

**CSE 421**

# **Introduction to Algorithms**

**Lecture 4: BFS, DFS Properties/Applications,  
Topological Sort**

# Generic Graph Traversal Algorithm

**Given:** Graph  $G = (V, E)$  vertex  $s \in V$

**Find:** set  $R$  of vertices reachable from  $s \in V$

**Reachable( $s$ ):**

$R \leftarrow \{s\}$

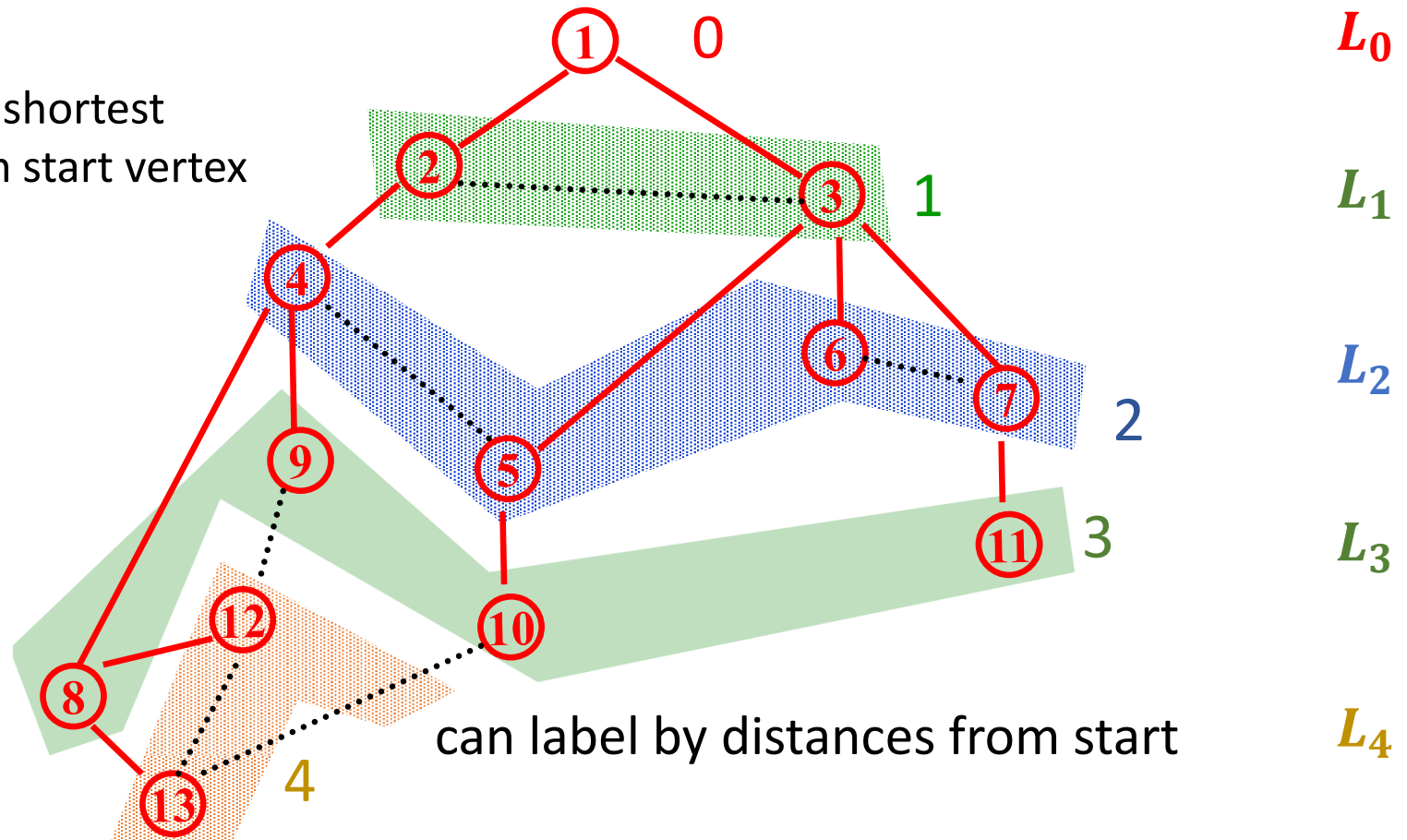
while there is a  $(u, v) \in E$  where  $u \in R$  and  $v \notin R$

    Add  $v$  to  $R$

return  $R$

# BFS Tree

Tree gives shortest paths from start vertex



# Undirected Graph Search Application: Connected Components

Want to answer questions of the form:

**Given:** vertices  $u$  and  $v$  in  $G$

Is there a path from  $u$  to  $v$ ?

**Idea:** create array  $A$  s.t

$A[u]$  = smallest numbered vertex connected to  $u$

Answer is yes iff  $A[u] = A[v]$

**Q:** Why is this better than an array  $\text{Path}[u, v]$ ?

# Undirected Graph Search Application: Connected Components

Initial state: all  $v$  unvisited

for  $s \leftarrow 1$  to  $n$  do

  if  $\text{state}(s) \neq \text{fully-explored}$  then

    BFS( $s$ ): setting  $A[u] \leftarrow s$  for each  $u$  found

    (and marking  $u$  visited/fully-explored)

endfor

Total cost:  $O(n + m)$

- Each vertex is touched once in outer procedure and edges examined in different BFS runs are disjoint
- Works also with Depth First Search ...

# DFS( $u$ ) – Recursive Procedure

Global Initialization: mark all vertices "unvisited"

DFS( $u$ )

mark  $u$  "visited" and add  $u$  to  $R$

for each edge  $(u, v)$

if ( $v$  is "unvisited")

DFS( $v$ )

end for

mark  $u$  "fully-explored"

# Properties of DFS( $s$ )

Like BFS( $s$ ):

- DFS( $s$ ) visits  $x$  iff there is a path in  $G$  from  $s$  to  $x$
- Edges into undiscovered vertices define **depth-first spanning tree** of  $G$

Unlike the BFS tree:

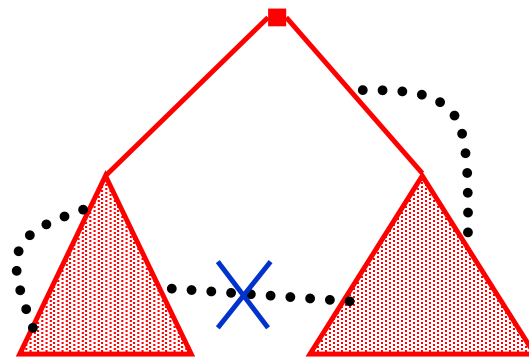
- the DFS spanning tree *isn't* minimum depth
- its levels *don't* reflect min distance from the root
- non-tree edges *never* join vertices on the same or adjacent levels

BUT...

# Non-tree edges in DFS tree of undirected graphs

**Claim:** All non-tree edges join a vertex and one of its descendants/ancestors in the DFS tree

- In other words ... No “cross edges”.





# No cross edges in DFS on undirected graphs

**Claim:** During  $\text{DFS}(x)$  every vertex marked “visited” is a descendant of  $x$  in the DFS tree  $T$

**Claim:** For every  $x, y$  in the DFS tree  $T$ , if  $(x, y)$  is an edge *not* in  $T$  then one of  $x$  or  $y$  is an ancestor of the other in  $T$

## Proof:

- One of  $\text{DFS}(x)$  or  $\text{DFS}(y)$  is called first, suppose WLOG that  $\text{DFS}(x)$  was called before  $\text{DFS}(y)$
- During  $\text{DFS}(x)$ , the edge  $(x, y)$  is examined
- Since  $(x, y)$  is a *not* an edge of  $T$ ,  $y$  was already visited when edge  $(x, y)$  was examined during  $\text{DFS}(x)$
- Therefore  $y$  was visited during the call to  $\text{DFS}(x)$  so  $y$  is a descendant of  $x$ . ■

## Applications of Graph Traversal: Bipartiteness Testing

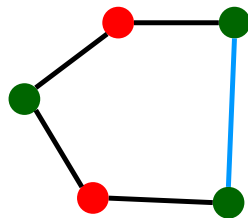
**Definition:** An undirected graph  $G$  is **bipartite** iff we can color its vertices **red** and **green** so each edge has different color endpoints

**Input:** Undirected graph  $G$

**Goal:** If  $G$  is bipartite, output a coloring;  
otherwise, output “NOT Bipartite”.

**Fact:** Graph  $G$  contains an odd-length cycle  $\Rightarrow$  it is not bipartite

Just coloring the cycle part  
of  $G$  is impossible



On a cycle the two colors must alternate, so

- **green** every 2<sup>nd</sup> vertex
- **red** every 2<sup>nd</sup> vertex

Can't have either if length is not divisible by 2.

## Applications of Graph Traversal: Bipartiteness Testing

**WLOG** (“without loss of generality”): Can assume that  $G$  is connected

- Otherwise run on each component

**Simple idea:** start coloring nodes starting at a given node  $s$

- Color  $s$  **red**
- Color all neighbors of  $s$  **green**
- Color all their neighbors **red**, etc.
- If you ever hit a node that was already colored
  - the **same** color as you want to color it, ignore it
  - the **opposite** color, output “NOT Bipartite” and halt

## BFS gives Bipartiteness

Run BFS assigning all vertices from layer  $L_i$  the color  $i \bmod 2$

- i.e., **red** if they are in an even layer, **green** if in an odd layer
- if no edge joining two vertices of the same color
  - then it is a good coloring
- otherwise
  - there is a bad edge; output “Not Bipartite”

Why is that “Not Bipartite” output correct?

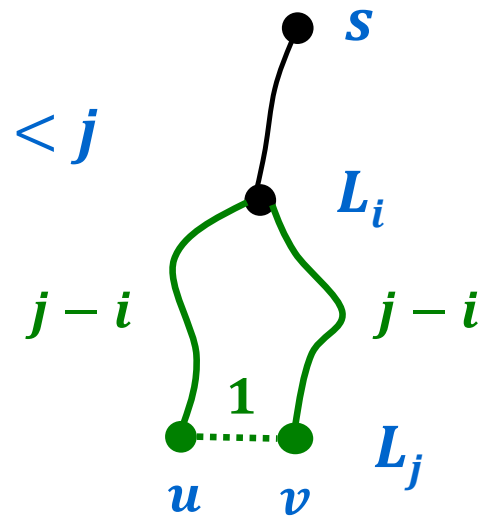
# Why does BFS work for Bipartiteness?

**Recall:** All edges join vertices on the same or adjacent BFS layers  
⇒ Any bad edge must join two vertices  $u$  and  $v$  in the same layer

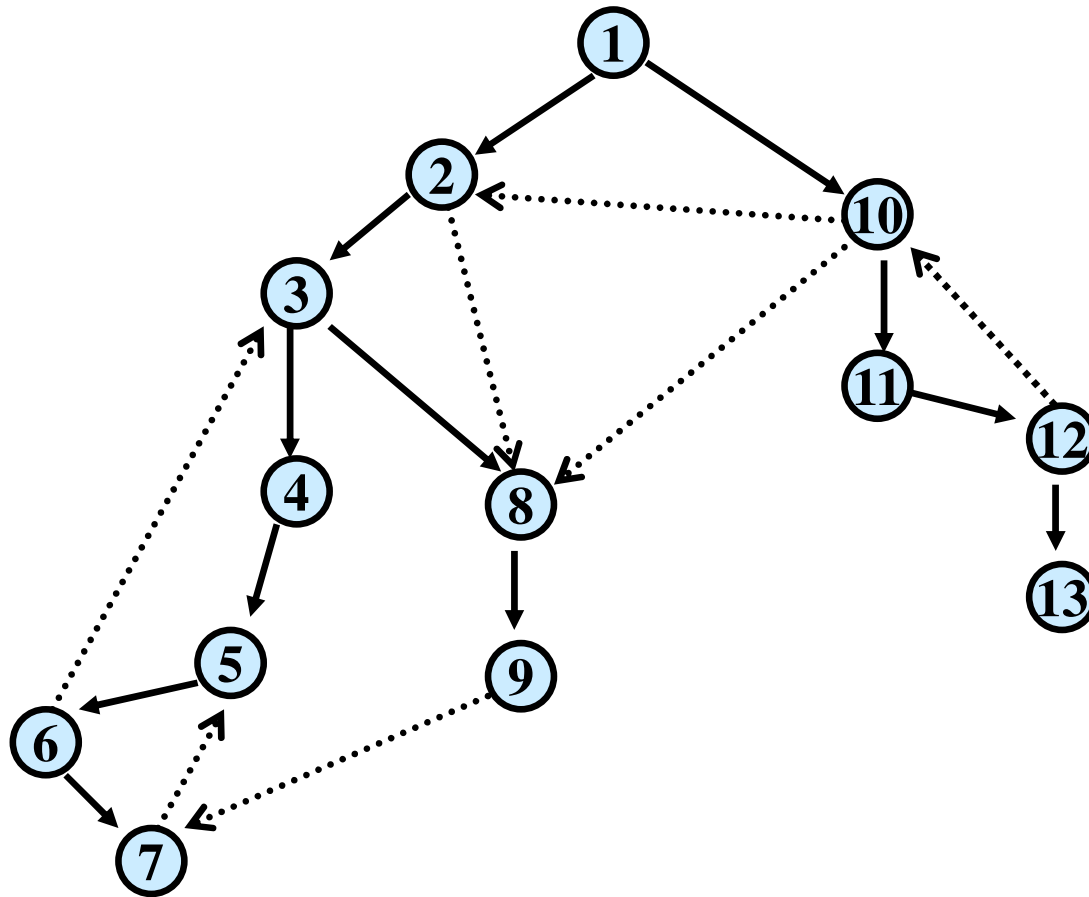
Say the layer with  $u$  and  $v$  is  $L_j$

$u$  and  $v$  have common ancestor at some level  $L_i$  for  $i < j$

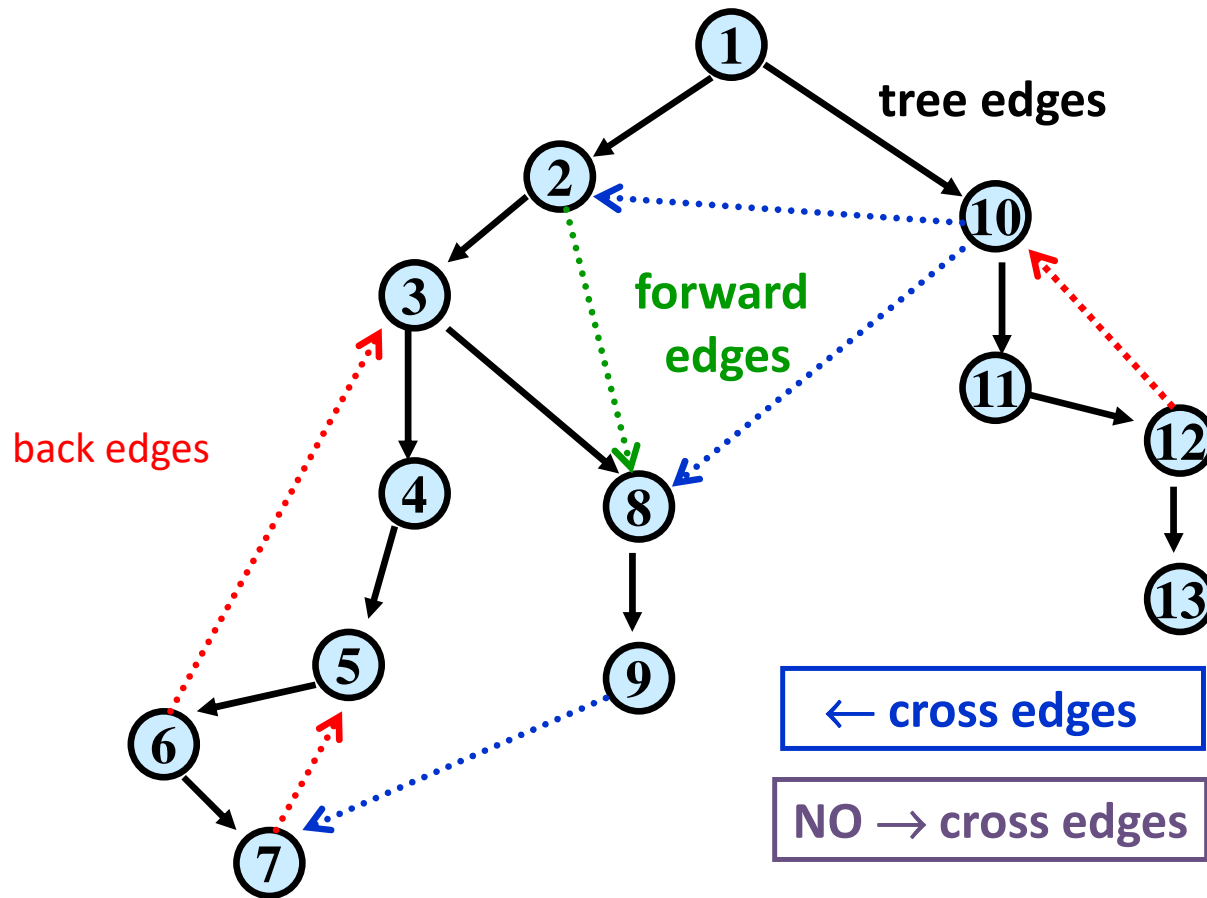
Odd cycle of length  $2(j - i) + 1$   
⇒ Not Bipartite



# DFS( $v$ ) for a directed graph



# DFS( $v$ )



## Properties of Directed DFS

- Before  $\text{DFS}(s)$  returns, it visits all previously unvisited vertices reachable via directed paths from  $s$
- Every cycle contains a back edge in the DFS tree



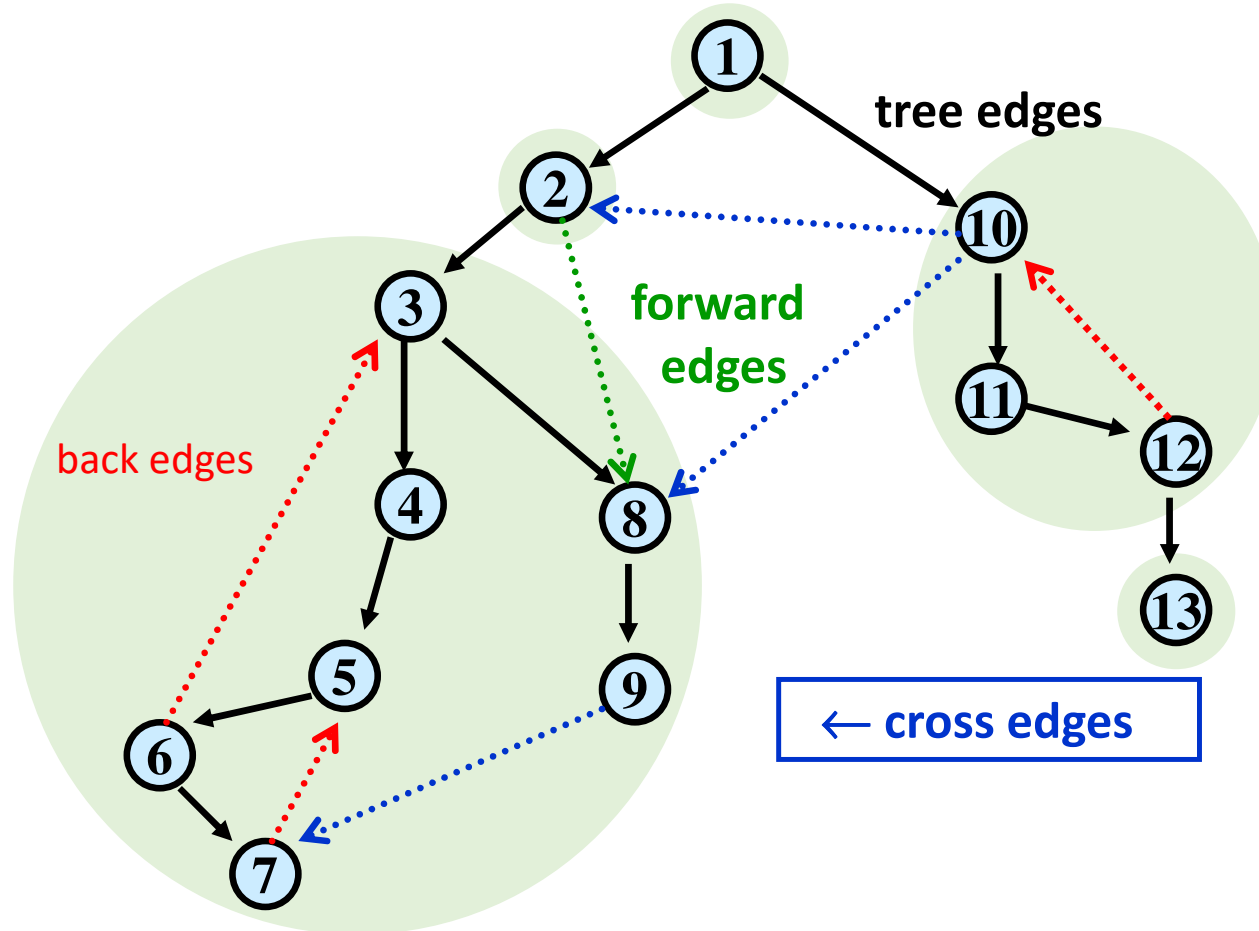
# Strongly Connected Components of Directed Graphs

**Defn:** Vertices  $u$  and  $v$  are **strongly connected** iff they are on a directed cycle (there are paths from  $u$  to  $v$  and from  $v$  to  $u$ ).

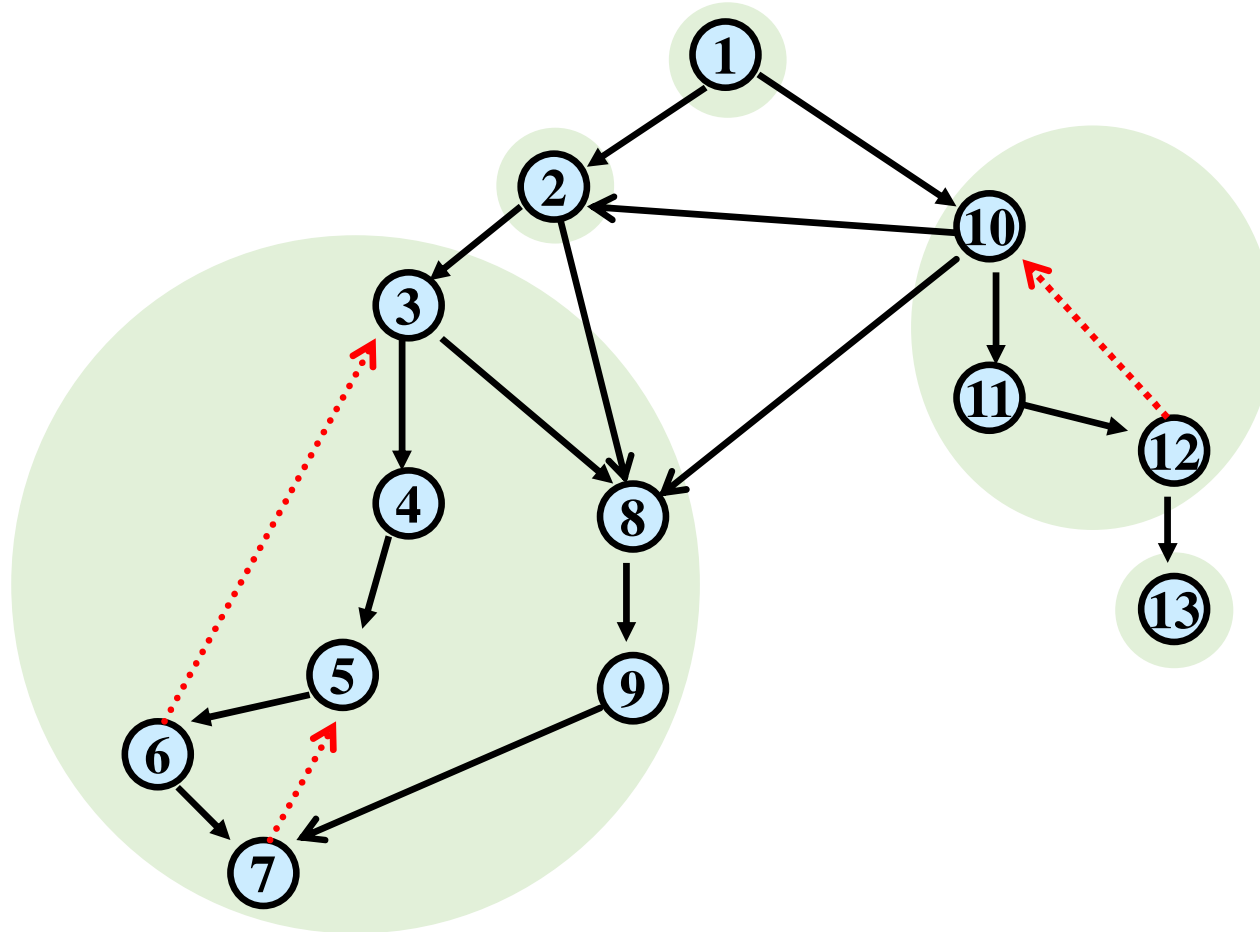
**Defn:** Can partition vertices of any directed graph into **strongly connected components**:

1. all pairs of vertices in the same component are strongly connected
  2. can't merge components and keep property 1
- Strongly connected components can be stored efficiently just like connected components
  - Can be found by extending DFS algorithm in  $O(n + m)$  time using extra bookkeeping
    - We won't cover the details

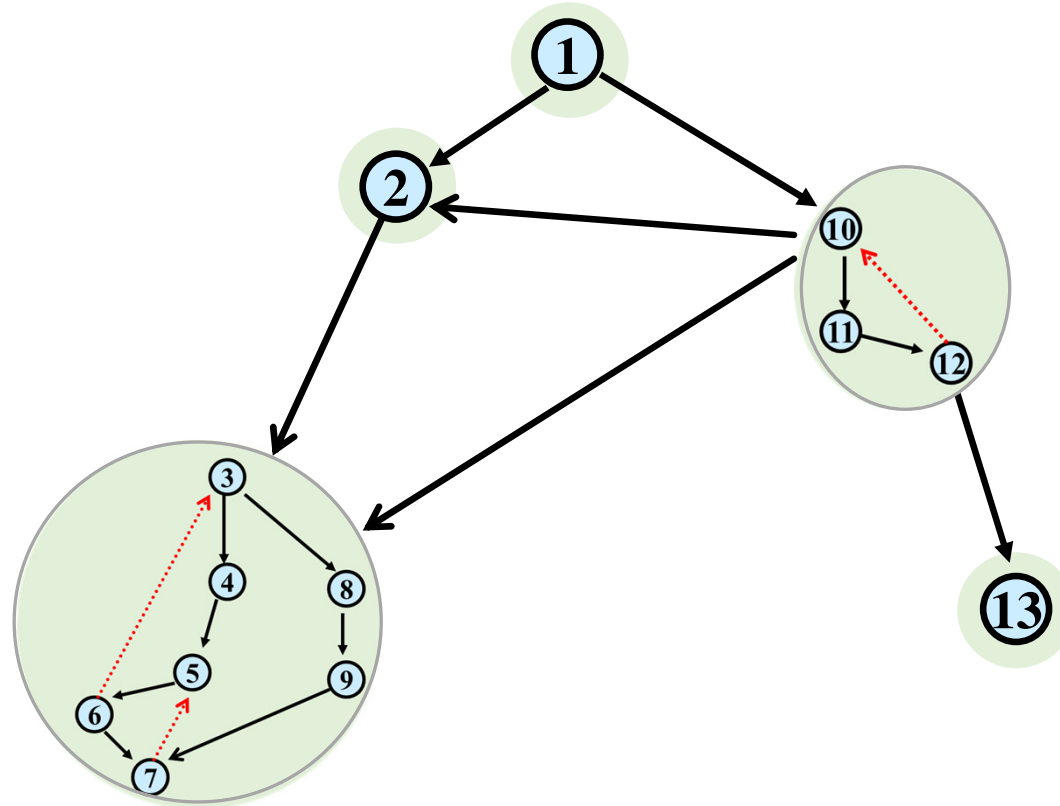
# Strongly Connected Components



# Strongly Connected Components



# Strongly Connected Components



# Directed Acyclic Graphs

A directed graph  $G = (V, E)$  is **acyclic** iff it has no directed cycles

**Terminology:** A **directed acyclic graph** is also called a **DAG**

After shrinking the strongly connected components of a directed graph to single vertices, the result is a DAG

# Topological Sort

**Given:** a directed acyclic graph (DAG)  $G = (V, E)$

**Output:** numbering of the vertices of  $G$  with distinct numbers from  $1$  to  $n$  so that edges only go from lower numbered to higher numbered vertices

Applications:

- nodes represent tasks
- edges represent precedence between tasks
- topological sort gives a sequential schedule for solving them

Nice algorithmic paradigm for general directed graphs:

- Process strongly connected components one-by-one in the order given by topological sort of the DAG you get from shrinking them.



## In-degree 0 vertices

**Claim:** Every DAG has a vertex of in-degree 0

**Proof:** By contradiction

Suppose every vertex has some incoming edge

Consider following procedure:

while (true) do

$v \leftarrow$  some predecessor of  $v$

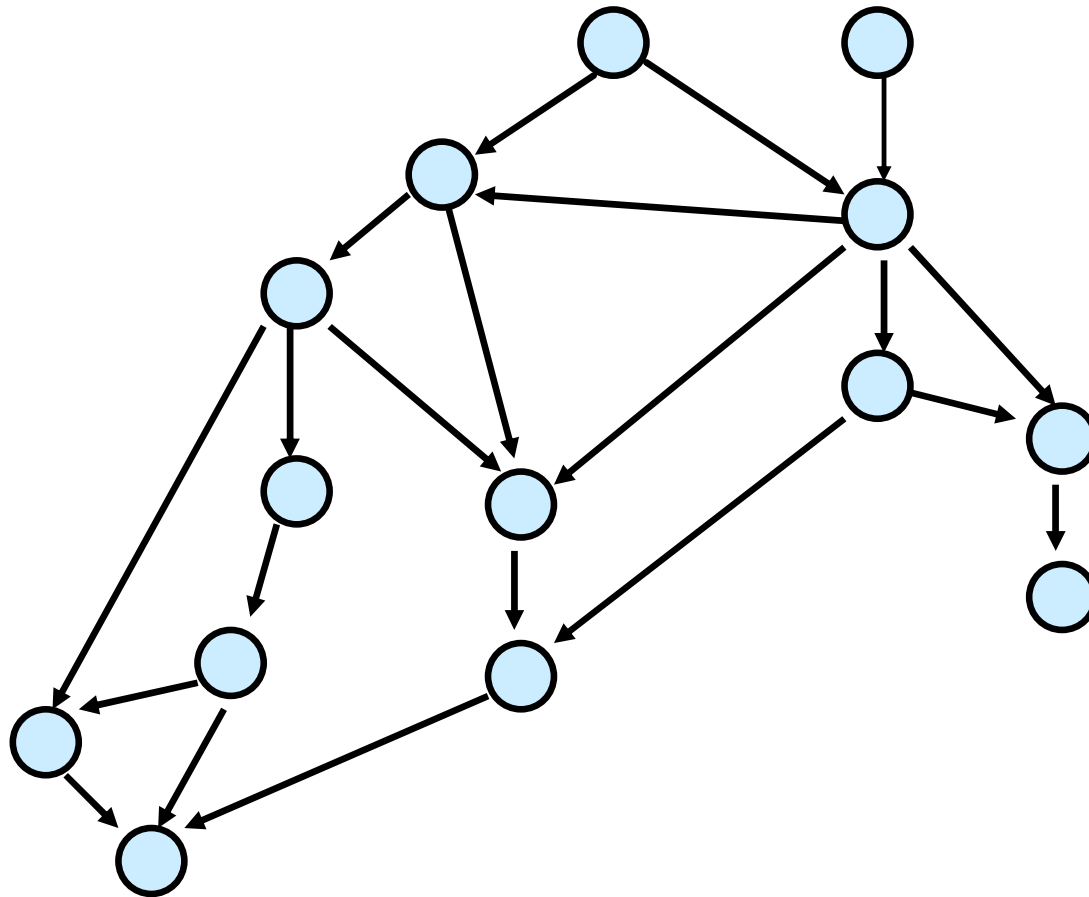
- After  $n + 1$  steps where  $n = |V|$  there will be a repeated vertex
  - This yields a cycle, contradicting that it is a DAG. ■



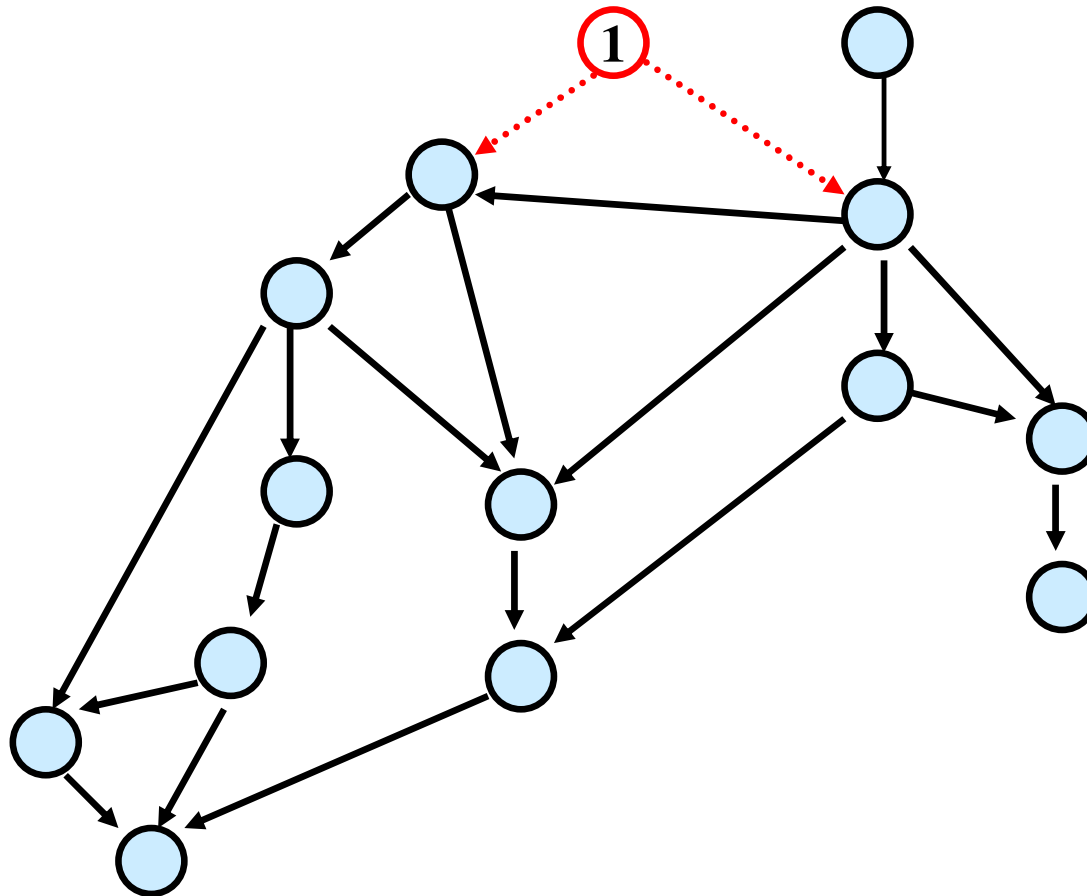
# Topological Sort

- Can do using DFS
- Alternative simpler idea:
  - Any vertex of in-degree 0 can be given number 1 to start
  - Remove it from the graph
  - Then give a vertex of in-degree 0 number 2
  - Etc.

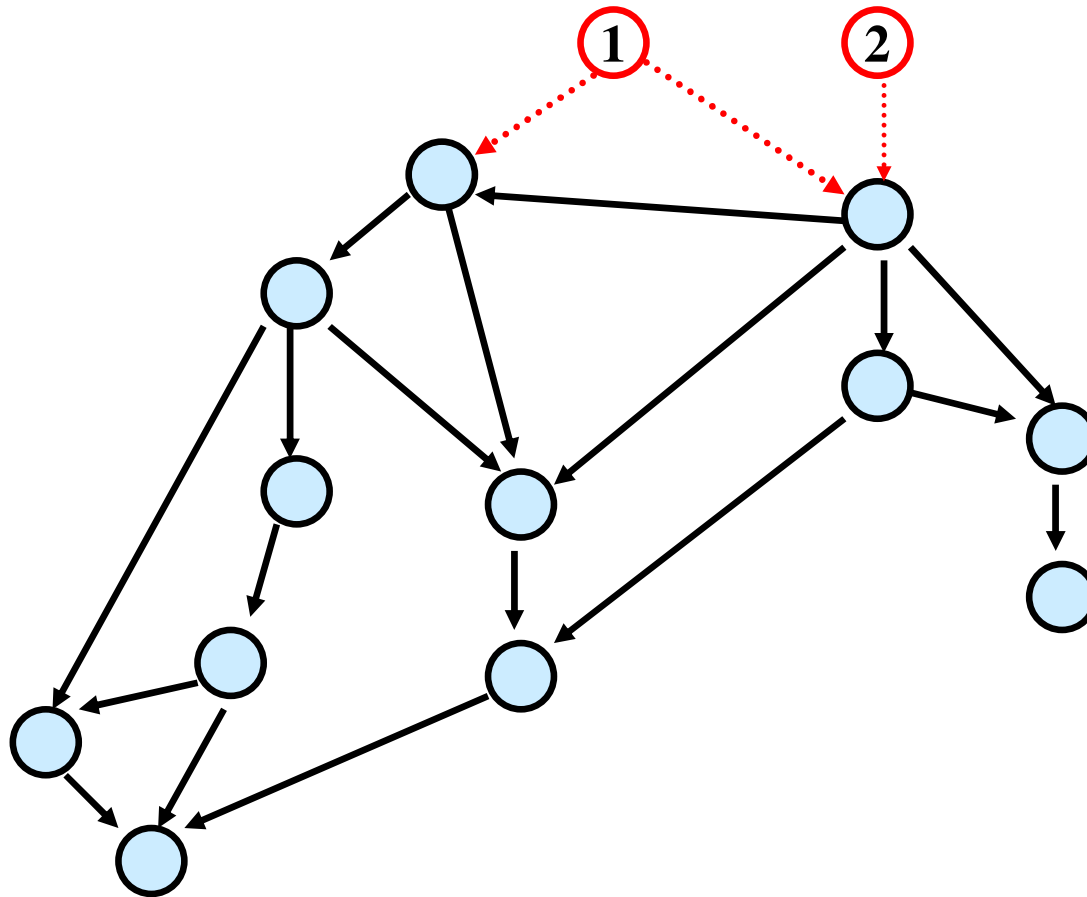
# Topological Sort



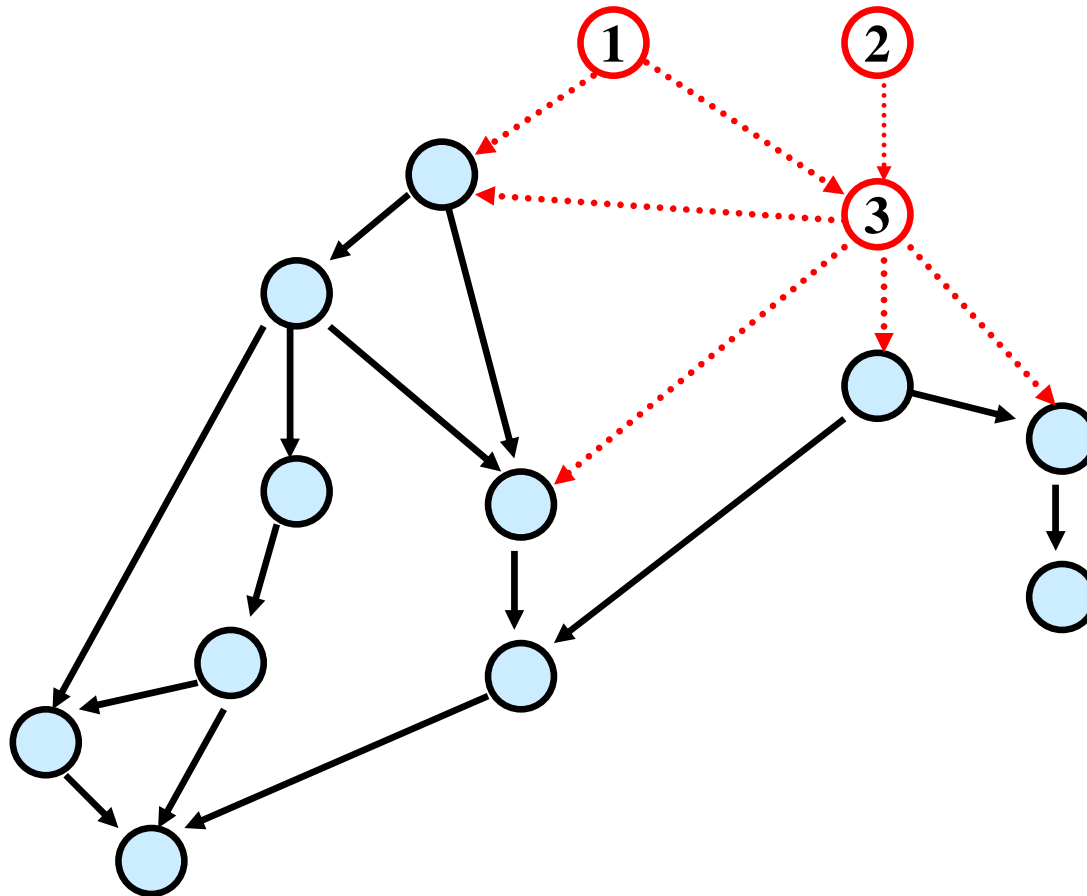
# Topological Sort



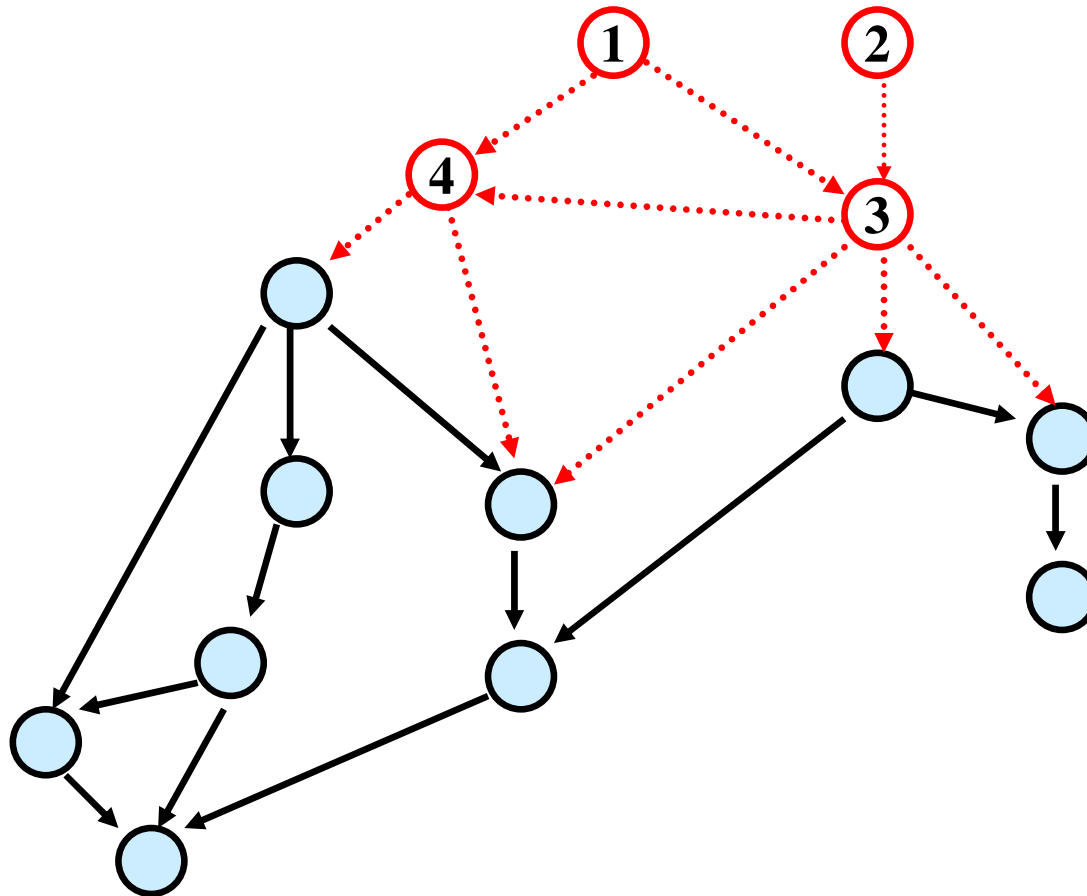
# Topological Sort



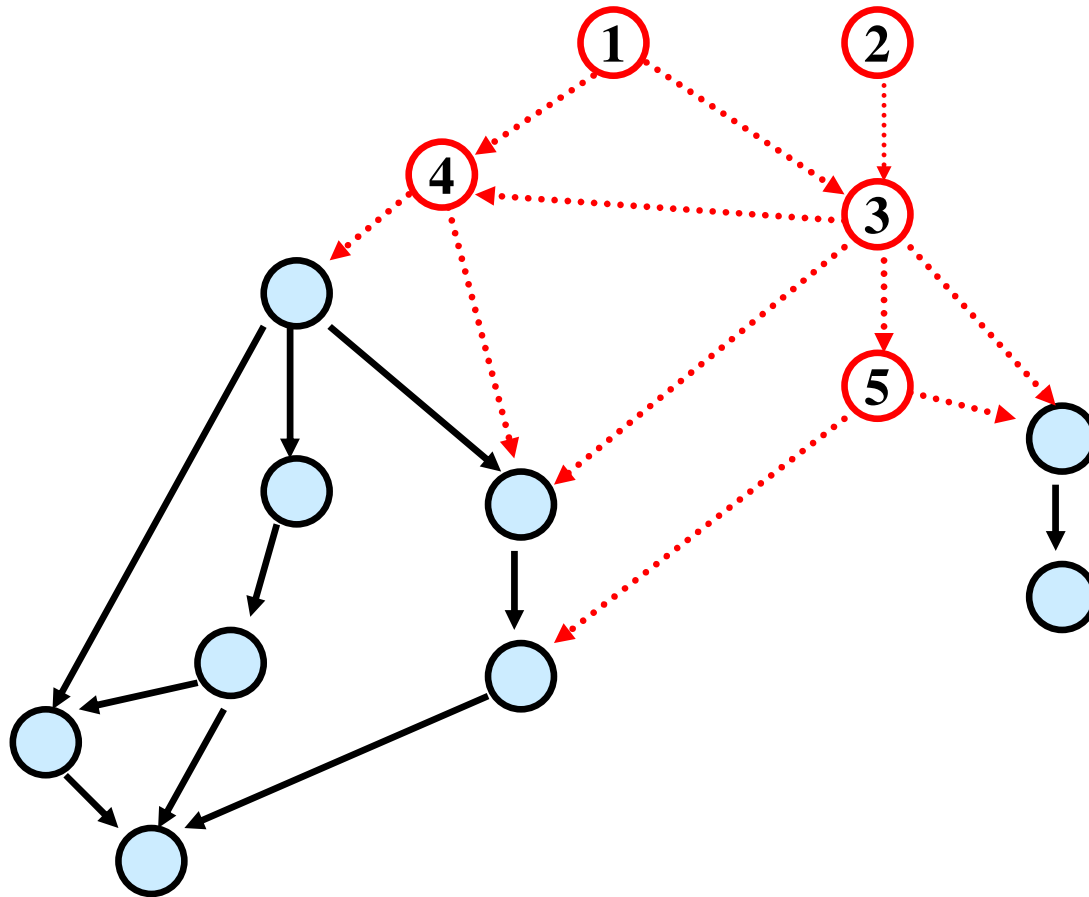
# Topological Sort



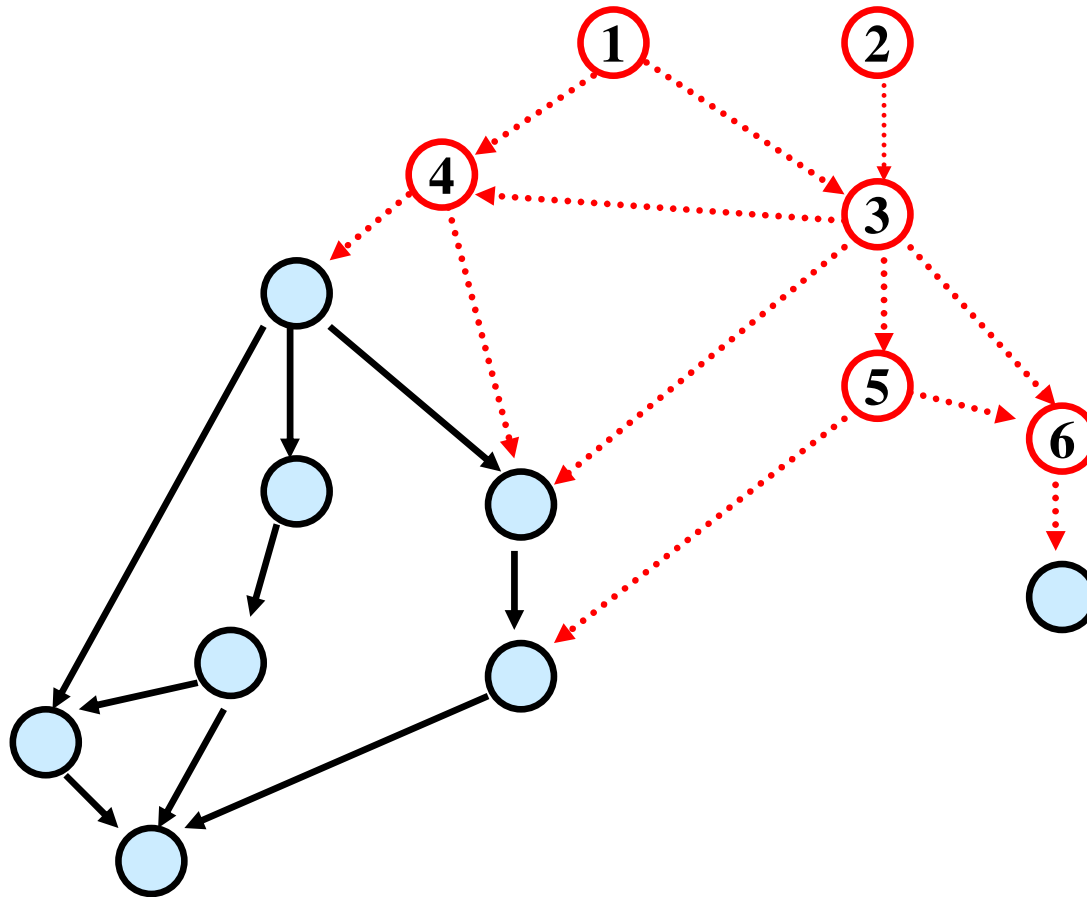
# Topological Sort



# Topological Sort

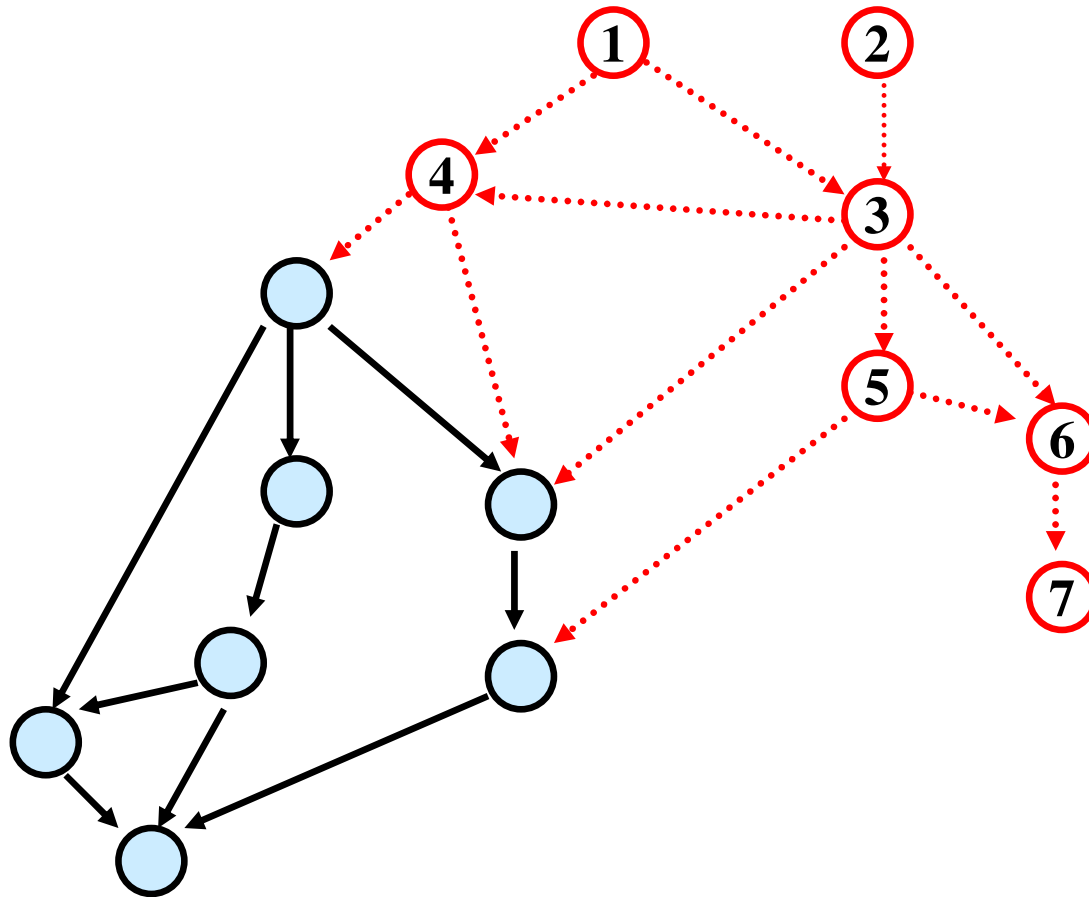


# Topological Sort



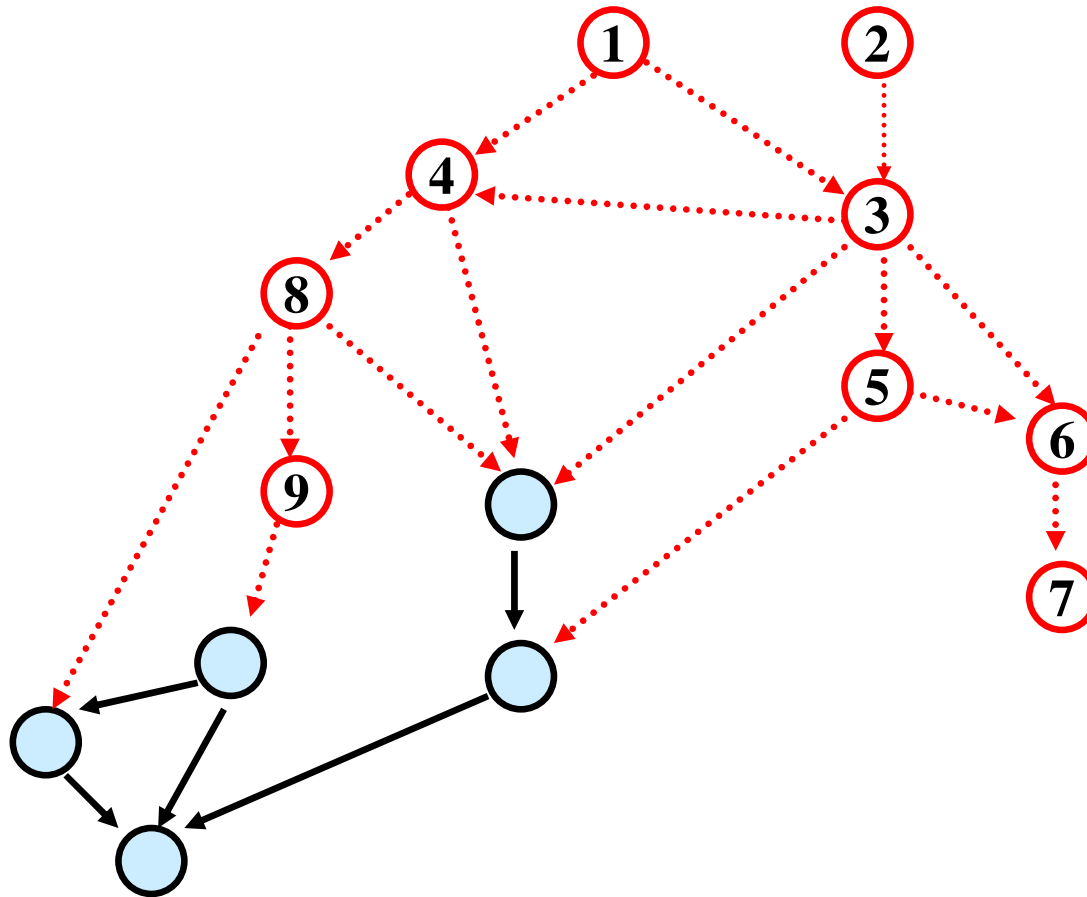


# Topological Sort

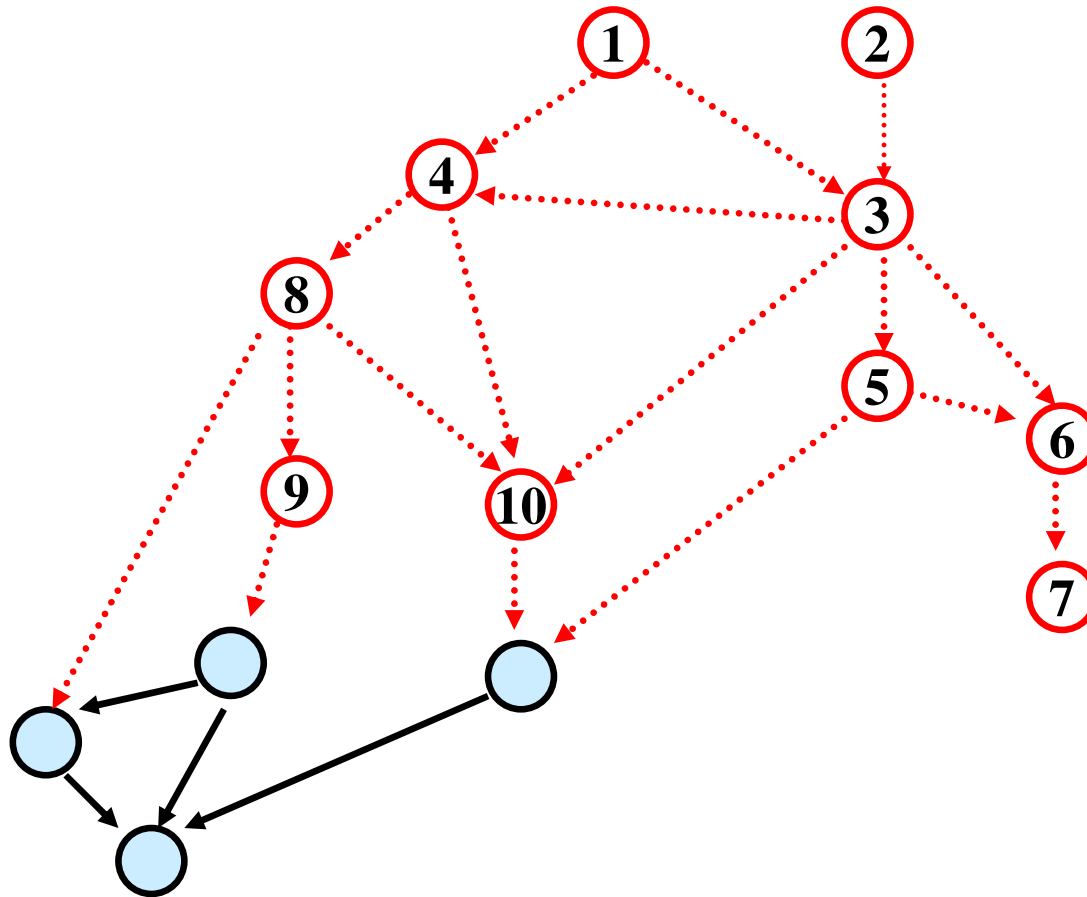




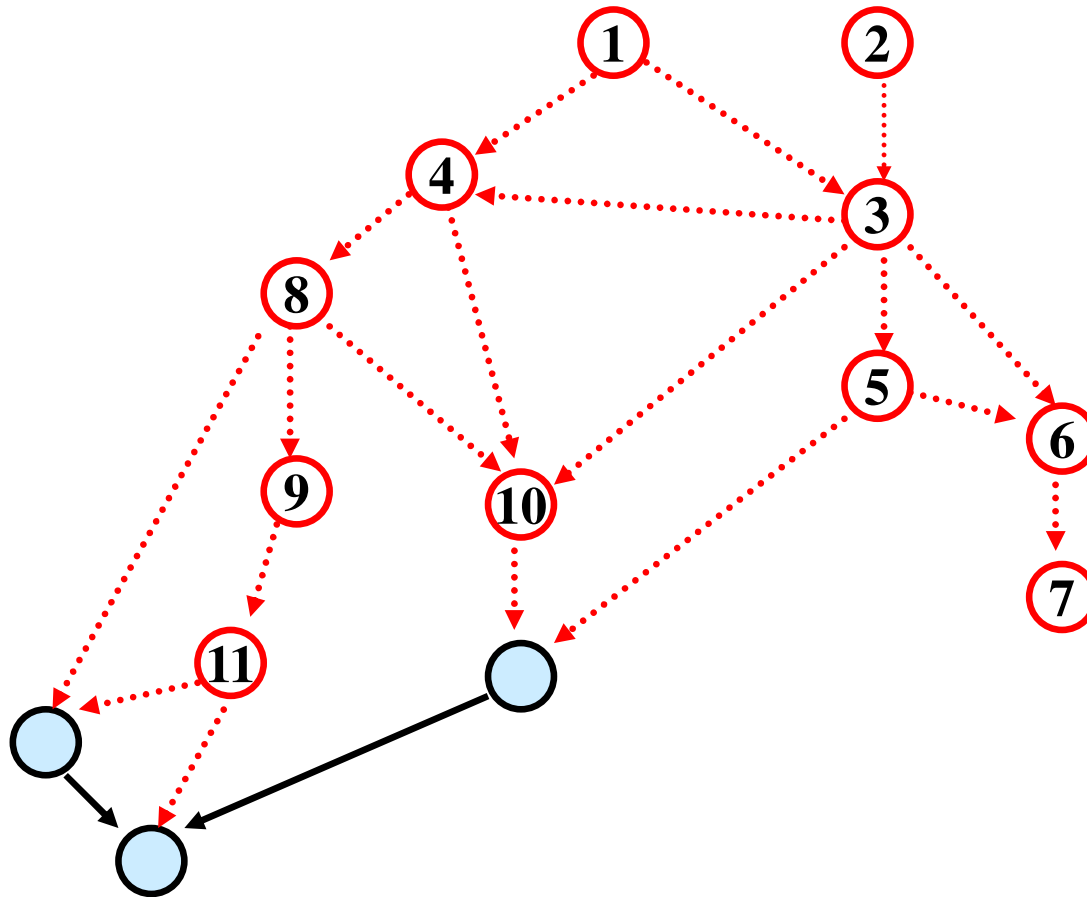
# Topological Sort



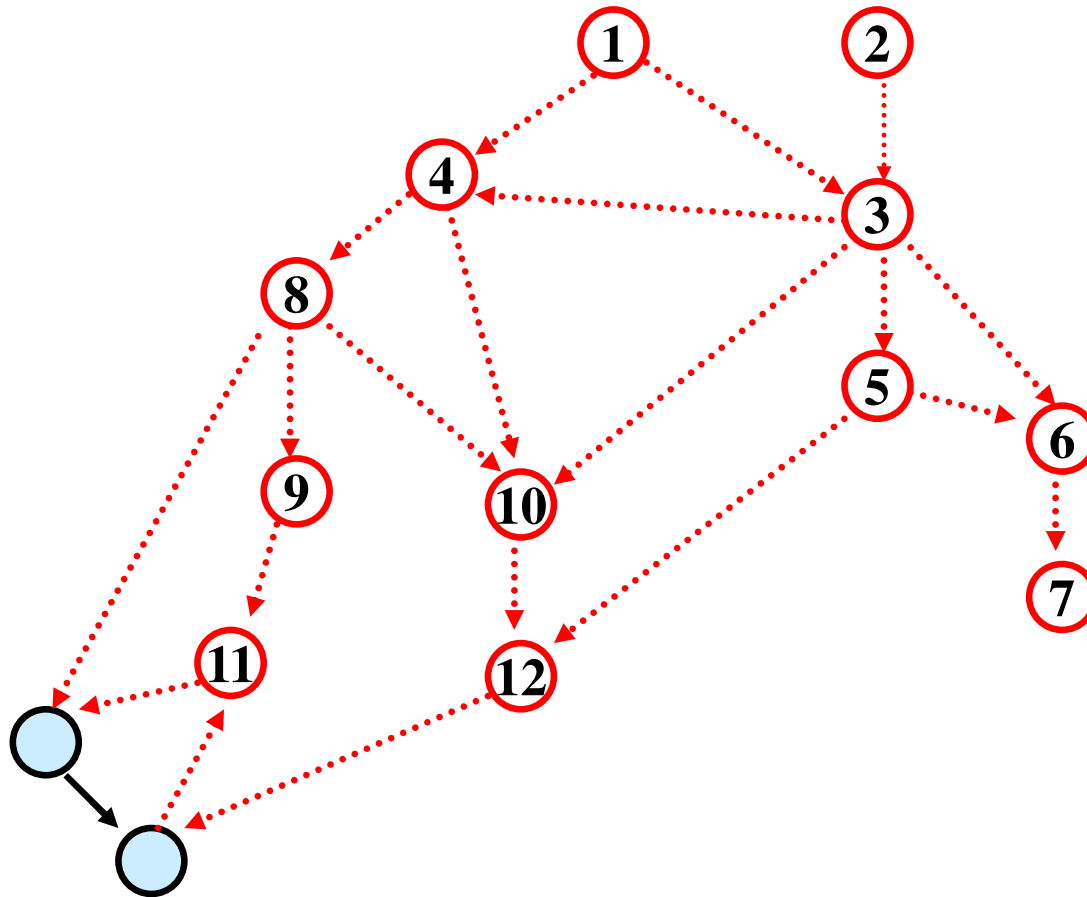
# Topological Sort



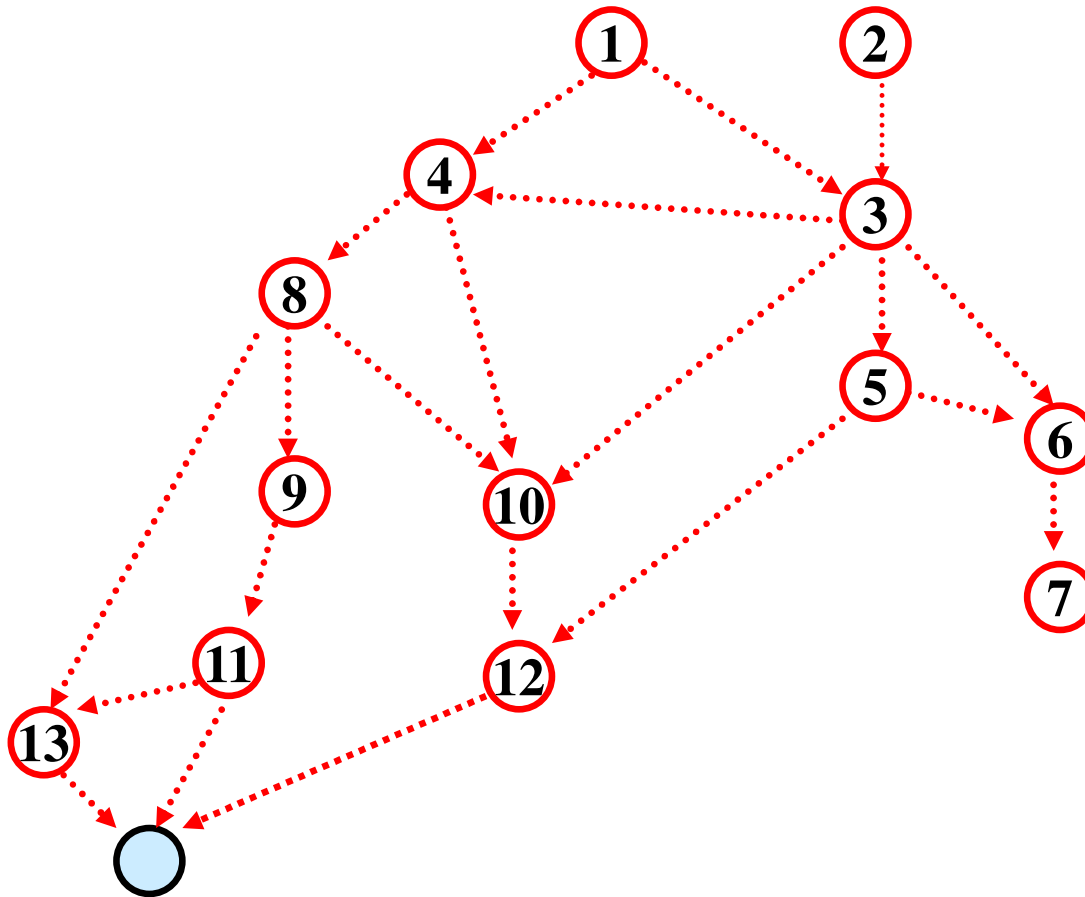
# Topological Sort



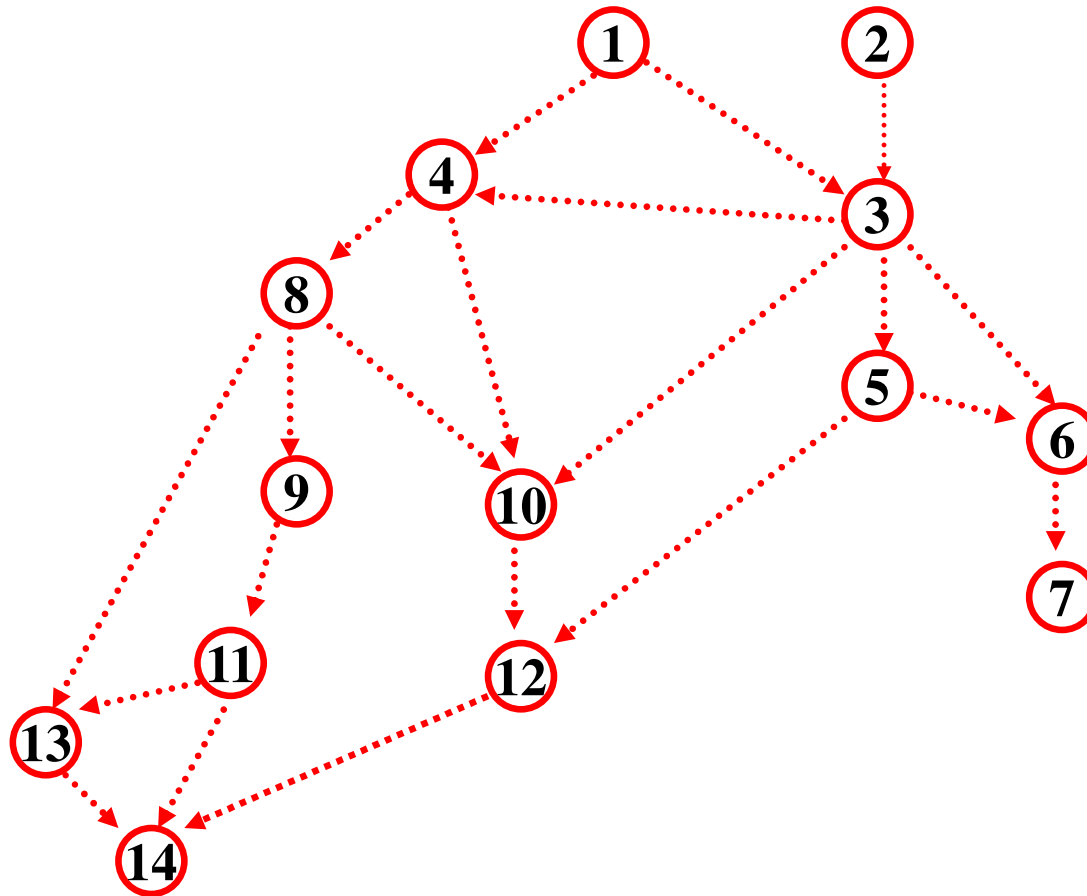
# Topological Sort



# Topological Sort



# Topological Sort





# Implementing Topological Sort

- Go through all edges, computing array with in-degree for each vertex  
 $O(m + n)$
- Maintain a list of vertices of in-degree **0**
- Remove any vertex in list and number it
- When a vertex is removed, decrease in-degree of each neighbor by **1**  
and add them to the list if their degree drops to **0**

Total cost:  $O(m + n)$