Graphs and Depth-First Search

Representations, DFS, Components, Topological Sort, Cycles

Graphs

A graph $G = (V, E)$ is a set of vertices V and a set of edges E. Throughout,

$$n = |V|, \qquad m = |E|.$$

Graphs

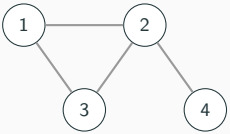
A graph $G = (V, E)$ is a set of vertices V and a set of edges E. Throughout,

$$n = |V|, \qquad m = |E|.$$

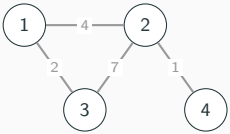
Graphs model pairwise relationships:

- the web: pages are vertices, links are edges;
- social networks: people are vertices, friendships are edges;
- road maps, flight routes, biological networks, ...

Four Kinds of Graphs



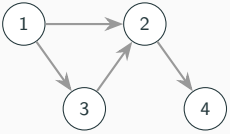
undirected, unweighted



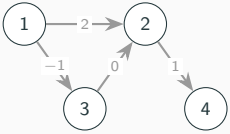
undirected, weighted

friendship

road distances



directed, unweighted



directed, weighted

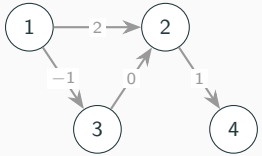
following

flight prices

Graph Representations

Adjacency Matrix

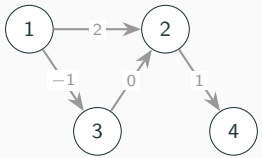
An $n \times n$ matrix with $A[i, j] =$ weight of edge $i \to j$, or $\emptyset$ if there is none. Undirected graphs have $A[i, j] = A[j, i]$.



$$A = \begin{bmatrix} \emptyset & 2 & -1 & \emptyset \\ \emptyset & \emptyset & \emptyset & 1 \\ \emptyset & 0 & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset & \emptyset \end{bmatrix}$$

Adjacency Matrix

An $n \times n$ matrix with $A[i, j] =$ weight of edge $i \to j$, or $\emptyset$ if there is none. Undirected graphs have $A[i, j] = A[j, i]$.



$$A = \begin{bmatrix} \emptyset & 2 & -1 & \emptyset \\ \emptyset & \emptyset & \emptyset & 1 \\ \emptyset & 0 & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset & \emptyset \end{bmatrix}$$

- Space: $O(n^2)$, wasteful when the graph is sparse.
- Is there an edge $i \to j$? $O(1)$.

Adjacency Matrix in Julia

Use `Inf` for "no edge":

```
A = [
    Inf 2.0 -1.0 Inf;
    Inf Inf Inf 1.0;
    Inf 0.0 Inf Inf;
    Inf Inf Inf Inf
]
```

Row i lists what vertex i points to. Most entries are `Inf`.

Adjacency List

For each vertex, store the list of its neighbours. For weighted graphs, store pairs (v, w): neighbour and edge weight.

$$L[1] = \{(2,\ 2),\ (3,\ -1)\}$$
$$L[2] = \{(4,\ 1)\}$$
$$L[3] = \{(2,\ 0)\}$$
$$L[4] = \emptyset$$

```
L = [
    [(2, 2), (3, -1)], # vertex 1
    [(4, 1)], # vertex 2
    [(2, 0)], # vertex 3
    [] # vertex 4
]
```

Adjacency List

For each vertex, store the list of its neighbours. For weighted graphs, store pairs (v, w): neighbour and edge weight.

$$L[1] = \{(2,\ 2),\ (3,\ -1)\}$$
$$L[2] = \{(4,\ 1)\}$$
$$L[3] = \{(2,\ 0)\}$$
$$L[4] = \emptyset$$

```
L = [
    [(2, 2), (3, -1)], # vertex 1
    [(4, 1)], # vertex 2
    [(2, 0)], # vertex 3
    [] # vertex 4
]
```

- Space: $O(n + m)$, proportional to the size of the graph.
- Is there an edge $u \to v$? $O(\deg(u))$: scan $L[u]$.

Matrix or List?

	matrix	list
space	$O(n^2)$	$O(n+m)$
is $u \to v$ an edge?	$O(1)$	$O(\deg(u))$
list neighbours of u	$O(n)$	$O(\deg(u))$

Real graphs are usually sparse ($m \ll n^2$), and traversals only ever ask for neighbours.

We use adjacency lists from now on.

Depth-First Search

Depth-First Search

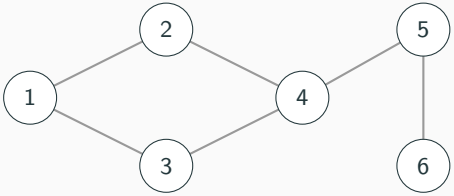
Starting from a source vertex, go as deep as possible along each branch, and backtrack only when stuck.

```
visited = falses(n)

function visit(adj, s)
    visited[s] = true # mark s as visited
    for v in adj[s] # for weighted graphs: for (v, _) in adj[s]
        if !visited[v]
            visit(adj, v) # go deeper
        end
    end
end # every neighbour handled: backtrack
```

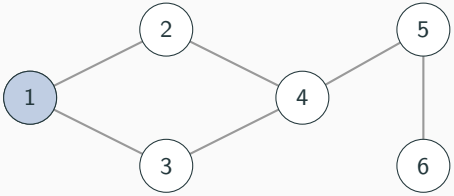
DFS in Action

Call `visit(adj, 1)`, scanning neighbours in increasing order.



DFS in Action

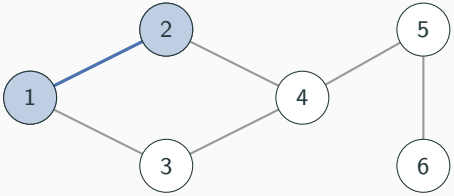
Call `visit(adj, 1)`, scanning neighbours in increasing order.



- visit 1

DFS in Action

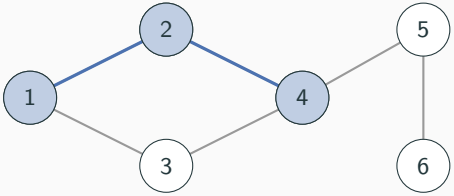
Call `visit(adj, 1)`, scanning neighbours in increasing order.



- visit 1
- 1's first neighbour 2: visit 2

DFS in Action

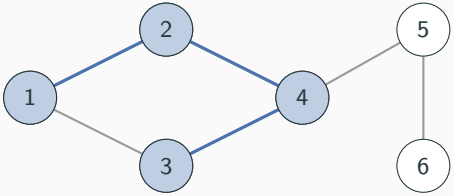
Call `visit(adj, 1)`, scanning neighbours in increasing order.



- visit 1
- 1's first neighbour 2: visit 2
- 2's neighbours 1, 4: visit 4

DFS in Action

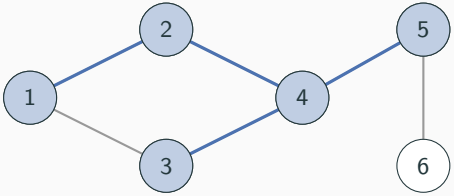
Call `visit(adj, 1)`, scanning neighbours in increasing order.



- visit 1
- 1's first neighbour 2: visit 2
- 2's neighbours 1, 4: visit 4
- 4's neighbours 2, 3, 5: visit 3
  3 has nothing new: backtrack to 4

DFS in Action

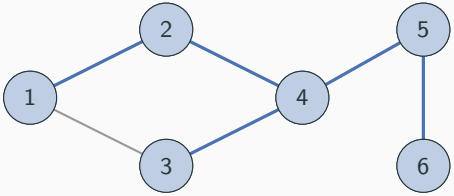
Call `visit(adj, 1)`, scanning neighbours in increasing order.



- visit 1
- 1's first neighbour 2: visit 2
- 2's neighbours 1, 4: visit 4
- 4's neighbours 2, 3, 5: visit 3
  3 has nothing new: backtrack to 4
- visit 5

DFS in Action

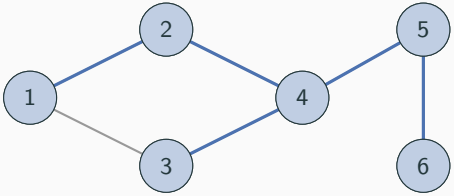
Call `visit(adj, 1)`, scanning neighbours in increasing order.



- visit 1
- 1's first neighbour 2: visit 2
- 2's neighbours 1, 4: visit 4
- 4's neighbours 2, 3, 5: visit 3
  3 has nothing new: backtrack to 4
- visit 5
- visit 6, then backtrack all the way

DFS in Action

Call `visit(adj, 1)`, scanning neighbours in increasing order.



- visit 1
- 1's first neighbour 2: visit 2
- 2's neighbours 1, 4: visit 4
- 4's neighbours 2, 3, 5: visit 3
  3 has nothing new: backtrack to 4
- visit 5
- visit 6, then backtrack all the way

Visit order $1, 2, 4, 3, 5, 6$. The blue edges form the DFS tree; edge 1–3 is never followed.

Running Time

- Each vertex is marked once, so `visit` is called at most n times: $O(n)$.

Running Time

- Each vertex is marked once, so `visit` is called at most n times: $O(n)$.
- The call `visit(adj, s)` scans the list `adj[s]` once. Over all vertices,

$$\sum_{s \in V} \deg(s) = \begin{cases} m & \text{directed,} \\ 2m & \text{undirected,} \end{cases}$$

  so the scanning costs $O(m)$.

Running Time

- Each vertex is marked once, so `visit` is called at most n times: $O(n)$.
- The call `visit(adj, s)` scans the list `adj[s]` once. Over all vertices,

$$\sum_{s \in V} \deg(s) = \begin{cases} m & \text{directed,} \\ 2m & \text{undirected,} \end{cases}$$

so the scanning costs $O(m)$.

DFS runs in $O(n + m)$ time: linear in the size of the graph.

Reachability

Question. Which vertices can be reached from s?

Answer. Call `visit(adj, s)`. Afterwards,

$$\texttt{visited[v]} = \texttt{true} \iff \text{there is a path from } s \text{ to } v.$$

Reachability

Question. Which vertices can be reached from s?

Answer. Call `visit(adj, s)`. Afterwards,

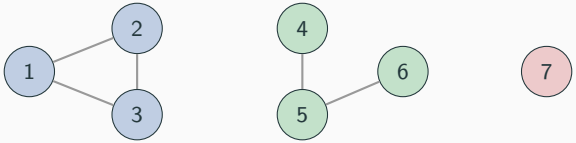
$$\texttt{visited[v]} = \texttt{true} \quad \iff \quad \text{there is a path from } s \text{ to } v.$$

- ($\Rightarrow$) DFS only moves along edges, so it only marks vertices it can walk to from s.
- ($\Leftarrow$) If some reachable v stayed unmarked, take the first unmarked vertex on a path $s \rightsquigarrow v$. Its predecessor on the path was marked, and its scan would have visited it. Contradiction.

Connected Components

Connected Components

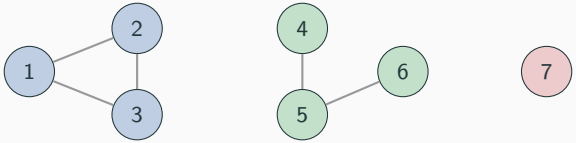
A connected component of an undirected graph is a maximal set of vertices in which every pair is joined by a path.



Three components: $\{1, 2, 3\}$, $\{4, 5, 6\}$, $\{7\}$.

Connected Components

A connected component of an undirected graph is a maximal set of vertices in which every pair is joined by a path.



Three components: $\{1, 2, 3\}$, $\{4, 5, 6\}$, $\{7\}$.

Key observation. `visit(adj, s)` marks exactly the vertices reachable from s, which is exactly the component of s.

Counting Components

Scan the vertices. Each time we find one not yet visited, it starts a new component: launch a fresh DFS from it.

counter $\leftarrow 0$
for each vertex $v \in V$:
  if not `visited[v]`:
    counter $\leftarrow$ *counter* $+ 1$
    `visit(adj, v)`

Counting Components

Scan the vertices. Each time we find one not yet visited, it starts a new component: launch a fresh DFS from it.

counter $\leftarrow 0$
for each vertex $v \in V$:
  if not `visited[v]`:
    counter $\leftarrow$ *counter* $+ 1$
    `visit(adj, v)`

- Each `visit` from the outer loop marks one whole component.
- Across *all* the calls, every vertex is visited once and every list is scanned once: still $O(n + m)$.

Connected Components in Julia

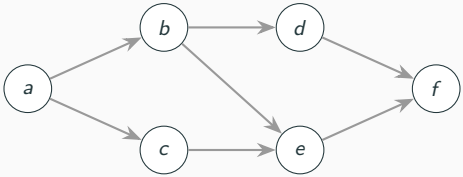
```julia
function connected_components(n, adj)
    visited = falses(n)
    component = zeros(Int, n) # component[v] = label of v's component
    num_components = 0
    function visit(s)
        visited[s] = true
        component[s] = num_components
        for v in adj[s]
            if !visited[v]
                visit(v)
            end
        end
    end
    for s in 1:n
        if !visited[s]
            num_components += 1
            visit(s)
        end
    end
    return num_components, component
end
```

Topological Sorting

Topological Sorting

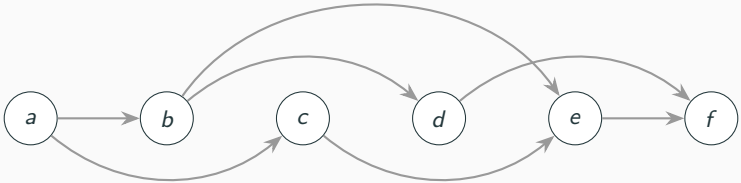
A topological order of a directed graph lists the vertices so that every edge $u \to v$ has u before v.

Think of tasks with dependencies: $u \to v$ means u must be done before v.



One Valid Order

Lay the vertices out as a, b, c, d, e, f:



Every edge points left to right.

Only DAGs Have One

Fact

A directed graph with a cycle has no topological order.

If $v_1 \to v_2 \to \cdots \to v_k \to v_1$ is a cycle, a topological order would need

$$v_1 \text{ before } v_2 \text{ before } \cdots \text{ before } v_k \text{ before } v_1.$$

Contradiction.

Only DAGs Have One

Fact

A directed graph with a cycle has no topological order.

If $v_1 \to v_2 \to \cdots \to v_k \to v_1$ is a cycle, a topological order would need

$$v_1 \text{ before } v_2 \text{ before } \cdots \text{ before } v_k \text{ before } v_1.$$

Contradiction.

So we only sort directed acyclic graphs (DAGs). Every DAG has a topological order, and DFS finds one.

The Algorithm

Run DFS. When a vertex finishes (all its descendants are done), put it at the front of the order.

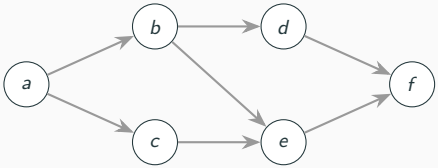
```
visited = falses(n)
topo_order = Int[]

function visit(adj, s)
    visited[s] = true
    for v in adj[s]
        if !visited[v]
            visit(adj, v)
        end
    end
    pushfirst!(topo_order, s) # s finishes: prepend it
end

for s in 1:n
    if !visited[s]
        visit(adj, s)
    end
end
```

Trace

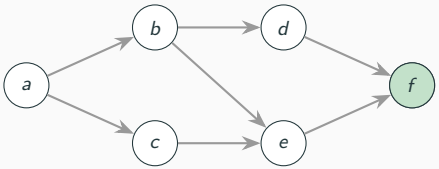
DFS from *a*, neighbours in alphabetical order.



event	topo_order

Trace

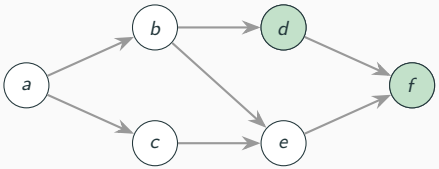
DFS from a, neighbours in alphabetical order.



event	`topo_order`
$a \to b \to d \to f$, f finishes	$[f]$

Trace

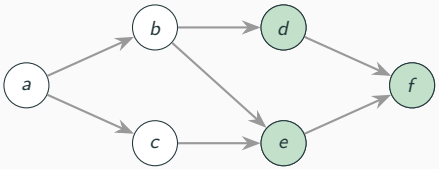
DFS from a, neighbours in alphabetical order.



event	`topo_order`
$a \to b \to d \to f$, f finishes	$[f]$
d finishes	$[d, f]$

Trace

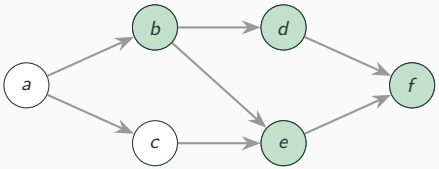
DFS from a, neighbours in alphabetical order.



event	`topo_order`
$a \to b \to d \to f$, f finishes	$[f]$
d finishes	$[d, f]$
$b \to e$ (f done), e finishes	$[e, d, f]$

Trace

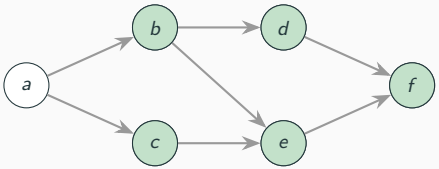
DFS from a, neighbours in alphabetical order.



event	`topo_order`
$a \to b \to d \to f$, f finishes	$[f]$
d finishes	$[d, f]$
$b \to e$ (f done), e finishes	$[e, d, f]$
b finishes	$[b, e, d, f]$

Trace

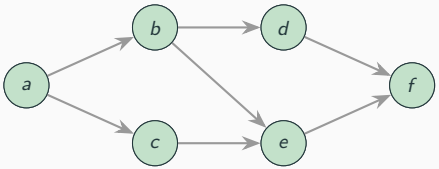
DFS from a, neighbours in alphabetical order.



event	`topo_order`
$a \to b \to d \to f$, f finishes	$[f]$
d finishes	$[d, f]$
$b \to e$ (f done), e finishes	$[e, d, f]$
b finishes	$[b, e, d, f]$
$a \to c$ (e done), c finishes	$[c, b, e, d, f]$

Trace

DFS from a, neighbours in alphabetical order.



event	`topo_order`
$a \to b \to d \to f$, f finishes	$[f]$
d finishes	$[d, f]$
$b \to e$ (f done), e finishes	$[e, d, f]$
b finishes	$[b, e, d, f]$
$a \to c$ (e done), c finishes	$[c, b, e, d, f]$
a finishes	$[a, c, b, e, d, f]$

Check: every edge goes left to right in a, c, b, e, d, f.

Why It Works

Claim

In a DAG, every edge $u \to v$ has v finishing before u, so u is placed in front of v.

Look at the moment DFS scans the edge $u \to v$ inside `visit(u)`.

- v **not yet visited.** We call `visit(v)` and it returns before u finishes.

Why It Works

Claim

In a DAG, every edge $u \to v$ has v finishing before u, so u is placed in front of v.

Look at the moment DFS scans the edge $u \to v$ inside `visit(u)`.

- v **not yet visited.** We call `visit(v)` and it returns before u finishes.
- v **already finished.** Then v finished first. Done.

Why It Works

Claim

In a DAG, every edge $u \to v$ has v finishing before u, so u is placed in front of v.

Look at the moment DFS scans the edge $u \to v$ inside `visit(u)`.

- v **not yet visited.** We call `visit(v)` and it returns before u finishes.
- v **already finished.** Then v finished first. Done.
- v **visited but not finished.** Then v is still on the call stack below u, so there is a path $v \rightsquigarrow u$. With $u \to v$ that is a cycle: impossible in a DAG.

Why It Works

Claim

In a DAG, every edge $u \to v$ has v finishing before u, so u is placed in front of v.

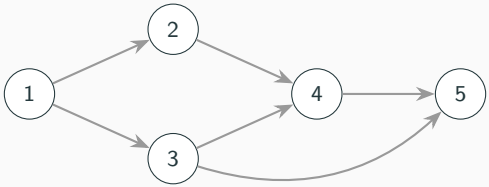
Look at the moment DFS scans the edge $u \to v$ inside `visit(u)`.

- v **not yet visited.** We call `visit(v)` and it returns before u finishes.
- v **already finished.** Then v finished first. Done.
- v **visited but not finished.** Then v is still on the call stack below u, so there is a path $v \rightsquigarrow u$. With $u \to v$ that is a cycle: impossible in a DAG.

One DFS over the whole graph: $O(n + m)$.

Exercise

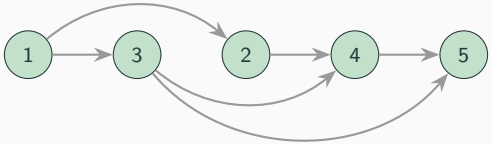
Run the DFS algorithm (neighbours in increasing order, outer loop $1, \ldots, 5$) and give the topological order it produces.



Solution

- $1 \to 2 \to 4 \to 5$: finish 5, then 4, then 2.
- Back at 1: $1 \to 3$. Both 4 and 5 are done, so 3 finishes, then 1.

Finish order $5, 4, 2, 3, 1$. Reversed:



$1, 2, 3, 4, 5$ is also valid. A DAG can have many topological orders.

Detecting Cycles

Pre and Post Numbers

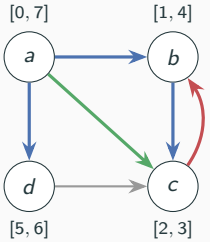
Keep a clock. Stamp each vertex when DFS enters it (*pre*) and when it leaves it (*post*).

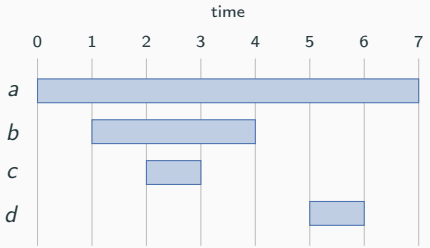
```
time = 0

function visit(adj, s)
    pre[s] = time; time += 1
    visited[s] = true
    for v in adj[s]
        if !visited[v]
            visit(adj, v)
        end
    end
    post[s] = time; time += 1
end
```

$[pre[v], post[v]]$ is the time interval v spends on the call stack.

Intervals Nest





DFS from *a*, neighbours in alphabetical order. Two intervals are either nested (one vertex is a descendant of the other) or disjoint. They never partially overlap.

Classifying Edges

For an edge $u \to v$:

edge type	intervals	example
tree / forward	$pre[u] < pre[v] < post[v] < post[u]$	$a \to b$, $a \to c$
back	$pre[v] < pre[u] < post[u] < post[v]$	$c \to b$
cross	$pre[v] < post[v] < pre[u] < post[u]$	$d \to c$

- Tree edge: DFS discovers v through it. Forward edge: v is a descendant found another way.
- Back edge: v is an ancestor of u, still on the stack.

Classifying Edges

For an edge $u \to v$:

edge type	intervals	example
tree / forward	$pre[u] < pre[v] < post[v] < post[u]$	$a \to b$, $a \to c$
back	$pre[v] < pre[u] < post[u] < post[v]$	$c \to b$
cross	$pre[v] < post[v] < pre[u] < post[u]$	$d \to c$

- Tree edge: DFS discovers v through it. Forward edge: v is a descendant found another way.
- Back edge: v is an ancestor of u, still on the stack.

Only back edges have $post[u] < post[v]$.

Cycles and Back Edges

Claim

A directed graph has a cycle $\iff$ DFS finds a back edge.

($\Leftarrow$) A back edge $u \to v$ means v is an ancestor of u in the DFS tree, so there is a path $v \rightsquigarrow u$. Adding $u \to v$ closes a cycle.

Cycles and Back Edges

Claim

A directed graph has a cycle $\iff$ DFS finds a back edge.

($\Leftarrow$) A back edge $u \to v$ means v is an ancestor of u in the DFS tree, so there is a path $v \rightsquigarrow u$. Adding $u \to v$ closes a cycle.

($\Rightarrow$) Let $v_1 \to v_2 \to \cdots \to v_k \to v_1$ be a cycle, and let v_1 be the first of its vertices that DFS visits. At that moment the rest of the cycle is unvisited, so DFS reaches v_k before v_1 finishes:

$$post[v_k] < post[v_1].$$

So the edge $v_k \to v_1$ is a back edge. $\square$

Cycle Detection

Compute pre/post numbers, then check every edge:

```
function has_cycle(n, adj)
    pre, post = dfs_prepost(n, adj) # DFS from every unvisited vertex
    for u in 1:n
        for v in adj[u]
            if post[u] < post[v] # back edge
                return true
            end
        end
    end
    return false
end
```

- One DFS: $O(n + m)$. One pass over the edges: $O(m)$.
- Total: $O(n + m)$.

A graph is a DAG exactly when this returns `false`, so we can check before topologically sorting.