Week 05
More Graphs
Greedy Algorithms

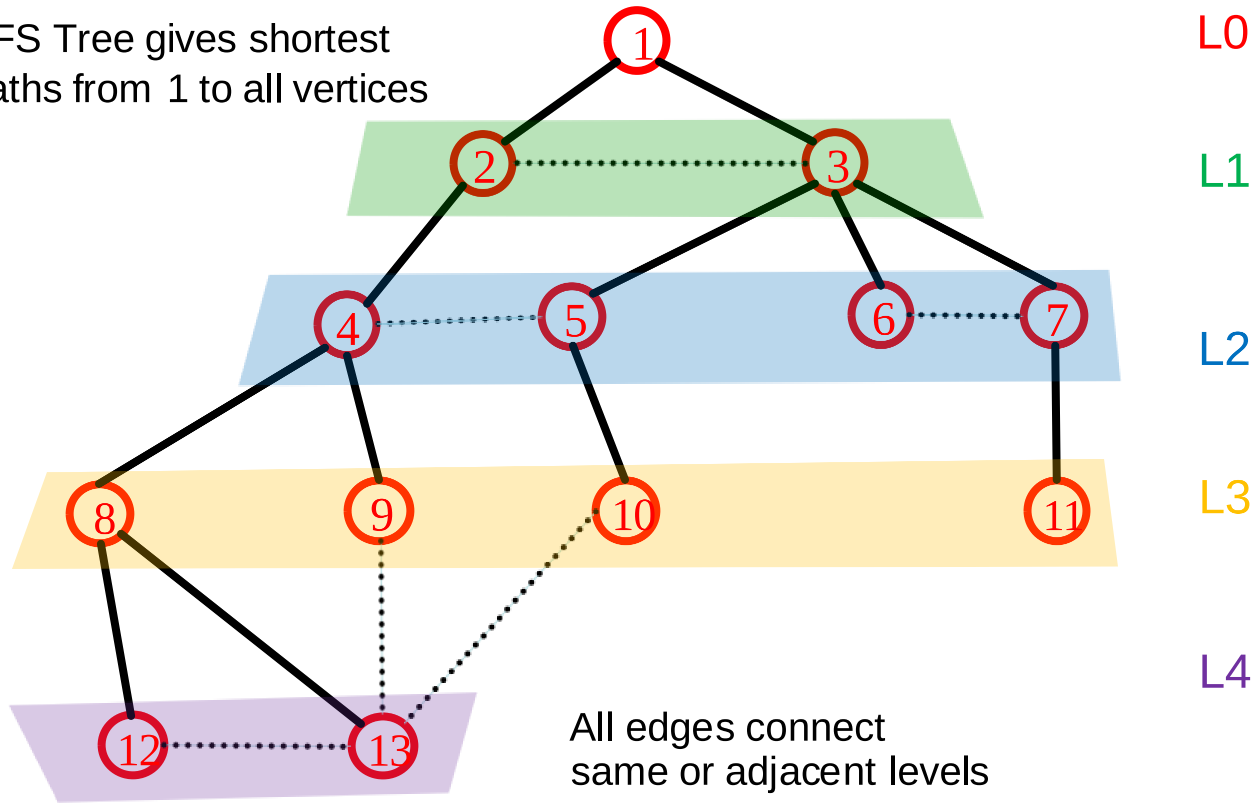
BFS implementation

Global initialization: mark all vertices "undiscovered"

```
BFS(s)
    mark  s "discovered"
    queue = { s }
    while queue not empty
        u = remove_first(queue)
        for each edge {u,x}
            if (x is undiscovered)
                mark x discovered
                append x on queue
        mark u fully-explored
```

BFS Tree

BFS Tree gives shortest
paths from 1 to all vertices



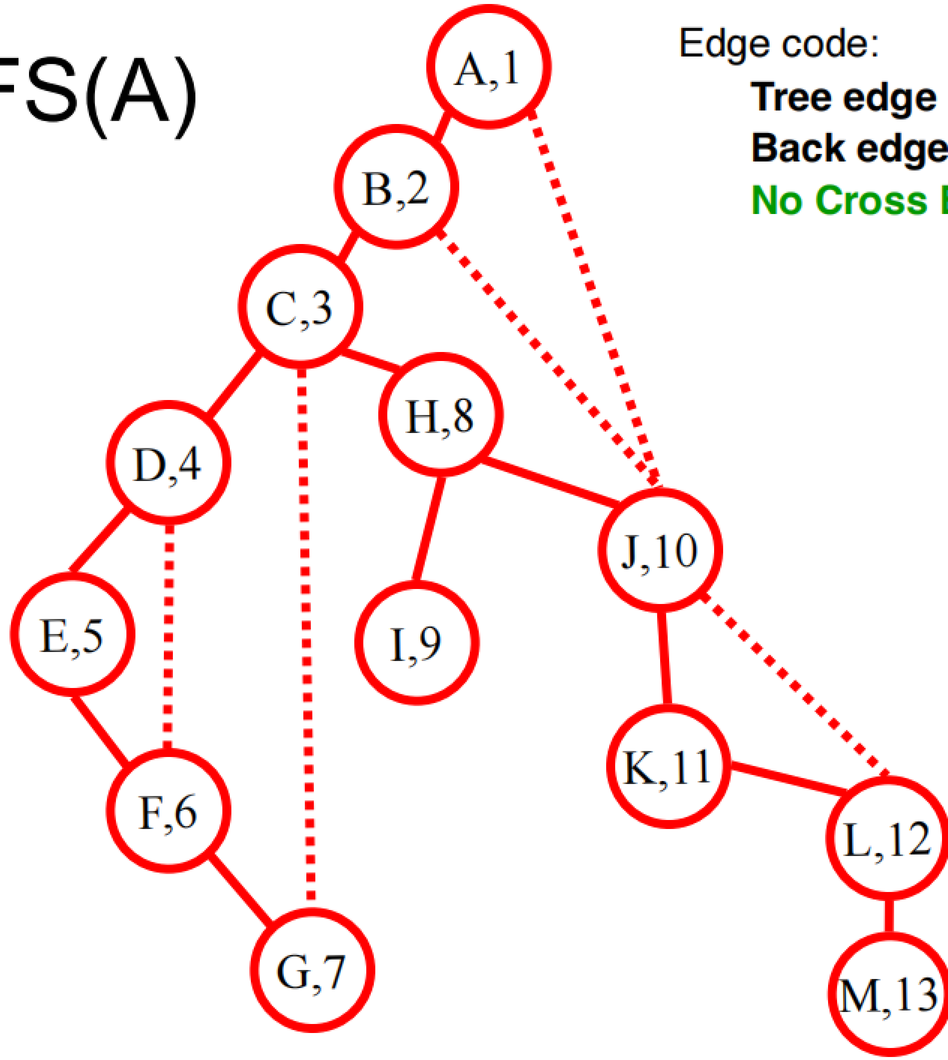
All edges connect
same or adjacent levels

DFS(s) - Recursive Procedure

Global Initialization: mark all vertices undiscovered

```
DFS(v)
   mark v discovered and add v to stack
   for each edge {v,x}
      if (x is undiscovered)
         DFS(x)
   end for
   mark v full-discovered
```

DFS(A)



Edge code:

Tree edge

Back edge

No Cross Edges!

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

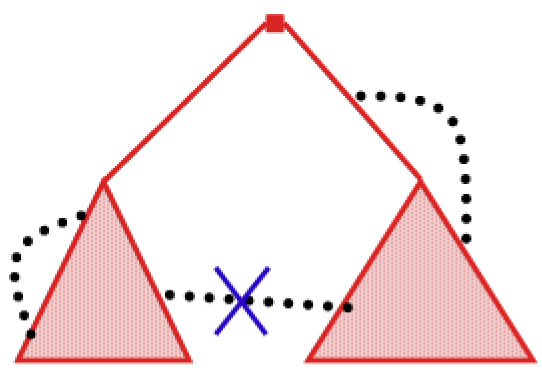
Unlike the BFS tree:

- 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 level

Non-tree edges in DFS tree of undirected graphs

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

- In other words, … No "cross edges".



No cross edges in DFS on undirected graphs

Claim: During DFS(x) every vertex marked "discovered" is adescendant 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 DFS(x) or DFS(y) is called first, suppose WLOG that DFS(x) was called before DFS(y)
- During DFS(x), the edge (x, y) is examined
- Since (x, y) is a *not* an edge of T, y was already discovered when edge (x, y) was examined during DFS(x)
- Therefore, y was discovered during the call to DFS(x) so y is a descendant of x. ■

Applications of Graph Traversal: Bipartiteness Testing

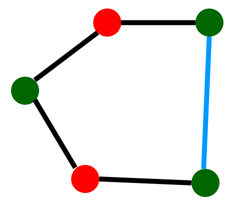
Definition: An undirected graph G is bipartite iff we can color its vertices **red** and **green** so each edge has different color endpoints

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 2nd vertex
- **red** every 2nd 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?

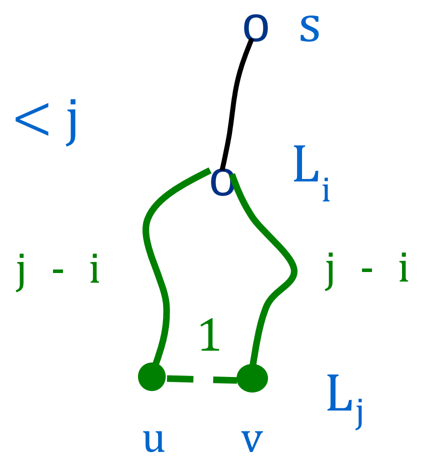
Why does BFS work for Bipartiteness?

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

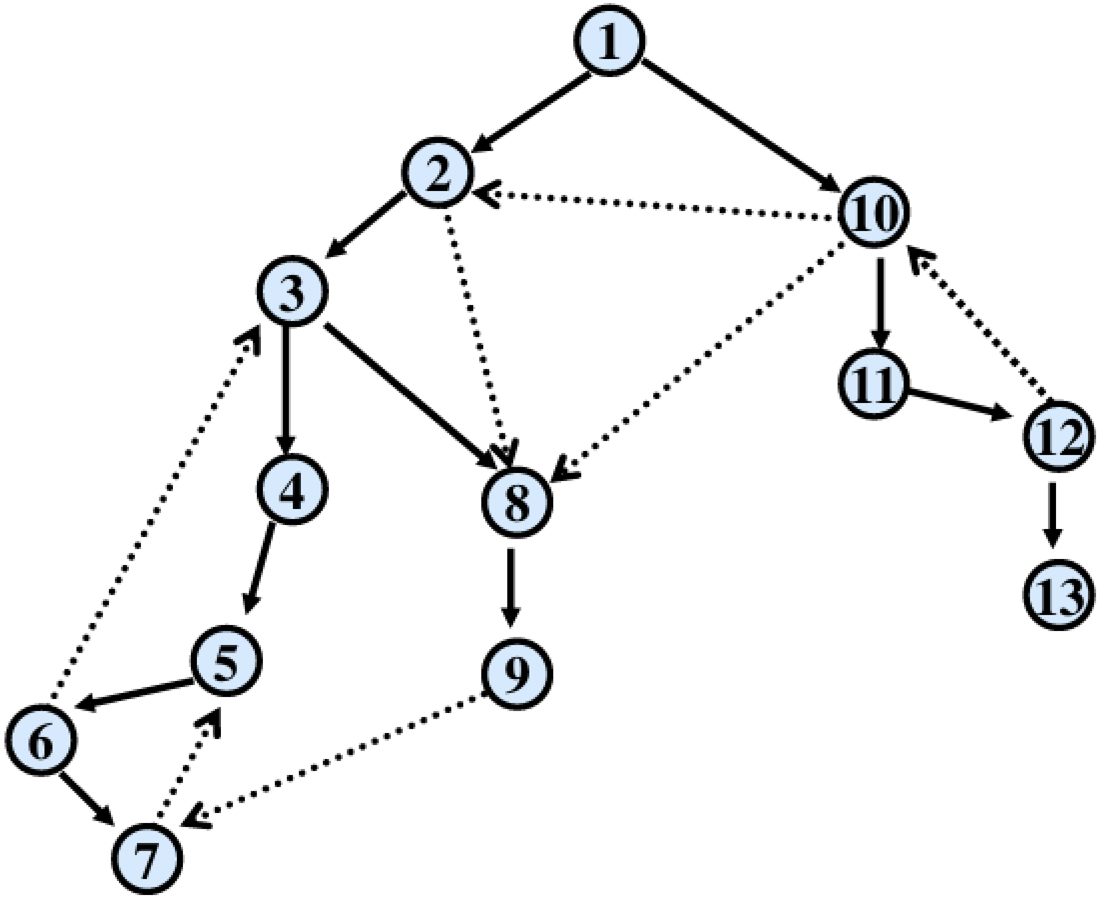
Say the layer with u and v is L_j

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

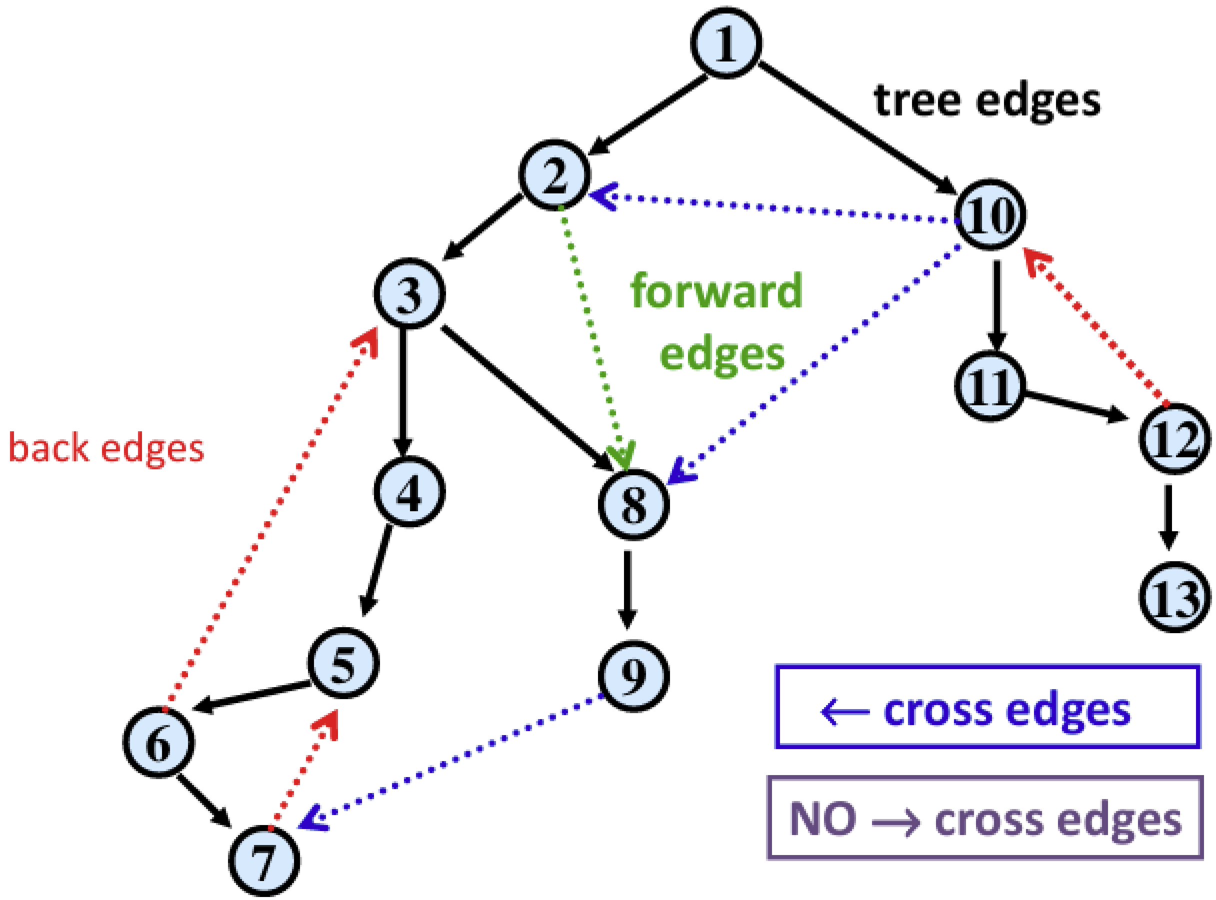


Odd cycle of length 2(j-i)+1
→ Not Bipartite

DFS(v) for a directed graph



DFS(v)



1
2
3
4
5
6
7
8
9
10
11
12
13
tree edges
forward edges
back edges
← cross edges
NO → cross edges

Properties of Directed DFS

- Before 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

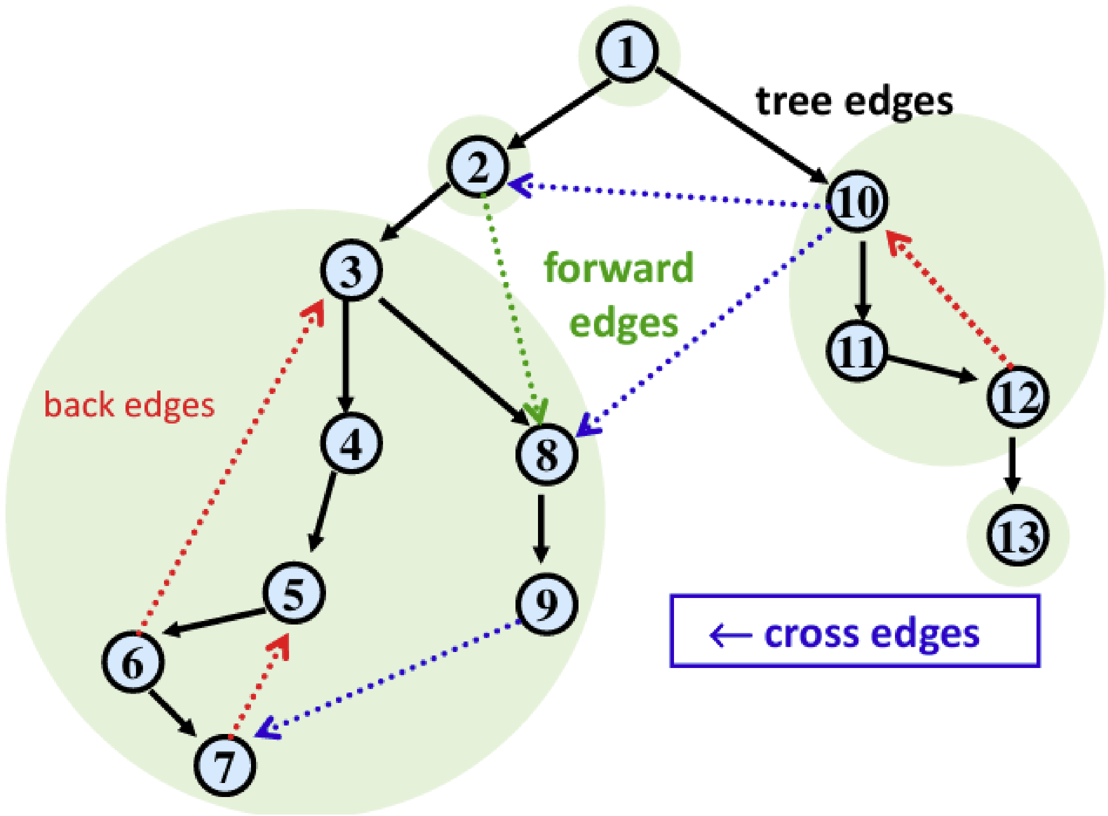
Defn: Vertices u and v are strongly connected iff they are on a directed cycle (there are paths from u to v and from v to u).

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

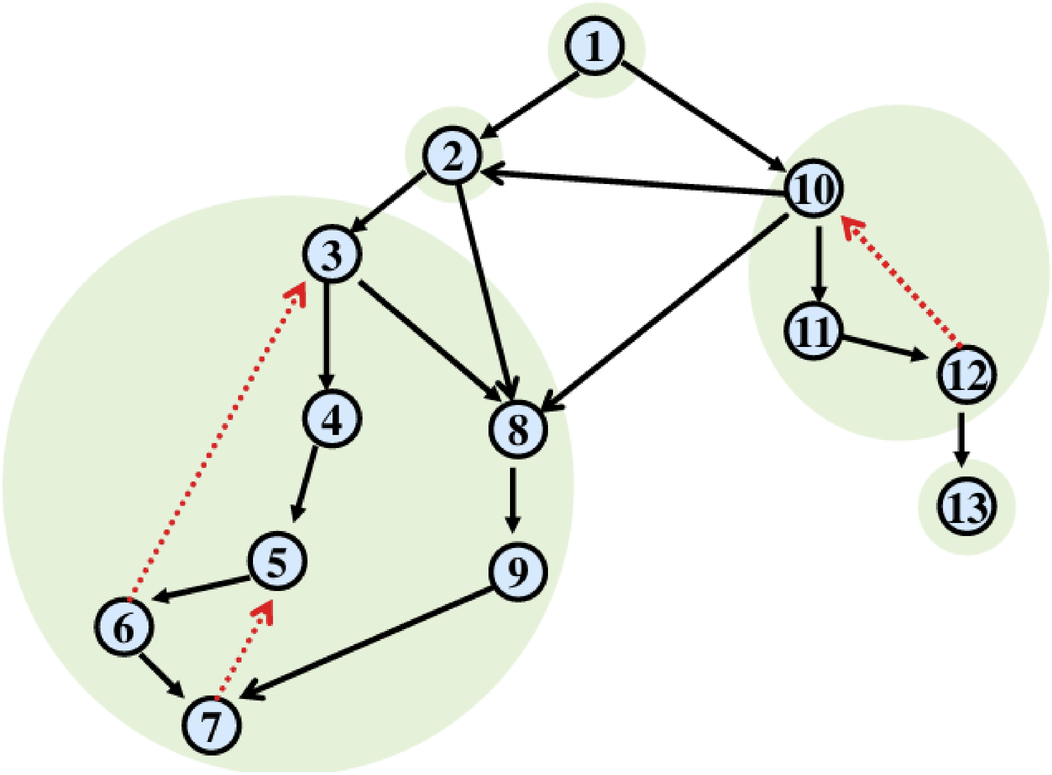
1. all pairs of vertices in the same component are strongly connected
2. can't merge components and keep property 1

- Strongly connected components can be stored efficiently just like connected components (e.g. the A[u] array)
- Can be found by extending DFS algorithm in $O(n + m)$ time using extra bookkeeping
  - We won't cover the details (Kosaraju's Algorithm)

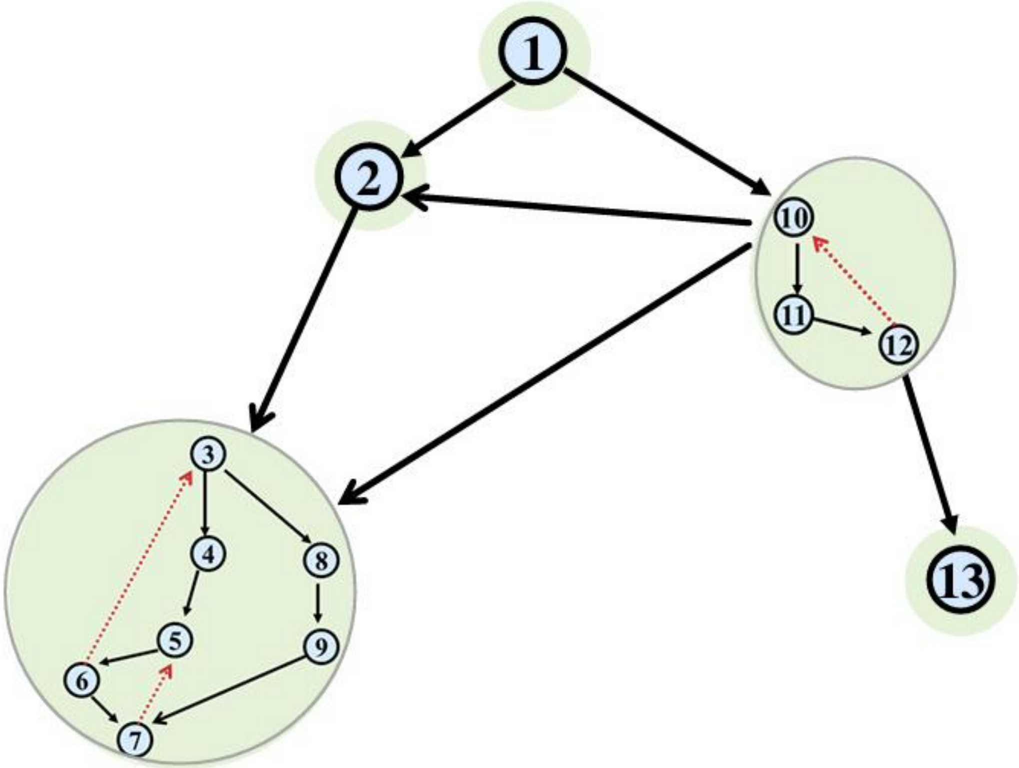
Strongly Connected Components



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:

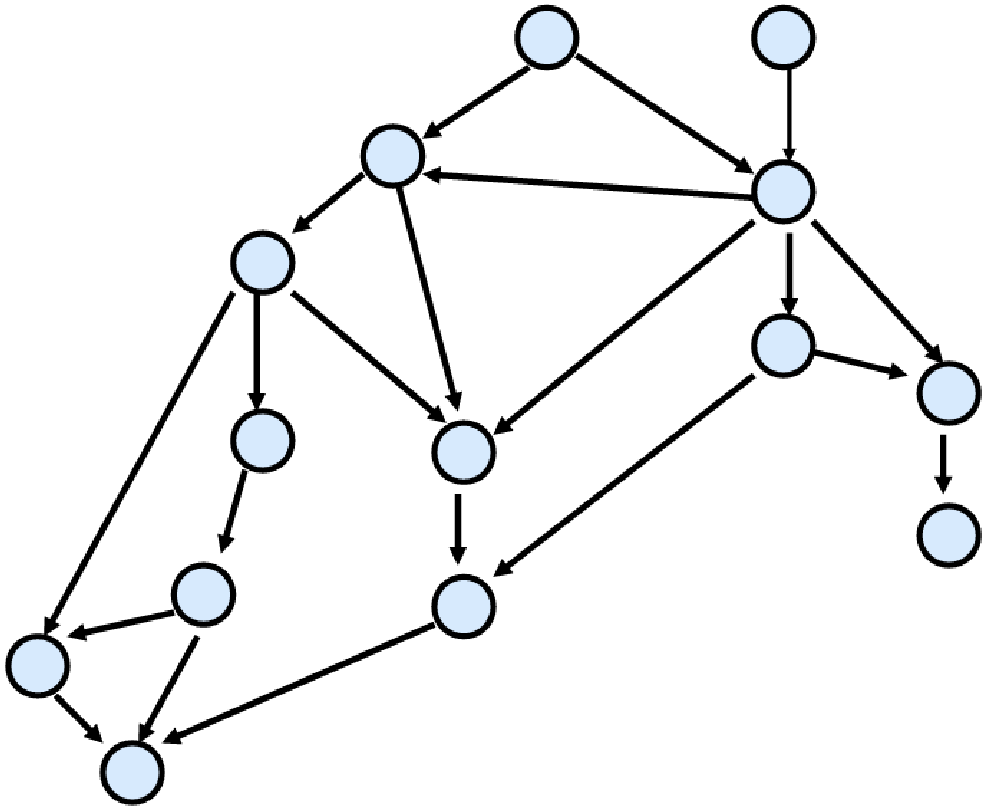
- nodes represent tasks
- edges represent precedence between tasks
- topological sort gives a sequential schedule for solving them

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.

UTAH TECH

Directed Acyclic Graph



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 ← some predecessor of v
```

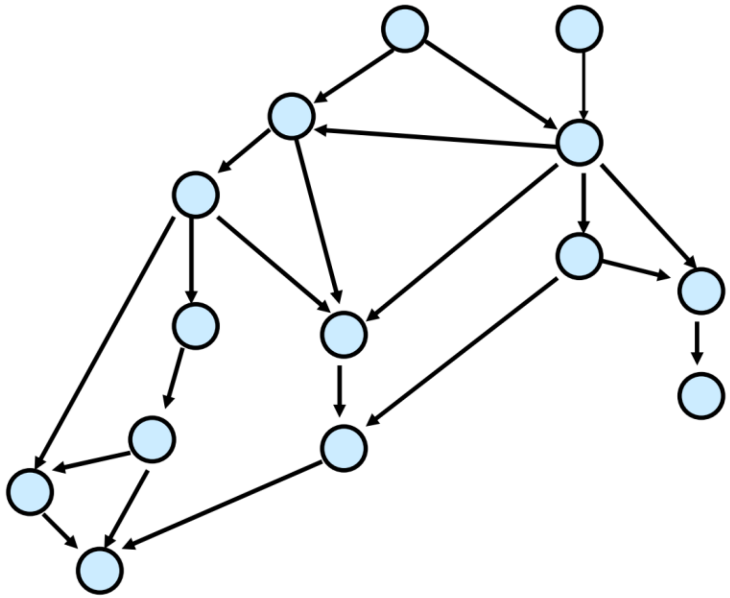
- After $n + 1$ steps where $n = |V|$ there will be a repeated vertex
  - This yields a cycle, contradicting that it is a DAG. ■

Topological Sort

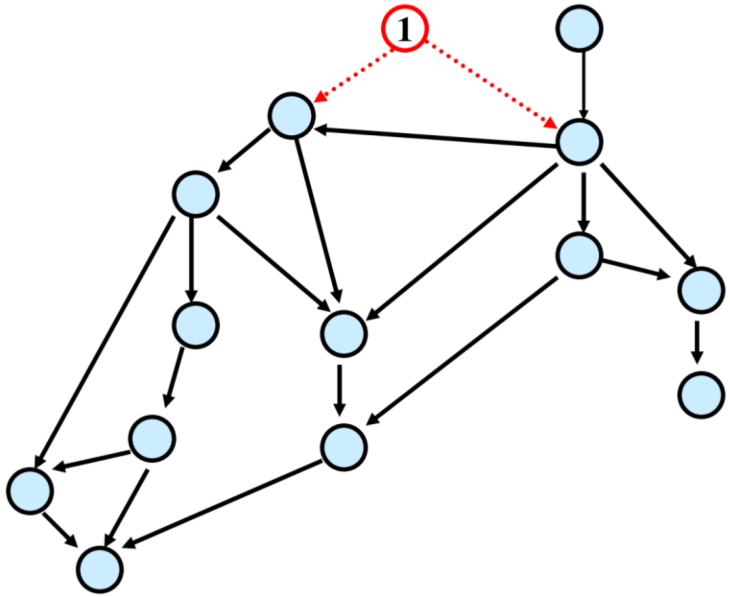
- Can do using DFS

- 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.

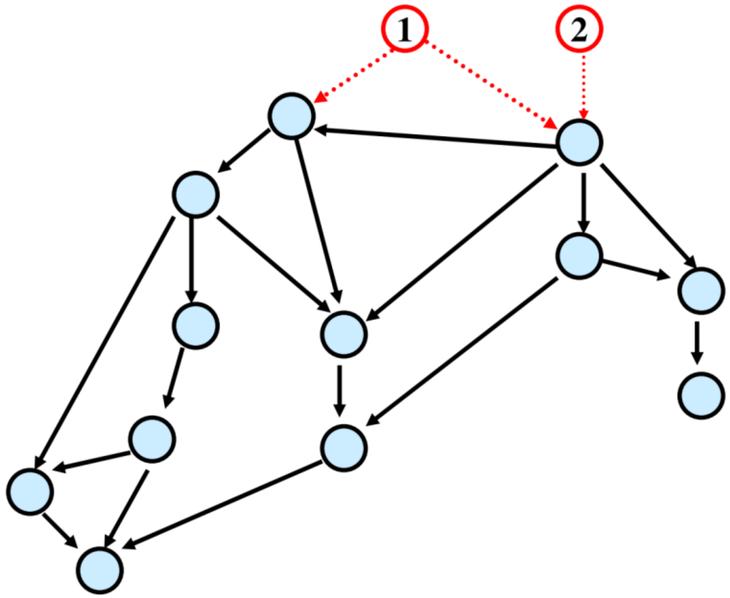
Topological Sort



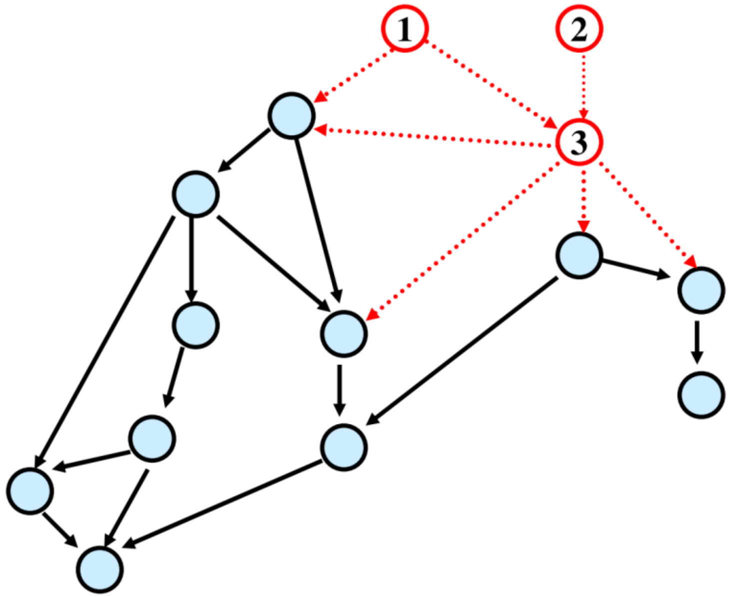
Topological Sort



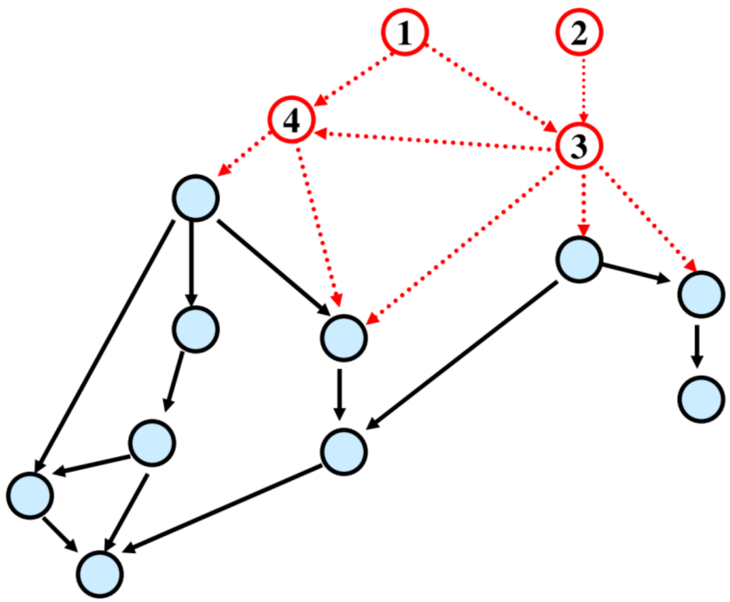
Topological Sort



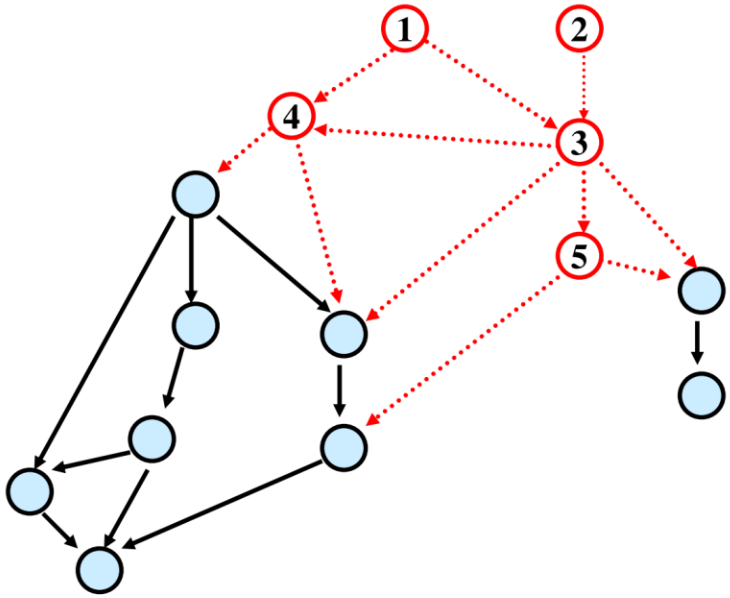
Topological Sort



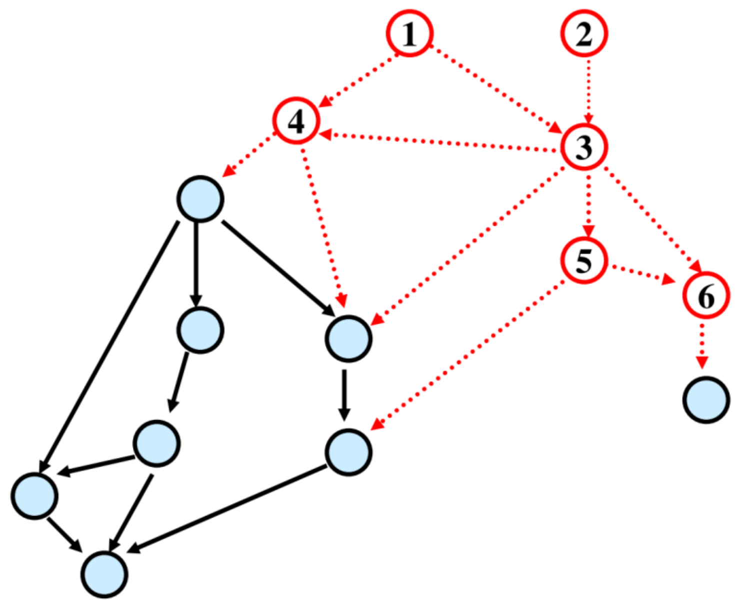
Topological Sort



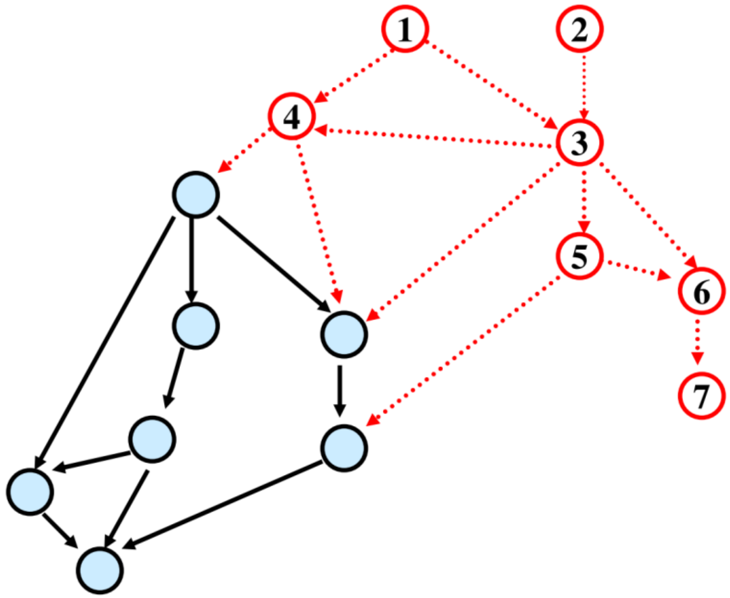
Topological Sort



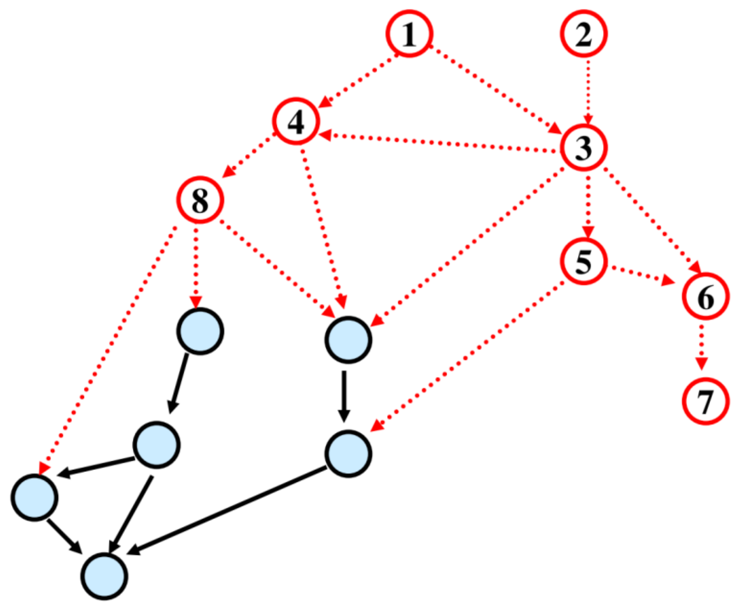
Topological Sort



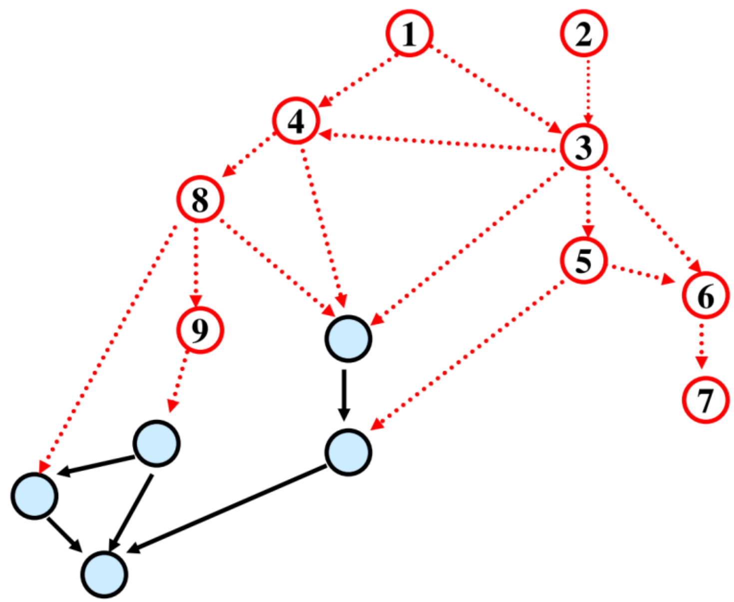
Topological Sort



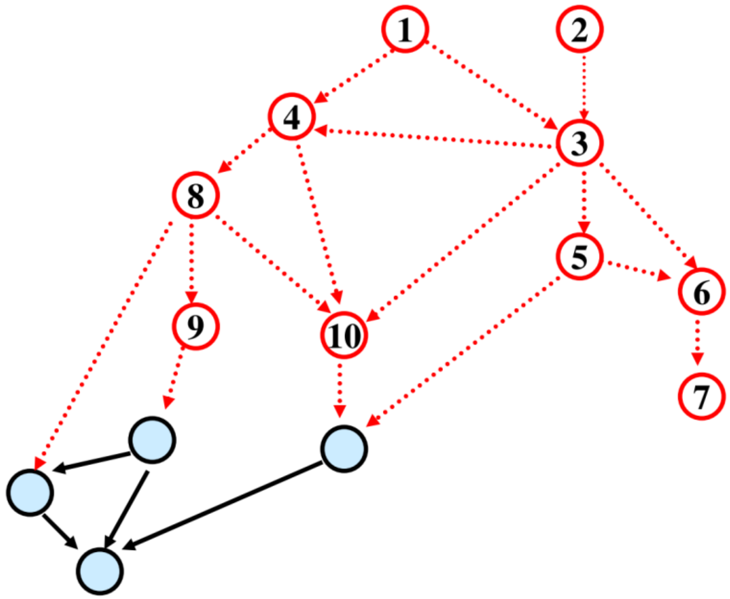
Topological Sort



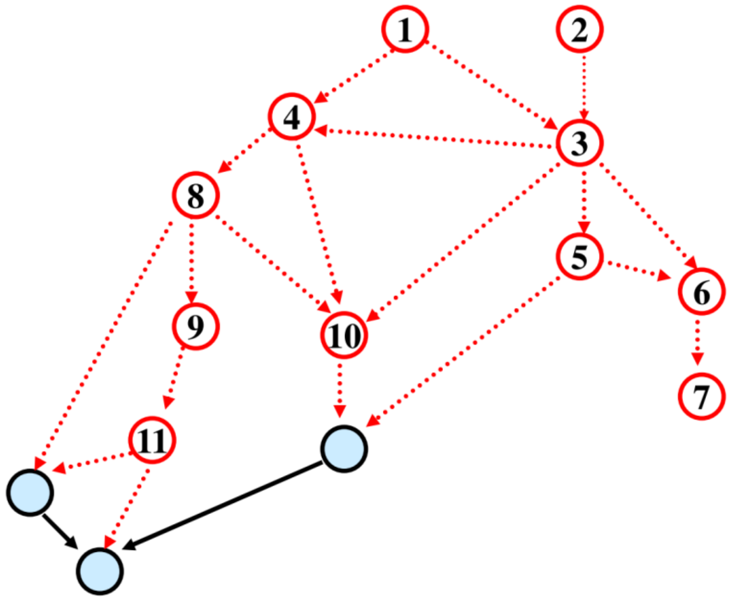
Topological Sort



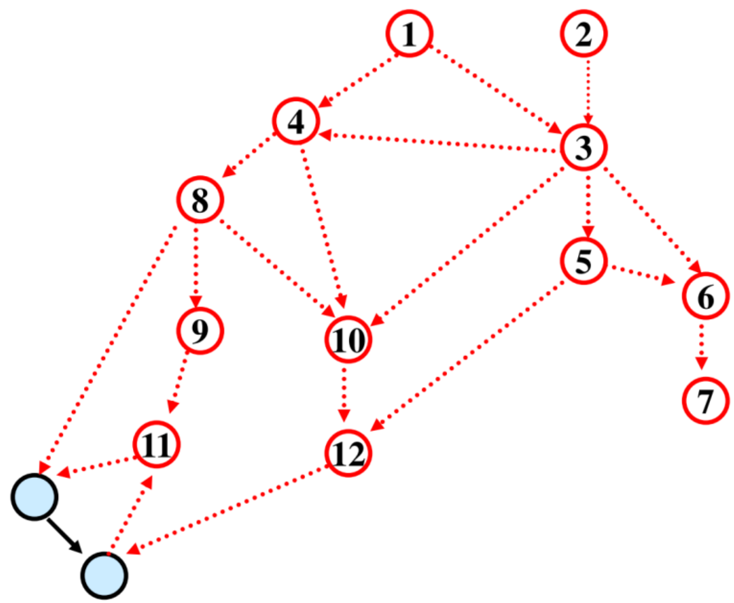
Topological Sort



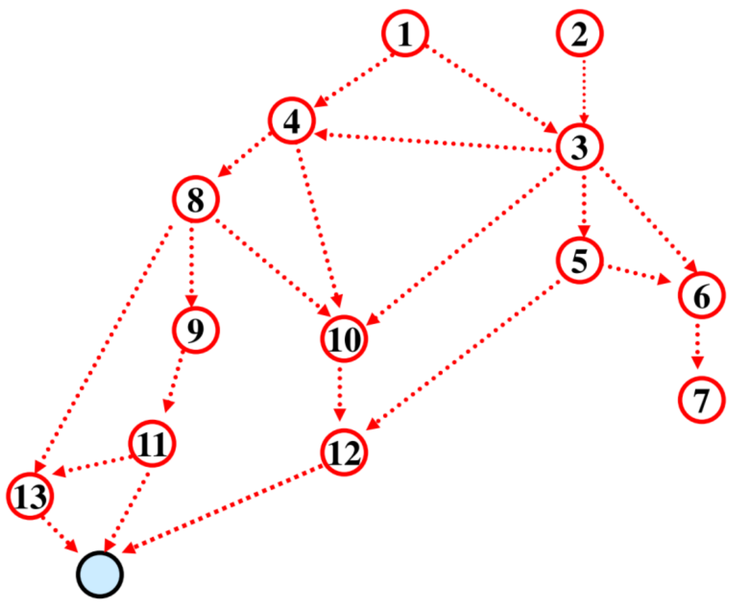
Topological Sort



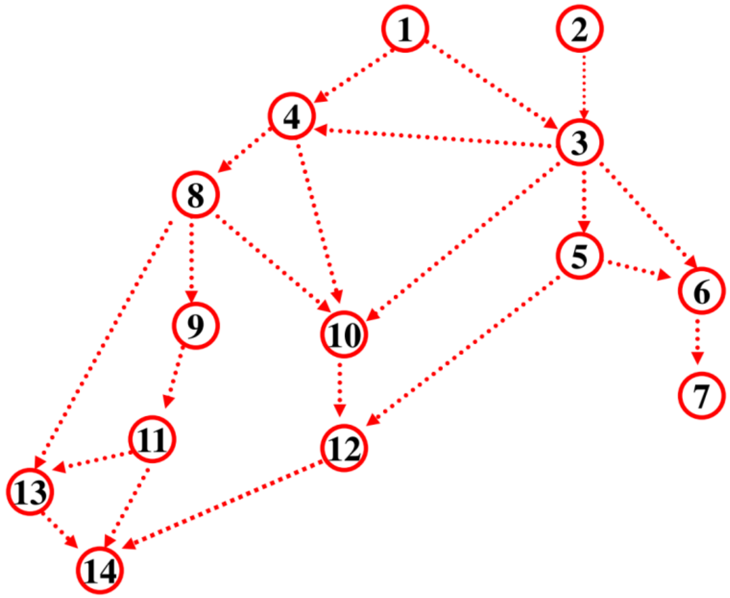
Topological Sort



Topological Sort



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: $O(m + n)$