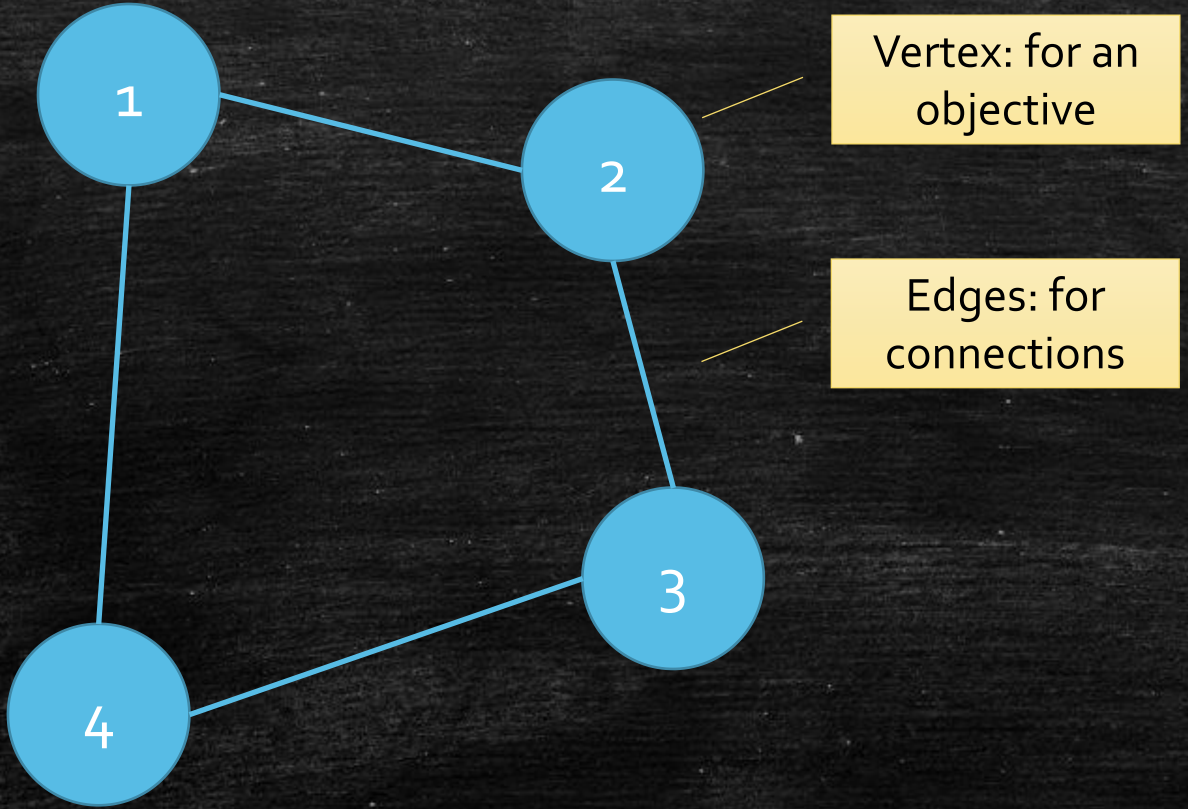
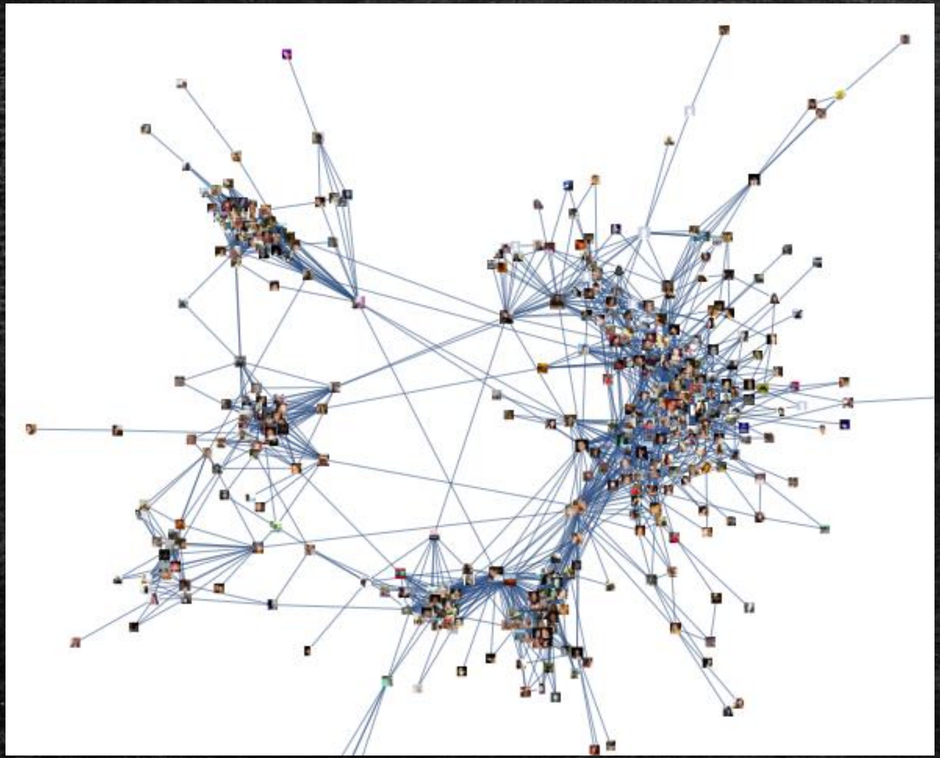
Basic Graph Algorithms

Depth First Search and Its Applications

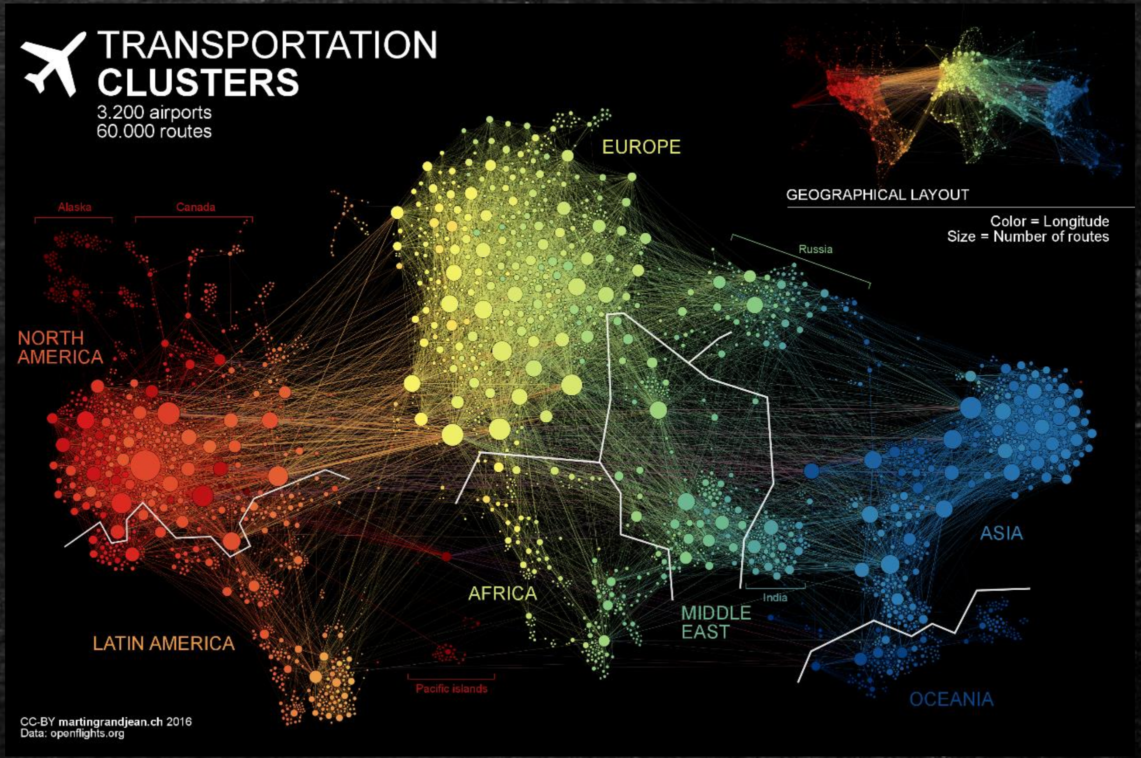
What is graphs?



Large Graphs in Real World

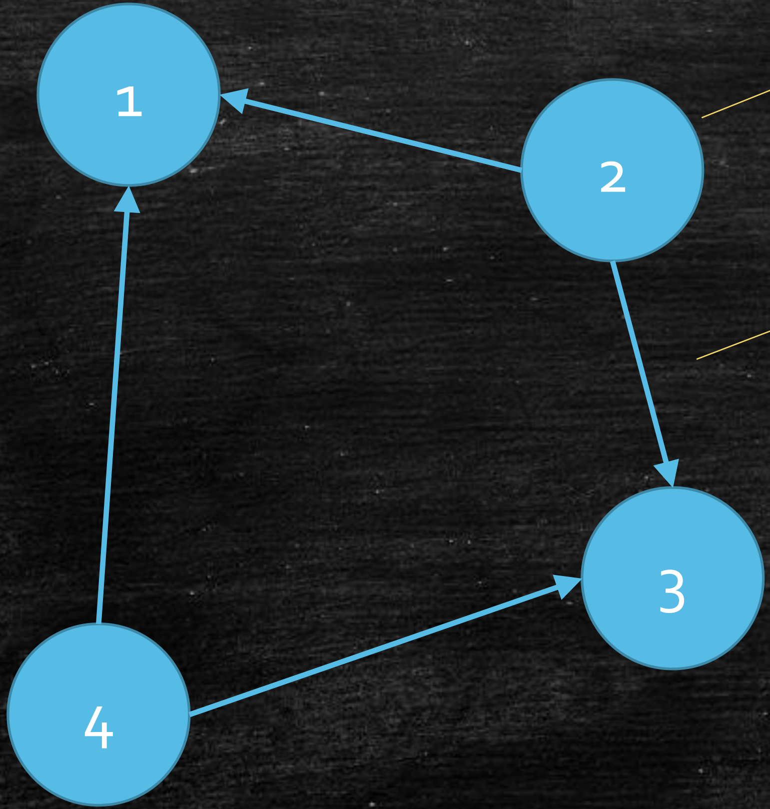


Facebook friends



Airlines

We can have directions!



Vertex: for an objective

Arcs: for directed connections

Discussions

- In a directed graph
  - Arc (u, v) means we can only go from u to v.
- In an undirected graph
  - Edge (u, v) means we can go from u to v or go from v to u.
- Undirected graph & directed graph
  - Undirected graph is a **SPECIAL** directed graph
  - edge $(u, v) \rightarrow$ arc (u, v) & (v, u)
- How many arcs at most in an undirected graph?
  - $G(V, E)$
  - $0 \le |E| \le |V|(|V| - 1) = O(|V|^2)$

How to store a graph?

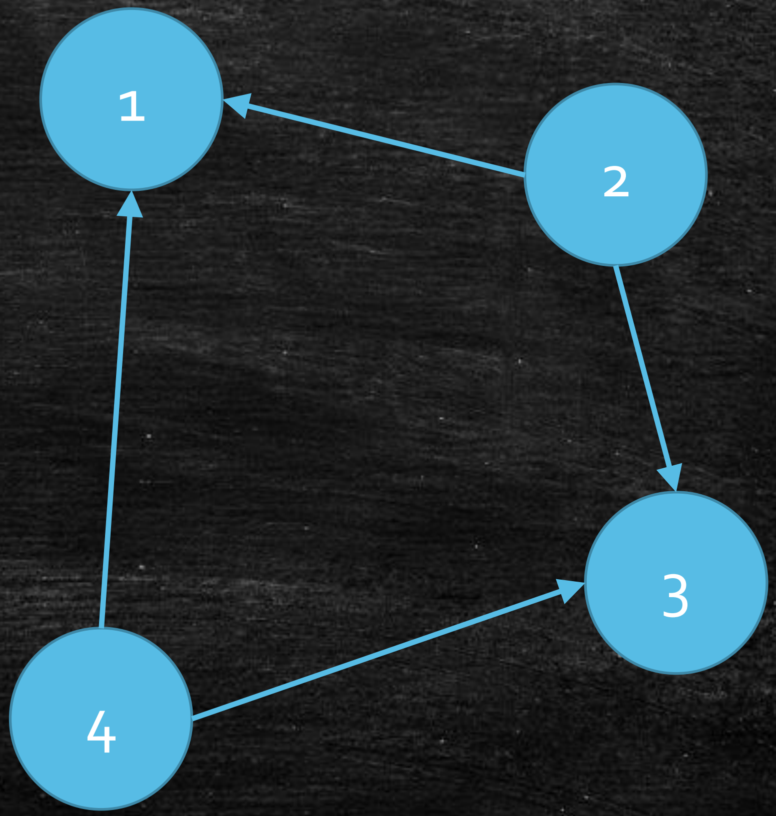
- Adjacency Matrix
- Adjacency List

Adjacency Matrix

Space: $O(|V|^2)$

- $|V| \times |V|$ matrix (2d array)
- $A[i,j] = \begin{cases} 1 & (i,j) \in E \\ 0 & (i,j) \notin E \end{cases}$

	1	2	3	4
1	0	0	0	0
2	1	0	1	0
3	0	0	0	0
4	1	0	1	0

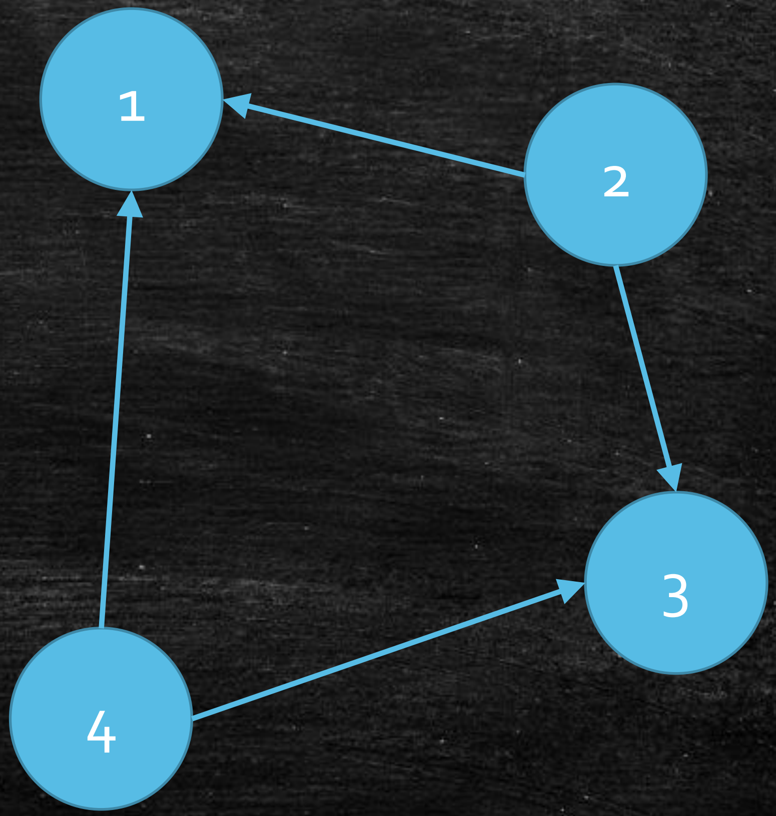


Adjacency List

Space: $O(|V| + |E|)$

- Linked list $adj[u]$ for each $u \in V$
- The list contains all u's **neighbor**.

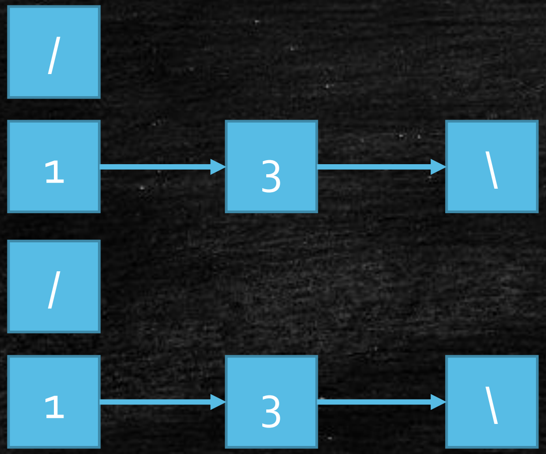
	1	2	3	4
1	0	0	0	0
2	1	0	1	0
3	0	0	0	0
4	1	0	1	0

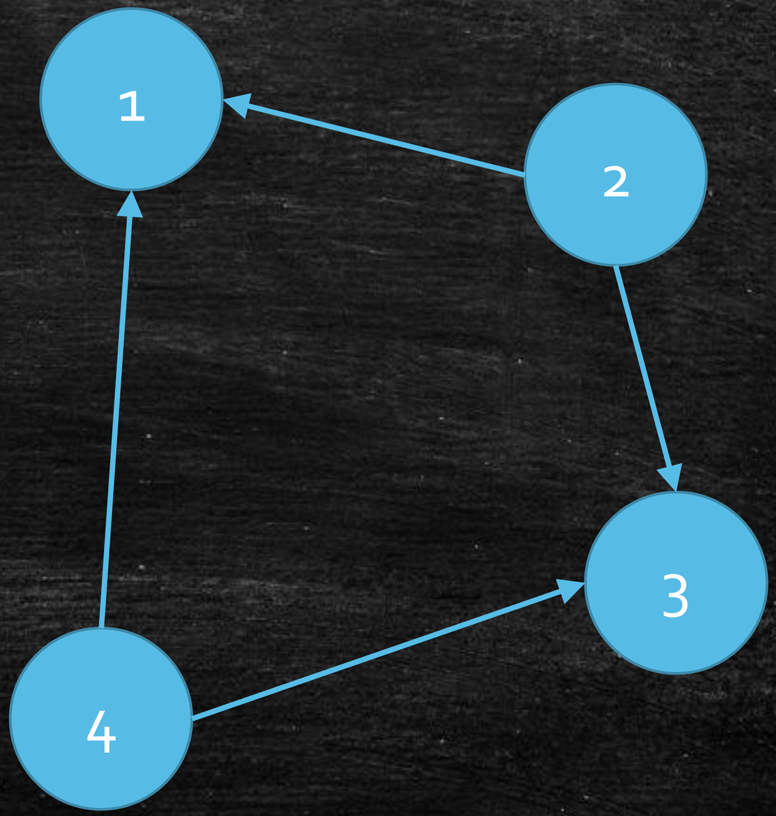


Adjacency List

Space: $O(|V| + |E|)$

- Linked list $adj[u]$ for each $u \in V$
- Node
  - v: the vertex
  - next
- Example
- $adj[1]$
- $adj[2]$
- $adj[3]$
- $adj[4]$





How to program?

- **Input:** The graph size $|V|$ and |E|, and |E| arcs.
- **Output:** The Adjacent Matrix or List

Create the Adjacent List

For each $(u, v) \in E$

$\quad node \leftarrow new\ Node$

$\quad node.v \leftarrow v$

$\quad node.next \leftarrow adj[u]$

$\quad \mathrm{adj}[u] = node$

Basic Graph Properties

- Reachability
  - Can we go from u to v?
  - Is v the friend of the friend of the friend ……. of v?
  - Can we travel from city u to v?
- Connected Components
  - Undirected version
  - A maximal subgraph that each two vertices are reachable.
  - A group of people who know each others
  - Directed version?

Reachability problem

- **Input:** A graph $G(V,E)$, represented by an Adjacent List, and a vertex u.
- **Output:** The set of vertices u can reach.

Observations

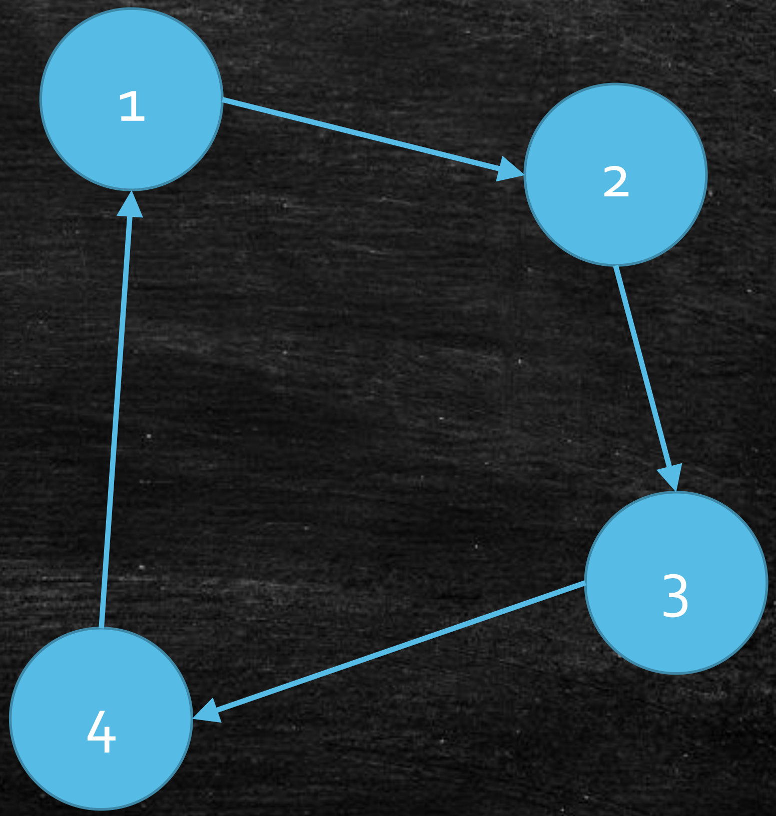
- Basic **observation**:
  - If v is in the Adjacent List (neighbor set) of u?
  - v is reachable.
- Advanced **observation**:
  - If v is reachable
  - Vertices in $v's$ Adjacent List (neighbor set) is also reachable.

Algorithmic Idea

- Explore & Explore
  - Explore from u
    - If v is in the Adjacent List of u
    - Continue to explore from v

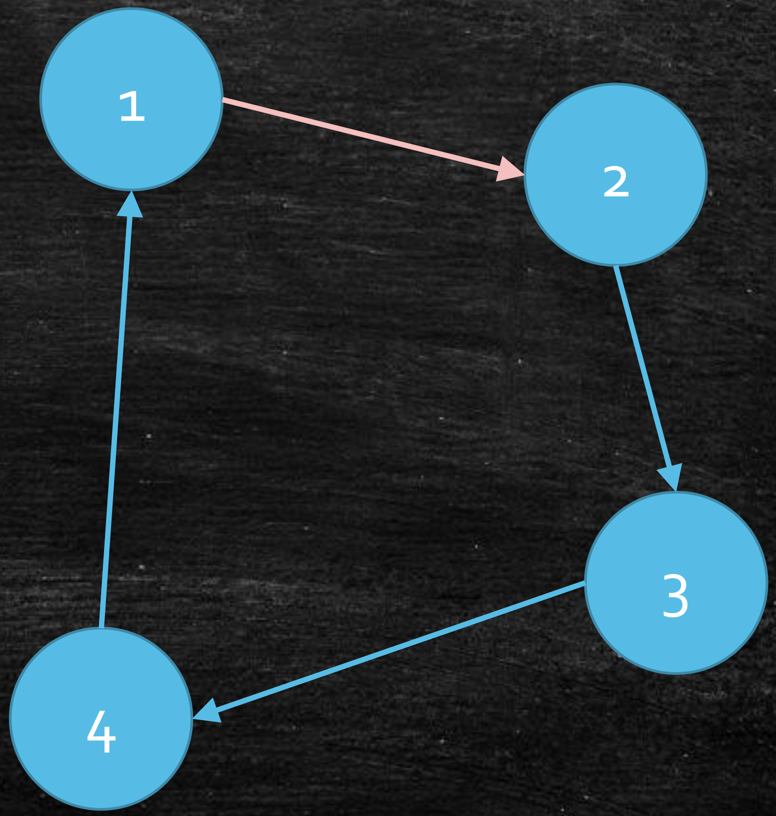
Algorithmic Idea

- Explore & Explore
  - Explore from u
    - If v is in the Adjacent List of u
    - Continue to explore from v
- Have a try!



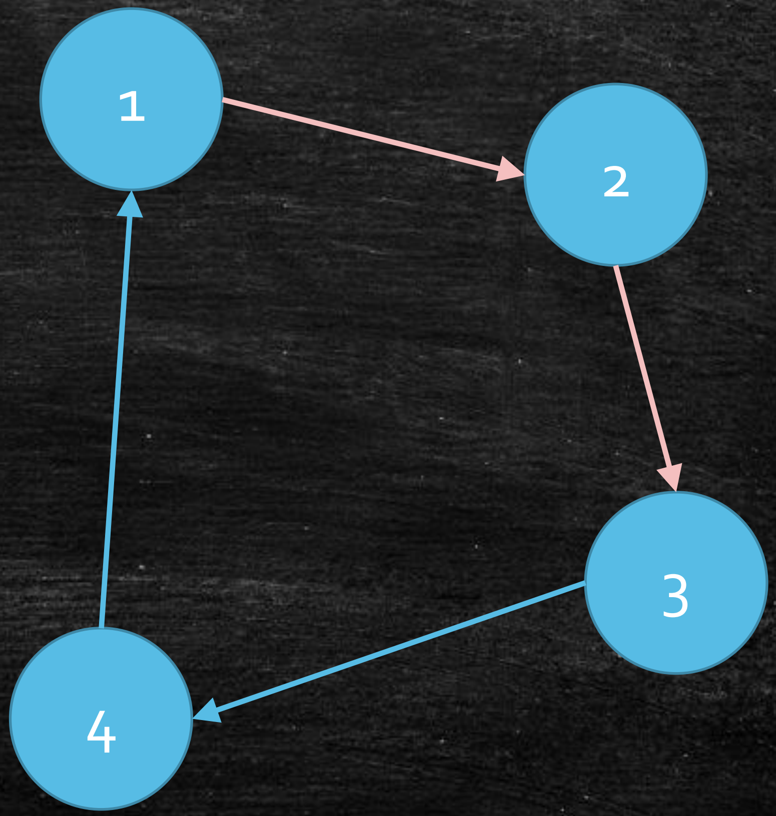
Algorithmic Idea

- Explore & Explore
  - Explore from u
    - If v is in the Adjacent List of u
    - Continue to explore from v
- Have a try!



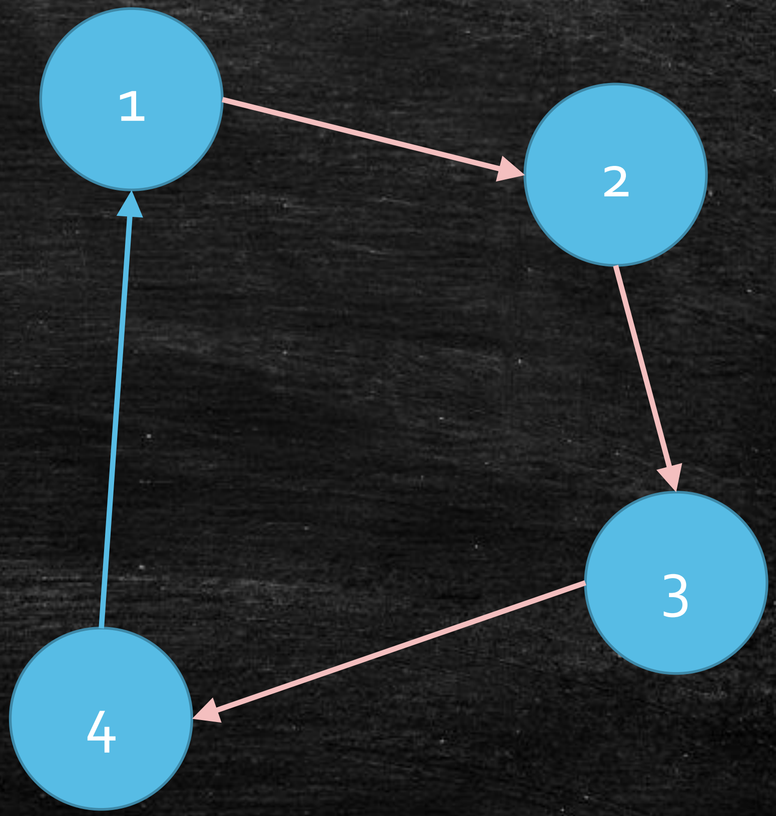
Algorithmic Idea

- Explore & Explore
  - Explore from u
    - If v is in the Adjacent List of u
    - Continue to explore from v
- Have a try!



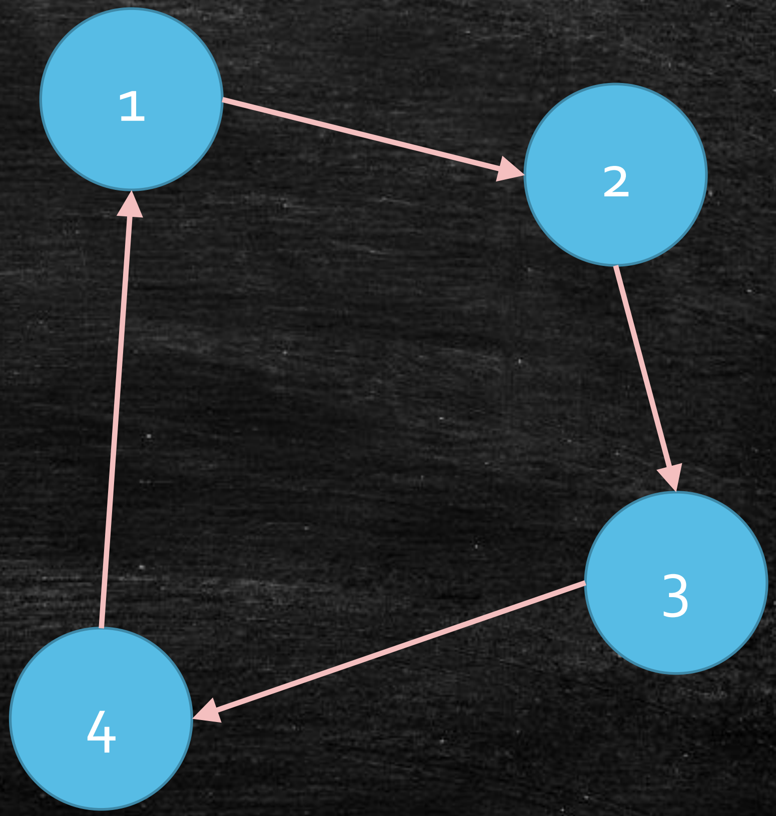
Algorithmic Idea

- Explore & Explore
  - Explore from u
    - If v is in the Adjacent List of u
    - Continue to explore from v
- Have a try!



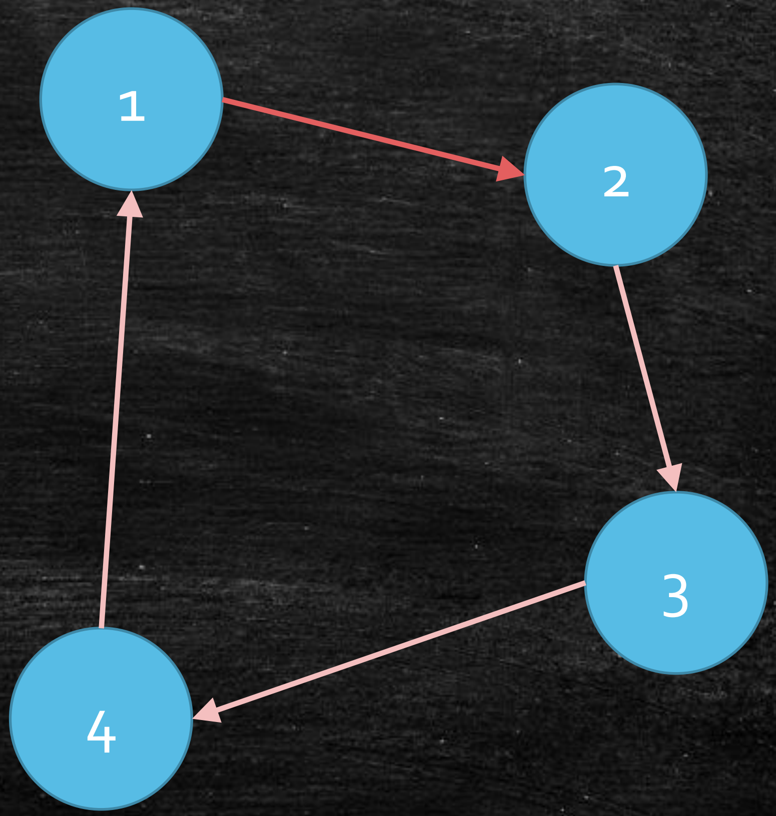
Algorithmic Idea

- Explore & Explore
  - Explore from u
    - If v is in the Adjacent List of u
    - Continue to explore from v
- Have a try!



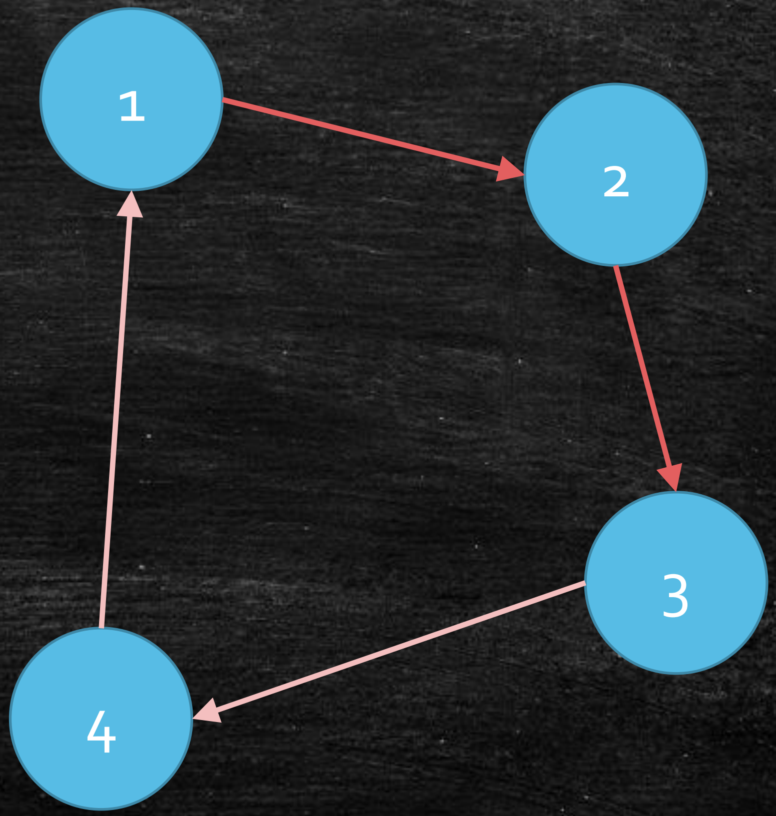
Algorithmic Idea

- Explore & Explore
  - Explore from u
    - If v is in the Adjacent List of u
    - Continue to explore from v
- Have a try!



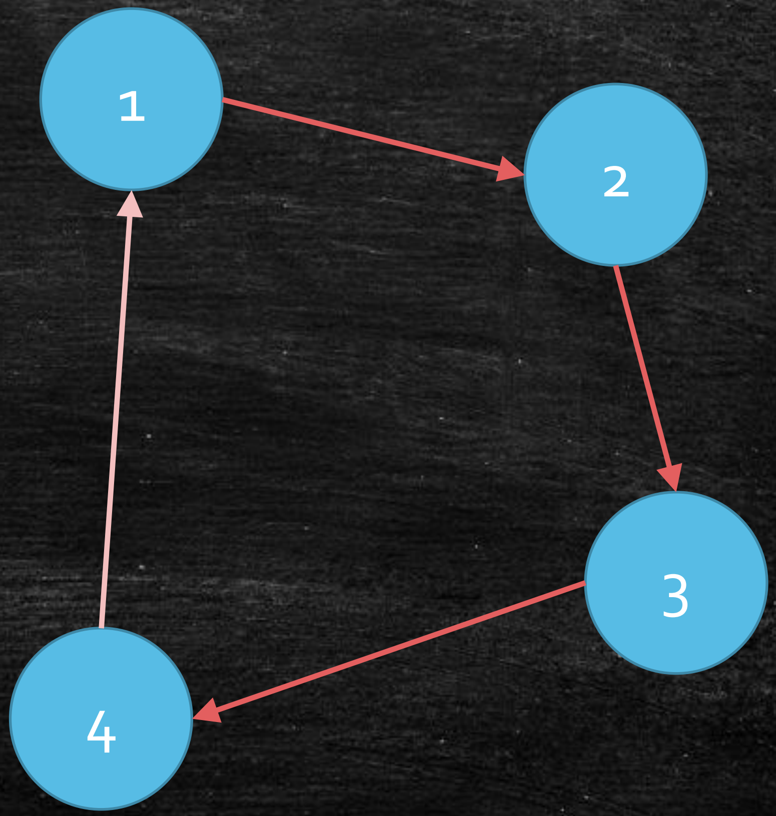
Algorithmic Idea

- Explore & Explore
  - Explore from u
    - If v is in the Adjacent List of u
    - Continue to explore from v
- Have a try!



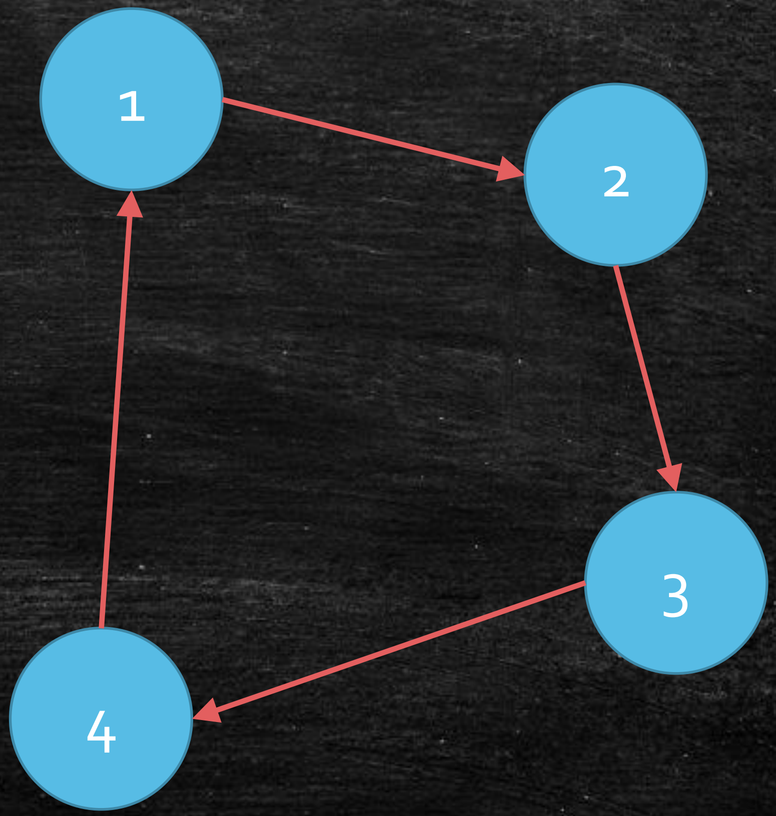
Algorithmic Idea

- Explore & Explore
  - Explore from u
    - If v is in the Adjacent List of u
    - Continue to explore from v
- Have a try!



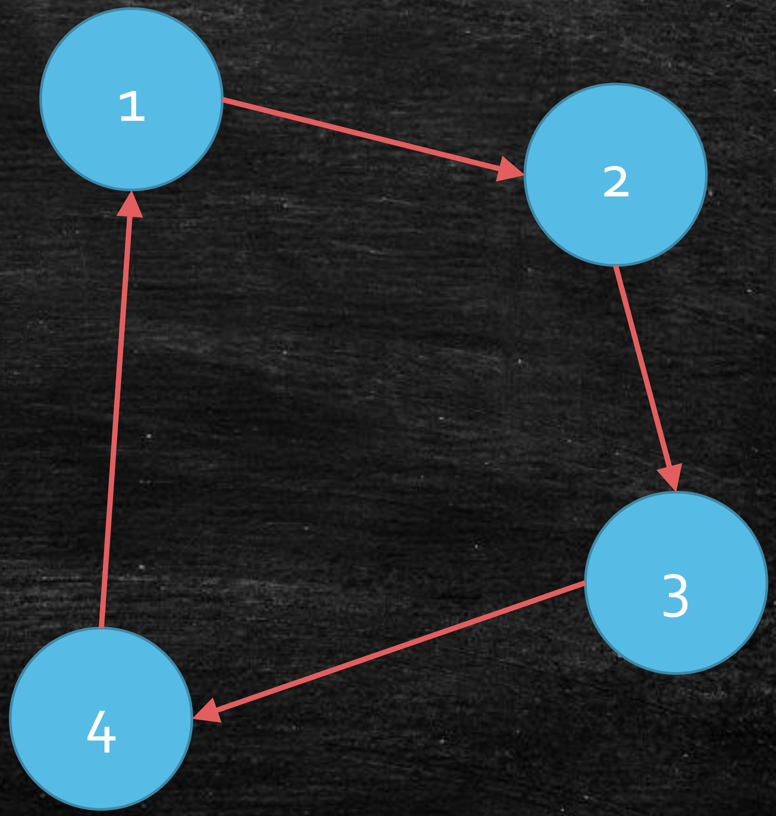
Algorithmic Idea

- Explore & Explore
  - Explore from u
    - If v is in the Adjacent List of u
    - Continue to explore from v
- Have a try!



Algorithmic Idea

- Explore & Explore
  - Explore from u
    - If v is in the Adjacent List of u
    - Continue to explore from v
- Have a try!
- Problem: Cycle!
  - $1 \to 2 \to 3 \to 4 \to 1$
- Solution
  - **Mark** a vertex when we reach it
  - Do not explore **marked** vertices



Exploring Algorithm

- After explore(v)
- The marked vertices can be reached.
- It is a connected component that contains v!
- What if we want to know all?
  - We want to know all connected components.
  - We want to search all the graph.

Function explore(v)
    $marked[v] \leftarrow true$
    for each $(u, v) \in E$
        if $marked[u] = false$
            explore(u)

Depth-First Search

- What is DFS?
  - Explore & Explore
- Questions
  - How to loop all $(u, v) \in E$?
  - What is the running time of DFS?

```
Function explore(v)
    marked[v] ← true
    for each (u, v) ∈ E
        if marked[u] = false
            explore(u)
```

```
Function dfs(G)
    for each v ∈ V
        if marked[v] = false
            explore(v)
```

Running Time of DFS

- Questions
  - What is the running time of DFS?
  - Seems $O(|V|^2)$?

Function explore(v)
    $marked[v] \leftarrow true$
    for each $(u, v) \in E$
        if $marked[u] = false$
            explore(u)

At most $|V|$ times.

Function dfs(G)
    for each $\mathrm{v} \in V$
        if $marked[v] = false$
            explore(v)

At most $|V|$ times.

Running Time of DFS

Function explore(v)
  $marked[v] \leftarrow true$
  for each $(u, v) \in E$
    if $marked[u] = false$
      explore(u)

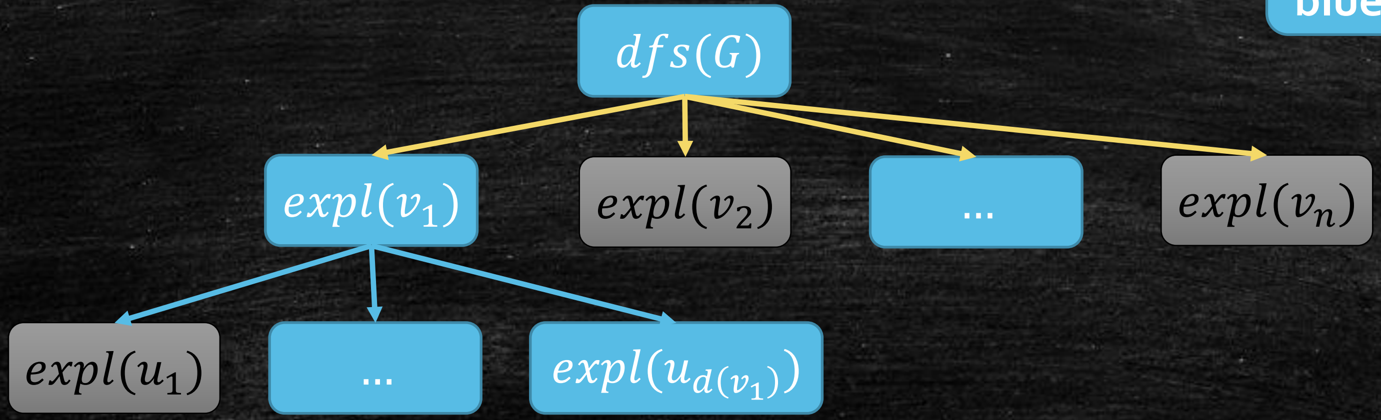
Charge to **blue edges**.

Charge to **blue vertices**.

Function dfs(G)
  for each $v \in V$
    if $marked[v] = false$
      explore(v)

Charge to **yellow edges**.

Charge to **blue vertices**.



Running Time of DFS

Function explore(v)
  $marked[v] \leftarrow true$
  for each $(u, v) \in E$
    if $marked[u] = false$
      explore(u)

Charge to **blue edges**.

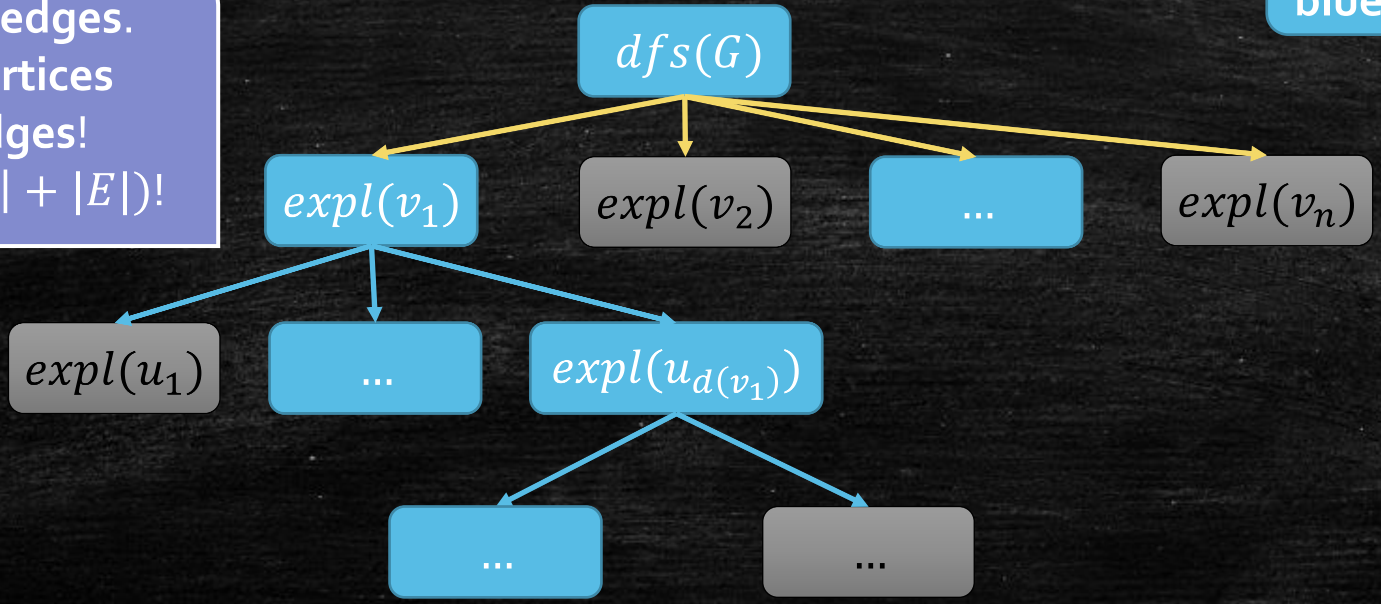
Charge to **blue vertices**.

Function dfs(G)
  for each v $\in V$
    if $marked[v] = false$
      explore(v)

Charge to **yellow edges**.

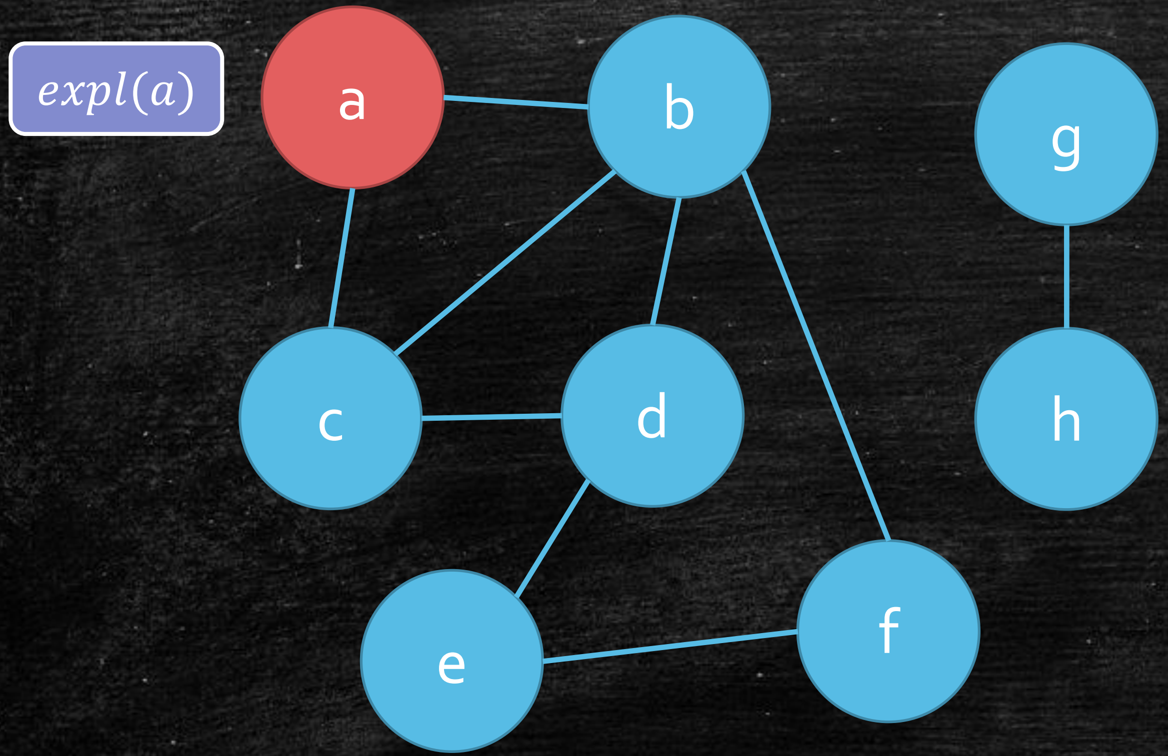
Charge to **blue vertices**.

- We have $|V|$ **yellow edges**.
- We have $|V|$ **blue vertices**
- We have $|E|$ **blue edges**!
- Running Time: $O(|V| + |E|)$!



DFS in undirected graphs

- How we DFS an undirected graph?

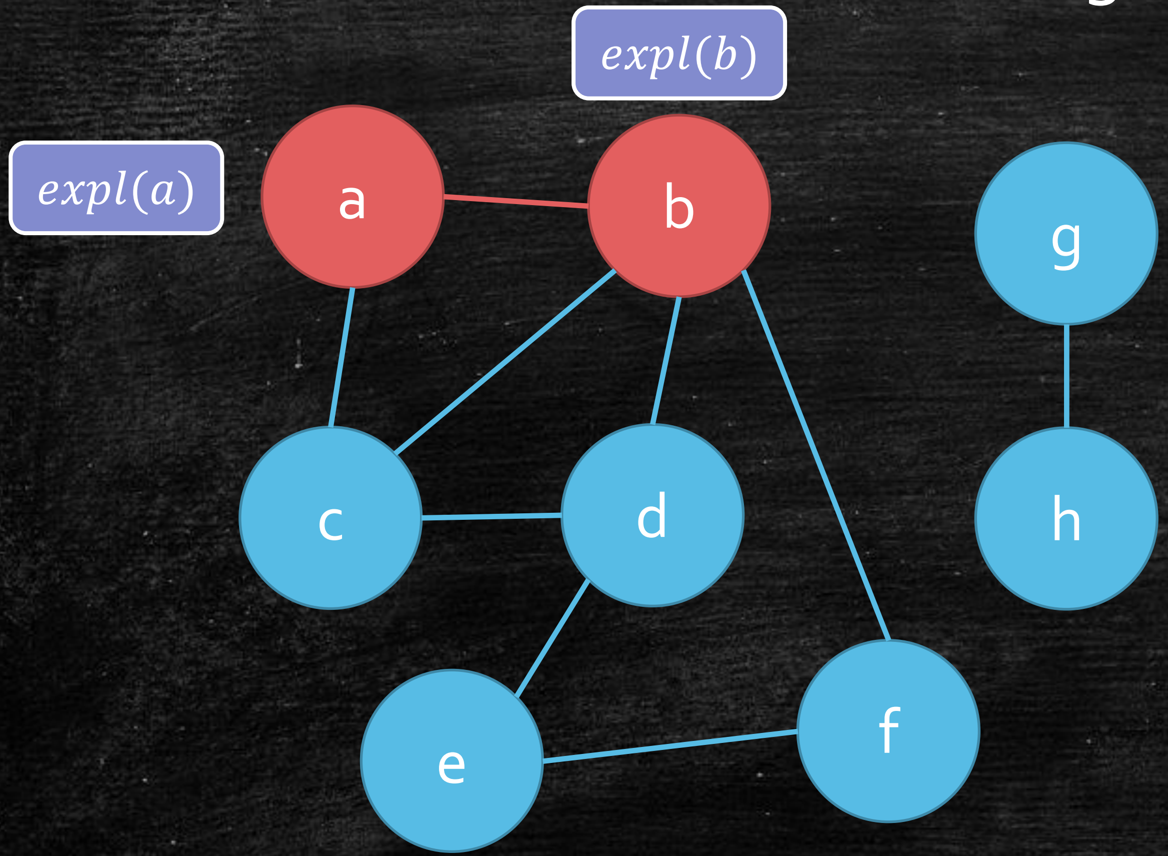


Function explore(v)
$\quad marked[v] \leftarrow true$
$\quad$ **for each** $(u, v) \in E$
$\quad\quad$ **if** $marked[u] = false$
$\quad\quad\quad$ explore(u)

Function dfs(G)
$\quad$ **for each** v $\in V$
$\quad\quad$ **if** $marked[v] = false$
$\quad\quad\quad$ explore(v)

DFS in undirected graphs

- How we DFS an undirected graph?

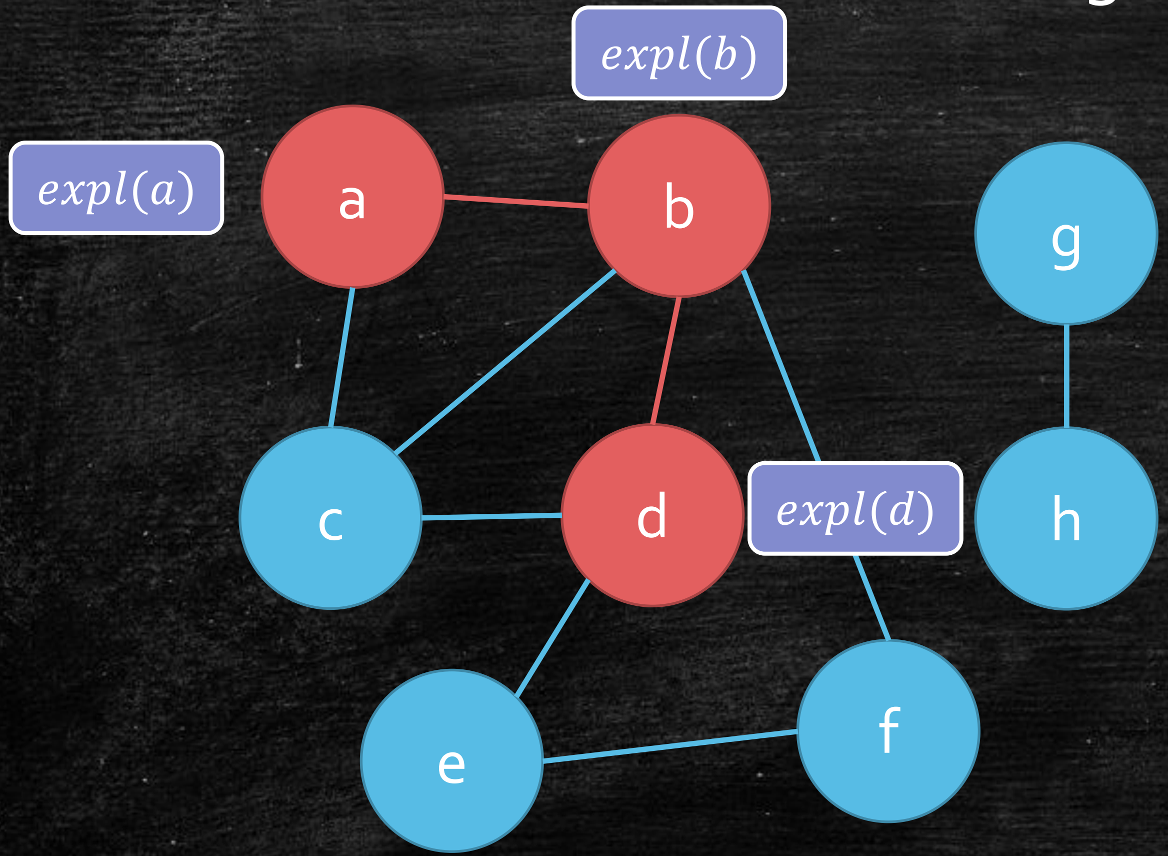


Function explore(v)
    $marked[v] \leftarrow true$
    for each $(u, v) \in E$
        if $marked[u] = false$
            explore(u)

Function dfs(G)
    for each v $\in V$
        if $marked[v] = false$
            explore(v)

DFS in undirected graphs

- How we DFS an undirected graph?

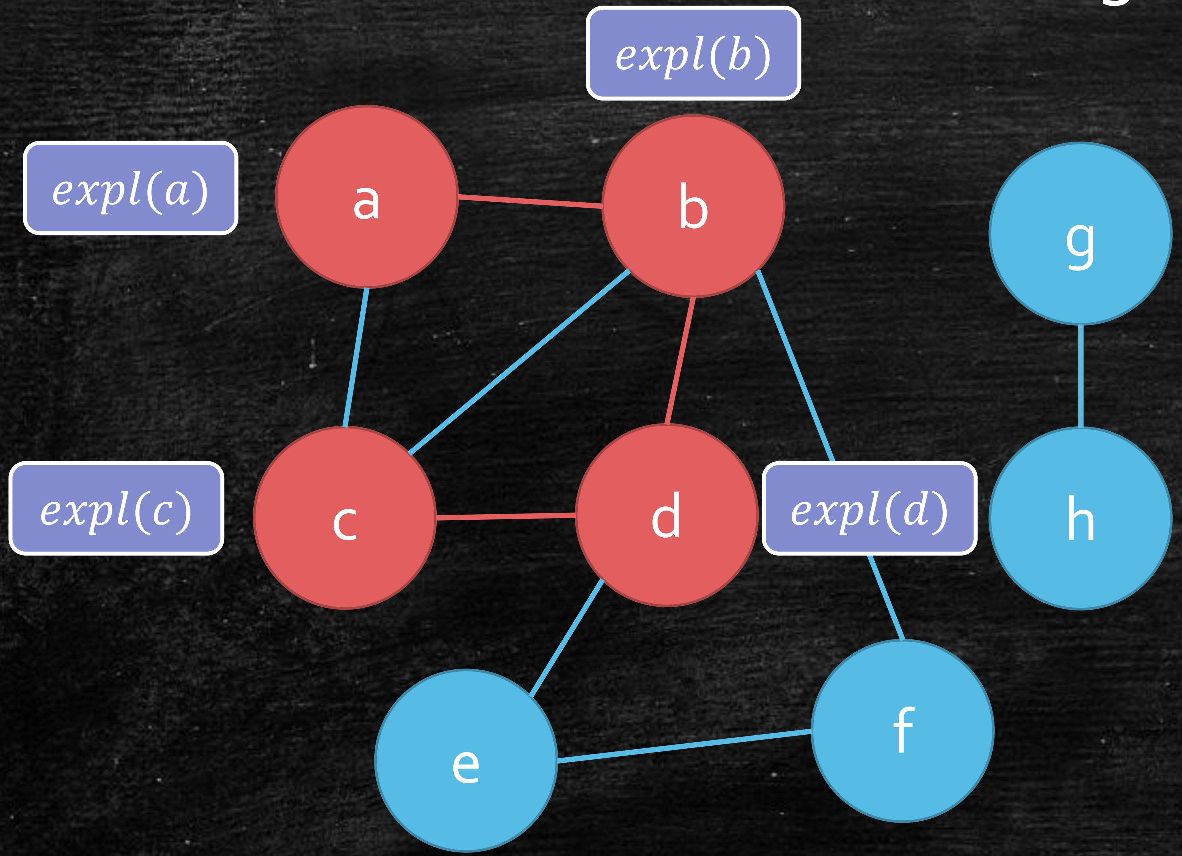


Function explore(v)
$marked[v] \leftarrow true$
for each $(u, v) \in E$
if $marked[u] = false$
explore(u)

Function dfs(G)
for each v $\in V$
if $marked[v] = false$
explore(v)

DFS in undirected graphs

- How we DFS an undirected graph?

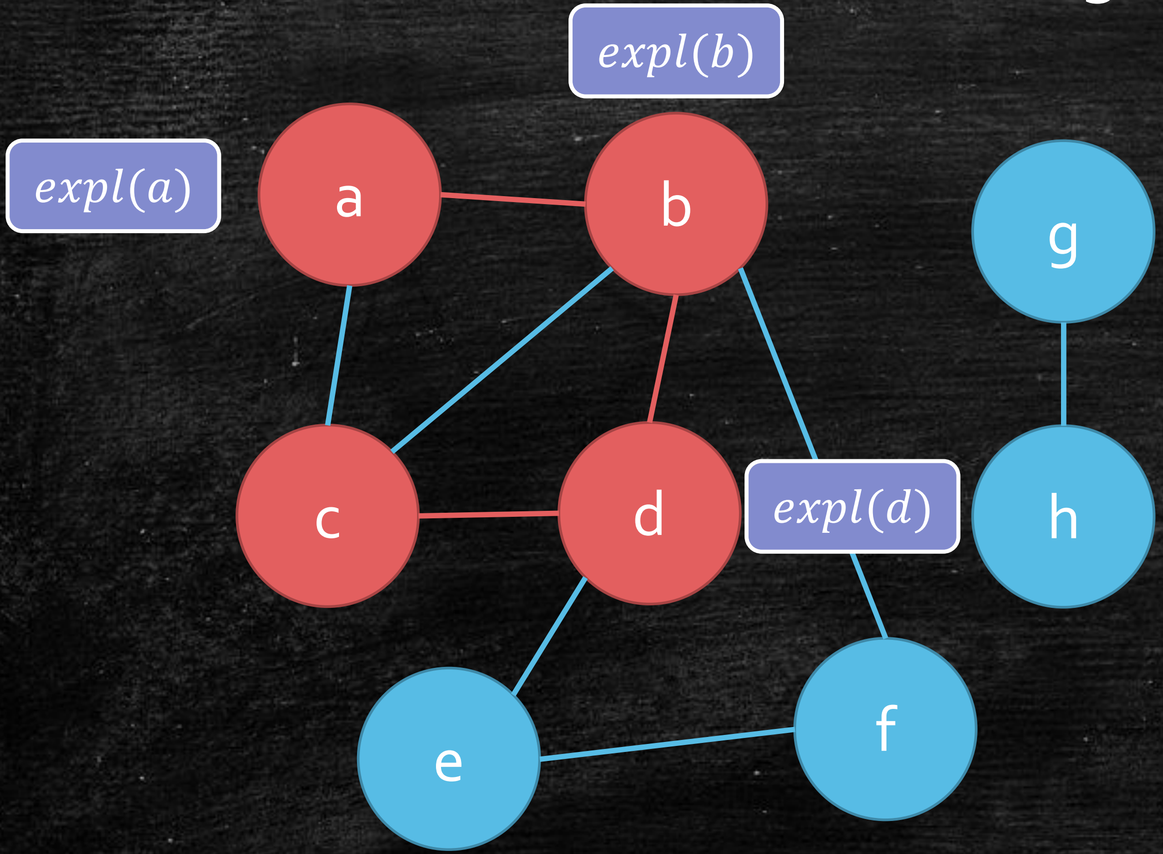


Function explore(v)
 $marked[v] \leftarrow true$
 for each $(u, v) \in E$
  if $marked[u] = false$
   explore(u)

Function dfs(G)
 for each v $\in V$
  if $marked[v] = false$
   explore(v)

DFS in undirected graphs

- How we DFS an undirected graph?

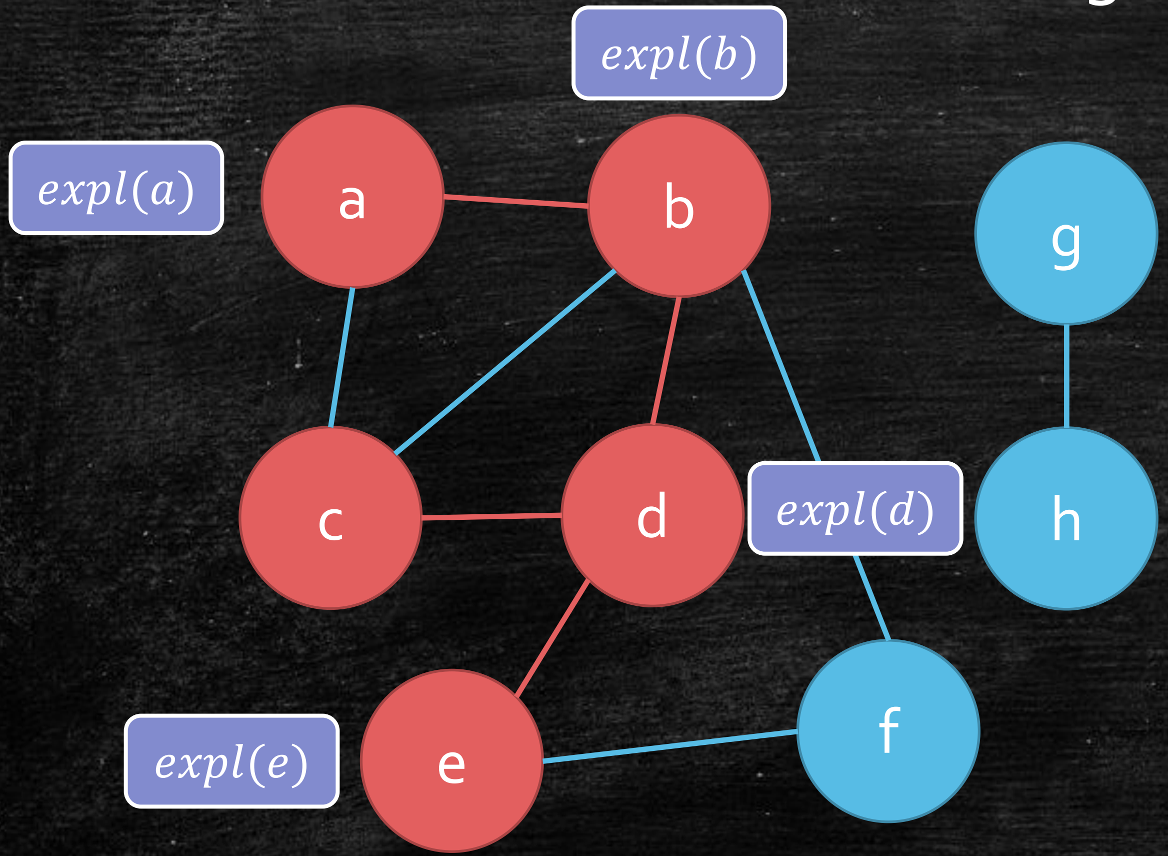


Function explore(v)
$marked[v] \leftarrow true$
for each $(u, v) \in E$
if $marked[u] = false$
explore(u)

Function dfs(G)
for each v $\in V$
if $marked[v] = false$
explore(v)

DFS in undirected graphs

- How we DFS an undirected graph?

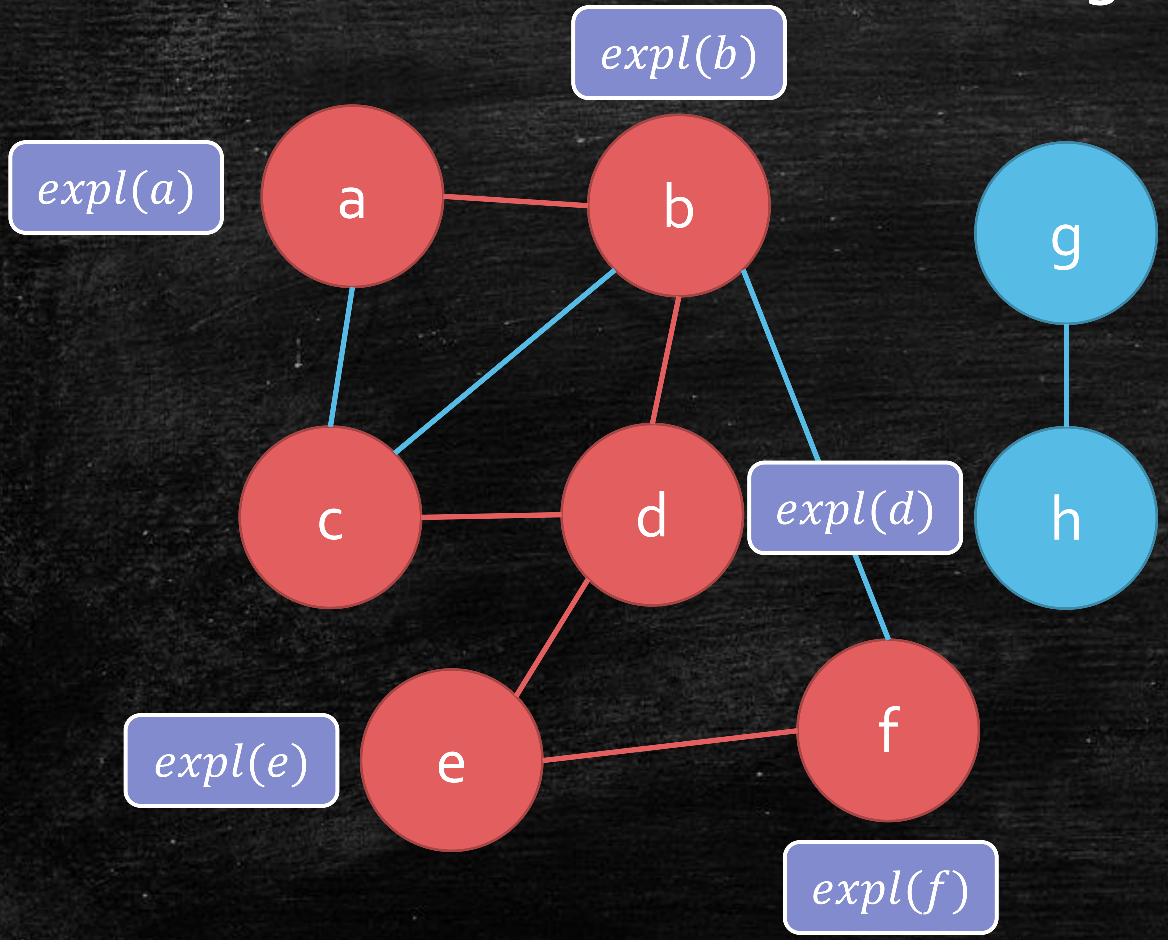


Function explore(v)
    $marked[v] \leftarrow true$
    for each $(u, v) \in E$
        if $marked[u] = false$
            explore(u)

Function dfs(G)
    for each v $\in V$
        if $marked[v] = false$
            explore(v)

DFS in undirected graphs

- How we DFS an undirected graph?

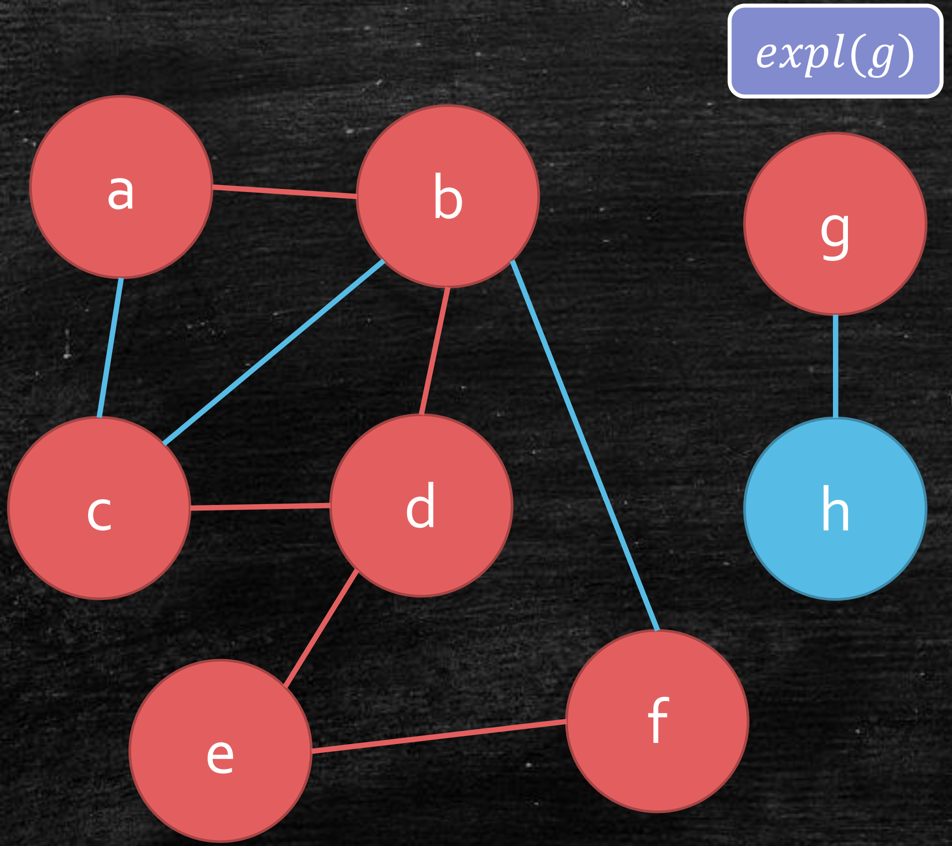


Function explore(v)
    $marked[v] \leftarrow true$
    for each $(u, v) \in E$
        if $marked[u] = false$
            explore(u)

Function dfs(G)
    for each v $\in V$
        if $marked[v] = false$
            explore(v)

DFS in undirected graphs

- How we DFS an undirected graph?

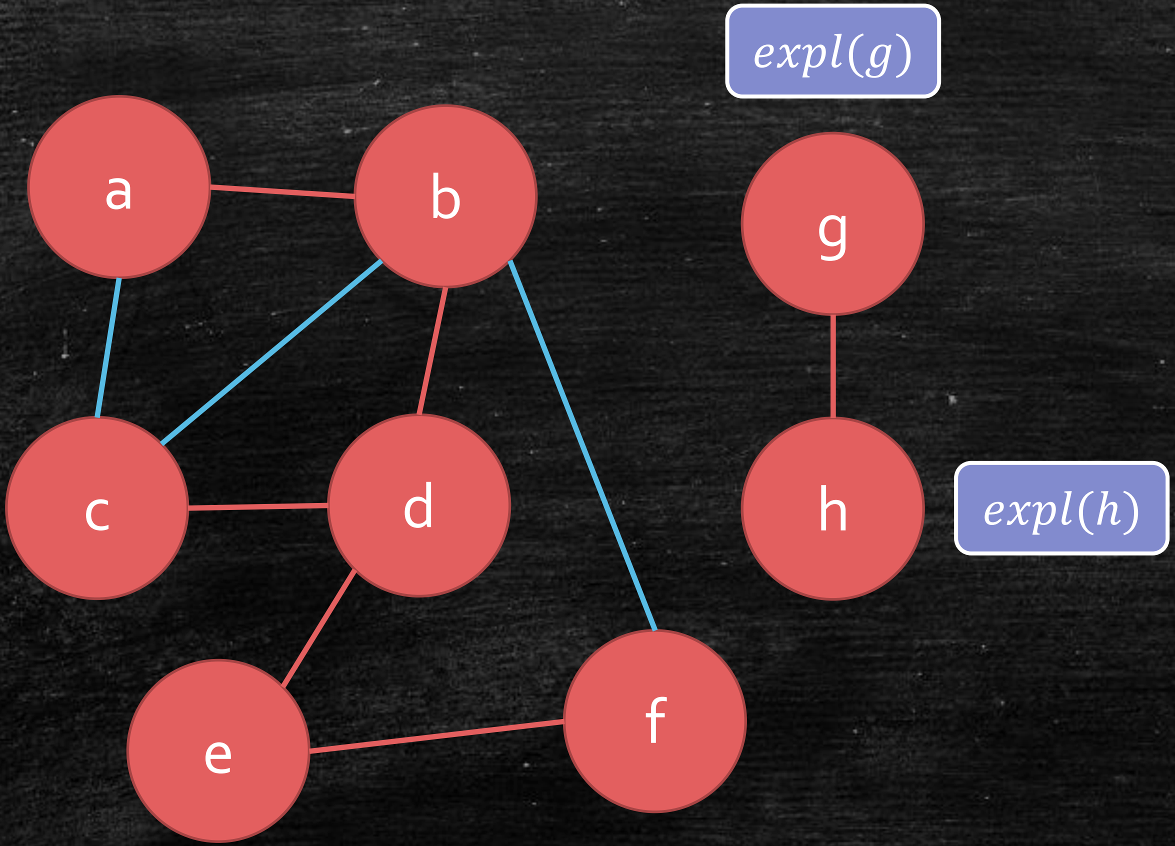


Function explore(v)
    $marked[v] \leftarrow true$
    for each $(u, v) \in E$
        if $marked[u] = false$
            explore(u)

Function dfs(G)
    for each v $\in V$
        if $marked[v] = false$
            explore(v)

DFS in undirected graphs

- How we DFS an undirected graph?

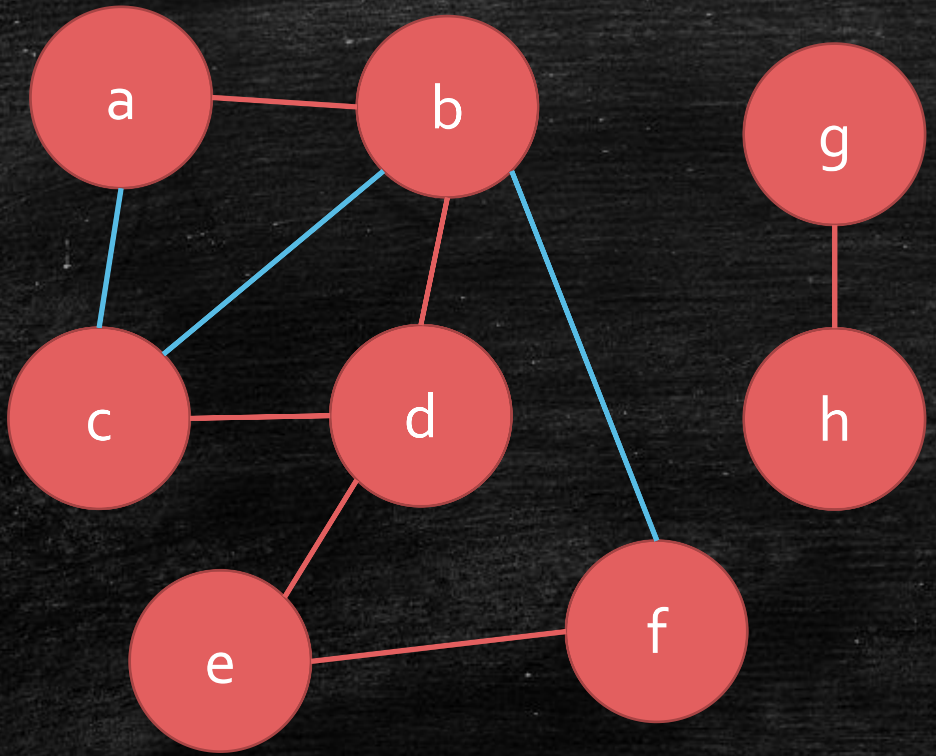


Function explore(v)
    $marked[v] \leftarrow true$
    for each $(u, v) \in E$
        if $marked[u] = false$
            explore(u)

Function dfs(G)
    for each v $\in V$
        if $marked[v] = false$
            explore(v)

Discussion

- How many connected components?
  - How to prove your solution?

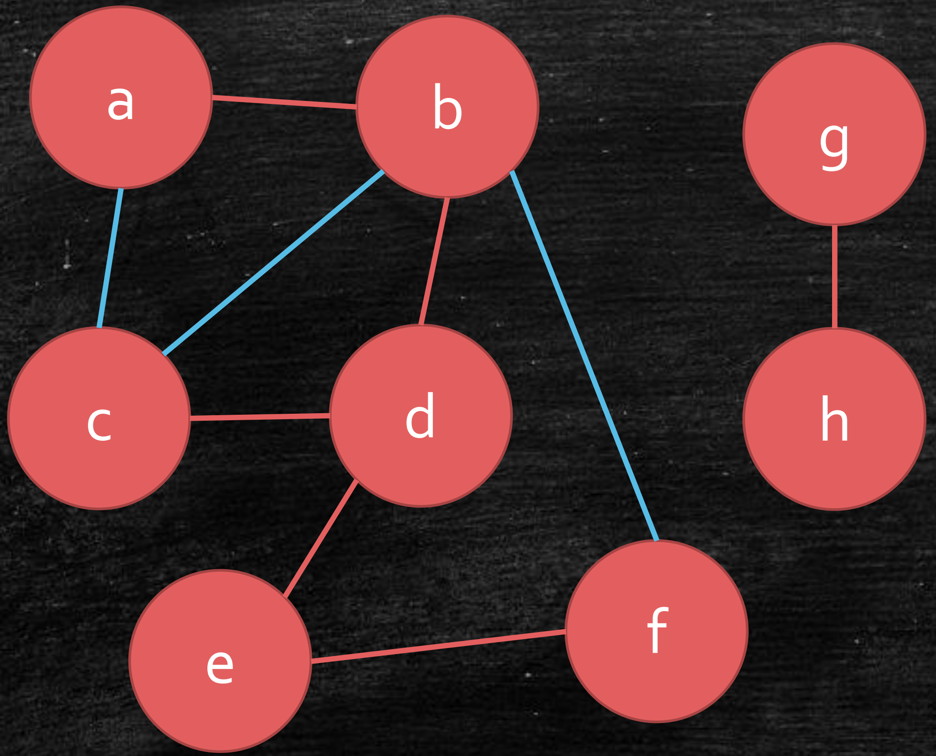


Function explore(v)
  $marked[v] \leftarrow true$
  for each $(u, v) \in E$
    if $marked[u] = false$
      explore(u)

Function dfs(G)
  for each v $\in V$
    if $marked[v] = false$
      explore(v)

Discussion

- How many connected components?
  - How to prove your solution?



Function explore(v)
  $marked[v] \leftarrow true$
  for each $(u, v) \in E$
    if $marked[u] = false$
      explore(u)

Function dfs(G)
  for each v $\in V$
    if $marked[v] = false$
      explore(v)

Times of "explore" = number of connected components

Simply get all the connected components!

Why it is correct?

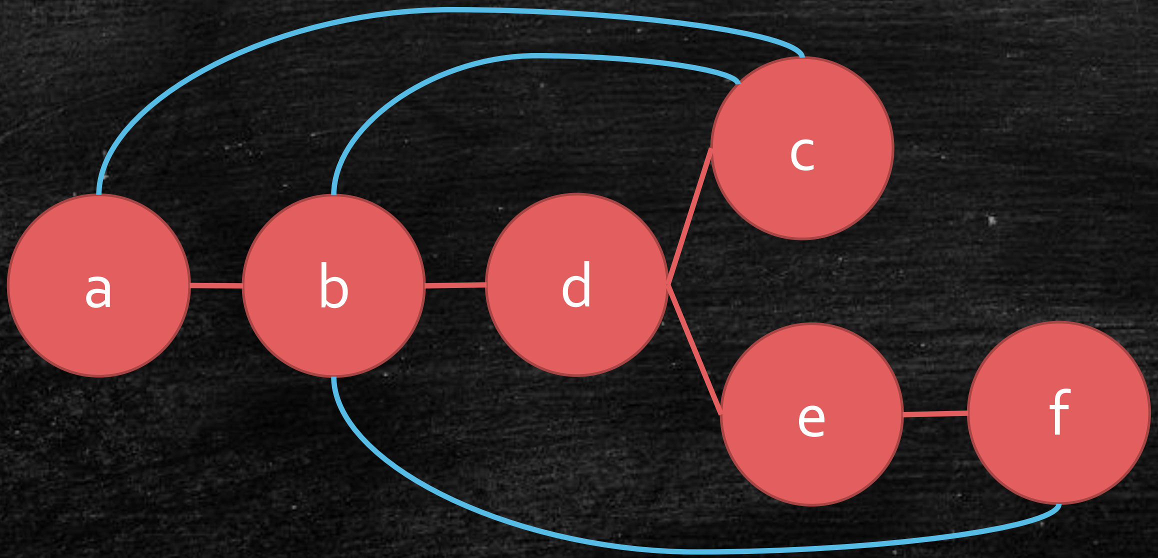
DFS is more powerful than you think!

Let us discuss some properties of DFS.

DFS Tree (One Connected Component)

- Show the relationship among vertices
  - Root: the first explored vertex
  - If we explore v from u, then v is u's child.

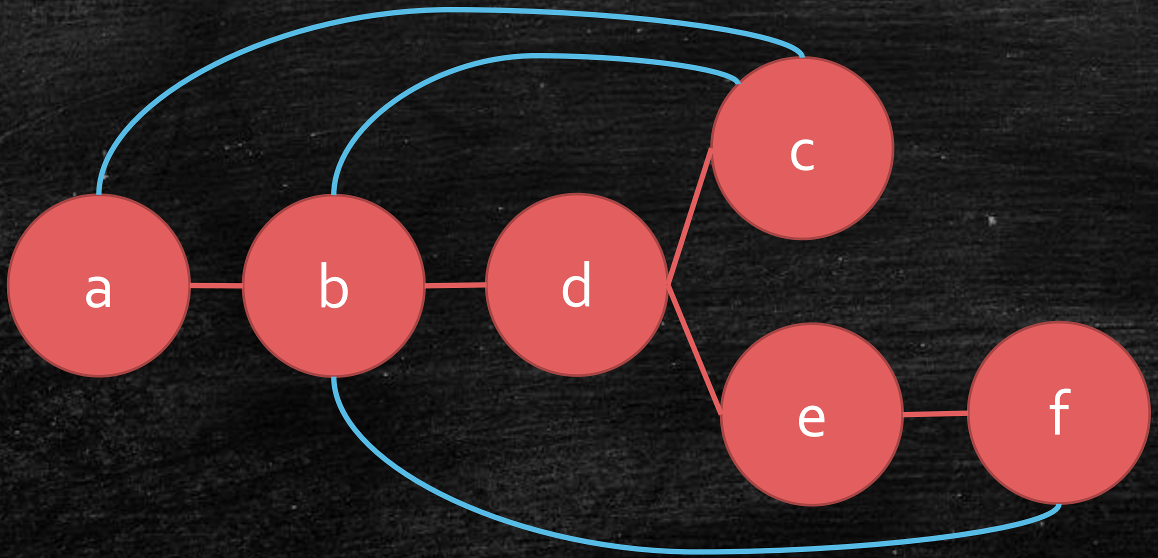
Function explore(v)
    $marked[v] \leftarrow true$
    for each $(u, v) \in E$
        if $marked[u] = false$
            explore(u)



Why it is a tree?

DFS Tree (One Connected Component)

- Show the relationship among vertices
  - Root: the first explored vertex
  - If we explore v from u, then v is u's child.



Function explore(v)
    $marked[v] \leftarrow true$
    for each $(u, v) \in E$
        if $marked[u] = false$
            explore(u)

- Kind of edges
  - Tree edges
  - Back edges

Why we introduce the DFS tree?

- Do we have **cycles** in an undirected graph?
- What is a **cycle**?
  - $(a, b), (b, c), (c, d), \ldots.., (z, a)$
- Observation
  - There must be a marked vertex a.
  - (z, a) should be a back edge.
- T: DFS tree of G
- **Conjecture:** T has back edges $\leftarrow\rightarrow$ G has cycles
- How to prove it?

Proof of The Conjecture

- **Conjecture:** T has back edges $\longleftrightarrow$ G has cycles
- Proof
- $\rightarrow$: If T has a back edge, then G has a cycle.
  - Can we point out a cycle based on this back edge?
- $\leftarrow$: If G has a cycle, then T has a back edge.
  - Can we point out one back edge in the cycle?

Conclusion

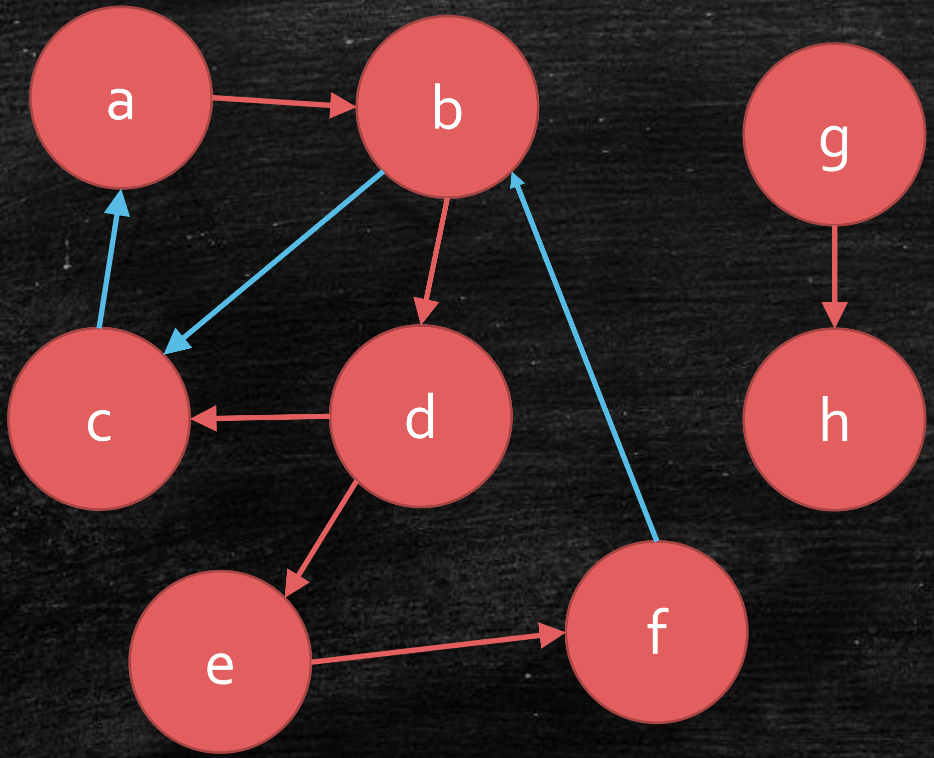
- On undirected graphs, DFS can:
  - Find $v's$ connected component.
  - Find all connected components.
  - Detect whether the graph contains cycles.
- Let us move to directed graphs!

DFS on Directed Graphs

What is the difference?

DFS on Directed Graphs

- Answer: verbatim, but with directions.

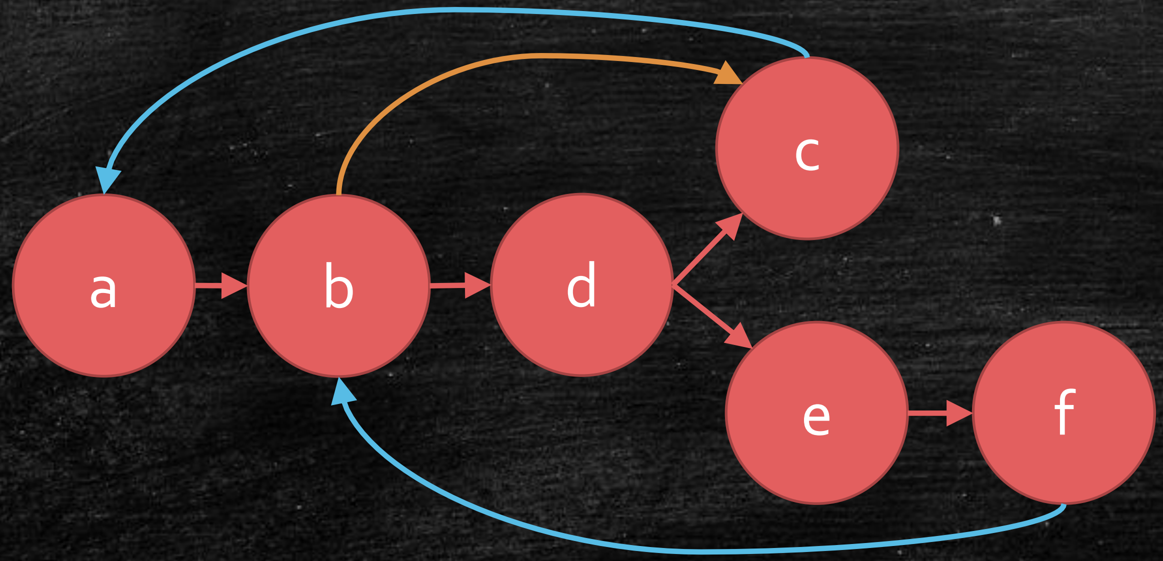


Function explore(v)
    $marked[v] \leftarrow true$
    for each $(v, u) \in E$
        if $marked[u] = false$
            explore(u)

Function dfs(G)
    for each v $\in V$
        if $marked[v] = false$
            explore(v)

DFS on Directed Graphs

- What about DFS trees?

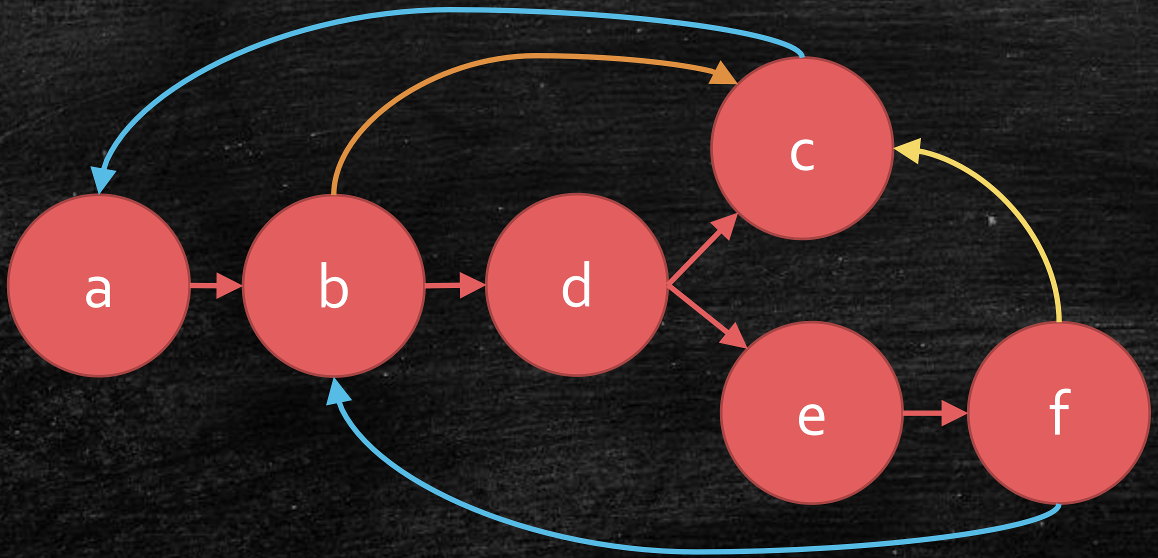


Function explore(v)
  $marked[v] \leftarrow true$
  for each $(v, u) \in E$
    if $marked[u] = false$
      explore(u)

- Kind of edges
  - Tree edges
  - Back edges

DFS on Directed Graphs

- What about DFS trees?

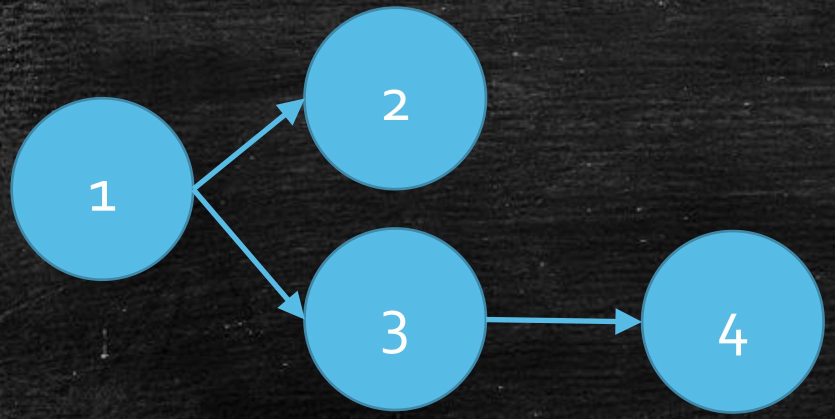


Function explore(v)
$\quad marked[v] \leftarrow true$
$\quad$ **for each** $(v,u) \in E$
$\quad\quad$ **if** $marked[u] = false$
$\quad\quad\quad$ explore(u)

- Kind of edges
  - Tree edges
  - Forward edges
  - Back edges
  - Cross edges

Application: Topological Ordering

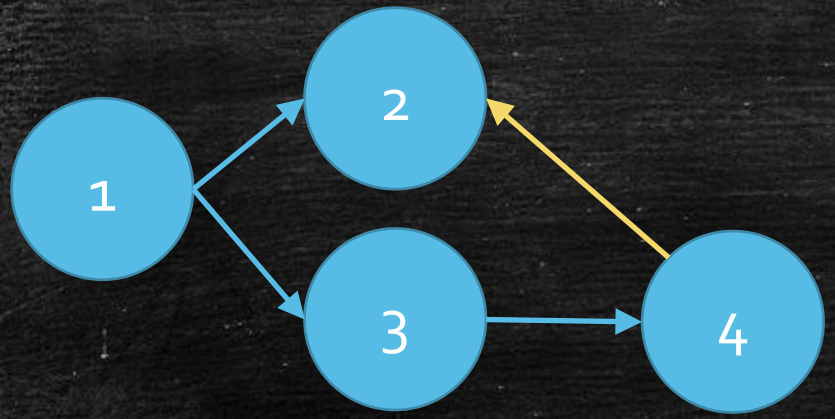
- A pre-requisite requirements graph



- We want to find an order to finish these course.
- Can we find an order in any given graph?

Application: Topological Ordering

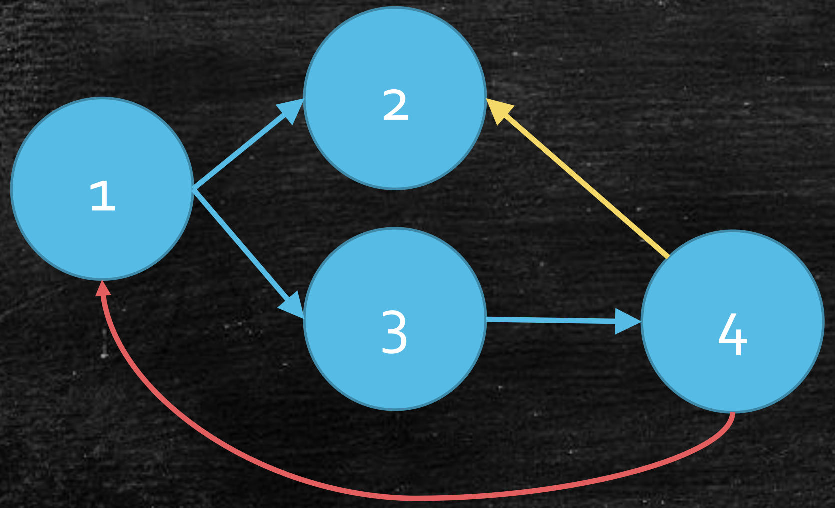
- A pre-requisite requirements graph



- We want to find an order to finish these course.
- Can we find an order in any given graph?

Application: Topological Ordering

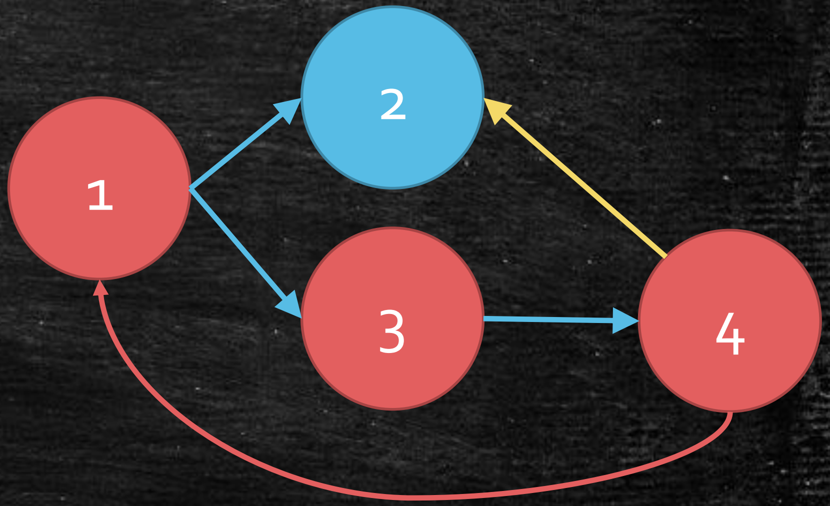
- A pre-requisite requirements graph



- We want to find an order to finish these course.
- Can we find an order in any given graph?

Why we can not find an order?

- **A directed Cycle**
  - 1 -> 3 -> 4 -> 1
  - Contradiction!
- What if there is no cycle?
- **Directed Acyclic Graph (DAG)**
  - a directed graph that does not contain any cycle.

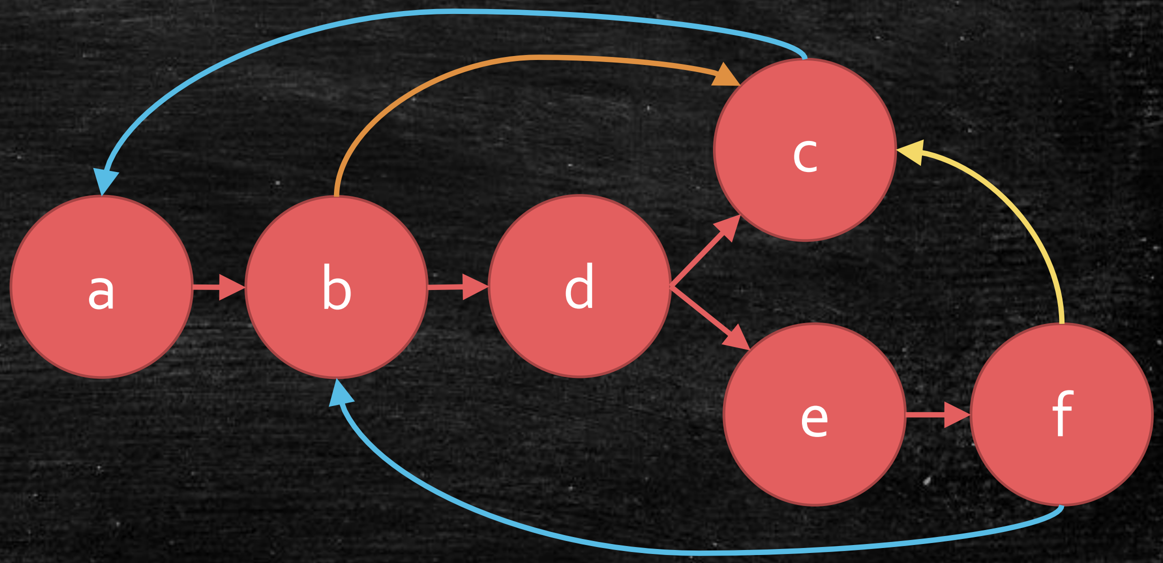


Topological Ordering for DAG

- Is DAG equals to a topological order?
- Known: not DAG -> no order
- Unknown: DAG -> an order
- How to prove?
- Construct a topological ordering for every DAG.
- Design an algorithm do topological ordering for DAG.

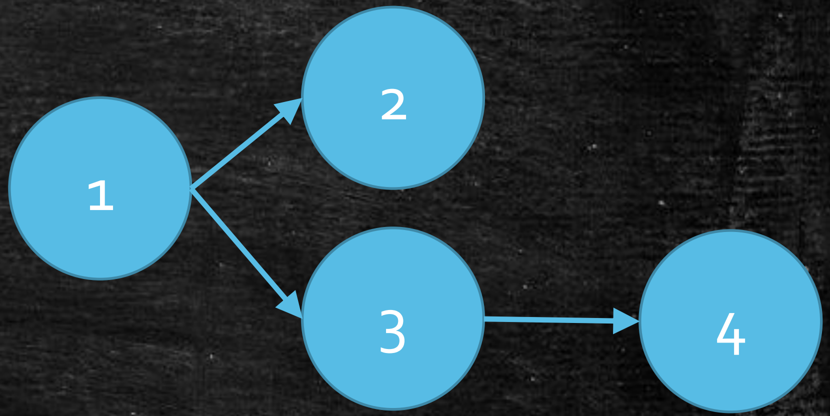
Topological Ordering for DAG

- Observation
  - DAG must have a **tail**.
  - **Tail:** vertices that do not have outgoing edges.
- Proof
  - Start from v
  - Does v has outgoing edges?
  - Yes: go to next v'
  - No: we are ok
  - Fact: we do not have cycles
  - → we can not go back
  - → we must stop at a tail.



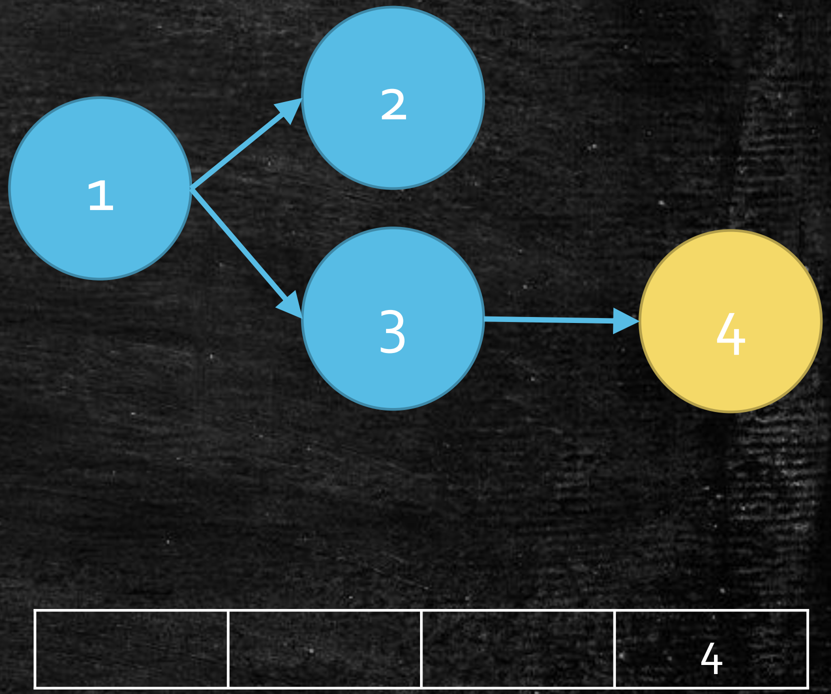
Topological Ordering for DAG

- Observation
  - DAG must have a **tail**.
  - **Tail:** vertices that do not have outgoing edges.
  - **Tail** can be the last one in the topological order.
- Algorithm
  - Find a **tail**.
  - Put **it** to be the last one in the topological order.
  - Remove the **tail** in the graph.
  - Repeat...



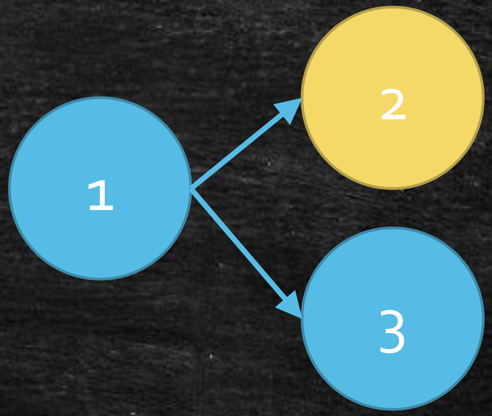
Topological Ordering for DAG

- Observation
  - DAG must have a **tail**.
  - **Tail:** vertices that do not have outgoing edges.
  - **Tail** can be the last one in the topological order.
- Algorithm
  - Find a **tail**.
  - Put **it** to be the last one in the topological order.
  - Remove the **tail** in the graph.
  - Repeat…



Topological Ordering for DAG

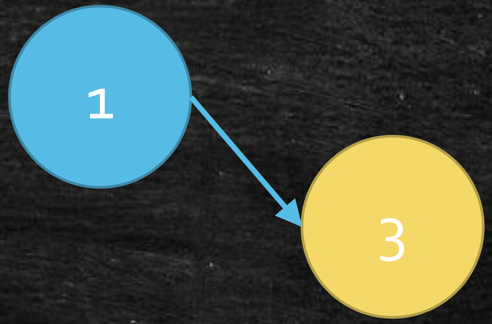
- Observation
  - DAG must have a **tail**.
  - **Tail:** vertices that do not have outgoing edges.
  - **Tail** can be the last one in the topological order.
- Algorithm
  - Find a **tail**.
  - Put **it** to be the last one in the topological order.
  - Remove the **tail** in the graph.
  - Repeat…



		2	4

Topological Ordering for DAG

- Observation
  - DAG must have a **tail**.
  - **Tail:** vertices that do not have outgoing edges.
  - **Tail** can be the last one in the topological order.
- Algorithm
  - Find a **tail**.
  - Put **it** to be the last one in the topological order.
  - Remove the **tail** in the graph.
  - Repeat…



	3	2	4

Topological Ordering for DAG

- Observation
  - DAG must have a **tail**.
  - **Tail:** vertices that do not have outgoing edges.
  - **Tail** can be the last one in the topological order.
- Algorithm
  - Find a **tail**.
  - Put **it** to be the last one in the topological order.
  - Remove the **tail** in the graph.
  - Repeat…

1	3	2	4

Topological Ordering for DAG

- Observation
  - DAG must have a **tail**.
  - **Tail:** vertices that do not have outgoing edges.
  - **Tail** can be the last one in the topological order.
- Algorithm
  - Find a **tail**.
  - Put **it** to be the last one in the topological order.
  - Remove the **tail** in the graph.
  - Repeat…

1	3	2	4

Correctness: Is the order feasible?

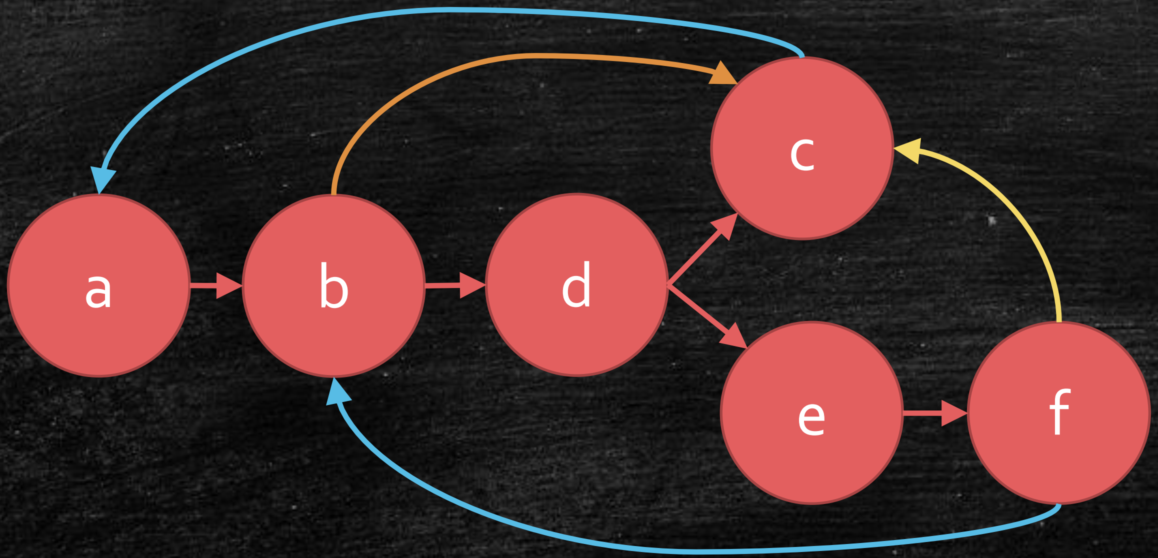
Running Time?

- Conclusion
  - We can find a feasible topological order for DAG.
  - DAG ←→ A topological order
- Algorithm
  - Find a **tail**.
  - Put **it** to be the last one in the topological order.
  - Remove the **tail** in the graph.
  - Repeat…
- Running Time
  - $|V|$ rounds
  - Find a tail: $O(|V|)$
  - Remove a tail & update: $O(|V|)$
  - Total: $\boldsymbol{O(|V|^2)}$

Can we do better?

Improve it by DFS

- DFS tree for a DAG

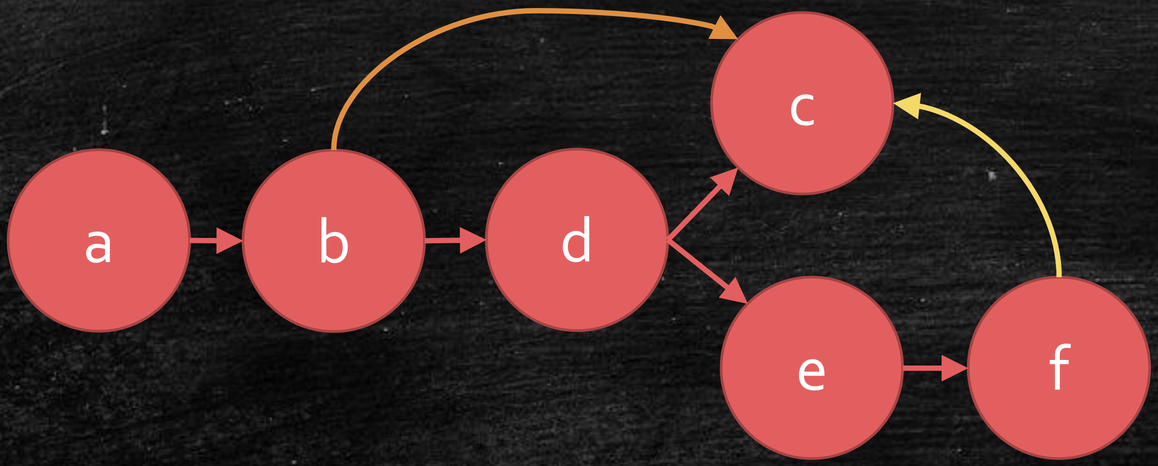


Function explore(v)
  $marked[v] \leftarrow true$
  for each $(v,u) \in E$
    if $marked[u] = false$
      explore(u)

- Kind of edges
  - Tree edges
  - Forward edges
  - Back edges
  - Cross edges

Improve it by DFS

- Observation
  - We do not have back edges in DAG.

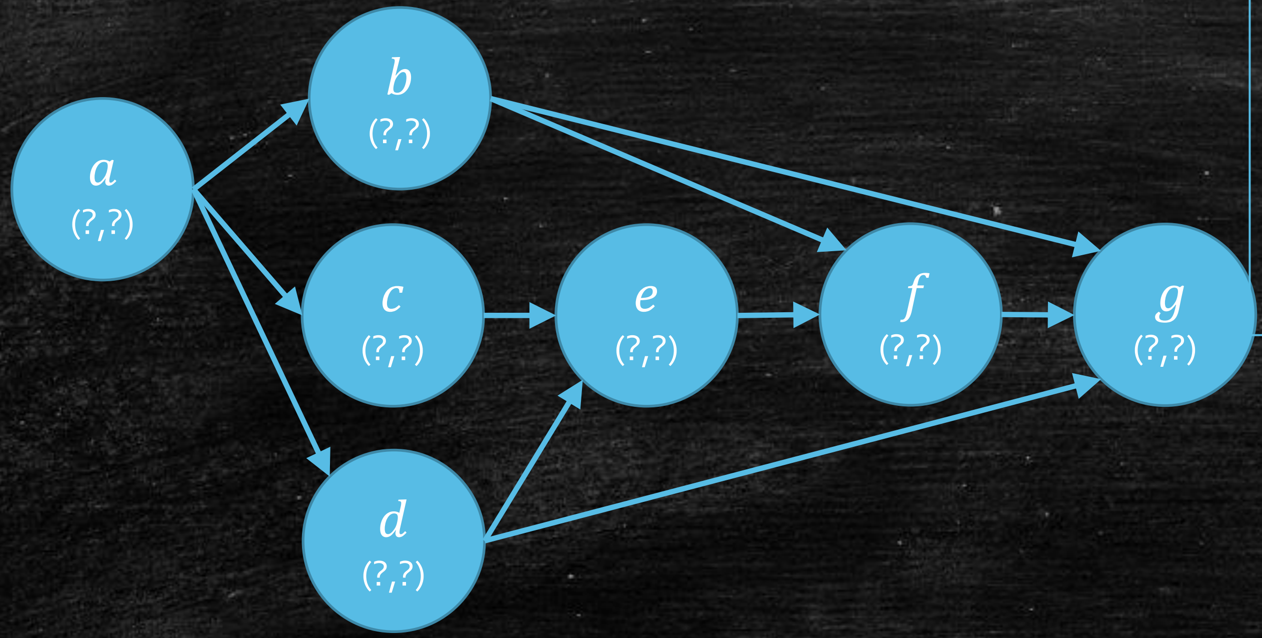


Function explore(v)
  $marked[v] \leftarrow true$
  for each $(v, u) \in E$
    if $marked[u] = false$
      explore(u)

- Kind of edges
  - Tree edges
  - Forward edges
  - ~~Back edges~~
  - Cross edges

Topological Ordering by DFS

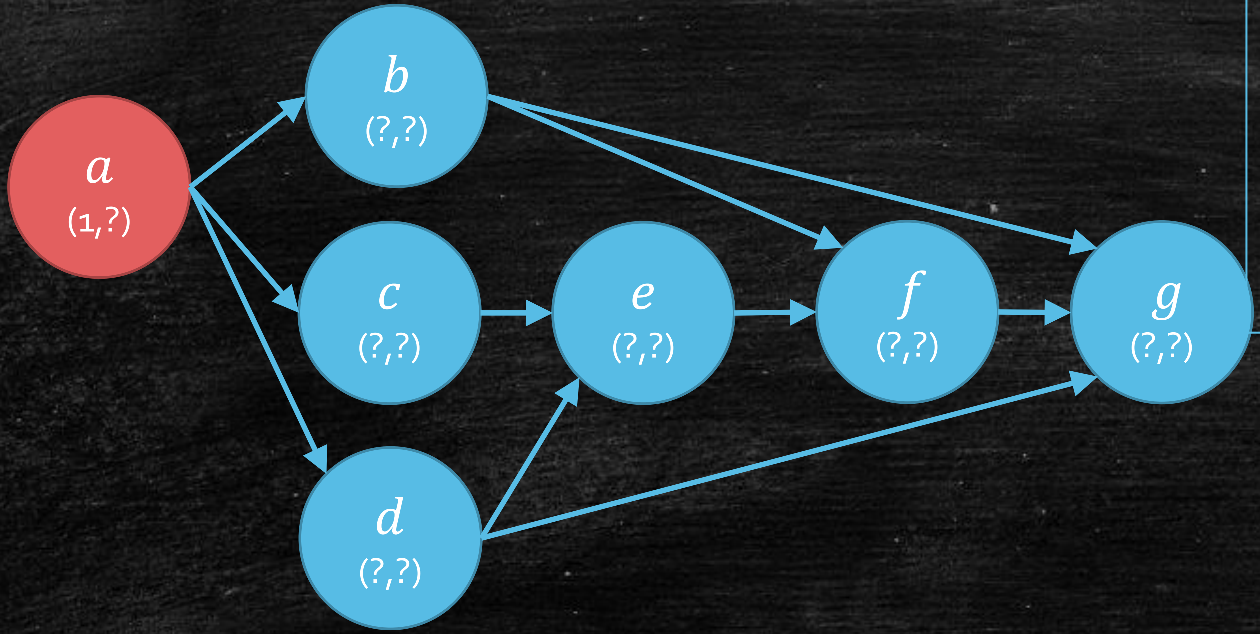
- Run DFS first!
- Record the start time and finish time.



$time \leftarrow 0$
Function explore(v)
    $start[v] \leftarrow time$
    $time++$
    $marked[v] \leftarrow true$
    for each $(v,u) \in E$
        if $marked[u] = false$
            explore(u)
    $finish[v] \leftarrow time$
    $time++$

Topological Ordering by DFS

- Run DFS first!
- Record the start time and finish time.

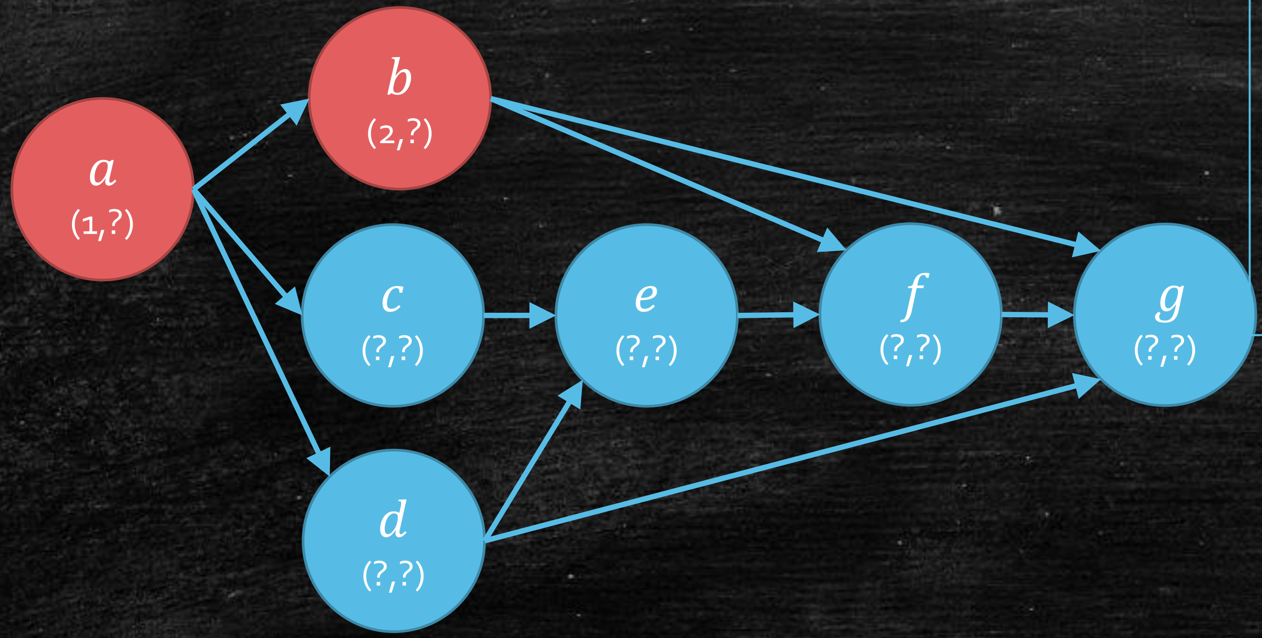


$time \leftarrow 0$

Function explore(v)
  $start[v] \leftarrow time$
  $time++$
  $marked[v] \leftarrow true$
  for each $(v, u) \in E$
    if $marked[u] = false$
      explore(u)
  $finish[v] \leftarrow time$
  $time++$

Topological Ordering by DFS

- Run DFS first!
- Record the start time and finish time.



$time \leftarrow 0$

Function explore(v)

  $start[v] \leftarrow time$

  $time++$

  $marked[v] \leftarrow true$

  for each $(v, u) \in E$

    if $marked[u] = false$

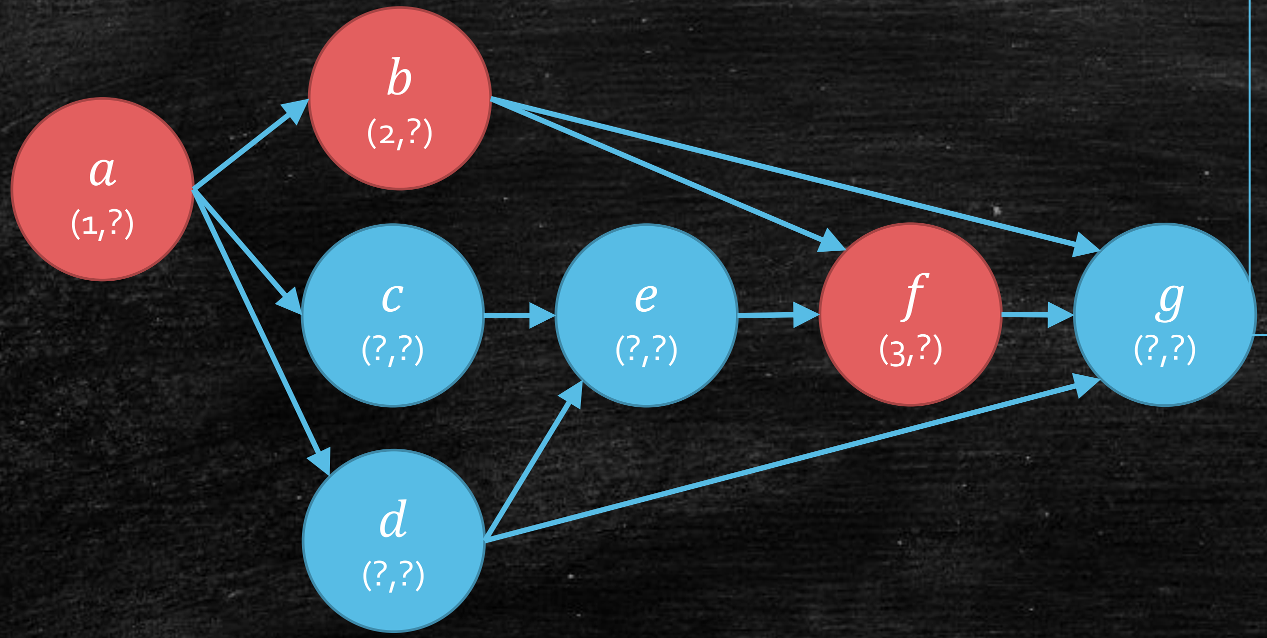
      explore(u)

  $finish[v] \leftarrow time$

  $time++$

Topological Ordering by DFS

- Run DFS first!
- Record the start time and finish time.

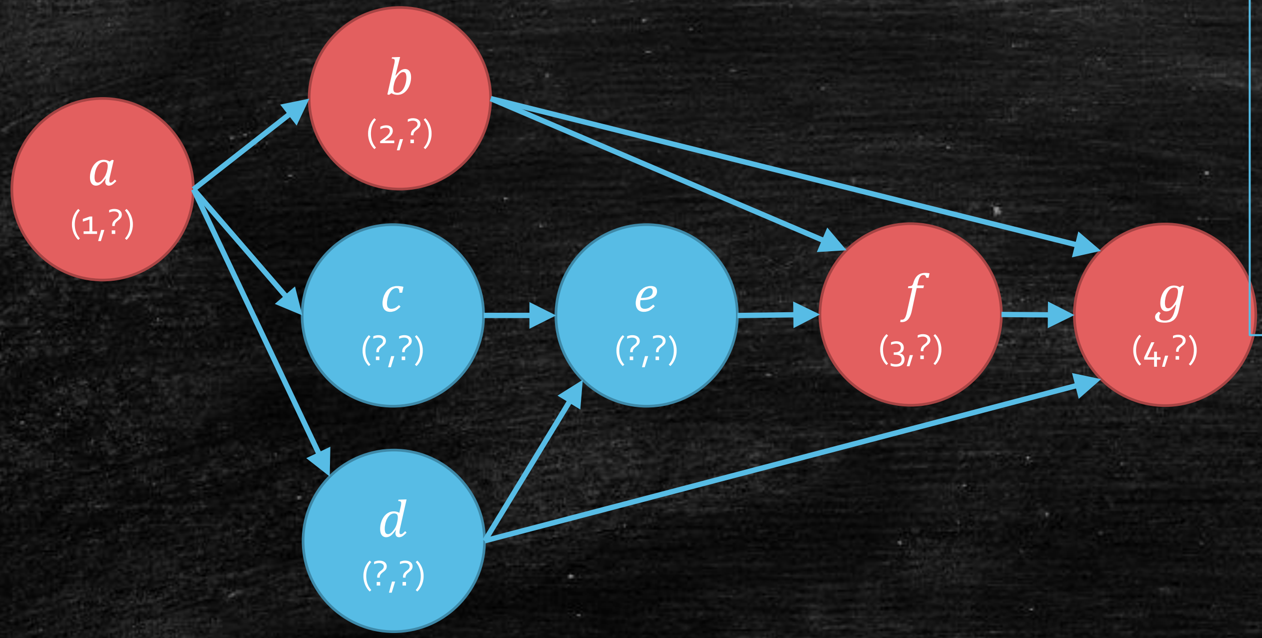


$time \leftarrow 0$

Function explore(v)
- $start[v] \leftarrow time$
- $time++$
- $marked[v] \leftarrow true$
- **for each** $(v, u) \in E$
  - **if** $marked[u] = false$
    - explore(u)
- $finish[v] \leftarrow time$
- $time++$

Topological Ordering by DFS

- Run DFS first!
- Record the start time and finish time.



$time \leftarrow 0$

Function explore(v)

    $start[v] \leftarrow time$

    $time++$

    $marked[v] \leftarrow true$

    for each $(v, u) \in E$

        if $marked[u] = false$

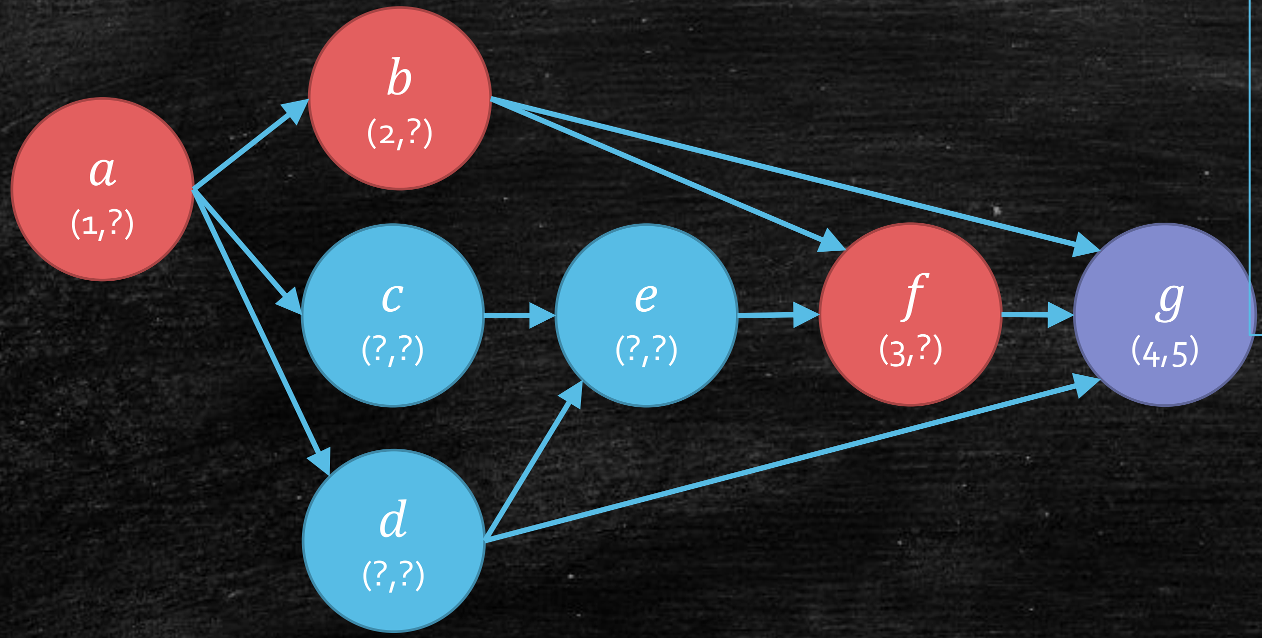
            explore(u)

    $finish[v] \leftarrow time$

    $time++$

Topological Ordering by DFS

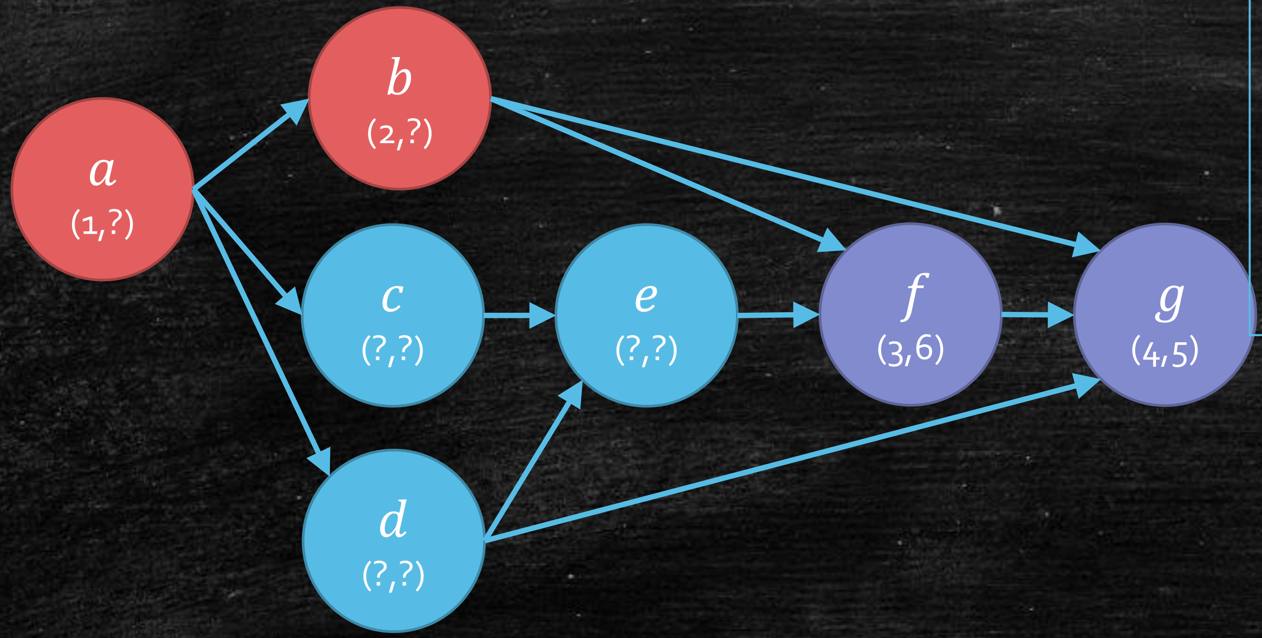
- Run DFS first!
- Record the start time and finish time.



$time \leftarrow 0$
Function explore(v)
    $start[v] \leftarrow time$
    $time++$
    $marked[v] \leftarrow true$
    for each $(v,u) \in E$
        if $marked[u] = false$
            explore(u)
    $finish[v] \leftarrow time$
    $time++$

Topological Ordering by DFS

- Run DFS first!
- Record the start time and finish time.



$time \leftarrow 0$

Function explore(v)

  $start[v] \leftarrow time$

  $time++$

  $marked[v] \leftarrow true$

  for each $(v,u) \in E$

    if $marked[u] = false$

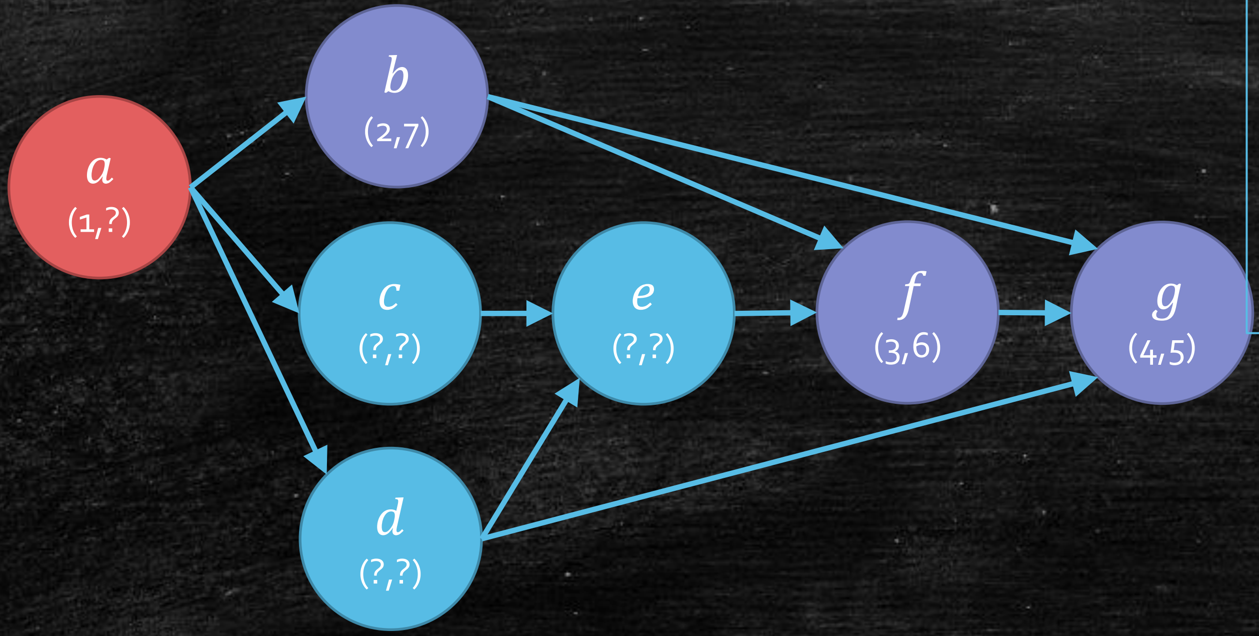
      explore(u)

  $finish[v] \leftarrow time$

  $time++$

Topological Ordering by DFS

- Run DFS first!
- Record the start time and finish time.



$time \leftarrow 0$

Function explore(v)

  $start[v] \leftarrow time$

  $time++$

  $marked[v] \leftarrow true$

  for each $(v, u) \in E$

    if $marked[u] = false$

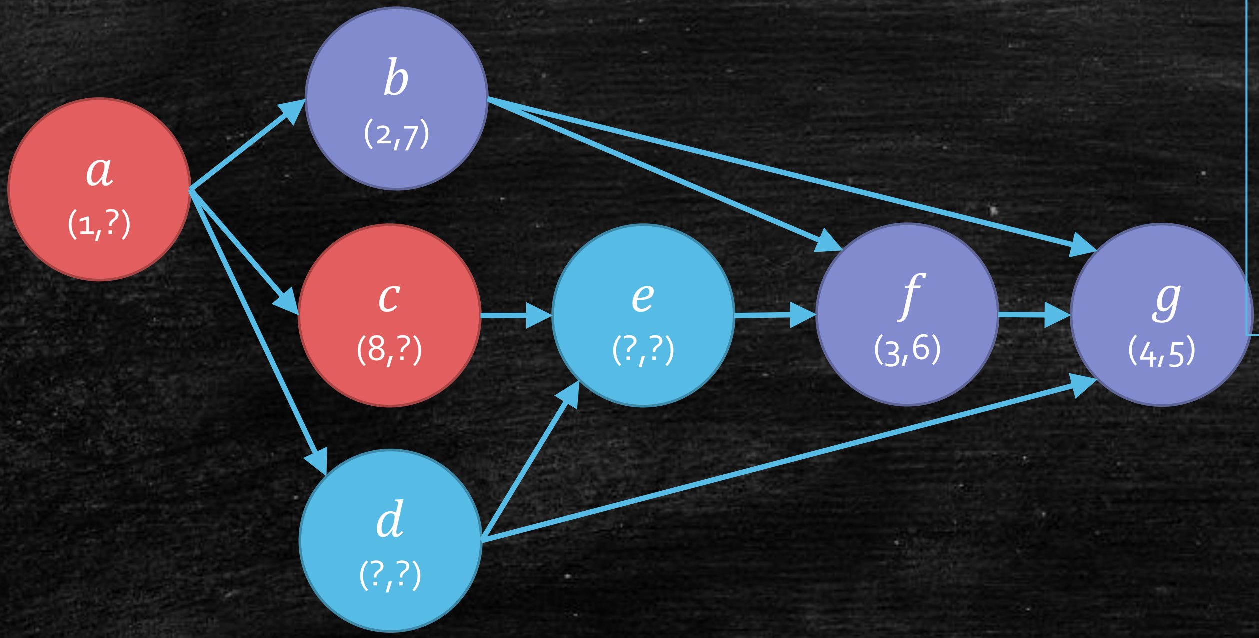
      explore(u)

  $finish[v] \leftarrow time$

  $time++$

Topological Ordering by DFS

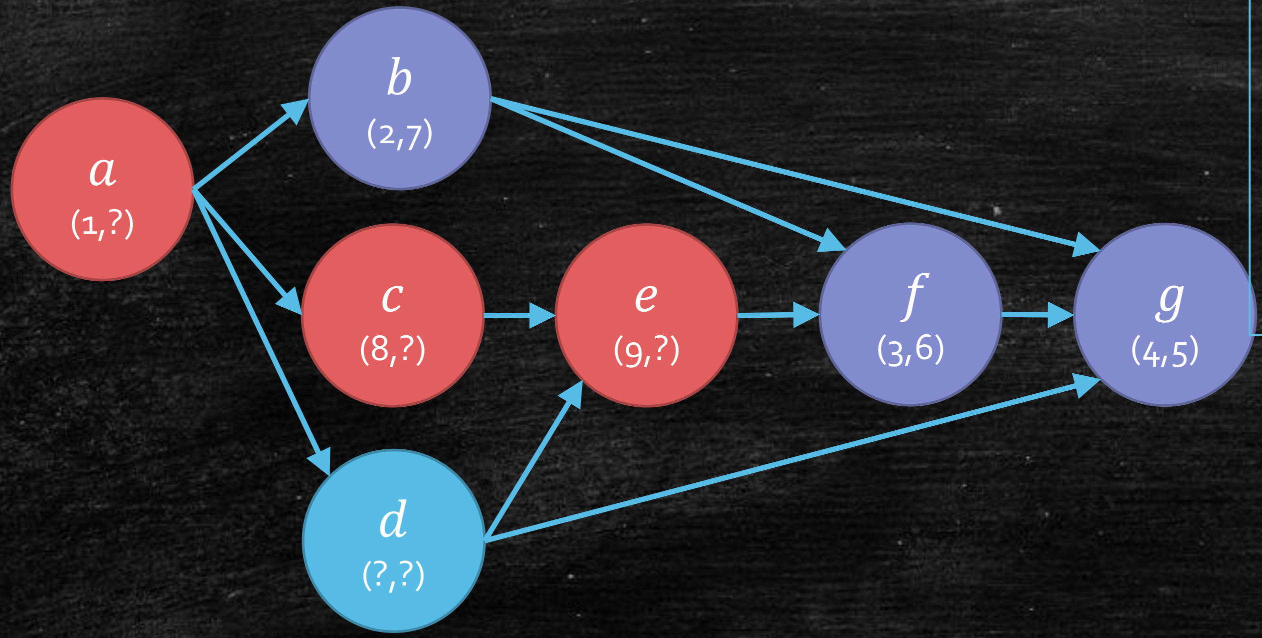
- Run DFS first!
- Record the start time and finish time.



$time \leftarrow 0$
Function explore(v)
  $start[v] \leftarrow time$
  $time++$
  $marked[v] \leftarrow true$
  for each $(v,u) \in E$
    if $marked[u] = false$
      explore(u)
  $finish[v] \leftarrow time$
  $time++$

Topological Ordering by DFS

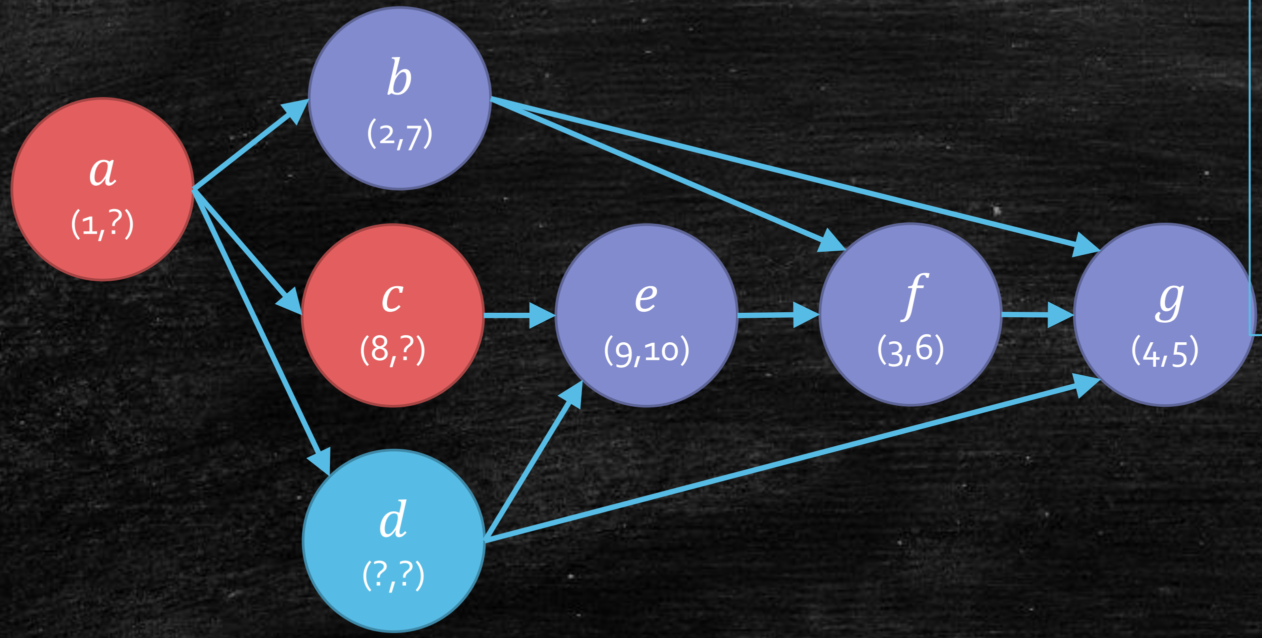
- Run DFS first!
- Record the start time and finish time.



$time \leftarrow 0$
Function explore(v)
    $start[v] \leftarrow time$
    $time++$
    $marked[v] \leftarrow true$
    for each $(v, u) \in E$
        if $marked[u] = false$
            explore(u)
    $finish[v] \leftarrow time$
    $time++$

Topological Ordering by DFS

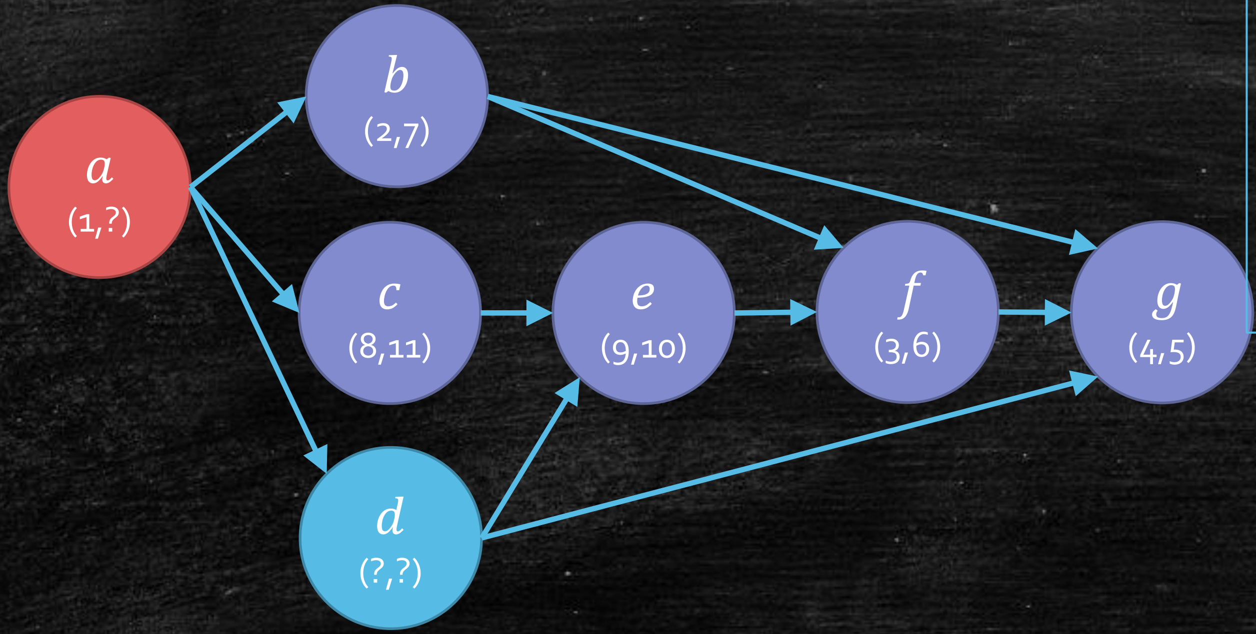
- Run DFS first!
- Record the start time and finish time.



$time \leftarrow 0$
Function explore(v)
  $start[v] \leftarrow time$
  $time++$
  $marked[v] \leftarrow true$
  for each $(v, u) \in E$
    if $marked[u] = false$
      explore(u)
  $finish[v] \leftarrow time$
  $time++$

Topological Ordering by DFS

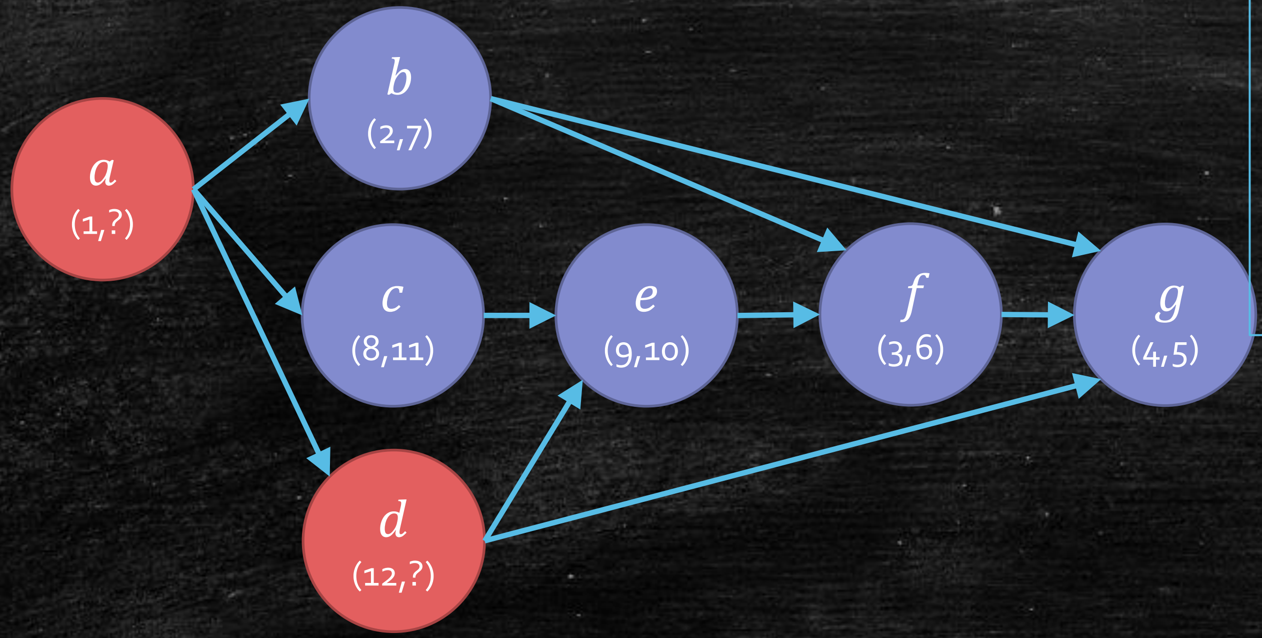
- Run DFS first!
- Record the start time and finish time.



$time \leftarrow 0$
Function explore(v)
 $start[v] \leftarrow time$
 $time++$
 $marked[v] \leftarrow true$
 for each $(v,u) \in E$
  if $marked[u] = false$
   explore(u)
 $finish[v] \leftarrow time$
 $time++$

Topological Ordering by DFS

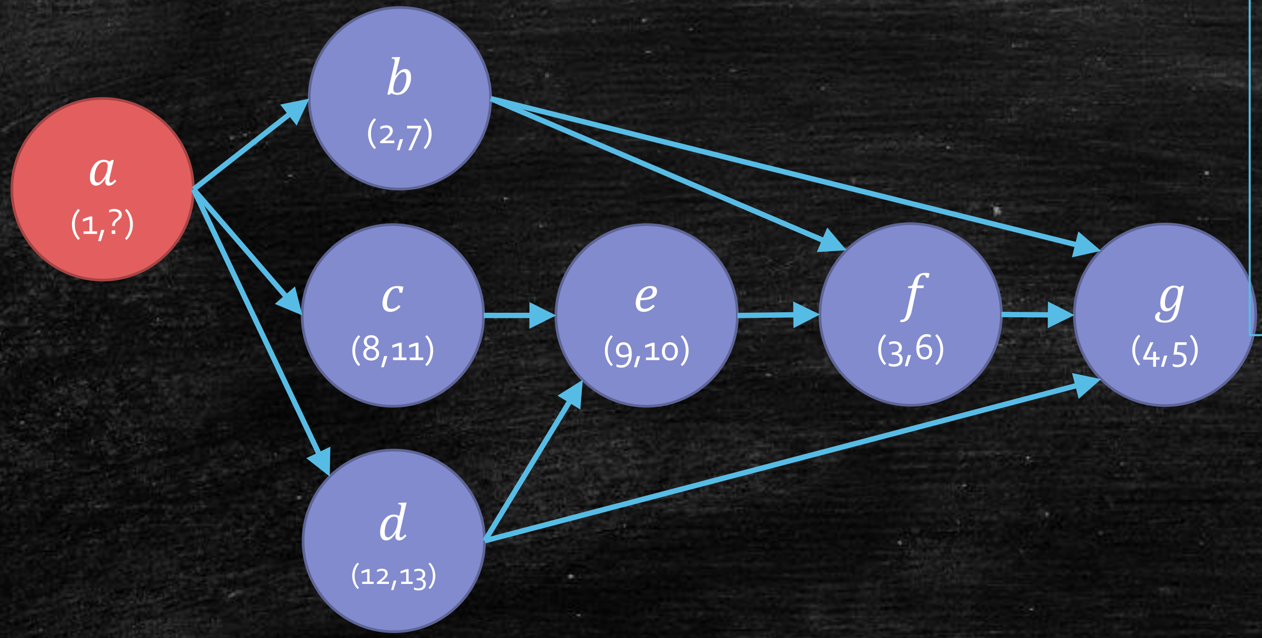
- Run DFS first!
- Record the start time and finish time.



$time \leftarrow 0$
Function explore(v)
    $start[v] \leftarrow time$
    $time++$
    $marked[v] \leftarrow true$
    for each $(v, u) \in E$
        if $marked[u] = false$
            explore(u)
    $finish[v] \leftarrow time$
    $time++$

Topological Ordering by DFS

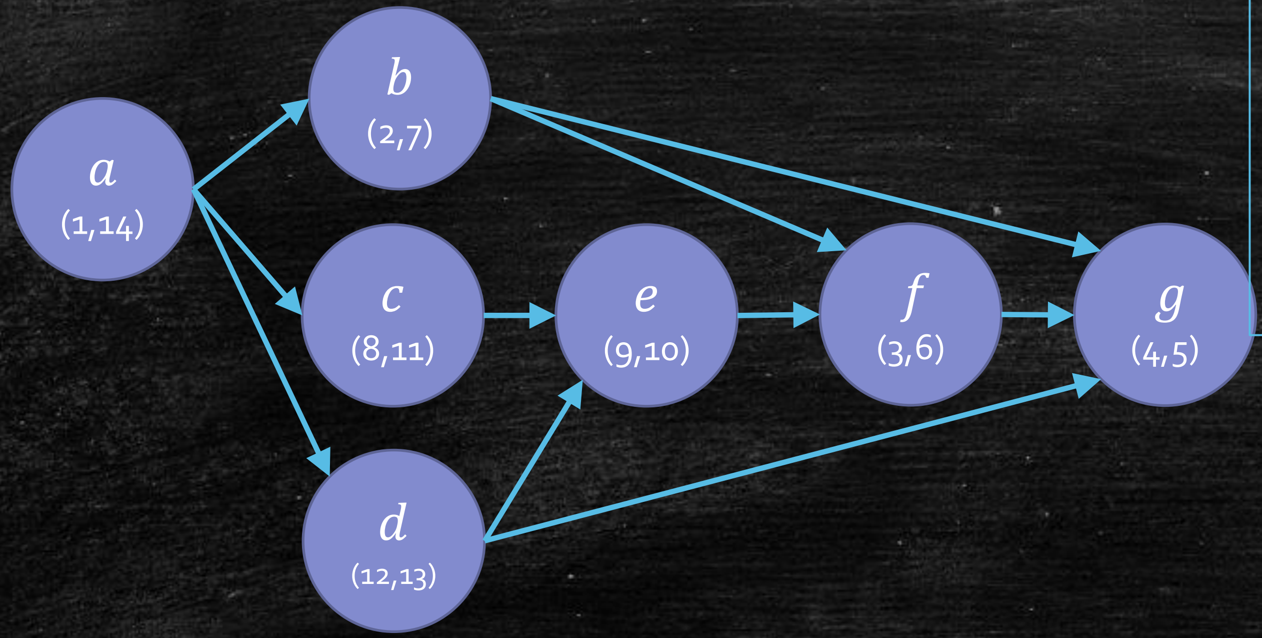
- Run DFS first!
- Record the start time and finish time.



$time \leftarrow 0$
Function explore(v)
    $start[v] \leftarrow time$
    $time++$
    $marked[v] \leftarrow true$
    for each $(v,u) \in E$
        if $marked[u] = false$
            explore(u)
    $finish[v] \leftarrow time$
    $time++$

Topological Ordering by DFS

- Run DFS first!
- Record the start time and finish time.



$time \leftarrow 0$

Function explore(v)

$\quad start[v] \leftarrow time$

$\quad time++$

$\quad marked[v] \leftarrow true$

$\quad$ **for each** $(v,u) \in E$

$\quad\quad$ **if** $marked[u] = false$

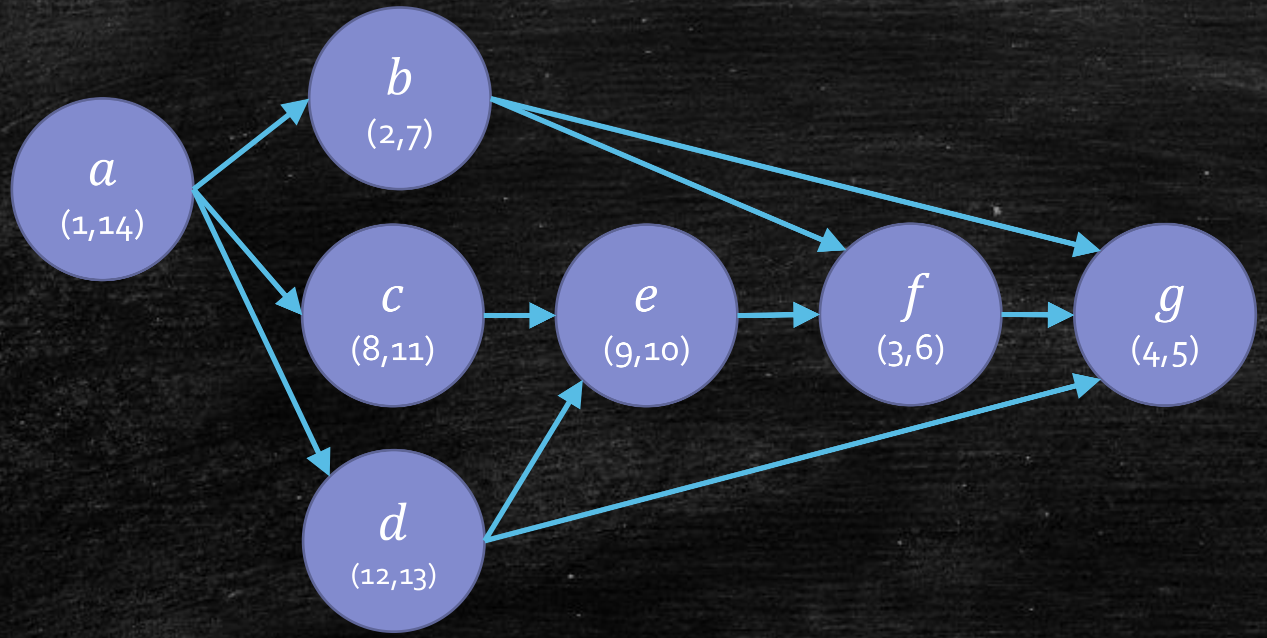
$\quad\quad\quad$ explore(u)

$\quad finish[v] \leftarrow time$

$\quad time++$

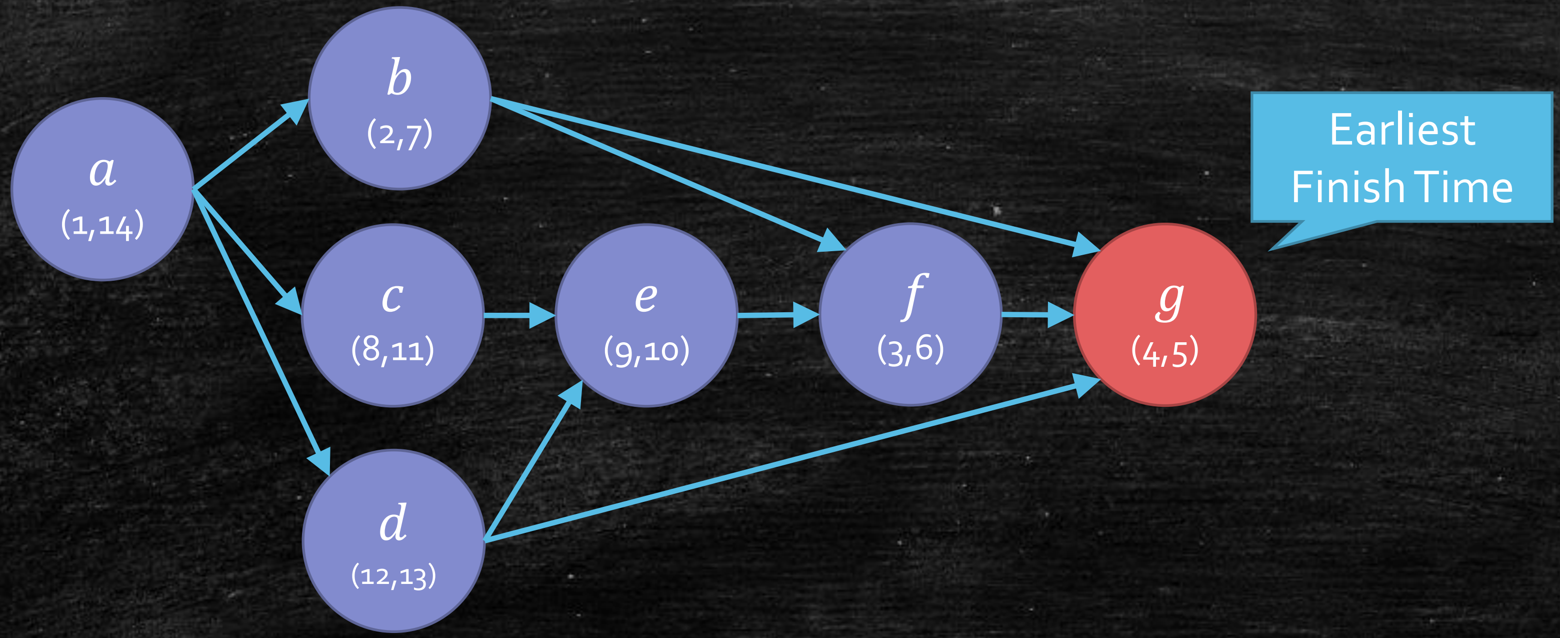
Discussion

- We need repeat finding a **tail.**
- Who must be a **tail** in DFS?



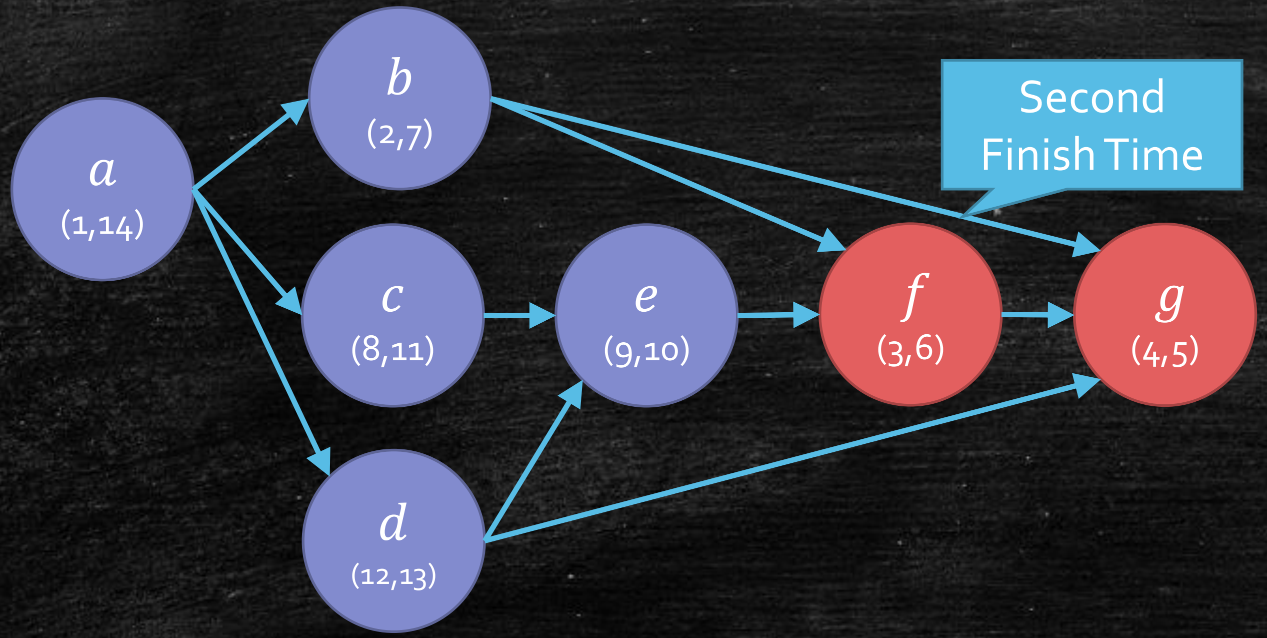
Discussion

- We need repeat finding a **tail.**
- After removing the g, who mut be a **tail**?



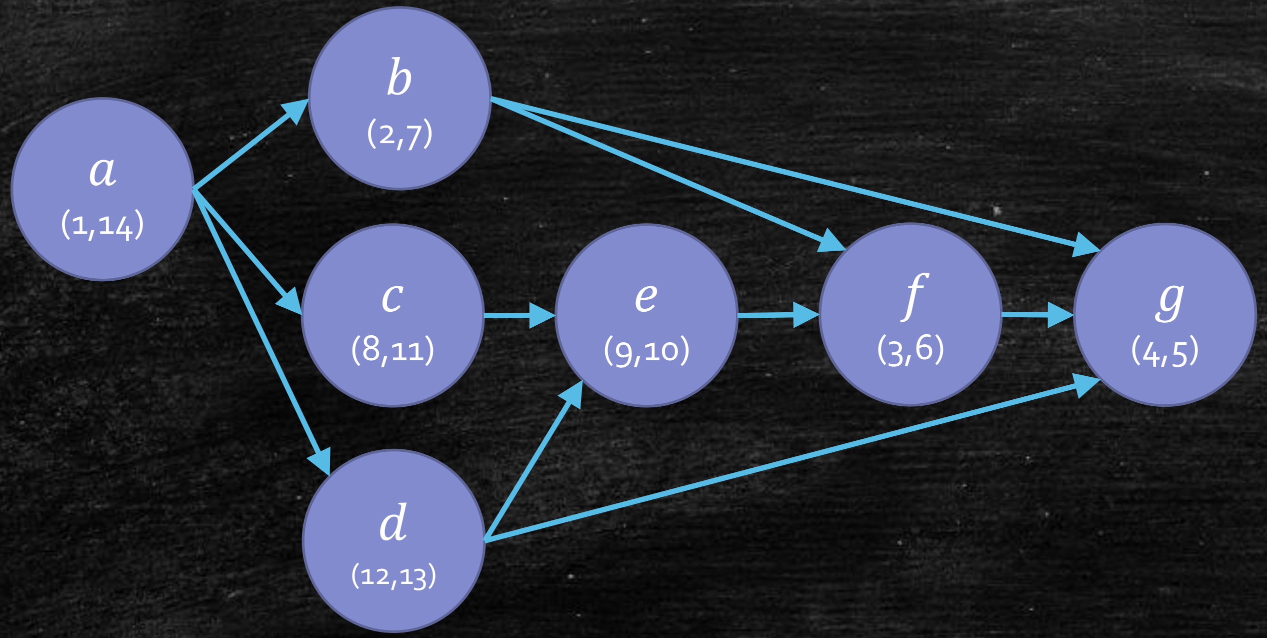
Discussion

- We need repeat finding a **tail.**
- Who must be a **tail** when we do it again?



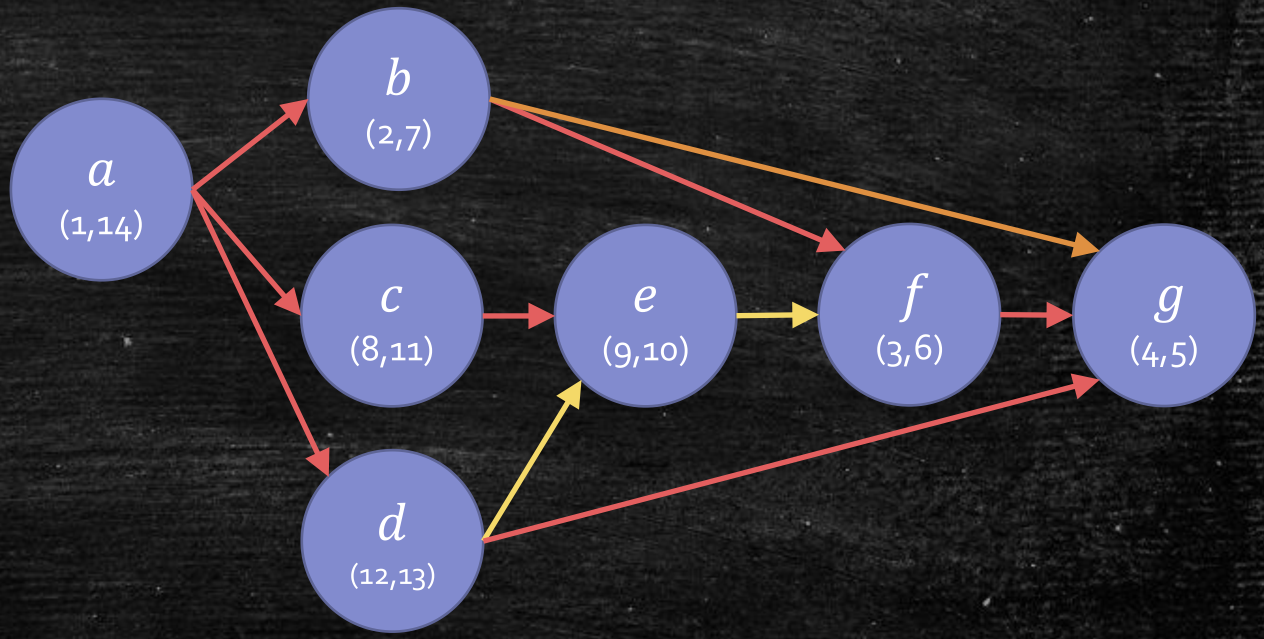
Conjecture

- We can select the vertex with the **earliest** finish time to be the **tail**.
- Algorithm: sort vertices by descending order of finish time.



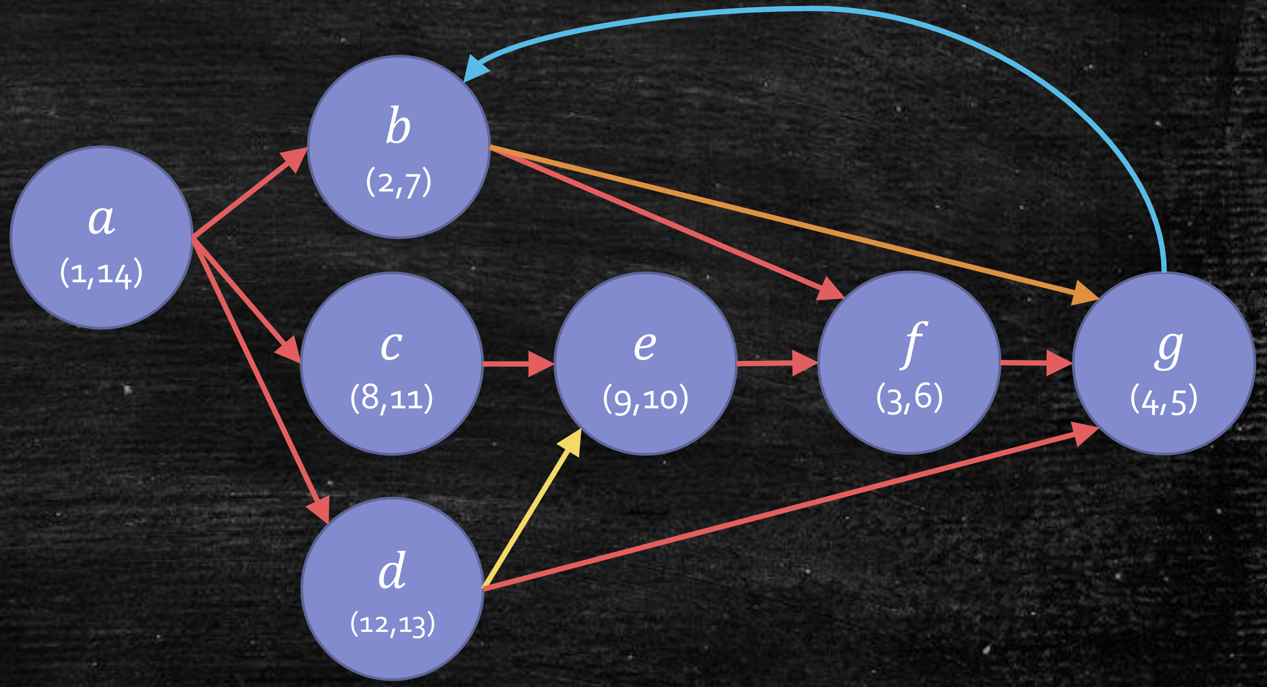
Prove the conjecture

- Claim: no arc (u, v), if $finish[v] > finish[u]$.
- Proof:
  - If (u, v) exists,
  - Can (u, v) be a tree edge?
  - Can (u, v) be a forward edge?
  - Can (u, v) be a cross edge?



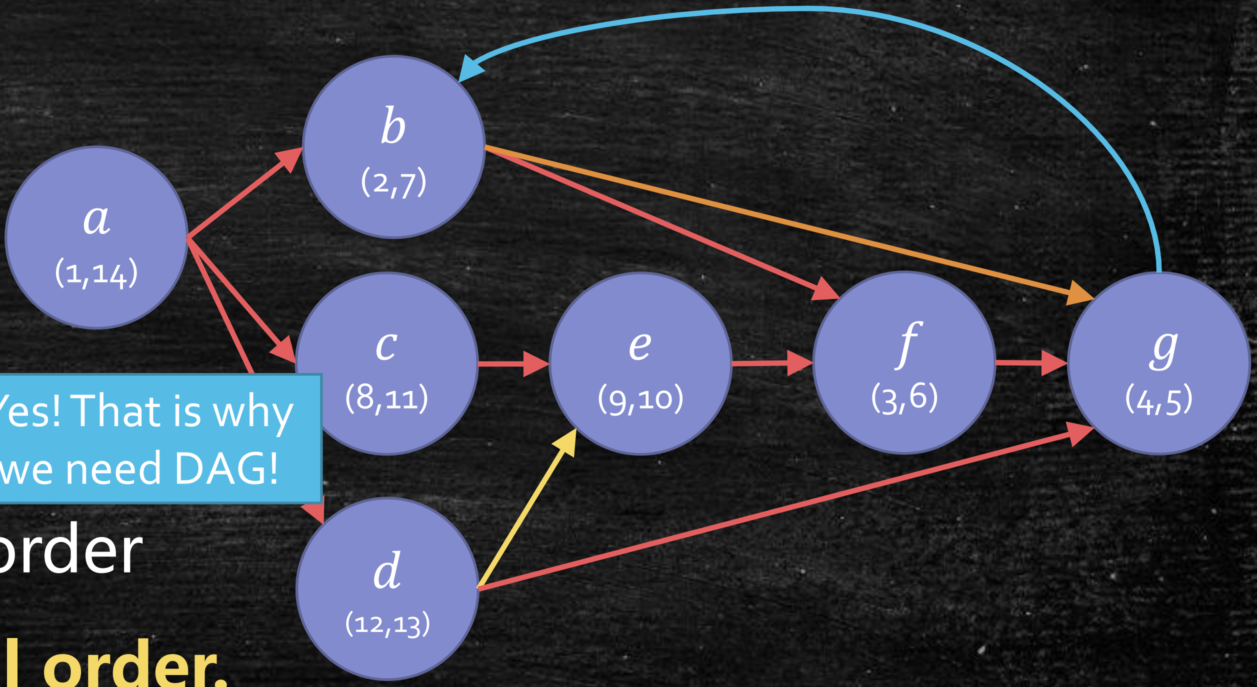
Prove the conjecture

- Claim: no arc (u, v), if $finish[v] > finish[u]$.
- Proof:
  - If (u, v) exists,
  - ~~Can (u, v) be a tree edge?~~
  - ~~Can (u, v) be a forward edge?~~
  - ~~Can (u, v) be a cross edge?~~
  - Can (u, v) be a back edge?



Prove the conjecture

- Claim: no arc (u, v), if $finish[v] > finish[u]$.
- Proof:
  - If (u, v) exists,
  - ~~Can (u, v) be a tree edge?~~
  - ~~Can (u, v) be a forward edge?~~
  - ~~Can (u, v) be a cross edge?~~
  - Can (u, v) be a back edge? – Yes! That is why we need DAG!
- Corollary: the descending order of finish time is a **topological order.**
- Question: running time?



Running Time

- $O(|V|\log|V| + |E| + |V|)$?
  - Run **DFS** with **finish time**
  - **Sort** the finish time
  - Output the topological order

Running Time

- $O(|V|\log|V| + |E| + |V|)$?
  - Run **DFS** with **finish time**
  - **Sort** the finish time
  - Output the topological order
- Smarter implementation
  - During the **DFS**,
  - When we **finish** a vertex,
  - **Append** it to the topological order!
  - It follows the order of finish time!
- $\boldsymbol{O}(|\boldsymbol{V}| + |\boldsymbol{E}|)$?

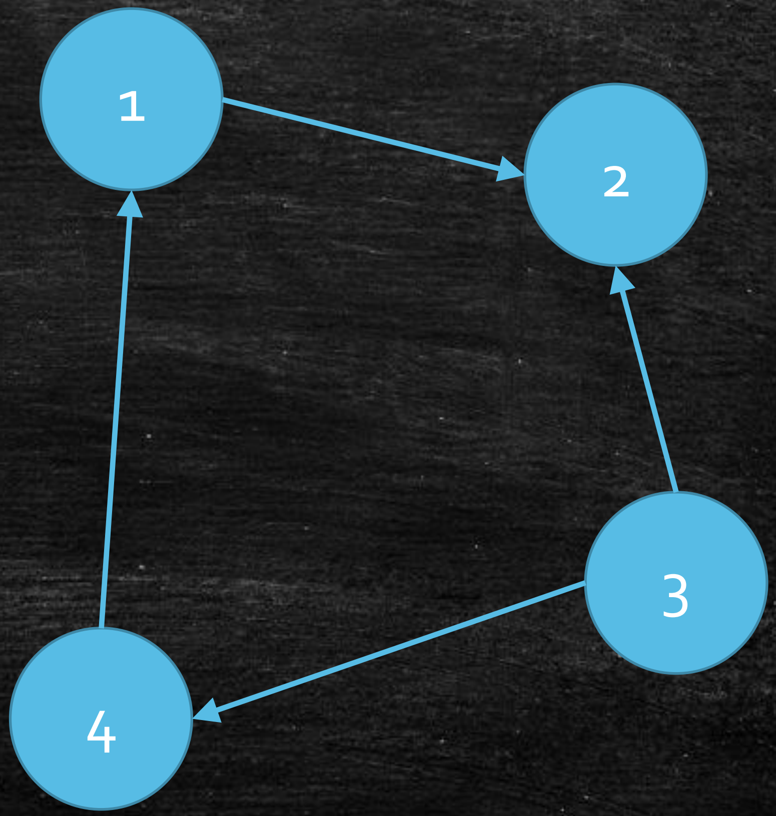
Connectivity in Directed Graphs

Recall

- **Connect Component(CC)** in undirected graphs
- **DFS** can directly find **CC** in undirected graphs.
- How to define **CC** in **directed** graphs?

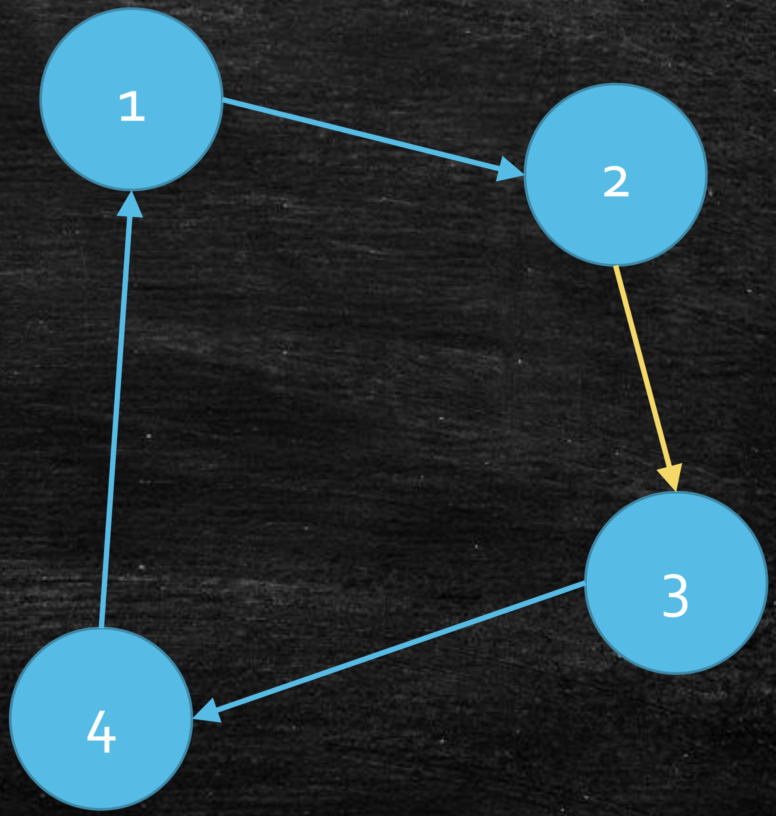
Connect Components in Directed Graphs

- Is the component connected?
- It is weakly connected
  - A weak connected component
  - Undirected version is connected
- How to make it **stronger?**
- What do we mean **strong?**
  - Each pair (u, v)
  - u can reach v, v can reach u.



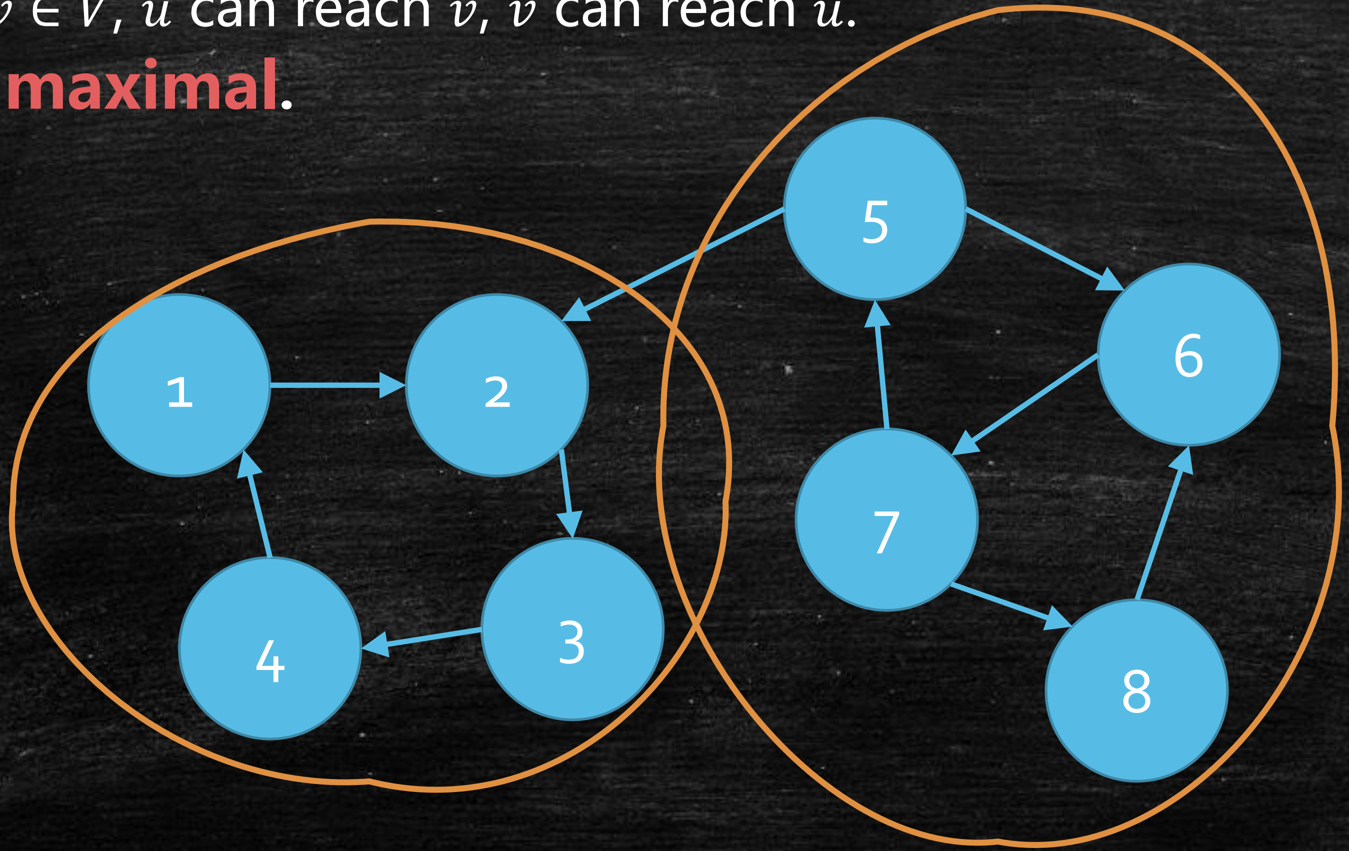
Connect Components in Directed Graphs

- Is the component connected?
- It is weakly connected
  - A weak connected component
  - Undirected version is connected
- How to make it **stronger?**
- What do we mean **strong?**
  - Each pair (u, v)
  - u can reach v, v can reach u.
  - Called **strongly connected**



Strongly Connected Component (SCC)

- $C \subset V$ is a SCC
  - $\forall u, v \in V$, u can reach v, v can reach u.
  - It is **maximal**.



Do SCCs Partition a graph?

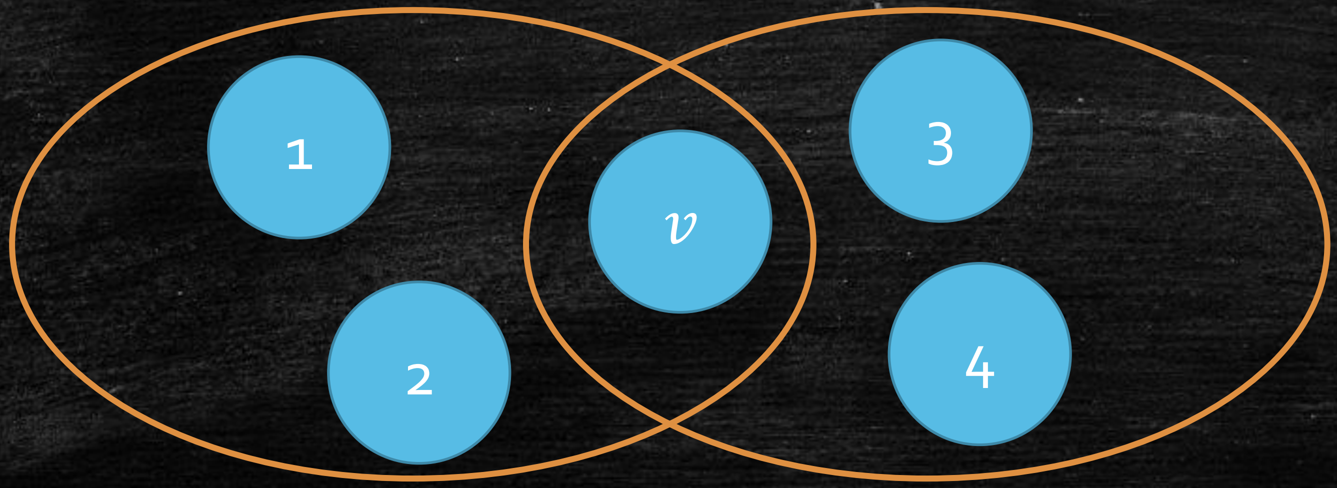
CCs can partition a graph.

Claim

- **Want to prove**
  - Let $C_1, C_2, C_3, \dots, C_m$ be all SCCs of $G(V, E)$,
  - $C_1 \cup C_2 \cup C_3 \cup \dots \cup C_m = V$.
  - $\forall C_i \neq C_j, C_i \cap C_j = \emptyset$.
- **Claim:**
  - For each vertex v
  - There exists and only exists one C_i that contains v.

Proof

- →: there exists a C_i contains v.
  - $\{v\}$ is strongly connected.
  - Keep explore $\{v\}$ until it is maximal.
  - It becomes a connected component.
- ←: only one C_i contains v.

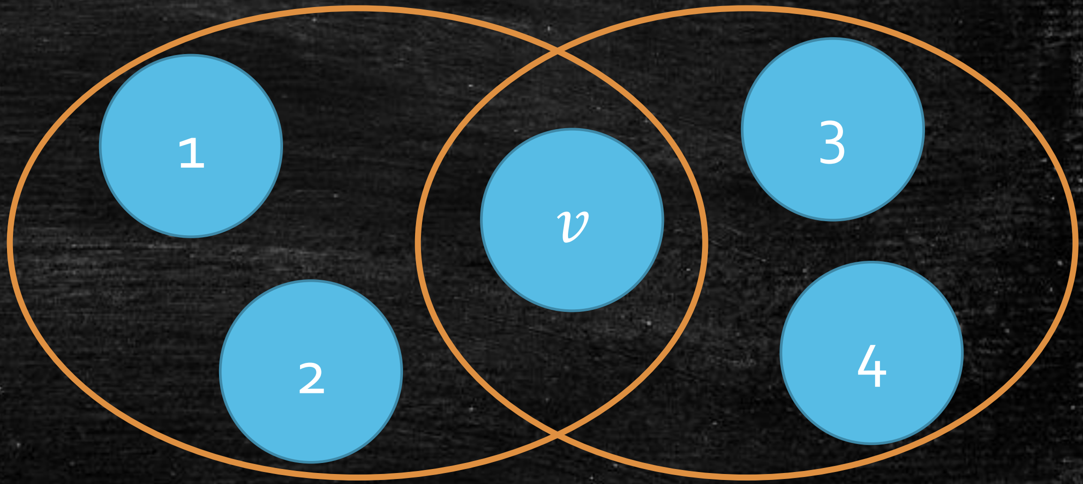


One more property of strongly connected

- **Transitivity**
  - If a and b are strongly connected, and b and c are strongly connected, then a and c are strongly connected.
- Proof
  - We have path $a \to b$ and $b \to a$.
  - We have path $b \to c$ and $c \to b$.
  - So, we have path $a \to b \to c$.
  - So, we have path $c \to b \to a$.
- Corollary
  - If a set C is strongly connected and b is strongly connected to $a \in C$, then $C \cup \{a\}$ is strongly connected.

Proof

- $\rightarrow$: there exists a C_i contains v.
  - $\{v\}$ is strongly connected.
  - Keep explore $\{v\}$ until it is maximal.
  - It becomes a connected component.
- $\leftarrow$: only one C_i contains v.
  - $\{1,2,v\}$ is strongly connected
  - $\{v,3,4\}$ is strongly connected
  - $\{1,2,3,4,v\}$ is strongly connected
  - **Contradiction!**

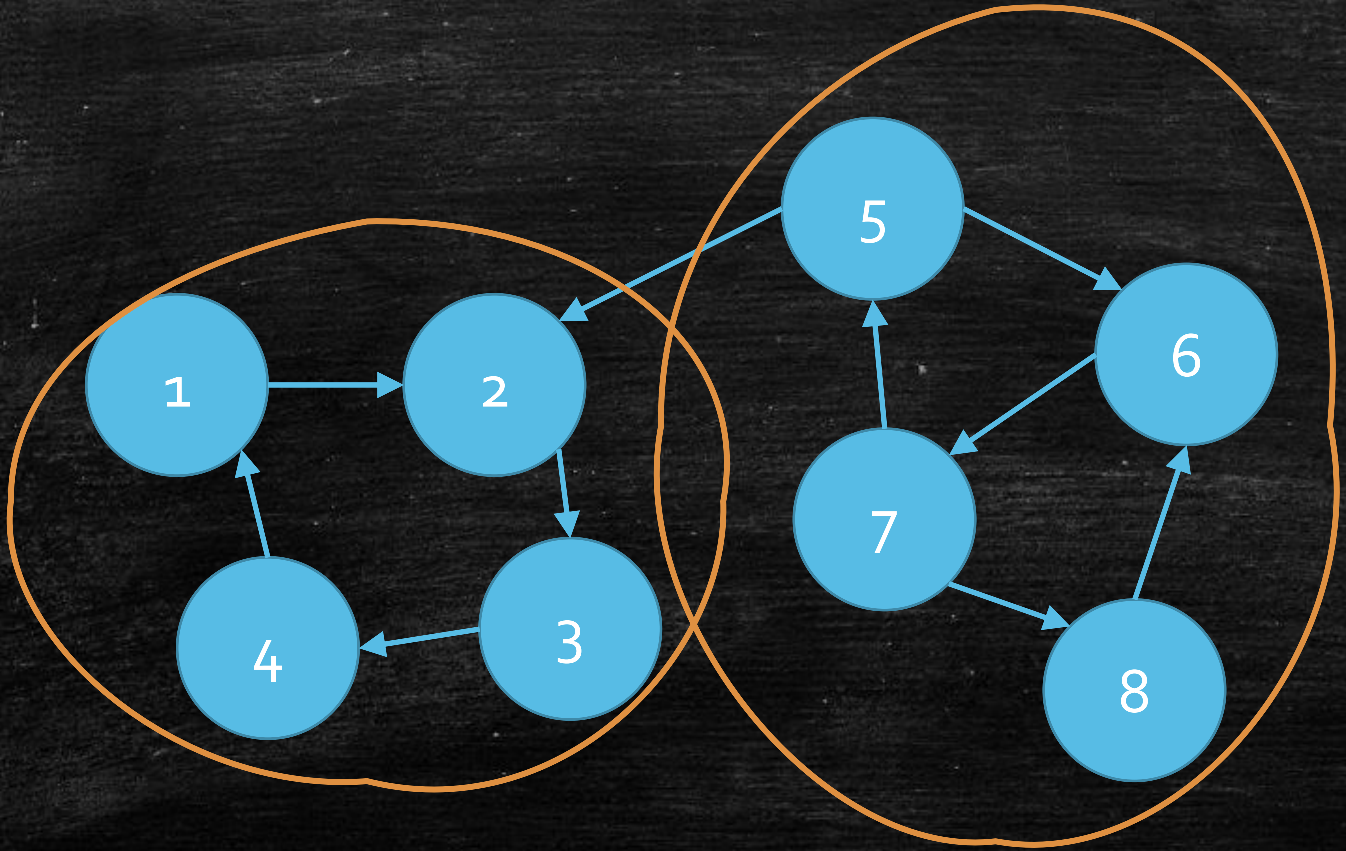


The set of SCCs forms a Partition of V!

Can we use DFS to find SCCs?

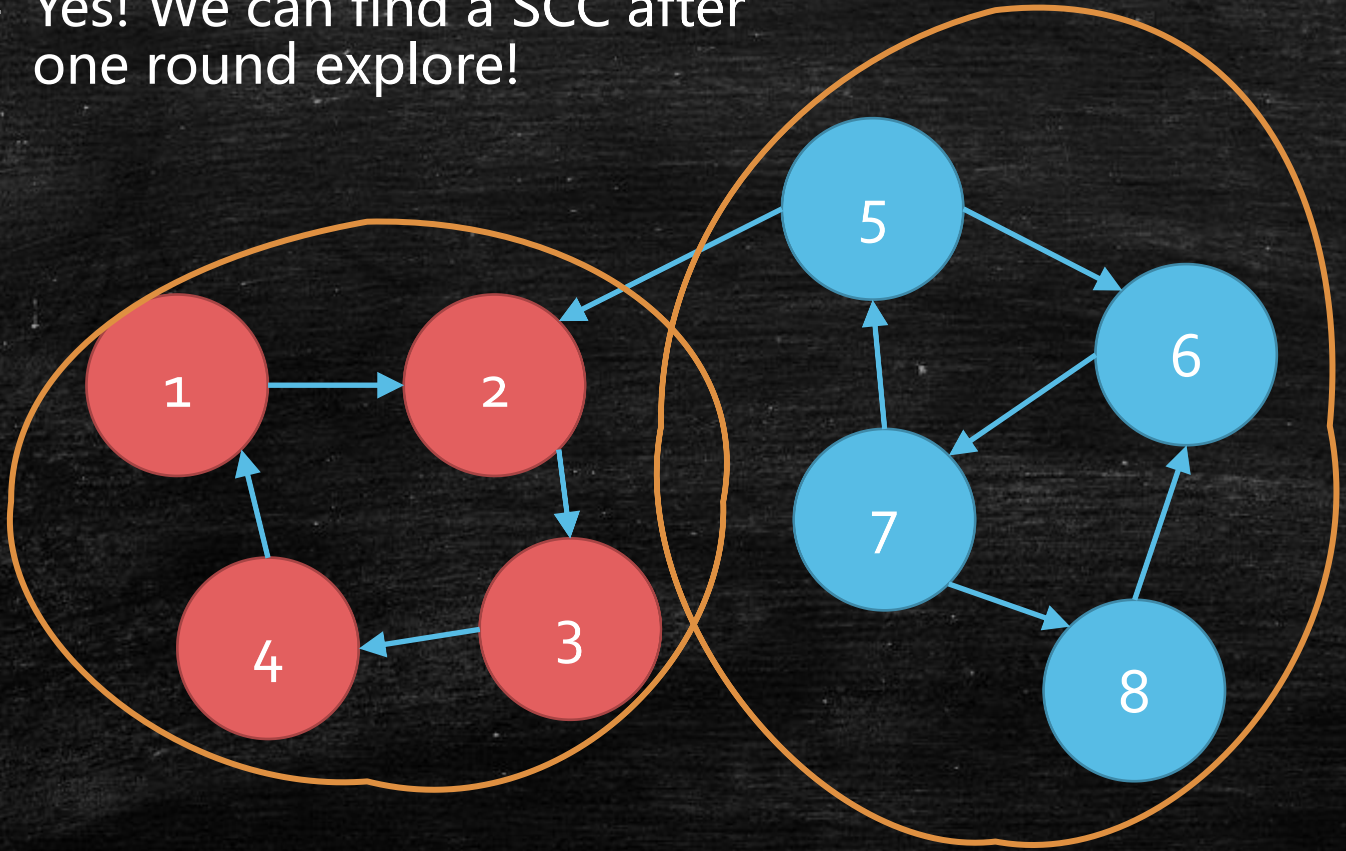
A Simple Attempt

- Start DFS from vertex 1.



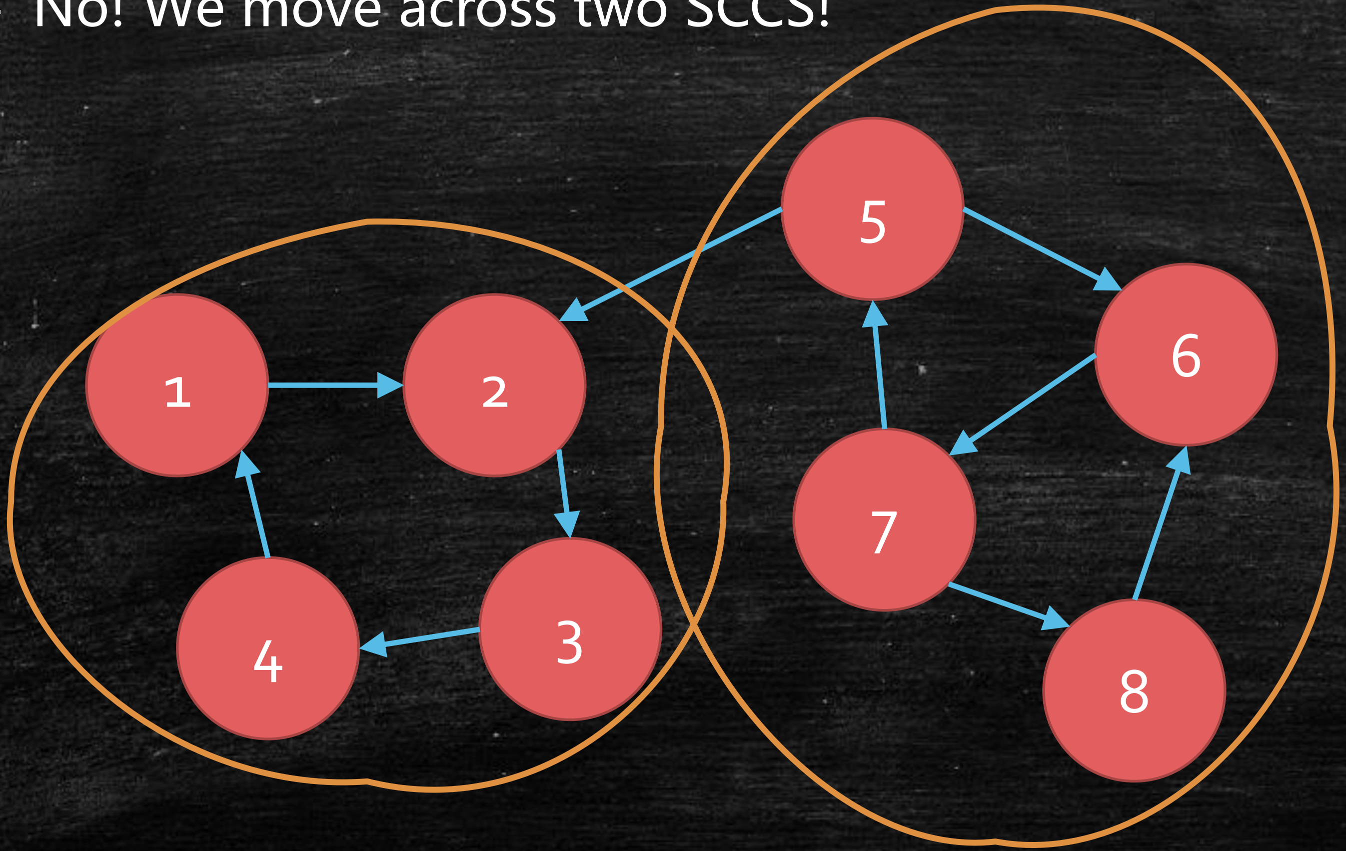
A Simple Attempt

- Start DFS from vertex 1.
  - Yes! We can find a SCC after one round explore!



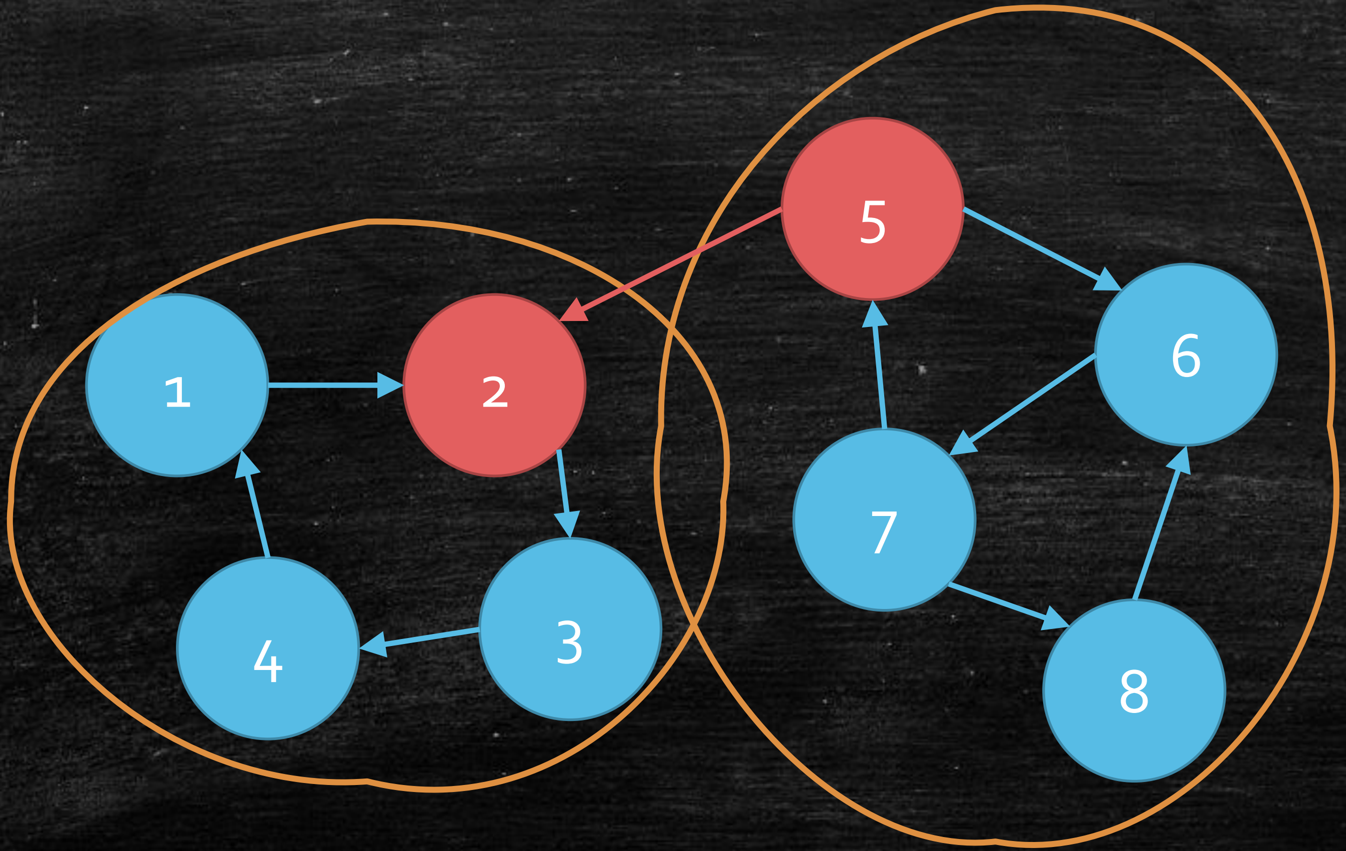
A Simple Attempt

- Start DFS from vertex 5.
  - No! We move across two SCCS!



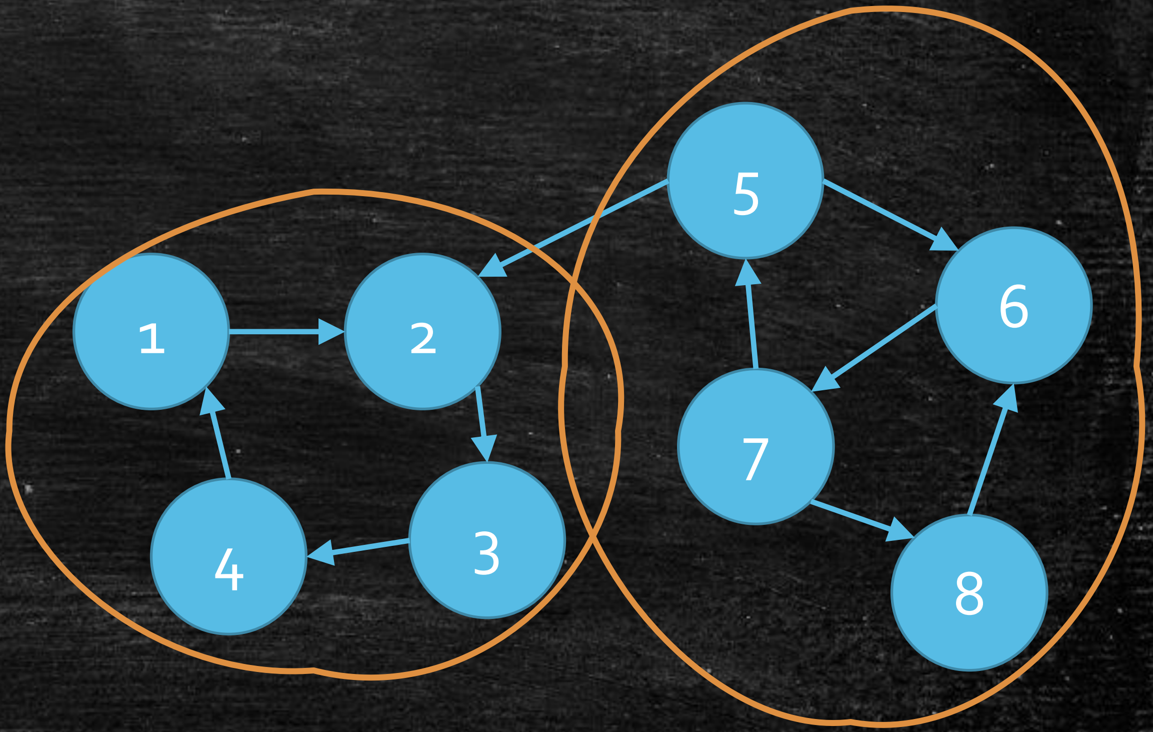
What is the trouble?

- Trouble: the outgoing edges



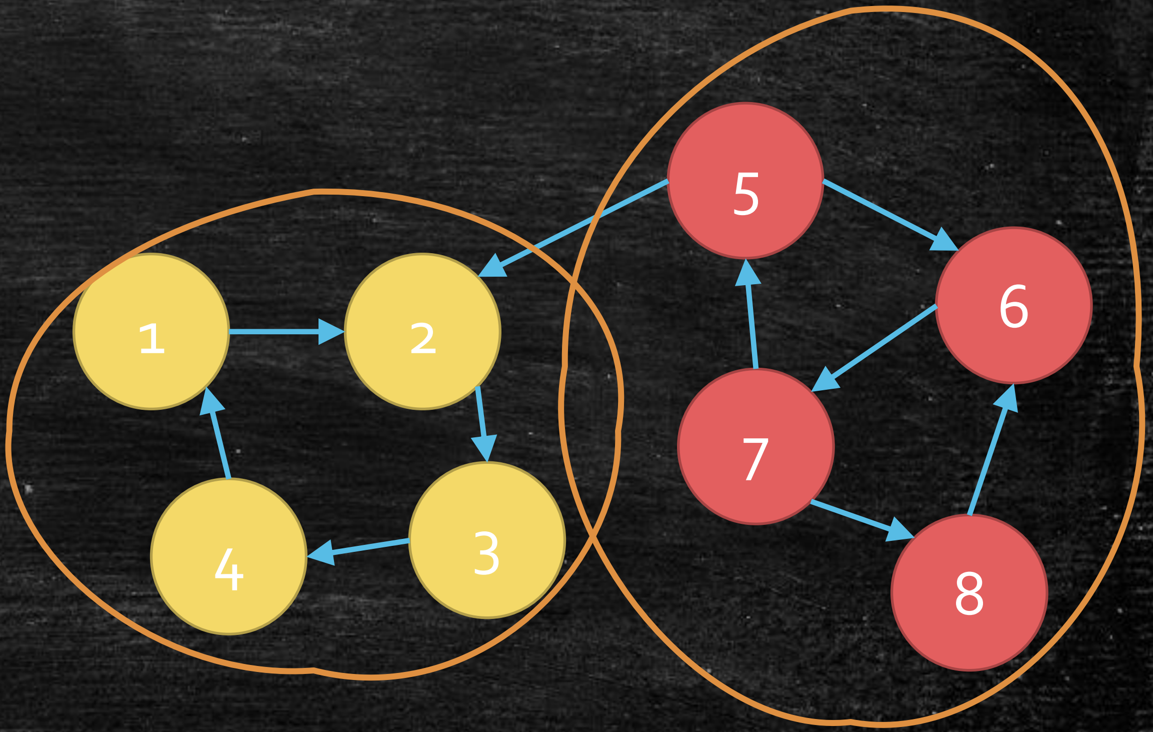
Question: can we handle it?

- Why start from 5 is bad?
- Why start from 1 is good?
- What kind of start points are good?



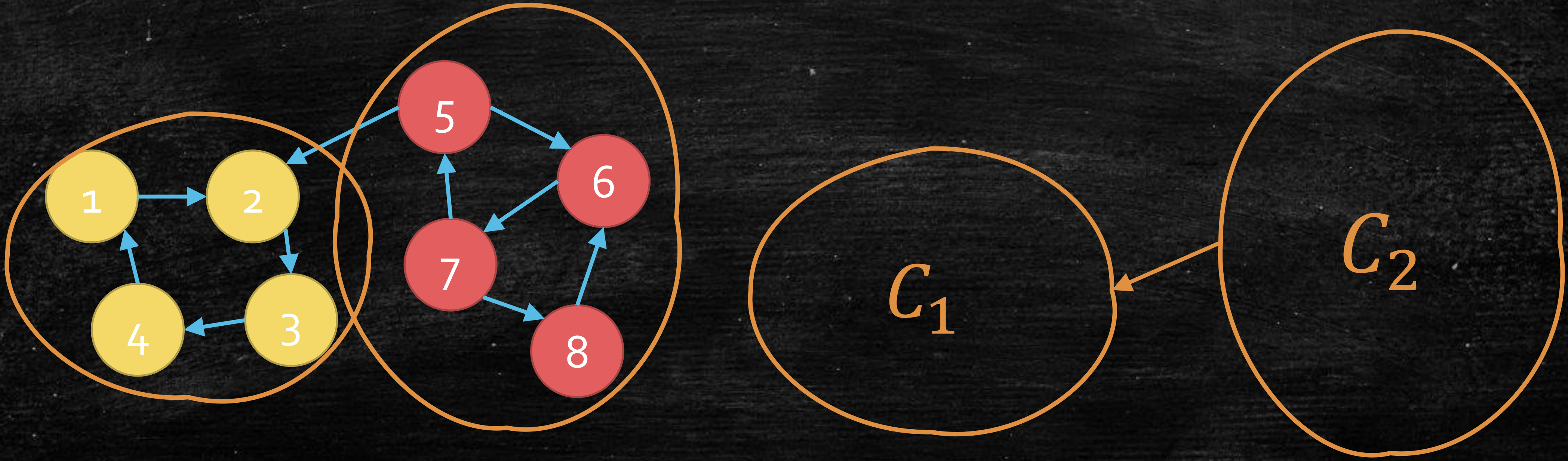
Question: can we handle it?

- Why start from 5 is bad?
- Why start from 1 is good?
- What kind of start points are good?
- It is good if we are in a SCC without outgoing edges.



Does such SCC exist?

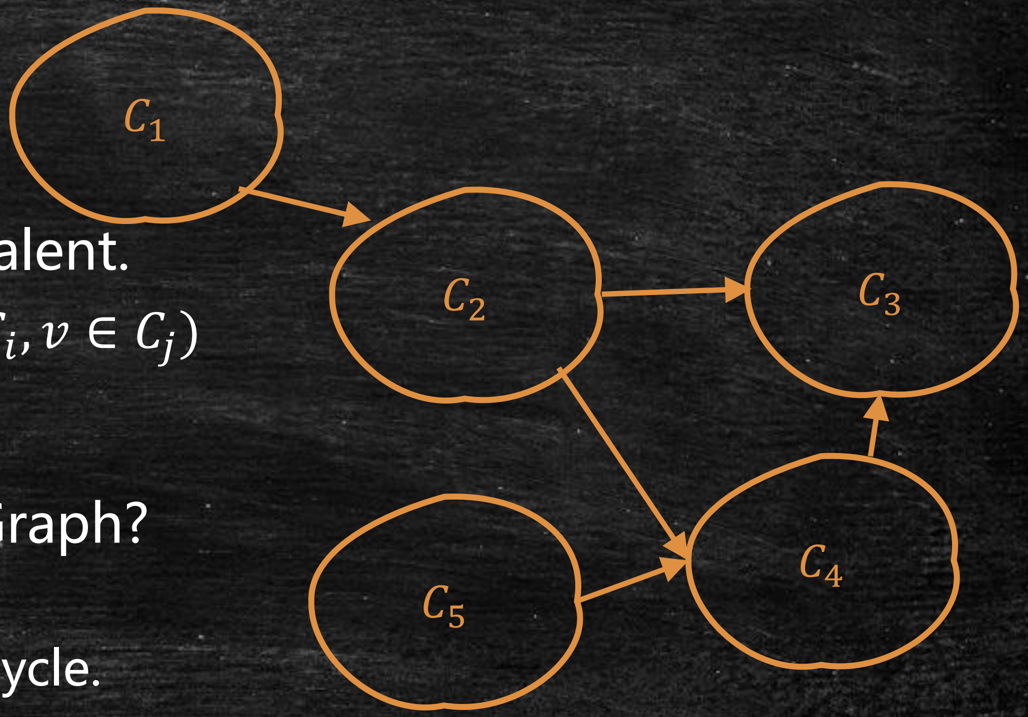
- Move to a big picture
  - View SCCs as Super Nodes.
  - Vertices inside are somehow equivalent.
  - (C_i, C_j) exists ←→ (u, v) exists $(u \in C_i, v \in C_j)$



Thinking: Why is good if we start from a vertex inside the tail SCC?

Does such SCC exist?

- Move to a big picture
  - Let SCCs be Super Nodes.
  - Vertices inside are somehow equivalent.
  - (C_i, C_j) exists <-> (u, v) exists $(u \in C_i, v \in C_j)$
- Questions
  - Can we find a **tail** SCC in the SCC Graph?
  - If we can not, what happens?
    - There is a cycle $C_1, C_2, \ldots, C_m$ forms a cycle.
    - $C_1 \cup C_2 \ldots \cup C_m$ is **strongly connected**.
    - They should be one SCC! Contradiction!
  - **Corollary: the SCC Graph is a DAG!**



A Better Attempt

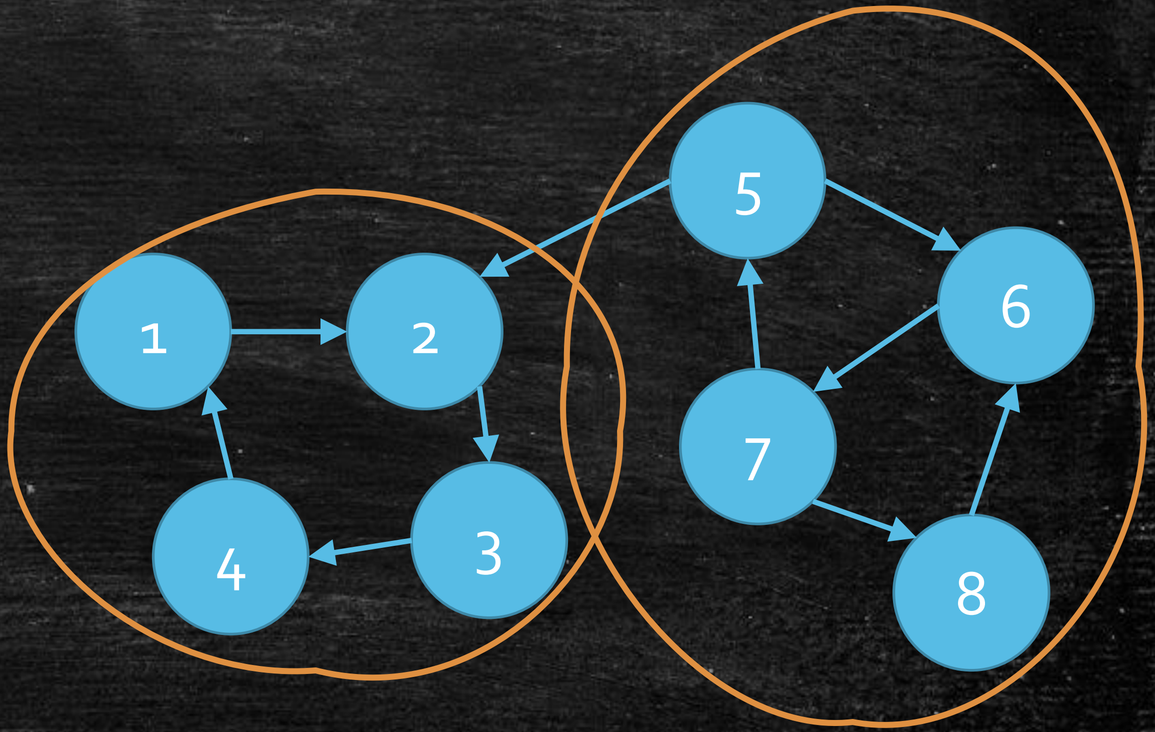
- Follow the descending topological order to DFS vertices.
  - Explore from a vertices inside the **tail** SCC.
  - Form the SCC and remove it from the graph.
  - Repeat……
- Puzzle
  - We want to know the tail SCC.
  - Then we choose a vertex inside.
  - We explore from the vertex and then we get the tail SCC.

A Better Attempt

- Follow the descending topological order to DFS vertices.
  - Explore from a vertices inside the **tail** SCC.
  - Form the SCC and remove it from the graph.
  - Repeat……
- Puzzle
  - We want to know the tail SCC.
  - Then we choose a vertex inside.
  - We explore from the vertex and then we get the tail SCC.
- Answer
  - We have an **AMAZING** way to find one vertex surely inside the **tail** SCC.

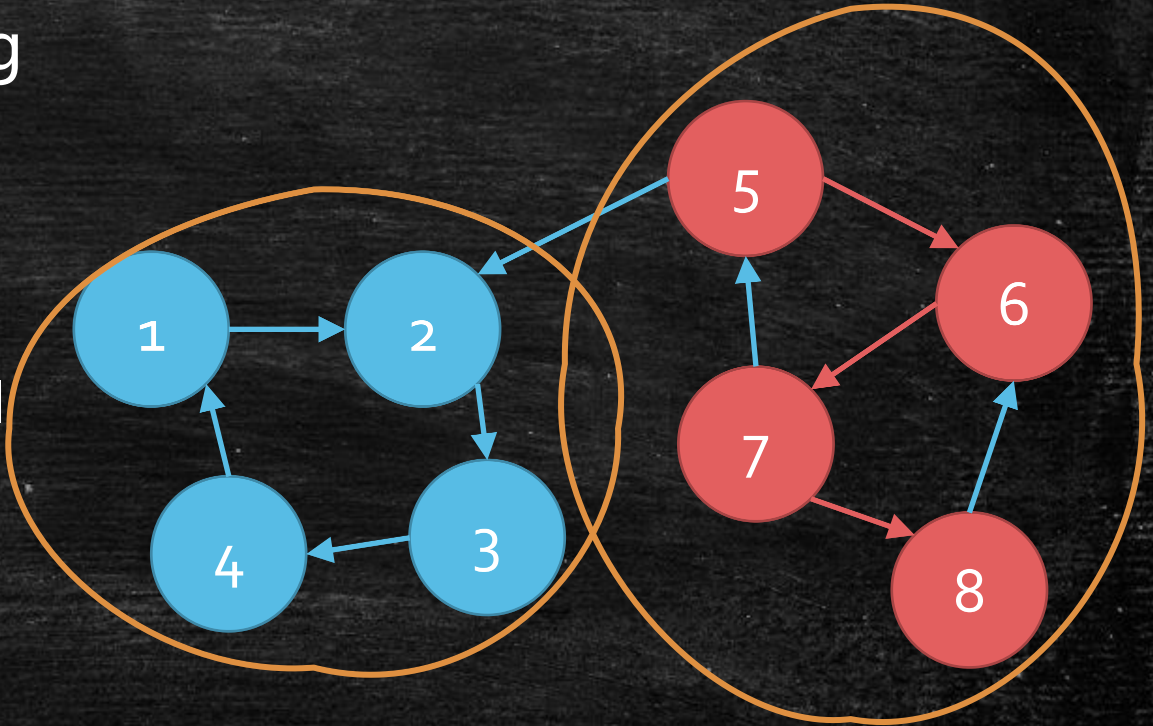
Find one vertex surely inside the **tail** SCC.

- Recall the topological ordering
  - **Tail vertex** is the one with earliest finish time.
  - Can we apply it here?
  - Assume the DFS Start from 5!
  - Conjecture: tail vertex is in the tail SCC!



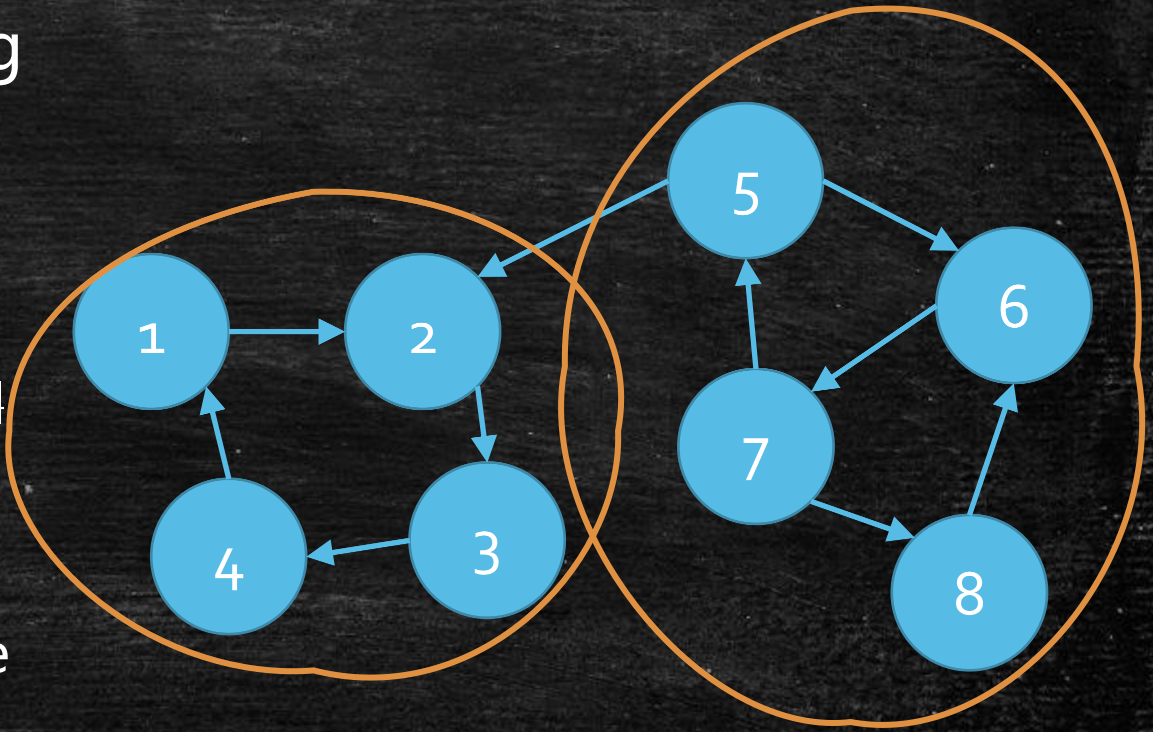
Find one vertex surely inside the **tail** SCC.

- Recall the topological ordering
  - **Tail** is the one with smallest finish time.
  - Can we apply it here?
  - Start from 5?
  - Conjecture: tail vertex is in the tail SCC!
- Problems
  - 8 is not in the **Tail** SCC.
  - We may have back edges.
  - 8 can still have a way to go out!



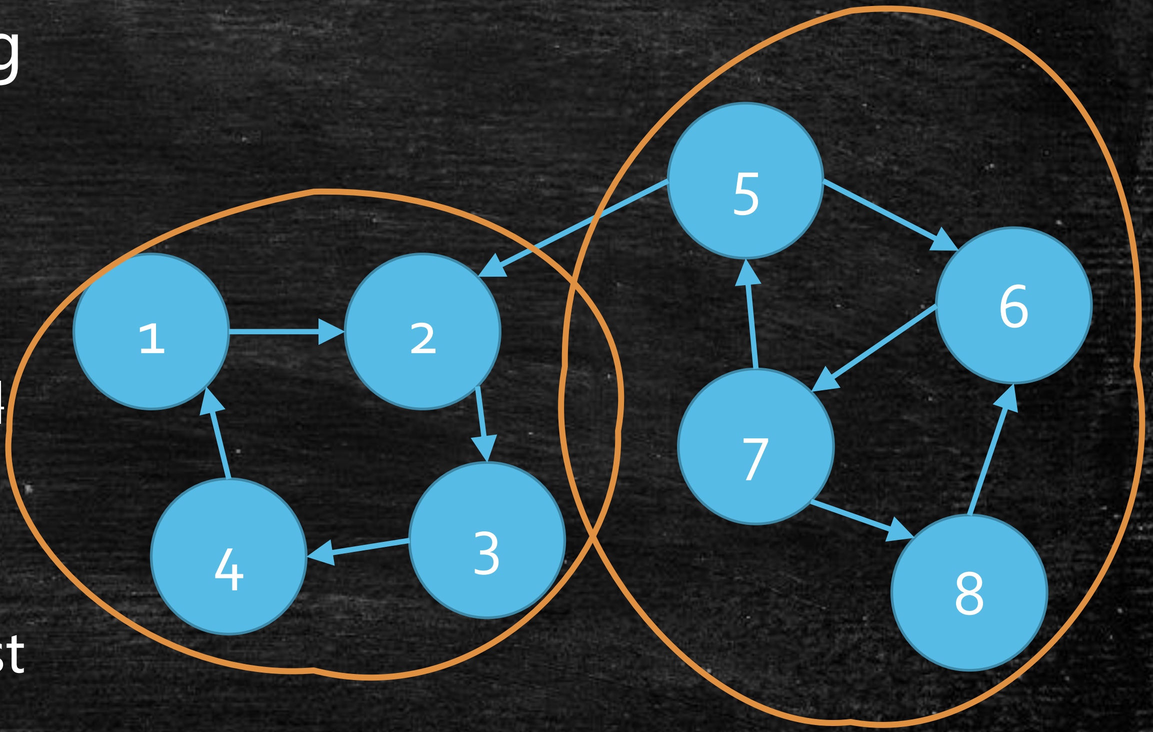
Find one vertex surely inside the **tail** SCC.

- Recall the topological ordering
  - **Tail** is the one with smallest finish time.
  - Can we apply it here?
  - Start from 5?
  - ~~Conjecture: tail vertex is in the tail SCC!~~
- But
  - What does it mean if we finish the explore from 5?
  - It means we has discovered every vertices that 5 can reach!



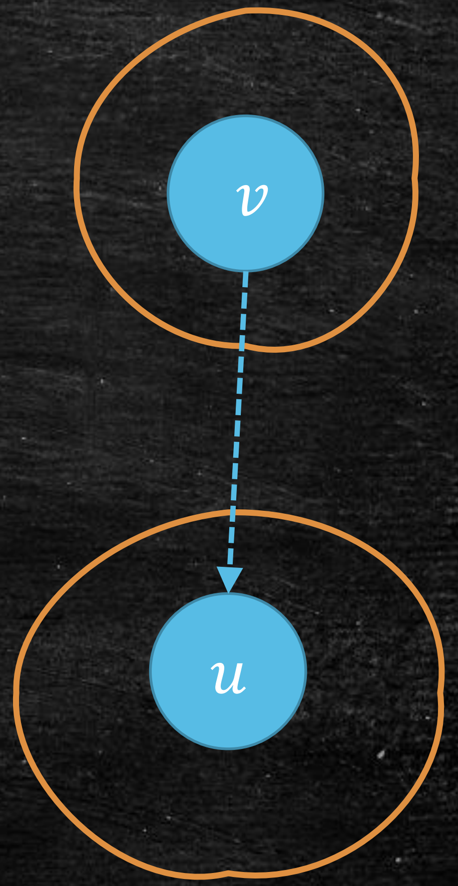
Find one vertex surely inside the **tail** SCC.

- Recall the topological ordering
  - **Tail** is the one with smallest finish time.
  - Can we apply it here?
  - Start from 5?
  - ~~Conjecture: tail vertex is in the tail SCC!~~
- But
  - What about the vertex with largest finish time?



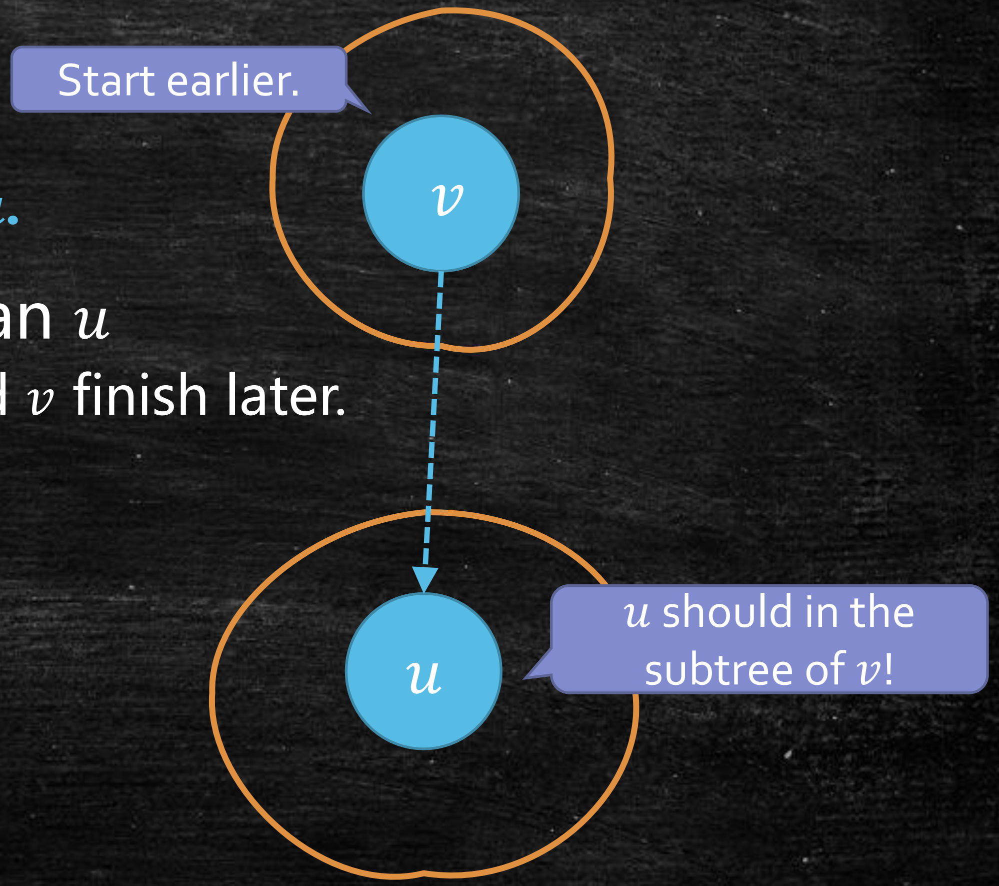
Find one vertex surely inside the **tail** SCC.

- **Naïve Idea:** the SCC contains the **largest** finish time vertex must be the **head SCC**.
- **Proof by Contradiction**
  - Assume it is not true
    - u has the largest finish time.
    - v inside another SCC has a path to u.



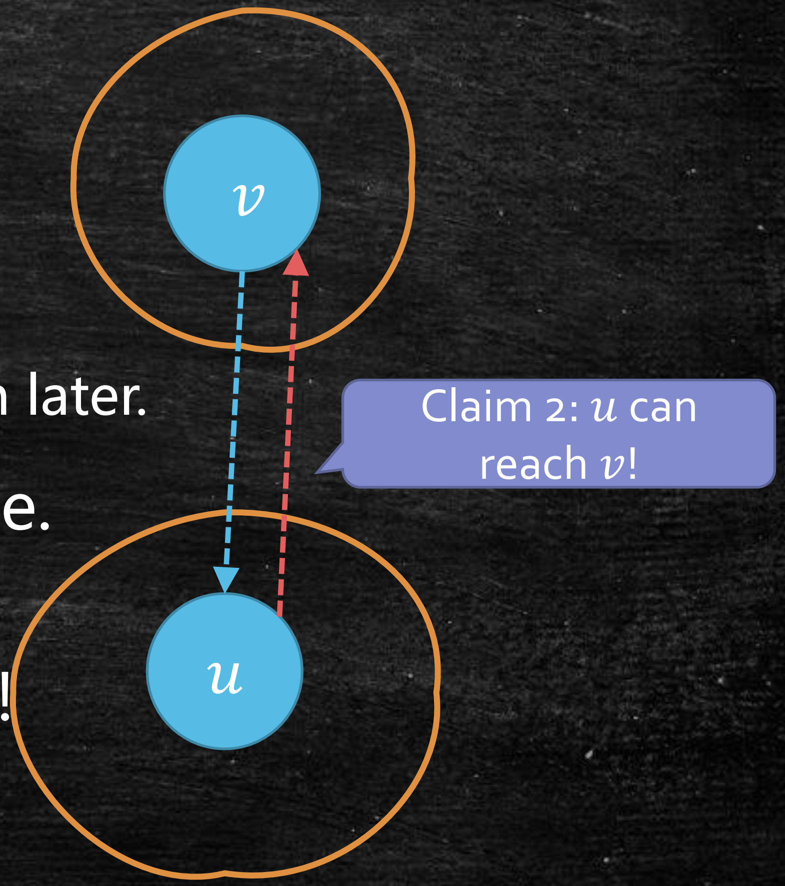
Find contradiction!

- **Assumption!**
  - **u has the largest finish time.**
  - **v inside another SCC has a path to u.**
- Claim 1: v can not start earlier than u
  - Otherwise, u is in the subtree of v and v finish later.



Find contradiction!

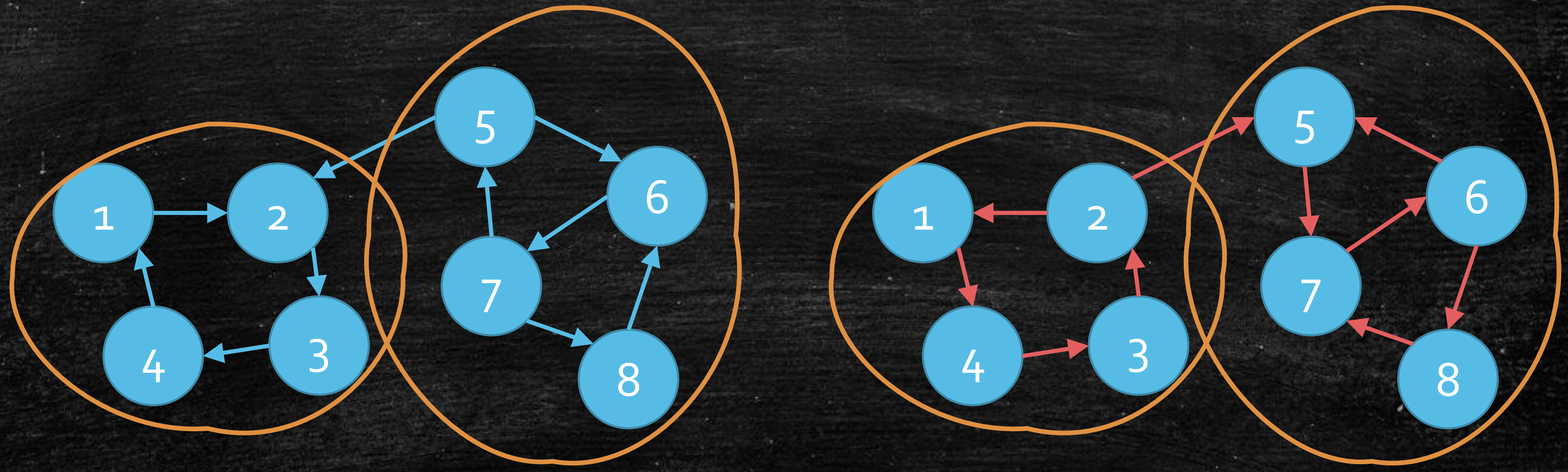
- **Assumption!**
  - **u has the largest finish time.**
  - **v inside another SCC has a path to u.**
- Claim 1: v can not start earlier than u
  - Otherwise, u is in the subtree of v and v finish later.
- Claim 2: v can not be in another DFS tree.
  - Because v start later but finish earlier!
- Claim 3: v and u are strongly connected!
  - **Contradiction!**



How to use this property?

Find one vertex surely inside the **tail** SCC.

- The amazing idea!
  - Find the vertex in the head SCC in the reverse graph!



How efficient you can do?

Realize the idea efficiently

- **Basic Plan**

1. Construct G^R
2. DFS G^R with **finish time**.
3. Choose v with the largest **finish time**.
4. Explore(v) in G.
5. When it returns, reached vertices form one SCC ($|V_1|$).
6. Remove them in both G and G^R.
   - $|V| \leftarrow |V| - |V_1|$
   - $|E| \leftarrow |E| - |\Delta E|$
7. Repeat from 2.

$O(|E|)$

$O(|V'| + |E'|)$

$O(1)$

$O(|V_1| + |E_1|)$

$O(|V_1|)$

$O(|\Delta E|)$

At most $|V|$ rounds.

Realize the idea efficiently

- **Super Plan**
  1. DFS G^R and maintain a **sorted list** by the finish time.
  2. DFS G by the **descending order** of the finish time.
     1. Keep explore vertices by the descending order.
     2. Do not start from a reached vertex.
  3. Each explore() forms a SCC.

$O(|V'| + |E'|)$

$O(|V'| + |E'|)$

Is it correct.

- It is not straightforward.
- The Claim we have: the SCC contains the **largest** finish time vertex must be the **head SCC**.
- Not enough!

The Correctness of The Super Plan

- Prove each start point we choose is in the head SCC among the remaining graph.
- A Generalize Lemma:
- If v can reach u in G^R, and they are from different SCCs, then we have $finish[v] > finish[u]$.

Today's goal

- Learn **DFS.**
- Learn applications of **DFS.**
  - Connected Components
  - Cycle Check
  - Topological Order
  - Strongly Connected Components
- Learn to form a **nice** property of graphs.
  - Strongly Connected Components
- Learn to analyze the **correctness** of graph algorithms.