

COL 351 : ANALYSIS & DESIGN OF ALGORITHMS

LECTURE 11

GRAPH ALGORITHMS IV :

TOPOLOGICAL ORDERING & STRONGLY CONNECTED COMPONENTS

AUG 18, 2026

|

ROHIT VAISH

DEPTH-FIRST SEARCH

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)

DEPTH-FIRST SEARCH

Claim 1: At the end of DFS

v is explored $\iff \exists$ path from s to v in G

Reason : DFS is a special case of Generic Algorithm.

DEPTH-FIRST SEARCH

Claim 1: At the end of DFS

v is explored $\iff \exists$ path from s to v in G

Reason: DFS is a special case of Generic Algorithm.

Claim 2: Running time is $O(n_s + m_s)$.

reachable vertices

reachable edges

DEPTH-FIRST SEARCH

Claim 1: At the end of DFS

v is explored $\iff \exists$ path from s to v in G

Reason: DFS is a special case of Generic Algorithm.

Claim 2: Running time is $O(n_s + m_s)$.

reachable vertices

reachable edges

Corollary: DFS can compute connected components in linear time.

APPLICATIONS OF DFS

Topological ordering

Strongly Connected Components

APPLICATIONS OF DFS

Topological ordering

Strongly Connected Components

TOPOLOGICAL ORDERING

Directed graph $G = (V, E)$

TOPOLOGICAL ORDERING

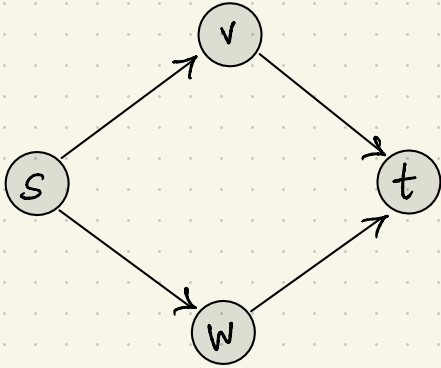
Directed graph $G = (V, E)$

A labeling f of G 's vertices such that :

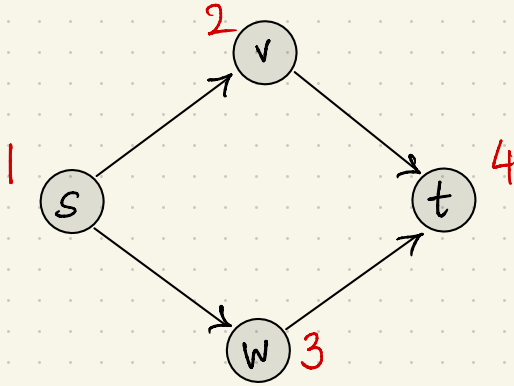
* unique $f(v) \in \{1, 2, \dots, n\}$ for every $v \in V$

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

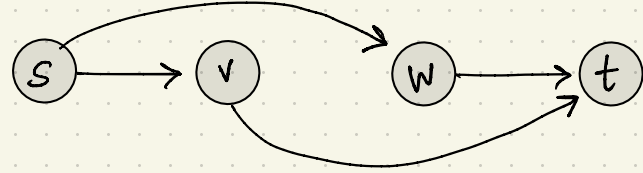
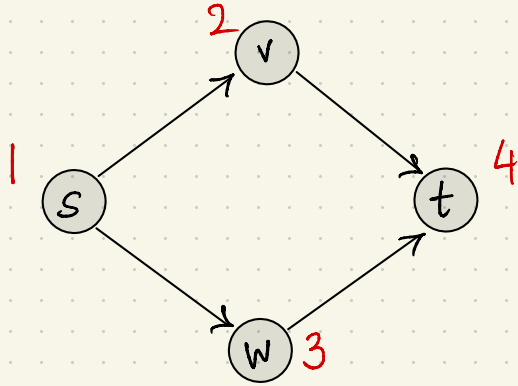
TOPOLOGICAL ORDERING



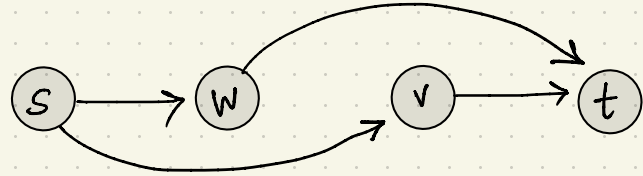
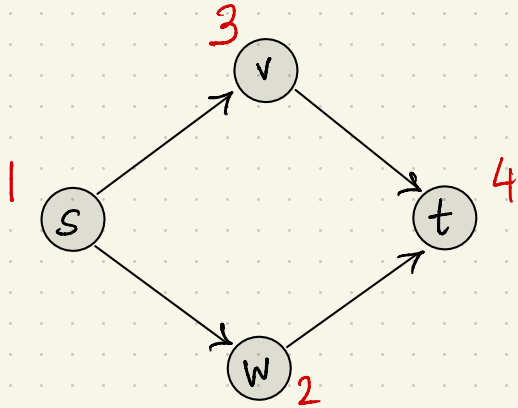
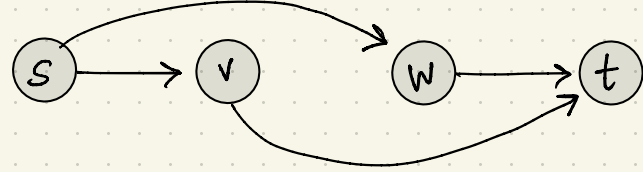
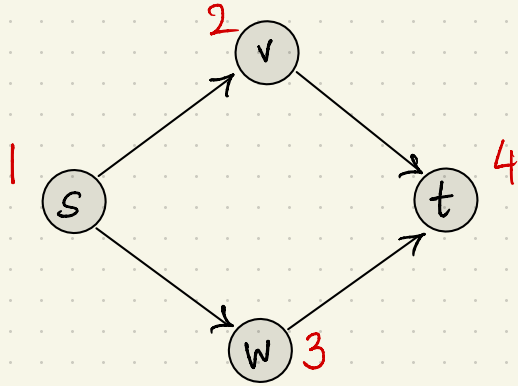
TOPOLOGICAL ORDERING



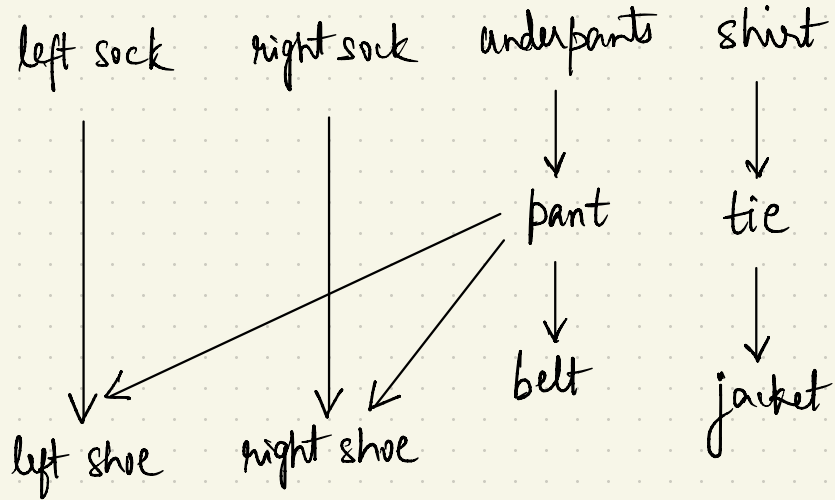
TOPOLOGICAL ORDERING



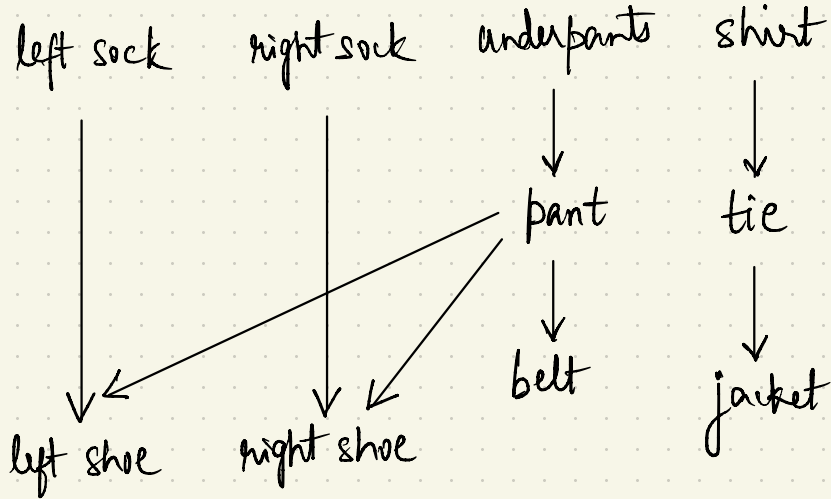
TOPOLOGICAL ORDERING



TOPOLOGICAL ORDERING



TOPOLOGICAL ORDERING



underpants

shirt

pant

belt

tie

jacket

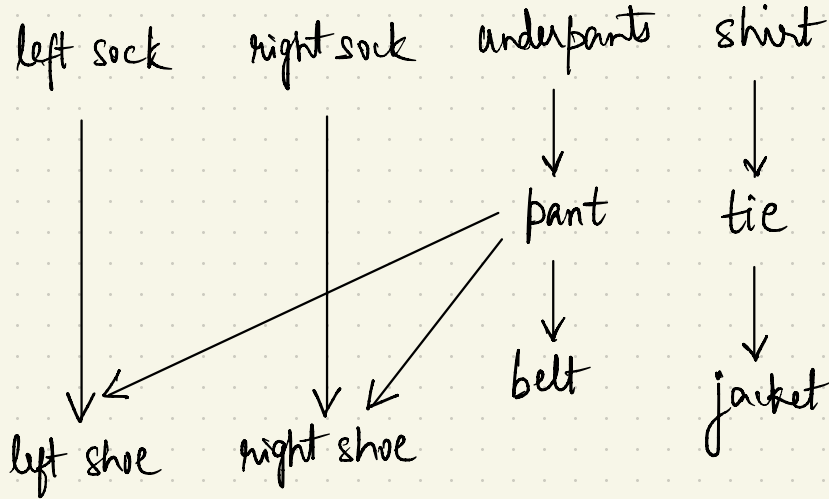
left sock

right sock

left shoe

right shoe

TOPOLOGICAL ORDERING



underpants

shirt

pant

belt

tie

jacket

left sock

right sock

left shoe

right shoe

left sock

shirt

tie

underpants

right sock

pants

right shoe

belt

jacket

left shoe

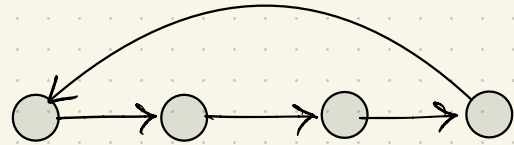
TOPOLOGICAL ORDERING

When does a directed graph have a topological ordering?

TOPOLOGICAL ORDERING

When does a directed graph have a topological ordering?

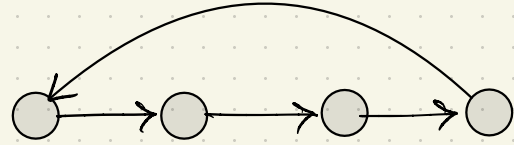
Necessary : No cycle



TOPOLOGICAL ORDERING

When does a directed graph have a topological ordering?

Necessary : No cycle



Also sufficient!

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.

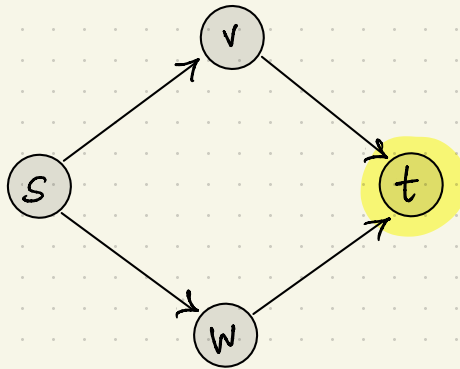
Lemma: Every directed acyclic graph has at least one sink.
a vertex with no outgoing edges

TOPOLOGICAL ORDERING

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

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.

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

a vertex with no outgoing edges

Proof of Lemma: Suppose not. Then, every vertex has some outgoing edge

"Following your nose" will give a cycle. Contradiction!



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

Lemma: Every directed acyclic graph has at least one source.
a vertex with no incoming edges

TOPOLOGICAL ORDERING

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

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

TOPOLOGICAL ORDERING

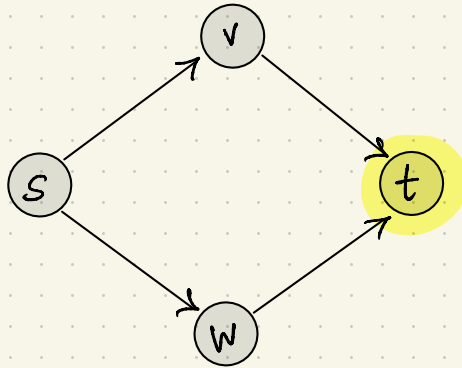
Theorem: Every directed acyclic graph has at least one topological ordering.

Proof: Assign $f[v] = n$ to sink vertex v (exists!).
Recurse on $G \setminus \{v\} \rightarrow$ must be directed acyclic. $\square$

TOPOLOGICAL ORDERING

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

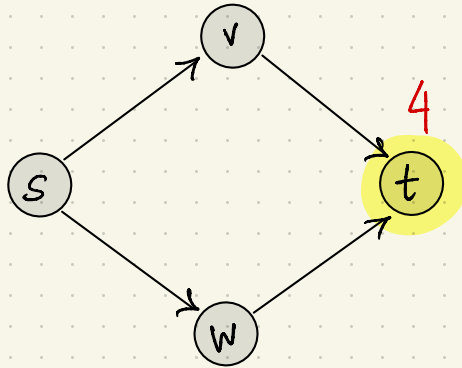
Proof: Assign $f[v] = n$ to sink vertex v (exists!).
Recurse on $G \setminus \{v\} \rightarrow$ must be directed acyclic. $\square$



TOPOLOGICAL ORDERING

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

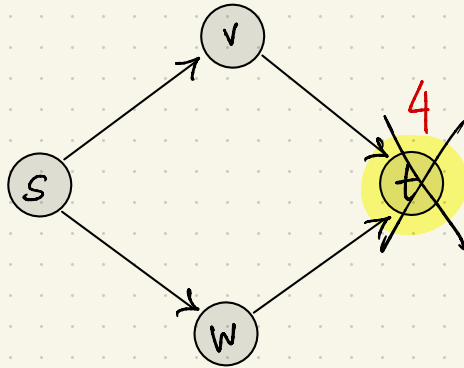
Proof : Assign $f[v] = n$ to sink vertex v (exists!).
Recurse on $G \setminus \{v\} \rightarrow$ must be directed acyclic. $\square$



TOPOLOGICAL ORDERING

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

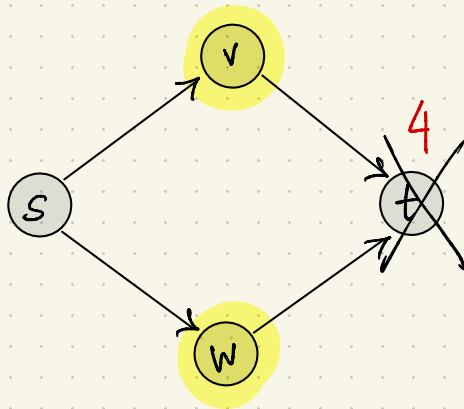
Proof: Assign $f[v] = n$ to sink vertex v (exists!).
Recurse on $G \setminus \{v\} \rightarrow$ must be directed acyclic. $\square$



TOPOLOGICAL ORDERING

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

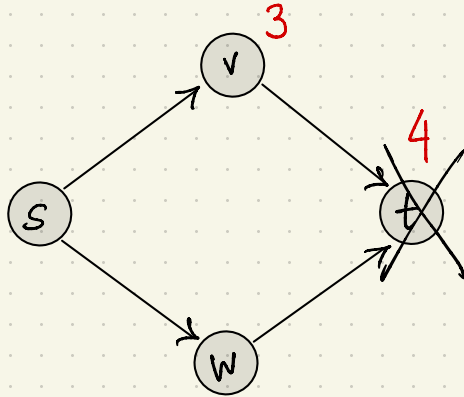
Proof: Assign $f[v] = n$ to sink vertex v (exists!).
Recurse on $G \setminus \{v\} \rightarrow$ must be directed acyclic. $\square$



TOPOLOGICAL ORDERING

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

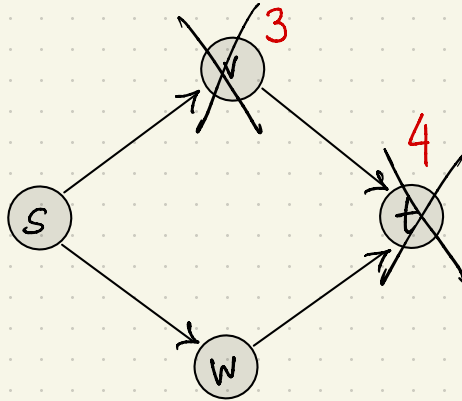
Proof: Assign $f[v] = n$ to sink vertex v (exists!).
Recurse on $G \setminus \{v\} \rightarrow$ must be directed acyclic. $\square$



TOPOLOGICAL ORDERING

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

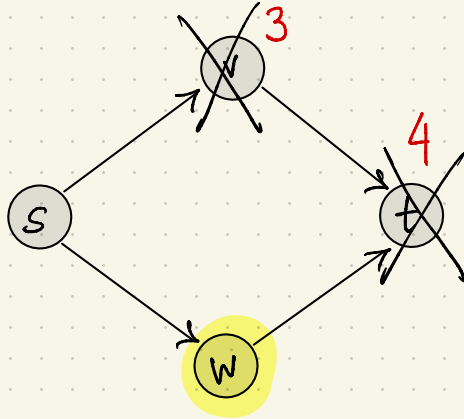
Proof : Assign $f[v] = n$ to sink vertex v (exists!).
Recurse on $G \setminus \{v\} \rightarrow$ must be directed acyclic. $\square$



TOPOLOGICAL ORDERING

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

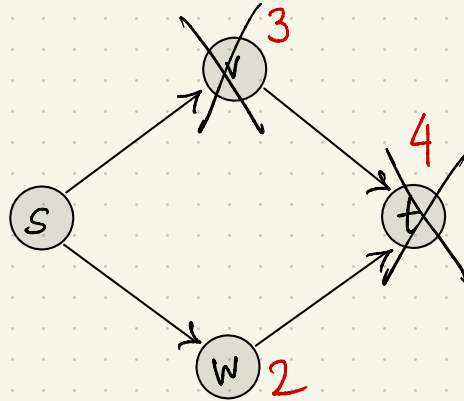
Proof: Assign $f[v] = n$ to sink vertex v (exists!).
Recurse on $G \setminus \{v\} \rightarrow$ must be directed acyclic. $\square$



TOPOLOGICAL ORDERING

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

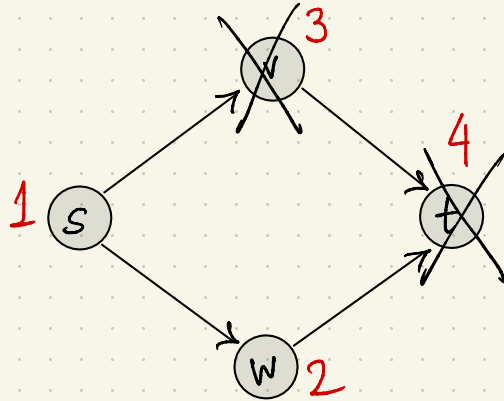
Proof: Assign $f[v] = n$ to sink vertex v (exists!).
Recurse on $G \setminus \{v\} \rightarrow$ must be directed acyclic. $\square$



TOPOLOGICAL ORDERING

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

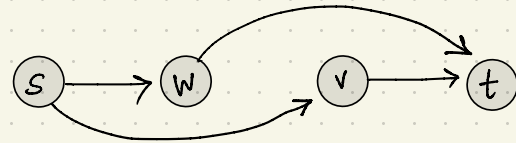
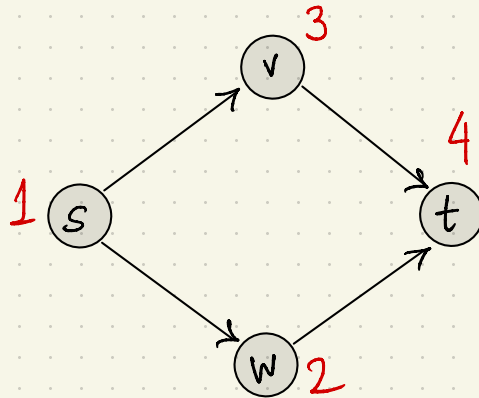
Proof : Assign $f[v] = n$ to sink vertex v (exists!).
Recurse on $G \setminus \{v\} \rightarrow$ must be directed acyclic. $\square$



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 $G \setminus \{v\} \rightarrow$ must be directed acyclic. $\square$



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

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

Running time

$$O(n^2)$$

TOPOLOGICAL ORDERING

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

Algorithm

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

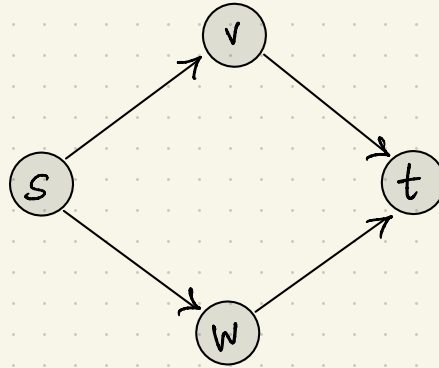
for each (s, v) in adj. list of s

└ if v is **unexplored**
└ └ DFS(G, v)

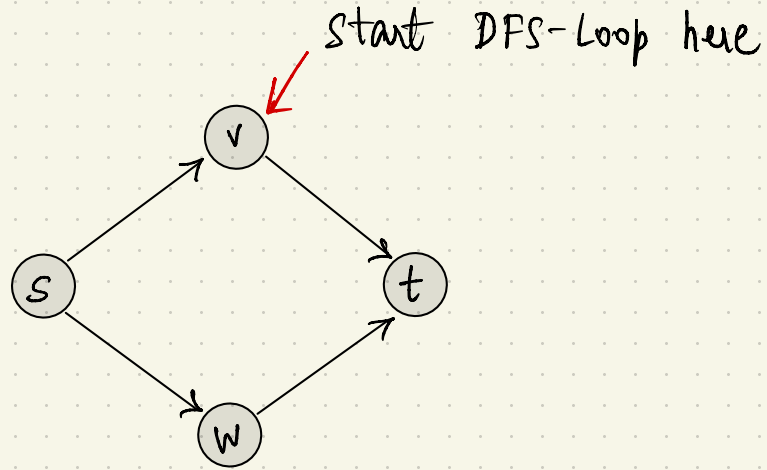
Set $f[s] = \text{current_label}$

decrease current_label by 1

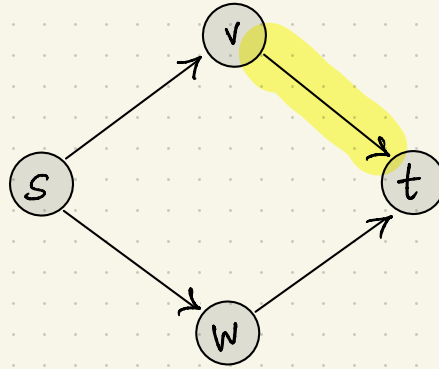
TOPOLOGICAL ORDERING via DFS



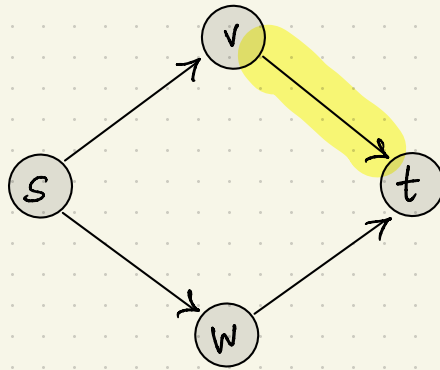
TOPOLOGICAL ORDERING via DFS



TOPOLOGICAL ORDERING via DFS

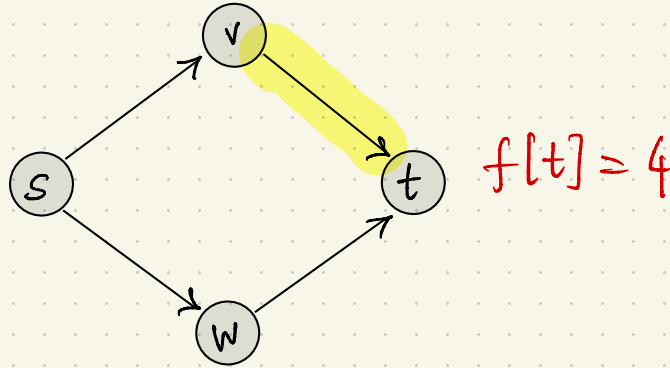


TOPOLOGICAL ORDERING via DFS

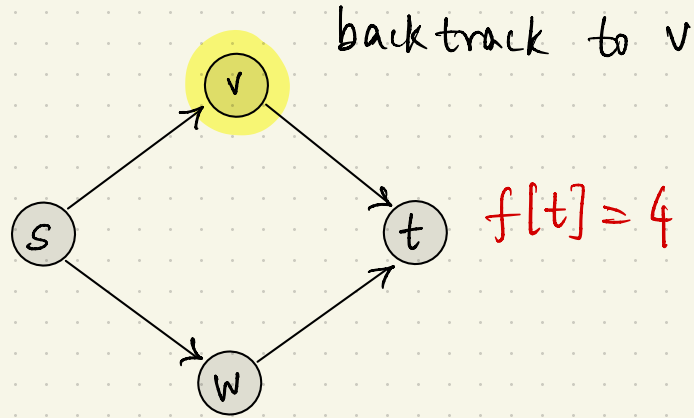


nowhere to go from t

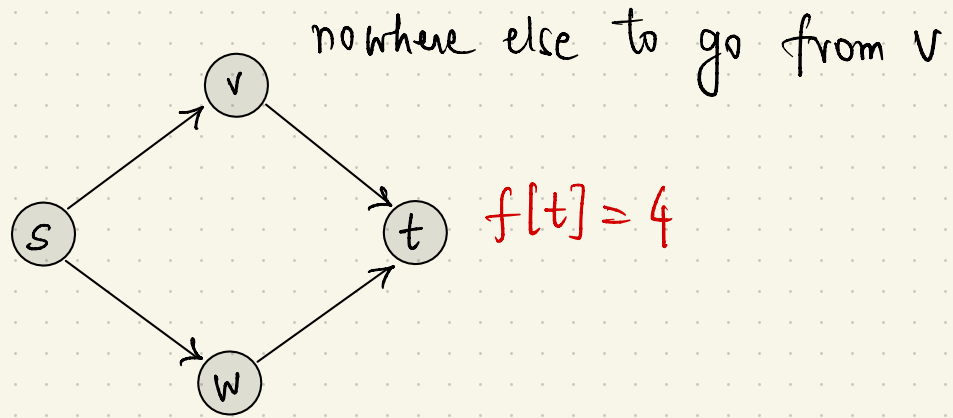
TOPOLOGICAL ORDERING via DFS



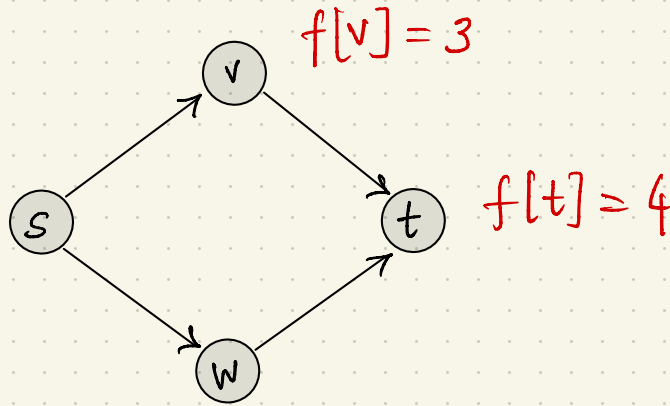
TOPOLOGICAL ORDERING via DFS



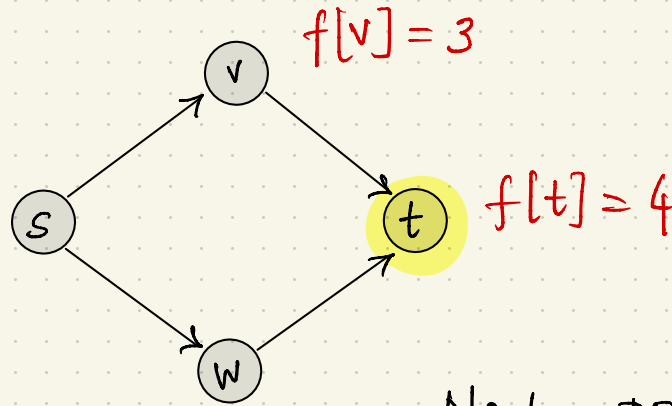
TOPOLOGICAL ORDERING via DFS



TOPOLOGICAL ORDERING via DFS



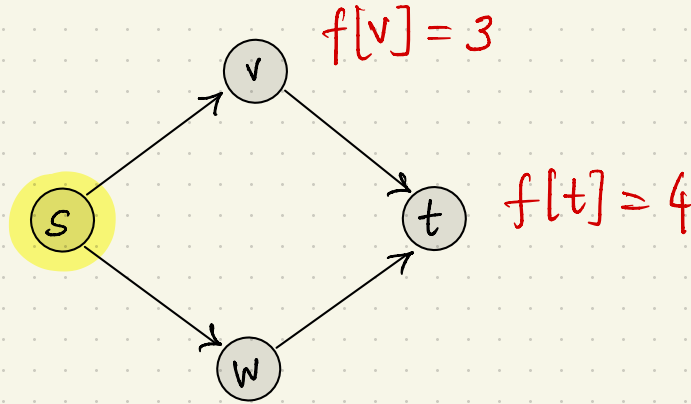
TOPOLOGICAL ORDERING via DFS



Next, DFS-loop considers t
but it is already explored
so skip it.

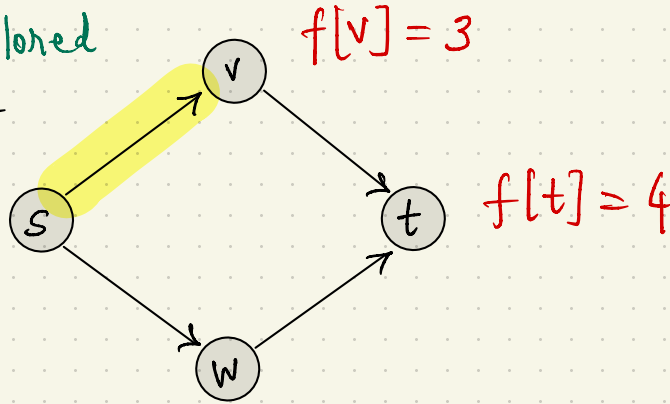
TOPOLOGICAL ORDERING via DFS

Next, DFS-Loop
considers s .

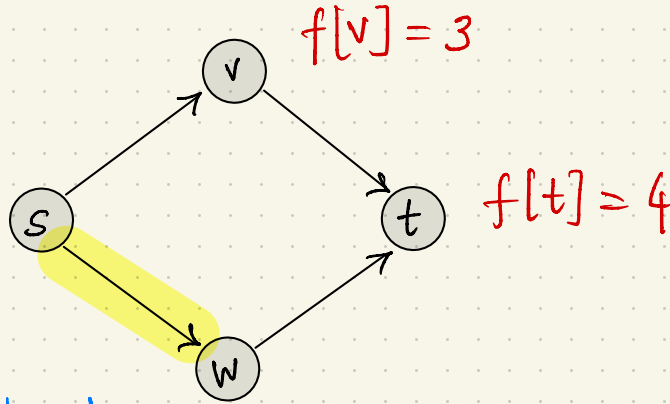


TOPOLOGICAL ORDERING via DFS

v is explored
so skip it

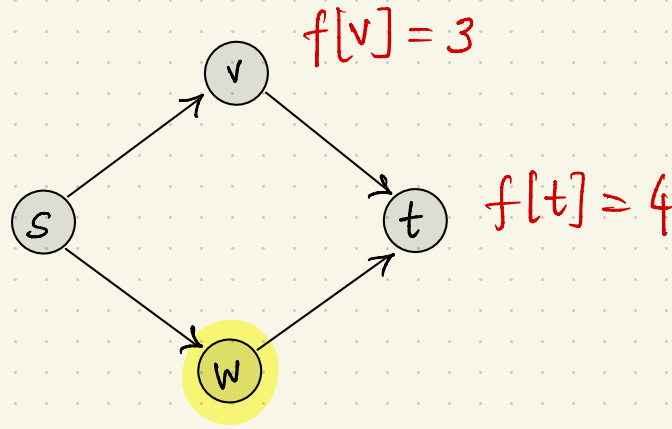


TOPOLOGICAL ORDERING via DFS



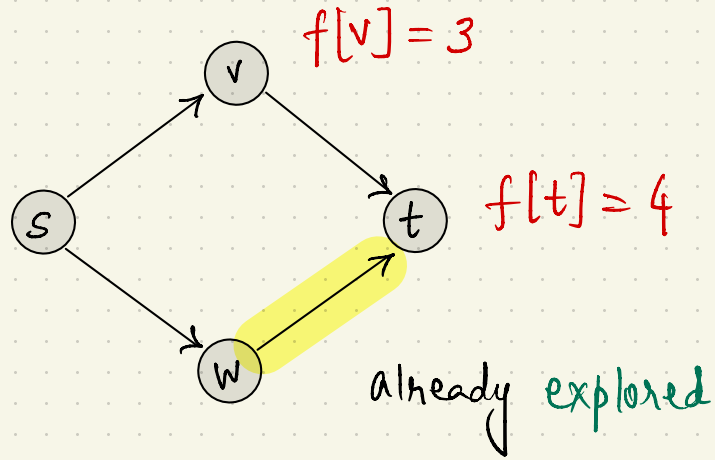
w is unexplored

TOPOLOGICAL ORDERING via DFS

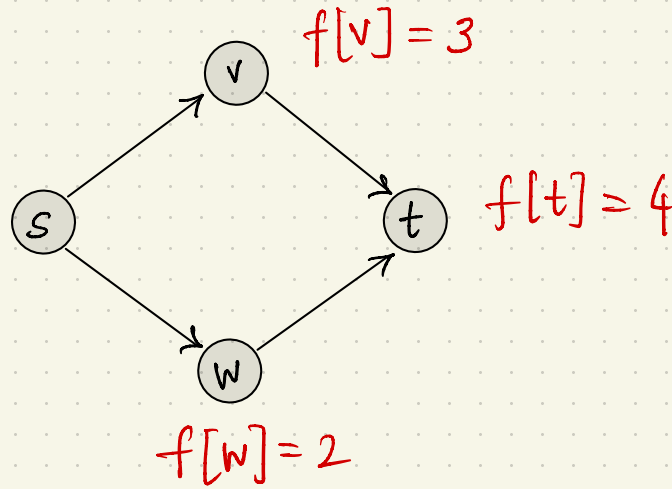


Call DFS on w

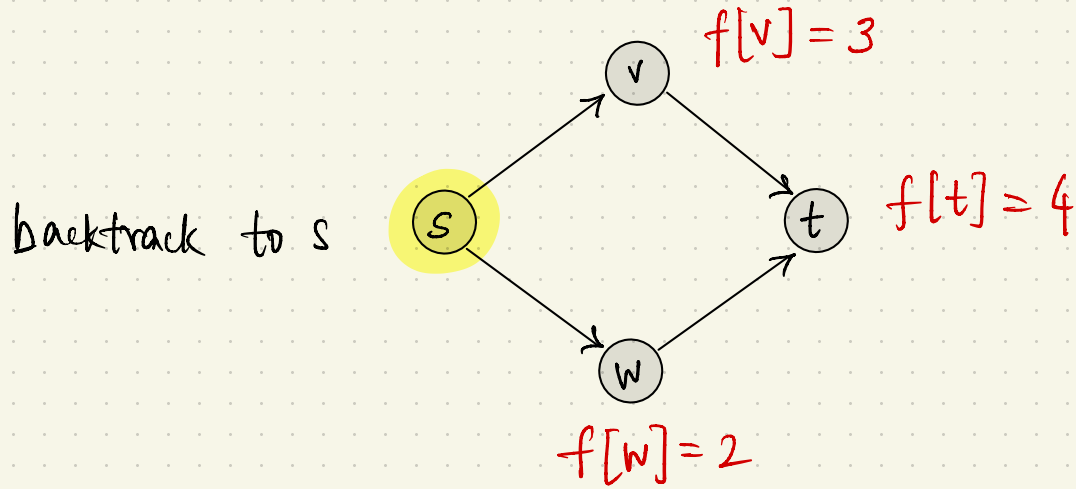
TOPOLOGICAL ORDERING via DFS



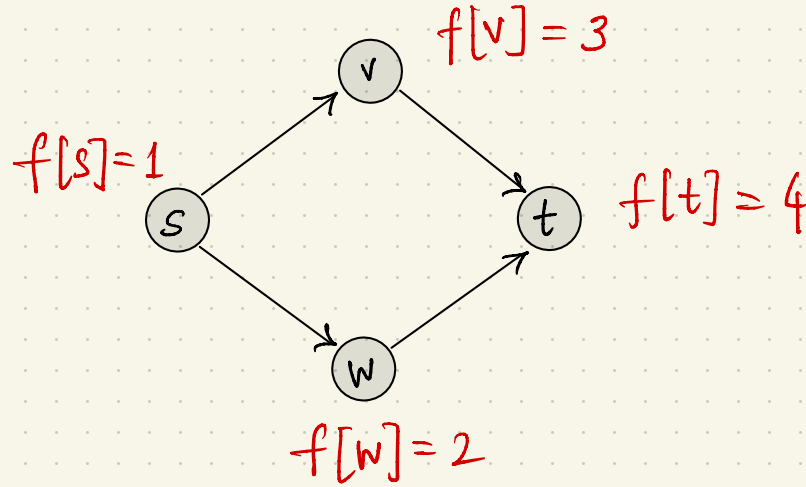
TOPOLOGICAL ORDERING via DFS



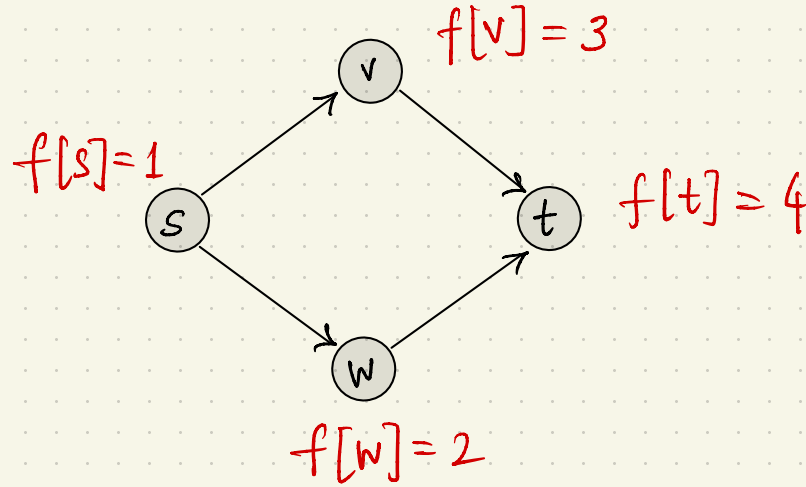
TOPOLOGICAL ORDERING via DFS



TOPOLOGICAL ORDERING via DFS



TOPOLOGICAL ORDERING via DFS



DFS- Loop runs out of vertices $\Rightarrow$ 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.

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 topological ordering

not necessarily
a DAG

APPLICATIONS OF DFS

Topological ordering

Strongly Connected Components

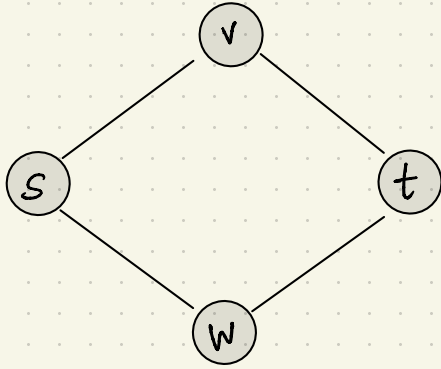
APPLICATIONS OF DFS

Topological ordering

Strongly Connected Components

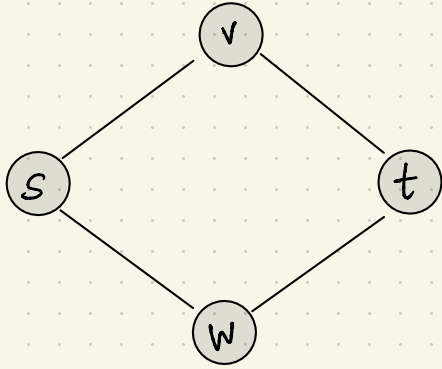
STRONGLY CONNECTED COMPONENTS

STRONGLY CONNECTED COMPONENTS

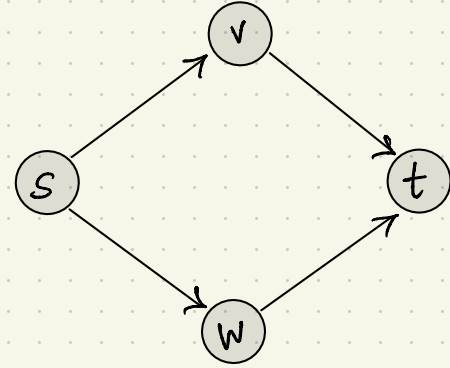


Connected!

STRONGLY CONNECTED COMPONENTS

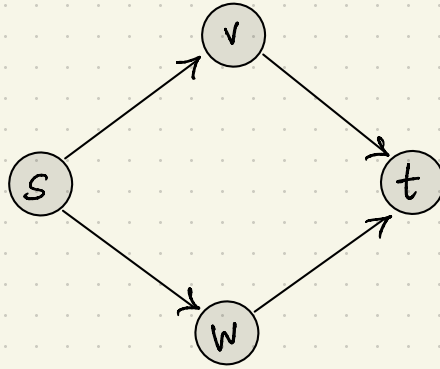


Connected!



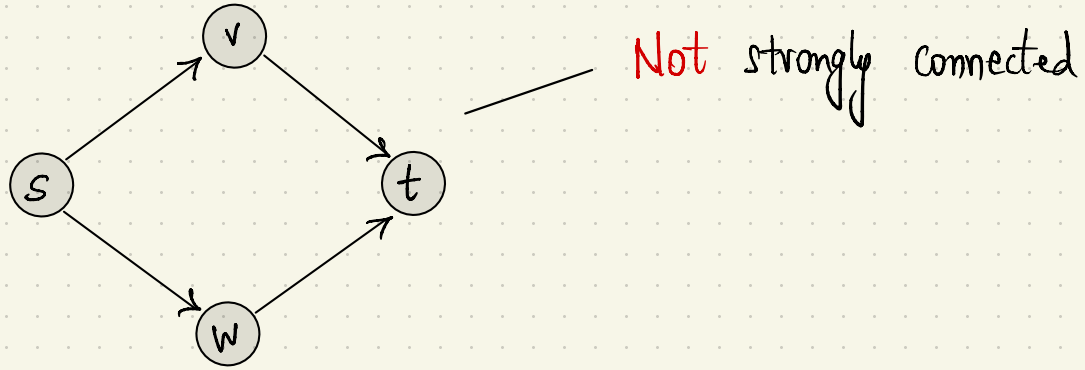
Connected?

STRONGLY CONNECTED COMPONENTS



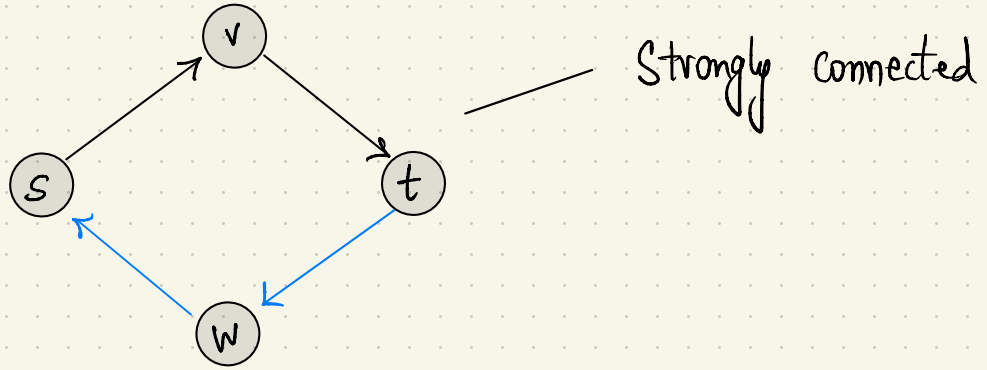
A directed graph is **strongly connected** if every vertex can be reached from every other vertex by a directed path.

STRONGLY CONNECTED COMPONENTS



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

STRONGLY CONNECTED COMPONENTS



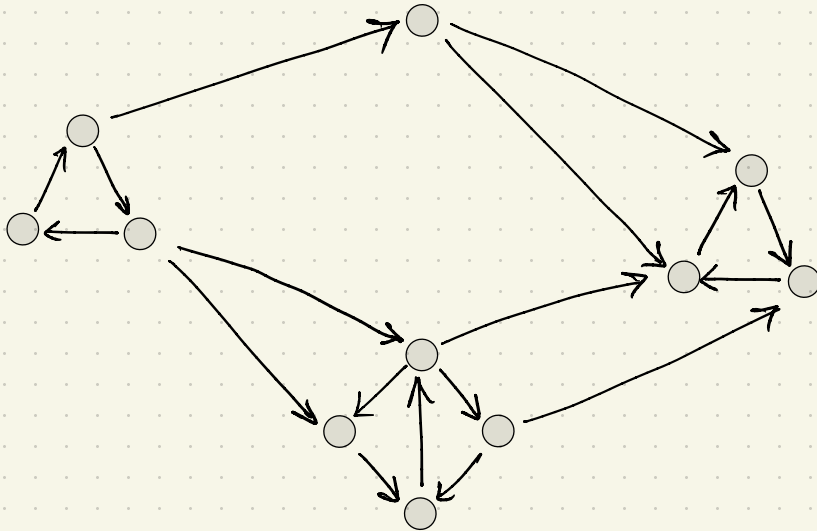
A directed graph is **strongly connected** if every vertex can be reached from every other vertex by a directed path.

STRONGLY CONNECTED COMPONENTS

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

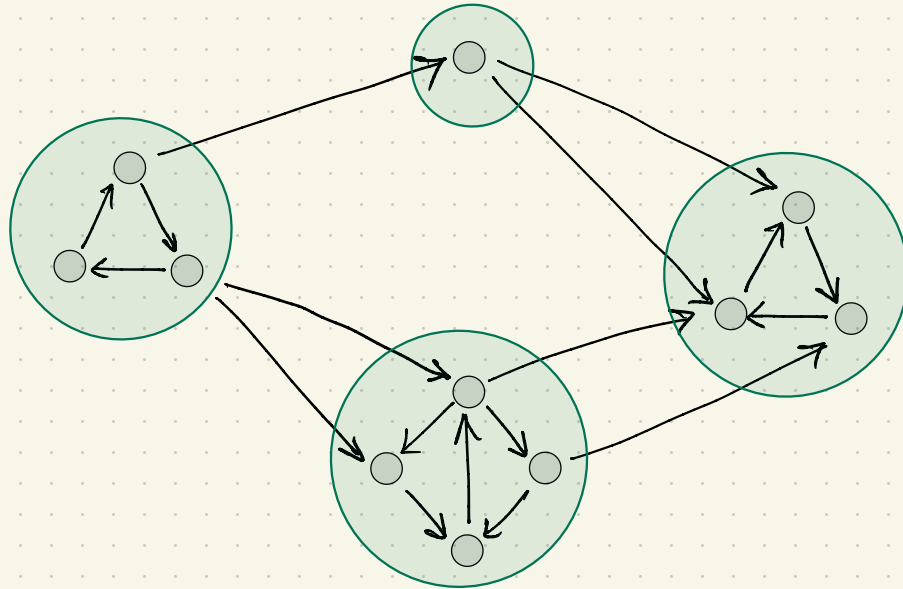
STRONGLY CONNECTED COMPONENTS

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



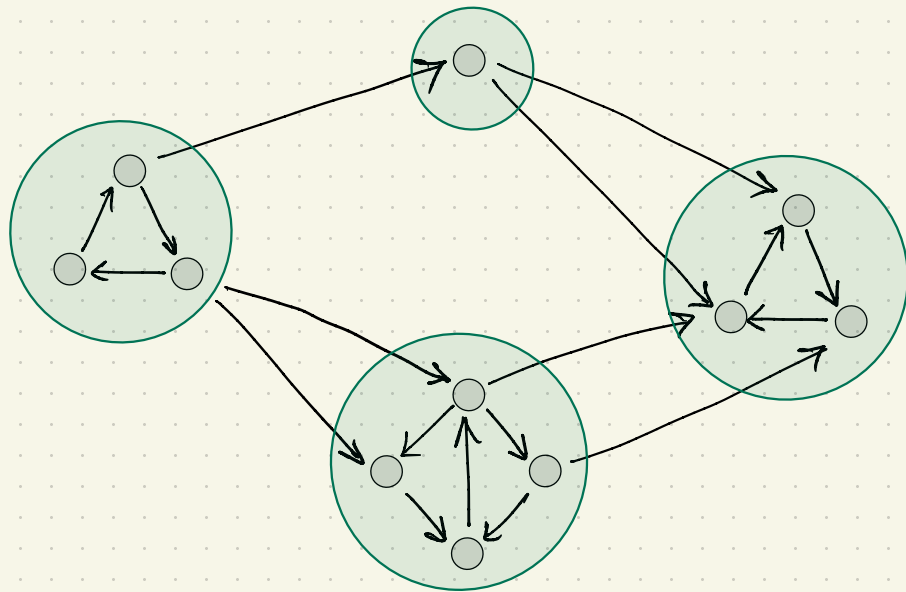
STRONGLY CONNECTED COMPONENTS

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



STRONGLY CONNECTED COMPONENTS

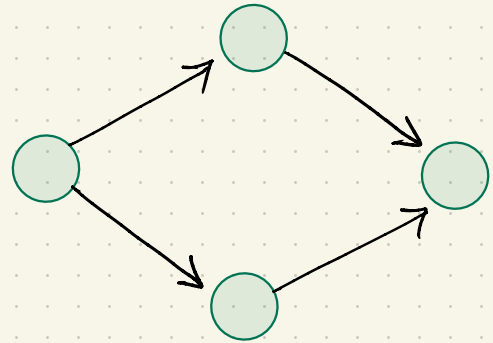
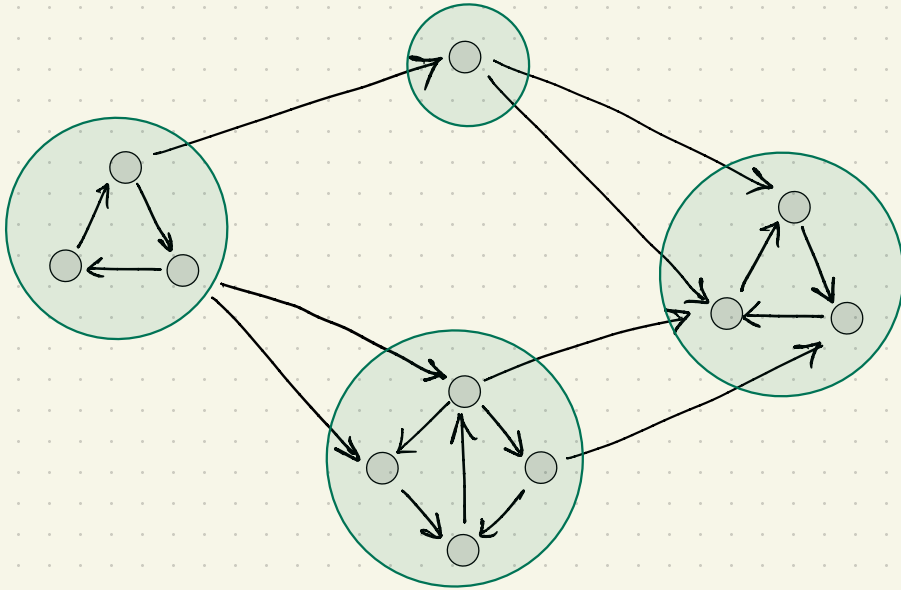
A strongly connected component of a directed graph G is a **maximal** subgraph of G that is strongly connected.



A DAG!

STRONGLY CONNECTED COMPONENTS

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

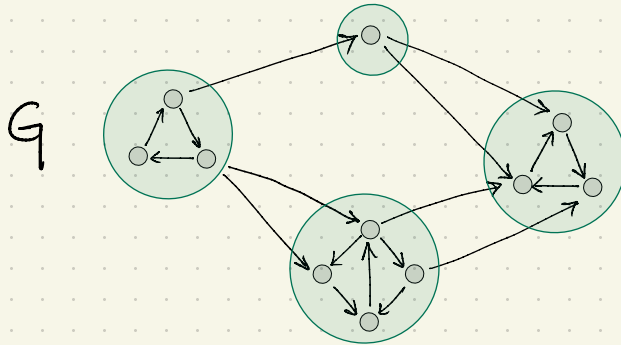


STRONGLY CONNECTED COMPONENTS

Theorem:

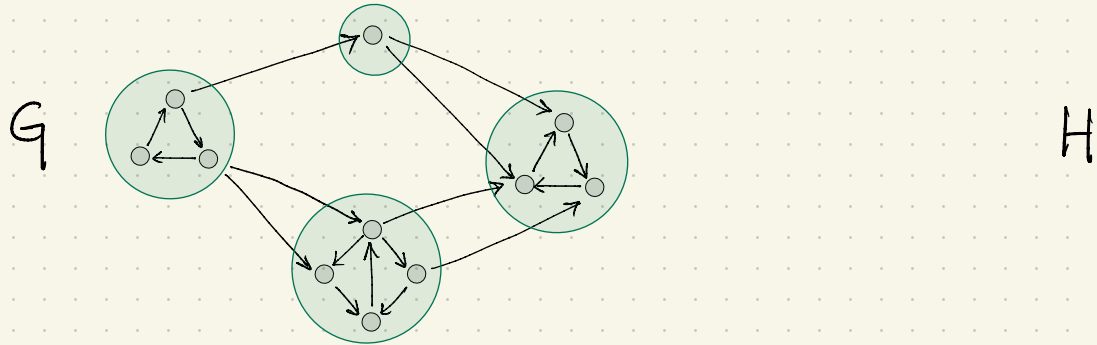
STRONGLY CONNECTED COMPONENTS

Theorem: Given a directed graph G



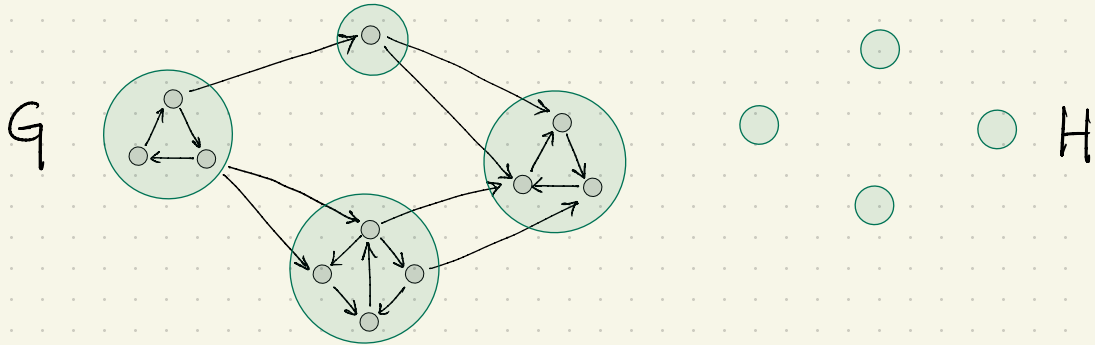
STRONGLY CONNECTED COMPONENTS

Theorem: Given a directed graph G , define a "meta" graph $H=(X,F)$



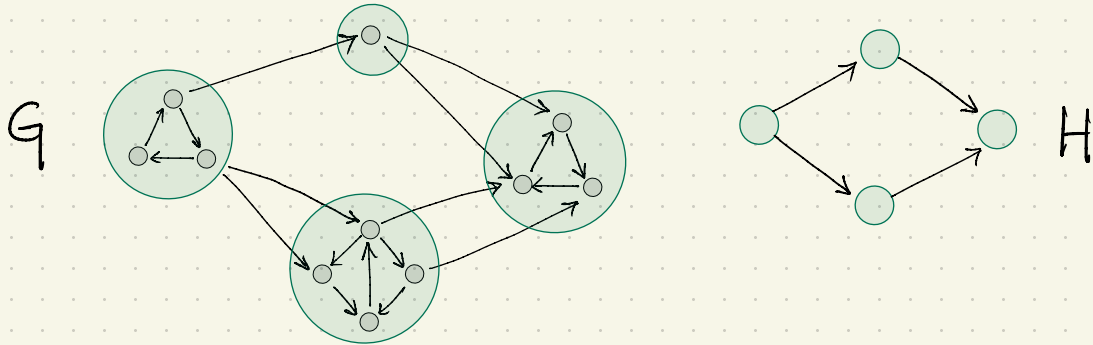
STRONGLY CONNECTED COMPONENTS

Theorem: Given a directed graph G , define a "meta" graph $H=(X,F)$ that has a vertex for every SCC of G



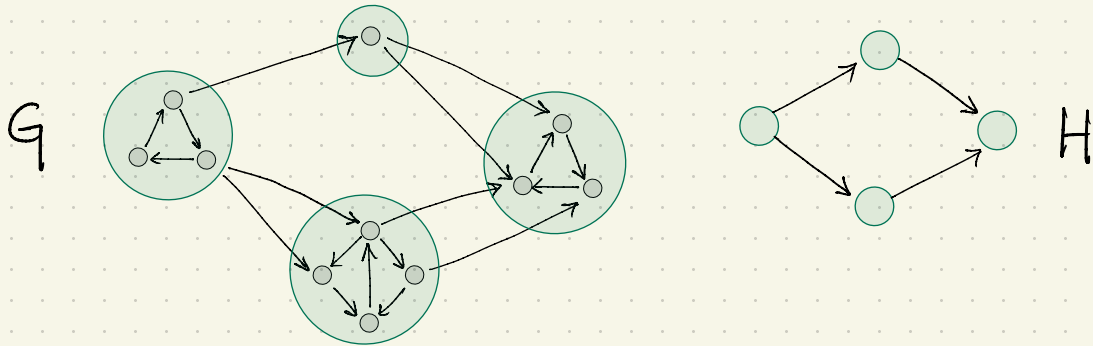
STRONGLY CONNECTED COMPONENTS

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 .



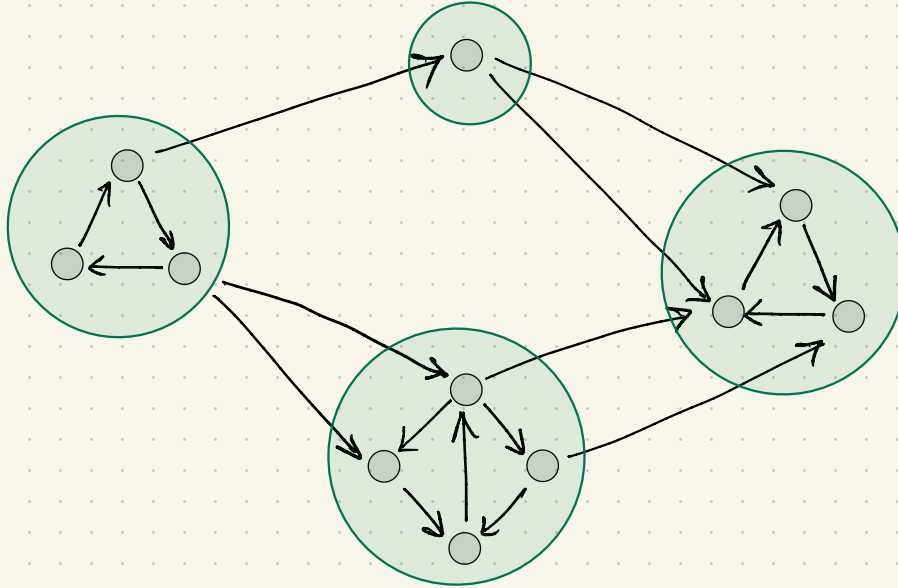
STRONGLY CONNECTED COMPONENTS

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.

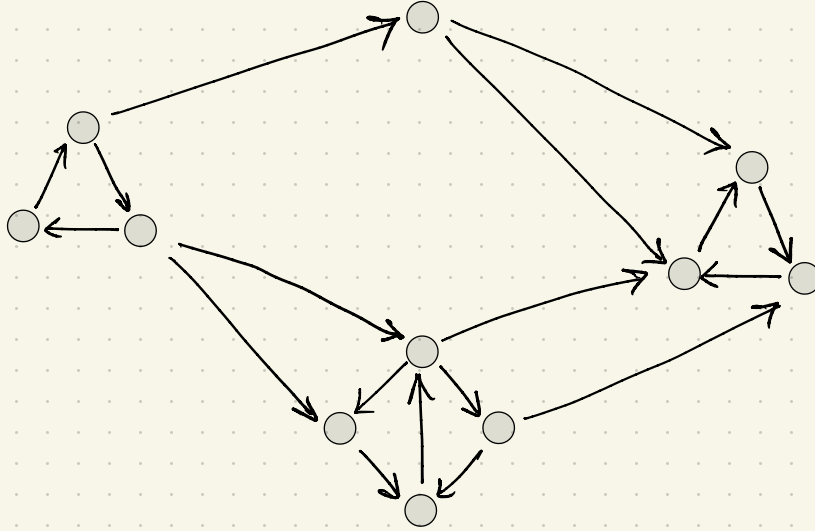


STRONGLY CONNECTED COMPONENTS via DFS

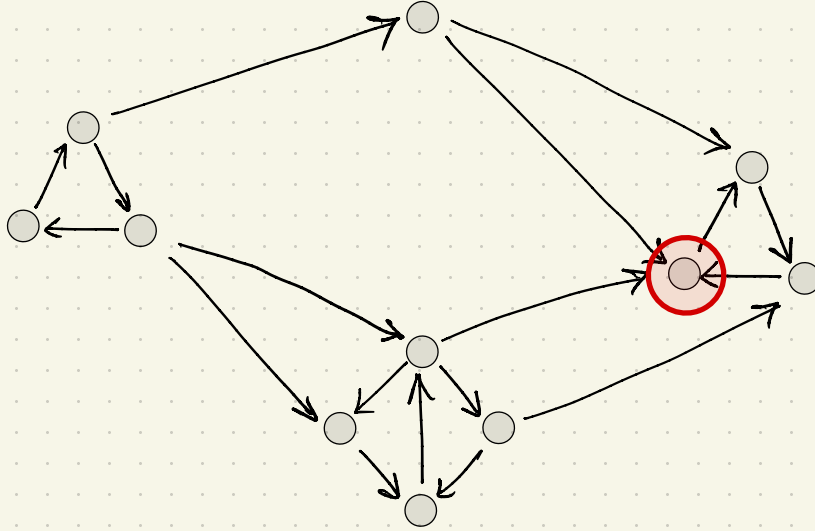
STRONGLY CONNECTED COMPONENTS via DFS



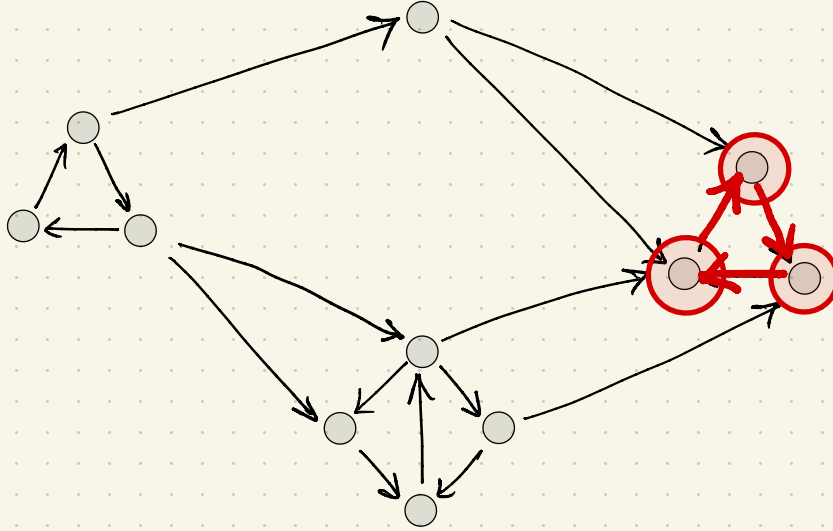
STRONGLY CONNECTED COMPONENTS via DFS



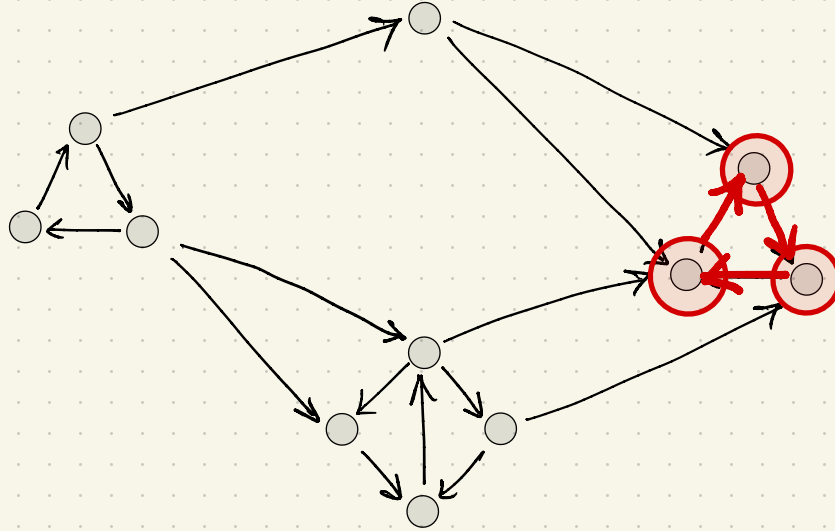
STRONGLY CONNECTED COMPONENTS via DFS



STRONGLY CONNECTED COMPONENTS via DFS

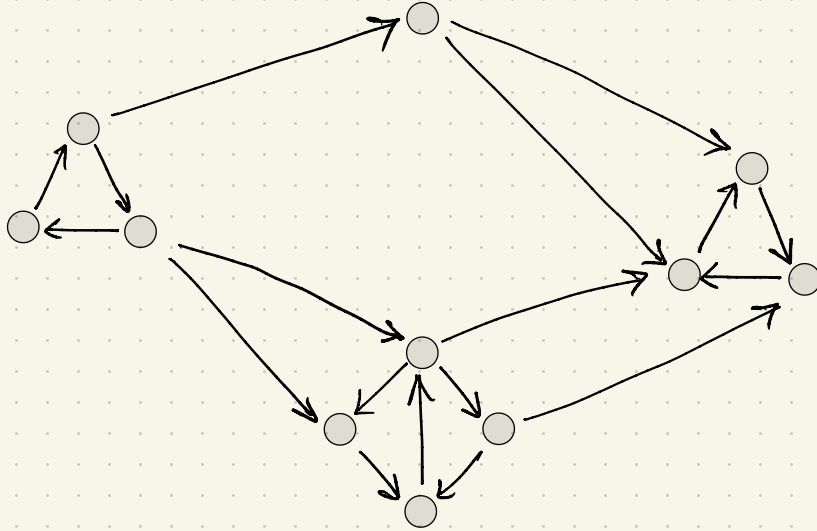


STRONGLY CONNECTED COMPONENTS via DFS



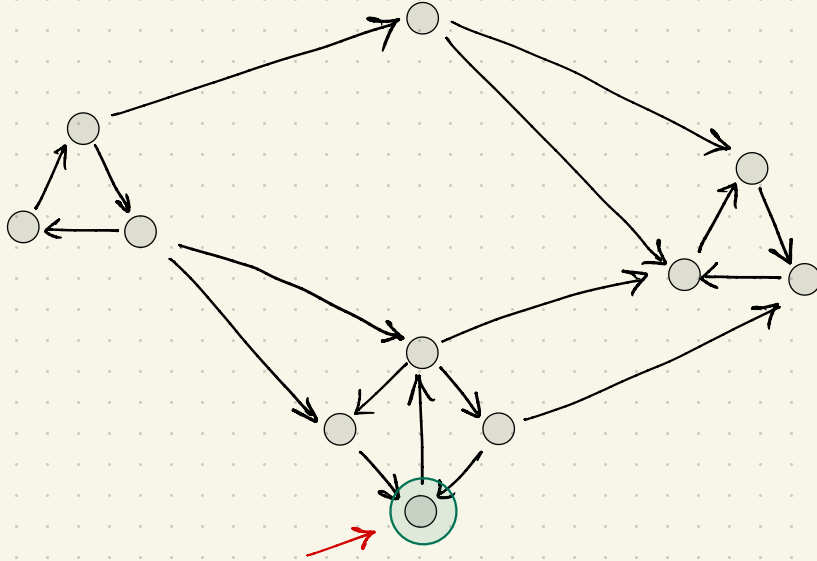
an SCC!

STRONGLY CONNECTED COMPONENTS via DFS



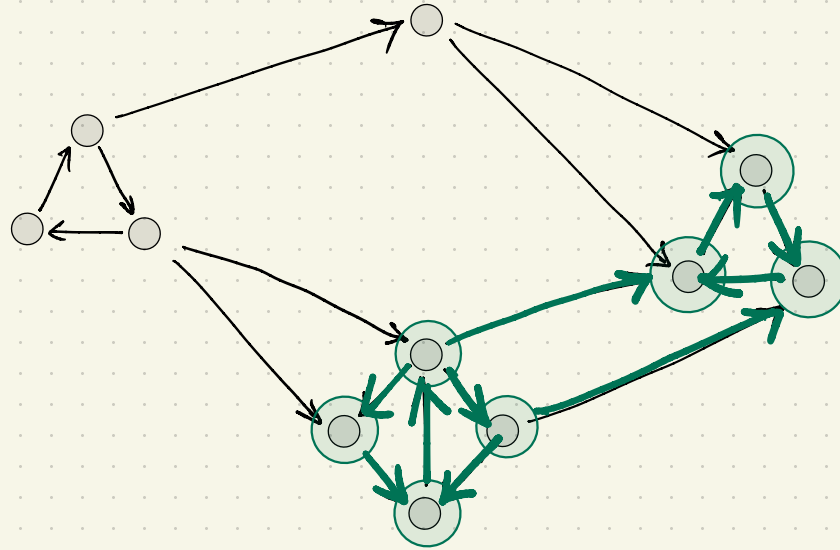
🤔 Maybe do a DFS from every node to identify all SCCs?

STRONGLY CONNECTED COMPONENTS via DFS



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

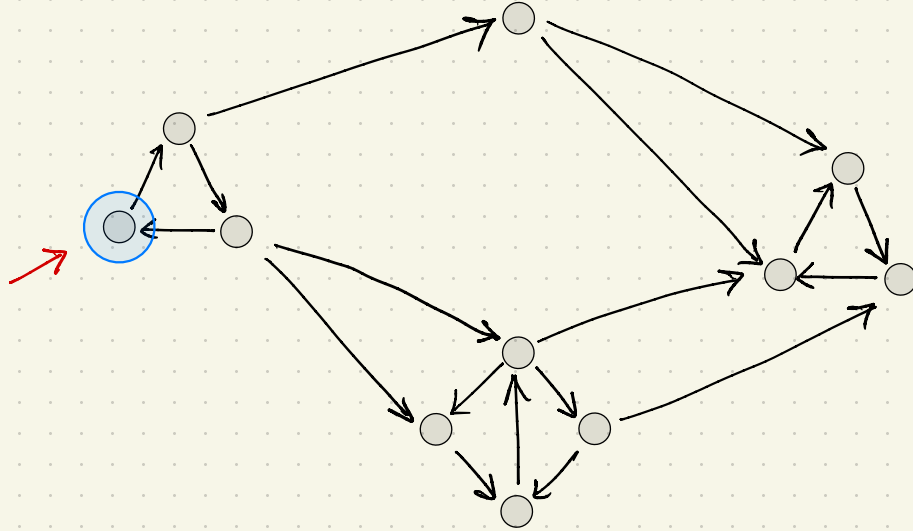
STRONGLY CONNECTED COMPONENTS via DFS



union of SCCs

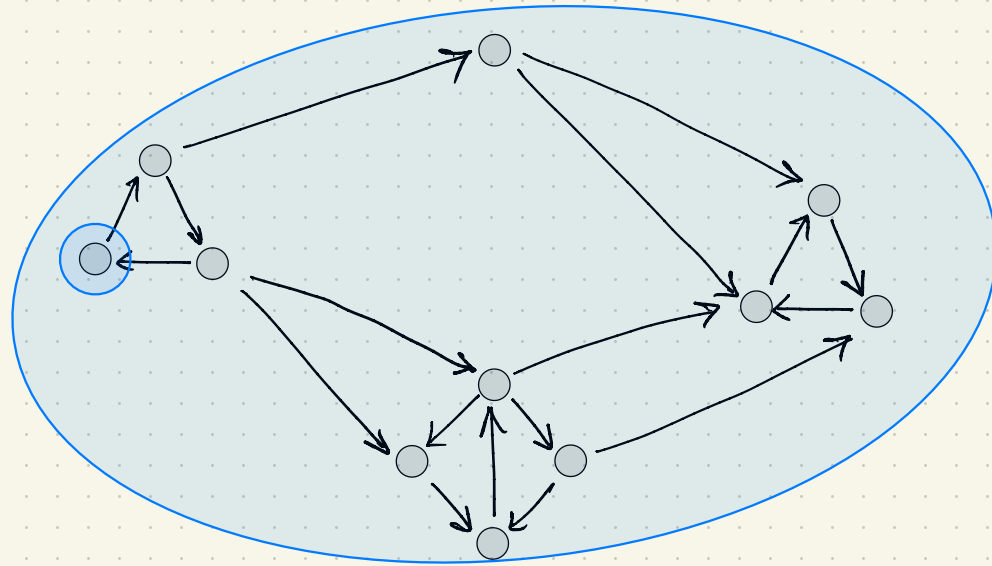
🤔 Maybe do a DFS from every node to identify all SCCs?

STRONGLY CONNECTED COMPONENTS via DFS



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

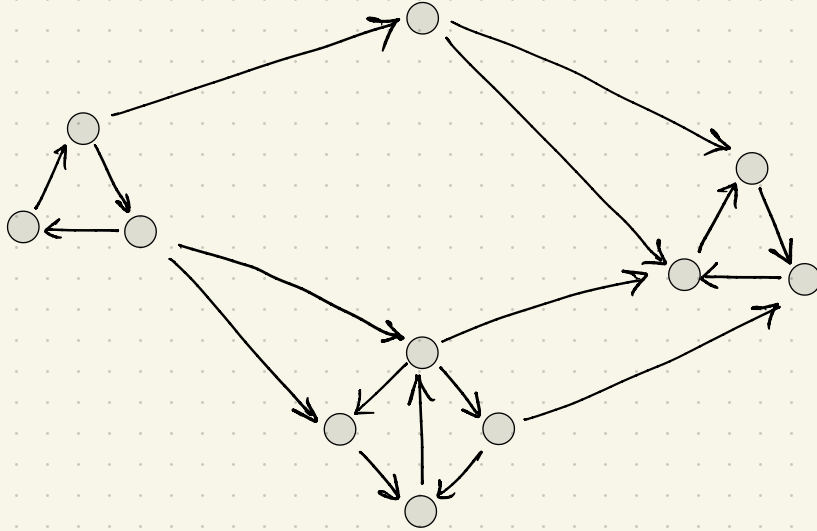
STRONGLY CONNECTED COMPONENTS via DFS



No information
at all!

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

STRONGLY CONNECTED COMPONENTS via DFS

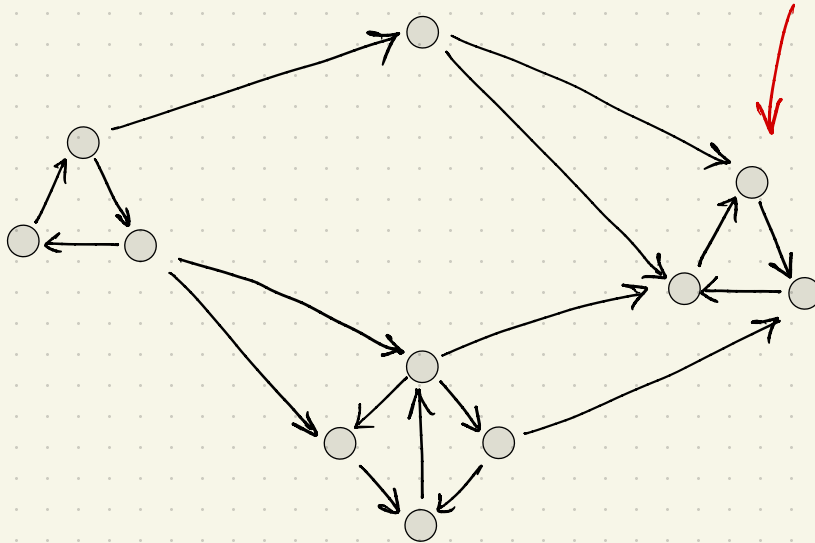


Starting point matters!

FIRST ATTEMPT

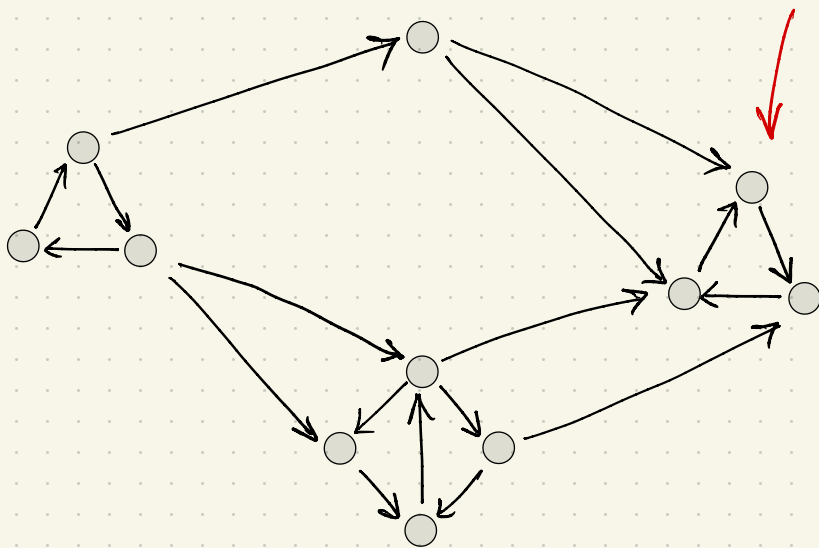
FIRST ATTEMPT

Recall: Starting here worked!



FIRST ATTEMPT

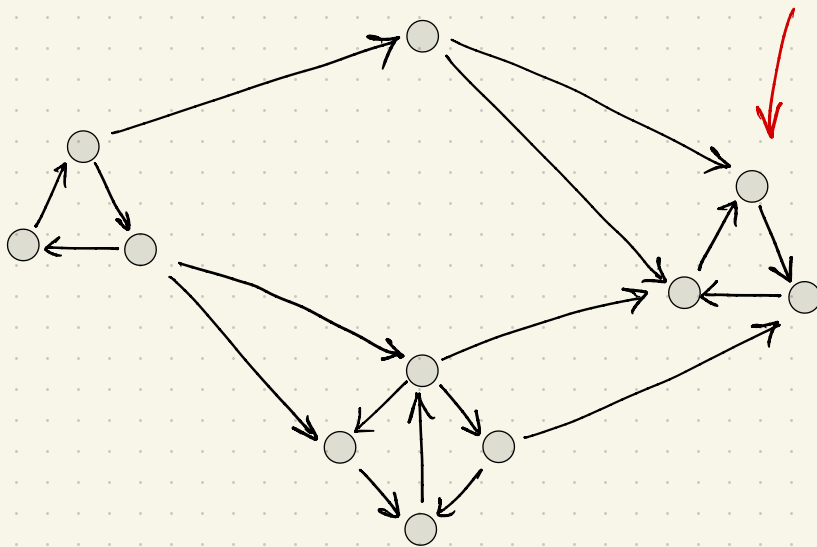
Recall: Starting here worked!



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

FIRST ATTEMPT

Recall: Starting here worked!



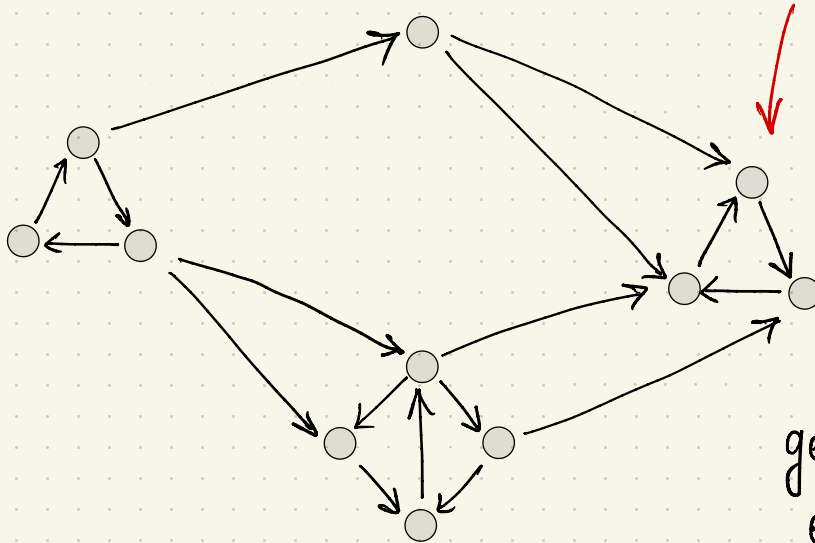
maybe it lies in the "sink" SCC?



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

FIRST ATTEMPT

Recall: Starting here worked!

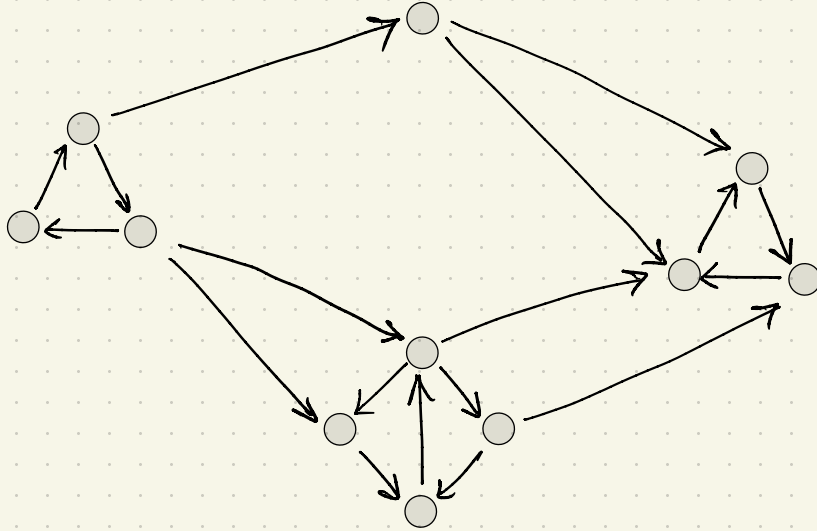


generates **some** ordering
even for cyclic graphs



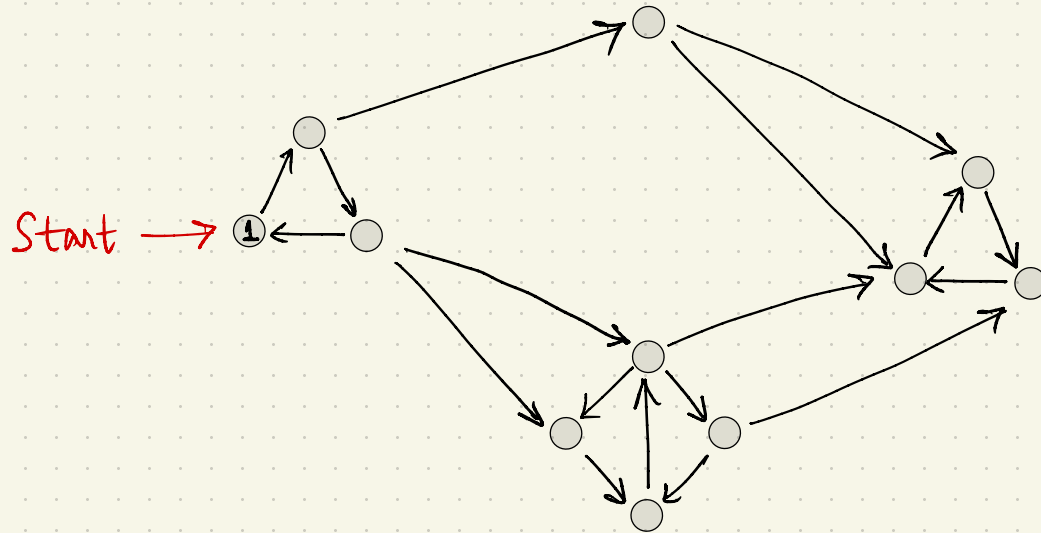
Start DFS at the "last" vertex
(as per **topological ordering algo**)

FIRST ATTEMPT

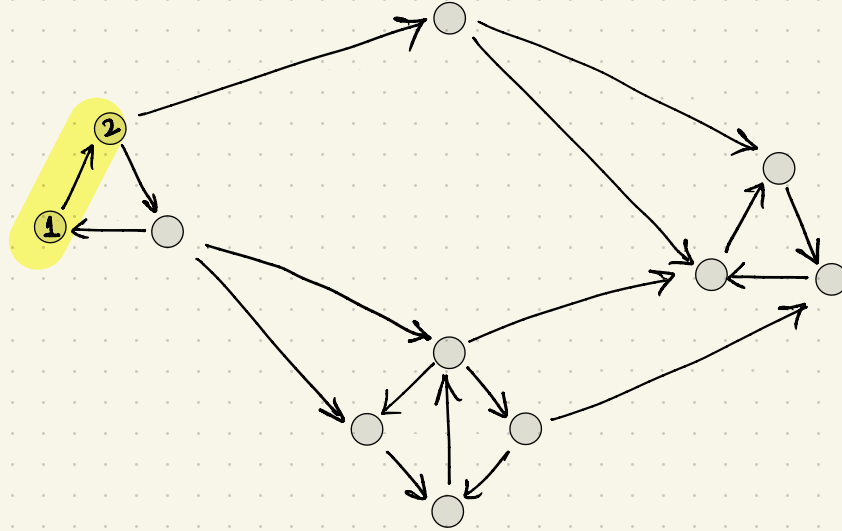


Let's run topological ordering (via DFS) and see what happens.

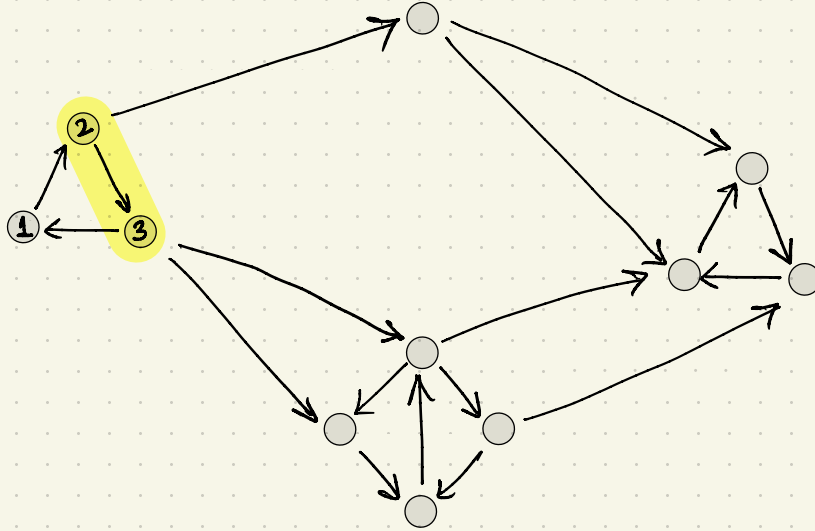
FIRST ATTEMPT



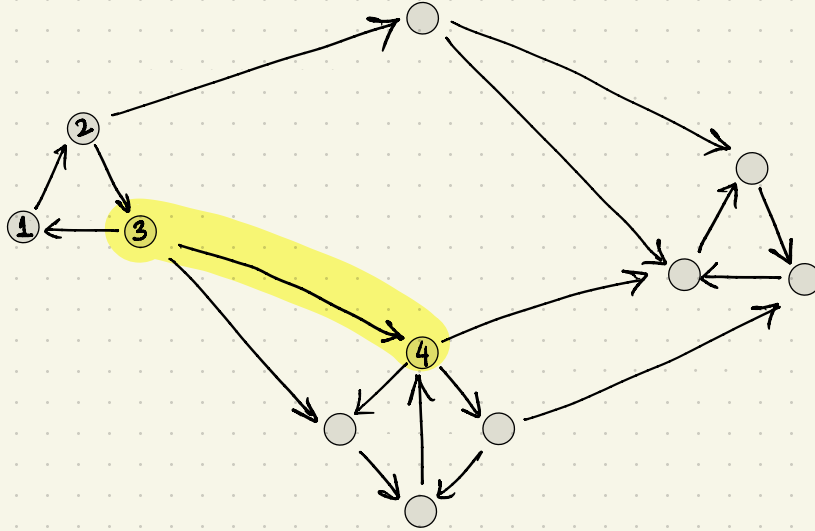
FIRST ATTEMPT



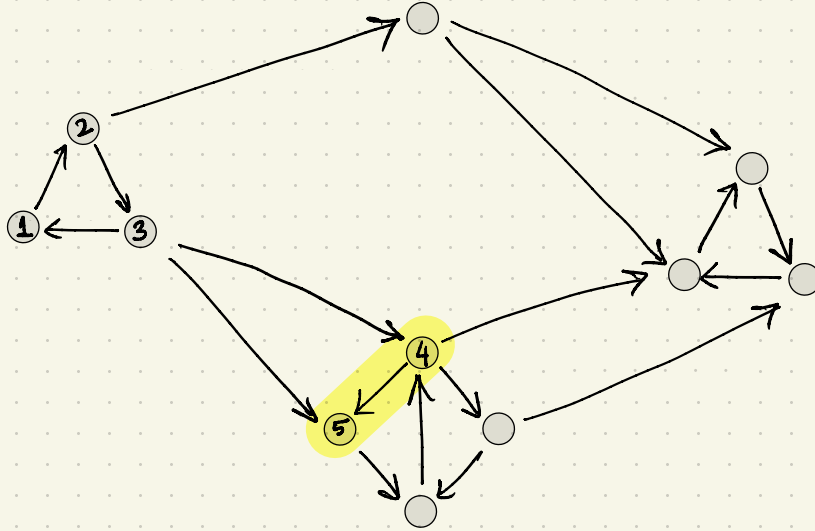
FIRST ATTEMPT



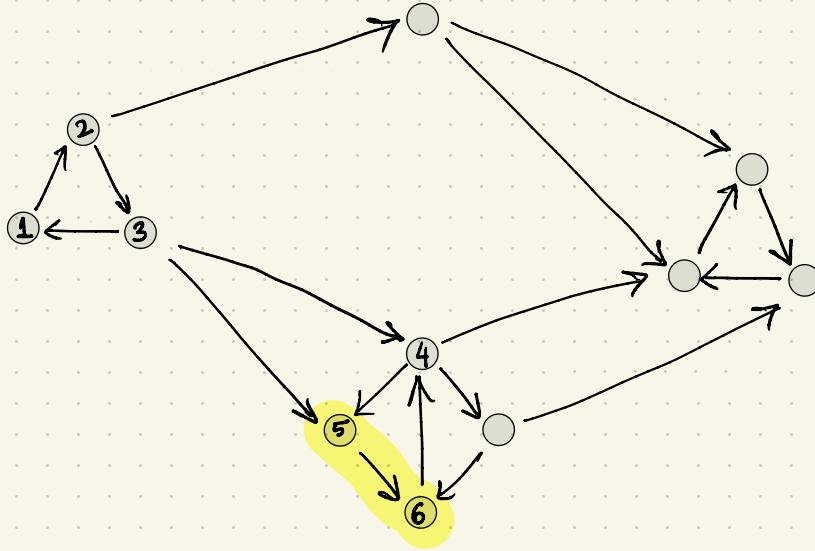
FIRST ATTEMPT



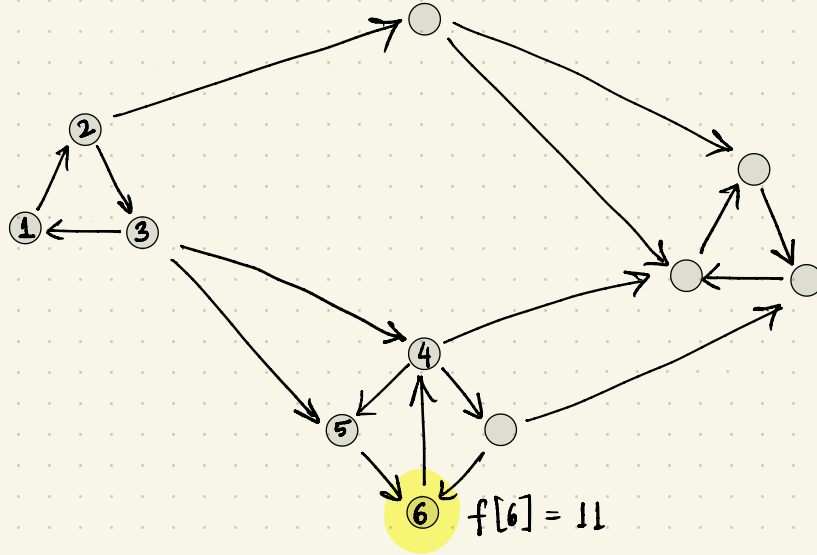
FIRST ATTEMPT



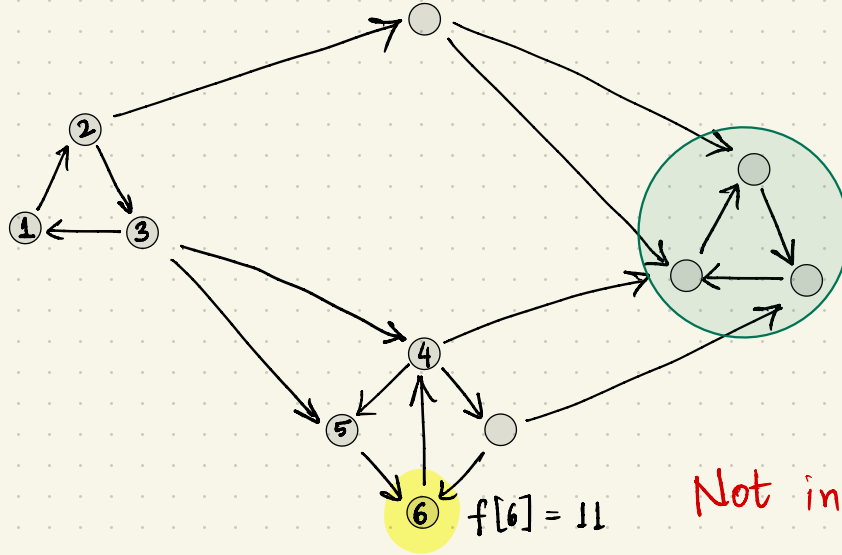
FIRST ATTEMPT



FIRST ATTEMPT

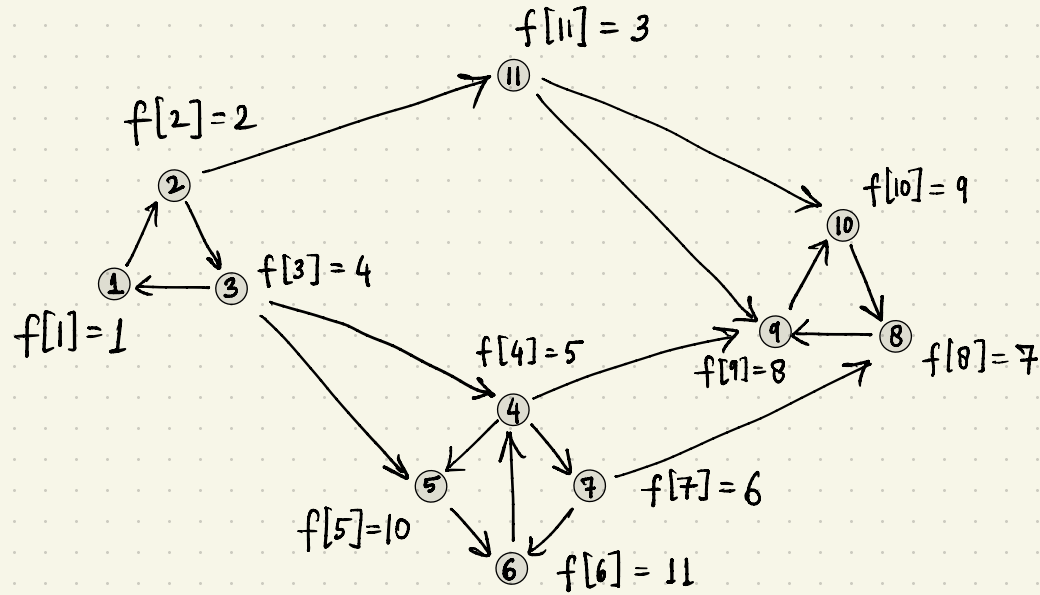


FIRST ATTEMPT

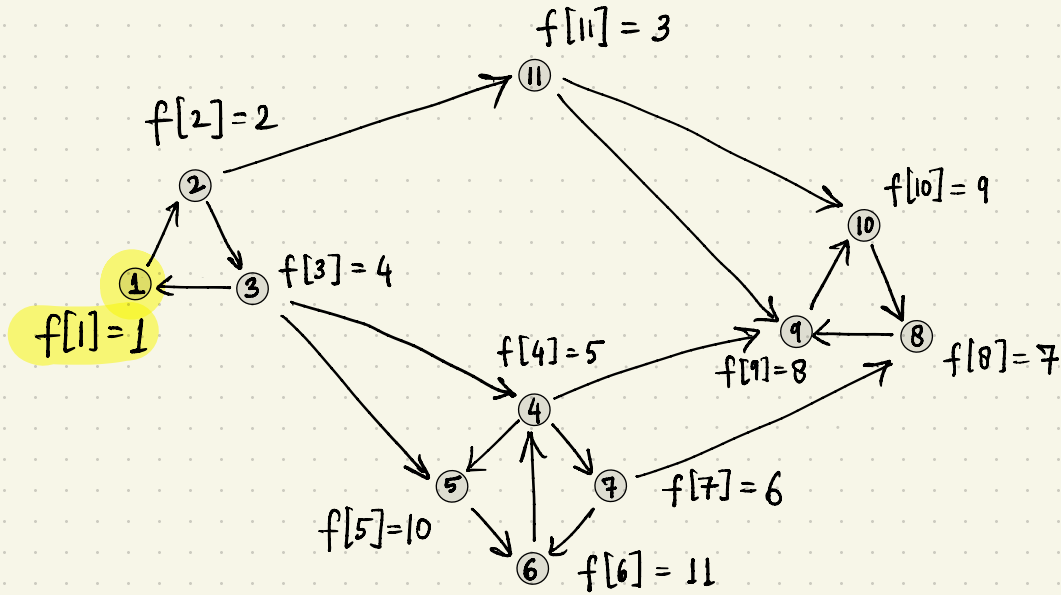


Not in "sink" SCC 😞

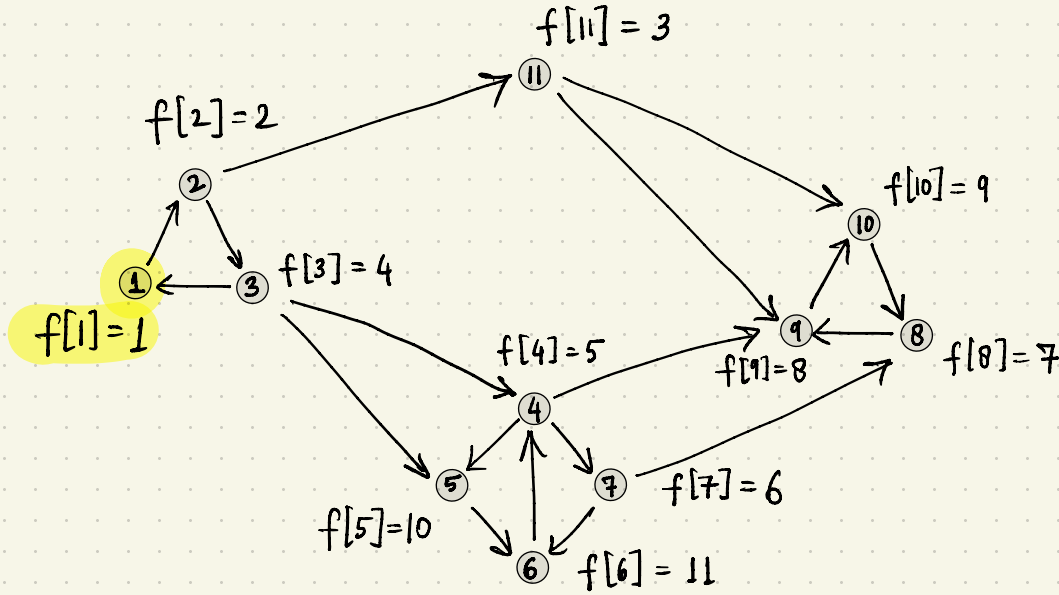
FIRST ATTEMPT



FIRST ATTEMPT

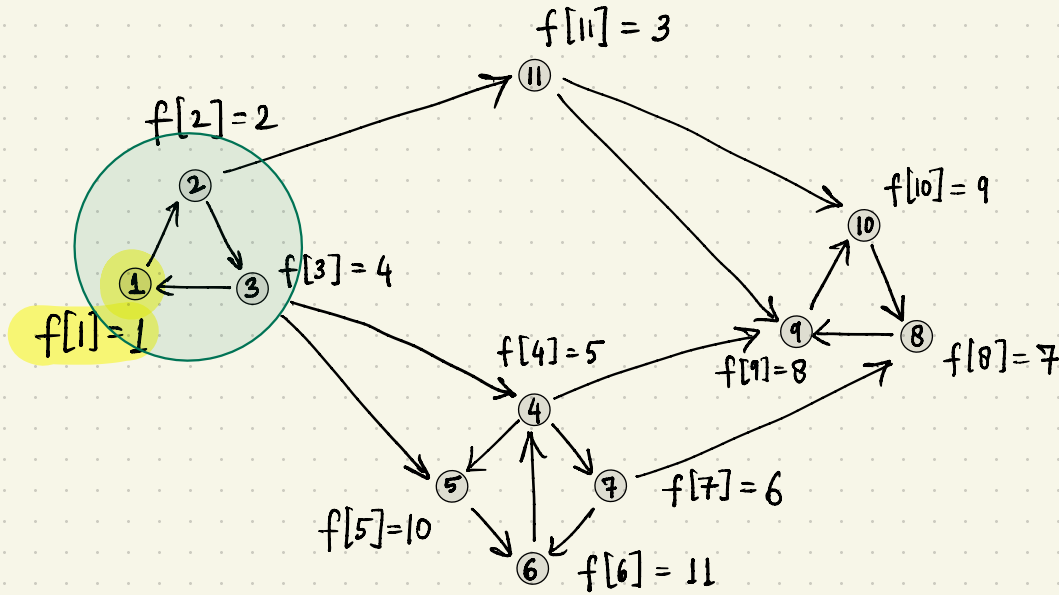


FIRST ATTEMPT



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)