A directed graph is a DAG if and only if a depth-first search encounters no back edges.

**Lemma 6.7**

**DFS APPLICATION:**

**TESTING WHETHER A GRAPH IS A DAG**

A directed graph $G$ is a directed acyclic graph, or DAG, if $G$ contains no directed cycle.

Case (⇐): Suppose $\exists$ a directed cycle. Show $\exists$ a back edge.

- Let $v_1, v_2, \ldots, v_k, v_1$ be a directed cycle
- WLOG let $v_1$ be the earliest discovered node in the cycle

- Consider edge $\{v_k, v_1\}$
- Since $d[v_1] < d[v_k]$, $\{v_k, v_1\}$ must be a back or cross edge

**TURNING THE LEMMA INTO AN ALGORITHM**

- Search for back edges

**DFS: TESTING WHETHER A GRAPH IS A DAG**

1. $\text{DFAssert}(\text{adj}[\cdot, \cdot], v)$
2. $\text{pred}[\cdot] = \text{null}, \text{null}, \ldots, \text{null}$
3. $\text{colour}[] = \text{white}, \text{white}, \ldots, \text{white}$
4. $\text{dfn}[] = \{0, 0, \ldots, 0\}$ // discovery times
5. $\text{fin}[] = \{0, 0, \ldots, 0\}$ // finish times
6. $\text{time} = 0$
7. $\text{USE} = \text{true}$
8. $\text{IsDAG}()$
9. $\text{for } w \in \text{adj}[v]$
10. $\text{if colour}[w] = \text{white}$
11. $\text{pred}[w] = v$
12. $\text{DFAssert(adj[,], w)}$
13. $\text{if colour}[v] = \text{gray}$
14. $\text{DAG} = \text{false}$
15. $\text{colour}[v] = \text{black}$
16. $\text{time} = \text{time} + 1$
17. $\text{d}[v] = \text{time}$
18. $\text{for each w in adj[v]}$
19. $\text{if colour}[w] = \text{white}$
20. $\text{pred}[w] = v$
21. $\text{DFAssert(adj[,], w)}$
22. $\text{if colour}[v] = \text{gray}$
23. $\text{DAG} = \text{false}$
24. $\text{colour}[v] = \text{black}$
25. $\text{time} = \text{time} + 1$
26. $\text{f}[v] = \text{time}$
EXAMPLE

Back edge found! So we set DAG = false

TOPOLOGICAL SORT
Finding node orderings that satisfy given constraints

DEPENDENCY GRAPH

Edges (u, v) mean u must be completed before v

Example problem: getting dressed in the morning

Socks before shoes
Watch any time
Pants before belt

Could do various things first. Which ones are possible? What do they have in common?

Topological sort
Try to order nodes linearly so there are only pointers from left to right!

MIGHT NOT BE POSSIBLE! How can this happen?

IFF there is a (directed) cycle!

FORMAL DEFINITION
A directed graph G = (V, E) has a topological ordering, or topological sort, if there is a linear ordering < of all the vertices in V such that u < v whenever u, v ∈ E.

More realistic use case: task scheduling
Goal: output a serial order of tasks that can be run without worrying about dependencies (because all dependencies are already satisfied by the time a task is run).

Can even schedule tasks to run in parallel! Can do 1|2 then 3|4 then 5|6 etc.

(Nodes are numbered according to one such order.)

Graph G

Topological sort of G

Edges are directed only left-to-right in this ordering
**USEFUL FACT**

**Lemma 6.5**

A DAG contains a vertex of indegree 0.

**Proof.**

Suppose we have a directed graph in which every vertex has positive indegree. Let \( v_i \) be any vertex. For every \( j \geq 1 \), let \( v_{i+j} \) be an arc. In the sequence \( v_i, v_2, v_3, \ldots \), consider the first repeated vertex, \( v_i = v_j \) where \( j > i \). Then \( v_{i+j}, v_{i+2j}, \ldots, v_j \) is a directed cycle.

**EXISTENCE OF A TOPOLOGICAL SORT ORDER**

**Theorem 6.6**

A directed graph \( D \) has a topological sort if and only if it is a DAG.

**Proof.**

(i) Suppose \( D \) has a directed cycle \( v_1, v_2, \ldots, v_k, v_1 \). Then \( v_1 < v_2 < \cdots < v_k < v_1 \), so a topological ordering does not exist.

(ii) Suppose \( D \) is a DAG. Then the algorithm below constructs a topological ordering.

```
1 Kahn's adj[1..n]
2 indeg[1..n] = [0, 0, ..., 0]
3 for each edge (v,w) in adj
4   indeg[w] = indeg[w] + 1
5 order = new list
6 queue Q
7 indeg[w] = indeg[w] - 1
8 if indeg[w] = 0 then q.enqueue(w)
9 order.append(w)
10 return order
```

**EXAMPLE (KAHN'S ALGORITHM)**

Compute indeg of all vertices

```
1 Kahn's adj[1..n]
2 indeg[1..n] = [0, 0, ..., 0]
3 for each edge (v,w) in adj
4   indeg[w] = indeg[w] + 1
5 order = new list
6 for i = 1..n
7   if q.empty() return NULL
8     v = q.dequeue()
9     order.append(v)
10    for each w in adj[v]
11       indeg[w] = indeg[w] - 1
12       if indeg[w] = 0 then q.enqueue(w)
13 return order
```

**TOPOLOGICAL SORT VIA DFS**

We can also implement topological sort by using DFS!

The **finishing times** of nodes help us understand this algo will be key for understanding strongly connected components.
Suppose $D$ is a DAG. Then $f(v) < f(u)$ for every arc $(u,v)$.

To see why, suppose $D$ is a DAG and we order nodes in this way, so $f(v_1) > f(v_2) > \ldots > f(v_{n-1}) > f(v_n)$. Since edge $(u,v)$ exists, the lemma implies $f(u) < f(v)$. But this contradicts the node ordering! So all edges are left-to-right, hence it is a topological sort.

Theorem: If $D$ is a DAG, and we order vertices in reverse order of finishing time, (i.e., by largest to smallest finish time) then we get a topological ordering!

Recall from DAG-finding: there are no back edges in a DAG.

For contradiction, suppose a right-to-left edge $(u,v)$ exists. Since edge $(u,v)$ exists, the lemma implies $f(v) < f(u)$. But this contradicts the node ordering! So all edges are left-to-right, hence this is a topological sort.

**HOME EXERCISE: RUN ON THIS GRAPH**

The initial calls are DFSVisit(1), DFSVisit(2) and DFSVisit(3).

The discovery/finish times are as follows:

<table>
<thead>
<tr>
<th>$v$</th>
<th>$d(v)$</th>
<th>$f(v)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>

The topological ordering is 3, 2, 5, 4, 1, 6 (reverse order of finishing time).

**STRONGLY CONNECTED COMPONENTS**

This graph could be divided into two graphs that are each strongly connected.

These are called strongly connected components (SCC).

**STRONGLY CONNECTED COMPONENTS**

- It could also be divided into three graphs...

- But we want our SCCs to be maximal (as large as possible)
STRONGLY CONNECTED COMPONENTS

- So, the goal is to find these (maximal) SCCs:

![Graph with strongly connected components]

APPLICATIONS OF SCCs AND COMPONENT GRAPHS

- Finding all cyclic dependencies in code
  - Can find single cycle with an easier DFS-based algorithm
  - But it is nicer to find all cycles at once, so you don’t have to fix one to expose another

APPLICATIONS OF SCCs AND COMPONENT GRAPHS

- Data filtering before running other algorithms
  - Consider Google maps: nodes = intersections, edges = roads
  - Don’t want to run path finding algorithm on the entire global graph!
  - First restrict execution to a rectangle
  - Then throw away everything except the (maximal) SCC containing source & target

COMPONENT GRAPH

- Consider this graph
- These are its SCCs

![Component graph]

FORMALLY DEFINING SCC

- For two vertices $u$ and $v$ of $G$, define $u \sim v$ if $u = v$, or if $u \neq v$ and there exist directed paths from $u$ to $v$ and from $v$ to $u$.
  - The relation $\sim$ is an equivalence relation.
  - The strongly connected components of $G$ are the equivalence classes of vertices defined by the relation $\sim$.
  - A strongly connected component of a digraph $G$ is a maximal strongly connected subgraph of $G$.

BRAINSTORMING AN ALGORITHM

- What if we run DFS, then reverse all edges, then run DFS (like checking whether an entire graph is strongly connected?)

![DFS algorithm diagram]
What if we run DFS, then reverse all edges, then run DFS?

Consider component graph \( G_0 \) of \( G \) (which we want to compute).

If we call DFSVisit in \( G_0 \) from largest to smallest finish times, we can reach other SCCs.

However, when we reverse the edges to get graph \( H \), calling DFSVisit on nodes ordered from largest to smallest finish times, we cannot reach other (unvisited) SCCs!

This is called Shoham’s algorithm (sometimes Kosaraju’s algorithm).

This paper first introduced it.

Want to prove that each top-level call to SCCVisit explores exactly the nodes in one SCC.

Proof hinges on a key lemma that talks about the finish times of SCCs in the component graph.

To talk about finish times of SCCs, we need a definition...

TIME COMPLEXITY?

**SCC ALGORITHM**

1. \( SCC(adj[]: n) \)
2. \( DPS(adj[]) \)
3. let order[1..n] = node labels sorted by largest to smallest finish time
4. reverse all edges in adj
5. colour[1..n] = [white, ..., white]
6. comp[1..n] = [0, ..., 0]
7. for \( i = 1..n \)
8. \( v = \text{order[i]} \)
9. if \( \text{colour[v]} = \text{white} \)
10. \( \text{SCCVisit(adj), v, comp, colour} \)
11. \( \text{DFSVisit(adj), v, comp, colour} \)
12. return comp

**CORRECTNESS**

Want to prove that each top-level call to SCCVisit explores exactly the nodes in one SCC.

Proof hinges on a key lemma that talks about the finish times of SCCs in the component graph.

To talk about finish times of SCCs, we need a definition...

TIME COMPLEXITY?

1. \( DCS(adj[]: n) \)
2. \( DS(adj[]) \)
3. let order[1..n] = node labels sorted by largest to smallest finish time
4. reverse all edges in adj
5. colour[1..n] = [white, ..., white]
6. comp[1..n] = [0, ..., 0]
7. for \( i = 1..n \)
8. \( v = \text{order[i]} \)
9. if \( \text{colour[v]} = \text{white} \)
10. \( \text{SCCVisit(adj), v, comp, colour} \)
11. \( \text{DFSVisit(adj), v, comp, colour} \)
12. return comp
For a strongly connected component $C$, let $d[C] = \min\{d[v] : v \in C\}$ and $f[C] = \max\{f[v] : v \in C\}$.

Since component graph is a DAG, there is $\bigcup \mathcal{C} < \mathcal{C} \in \mathcal{C} > j$. So, there is no edge $c,4 \rightarrow \mathcal{G}$. Let $f_5$ be the earliest discovered node in $\mathcal{C}$. Proof. $\mathcal{C}$ in $\min f_5 \rightarrow \max f_7$. So, we discover $\mathcal{G}$ and hence $c,4 \rightarrow \mathcal{G}$. Therefore $\mathcal{C}$ is discovered first.

Lemma: If $C_i, C_j$ are SCCs and there is an edge $C_i \rightarrow C_j$ in $\mathcal{G}$, then $f[C_i] > f[C_j]$.

Proof. Case 1 ($d[C_i] < d[C_j]$):
- Let $u$ be the earliest discovered node in $C_i$. All nodes in $C_i \cup C_j$ are reachable from $u$, so they are descendants in the DFS forest and finish before $u$.
- So $f[C_i] = f[u] > f[C_j]$.

A KEY LEMMA

A KEY DEFINITION

For a strongly connected component $C$, let $d[C] = \min\{d[v] : v \in C\}$ and $f[C] = \max\{f[v] : v \in C\}$.

Since component graph is a DAG, there is $\bigcup \mathcal{C} < \mathcal{C} \in \mathcal{C} > j$. Proof. Consider the first top level call visits precisely one SCC.

We prove each top-level SCCVisit call visits precisely one SCC.

Let $C$ be the SCC containing $u$ and $C'$ be any other SCC.

Since we call SCCVisit on nodes starting from the largest finish time,
- We know $f(C) > f(C')$.

COMPLETING THE PROOF

We know $f(C) > f(C')$.

By Lemma: if there were an edge $C' \rightarrow C$ in $\mathcal{G}$, then we would have $f(C') > f(C)$.

So there is no edge $C' \rightarrow C$ in $\mathcal{G}$ and hence no edge $C \rightarrow C'$ in $\mathcal{G}$.

So, SCCVisit($u$) in $\mathcal{H}$ cannot visit $C'$. $\therefore$