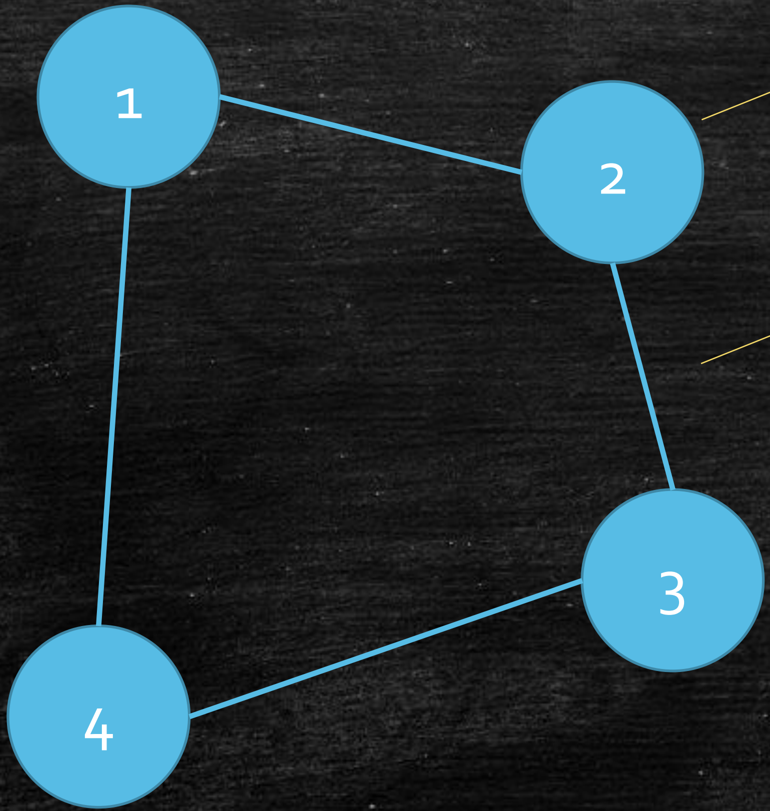
Basic Graph Algorithms

Depth First Search and Its Applications

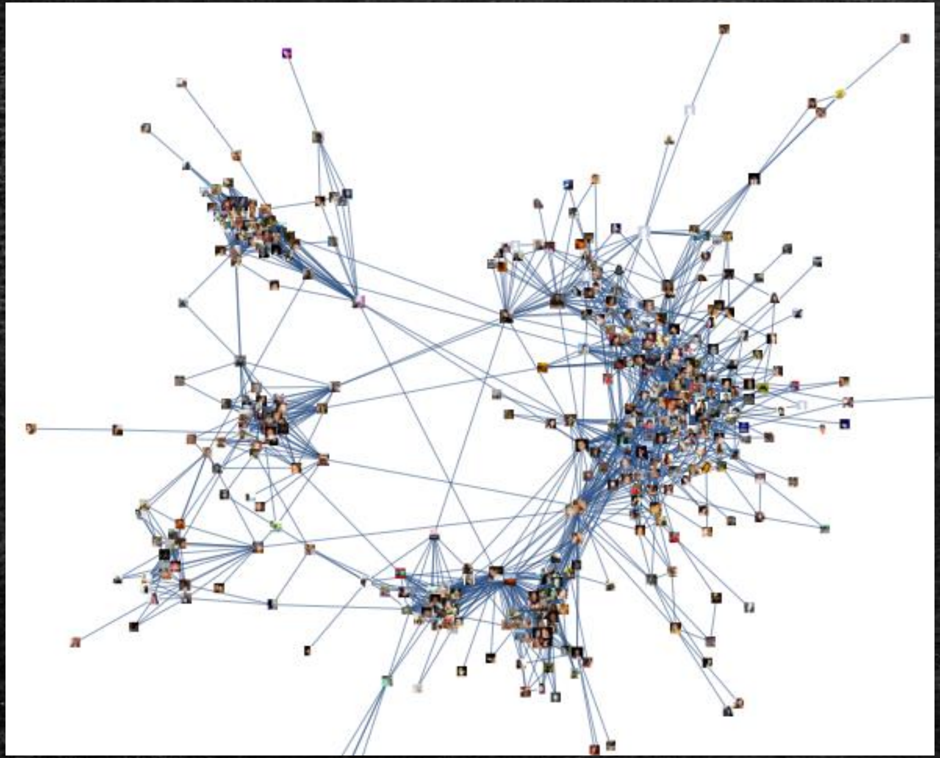
What is graphs?



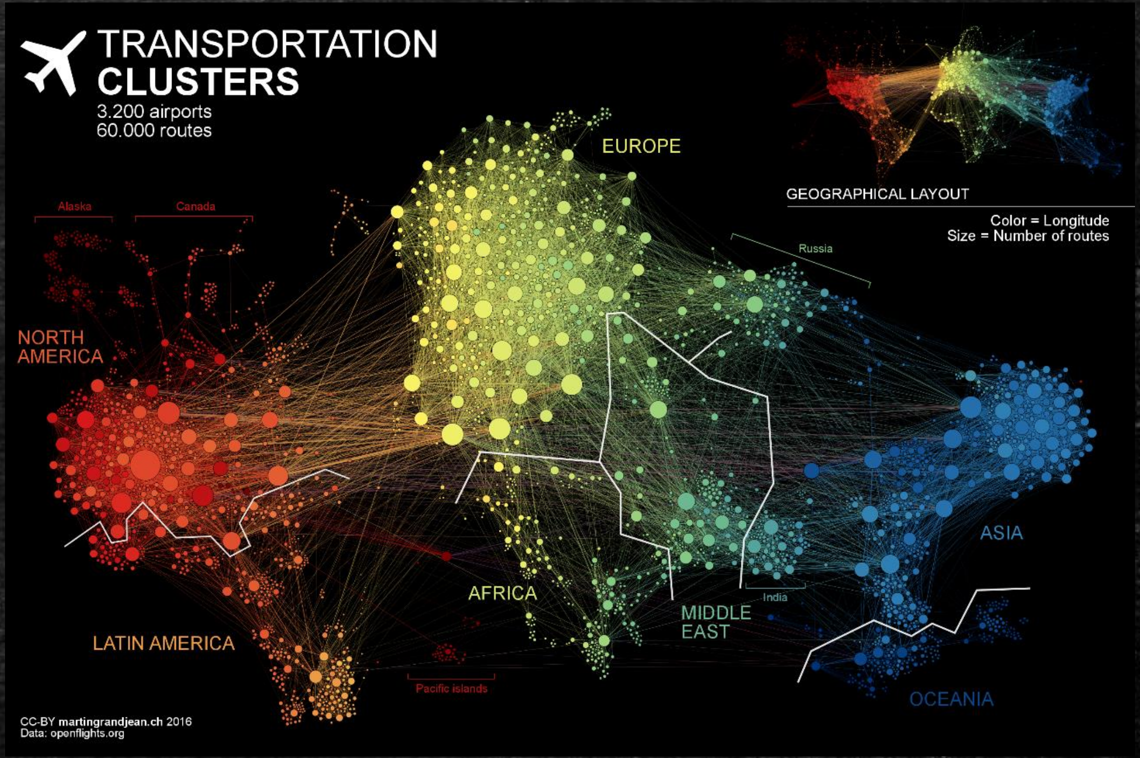
Vertex: for an objective

Edges: for connections

Large Graphs in Real World

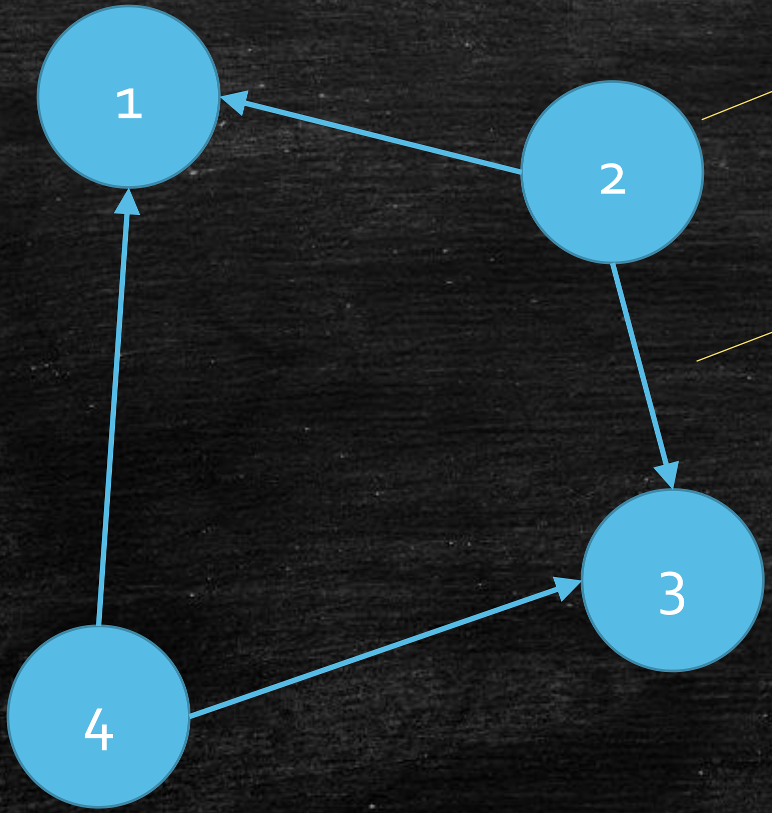


Facebook friends



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?
  - $G(V, E)$
  - $0 \leq |E| \leq |V|(|V| - 1) \leq O(|V|^2)$

How to store a graph?

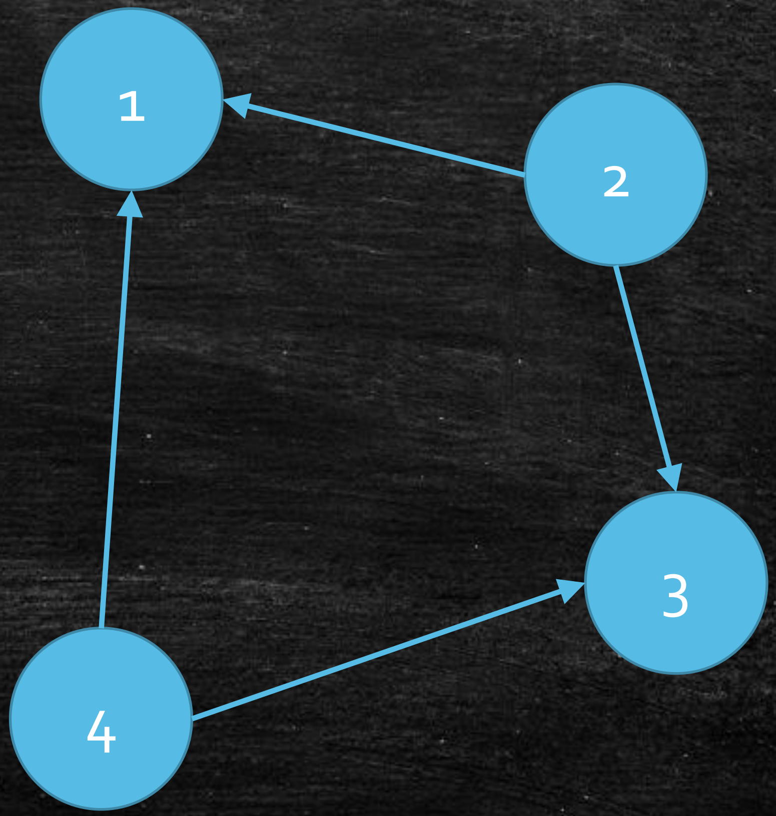
- Adjacency Matrix
- Adjacency List

Adjacency Matrix

Space: $\boldsymbol{O(V^2)}$

- $|V| \times |V|$ 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

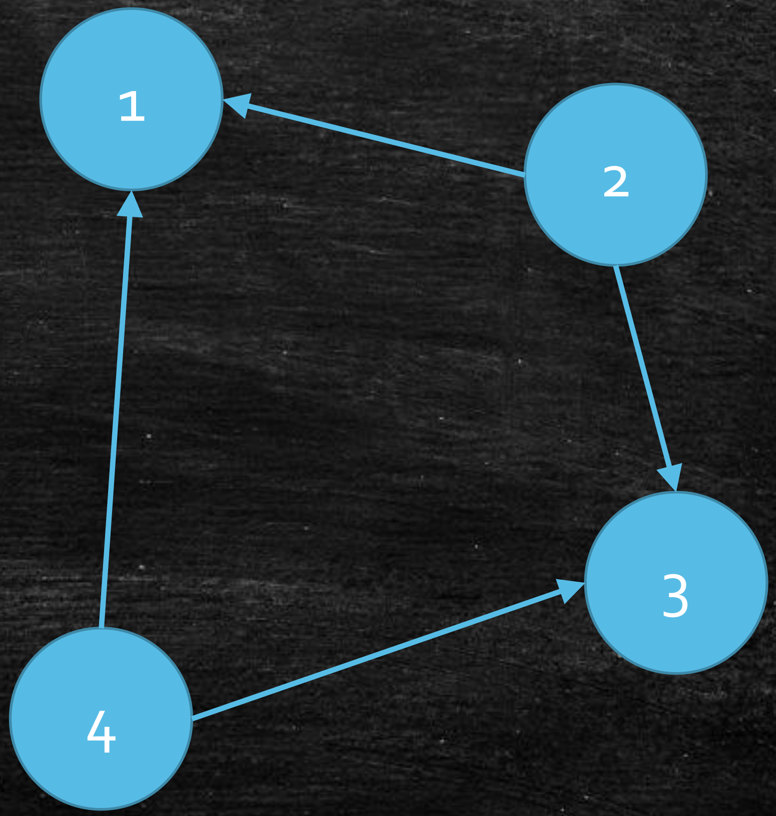


Adjacency List

Space: $O(V + E)$

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

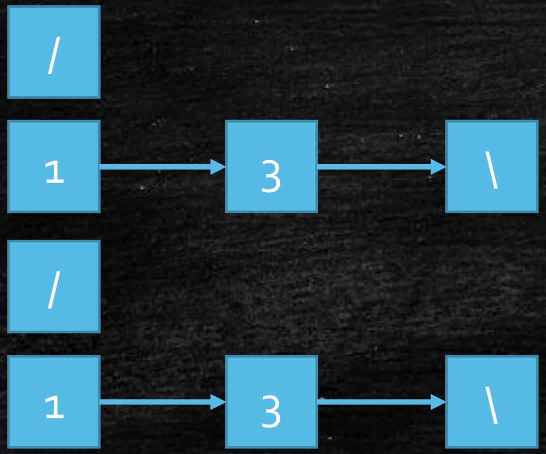
	1	2	3	4
1	0	0	0	0
2	1	0	1	0
3	0	0	0	0
4	1	0	1	0

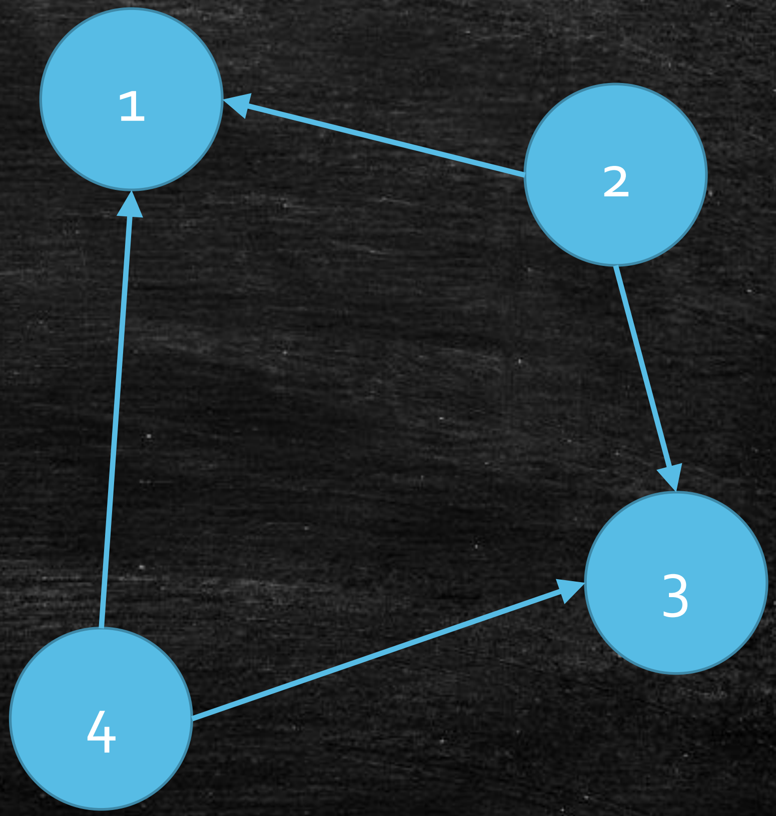


Adjacency List

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 $(u, v) \in E$

$node \leftarrow new\ Node$

$node.v \leftarrow v$

$node.next \leftarrow adj[u]$

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

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 Matrix, and a vertex u.
- **Output:** The set of vertices u can reach.

Observations

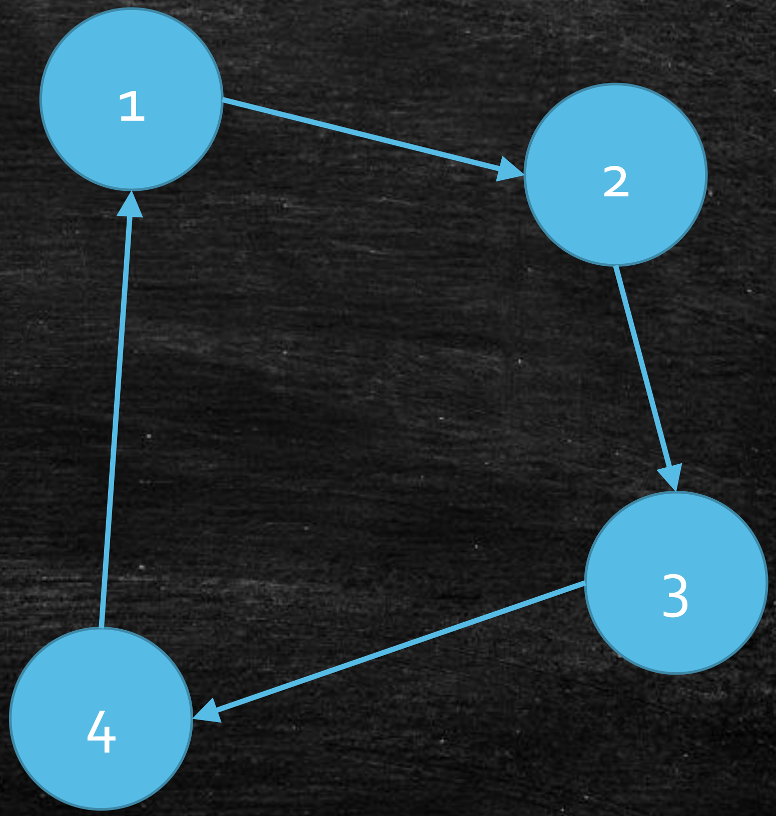
- Basic **observation**:
  - If v is in the Adjacent List (neighbor set) of u?
  - v is reachable.
- Advanced **observation**:
  - If v is reachable
  - Vertices in $v's$ Adjacent List (neighbor set) is also reachable.

Algorithmic Idea

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

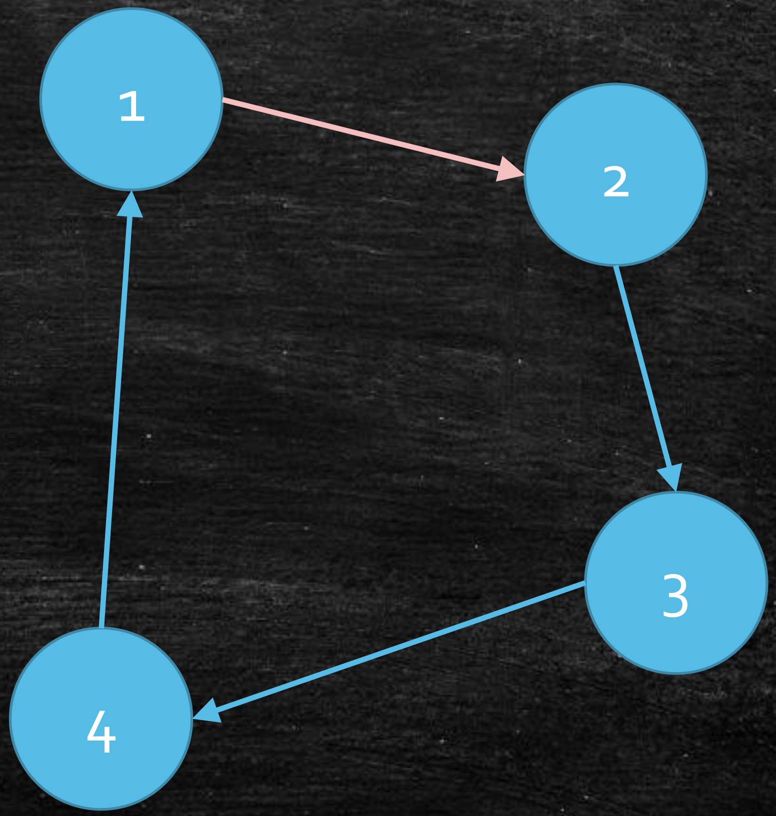
Algorithmic Idea

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



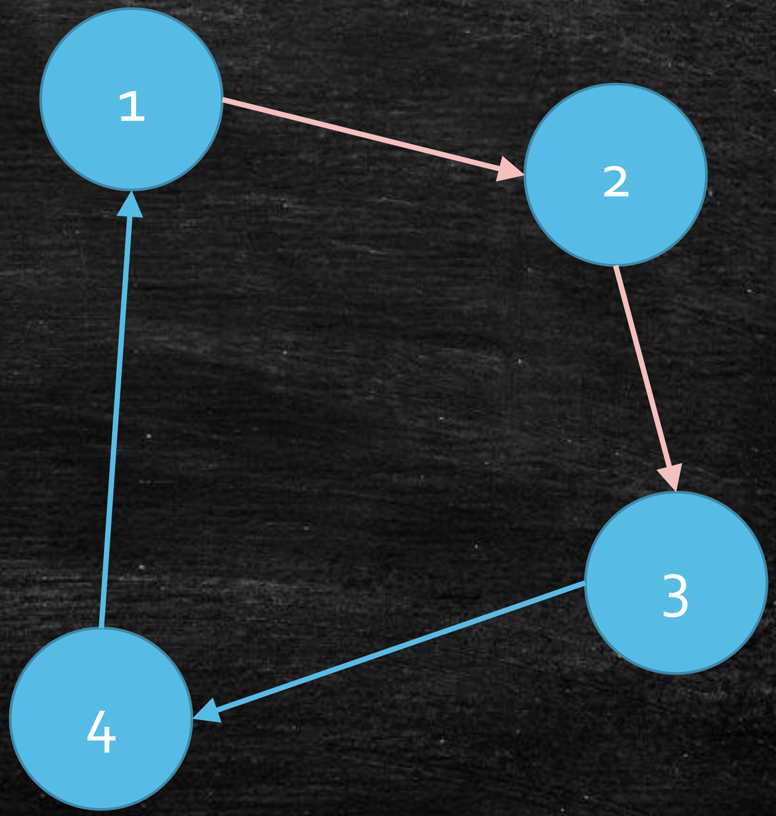
Algorithmic Idea

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



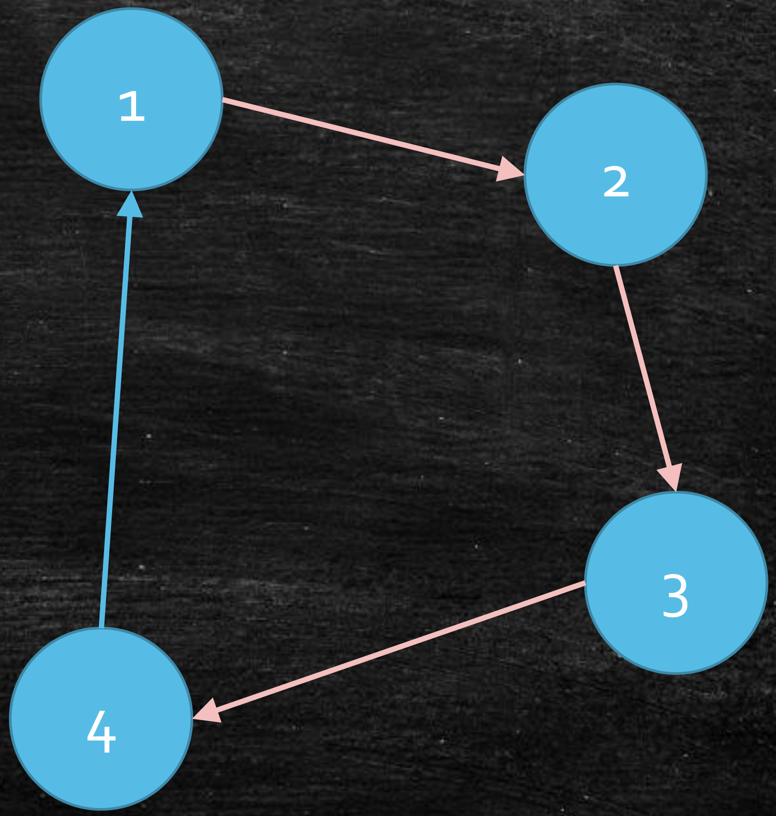
Algorithmic Idea

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



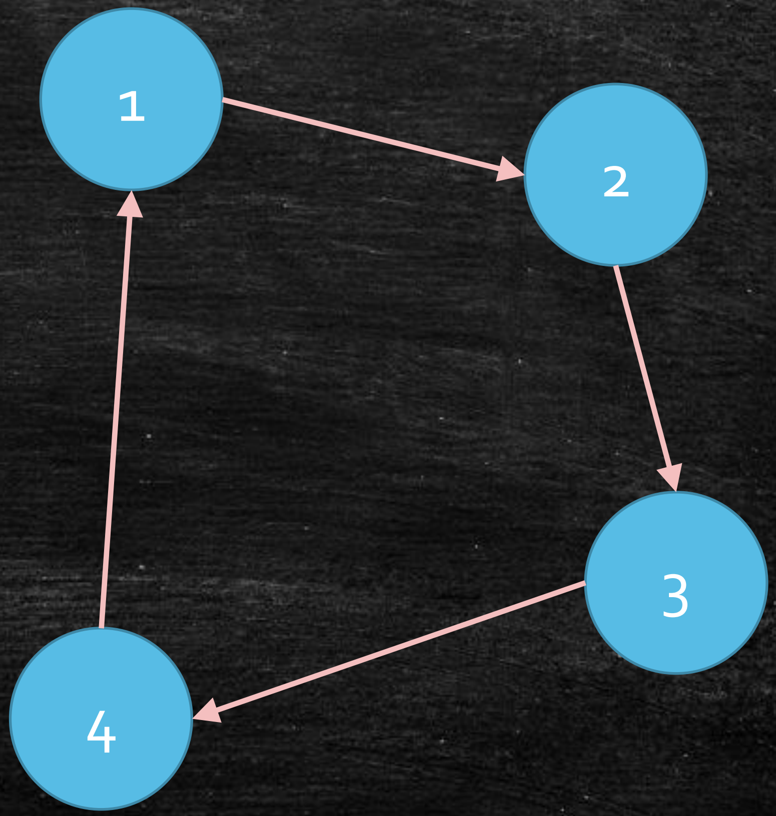
Algorithmic Idea

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



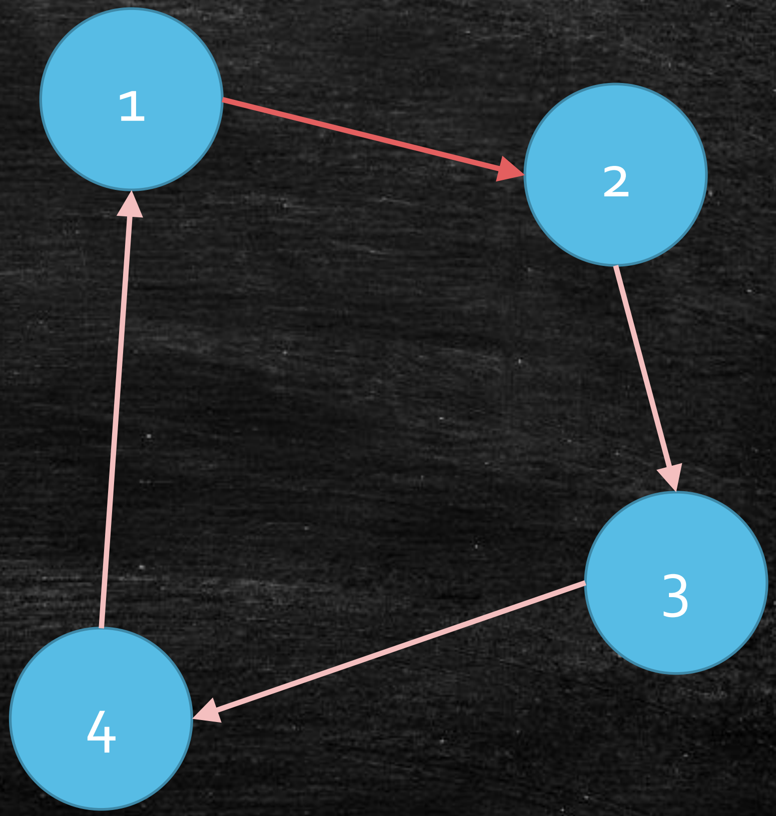
Algorithmic Idea

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



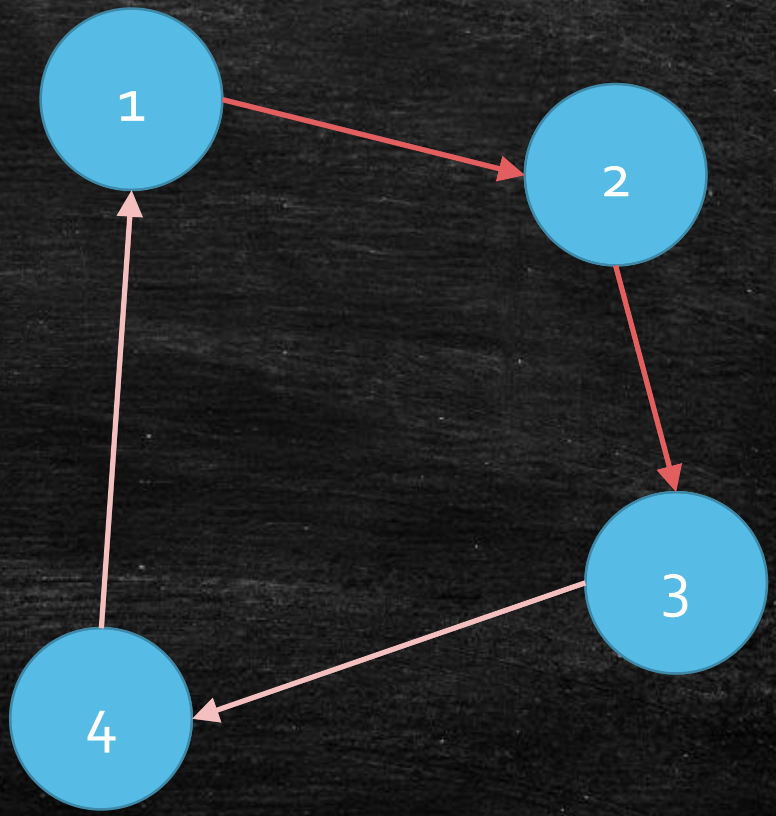
Algorithmic Idea

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



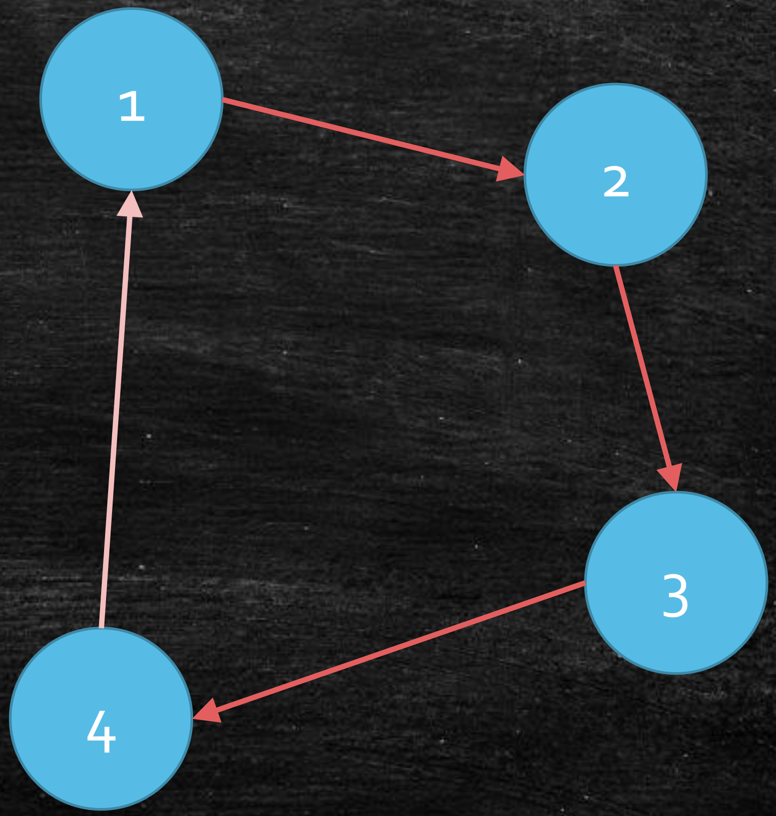
Algorithmic Idea

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



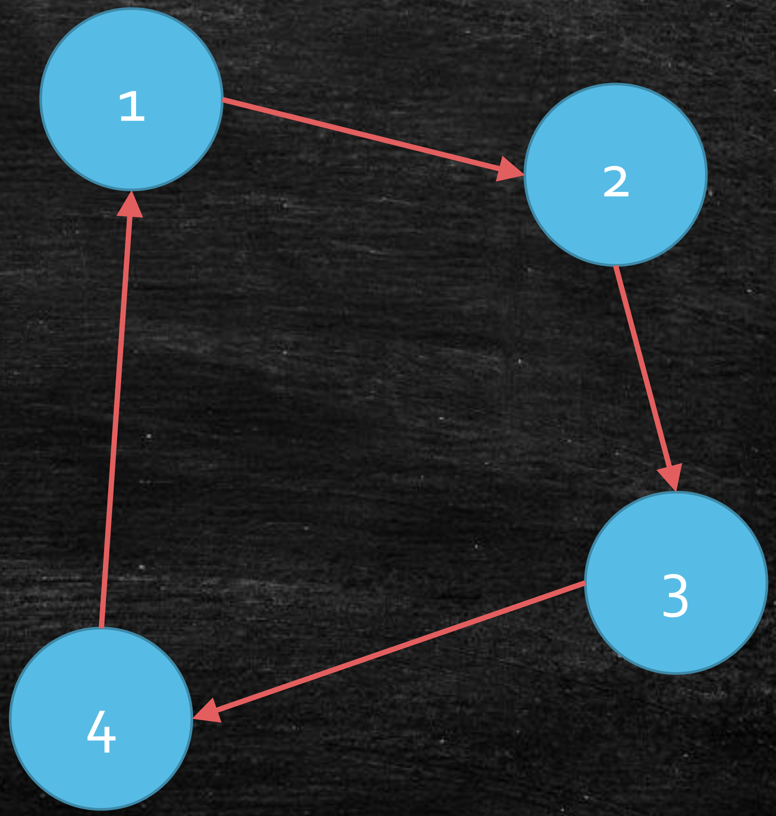
Algorithmic Idea

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



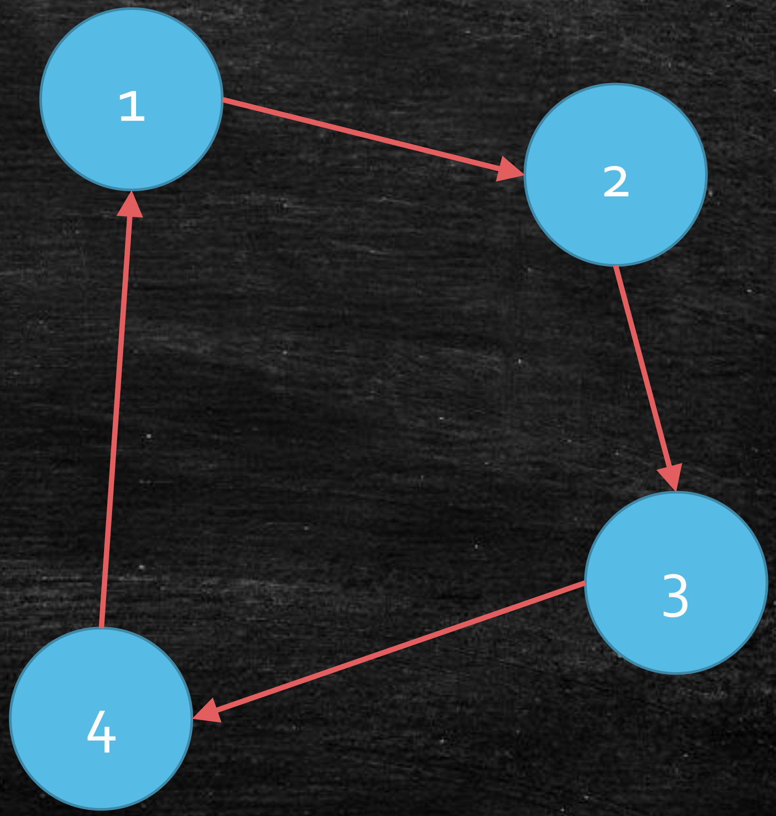
Algorithmic Idea

- 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!
  - $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 1$
- Solution
  - **Mark** a vertex when we reach it
  - Do not explore **marked** vertices



Depth-First Search

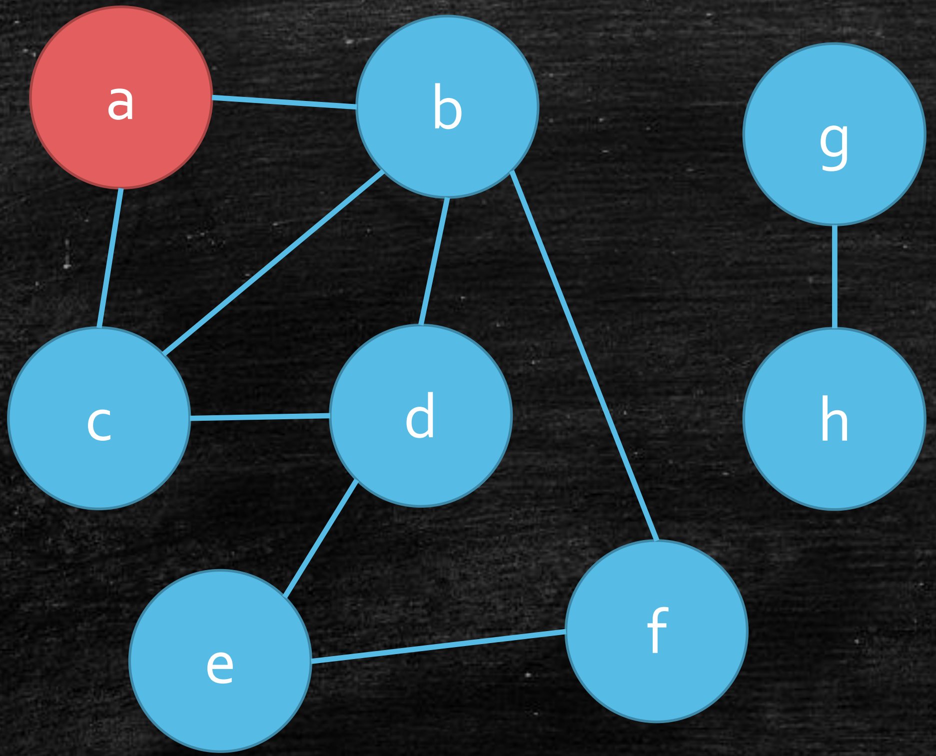
- Implement the Explore idea.
- What is DFS?
  - Explore & Explore
- Questions
  - How to loop all $(u, v) \in E$?
  - What is the running time of DFS?

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

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

DFS in undirected graphs

- How we DFS an undirected graph?

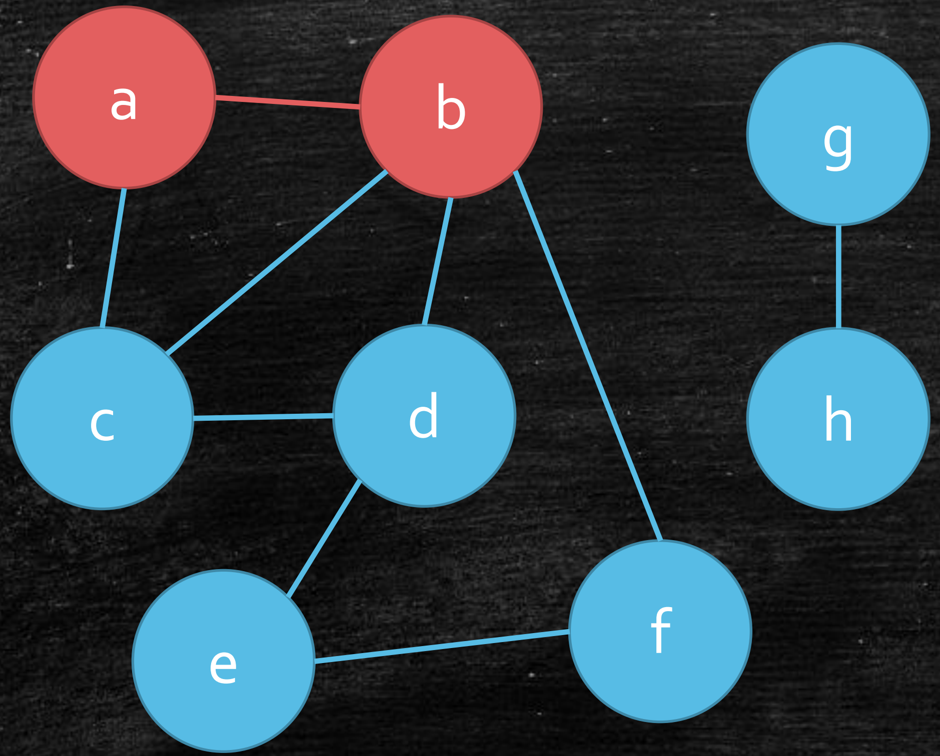


Function explore(v)
  $marked[v] \leftarrow true$
  for each $(u, v) \in E$
    if $marked[v] = false$
      explore(v)

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

DFS in undirected graphs

- How we DFS an undirected graph?

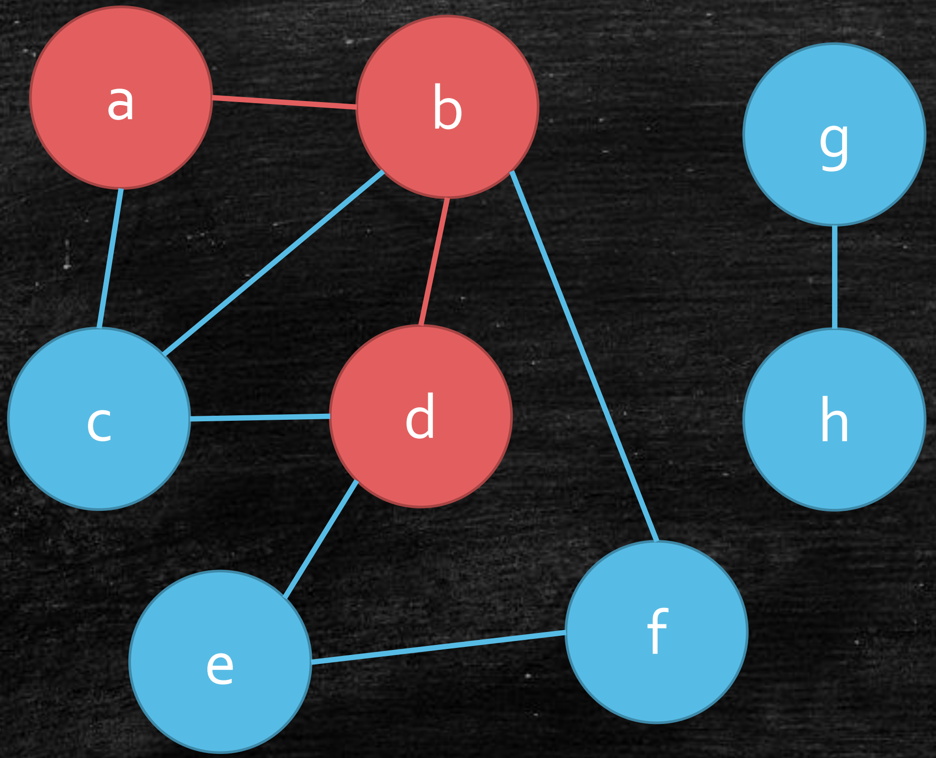


Function explore(v)
  $marked[v] \leftarrow true$
  for each $(u, v) \in E$
    if $marked[v] = false$
      explore(v)

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

DFS in undirected graphs

- How we DFS an undirected graph?

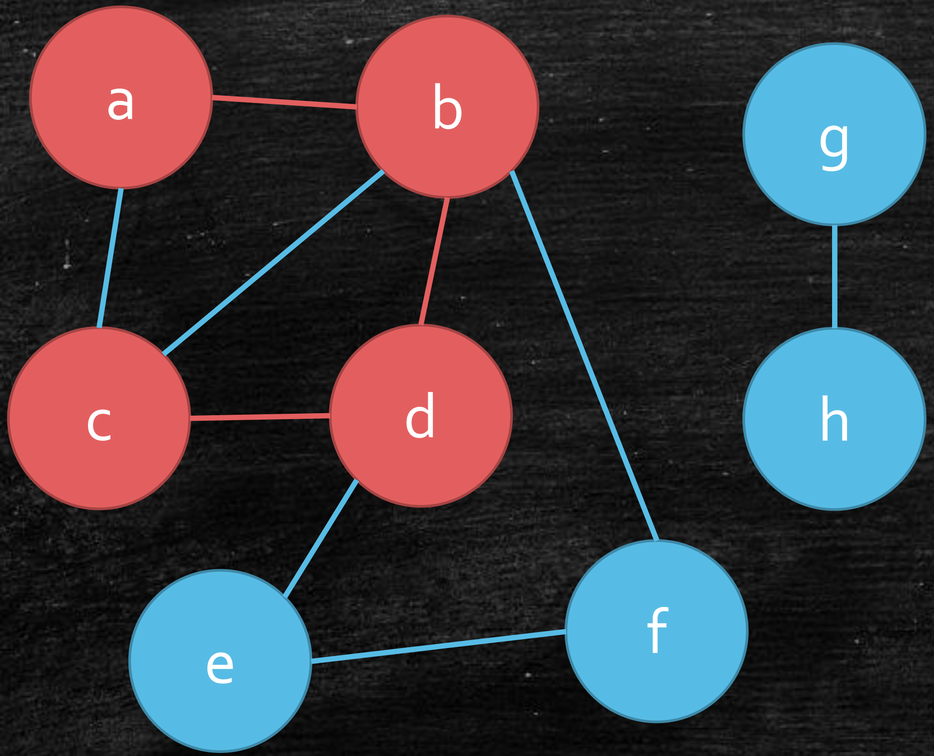


Function explore(v)
    $marked[v] \leftarrow true$
    for each $(u, v) \in E$
        if $marked[v] = false$
            explore(v)

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

DFS in undirected graphs

- How we DFS an undirected graph?

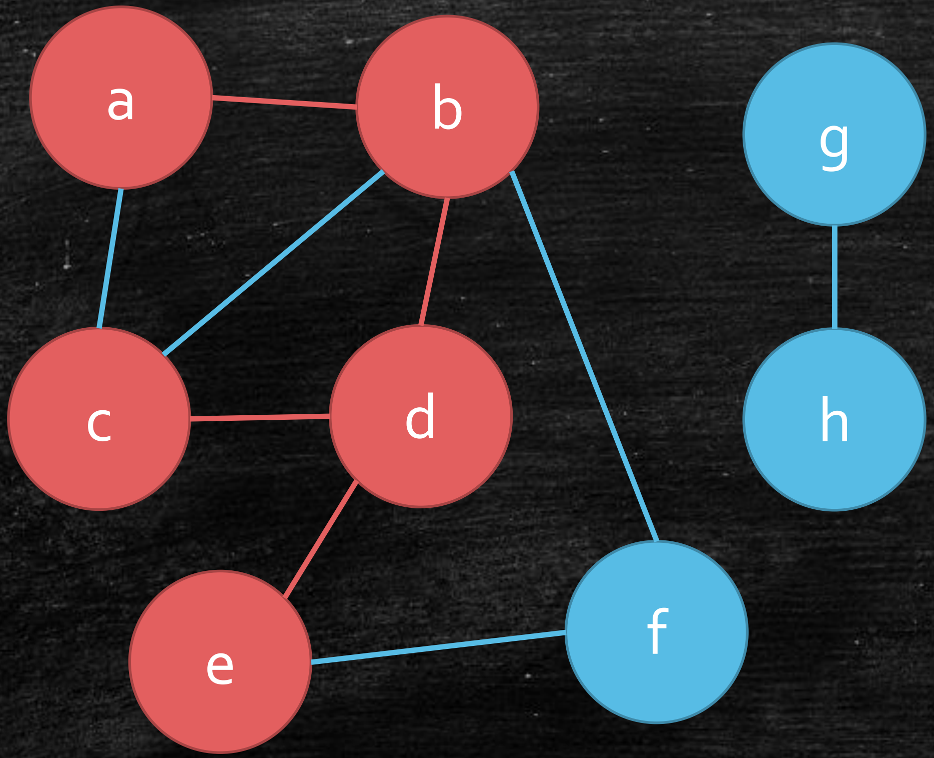


Function explore(v)
    $marked[v] \leftarrow true$
    for each $(u, v) \in E$
        if $marked[v] = false$
            explore(v)

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

DFS in undirected graphs

- How we DFS an undirected graph?

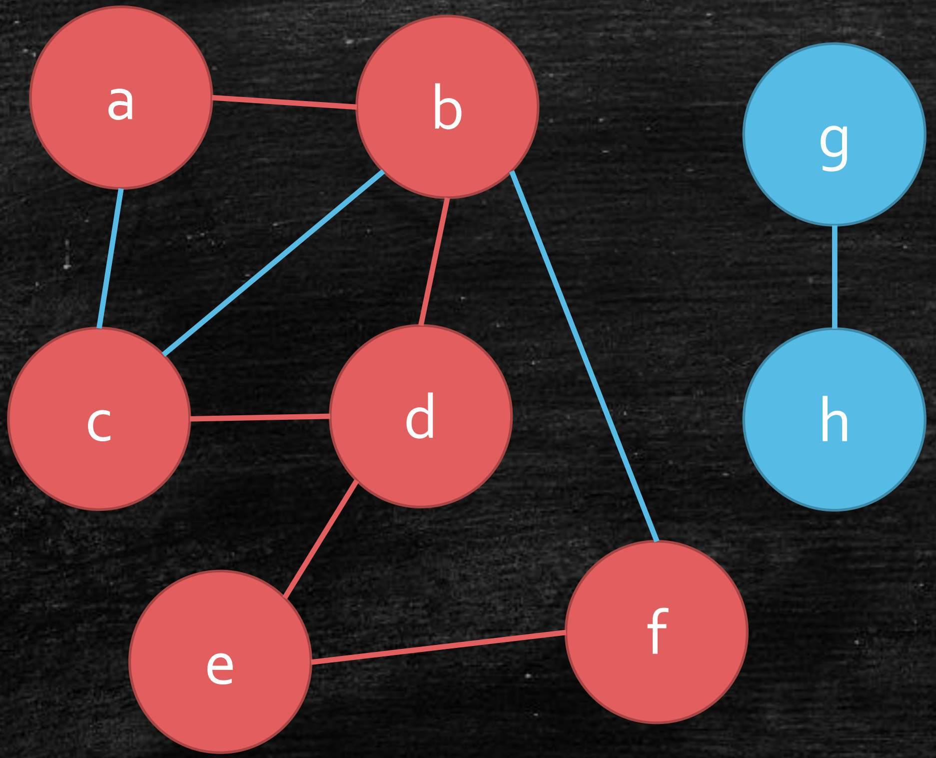


Function explore(v)
    $marked[v] \leftarrow true$
    for each $(u, v) \in E$
        if $marked[v] = false$
            explore(v)

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

DFS in undirected graphs

- How we DFS an undirected graph?

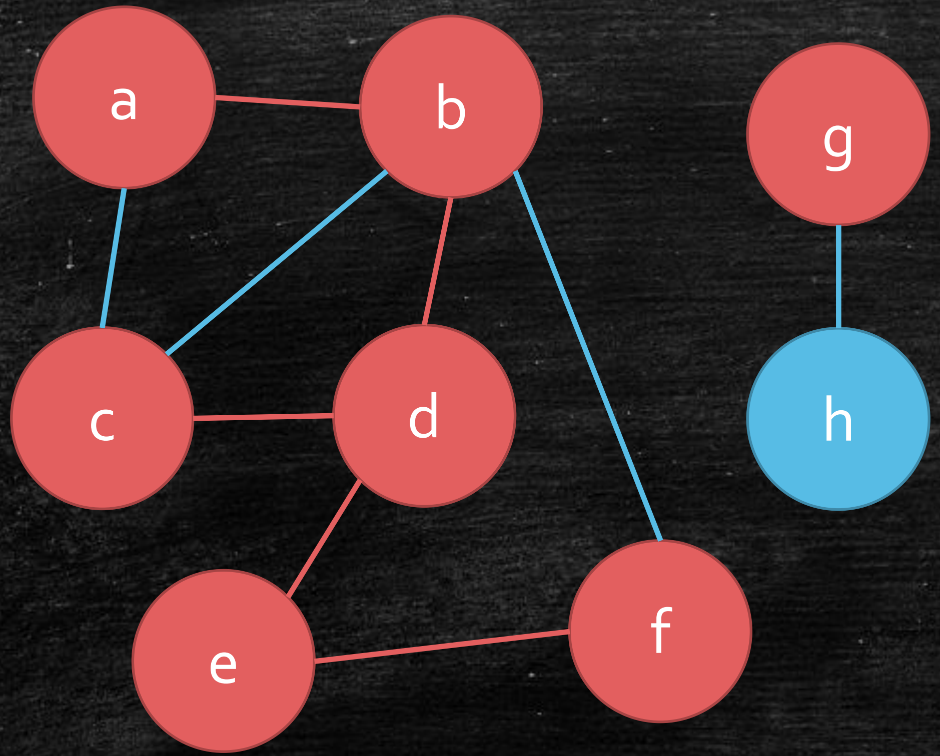


Function explore(v)
    $marked[v] \leftarrow true$
    for each $(u, v) \in E$
        if $marked[v] = false$
            explore(v)

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

DFS in undirected graphs

- How we DFS an undirected graph?

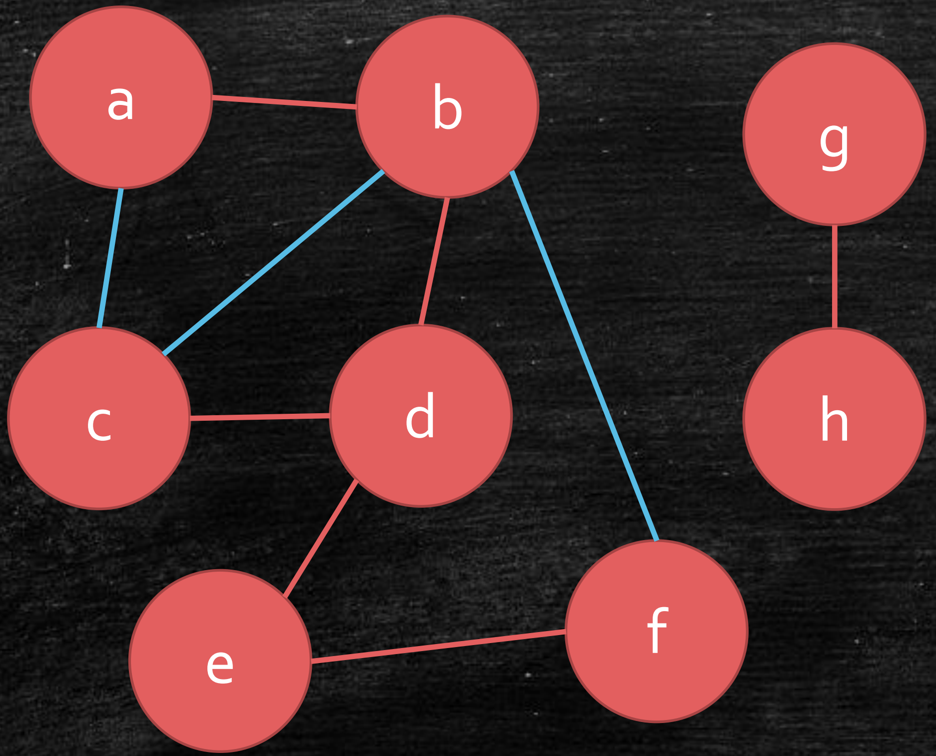


Function explore(v)
  $marked[v] \leftarrow true$
  for each $(u, v) \in E$
    if $marked[v] = false$
      explore(v)

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

DFS in undirected graphs

- How we DFS an undirected graph?

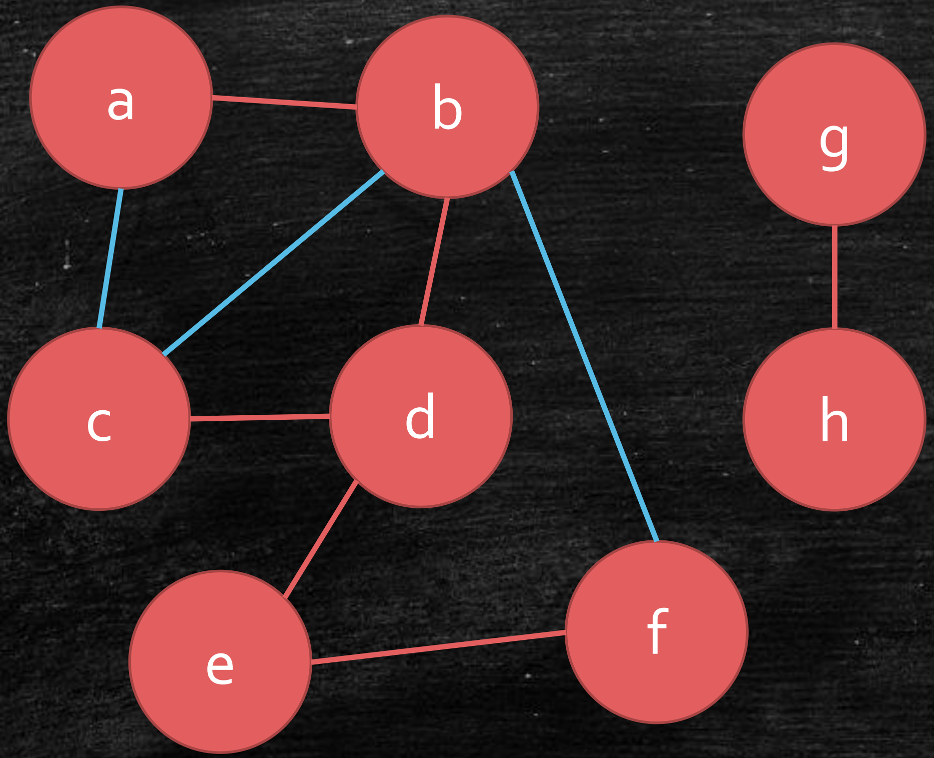


Function explore(v)
  $marked[v] \leftarrow true$
  for each $(u, v) \in E$
    if $marked[v] = false$
      explore(v)

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

Discussion

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



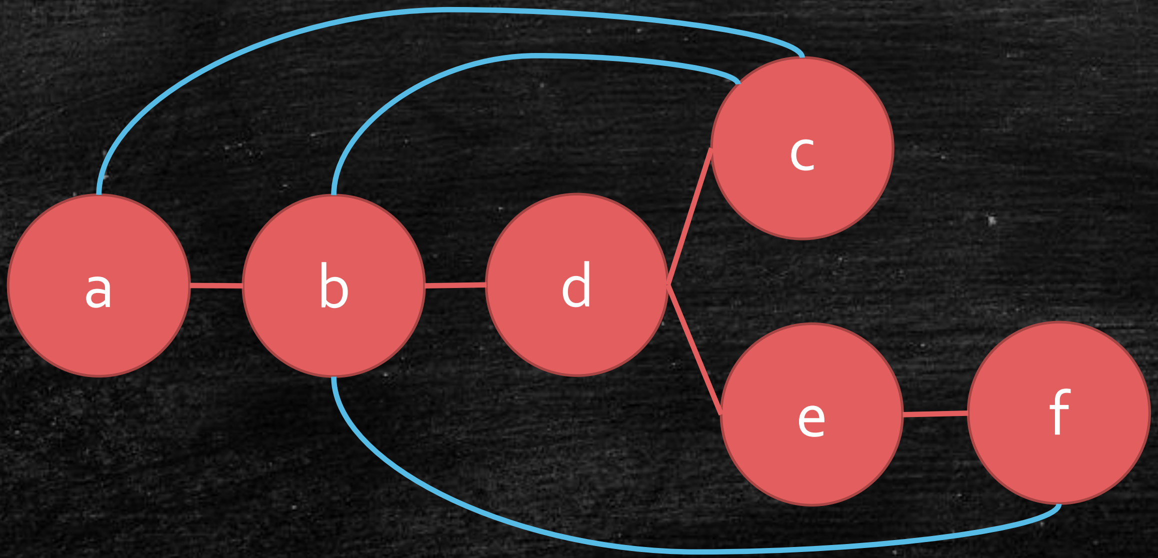
Function explore(v)
  $marked[v] \leftarrow true$
  for each $(u, v) \in E$
    if $marked[v] = false$
      explore(v)

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

DFS Tree (One Connected Component)

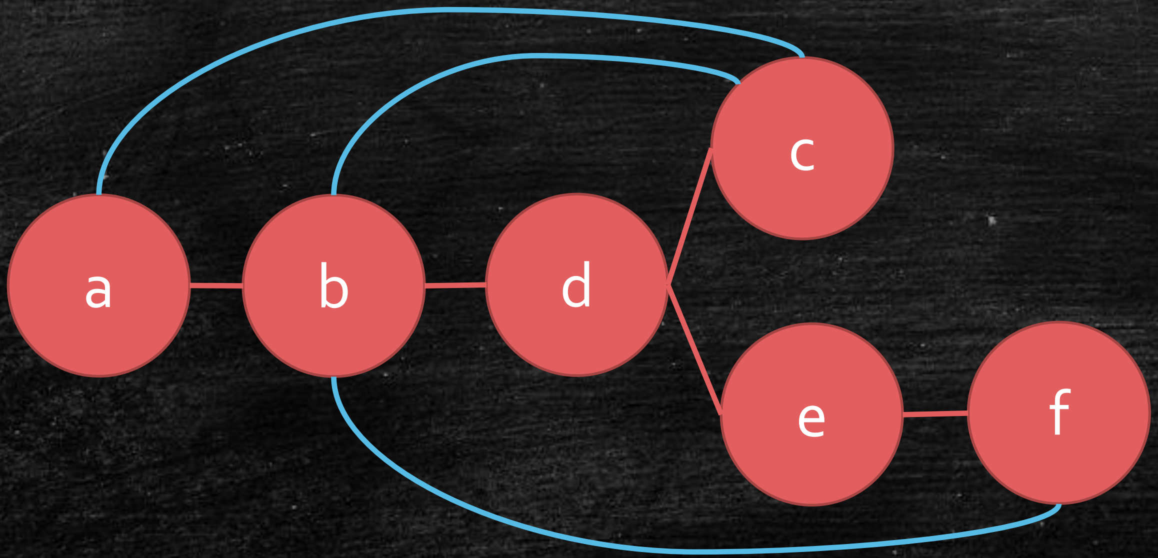
- Show the relationship among vertices
  - Root: the first explored vertex
  - If we explore v from u, then v is u's child.

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



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 $marked[v] = false$
      explore(v)

- 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 $\longleftrightarrow$ G has cycles
- How to prove it?

Proof of The Conjecture

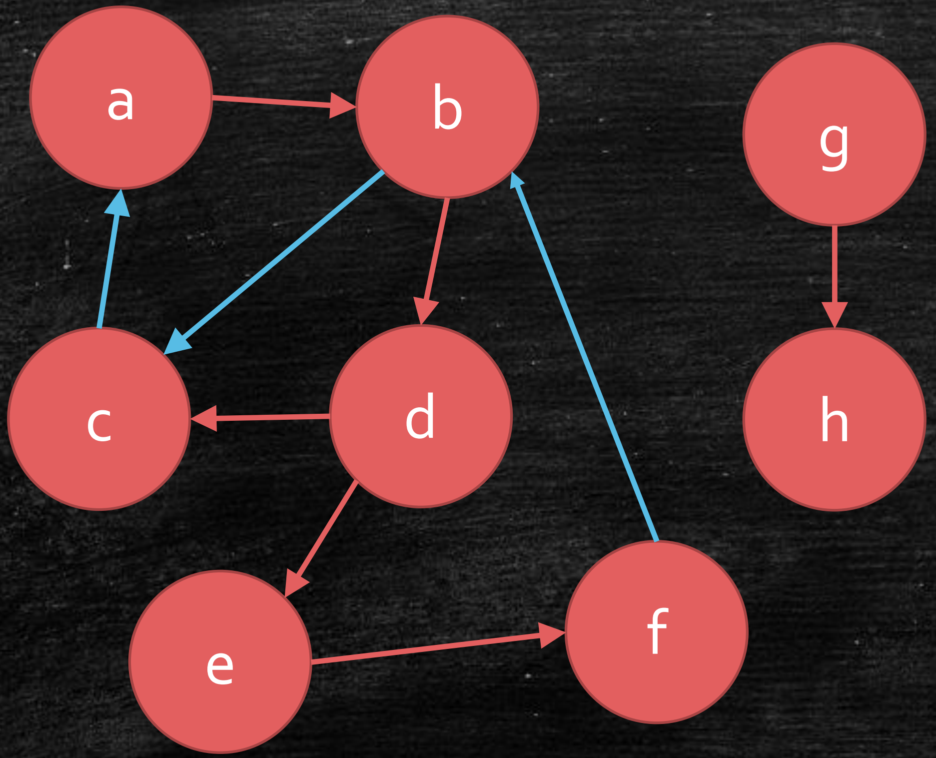
- **Conjecture:** T has back edges $\longleftrightarrow$ G has cycles
- Proof
- $\rightarrow$: If T has a back edge, then G has a cycle.
  - Can we point out a cycle based on this back edge?
- $\leftarrow$: If G has a cycle, then T has a back edge.
  - Can we point out one back edge in the cycle?

DFS on Directed Graphs

What is the difference?

DFS on Directed Graphs

- Answer: verbatim, but with directions.

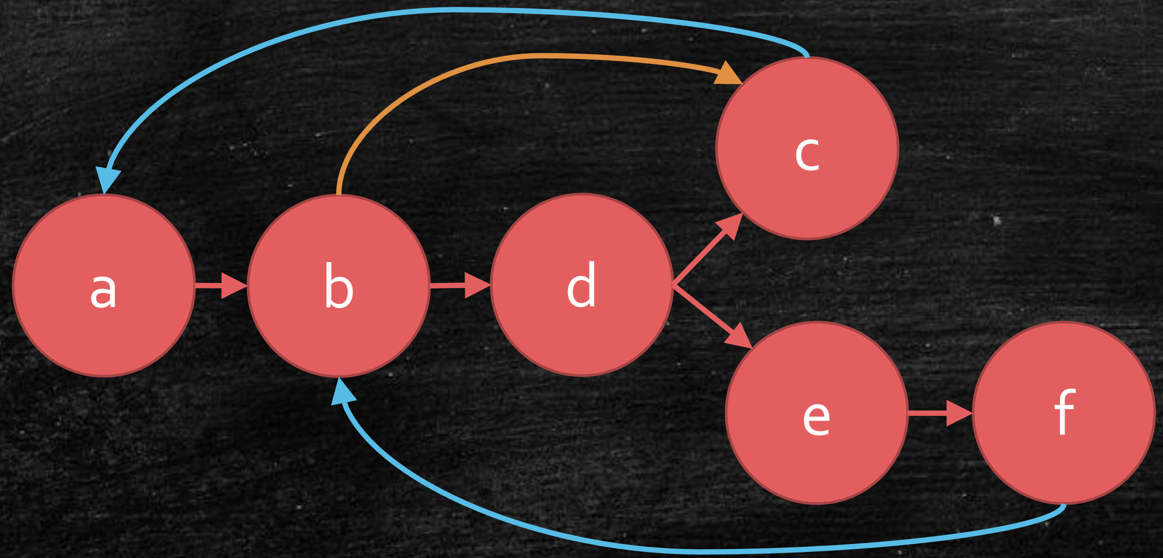


Function explore(v)
    $marked[v] \leftarrow true$
    for each $(u, v) \in E$
        if $marked[v] = false$
            explore(v)

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

DFS on Directed Graphs

- What about DFS trees?

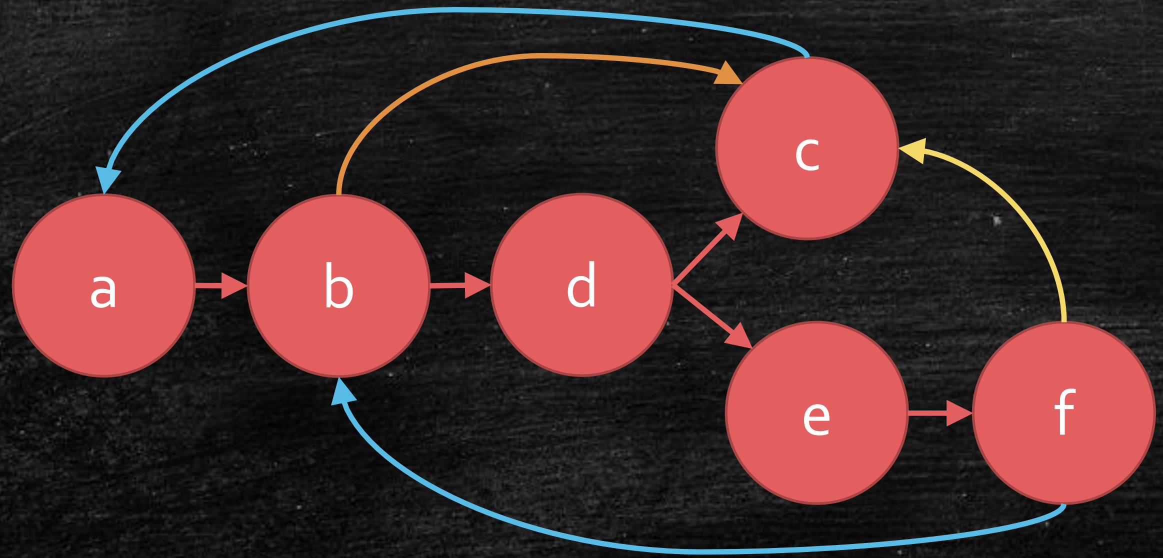


Function explore(v)
  $marked[v] \leftarrow true$
  for each $(u, v) \in E$
    if $marked[v] = false$
      explore(v)

- Kind of edges
  - Tree edges
  - Back edges

DFS on Directed Graphs

- What about DFS trees?

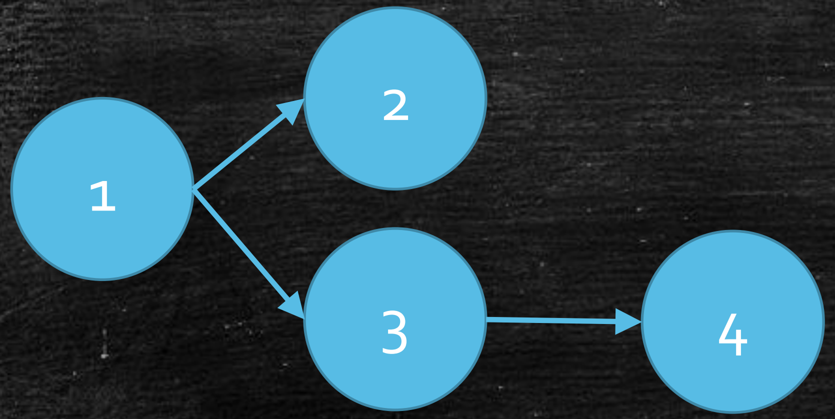


Function explore(v)
  $marked[v] \leftarrow true$
  for each $(u, v) \in E$
    if $marked[v] = false$
      explore(v)

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

Application: Topological Ordering

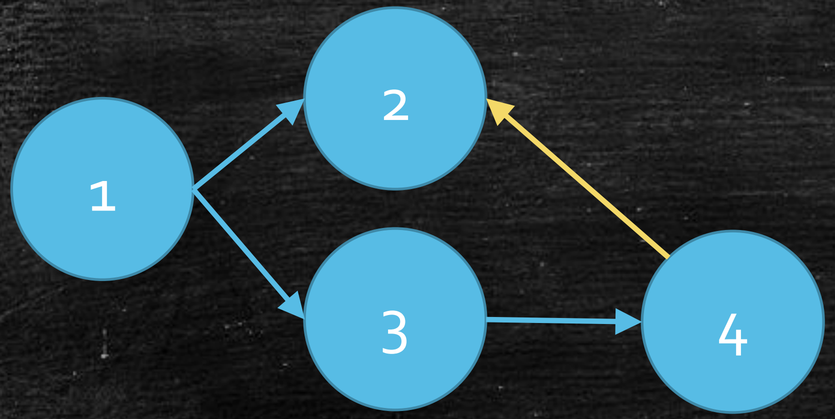
- A pre-requisite requirements graph



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

Application: Topological Ordering

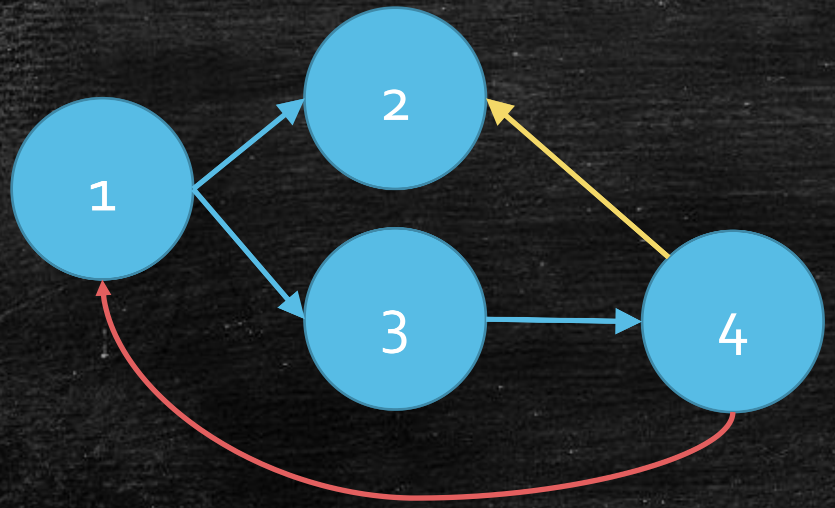
- A pre-requisite requirements graph



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

Application: Topological Ordering

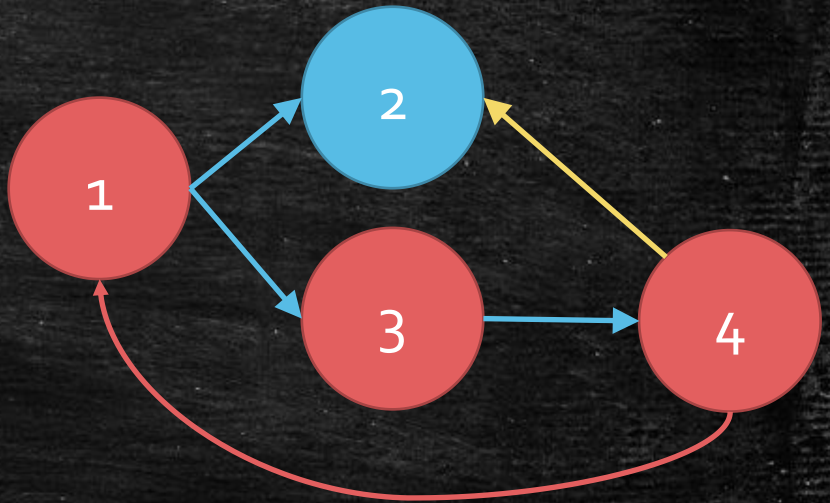
- A pre-requisite requirements graph



- 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
  - Compare to undirected cycle
- What if there is no cycle?
- **Directed Acyclic Graph (DAG)**
  - a directed graph that does not contain any cycle.



Topological Ordering for DAG

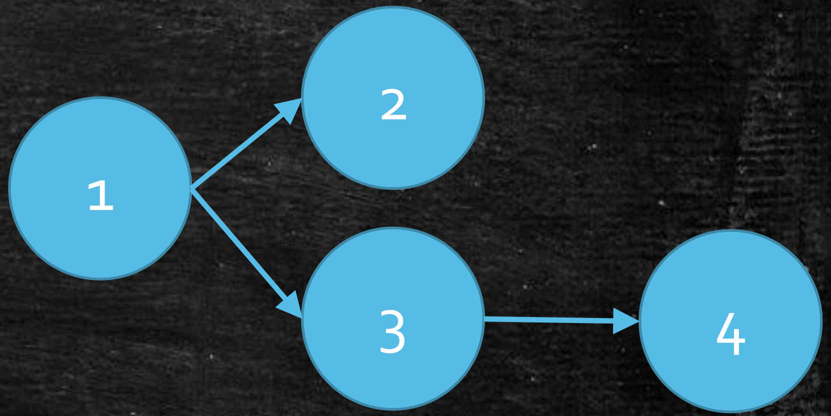
- Is DAG equals to a topological order?
- Known: not DAG -> no order
- Unknown: DAG -> an order
- How to prove?
- Design an algorithm do topological ordering for DAG.

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 cycle → we can not go back → we must stop at a tail.

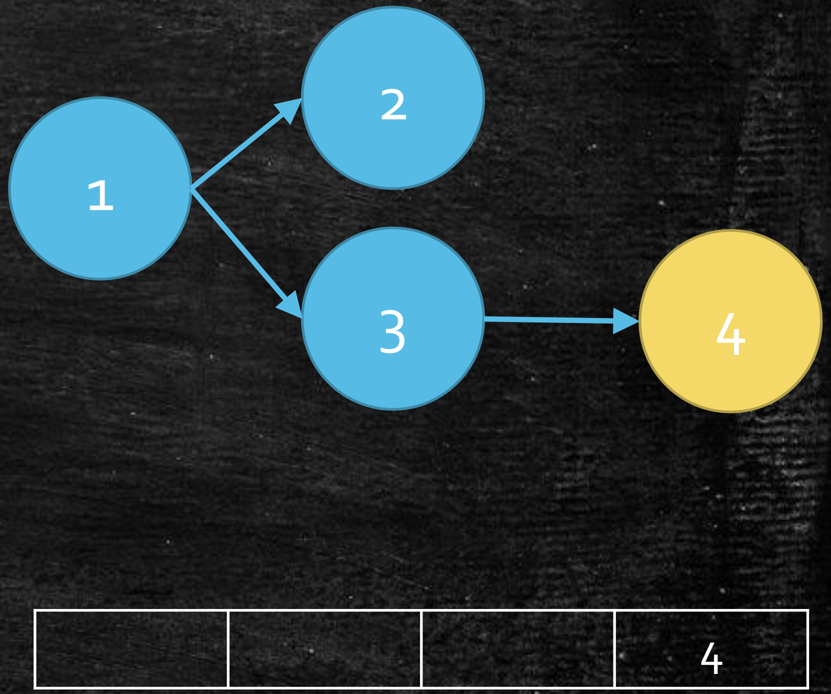
Topological Ordering for DAG

- 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...



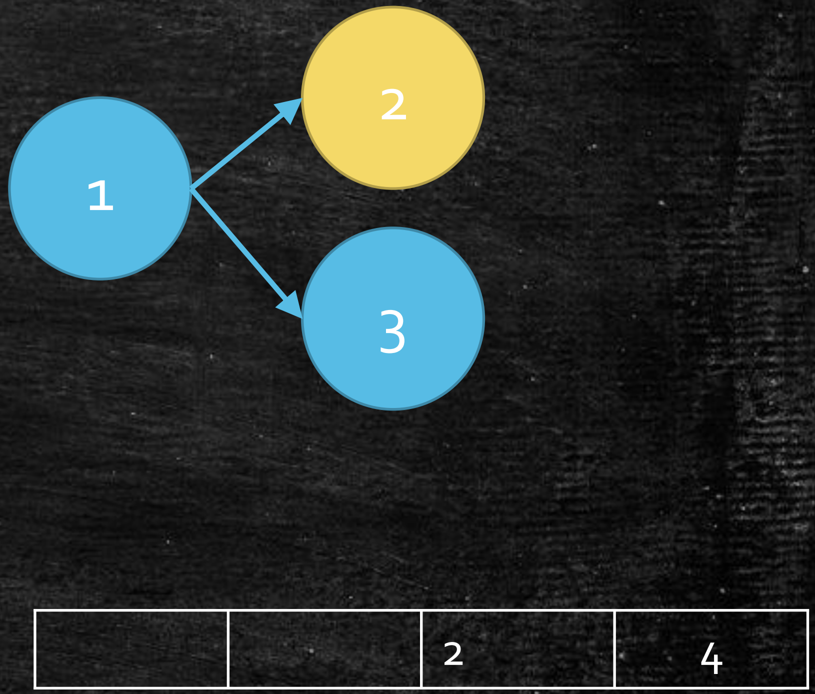
Topological Ordering for DAG

- 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…



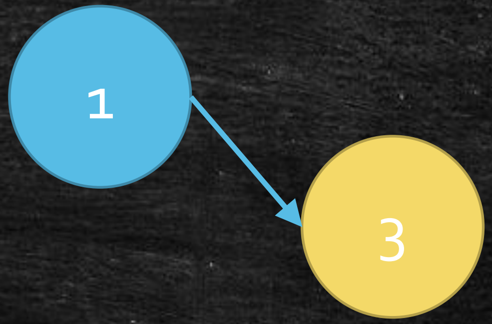
Topological Ordering for DAG

- 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…



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...

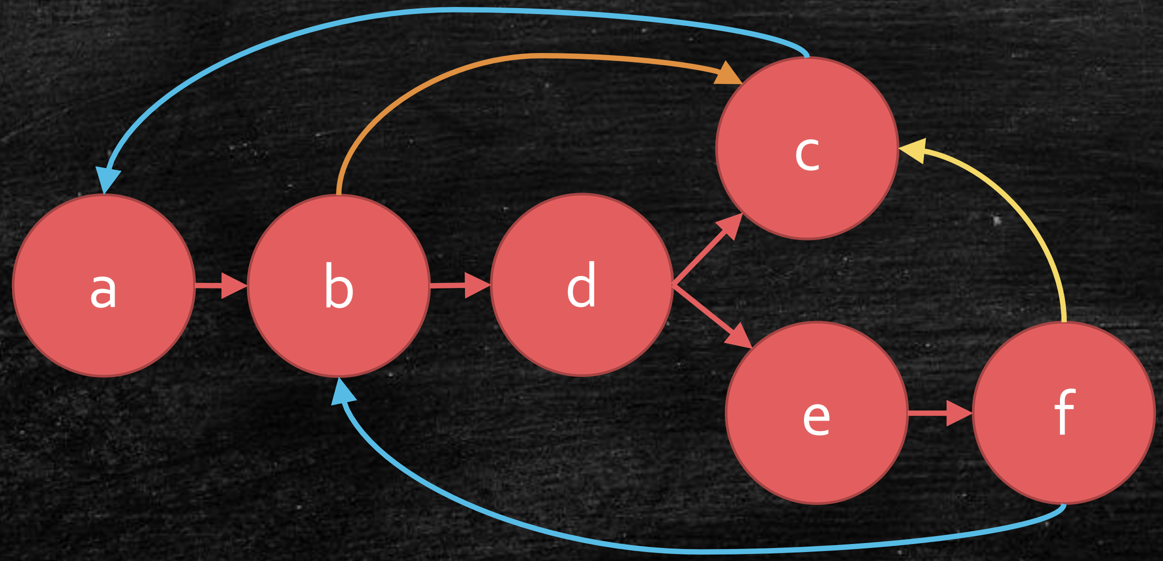
1	3	2	4

Running Time?

- Running Time
  - $|V|$ rounds
  - Find a tail: $O(|V|)$
  - Remove a tail update: $O(|V|)$
  - Total: $\boldsymbol{O(|V|^2)}$
- Is the order feasible?
- Conclusion
  - We can find a feasible topological order for DAG.
  - DAG ←→ A topological order

Improve it by DFS

- DFS tree for a DAG

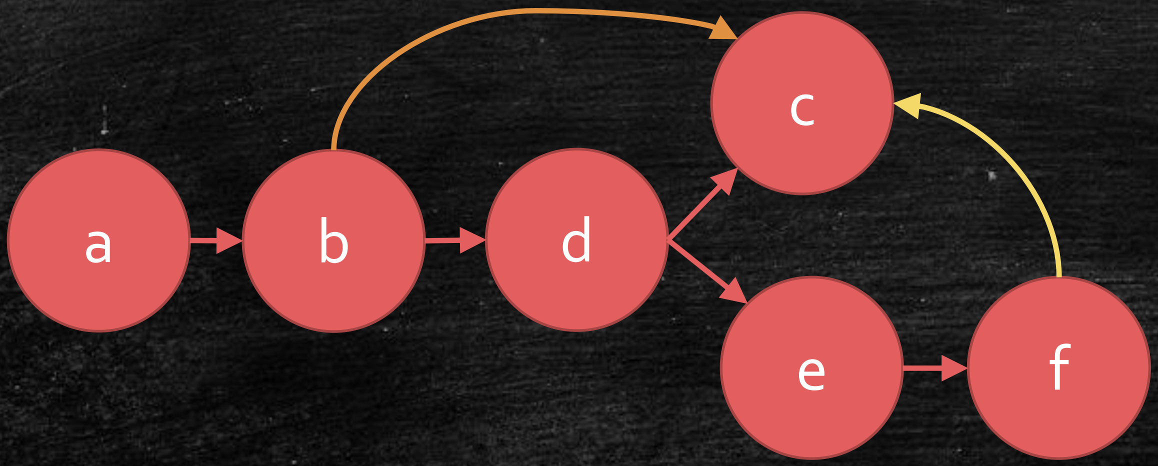


Function explore(v)
  $marked[v] \leftarrow true$
  for each $(u, v) \in E$
    if $marked[v] = false$
      explore(v)

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

Improve it by DFS

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

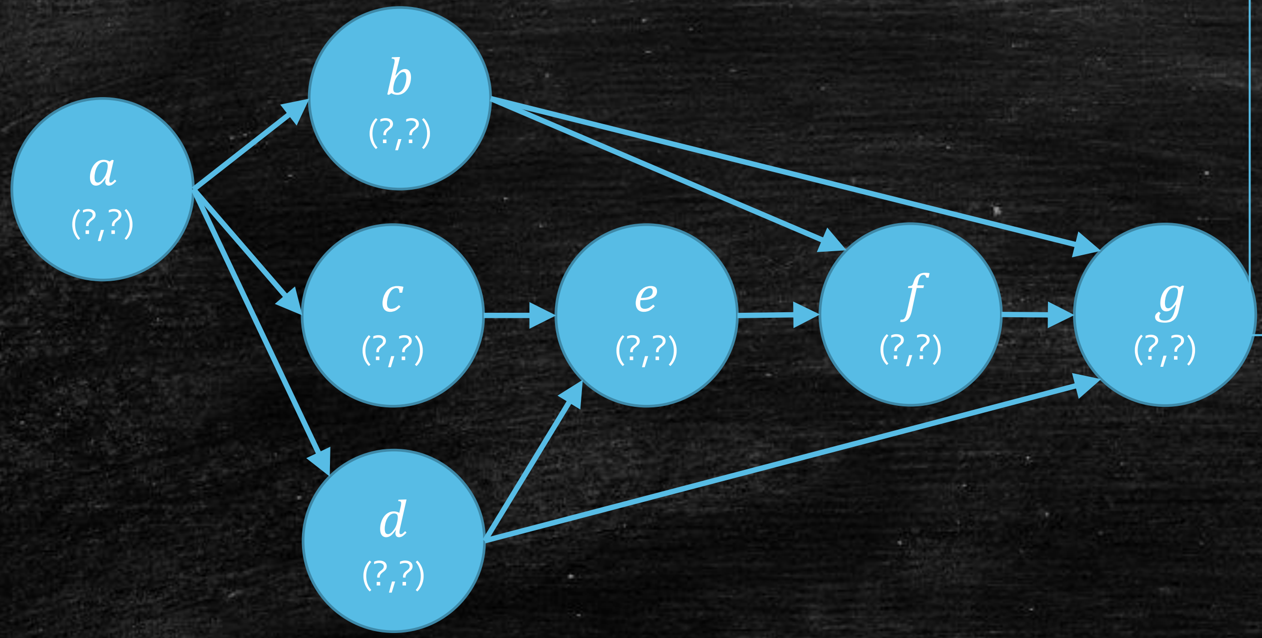


Function explore(v)
    $marked[v] \leftarrow true$
    for each $(u, v) \in E$
        if $marked[v] = false$
            explore(v)

- 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 $(u, v) \in E$

        if $marked[v] = false$

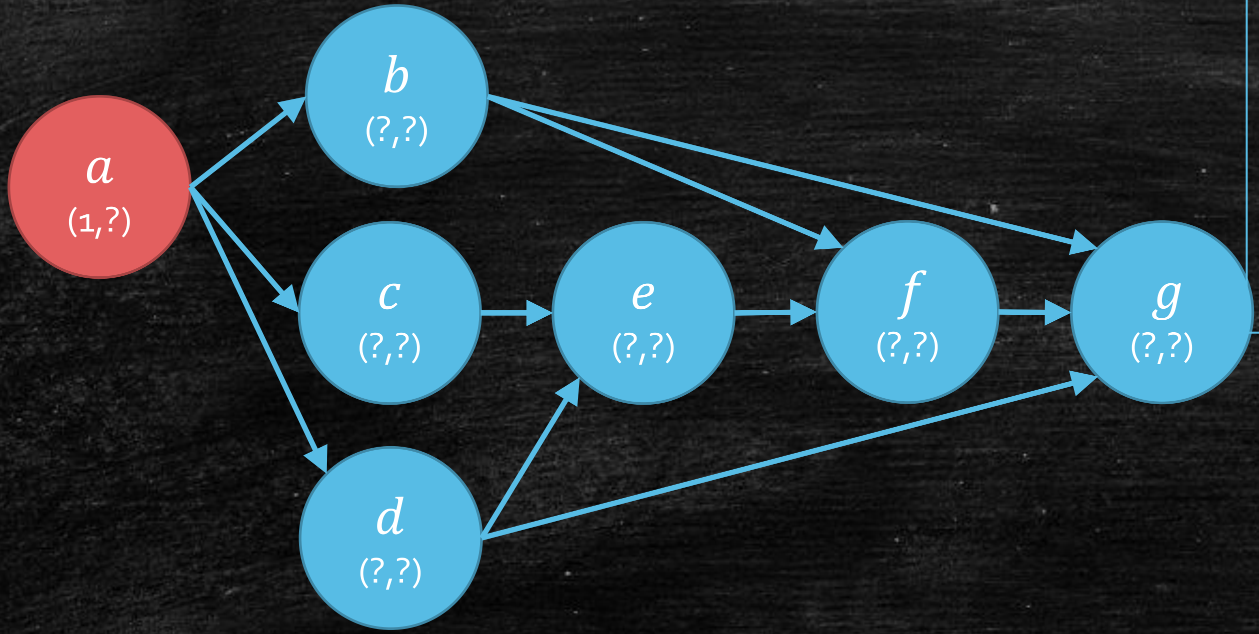
            explore(v)

    $finish[v] \leftarrow time$

    $time++$

Topological Ordering by DFS

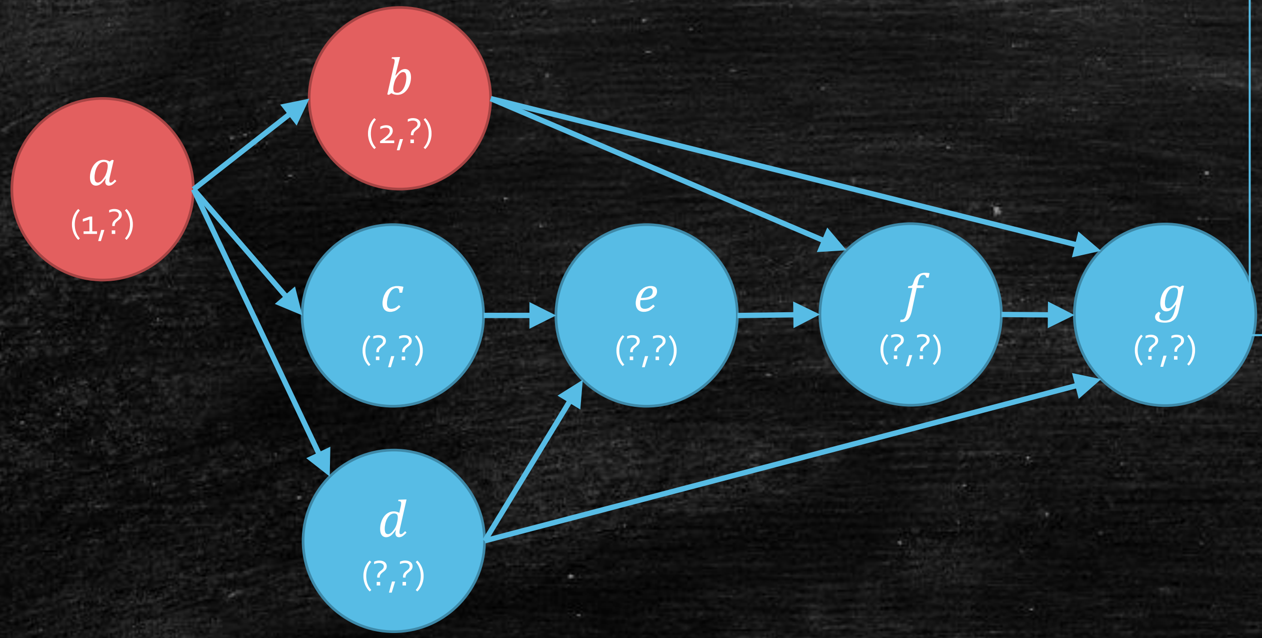
- Run DFS first!
- Record the start time and finish time.



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

Topological Ordering by DFS

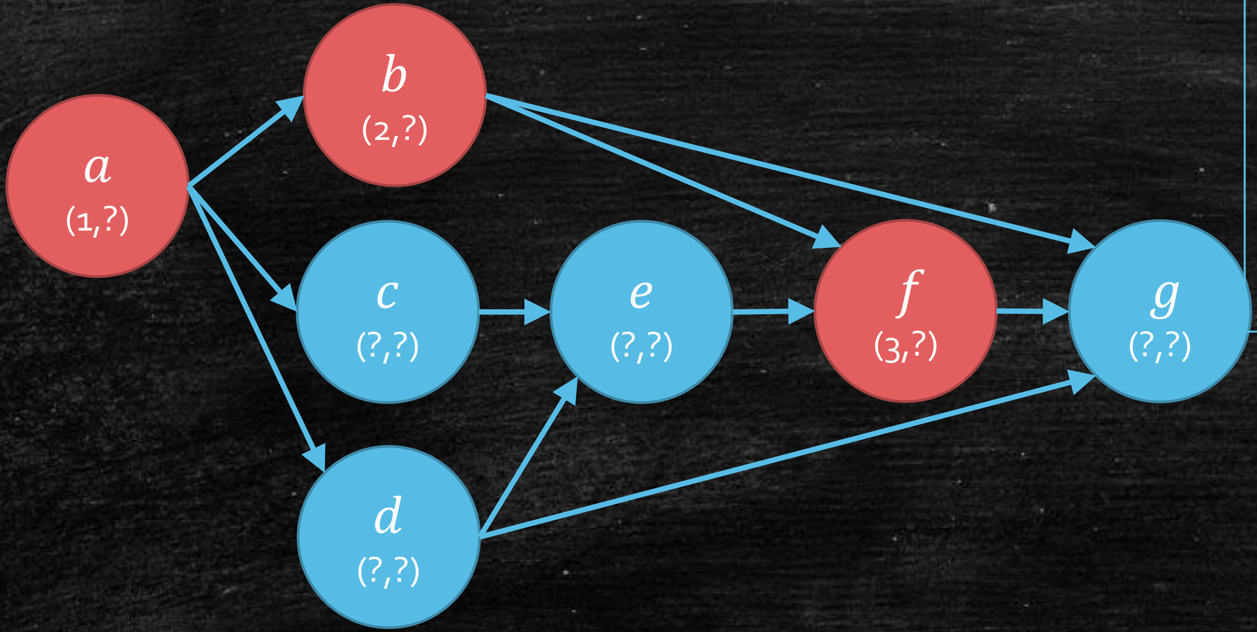
- Run DFS first!
- Record the start time and finish time.



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

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 $(u, v) \in E$

    if $marked[v] = false$

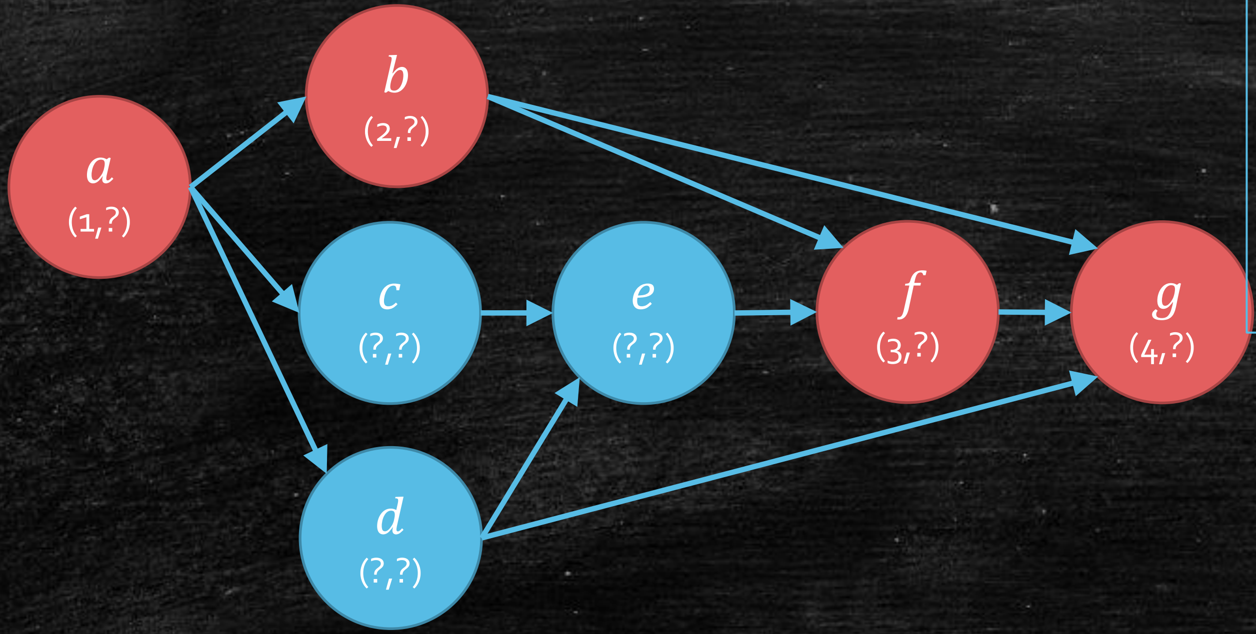
      explore(v)

  $finish[v] \leftarrow time$

  $time++$

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 $(u, v) \in E$

    if $marked[v] = false$

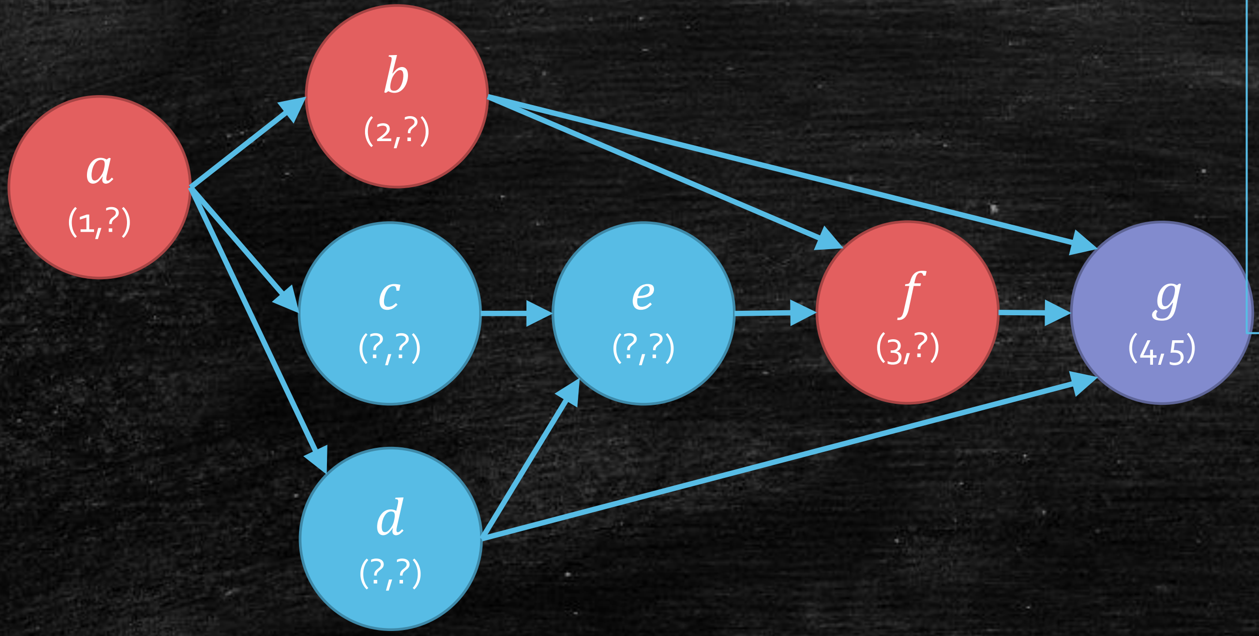
      explore(v)

  $finish[v] \leftarrow time$

  $time++$

Topological Ordering by DFS

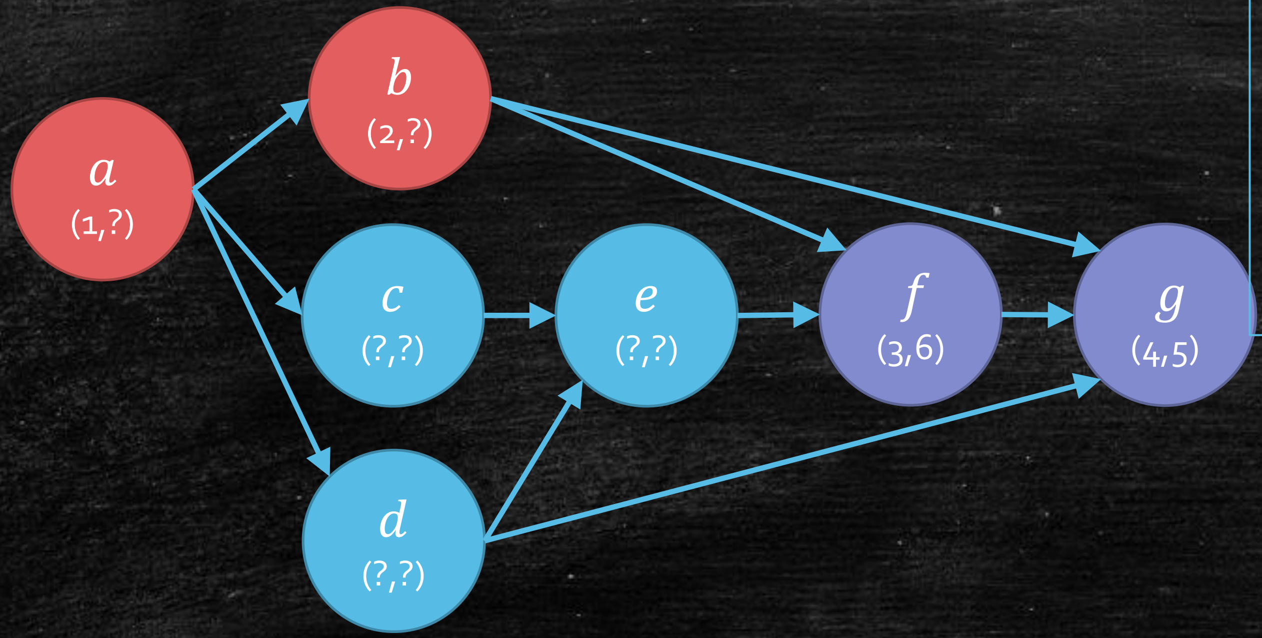
- Run DFS first!
- Record the start time and finish time.



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

Topological Ordering by DFS

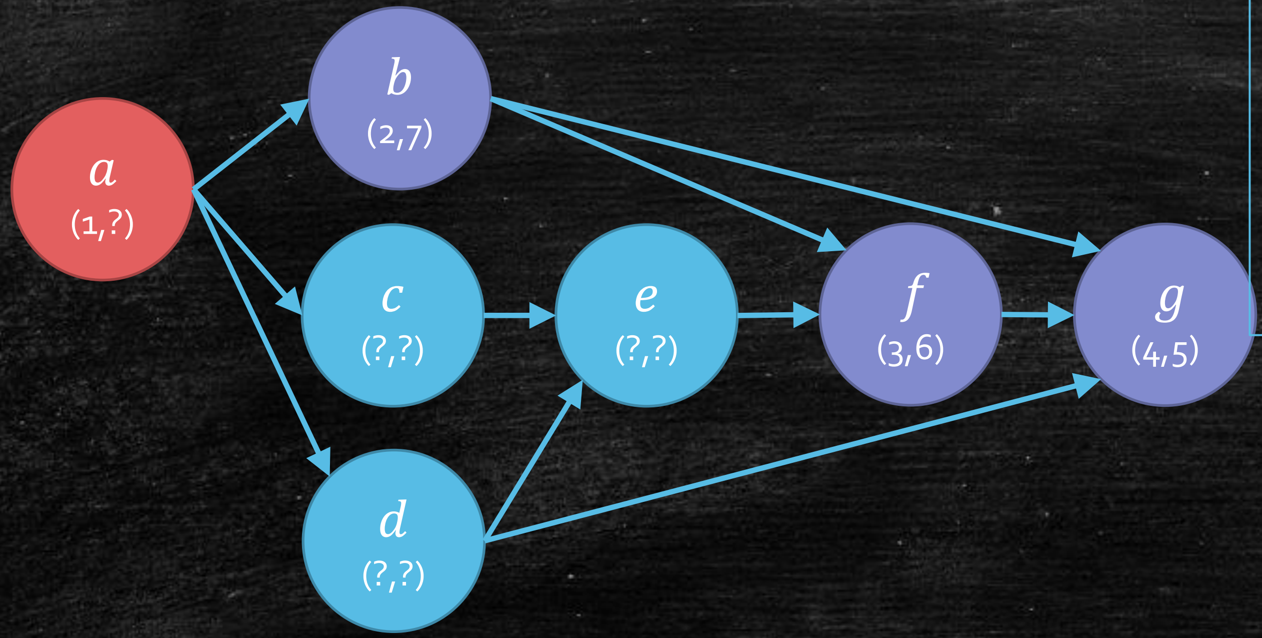
- Run DFS first!
- Record the start time and finish time.



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

Topological Ordering by DFS

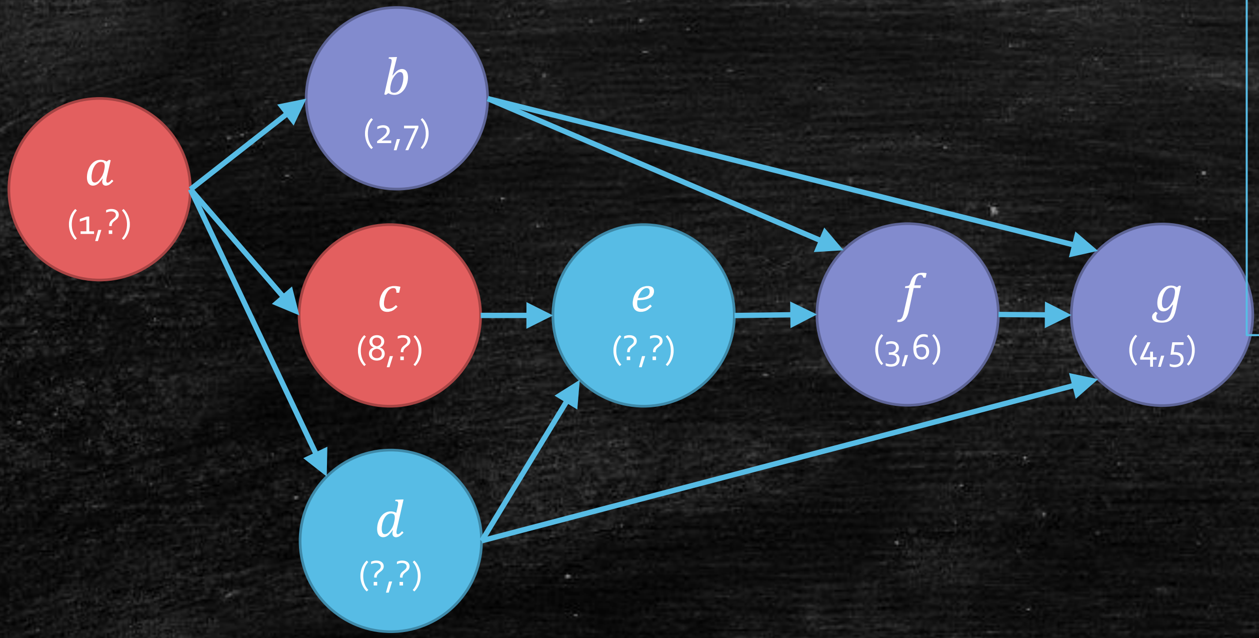
- Run DFS first!
- Record the start time and finish time.



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

Topological Ordering by DFS

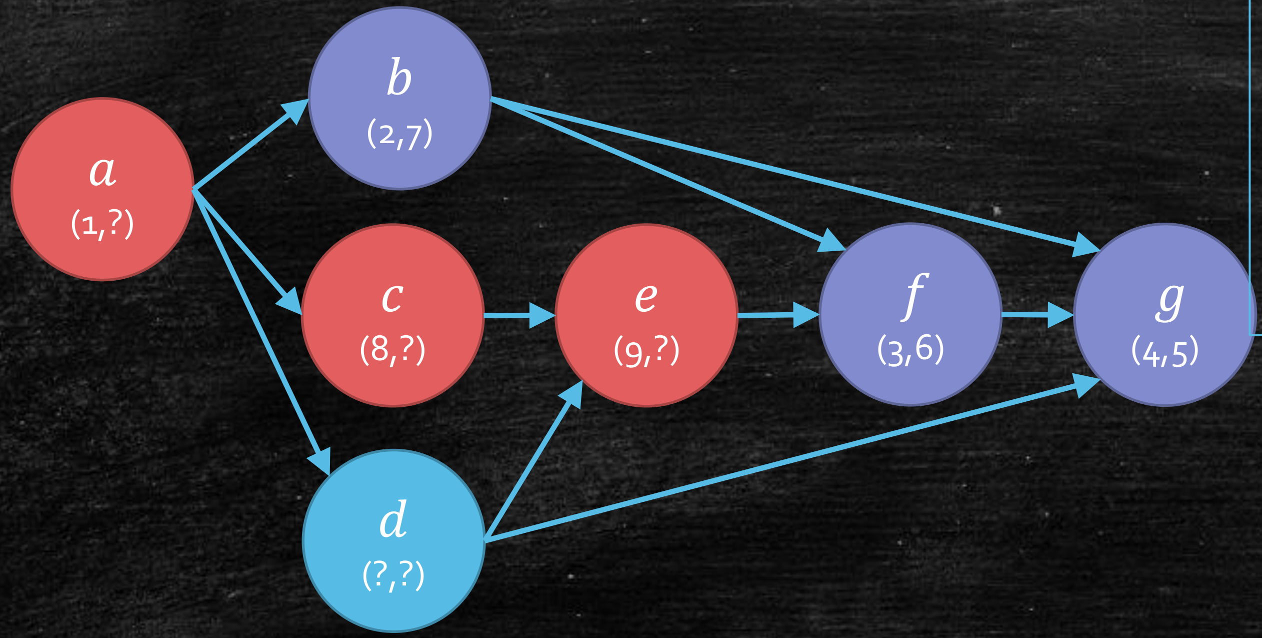
- Run DFS first!
- Record the start time and finish time.



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

Topological Ordering by DFS

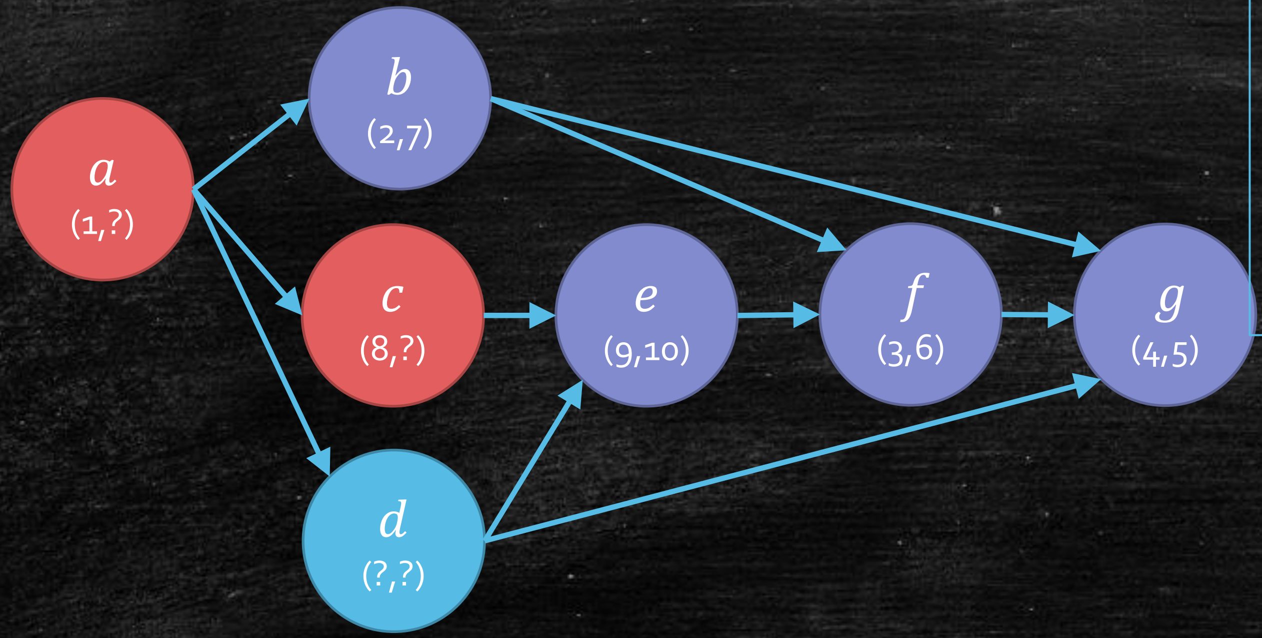
- Run DFS first!
- Record the start time and finish time.



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

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 $(u, v) \in E$

    if $marked[v] = false$

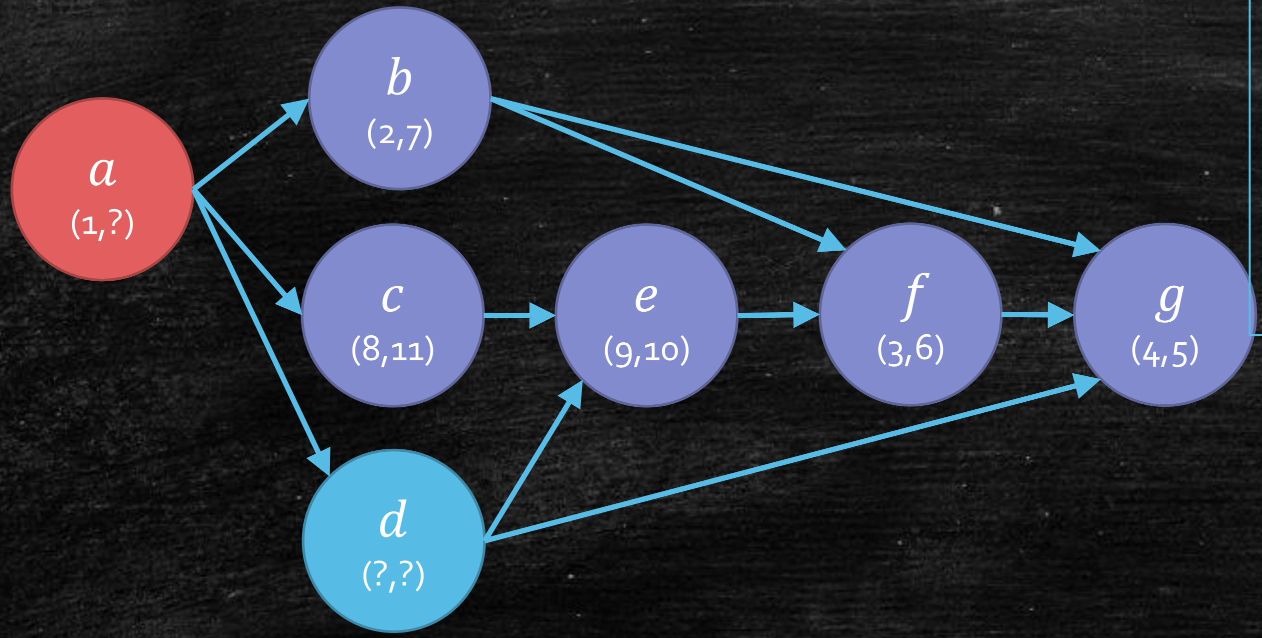
      explore(v)

  $finish[v] \leftarrow time$

  $time++$

Topological Ordering by DFS

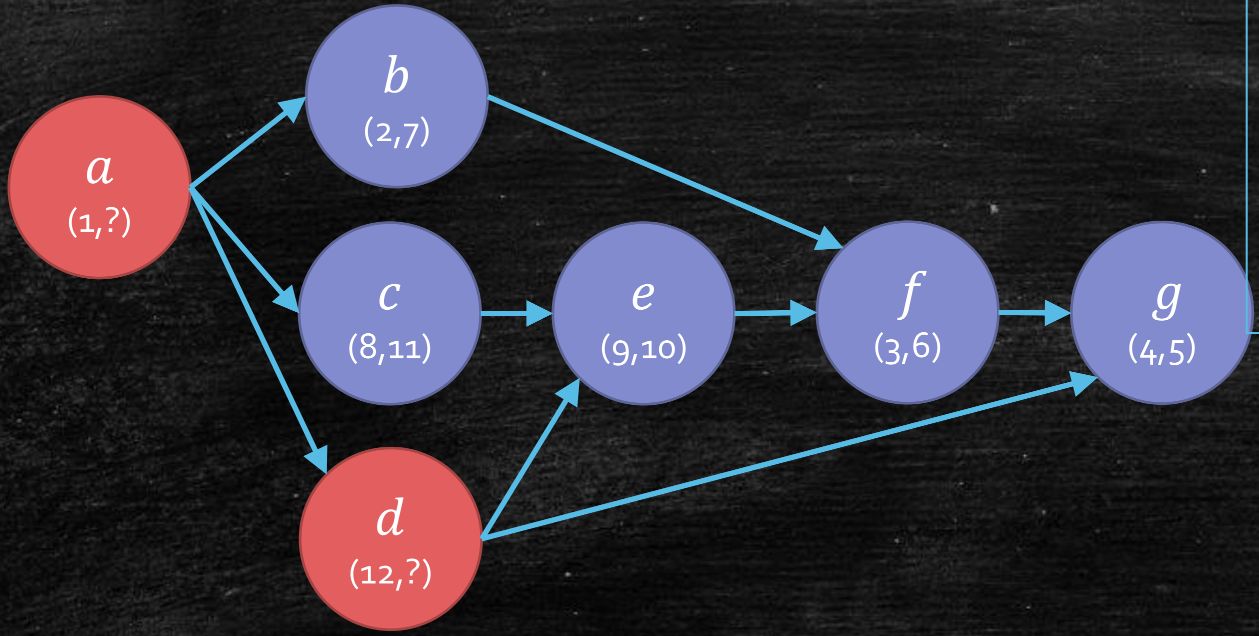
- Run DFS first!
- Record the start time and finish time.



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

Topological Ordering by DFS

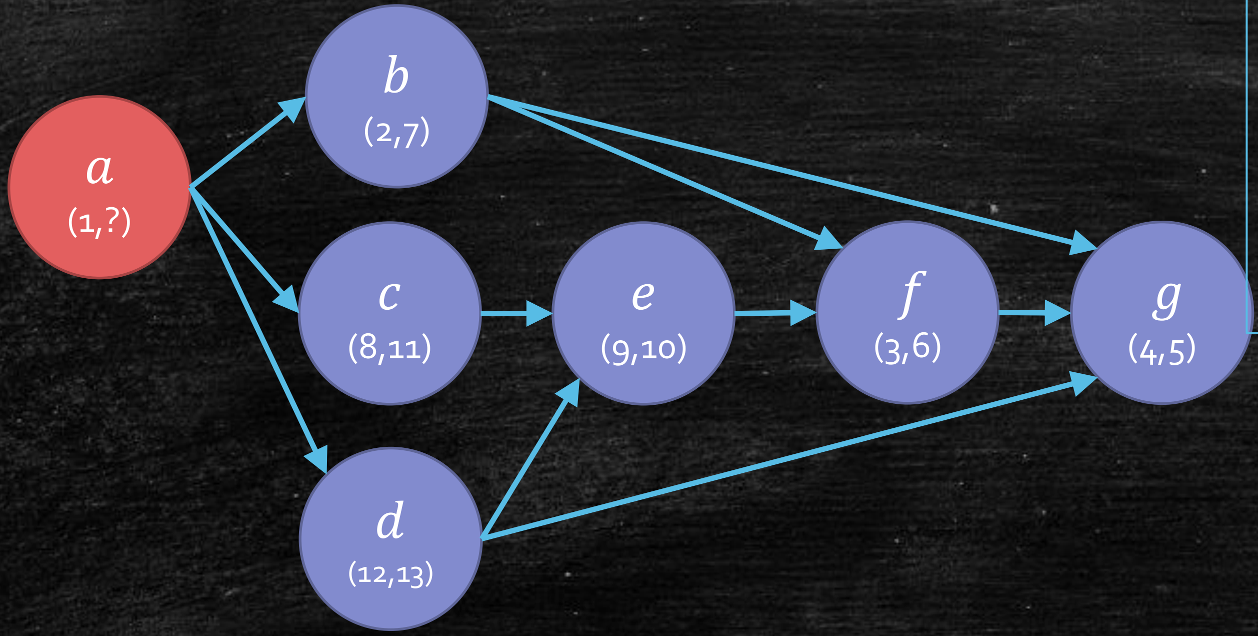
- Run DFS first!
- Record the start time and finish time.



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

Topological Ordering by DFS

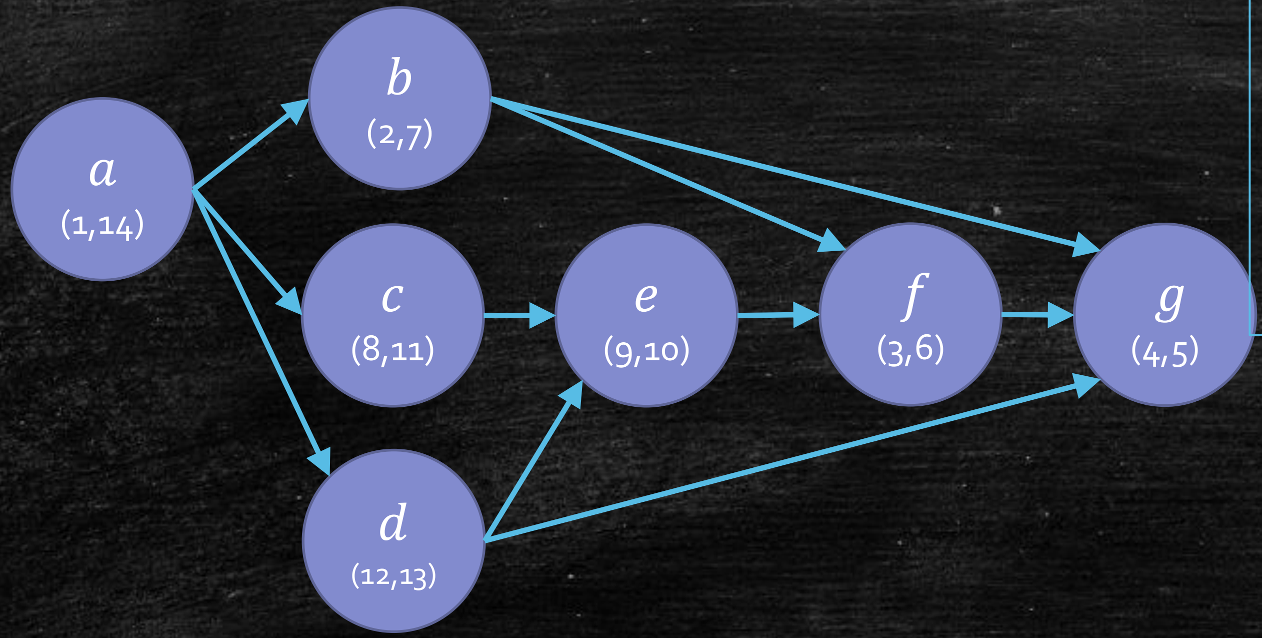
- Run DFS first!
- Record the start time and finish time.



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

Topological Ordering by DFS

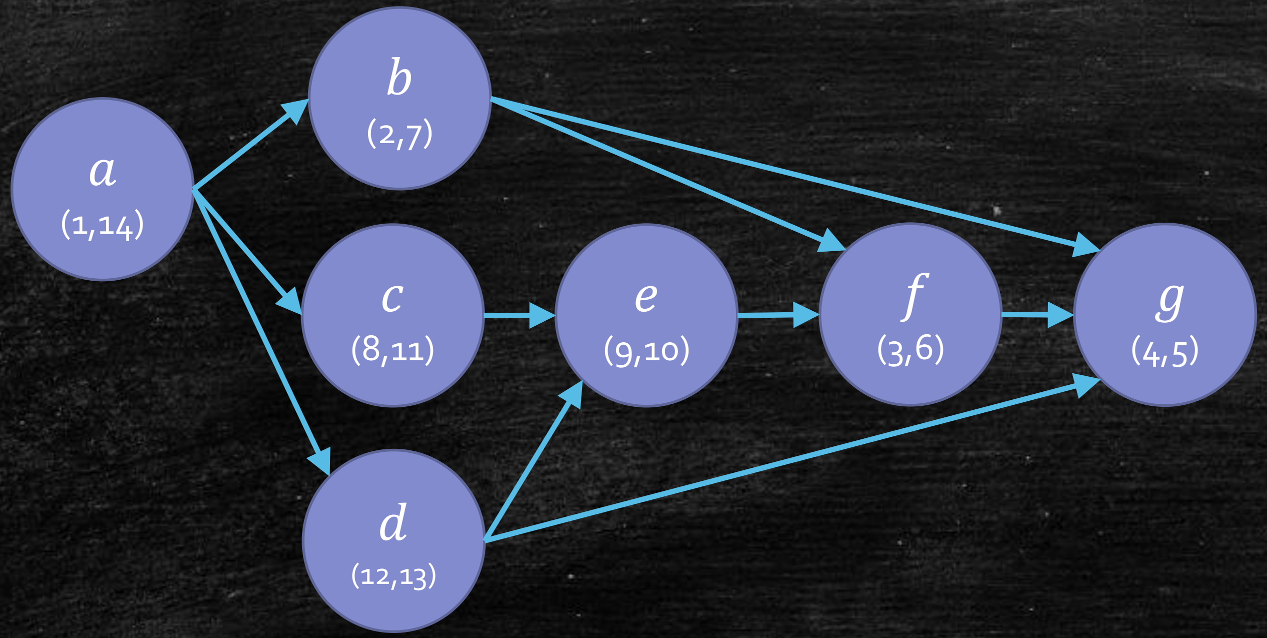
- Run DFS first!
- Record the start time and finish time.



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

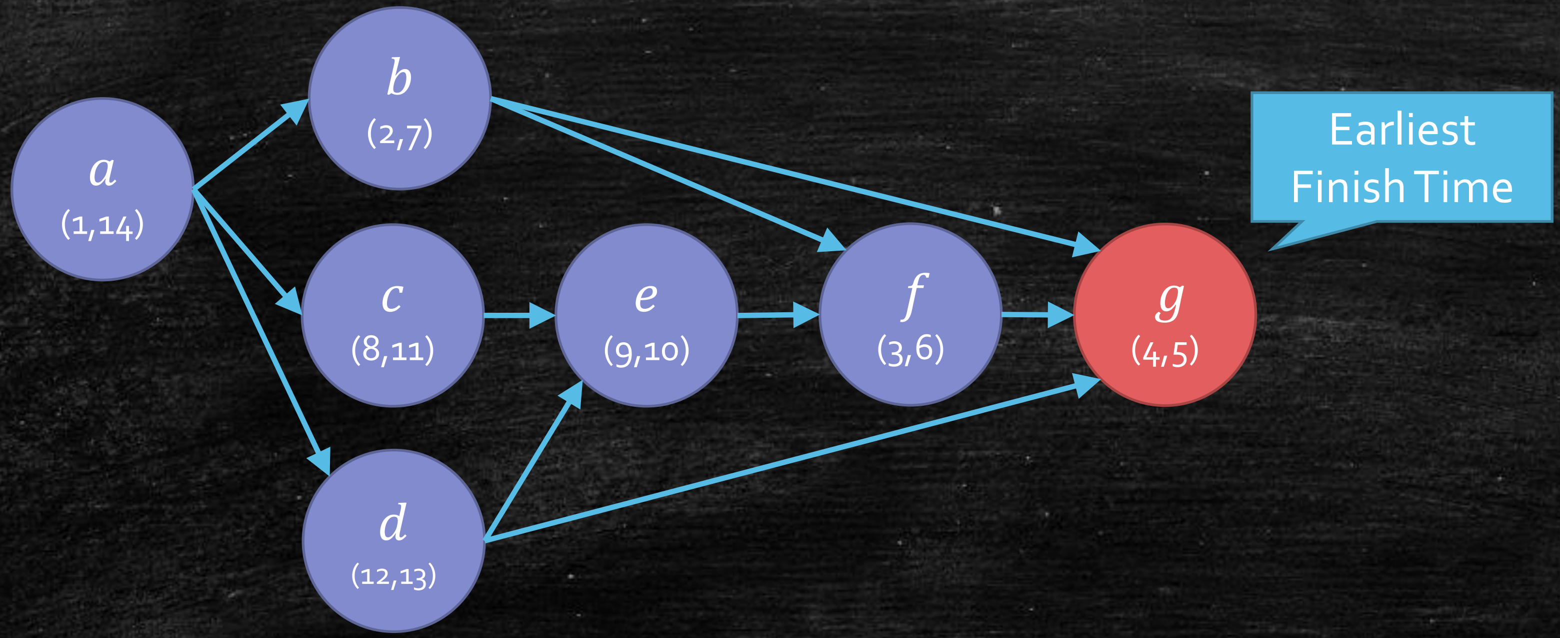
Discussion

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



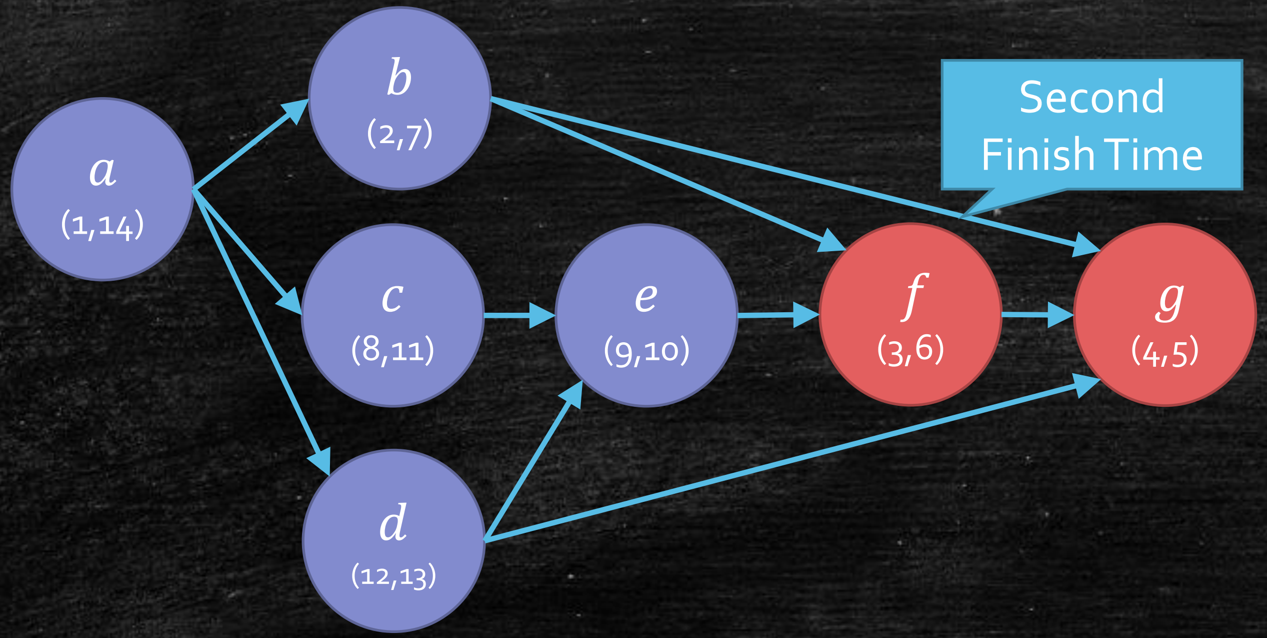
Discussion

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



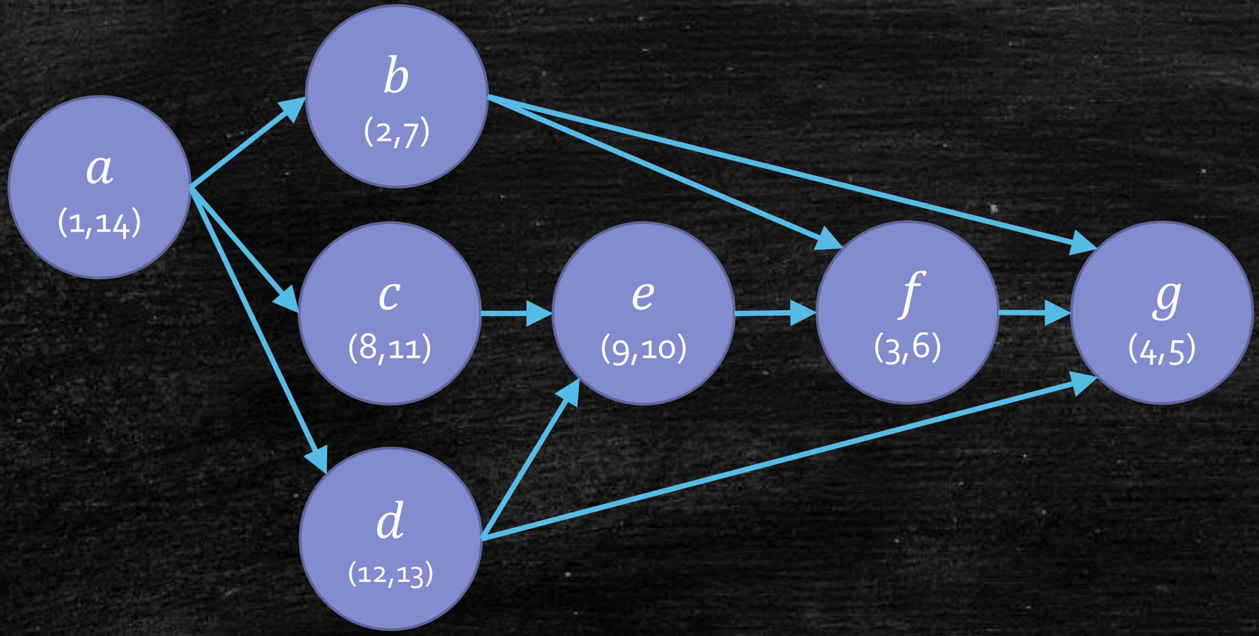
Discussion

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



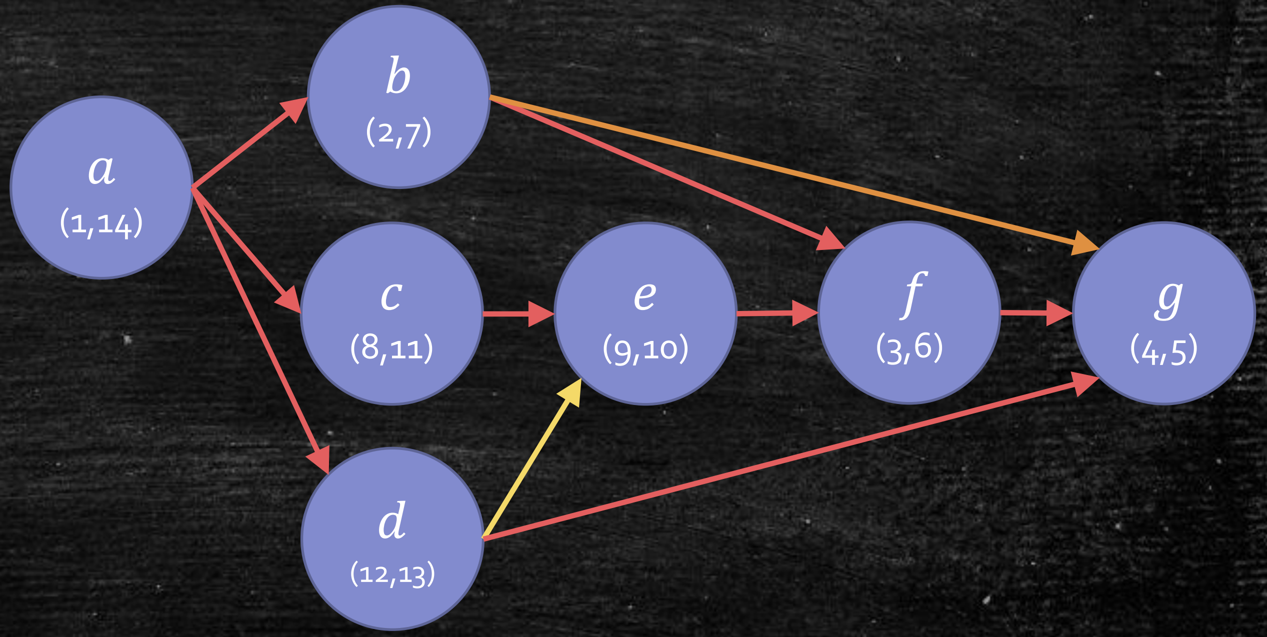
Conjecture

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



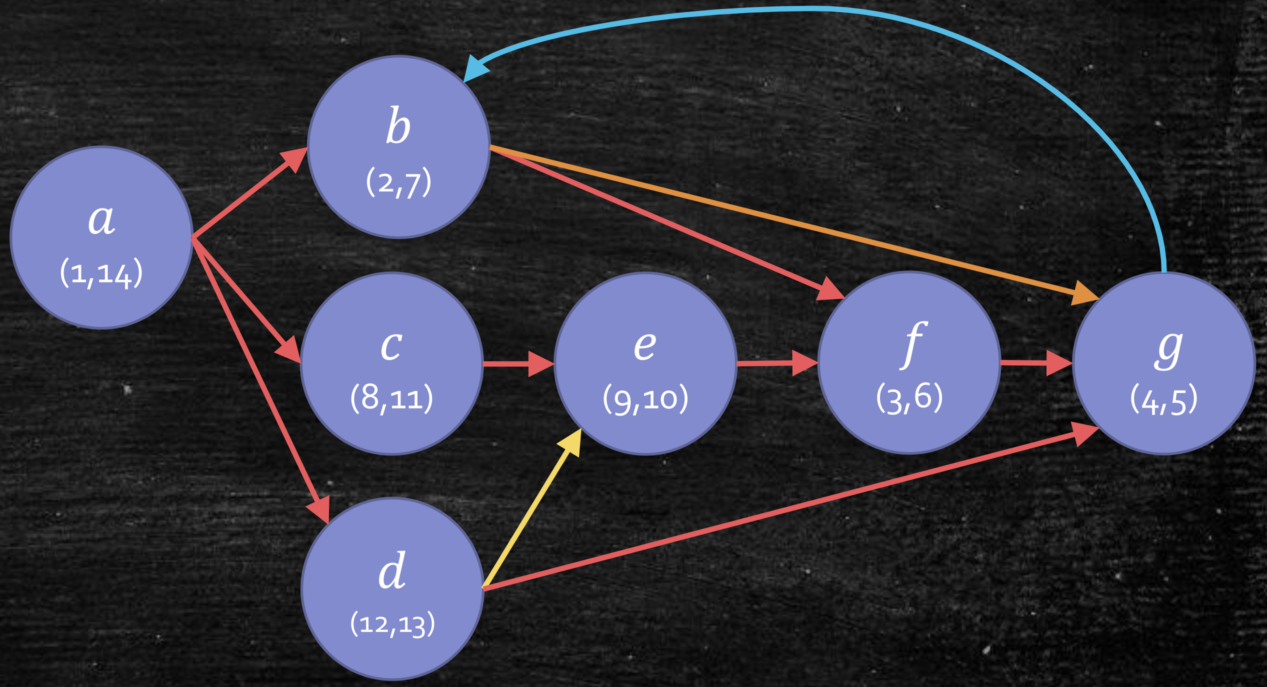
Prove the conjecture

- 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

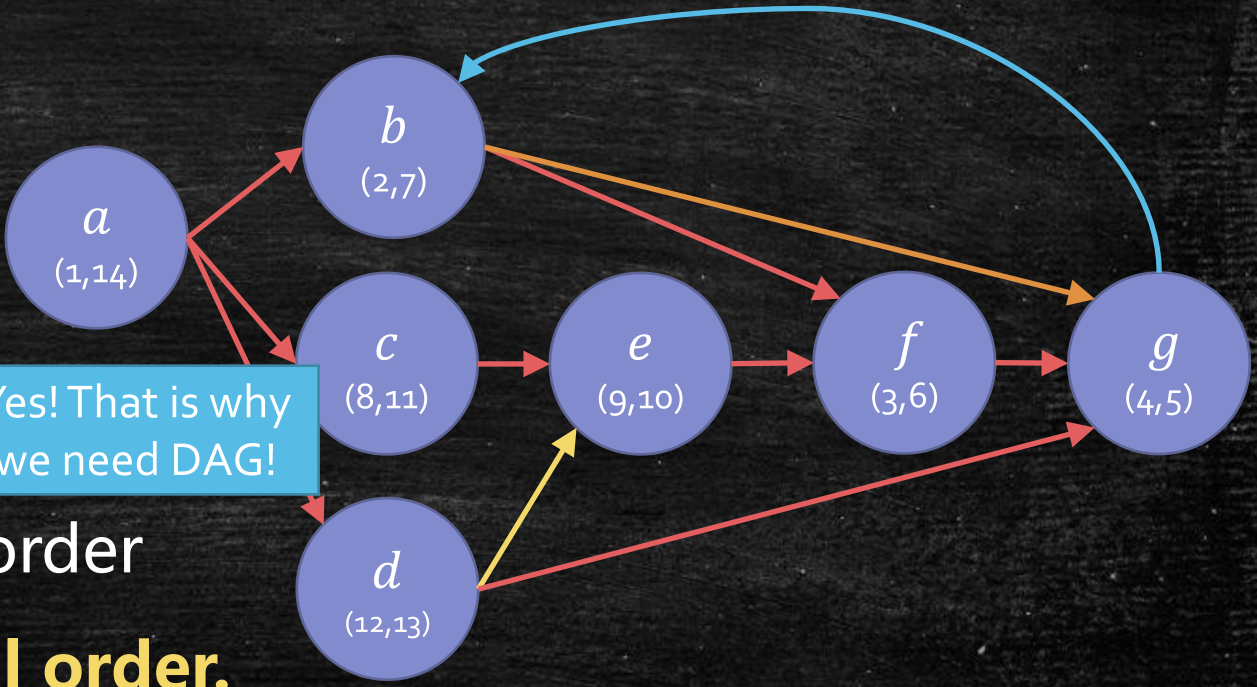
- Claim: no arc (u, v), if $finish[v] > finish[u]$.
- Proof:
  - If (u, v) exists,
  - ~~Can (u, v) be a tree edge~~?
  - ~~Can (u, v) be a forward edge~~?
  - ~~Can (u, v) be a cross edge~~?
  - Can (u, v) be a back edge?



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

- $O(|V|\log|V| + |E|)$?
  - 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!
- $\boldsymbol{O}(|\boldsymbol{V}| + |\boldsymbol{E}|)$?

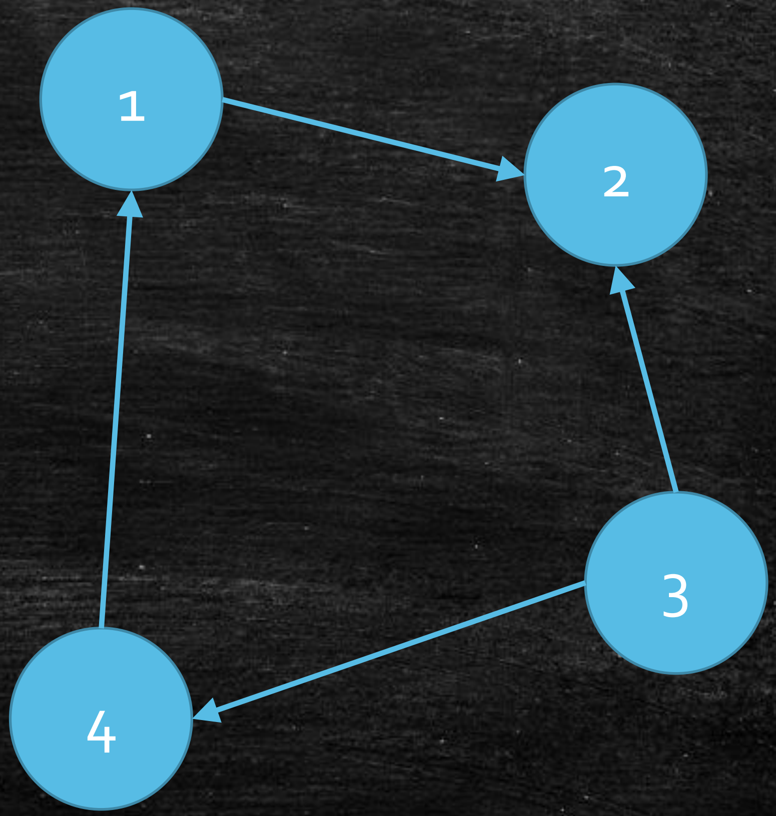
Connectivity in Directed Graphs

Recall

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

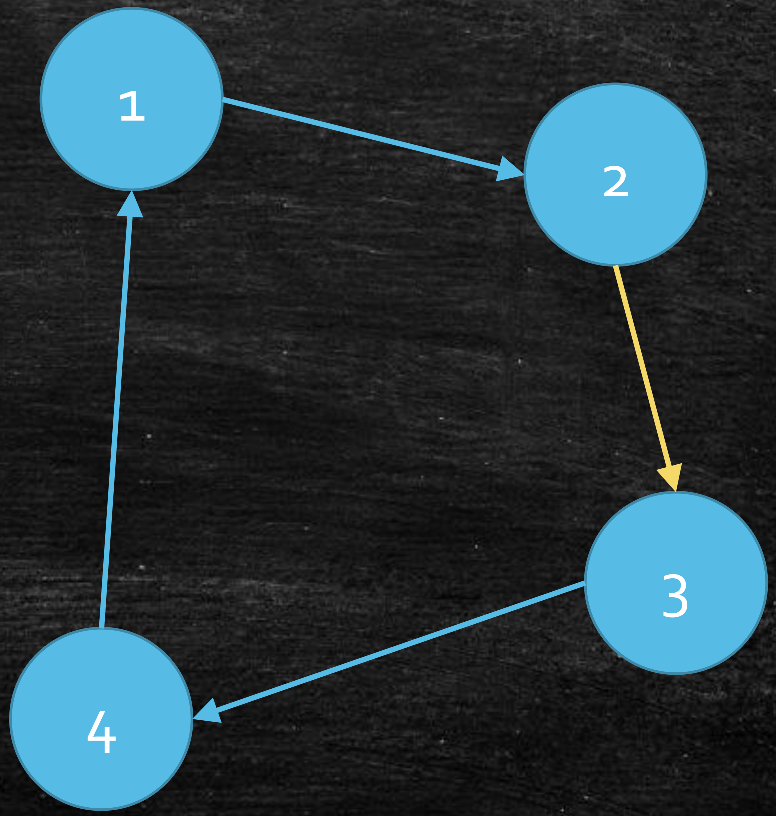
Connect Components in Directed Graphs

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



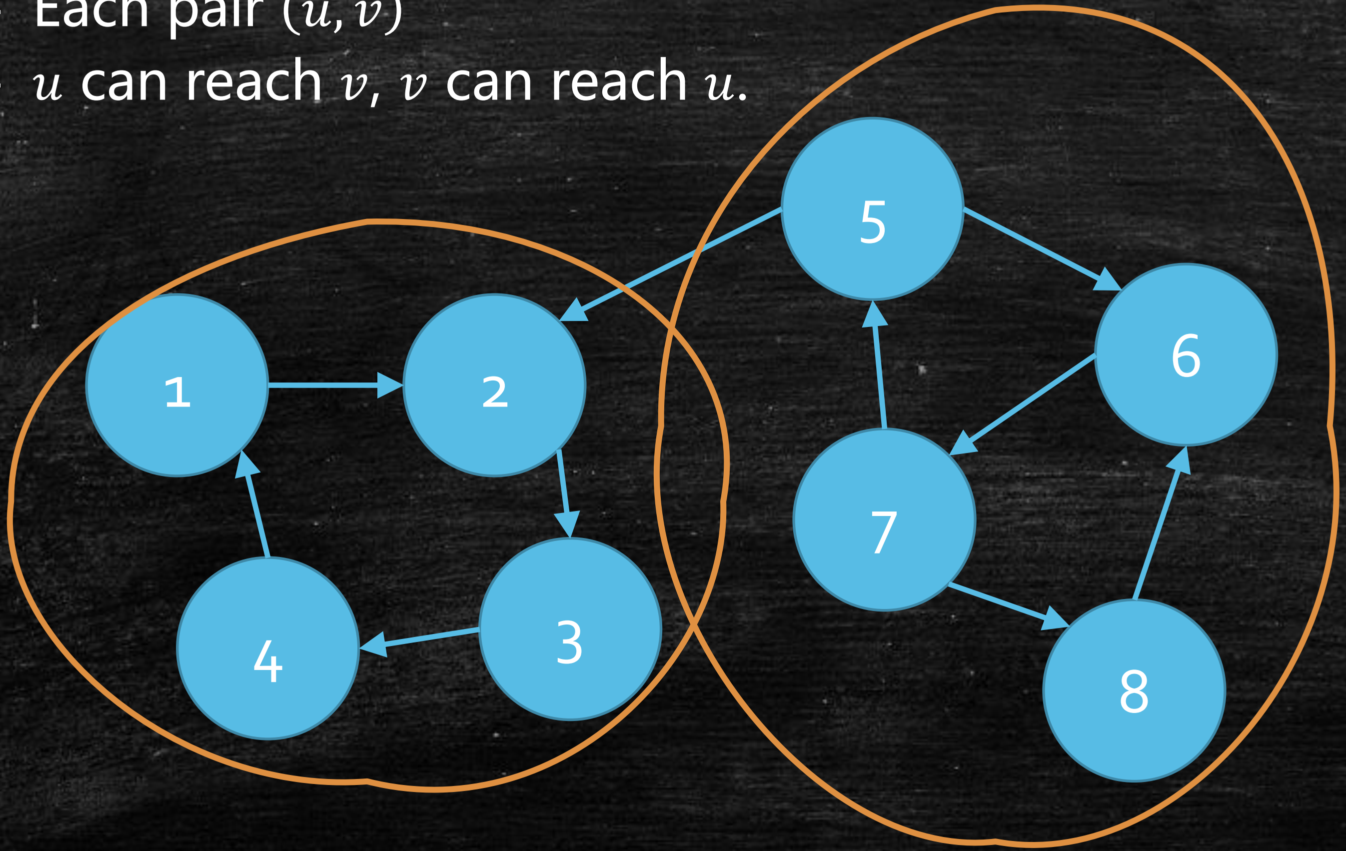
Connect Components in Directed Graphs

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



Strongly Connected Component (SCC)

- The **maximal** subset of vertices
  - Each pair (u, v)
  - u can reach v, v can reach u.



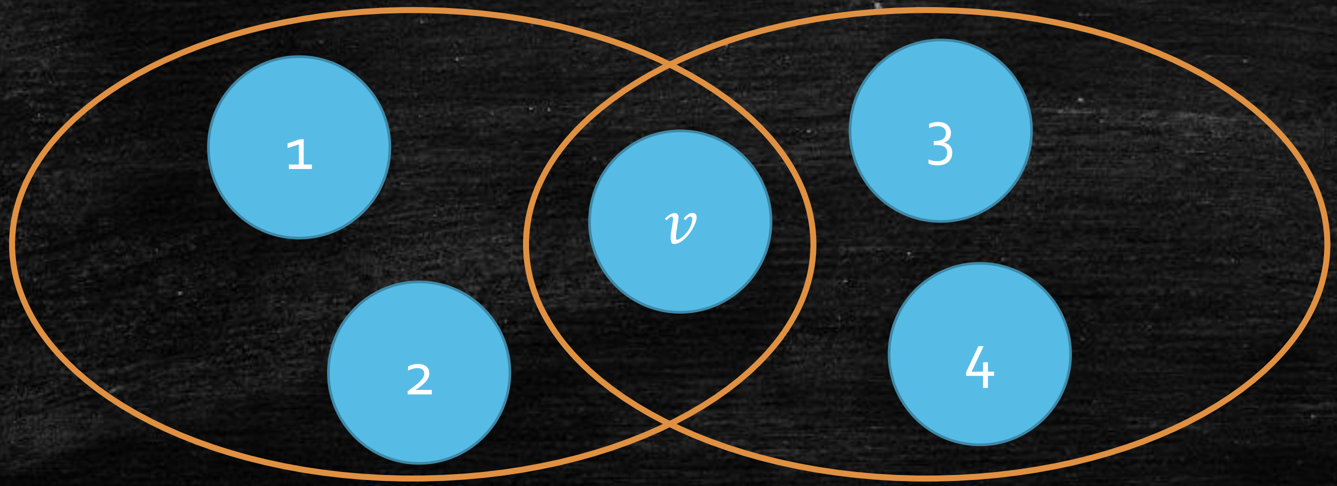
Is SCCs a Partition?

Claim

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

Proof

- $\rightarrow$: 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.

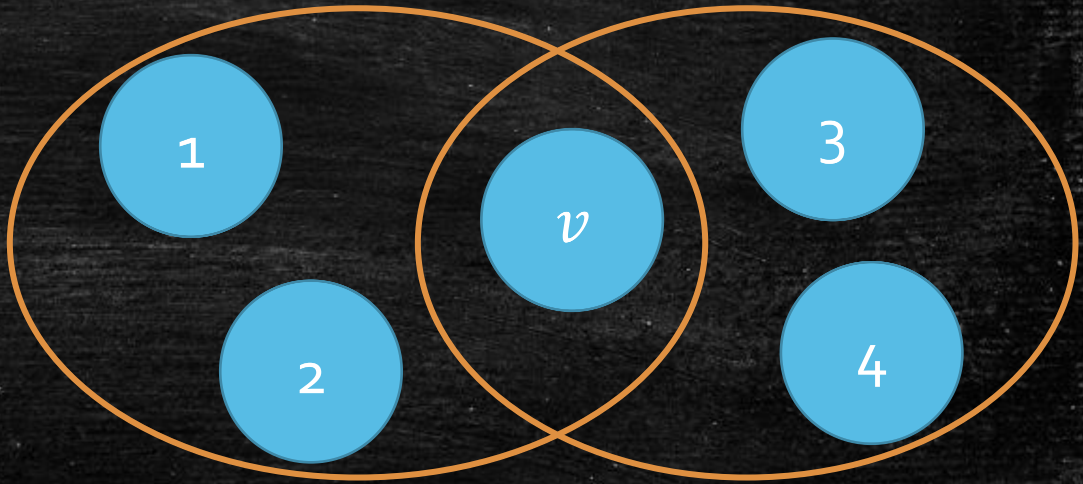


One more property of strongly connected

- **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 $a \rightarrow b$ and $b \rightarrow a$.
  - We have path $b \rightarrow c$ and $c \rightarrow b$.
  - So, we have path $a \rightarrow b \rightarrow c$.
  - So, we have path $c \rightarrow b \rightarrow a$.
- 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!**

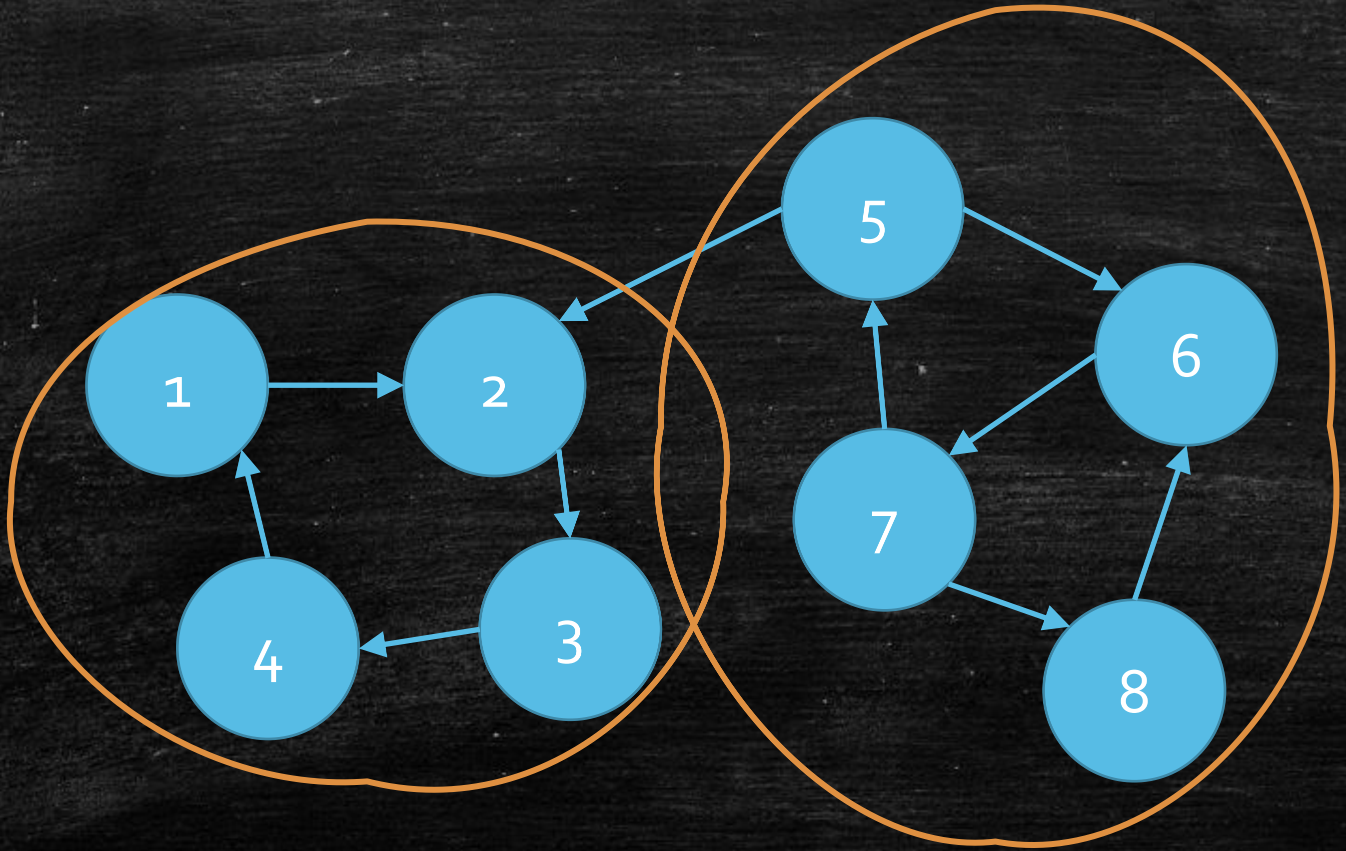


The set of SCCs forms a Partition of V!

Can we use DFS to find SCCs?

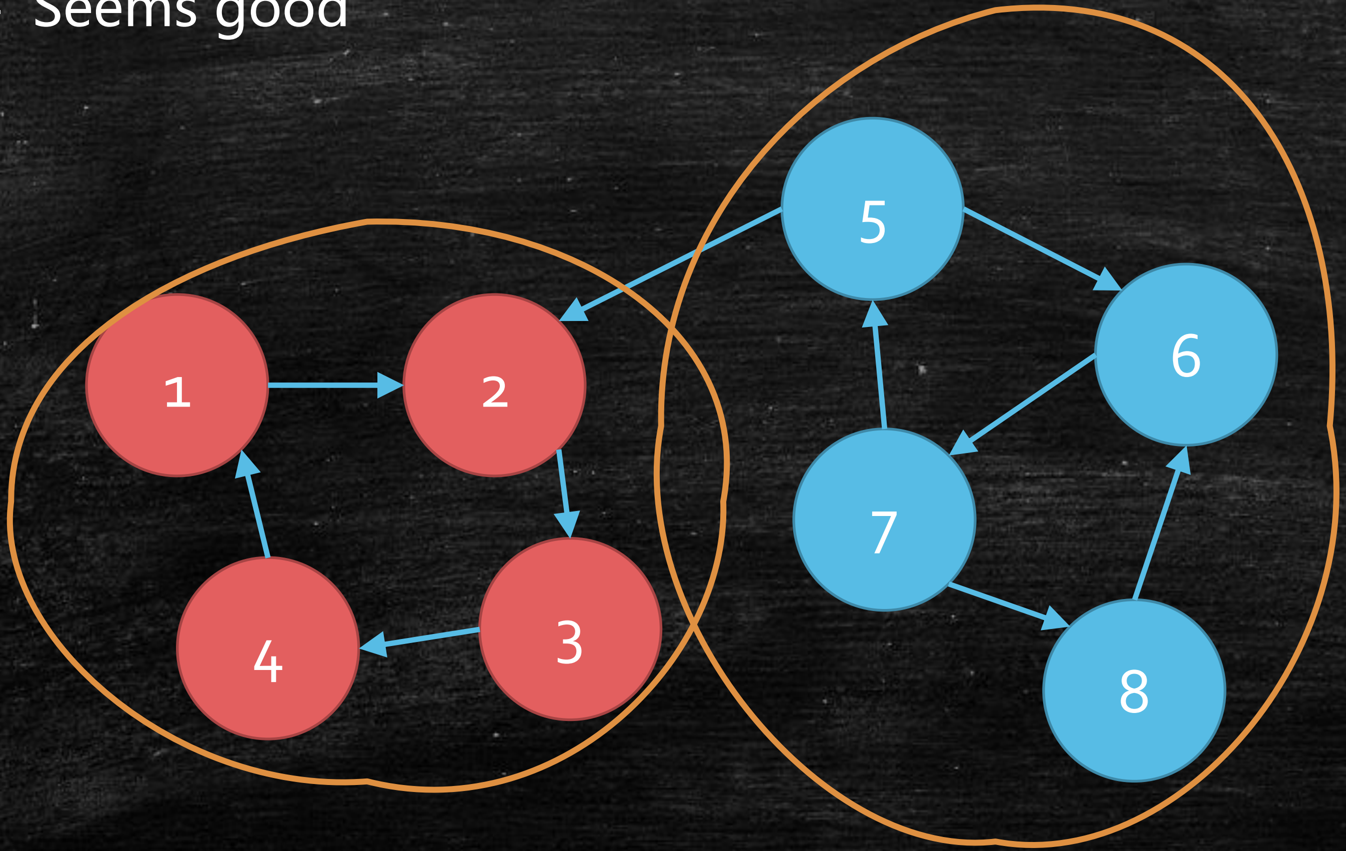
A Simple Attempt

- Start DFS from vertex 1.



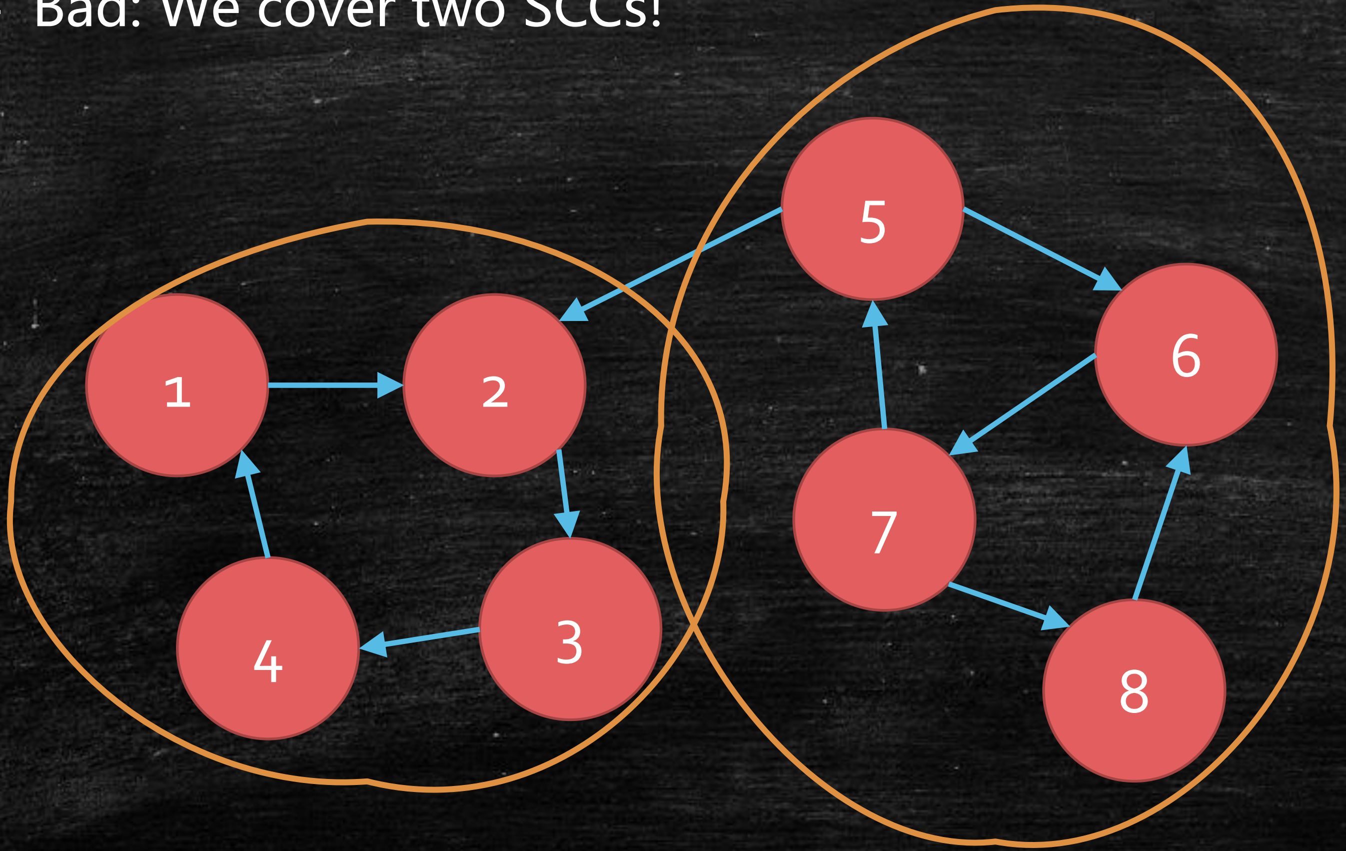
A Simple Attempt

- Start DFS from vertex 1.
  - Seems good



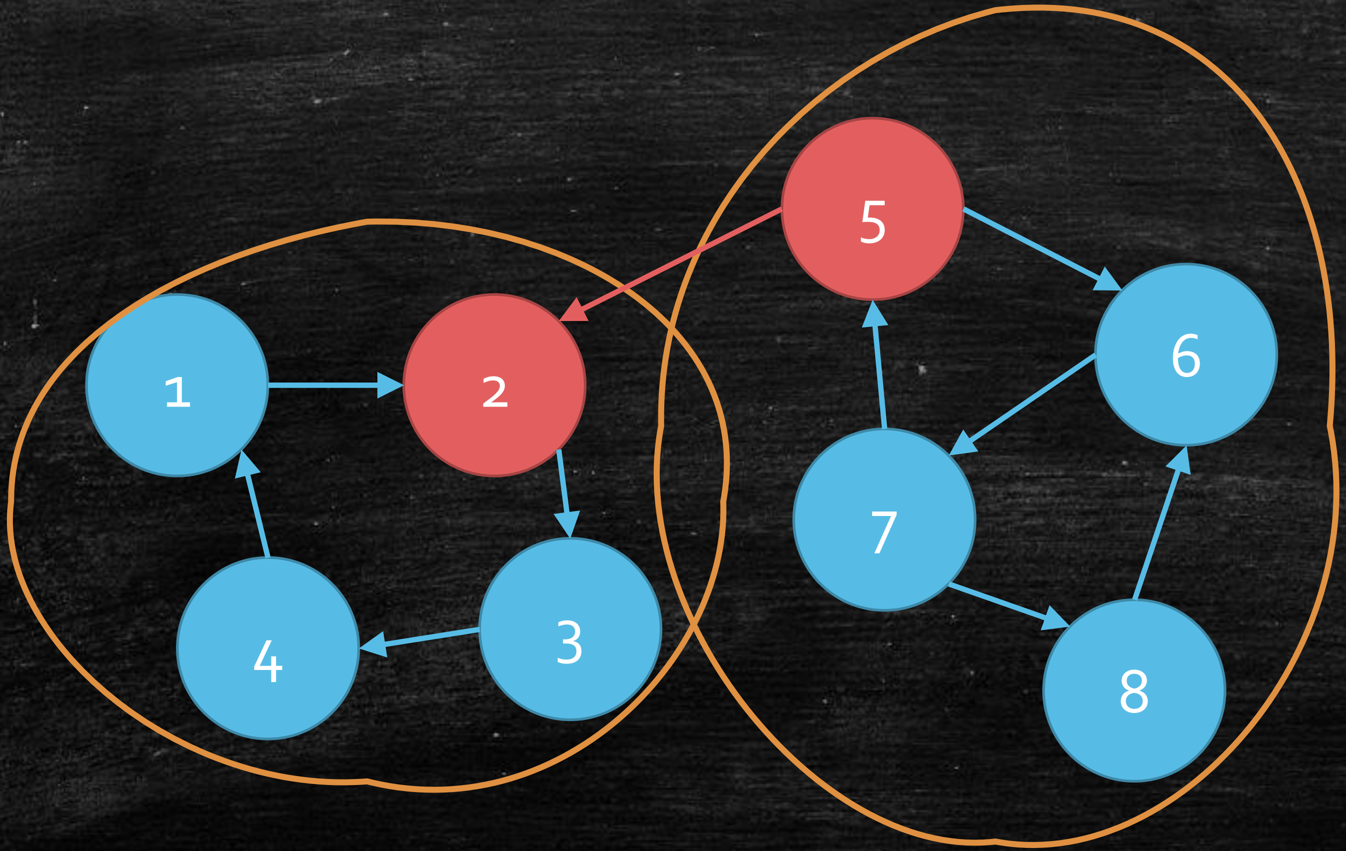
A Simple Attempt

- Start DFS from vertex 5.
  - Bad: We cover two SCCs!



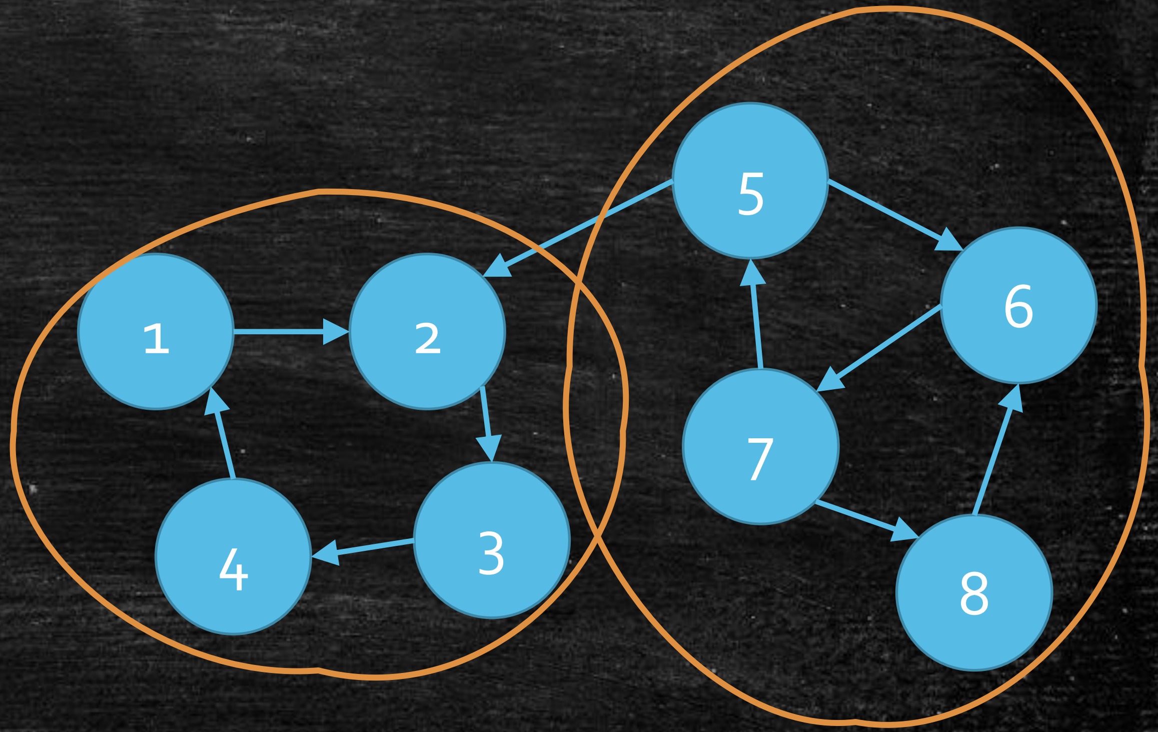
What is the trouble?

- Trouble: going out of the SCC



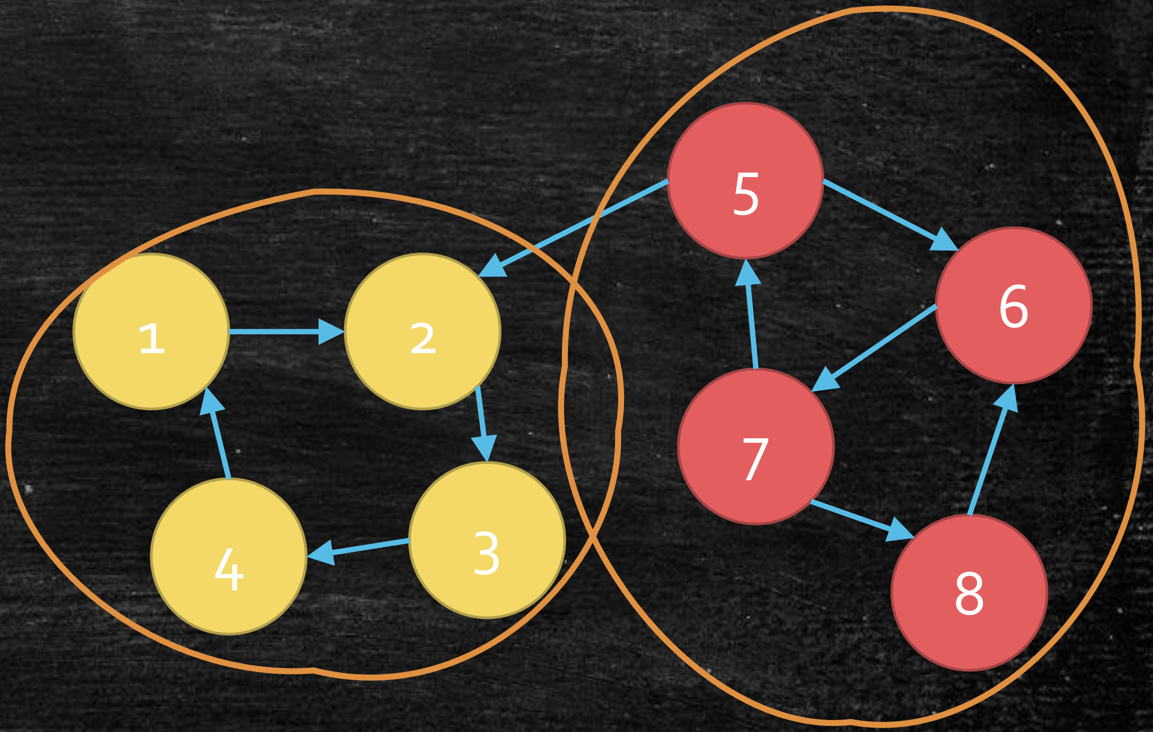
Question: can we handle it?

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



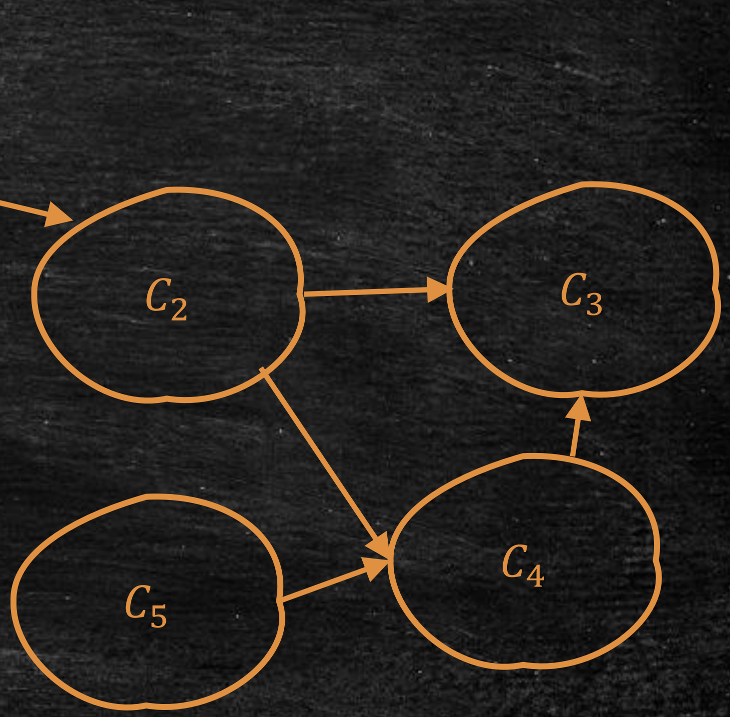
Question: can we handle it?

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



Does such SCC exist?

- Move to a big picture
  - Let SCCs be Super Nodes.
  - Vertices inside are somehow equivalent.
  - (C_i, C_j) exists $\leftarrow\rightarrow$ (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**.
  - **Corollary: the SCC Graph is a DAG!**

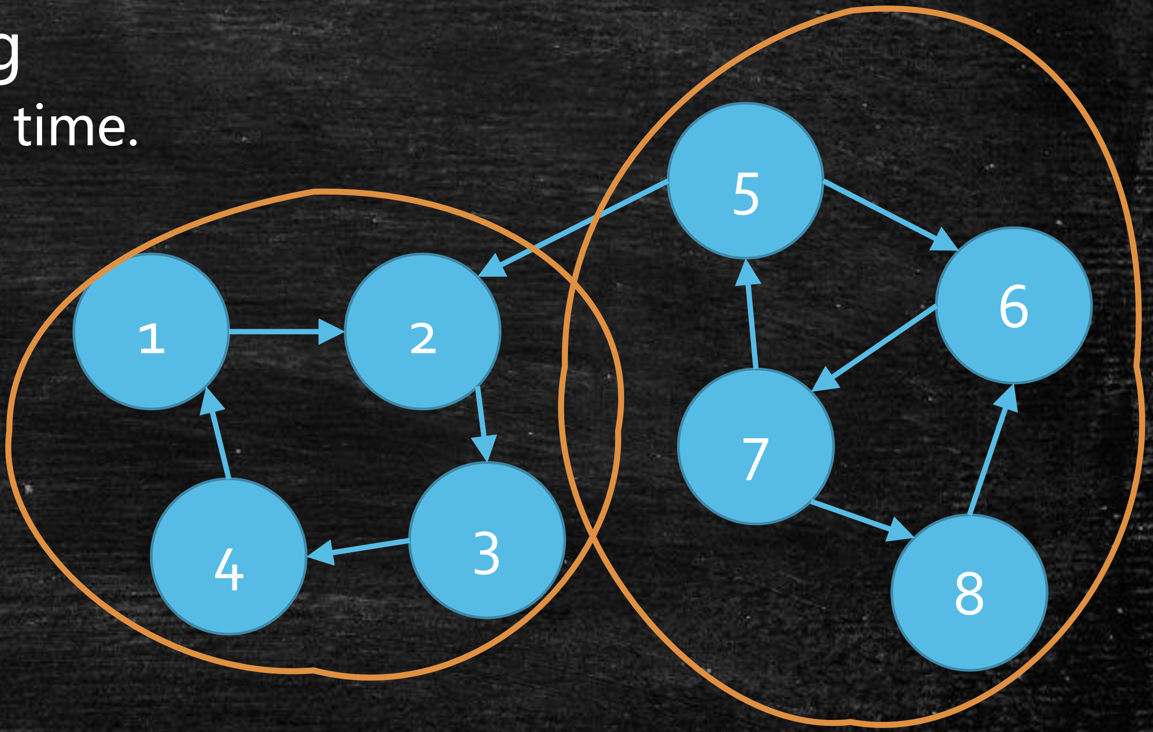


A Better Attempt

- 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
  - If we know the topological order, we know SCCs?
  - If we know who are in the tail SCC, why we need to form it?
- Answer
  - We have an **AMAZING** way to find one vertex surely inside the **tail** SCC.

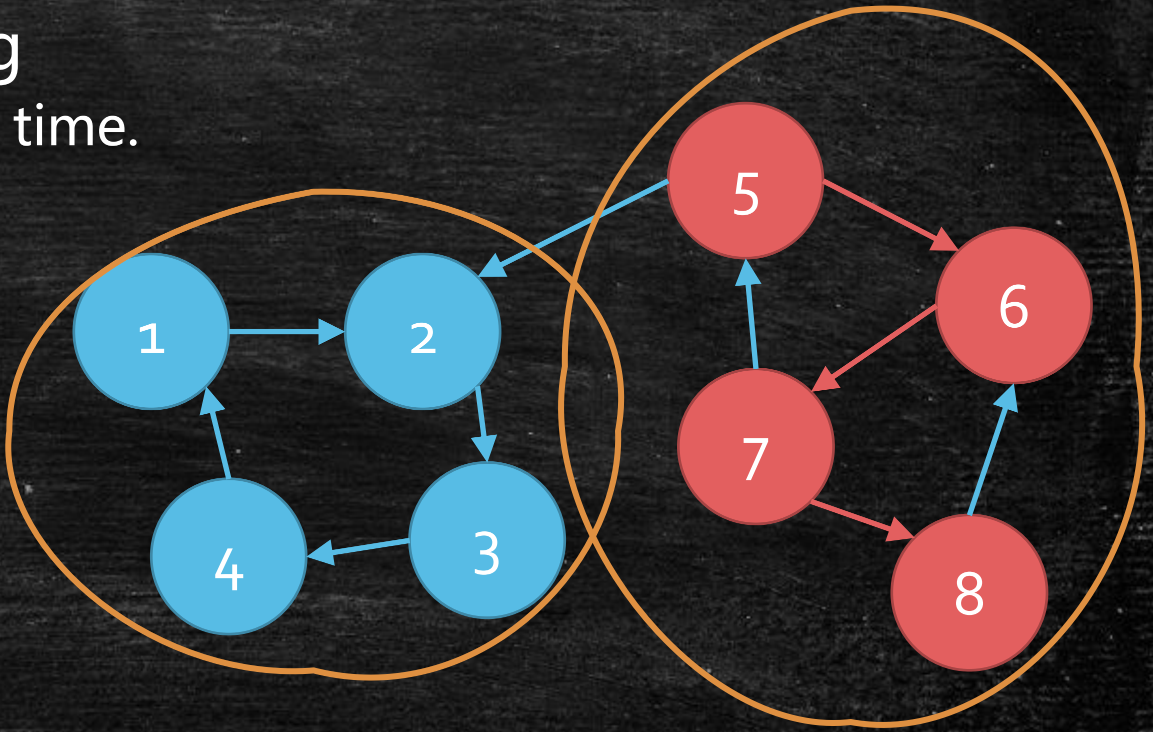
Find one vertex surely inside the **tail** SCC.

- Recall the topological ordering
  - **Tail** is the one with smallest finish time.
  - Can we apply it here?
  - Start from 5?



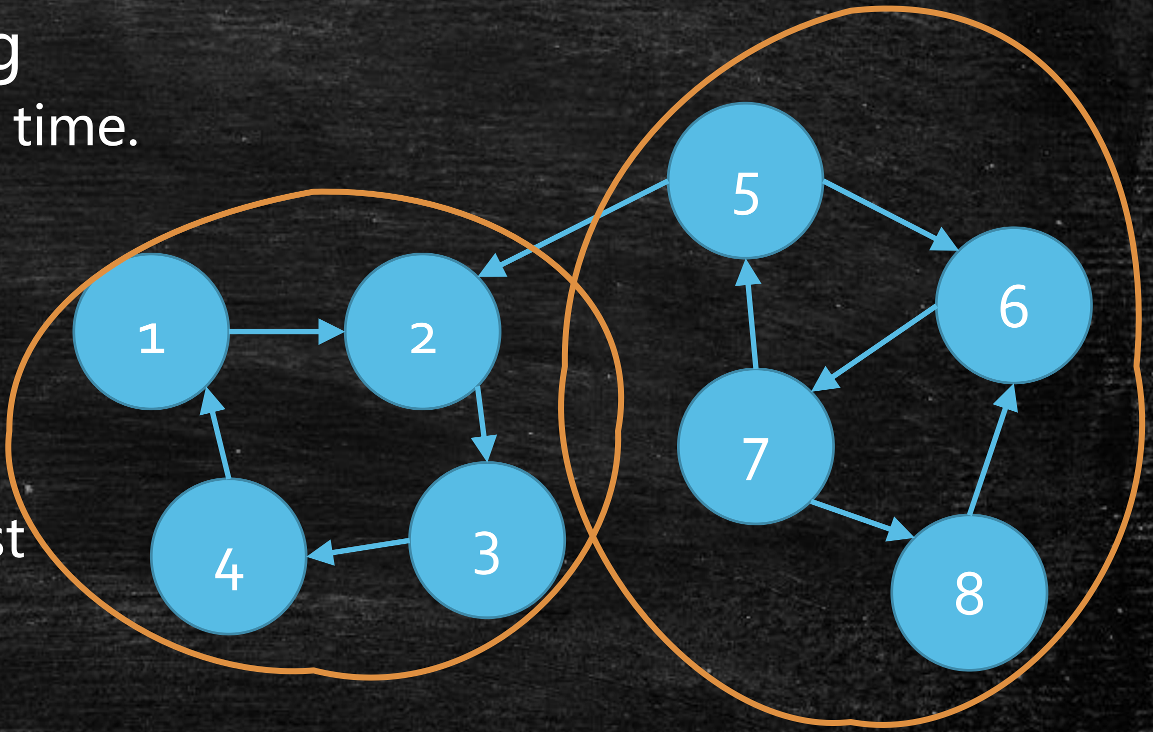
Find one vertex surely inside the **tail** SCC.

- Recall the topological ordering
  - **Tail** is the one with smallest finish time.
  - Can we apply it here?
  - Start from 5?
  - 8 is not in the **Tail** SCC.
- Problems
  - We may have back edges.



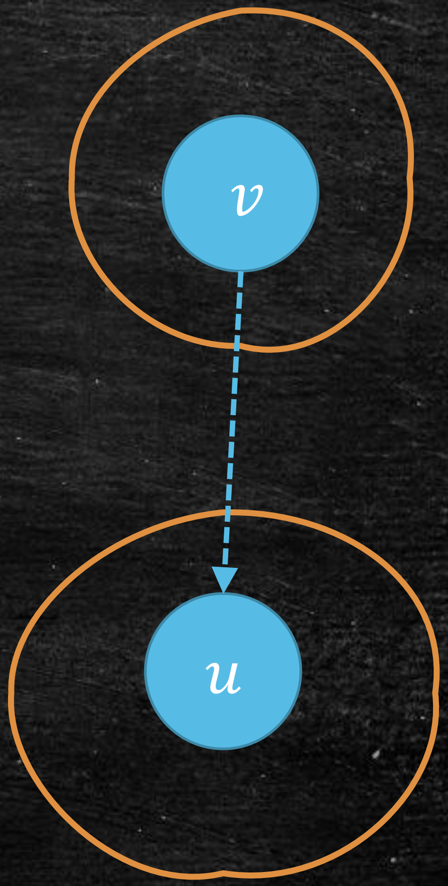
Find one vertex surely inside the **tail** SCC.

- Recall the topological ordering
  - **Tail** is the one with smallest finish time.
  - Can we apply it here?
  - Start from 5?
  - 8 is not in the **Tail** SCC.
- Can we find a **head**?
  - What about the vertex with largest finish time?



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

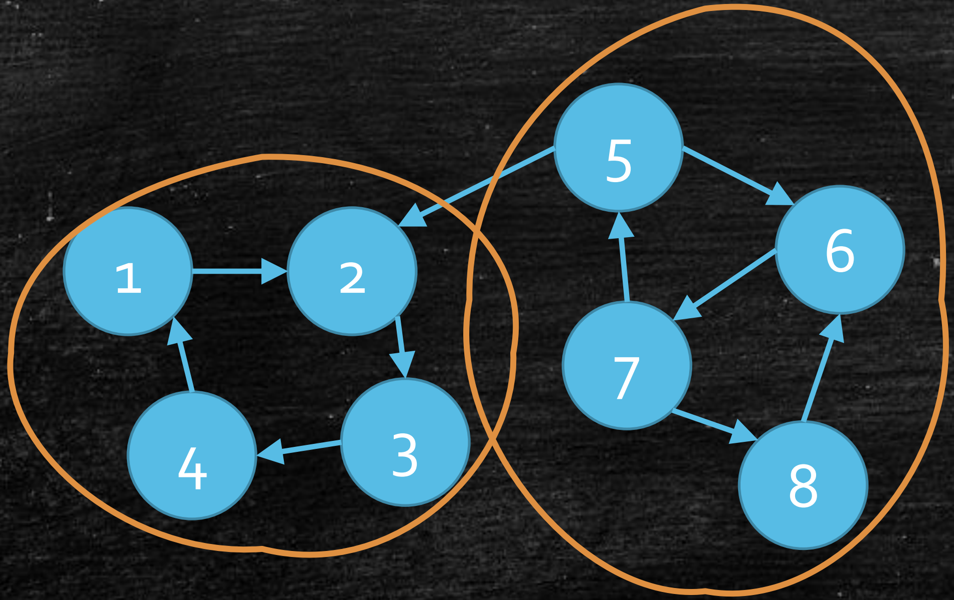
- **Naïve Idea:** the SCC contains the **largest** finish time vertex must be the **head SCC**.
- **Proof**
  - Assume
    - u has the largest finish time.
    - v inside another SCC has a path to u.
  - Claim 1: u is the root of one DFS tree.
    - Finish time property.
  - Claim 2: v can not start earlier than u
    - v is the root.
  - Claim 3: v can not in $u's$ DFS tree.
    - u, v can not be strongly connected.
  - Claim 4: v can not in another DFS tree.
    - v start later → v finish later.

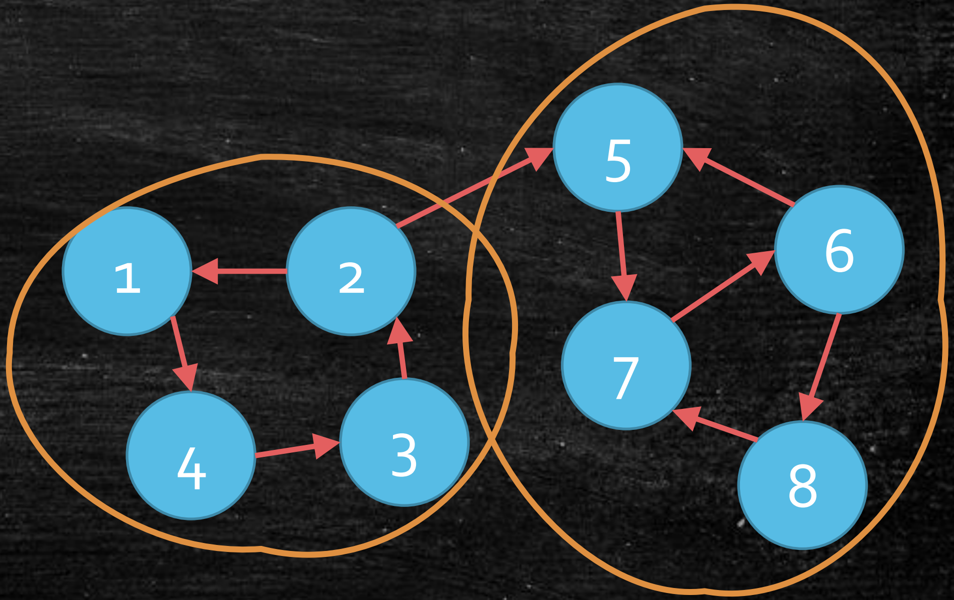


How to use this property?

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.
  6. Remove them in both G and G^R.
  7. Repeat from 2.

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.
  3. Each explore() forms a SCC.

Today's goal

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