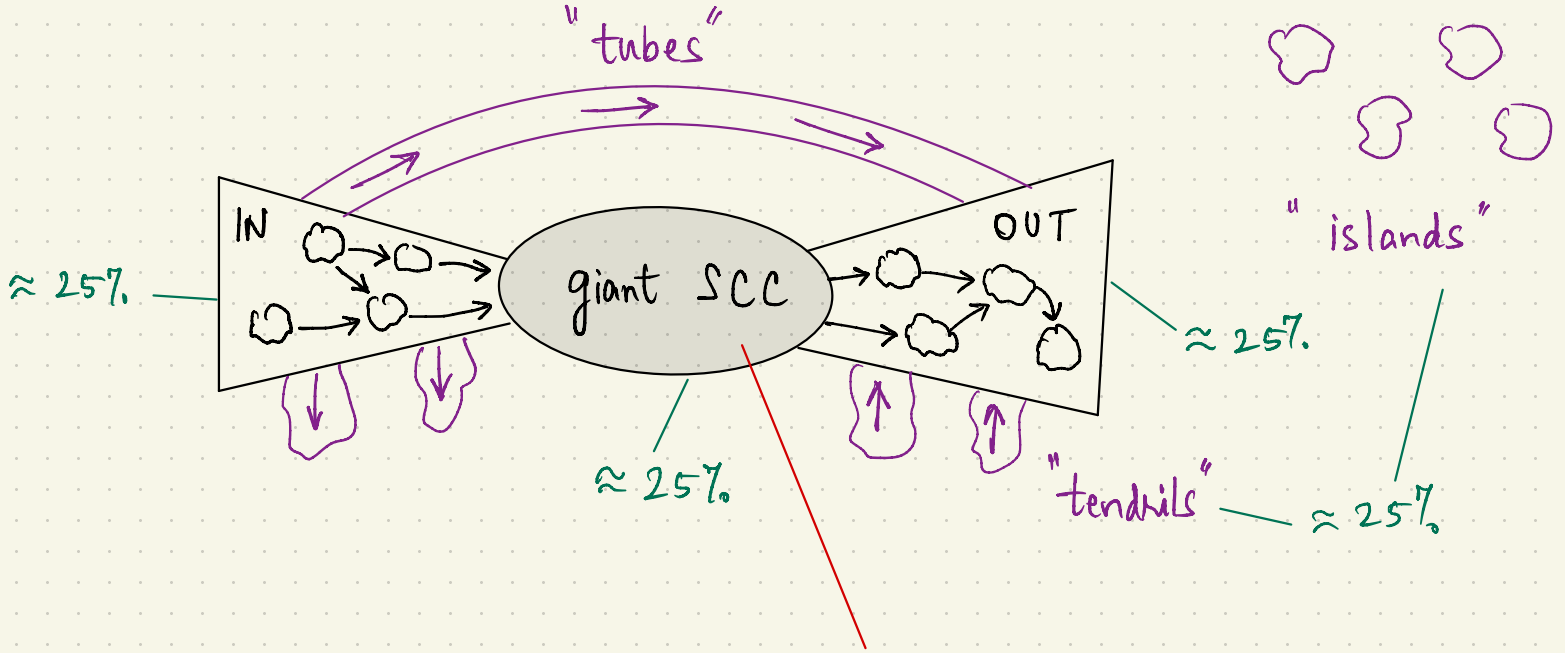
THE BOW TIE



THE BOW TIE


"tubes"
IN
giant SCC
OUT
"islands"
≈ 25%
≈ 25%
≈ 25%
"tendrils"
≈ 25%
"small world property" [Milgram]
(six degrees of separation)

THE BOW TIE



≈ 56 million vertices

typical short paths ≤ 20 hyperlinks

"small world property" [Milgram]

(six degrees of separation)

DIJKSTRA'S ALGORITHM

SINGLE-SOURCE SHORTEST PATH PROBLEM

SINGLE-SOURCE SHORTEST PATH PROBLEM

input: A directed graph $G = (V, E)$, starting vertex s, a non negative length l_e for each edge $e \in E$

output: dist (s, v) for every vertex $v \in V$.

SINGLE-SOURCE SHORTEST PATH PROBLEM

input: A directed graph $G = (V, E)$, starting vertex s, a non negative length l_e for each edge $e \in E$

output: dist (s, v) for every vertex $v \in V$.

length of the shortest path from s to v

SINGLE-SOURCE SHORTEST PATH PROBLEM

input: A directed graph $G = (V, E)$, starting vertex s, a non negative length l_e for each edge $e \in E$

output: dist (s, v) for every vertex $v \in V$.

length of the shortest path from s to v

sum of edge lengths

SINGLE-SOURCE SHORTEST PATH PROBLEM

input: A directed graph $G = (V, E)$, starting vertex s, a non negative length l_e for each edge $e \in E$

output: dist (s, v) for every vertex $v \in V$.

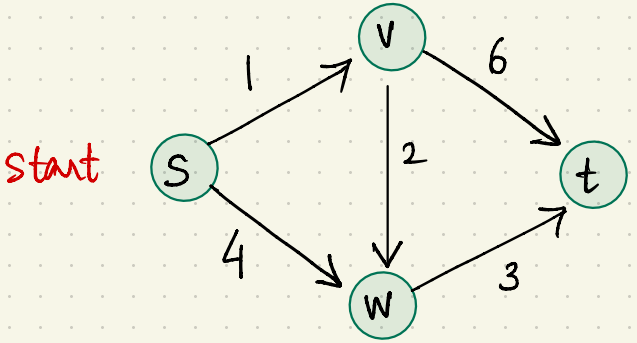
length of the shortest path from s to v if $s \rightsquigarrow v$ path exists

∞ otherwise

SINGLE-SOURCE SHORTEST PATH PROBLEM

input: A directed graph $G = (V, E)$, starting vertex s, a non negative length l_e for each edge $e \in E$

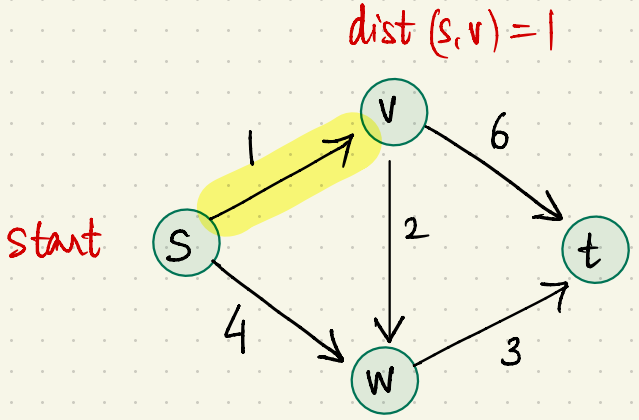
output: dist (s, v) for every vertex $v \in V$.



SINGLE-SOURCE SHORTEST PATH PROBLEM

input: A directed graph $G = (V, E)$, starting vertex s, a non negative length l_e for each edge $e \in E$

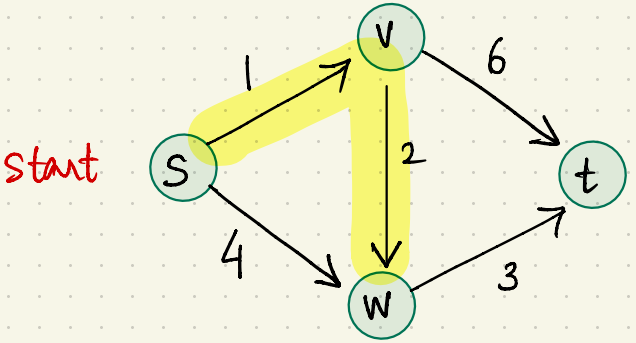
output: dist (s, v) for every vertex $v \in V$.



SINGLE-SOURCE SHORTEST PATH PROBLEM

input: A directed graph $G = (V, E)$, starting vertex s, a non negative length l_e for each edge $e \in E$

output: dist (s, v) for every vertex $v \in V$.

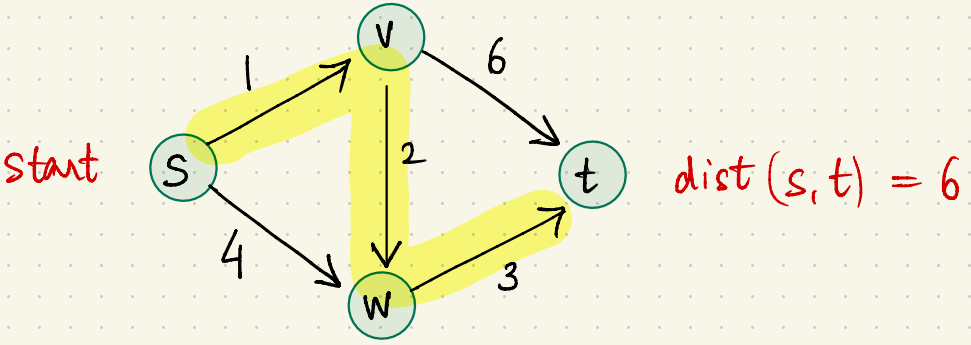


dist $(s, w) = 4$

SINGLE-SOURCE SHORTEST PATH PROBLEM

input: A directed graph $G = (V, E)$, starting vertex s, a non negative length l_e for each edge $e \in E$

output: dist (s, v) for every vertex $v \in V$.



SINGLE-SOURCE SHORTEST PATH PROBLEM

input: A directed graph $G = (V, E)$, starting vertex s, a non negative length ℓ_e for each edge $e \in E$

output: dist (s, v) for every vertex $v \in V$.

For negative lengths, use Bellman-Ford algorithm.

SINGLE-SOURCE SHORTEST PATH PROBLEM

Doesn't BFS already compute shortest paths?

SINGLE-SOURCE SHORTEST PATH PROBLEM

Doesn't BFS already compute shortest paths?

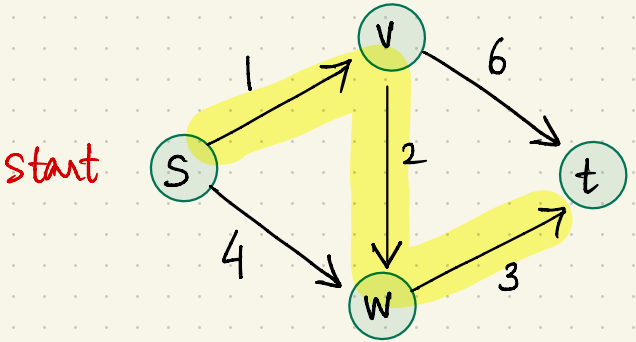
Yes, if length of a path = number of edges in it

($\ell_e = 1$ for every edge e)

SINGLE-SOURCE SHORTEST PATH PROBLEM

Doesn't BFS already compute shortest paths?

Yes, if length of a path = number of edges in it
($\ell_e = 1$ for every edge e)



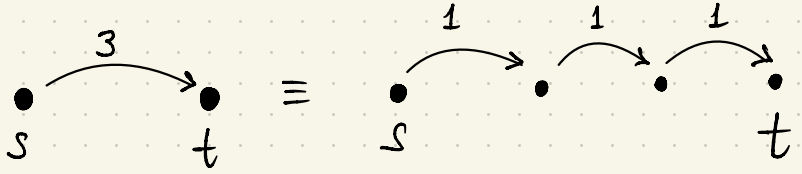
BFS fails

SINGLE-SOURCE SHORTEST PATH PROBLEM

Isn't an edge with length l_e the same as l_e unit length edges?

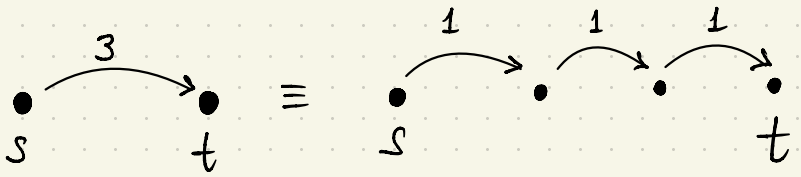
SINGLE-SOURCE SHORTEST PATH PROBLEM

🤔 Isn't an edge with length l_e the same as l_e unit length edges?



SINGLE-SOURCE SHORTEST PATH PROBLEM

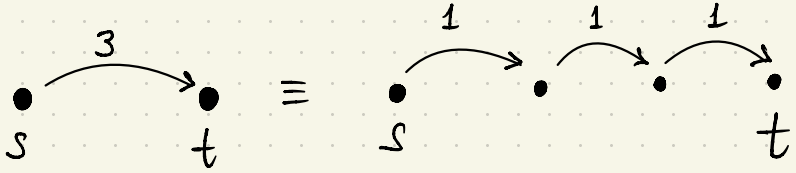
Isn't an edge with length l_e the same as l_e unit length edges?



This representation is faithful but wasteful.

SINGLE-SOURCE SHORTEST PATH PROBLEM

Isn't an edge with length l_e the same as l_e unit length edges?

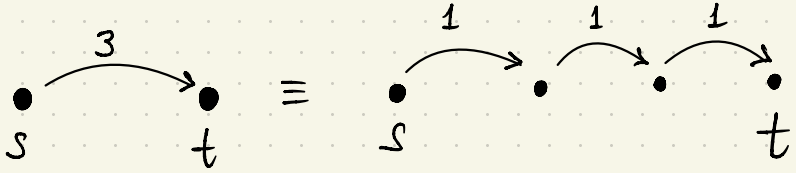


This representation is faithful but wasteful.

BFS time is linear in the size of the blown up graph

SINGLE-SOURCE SHORTEST PATH PROBLEM

Isn't an edge with length l_e the same as l_e unit length edges?



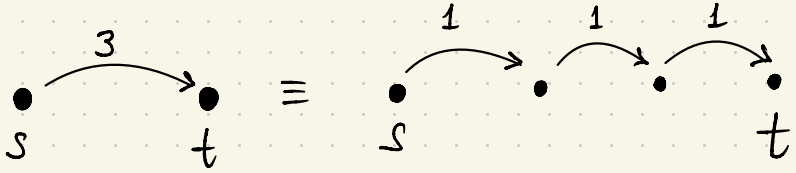
This representation is faithful but wasteful.

BFS time is linear in the size of the blown up graph

$$O\left(n + \sum_e l_e\right) >> O(n+m)$$

SINGLE-SOURCE SHORTEST PATH PROBLEM

Isn't an edge with length l_e the same as l_e unit length edges?



This representation is faithful but wasteful.

BFS time is linear in the size of the blown up graph

$$O\left(n + \sum_e l_e\right) >> O(n+m)$$

Solution: Dijkstra's algorithm

DIJKSTRA'S ALGORITHM

DIJKSTRA'S ALGORITHM

When $\ell_e = 1$ for all edges e, Dijkstra's algorithm becomes breadth-first search.

DIJKSTRA'S ALGORITHM

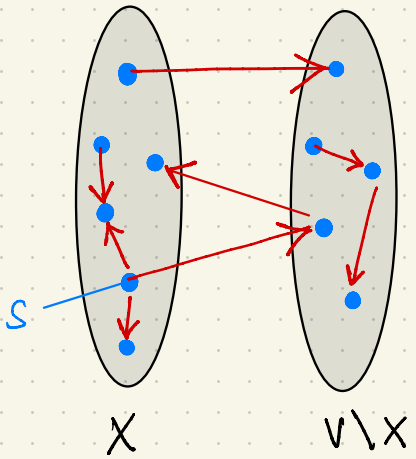
Initialize

$X := \{s\}$ vertices processed so far

DIJKSTRA'S ALGORITHM

Initialize

$X := \{s\}$ vertices processed so far

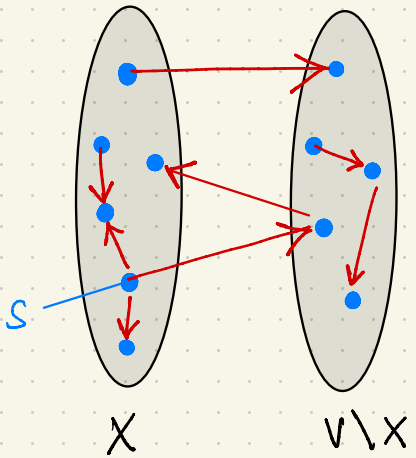


DIJKSTRA'S ALGORITHM

Initialize

$X := \{s\}$ vertices processed so far

$A[s] = 0$ computed shortest-path distances



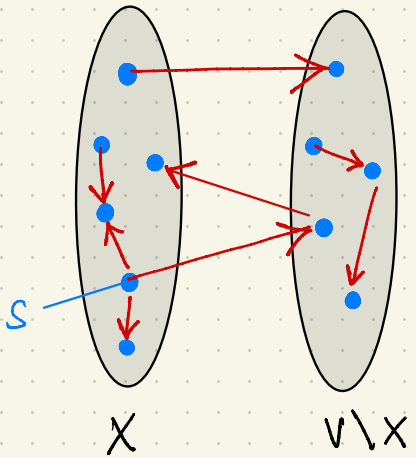
DIJKSTRA'S ALGORITHM

Initialize

$X := \{s\}$ vertices processed so far

$A[s] = 0$ computed shortest-path distances

$B[s] = \emptyset$ computed shortest paths



DIJKSTRA's ALGORITHM

Initialize

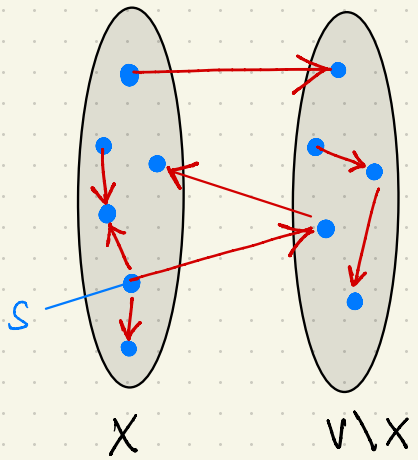
$X := \{s\}$ vertices processed so far

$A[s] = 0$ computed shortest-path distances

$B[s] = \emptyset$ computed shortest paths

// main loop

while $X \neq V$ need to grow X



DIJKSTRA'S ALGORITHM

Initialize

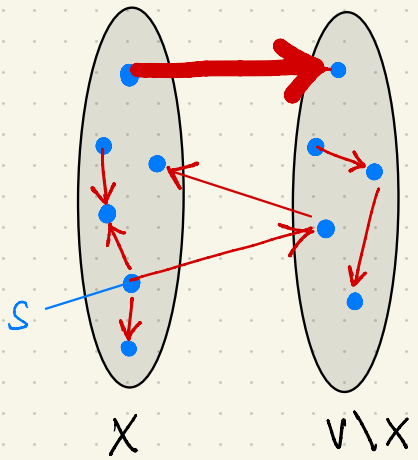
$X := \{s\}$ vertices processed so far

$A[s] = 0$ computed shortest-path distances

$B[s] = \emptyset$ computed shortest paths

// main loop

while $X \neq V$ need to grow X



among all edges $(v,w) \in E$ with $v \in X$ and $w \notin X$, pick one that minimizes

$$A[v] + \ell_{vw}$$

DIJKSTRA'S ALGORITHM

Initialize

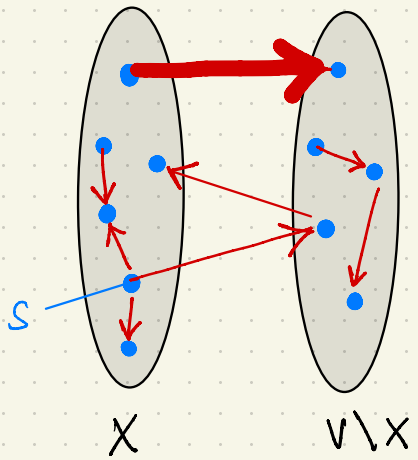
$X := \{s\}$ vertices processed so far

$A[s] = 0$ computed shortest-path distances

$B[s] = \emptyset$ computed shortest paths

// main loop

while $X \neq V$ need to grow X



among all edges $(v,w) \in E$ with $v \in X$ and $w \notin X$, pick one that minimizes

$A[v] + \ell_{vw}$ Dijkstra score

DIJKSTRA'S ALGORITHM

Initialize

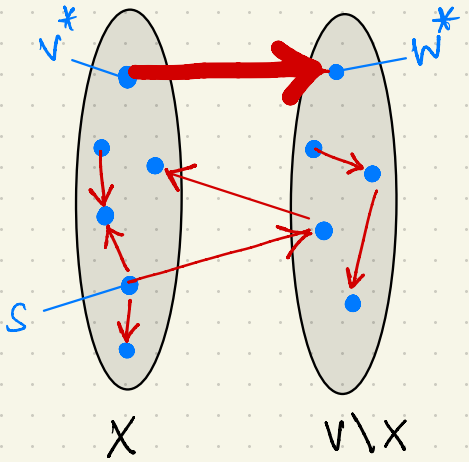
$X := \{s\}$ vertices processed so far

$A[s] = 0$ computed shortest-path distances

$B[s] = \emptyset$ computed shortest paths

// main loop

while $X \neq V$ need to grow X



among all edges $(v,w) \in E$ with $v \in X$ and $w \notin X$, pick one that minimizes

$$A[v] + \ell_{vw}$$ Dijkstra score

Call it (v^*, w^*).

DIJKSTRA'S ALGORITHM

Initialize

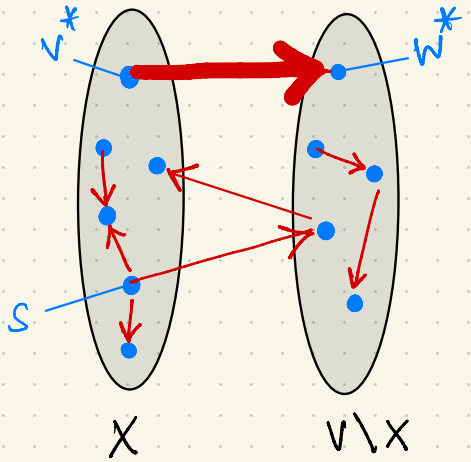
$X := \{s\}$ vertices processed so far

$A[s] = 0$ computed shortest-path distances

$B[s] = \emptyset$ computed shortest paths

// main loop

while $X \neq V$ need to grow X



among all edges $(v,w) \in E$ with $v \in X$ and $w \notin X$, pick one that minimizes

$$A[v] + \ell_{vw}$$ Dijkstra score

Call it (v^*, w^*).

Add w^* to X.

DIJKSTRA'S ALGORITHM

Initialize

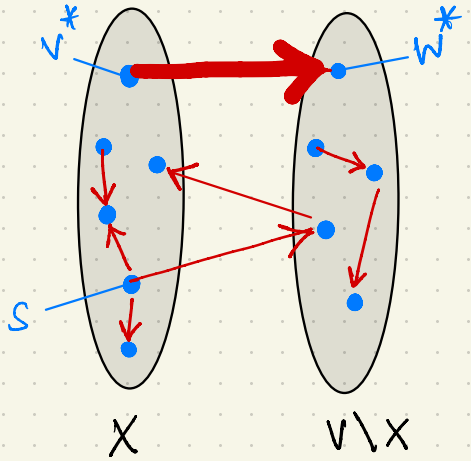
$X := \{s\}$ vertices processed so far

$A[s] = 0$ computed shortest-path distances

$B[s] = \emptyset$ computed shortest paths

// main loop

while $X \neq V$ need to grow X



among all edges $(v,w) \in E$ with $v \in X$ and $w \notin X$, pick one that minimizes

$$A[v] + \ell_{vw}$$ Dijkstra score

Call it (v^*, w^*).

Add w^* to X.

Set $A[w^*] := A[v^*] + \ell_{v^*w^*}$ and $B[w^*] := B[v^*] \cup (v^*, w^*)$.