COL 351 : ANALYSIS & DESIGN OF ALGORITHMS

LECTURE 11

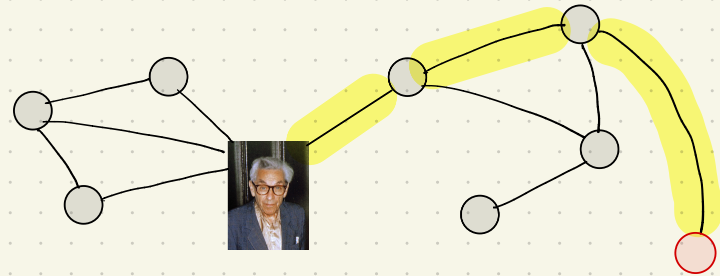
GRAPH ALGORITHMS IV :

TOPOLOGICAL ORDERING & STRONGLY CONNECTED COMPONENTS

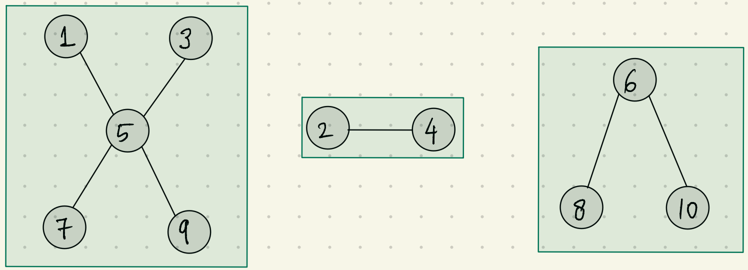
AUG 20, 2024 | ROHIT VAISH

APPLICATIONS OF BFS

Shortest paths



Connected Components



DEPTH-FIRST SEARCH

DEPTH-FIRST SEARCH

Iterative version

DEPTH-FIRST SEARCH

Iterative version

mark all vertices as unexplored

$S :=$ a stack data structure (LIFO), initialized with s

while $S \neq \emptyset$

- remove the top node of S, say v ("pop")
- if v is unexplored
    - mark v as explored
    - for each (v, w) in adj. list of v
        - add w to the front of S ("push")

DEPTH-FIRST SEARCH

Recursive version

```
DFS (G, s)                    // all vertices unexplored before the call

    mark s as explored

    for each (s, v) in adj. list of s
    |       if v is unexplored
    |       |      DFS (G, v)
```

APPLICATIONS OF DFS

Topological ordering

Strongly Connected Components

APPLICATIONS OF DFS

Topological ordering

Strongly Connected Components

TOPOLOGICAL ORDERING

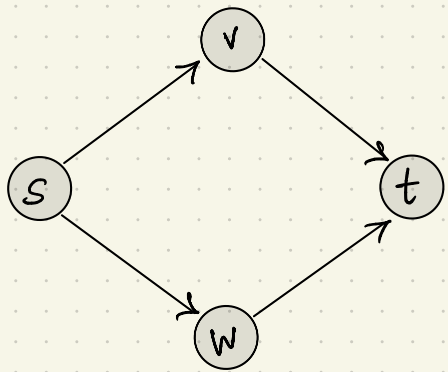
Directed graph $G = (V, E)$

A labeling f of G's vertices such that :

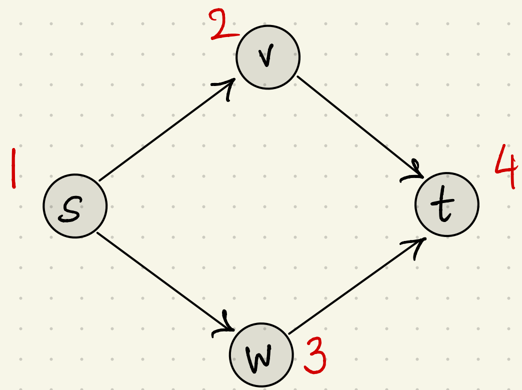
* unique $f(v) \in \{1, 2, \ldots, n\}$ for every $v \in V$

* for every $(v, w) \in E$ $\quad f(v) < f(w)$.

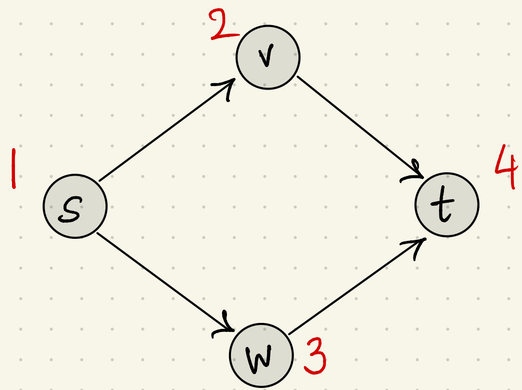
TOPOLOGICAL ORDERING

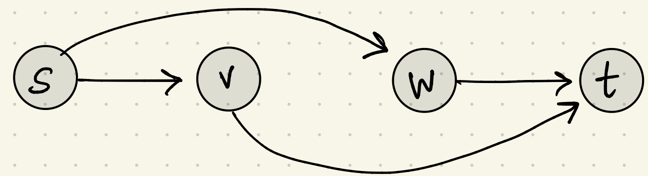


TOPOLOGICAL ORDERING

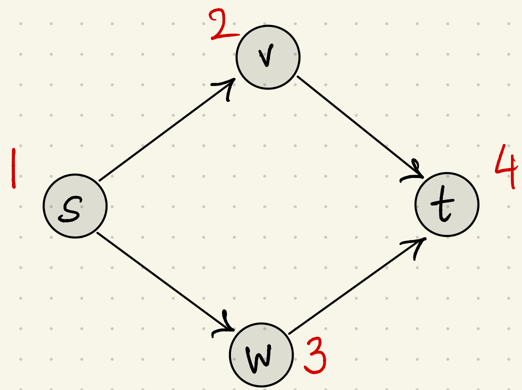


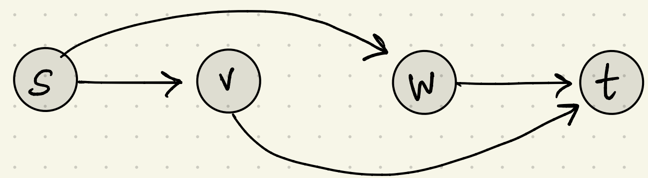
TOPOLOGICAL ORDERING

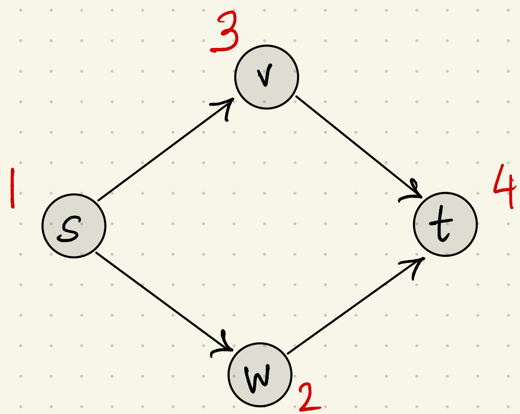


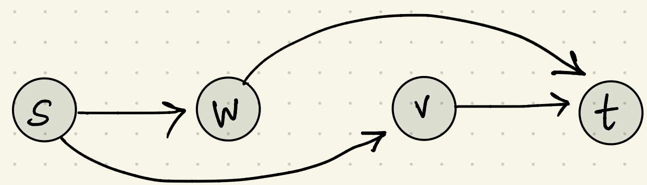


TOPOLOGICAL ORDERING









TOPOLOGICAL ORDERING

Theorem : Every directed acyclic graph has at least one topological ordering.

TOPOLOGICAL ORDERING

Theorem: Every directed acyclic graph has at least one topological ordering.

Lemma: Every directed acyclic graph has at least one sink.

a vertex with no outgoing edges

TOPOLOGICAL ORDERING

Theorem: Every directed acyclic graph has at least one topological ordering.

Proof: Assign $f[v] = n$ to sink vertex v (exists!)

Recurse on $G \setminus \{v\} \longrightarrow$ must be directed acyclic. ∎

TOPOLOGICAL ORDERING

Theorem: Every directed acyclic graph has at least one topological ordering.

Algorithm

1. Find a sink vertex v.

2. Assign to it the largest available label and recurse on $G \setminus \{v\}$.

TOPOLOGICAL ORDERING

Theorem: Every directed acyclic graph has at least one topological ordering.

Algorithm

1. Find a sink vertex v.
2. Assign to it the largest available label and recurse on $G \setminus \{v\}$.

Correctness

If $f[v] = i$, no edges from v to vertices with $f[v] < i$.

TOPOLOGICAL ORDERING

Theorem: Every directed acyclic graph has at least one topological ordering.

Algorithm

1. Find a sink vertex v.

2. Assign to it the largest available label and recurse on $G \setminus \{v\}$.

Running time

TOPOLOGICAL ORDERING

Theorem: Every directed acyclic graph has at least one topological ordering.

Algorithm

Running time

1. Find a sink vertex v. $\longrightarrow O(n)$

2. Assign to it the largest available label and recurse on $G \setminus \{v\}$.

$O(n^2)$

TOPOLOGICAL ORDERING

Theorem: Every directed acyclic graph has at least one topological ordering.

Algorithm

1. Find a sink vertex v. $\longrightarrow$ $O(n)$
2. Assign to it the largest available label and recurse on $G \setminus \{v\}$.

Running time

$O(n^2)$

Can we do better for sparse graphs?

TOPOLOGICAL ORDERING via DFS

TOPOLOGICAL ORDERING via DFS

DFS pseudocode

```
DFS (G, s)                    // all vertices unexplored before the call

    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 (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 $v \in V$

  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 $v \in V$

    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

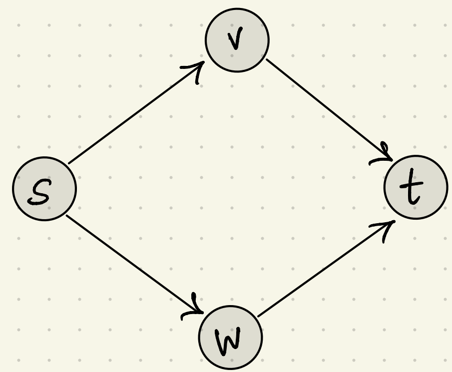
    if v is unexplored

        DFS (G, v)

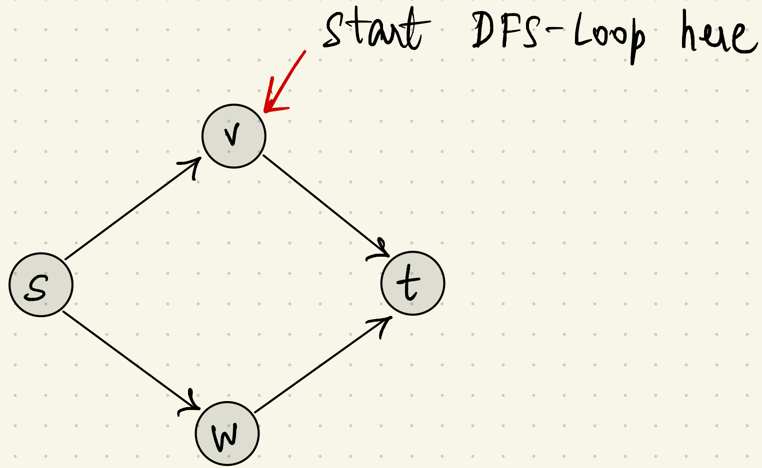
set $f[s]$ = current_label

decrease current_label by 1

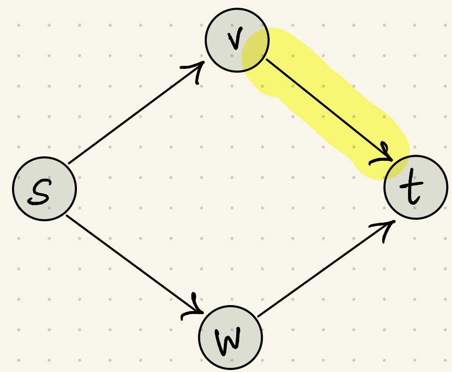
TOPOLOGICAL ORDERING via DFS



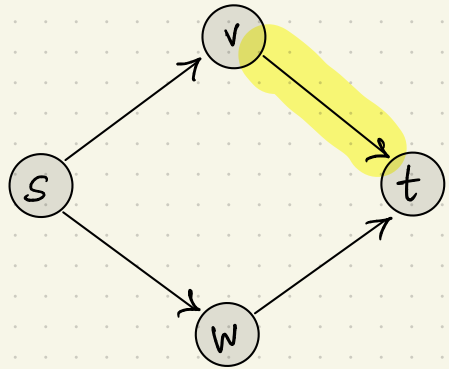
TOPOLOGICAL ORDERING via DFS



TOPOLOGICAL ORDERING via DFS

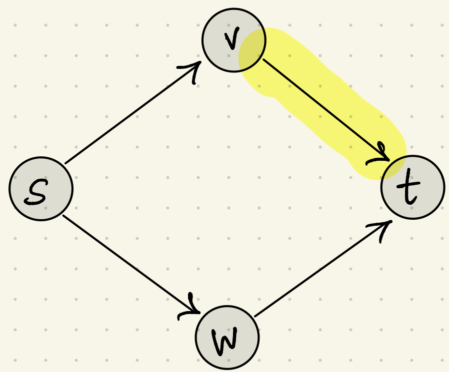


TOPOLOGICAL ORDERING via DFS



nowhere to go from t

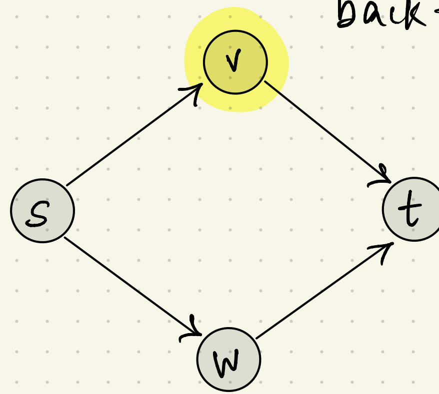
TOPOLOGICAL ORDERING via DFS



$f[t] = 4$

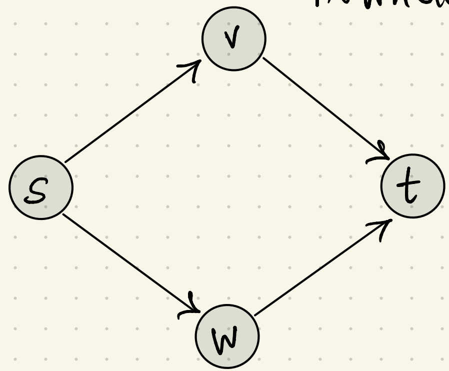
TOPOLOGICAL ORDERING via DFS

backtrack to v



$f[t] = 4$

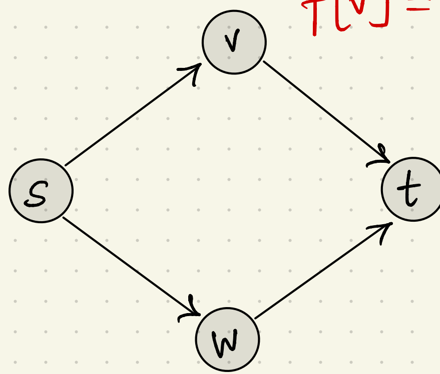
TOPOLOGICAL ORDERING via DFS



nowhere else to go from v

$f[t] = 4$

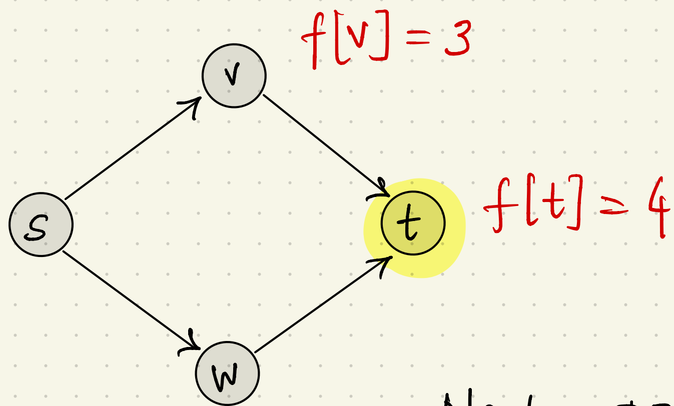
TOPOLOGICAL ORDERING via DFS



$f[v] = 3$

$f[t] = 4$

TOPOLOGICAL ORDERING via DFS



Next, DFS-Loop considers t

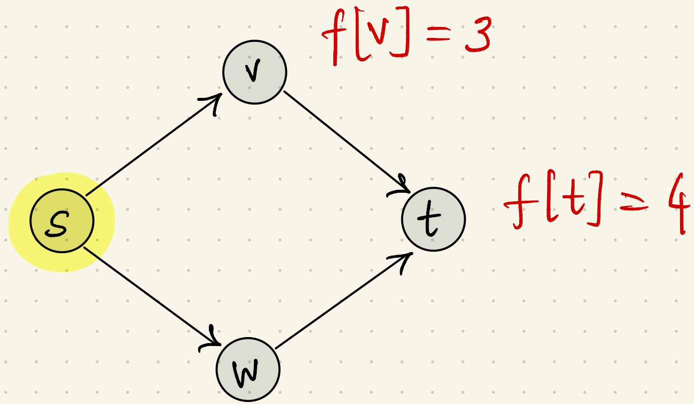
but it is already explored

so skip it.

TOPOLOGICAL ORDERING via DFS

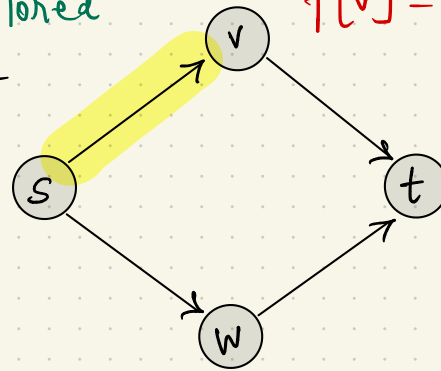
Next, DFS-Loop

Considers s.



TOPOLOGICAL ORDERING via DFS

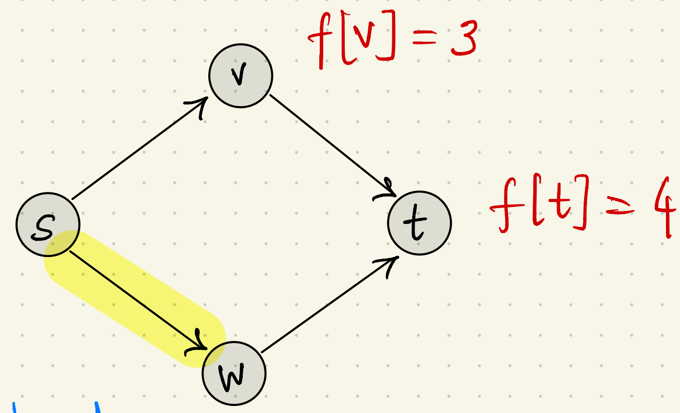
v is explored
so skip it



$f[v] = 3$

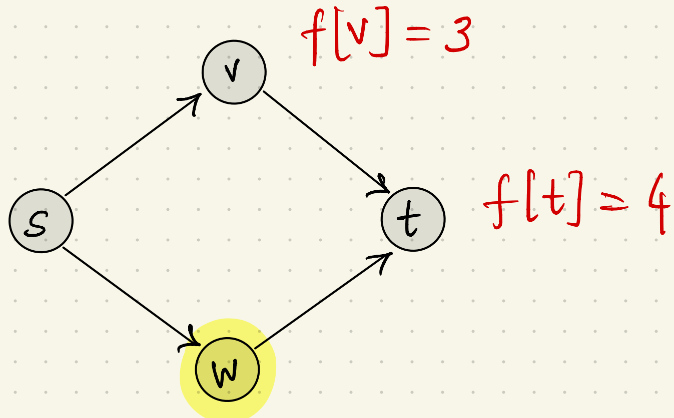
$f[t] = 4$

TOPOLOGICAL ORDERING via DFS



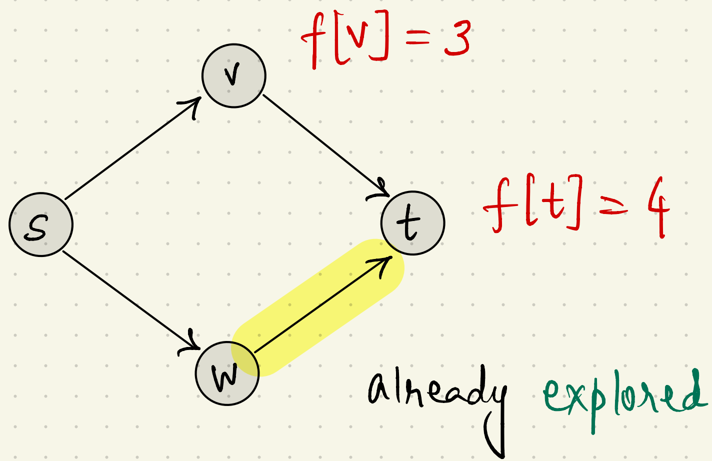
w is unexplored

TOPOLOGICAL ORDERING via DFS

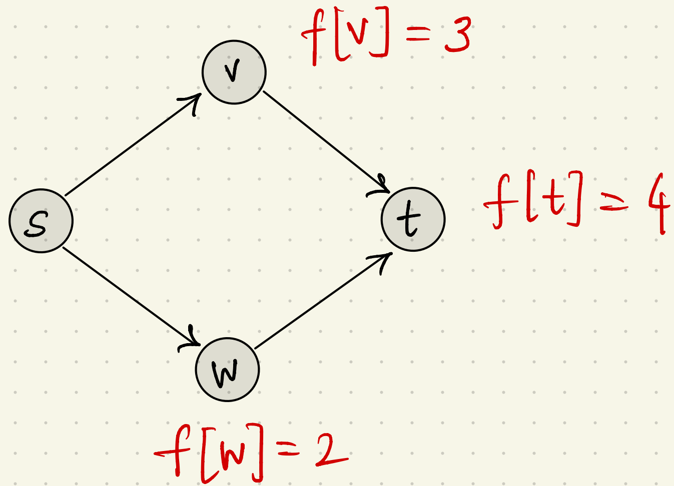


Call DFS on w

TOPOLOGICAL ORDERING via DFS

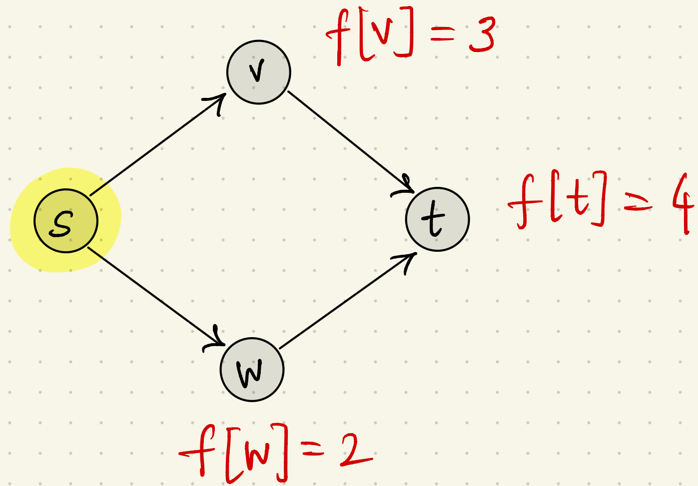


TOPOLOGICAL ORDERING via DFS

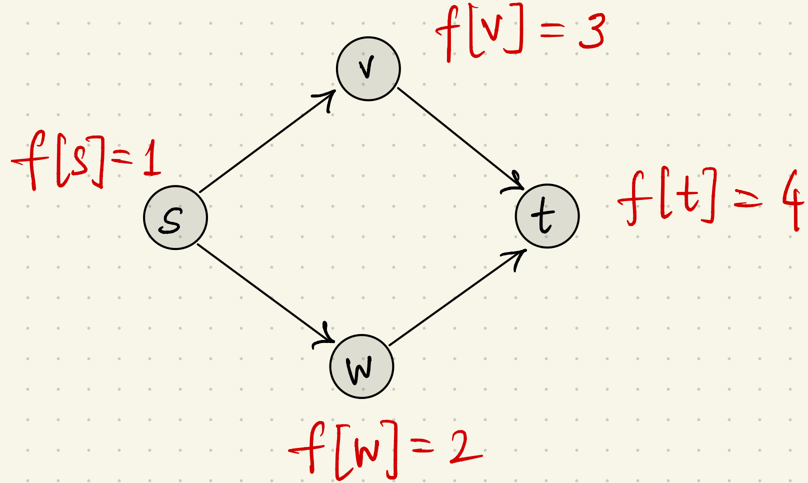


TOPOLOGICAL ORDERING via DFS

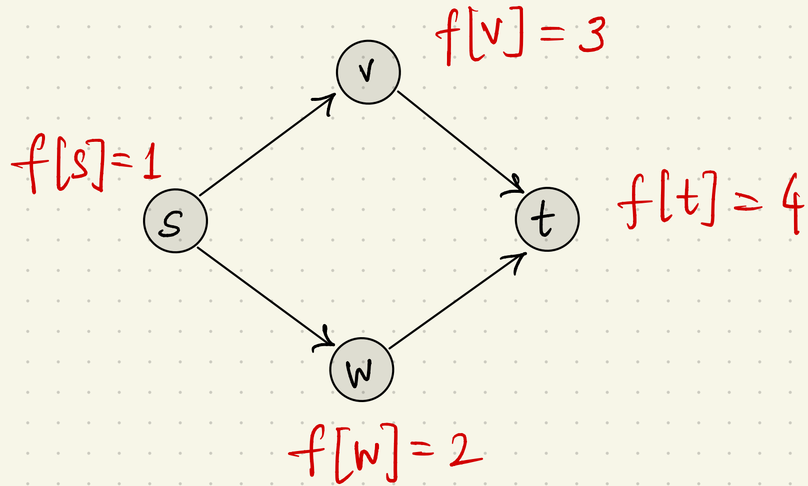
backtrack to s



TOPOLOGICAL ORDERING via DFS



TOPOLOGICAL ORDERING via DFS



DFS-Loop runs out of vertices ⇒ done!

TOPOLOGICAL ORDERING via DFS

Claim 1: DFS-Loop algorithm runs in $O(m+n)$ time.

TOPOLOGICAL ORDERING via DFS

Claim 1: DFS-Loop algorithm runs in $O(m+n)$ time.

Claim 2: Under DFS-Loop,

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

TOPOLOGICAL ORDERING via DFS

Claim 1: DFS-Loop algorithm runs in $O(m+n)$ time.

Claim 2: Under DFS-Loop,

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

Proof sketch: If u visited before v $\Rightarrow$

TOPOLOGICAL ORDERING via DFS

Claim 1: DFS-Loop algorithm runs in $O(m+n)$ time.

Claim 2: Under DFS-Loop,

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

Proof sketch: If u visited before v $\Rightarrow$ recursive call for v finishes before that of u

TOPOLOGICAL ORDERING via DFS

Claim 1: DFS-Loop algorithm runs in $O(m+n)$ time.

Claim 2: Under DFS-Loop,

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

Proof sketch: If u visited before v $\Rightarrow$ recursive call for v finishes before that of u

$\Rightarrow$ u's label $<$ v's label

TOPOLOGICAL ORDERING via DFS

Claim 1: DFS-Loop algorithm runs in $O(m+n)$ time.

Claim 2: Under DFS-Loop,

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

Proof sketch: If u visited before v $\Rightarrow$ recursive call for v finishes before that of u

$\Rightarrow$ u's label $<$ v's label

If v visited before u $\Rightarrow$

TOPOLOGICAL ORDERING via DFS

Claim 1: DFS-Loop algorithm runs in $O(m+n)$ time.

Claim 2: Under DFS-Loop,

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

Proof sketch: If u visited before v $\Rightarrow$ recursive call for v finishes before that of u

$\Rightarrow$ u's label $<$ v's label

If v visited before u $\Rightarrow$ no $v \rightsquigarrow u$ path (no cycle!)

TOPOLOGICAL ORDERING via DFS

Claim 1: DFS-Loop algorithm runs in $O(m+n)$ time.

Claim 2: Under DFS-Loop,

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

Proof sketch: If u visited before v $\Rightarrow$ recursive call for v finishes before that of u

$\Rightarrow$ u's label $<$ v's label

If v visited before u $\Rightarrow$ no $v \rightsquigarrow u$ path (no cycle!)

$\Rightarrow$ DFS(v) won't discover u

TOPOLOGICAL ORDERING via DFS

Claim 1: DFS-Loop algorithm runs in $O(m+n)$ time.

Claim 2: Under DFS-Loop,

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

Proof sketch: If u visited before v $\Rightarrow$ recursive call for v finishes before that of u

$\Rightarrow$ u's label $<$ v's label

If v visited before u $\Rightarrow$ no $v \rightsquigarrow u$ path (no cycle!)

$\Rightarrow$ DFS(v) won't discover u

$\Rightarrow$ v's recursive call finishes before u's. $\blacksquare$

TOPOLOGICAL ORDERING via DFS

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

TOPOLOGICAL ORDERING via DFS

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

not necessarily a DAG

not topological ordering

APPLICATIONS OF DFS

Topological ordering

Strongly Connected Components

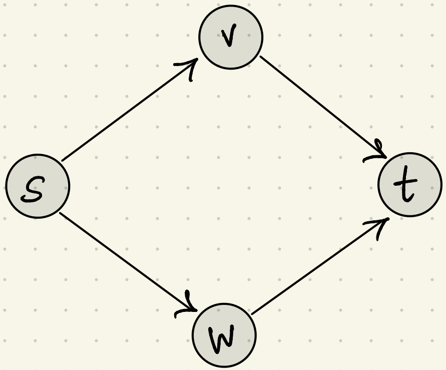
APPLICATIONS OF DFS

Topological ordering

Strongly Connected Components

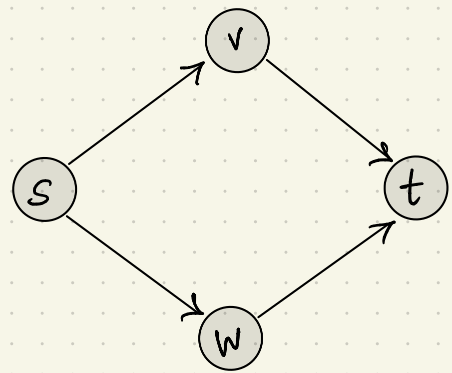
STRONGLY CONNECTED COMPONENTS

STRONGLY CONNECTED COMPONENTS



Connected ?

STRONGLY CONNECTED COMPONENTS

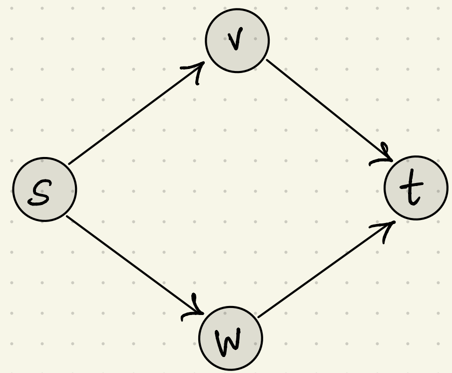


Connected?

Yes if one looks at the underlying undirected graph

No if one cares about reachability via directed edges

STRONGLY CONNECTED COMPONENTS

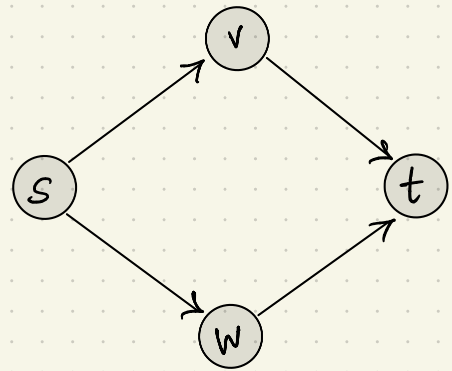


Connected ?

Yes if one looks at the underlying undirected graph

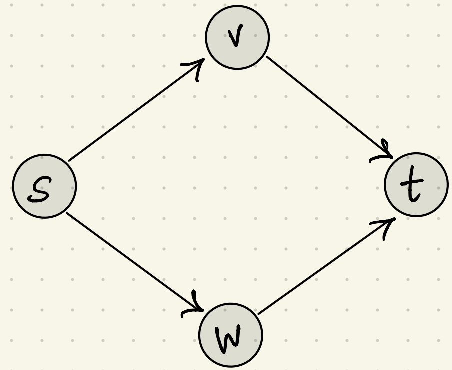
No if one cares about reachability via directed edges

STRONGLY CONNECTED COMPONENTS



A directed graph is strongly connected if every vertex can be reached from every other vertex by a directed path.

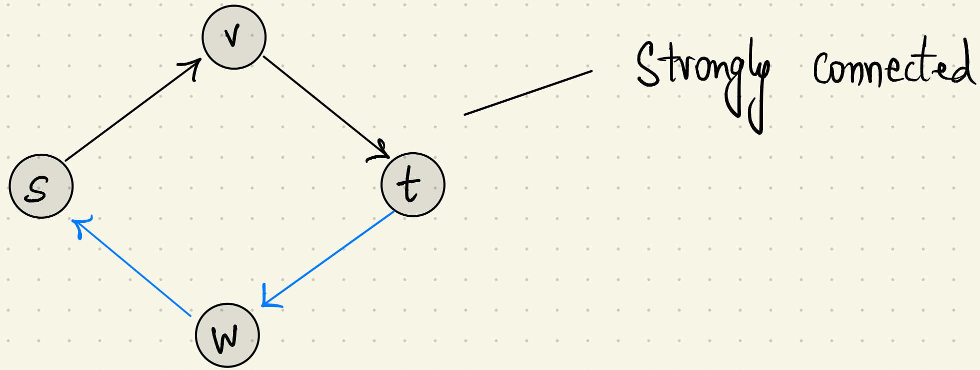
STRONGLY CONNECTED COMPONENTS



Not strongly connected

A directed graph is strongly connected if every vertex can be reached from every other vertex by a directed path.

STRONGLY CONNECTED COMPONENTS



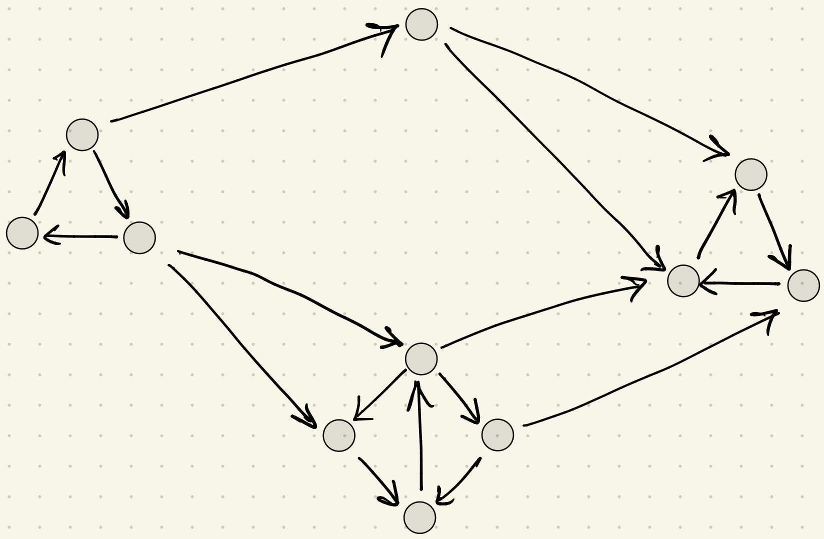
A directed graph is strongly connected if every vertex can be reached from every other vertex by a directed path.

STRONGLY CONNECTED COMPONENTS

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

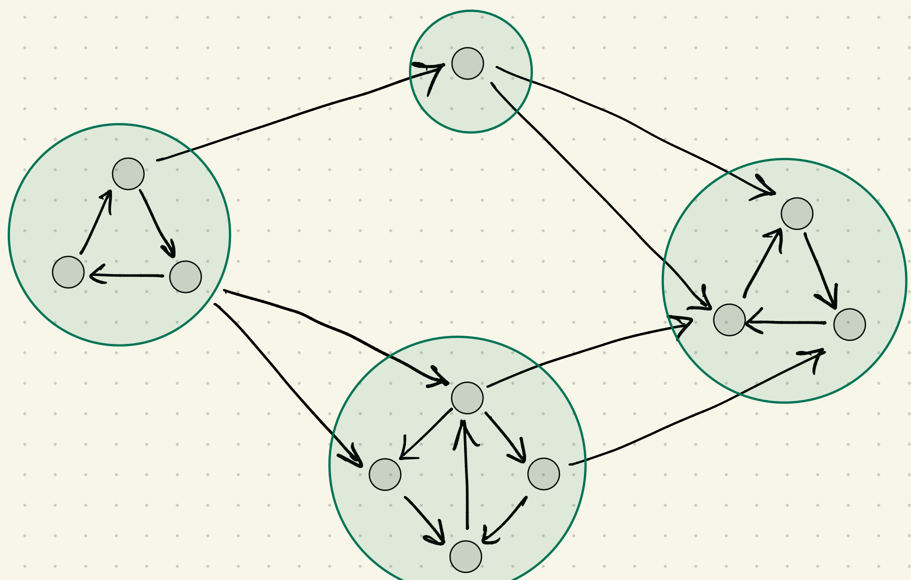
STRONGLY CONNECTED COMPONENTS

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



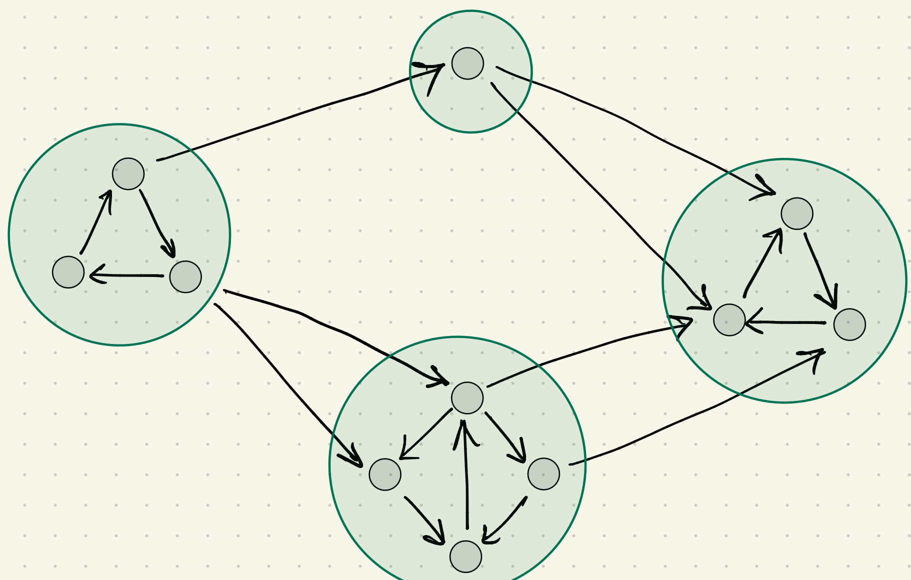
STRONGLY CONNECTED COMPONENTS

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



STRONGLY CONNECTED COMPONENTS

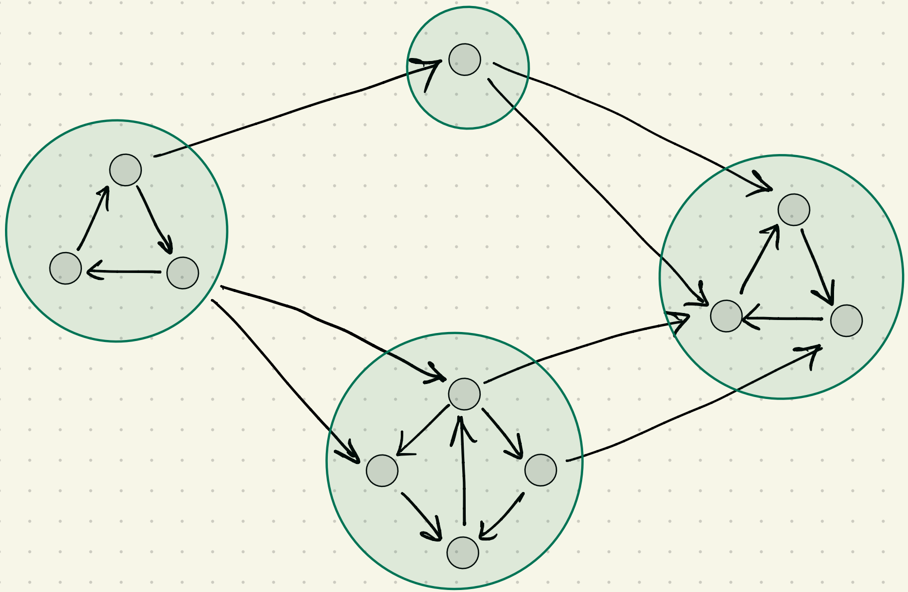
A strongly connected component of a directed graph G is a maximal subgraph of G that is strongly connected.

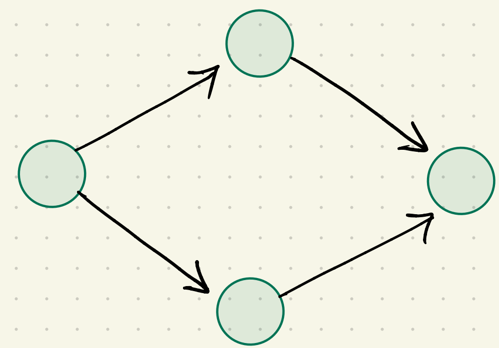


A DAG!

STRONGLY CONNECTED COMPONENTS

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

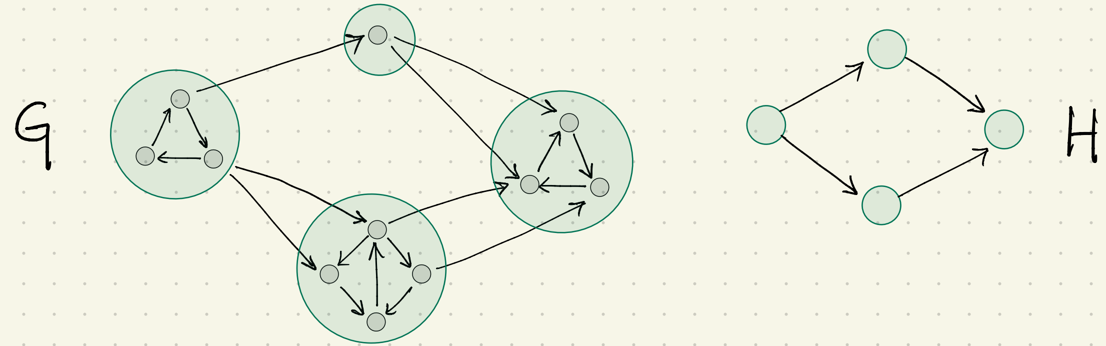




STRONGLY CONNECTED COMPONENTS

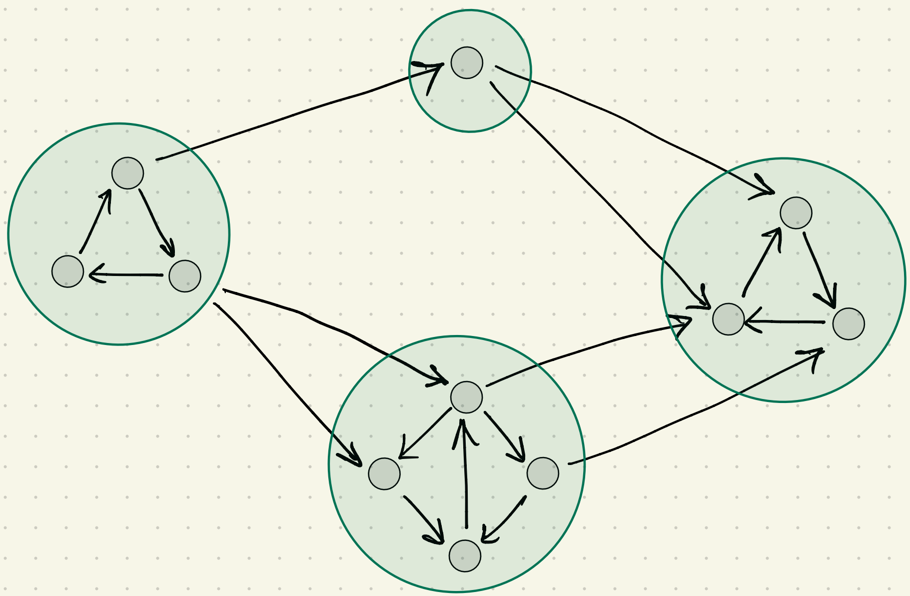
Theorem: Given a directed graph G, define a "meta" graph $H = (X, F)$ that has a vertex for every SCC of G, and an edge $(x, y) \in F$ if there is an edge from some vertex in SCC corresponding to x to some vertex in SCC corresponding to y.

Then, H is a DAG.

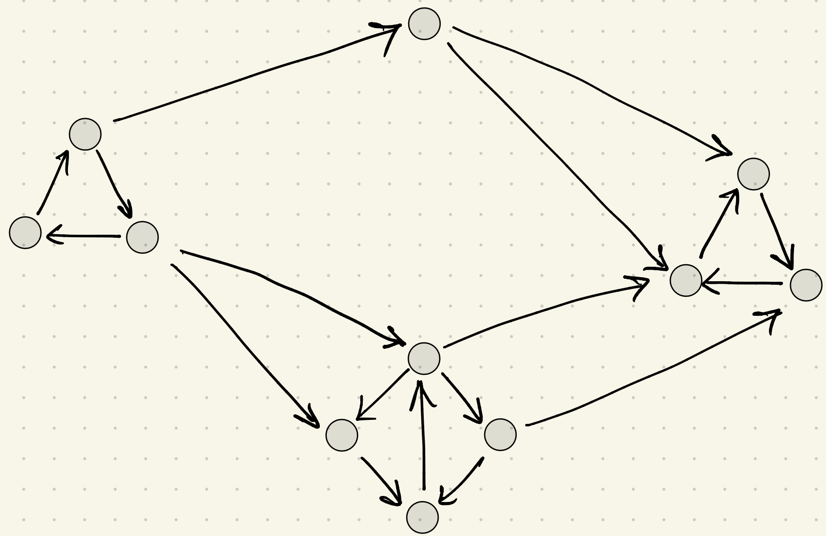


STRONGLY CONNECTED COMPONENTS via DFS

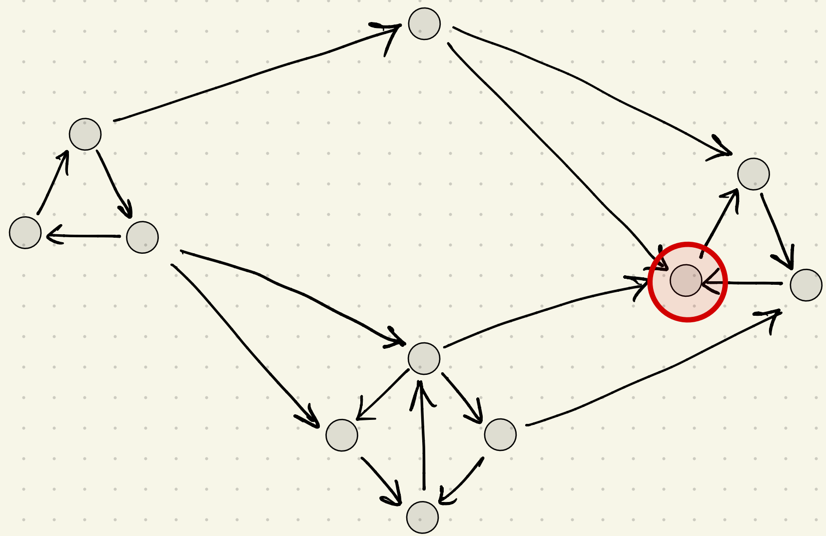
STRONGLY CONNECTED COMPONENTS via DFS



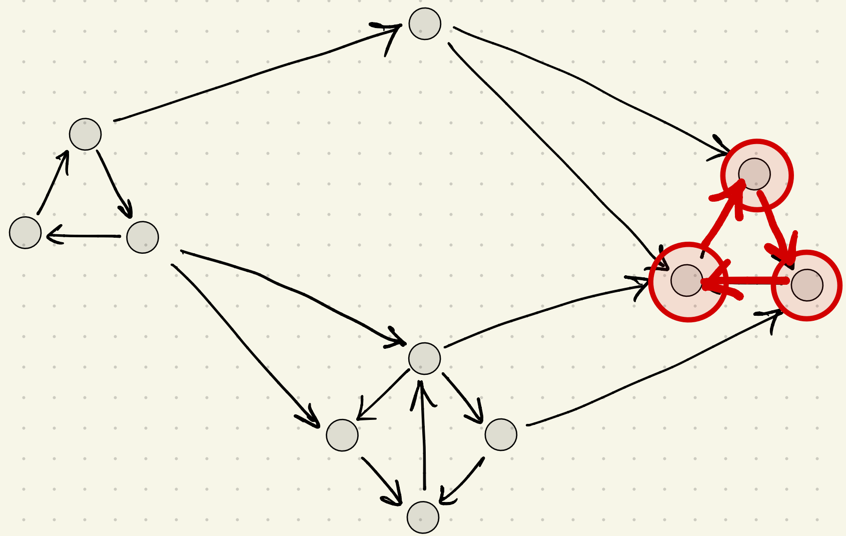
STRONGLY CONNECTED COMPONENTS via DFS



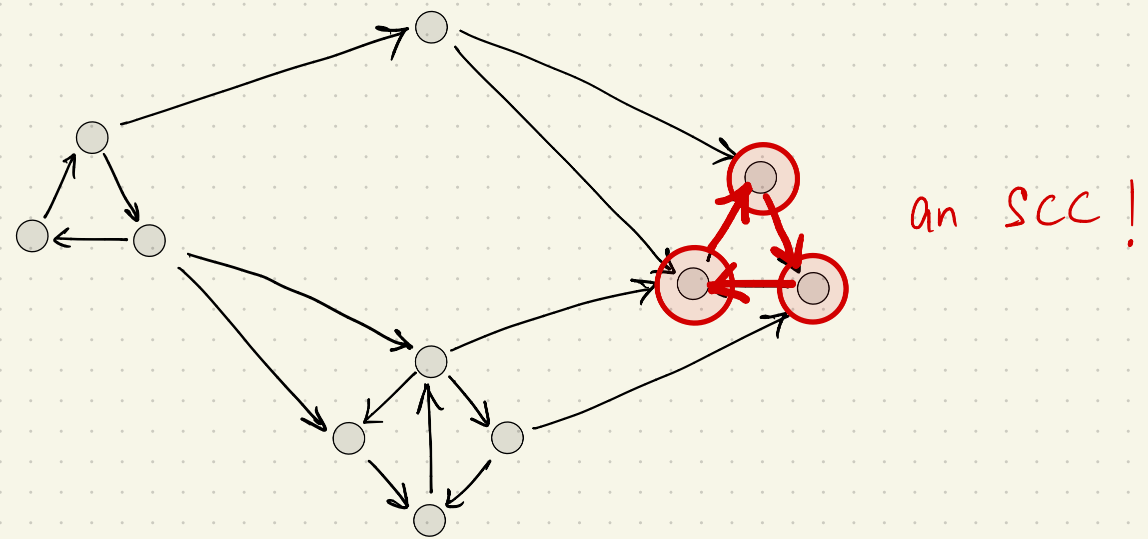
STRONGLY CONNECTED COMPONENTS via DFS



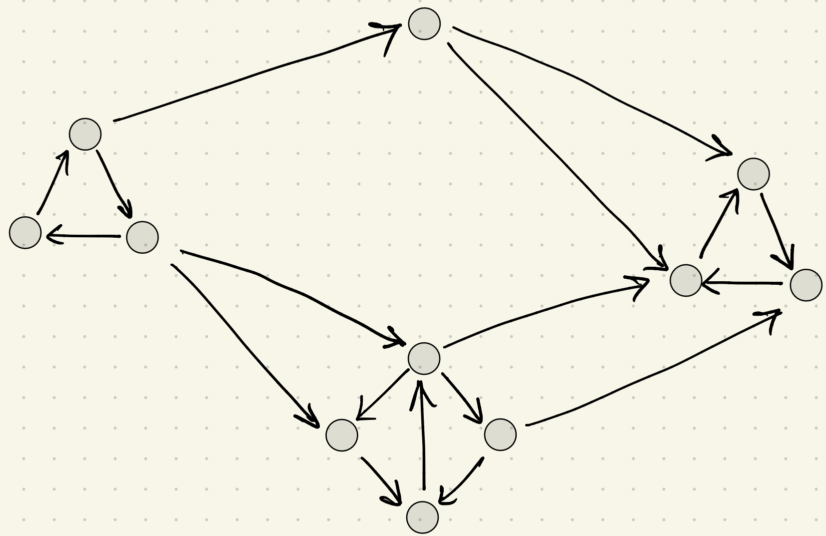
STRONGLY CONNECTED COMPONENTS via DFS



STRONGLY CONNECTED COMPONENTS via DFS

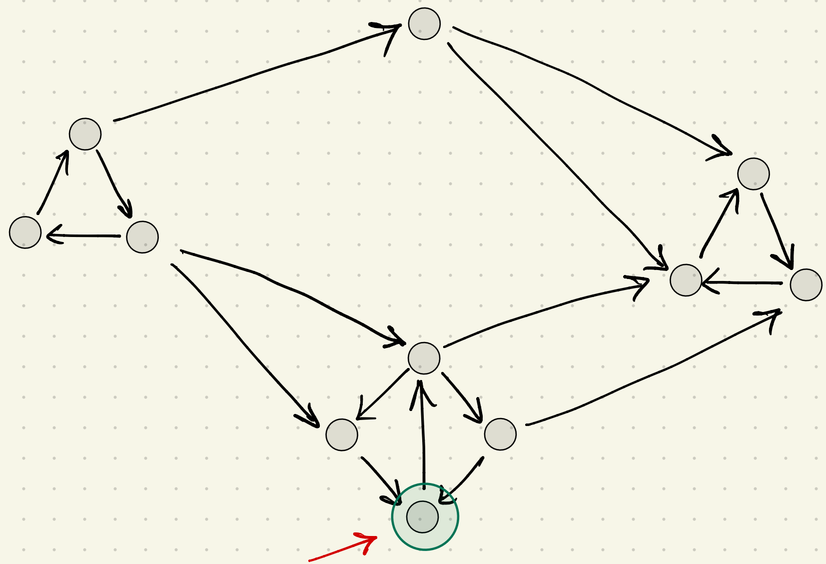


STRONGLY CONNECTED COMPONENTS via DFS



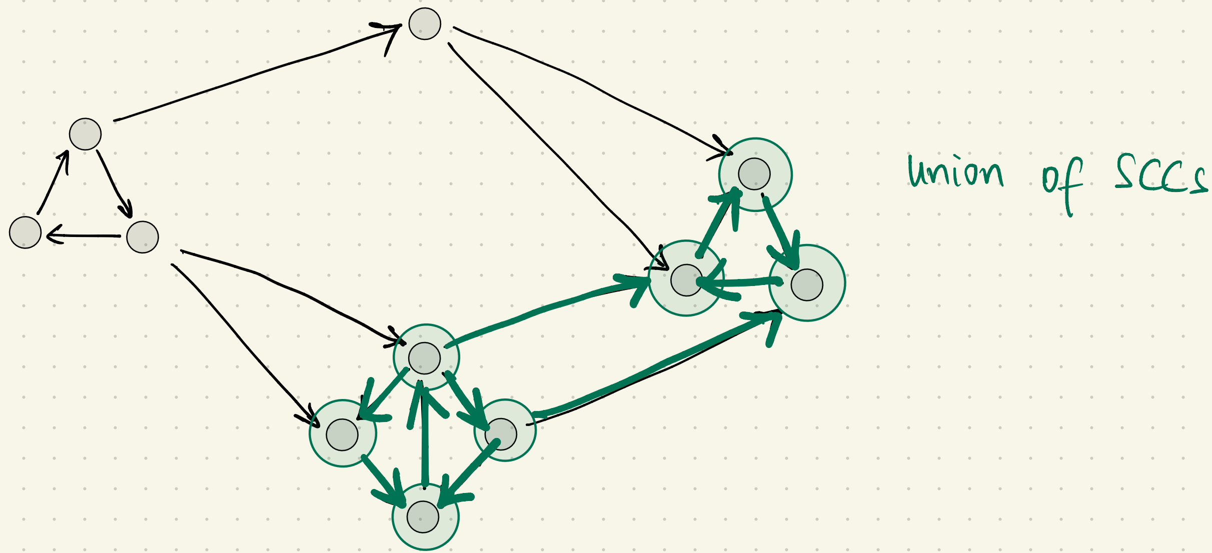
🤔 Maybe do a DFS from every node to identify all SCCs ?

STRONGLY CONNECTED COMPONENTS via DFS



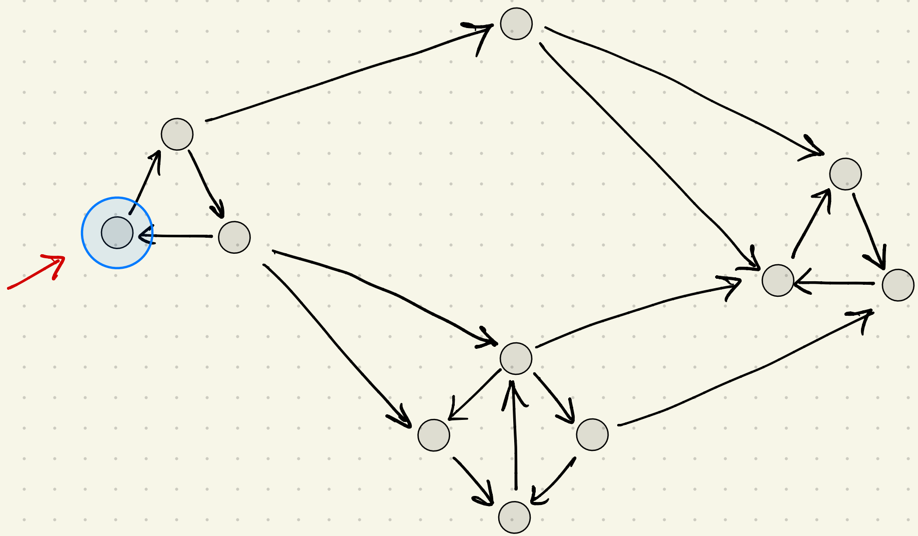
🤔 Maybe do a DFS from every node to identify all SCCs ?

STRONGLY CONNECTED COMPONENTS via DFS



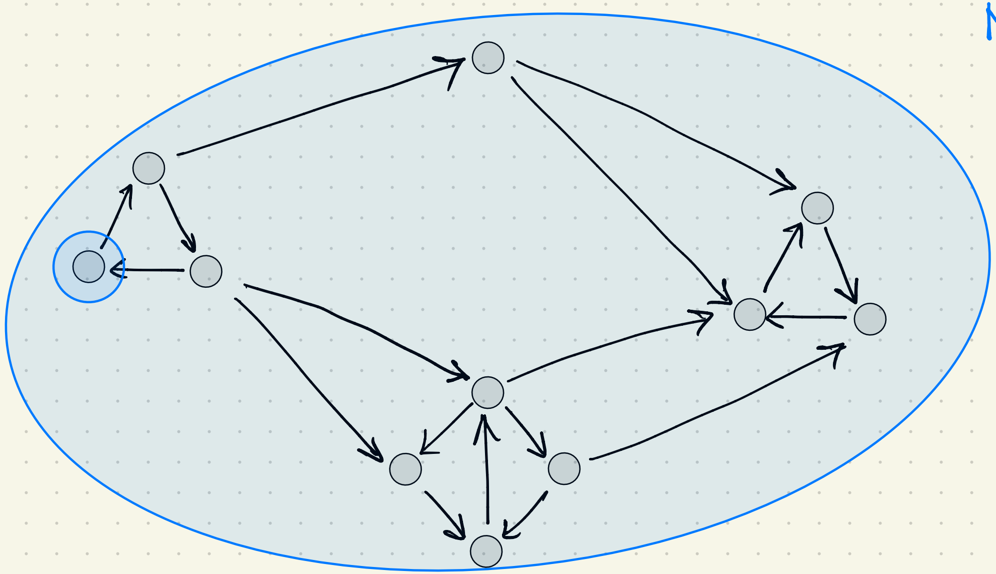
🤔 Maybe do a DFS from every node to identify all SCCs ?

STRONGLY CONNECTED COMPONENTS via DFS



🤔 Maybe do a DFS from every node to identify all SCCs ?

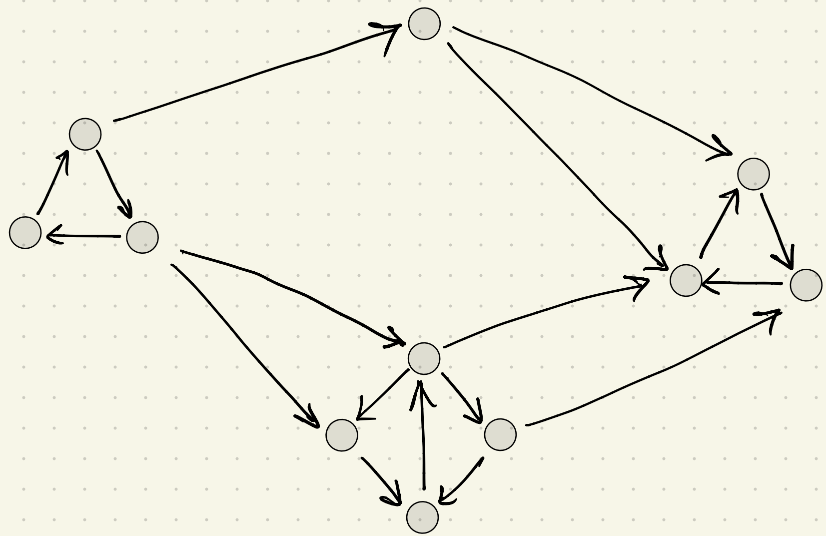
STRONGLY CONNECTED COMPONENTS via DFS



No information at all!

🤔 Maybe do a DFS from every node to identify all SCCs?

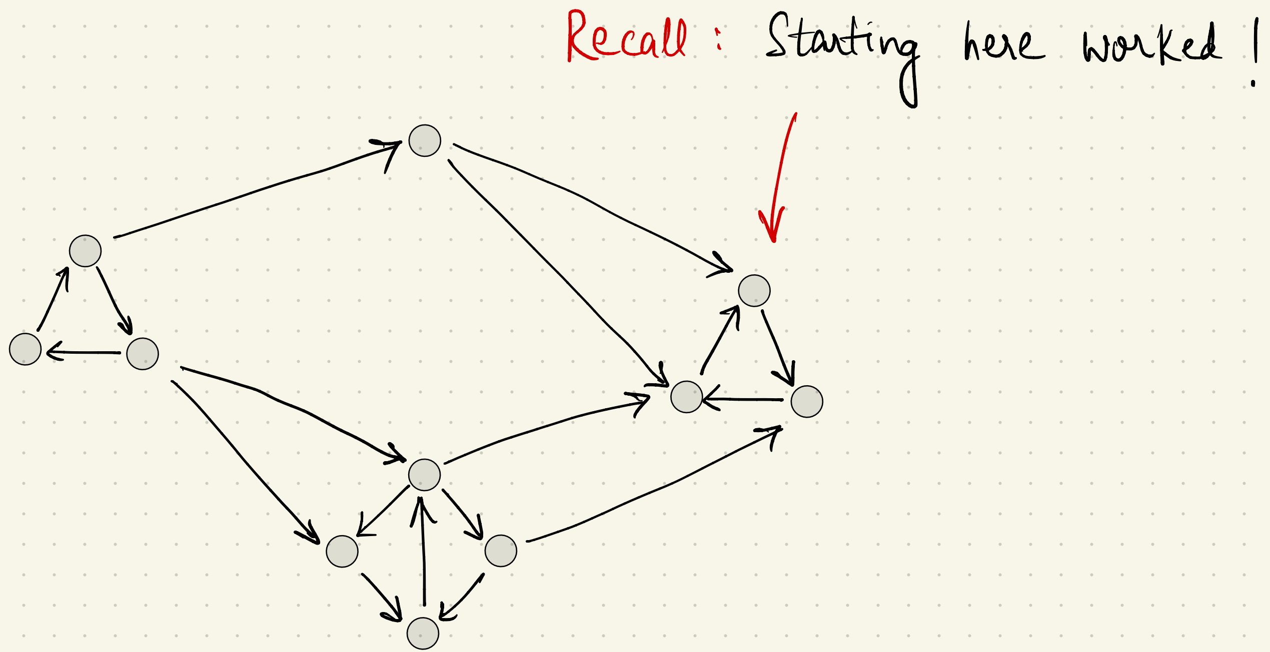
STRONGLY CONNECTED COMPONENTS via DFS



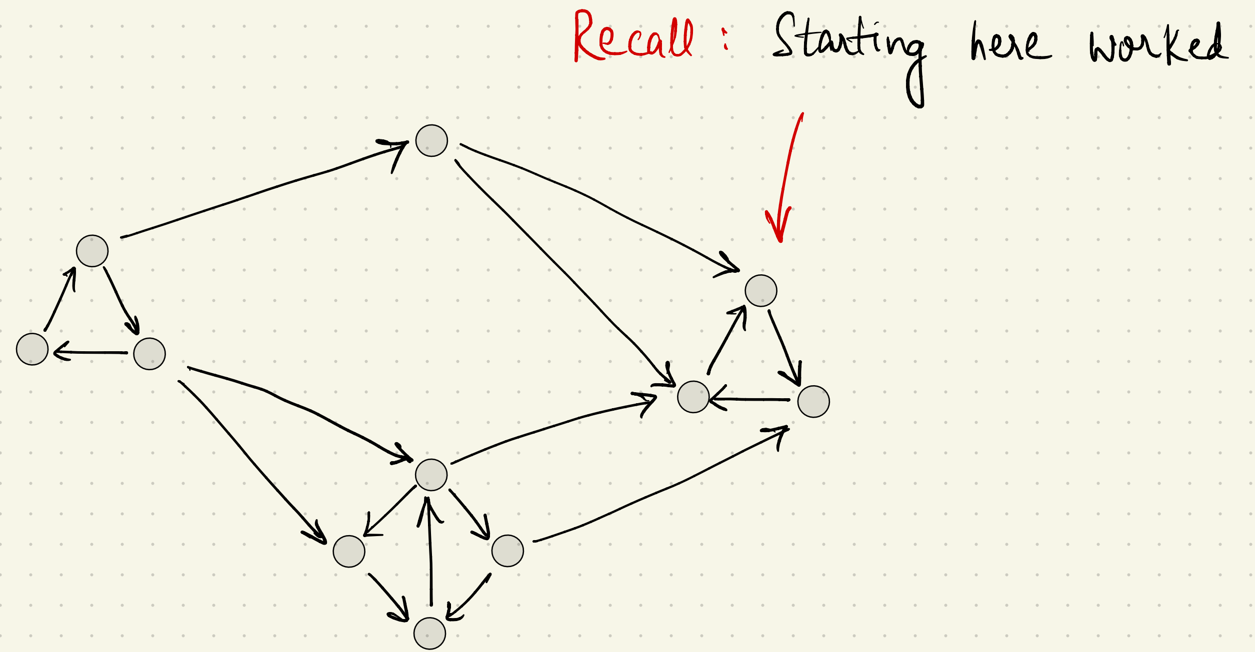
Starting point matters!

First Attempt

FIRST ATTEMPT



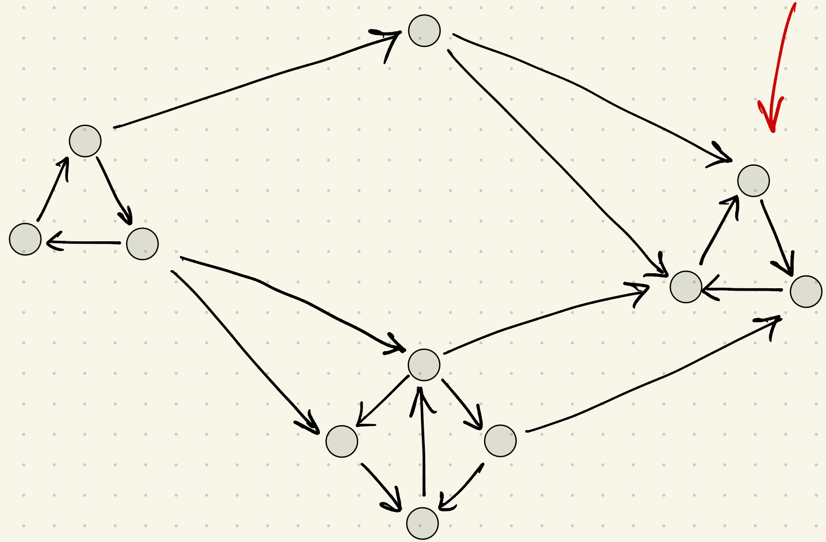
FIRST ATTEMPT



Start DFS at the "last" vertex
(as per topological ordering algo)

FIRST ATTEMPT

Recall : Starting here worked !

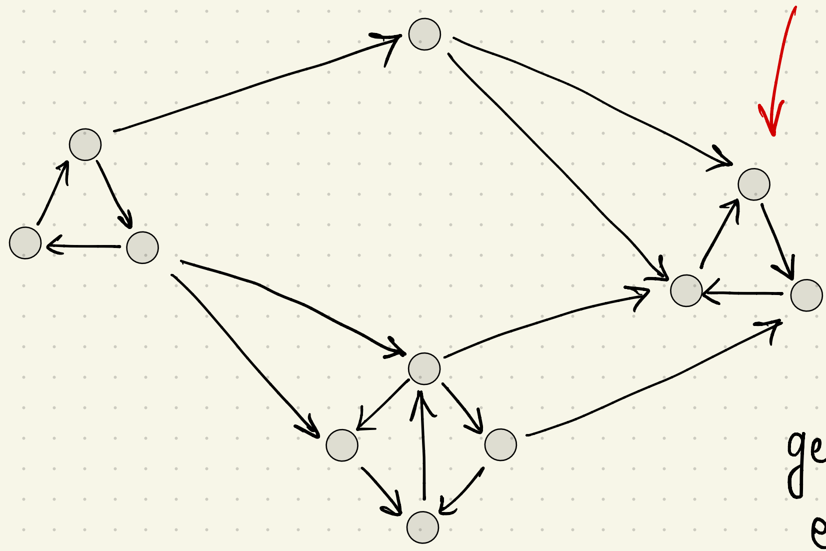


maybe it lies in the "sink" SCC ?

Start DFS at the "last" vertex
(as per topological ordering algo)

FIRST ATTEMPT

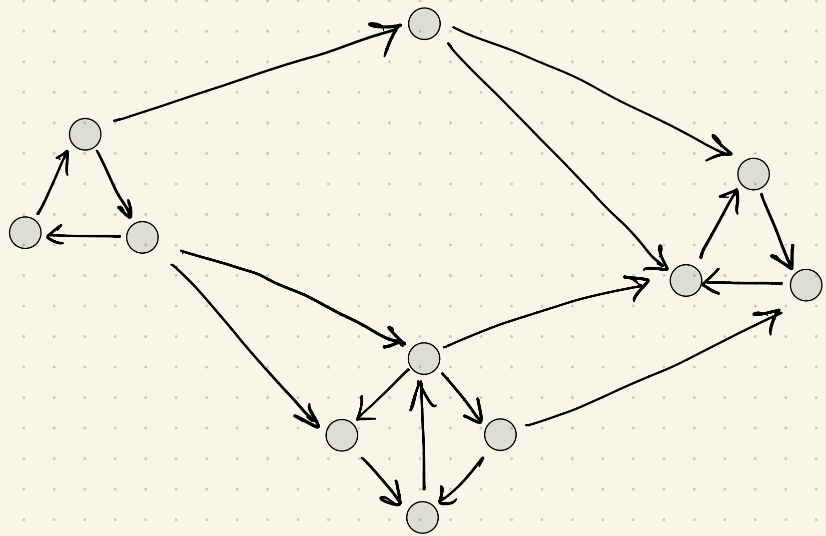
Recall : Starting here worked !



generates some ordering
even for cyclic graphs

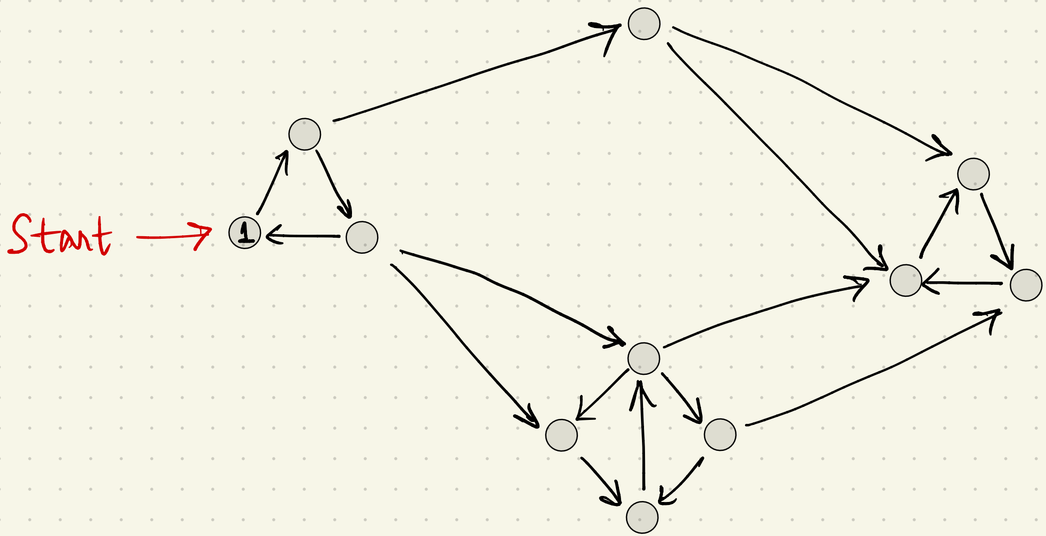
Start DFS at the "last" vertex
(as per topological ordering algo)

FIRST ATTEMPT

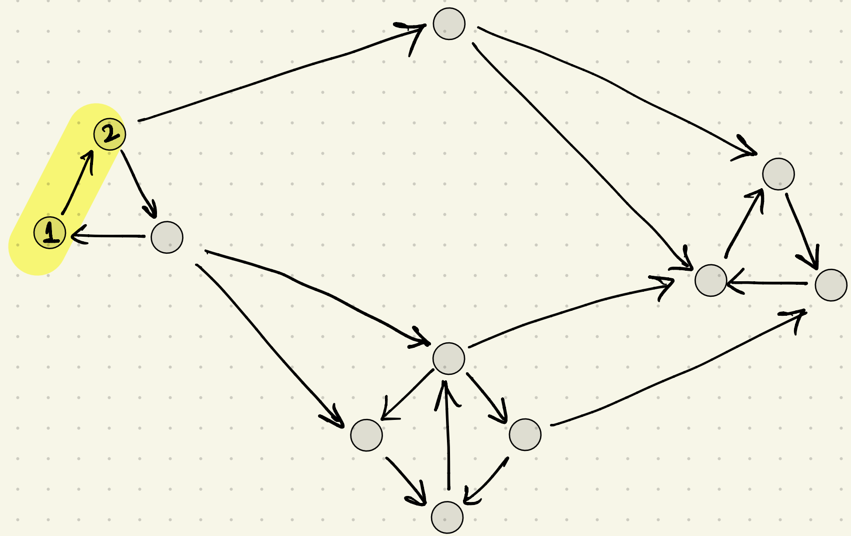


Let's run topological ordering (via DFS) and see what happens.

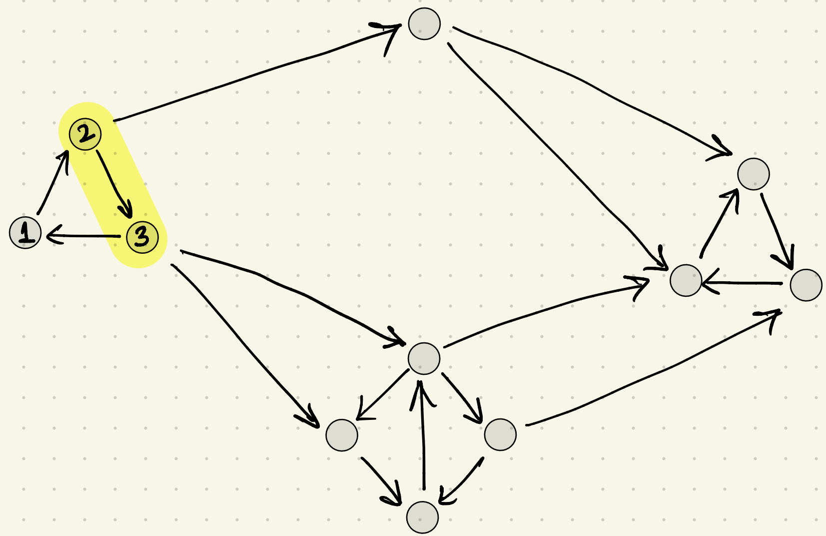
FIRST ATTEMPT



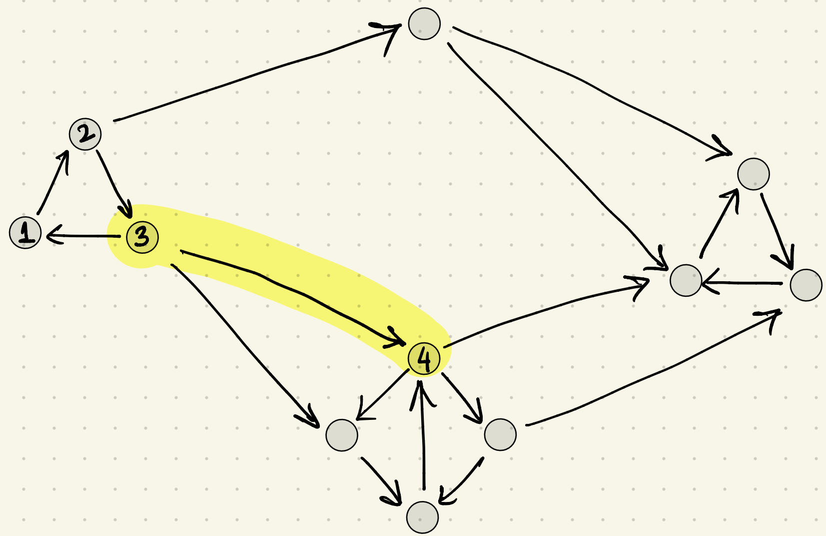
FIRST ATTEMPT



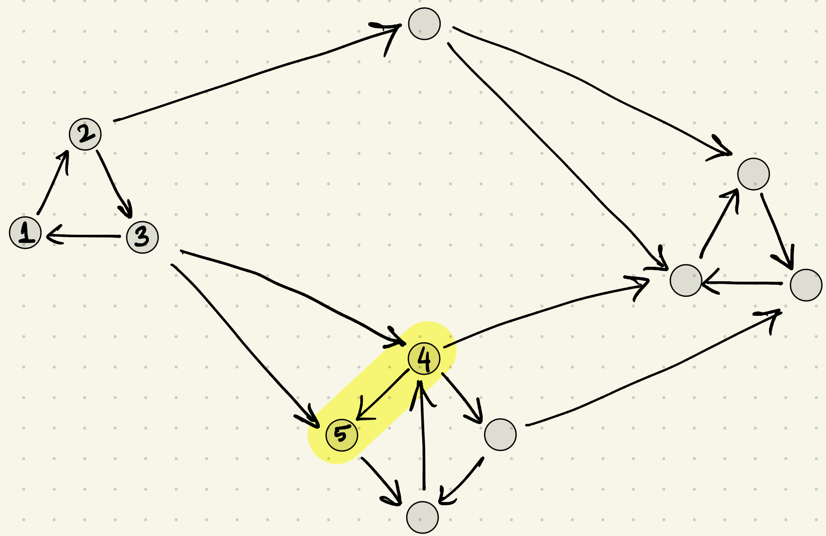
FIRST ATTEMPT



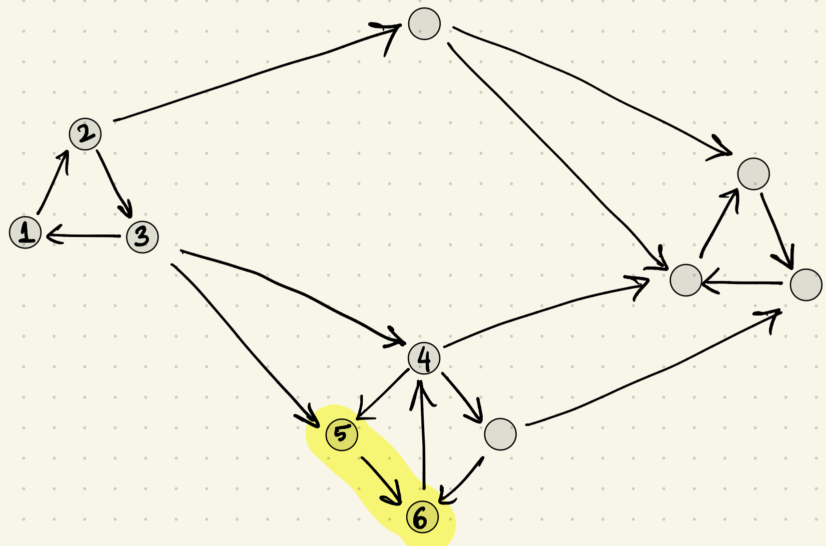
FIRST ATTEMPT



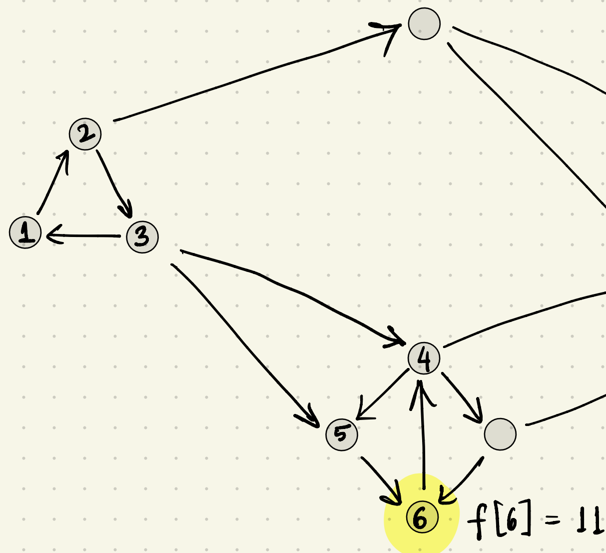
FIRST ATTEMPT



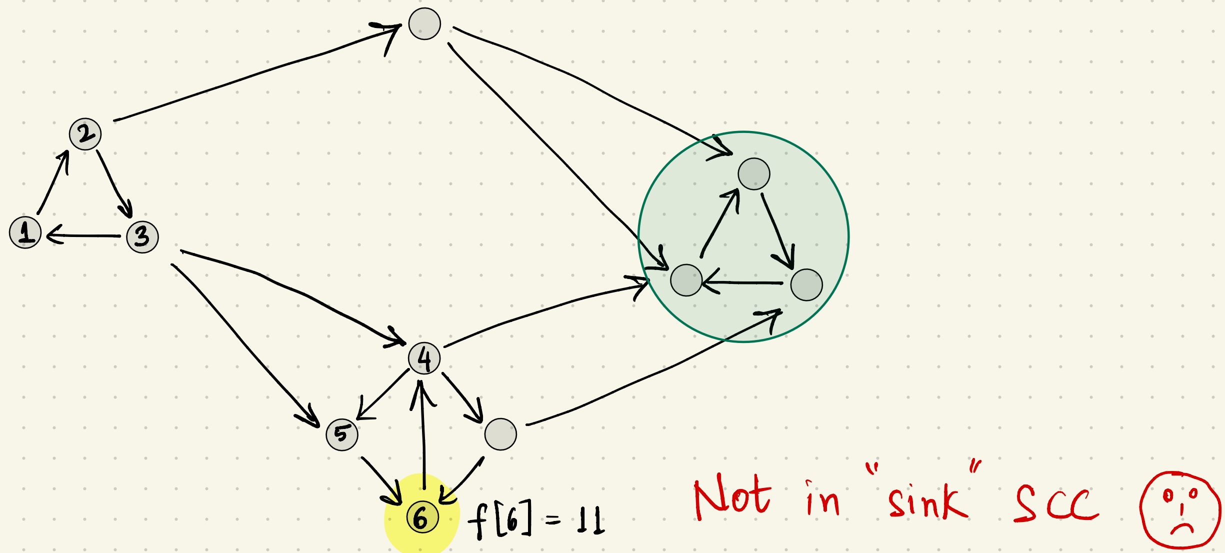
FIRST ATTEMPT



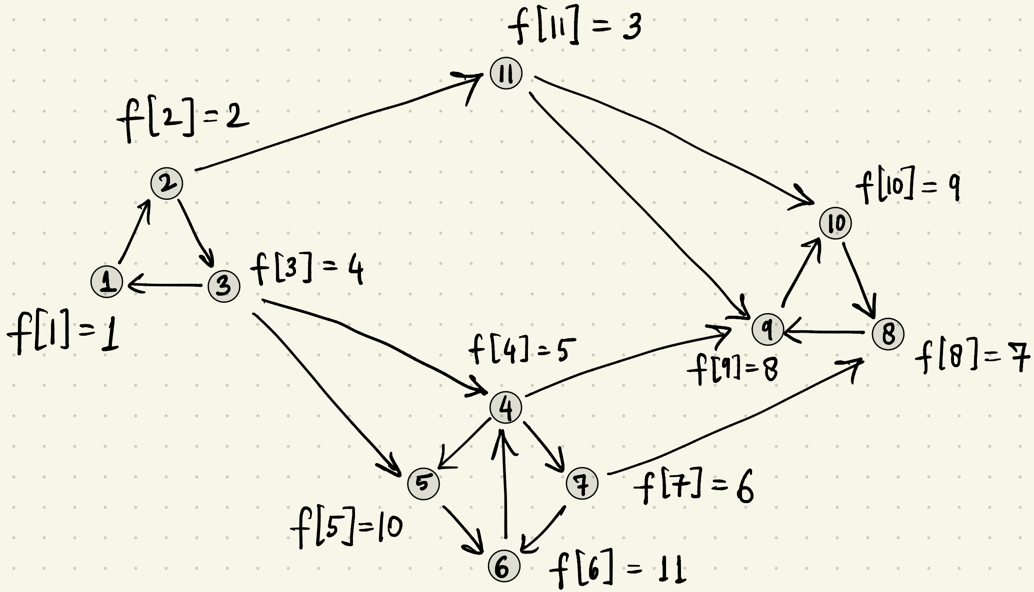
FIRST ATTEMPT



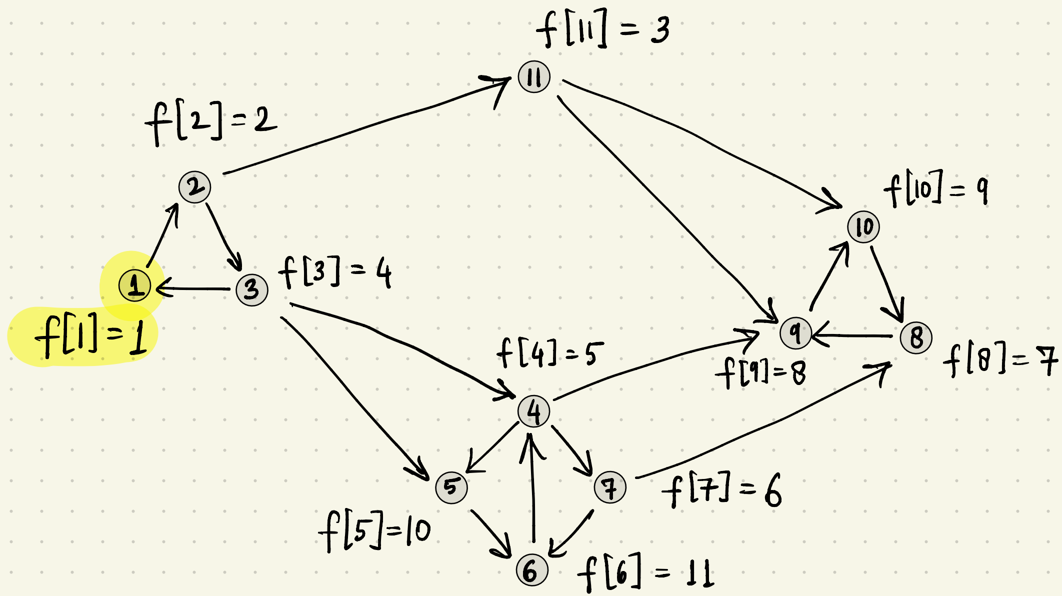
FIRST ATTEMPT



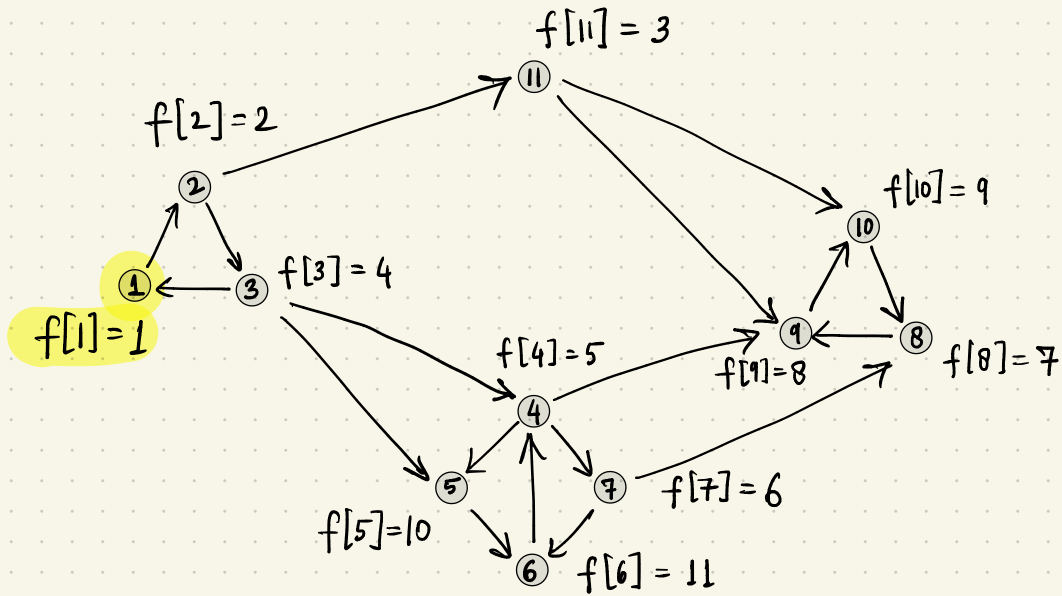
FIRST ATTEMPT



FIRST ATTEMPT

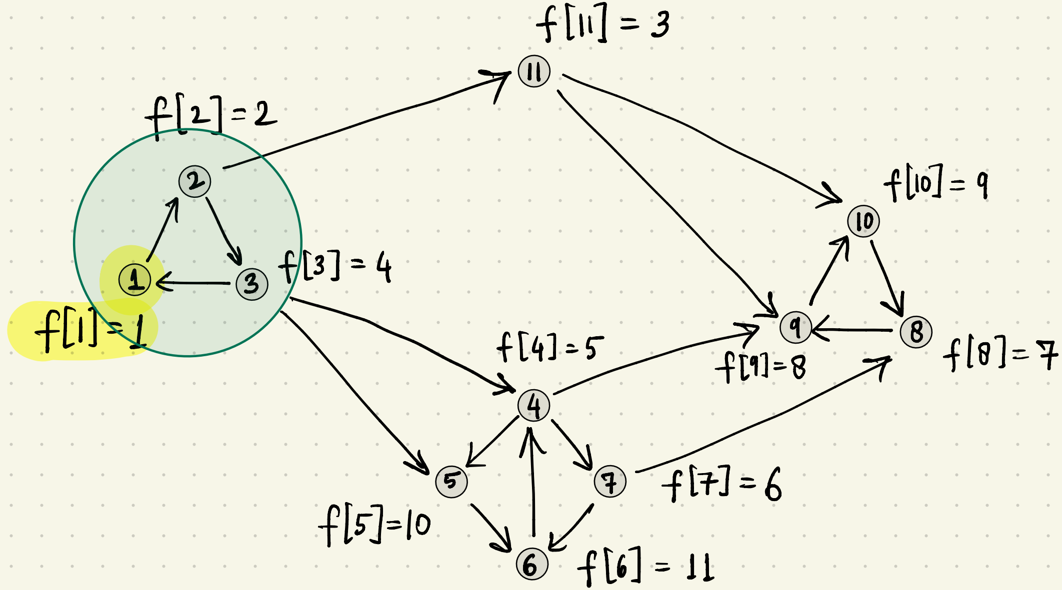


FIRST ATTEMPT



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

FIRST ATTEMPT



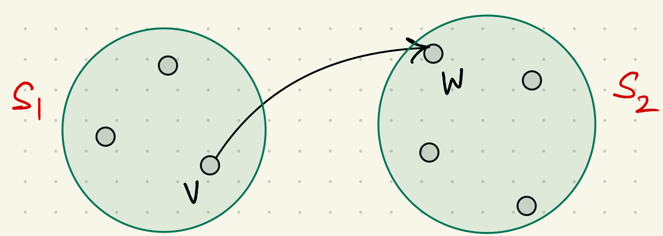
Perhaps the "smallest-label" vertex is always in "source" SCC ?
(as per topological ordering)

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,

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

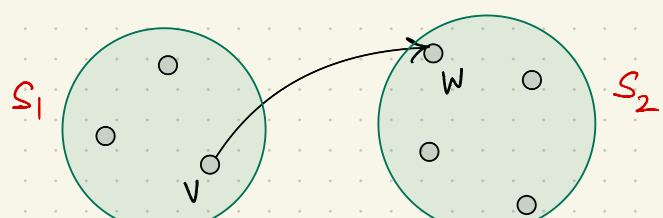


KEY OBSERVATION

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,

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

Generalization of "every DAG has a 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,

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

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

Proof:

Proof: Case I : Topological algo. discovers some vertex in S_1 before S_2.

Case II : Topological algo. discovers some vertex in S_2 before S_1.

Proof: Case I : Topological algo. discovers some vertex in S_1 before S_2.

All vertices in S_2 reachable from some vertex in S_1.

Case II : Topological algo. discovers some vertex in S_2 before S_1.

Proof: Case I: Topological algo. discovers some vertex in S_1 before S_2.

All vertices in S_2 reachable from some vertex in S_1.

$\Rightarrow$ Recursive calls for all vertices in S_2 finish before that of some vertex in S_1.

Case II: Topological algo. discovers some vertex in S_2 before S_1.

Proof: Case I : Topological algo. discovers some vertex in S_1 before S_2.

All vertices in S_2 reachable from some vertex in S_1.

$\Rightarrow$ Recursive calls for all vertices in S_2 finish before that of some vertex in S_1.

Case II : Topological algo. discovers some vertex in S_2 before S_1.

Recall : SCC meta-graph is acyclic.

Proof: Case I: Topological algo. discovers some vertex in S_1 before S_2.

All vertices in S_2 reachable from some vertex in S_1.

$\Rightarrow$ Recursive calls for all vertices in S_2 finish before that of some vertex in S_1.

Case II: Topological algo. discovers some vertex in S_2 before S_1.

Recall: SCC meta-graph is acyclic.

S_1 is unreachable from S_2.

Proof: Case I : Topological algo. discovers some vertex in S_1 before S_2.

All vertices in S_2 reachable from some vertex in S_1.

$\Rightarrow$ Recursive calls for all vertices in S_2 finish before that of some vertex in S_1.

Case II : Topological algo. discovers some vertex in S_2 before S_1.

Recall : SCC meta-graph is acyclic.

S_1 is unreachable from S_2.

$\Rightarrow$ Recursive calls for all vertices in S_2 finish before that of <u>every</u> vertex in S_1.

Proof: Case I: Topological algo. discovers some vertex in S_1 before S_2.

All vertices in S_2 reachable from some vertex in S_1.

$\Rightarrow$ Recursive calls for all vertices in S_2 finish before that of some vertex in S_1.

Case II: Topological algo. discovers some vertex in S_2 before S_1.

Recall: SCC meta-graph is acyclic.

S_1 is unreachable from S_2.

$\Rightarrow$ Recursive calls for all vertices in S_2 finish before that of <u>every</u> vertex in S_1. ∎

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

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

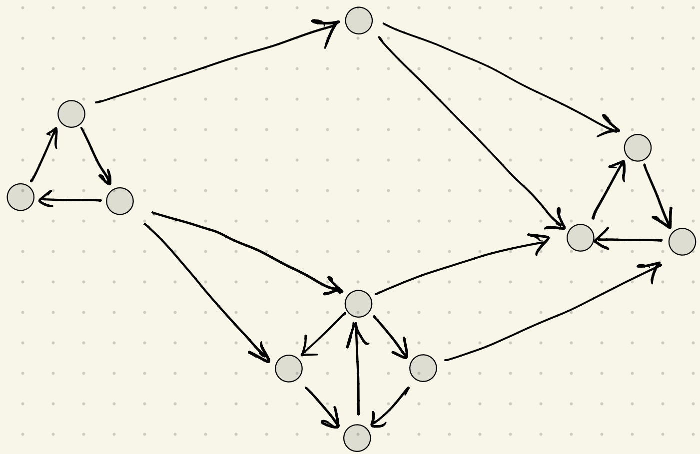
What's the fix:

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

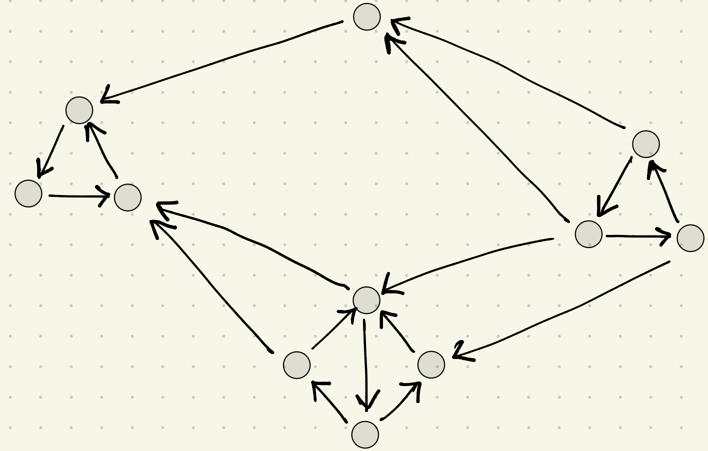
What we want : A way of identifying a vertex in sink SCC.

What's the fix : Reverse the graph!

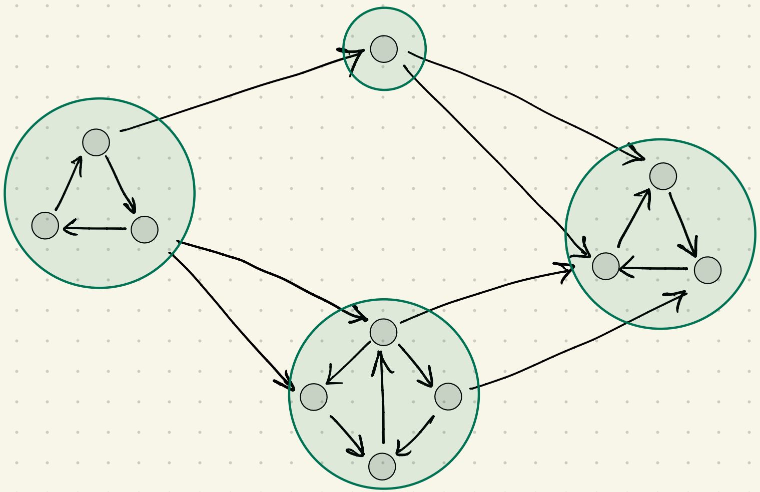
G



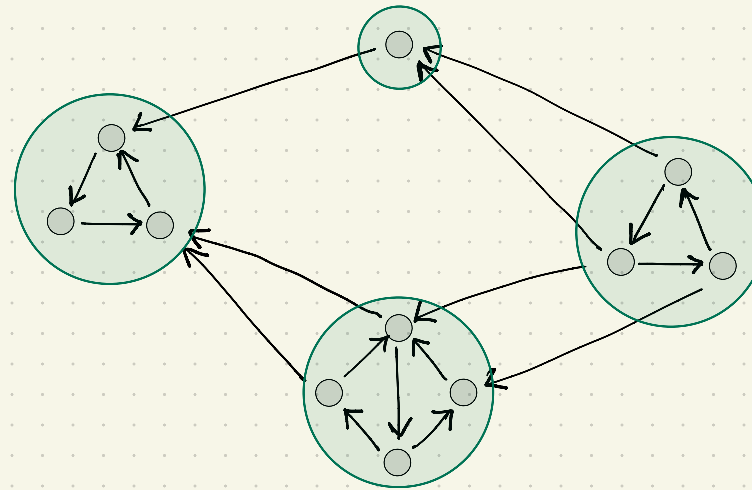
G^{rev}



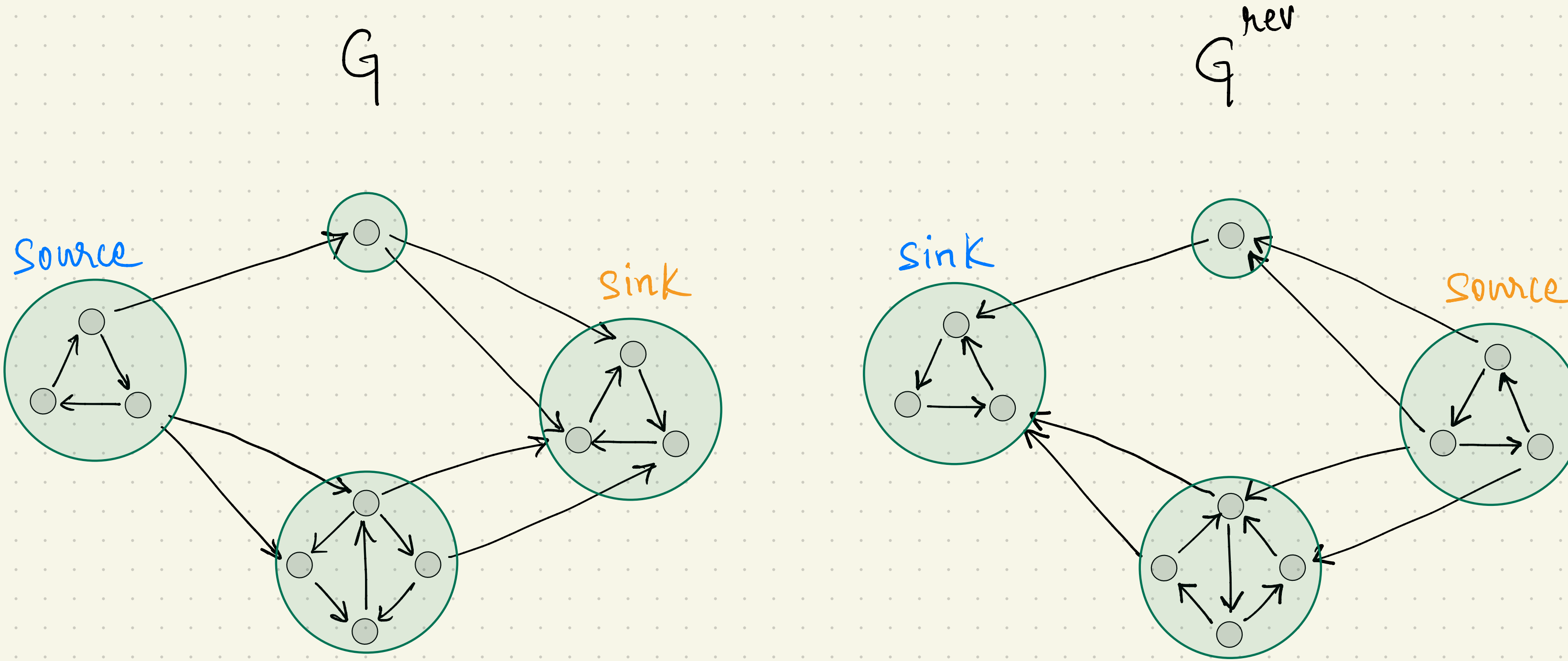
G



G^{rev}



SCCs stay the same!



Source / sink in meta graph of G

Sink / Source " " " G^{rev}