Week 9: Topological Sort & DFS

Algorithms & Data Structures

Thorben Klabunde

th-kl.ch

November 17, 2025

Agenda

Mini-Quiz

Let $G = (V, E)$ be *connected* and assume *all vertices of G have even degree*. Recall the algorithm $\mathrm{Euler\_Walk}$ from the lecture:

$\mathrm{Euler\_Walk}(u)$:

if there exists an edge $\{u, v\}$ which is not marked:

mark the edge $\{u, v\}$

$\mathrm{Euler\_Walk}(v)$

Suppose we start the recursion with a call to $\mathrm{Euler\_Walk}(u_0)$ for some vertex $u_0 \in V$.

True	False			
		We have an even number of marked edges which contain u_0 immediately after the start of any call to $\mathrm{Euler\_Walk}(u_0)$.		
		Let $u' \in V$ with $u' \neq u_0$, then we have an even number of marked edges which contain u' immediately after the start of any call to $\mathrm{Euler\_Walk}(u')$.		

Scoring method: **Subpoints**

Q1

For the initial vertex u_0, since we start the recursion there, there must be either 0 edges marked (first ever call) or there is an ingoing edge for every outgoing edge, so an even number of marked edges.
For every other vertex $u' \neq u_0$, we have an odd number of edges marked because the first call to u' must have one edge marked already, then the same reasoning as for u_0 means everytime we call $\mathrm{Euler\_Walk}(u')$ we have +2 edges marked.

We have an even number of marked edges which contain u_0 immediately after the start of any call to $\mathrm{Euler\_Walk}(u_0)$.
: True
Let $u' \in V$ with $u' \neq u_0$, then we have an even number of marked edges which contain u' immediately after the start of any call to $\mathrm{Euler\_Walk}(u')$.
: False

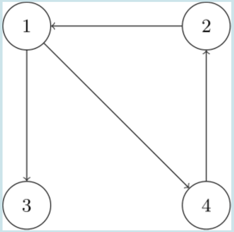
Suppose that $G = (V, E)$ has an Eulerian Walk. Then we can always find an Eulerian Walk of G in time $O(|V|)$.

- True
- False

If we have graphs with $|E| = \Omega(|V|^2)$ then we can't even output all edges in the graph, much less an Eulerian Walk.

The correct answer is 'False'.

Q3



What adjacency matrix describes the graph above? The indices are in increasing numerical order.

$$\begin{pmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Q3

The correct answer is: $\begin{pmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$

Suppose we restrict our inputs to graphs $G = (V, E)$ which are *connected* and satisfy $|E| \geq |V|^2/10$, rather than the class of all graphs. Then, depth-first search (DFS) runs (in the worst case over such graphs) asymptotically in the same time regardless of whether the graph is stored as adjacency lists or as an adjacency matrix.

- True ✓
- False

With adjacency lists we have a runtime of $O(|V| + |E|)$ and an adjacency matrix its $O(|V|^2)$ but plugging in $|V|^2 \geq |E| \geq |V|^2/10$, we can see that these have the same asymptotic runtime.

The correct answer is 'True'.

Q5

Let G be a directed graph that contains a (directed) cycle. Which of the following statements are true for all such G?

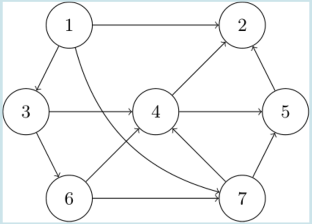
True	False			
		G has no source vertex.		
		G has no sink vertex.		
		G does not have a topological sorting of its vertices.		

Q5

In lecture, we saw that a directed graph with a cycle cannot admit a toplogical sorting. However, if we can have vertices not part of the cycle which act as sources or sinks, for example, consider a directed graph plus an isolated vertex.

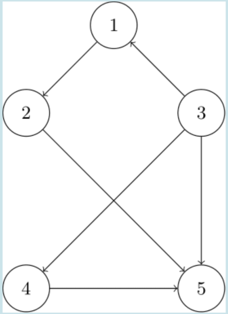
G has no source vertex.
: False
G has no sink vertex.
: False
G does not have a topological sorting of its vertices.
: True

Q6



Which vertex is at the end of every topological sorting of the graph above?

1. 1
2. 2 ✓



In every topological sorting of the graph above, vertex 1 must come before vertex 4.

- True
- False

Q8

We run *depth-first-search* (DFS) on a directed graph $G = (V, E)$ which contains a directed cycle. We determine all the *pre/post numbers*.
What can we conclude about the relationship of pre/post numbers?

True	False			
○ ⊗	◉	For every edge $(u, v) \in E$, we have $\mathrm{pre}(v) < \mathrm{pre}(u) < \mathrm{post}(u) < \mathrm{post}(v)$.	✓	In lecture, we saw these are only back edges. Not all edges can be back edges, since tree edges are not back edges.
◉	○ ⊗	There exists some edge $(u, v) \in E$ which has $\mathrm{pre}(v) < \mathrm{pre}(u) < \mathrm{post}(u) < \mathrm{post}(v)$.	✓	With a directed cycle, we must have a back edge, which is exactly this condition.

Enter Caption

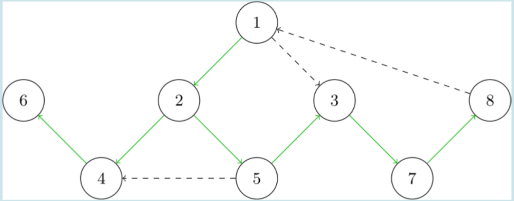
We run *depth-first-search* (DFS) on a directed graph $G = (V, E)$ and we determine all the *pre/post numbers*. Then for an edge $(u, v) \in E$, it's possible that $\mathrm{pre}(u) < \mathrm{pre}(v) < \mathrm{post}(u) < \mathrm{post}(v)$.

- True
- False ✓

We saw in lecture this is impossible. It's impossible because if we recurse on u first, then v before returning to u, then we must go back to v first before u.

The correct answer is 'False'.

Q10



In the graph above, the green, solid edges indicate a *depth-first search tree*. Classify the remaining edges. (*Hint: each option should be used **exactly once***).

$(5, 4)$	Cross Edge
$(8, 1)$	Back Edge
$(1, 3)$	Forward Edge

Assignment

Feedback Assignments 6 & 7

Common points from the last two assignments:

- **DP Solution Structure:** Once you have the correct subproblem and recursion, you can solve the theory DP exercises very mechanically and minimally. The important aspect is that you **formalize** your subproblem (meaning of the DP-entry), the recurrence and the computation order properly and provide a justification for the correctness. Have a **look at** the **Master Solution** for the expected structure.

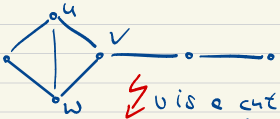
- **Precision:** It is important that you are **precise**. This includes **specifying variables** whenever you use them (once you use a variable, e.g., i, for the first time (e.g., $DP[i] = \ldots$, a **specification needs to follow** (e.g.: "for $i \in \mathbb{N}$ with $1 \leq i \leq n$ or at least $1 \leq i \leq n$ - sufficient when talking about indeces).

- **Justification:** Your **justification can be quite informal** (no formal proof like an induction required) **but** should **explain why** your recurrence computes the correct result using the meaning of the entries.

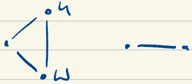
Remember to check the detailed feedback on Moodle! Reach out if you have any questions regarding the corrections.

a) If a vertex v is part of a cycle, then it is not a cut vertex.

counterexample:



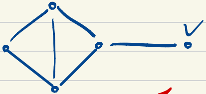
Remove v $\longrightarrow$



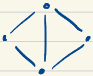
ϟ v is a cut vertex & part of a cycle

b) If a vertex v is not a cut vertex, then v must be part of a cycle.

counterexample:



Remove v $\longrightarrow$



ϟ vertices with degree 1 can be "trimmed" w.o. disconnecting the graph

What about the following cl.?

"If a vertex v is <u>not</u> a cut vertex <u>and</u> $\deg(v) \geq 2$, then v must be part of a cycle."

<u>pr.</u> True. Let w and u be two neighbors of v. There is a path (u,v,w). Since v is not a cut vertex another path connecting u,w must exist, not leading through v. The existence of a cycle follows. □

c) If an edge e is part of a cycle (i.e. e connects two consecutive vertices in a cycle), then it is not a cut edge.

pr. Let $G=(V,E)$ and $e=\{u,v\}\in E$ be part of a cycle. Suppose for sake of contradiction that e was a cut edge. Removing e would then disconnect the graph and u could no longer reach v. But notice that e is part of a cycle. W.l.o.g, let this be $(u, w_2, w_3, \dots, w_k, v, u)$, $w_i \in V$ for some $k \geq 2$ and $2 \leq i \leq k$. We see that there exists another path $(u, w_2, w_3, \dots, w_k, v)$ that does not incl. e. Hence, u and v are still connected, a contradiction. The result then follows. $\blacksquare$

d) If an edge e is not a cut edge, then e must be part of a cycle.

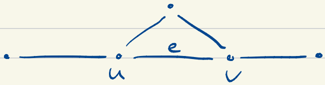
pr. Follow the definitions above and notice that since e is not a cut edge, u still reaches v and a path $(u, w_2, w_3, \dots, w_k, v)$ must exist for some $k \geq 2$.

We now add back e to obtain the cycle $(u, w_2, w_3, \dots, w_k, v, u)$, which proves the result. $\blacksquare$

Q3

e) If u and v are two adjacent cut vertices, then the edge $e = \{u, v\}$ is a cut edge.

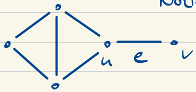
Counterexample:



Notice that u & v are cut vertices but e is part of a cycle $\Rightarrow$ e is not a cut-edge.

f) If $e = \{u, v\}$ is a cut edge, then u and v are cut vertices. What if we add the condition that u and v have degree at least 2 ?

Counterexample:



Notice that e is a cut-edge but v is not a cut-vertex.

If $\deg(u) \geq 2$ and $\deg(v) \geq 2$, the claim holds (same proof pattern as before, see Master Solution).

Q4

Exercise 8.4 *Introduction to Trees.*

In this exercise the goal is to prove a few basic properties of trees (for the definition of a tree, see Definition 1).

(a) A **leaf** is a vertex with degree 1. Prove that in every tree G with at least two vertices there exists a leaf.

(a) Let $G=(V,E)$ be a tree and $p=(v_0,v_1,\dots,v_k)$, $k\in\mathbb{N}$, be a longest path in G.

Consider v_k and suppose that $\deg(v_k)>1$. There must then exist another neighbor $u\neq v_{k-1}$. But notice that since G is a tree, $u\notin\{v_0,\dots,v_{k-2}\}$, otherwise we could construct a cycle $(u,v_i,v_{i+1},\dots,v_{k-2},v_{k-1},v_k,u)$ for some $i\in\{0,\dots,k-2\}$, a contradiction.

Also, $u\notin V\setminus\{v_0,\dots,v_{k-1}\}$, otherwise there would exist a strictly longer path $p'=(v_0,\dots,v_k,u)$, a contradiction. It follows that no such neighbor u can exist and $\deg(v_k)=1$. Hence, a leaf exists.

Q4

(b) Prove that every tree with n vertices has exactly $n-1$ edges.

Hint: *Prove the statement by using induction on n. In the induction step, use part (a) to find a leaf. Disconnect the leaf from the tree and argue that the remaining subgraph is also a tree. Apply the induction hypothesis and conclude.*

We proceed by induction on n.

B.C.: Let $n=1$ and notice that no edge can exist. Hence, there are $0=n-1$ edges. Additionally, for $n=2$, there exists a unique edge. The B.C. hold.

I.H.: Assume for some $n \geq 2$ that every tree on n vertices has exactly $n-1$ edges.

I.S.: Then let $G=(V,E)$ be a tree on $n+1$ vertices. By (a), G has a leaf u. Remove u to obtain $G' = G[V \setminus \{u\}]$.
Notice that:

- G' has n vertices
- G' is connected since for any $a,b \in V \setminus \{u\}$, there still exists a path from a to b since u can only be the endpoint of a path.
- G' is acyclic since if not then G would also contain a cycle but G is a tree.

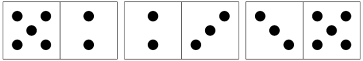
Hence G' is a tree and by I.H. must have $n-1$ edges. It follows that G has $(n-1)+1=n$ edges.

By the princ. of math. induction the claim holds for all $n \in \mathbb{N}$. ☒

Q5

(a) A domino set consists of all possible $\binom{6}{2} + 6 = 21$ different tiles of the form $[x|y]$, where x and y are numbers from $\{1, 2, 3, 4, 5, 6\}$. The tiles are symmetric, so $[x|y]$ and $[y|x]$ is the same tile and appears only once.

Show that it is impossible to form a line of all 21 tiles such that the adjacent numbers of any consecutive tiles coincide like in the example below.



(b) What happens if we replace 6 by an arbitrary $n \geq 2$? For which n is it possible to line up all $\binom{n}{2} + n$ different tiles along a line?

Proof Idea: Consider the graph $G=(V,E)$, where $V=\{1,\dots,n\}$ and $E=\{\{i,j\} \mid i,j\in V, i\neq j\}$.
Notice that each vertex represents one number and each edge represents one domino.

Notice then that we can reduce our problem of finding a domino-line to the problem of finding an Eulerian walk in G, since we need to use every edge/domino exactly once and are forced to join dominoes by number.

Finally, notice that $G = K_n \Rightarrow \forall v \in V(\deg(v) = n-1)$. We know that $n-1$ must be even for an Eulerian walk to exist. It follows that $n=2$ or n must be odd for a domino line to exist.

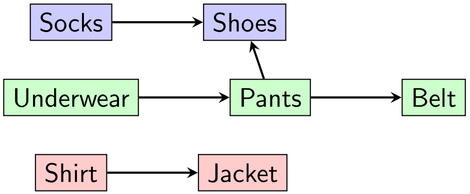
Directed Graphs

Motivation: Task Scheduling with Dependencies

Real-world problem: Getting dressed in the morning!

Dependencies:

- Socks $\rightarrow$ Shoes
- Underwear $\rightarrow$ Pants $\rightarrow$ Belt
- Shirt $\rightarrow$ Jacket

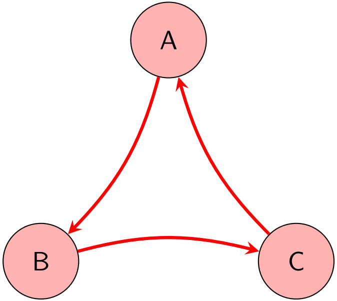


Valid order:
Underwear, Shirt, Socks, Pants, Belt, Shoes, Jacket

Question: Can we always find a valid ordering? What if there are circular dependencies?

When Ordering is Impossible

Circular dependencies create problems!



Directed cycle: $A \rightarrow B \rightarrow C \rightarrow A$

Problem:

- Each node has a predecessor
- No valid starting point!
- Cannot produce a linear ordering

Key Theorem (Preview)

A directed graph has a **topological ordering** if and only if it contains **no directed cycles**.

Definition: Directed Graphs

Definition 2.1 (Directed Graph)

A **directed graph** (or *digraph*) $G = (V, E)$ consists of:

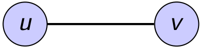
- V: a finite set of **vertices**
- $E \subseteq V \times V$: a set of **ordered pairs** called **directed edges**

We write $(u, v) \in E$ for a directed edge from u to v.

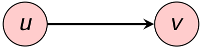
Important difference from undirected graphs:

- Undirected: $\{u, v\} = \{v, u\}$ (order doesn't matter)
- Directed: $(u, v) \neq (v, u)$ (order matters!)

Undirected



Directed



Terminology for Directed Graphs

Definition 2.2 (Successors and Predecessors)

For a directed edge $(u, v) \in E$:

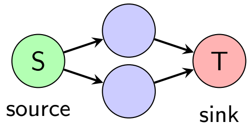
- v is a **successor** (Nachfolger) of u
- u is a **predecessor** (Vorgänger) of v

Definition 2.3 (In-Degree and Out-Degree)

For a vertex $u \in V$:

- $\deg_{\text{in}}(u) := |\{v \mid (v, u) \in E\}|$ is the **in-degree** (Eingangsgrad)
- $\deg_{\text{out}}(u) := |\{v \mid (u, v) \in E\}|$ is the **out-degree** (Ausgangsgrad)

- A **source** is a vertex with $\deg_{\text{in}}(u) = 0$
- A **sink** is a vertex with $\deg_{\text{out}}(u) = 0$



Paths and Cycles in Directed Graphs

Definition 2.4 (Directed Path)

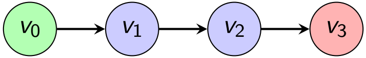
A **directed path** from v_0 to v_k is a sequence of vertices $(v_0, v_1, \ldots, v_k)$ where $(v_i, v_{i+1}) \in E$ for all $0 \leq i < k$.

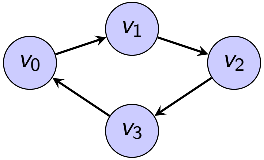
Definition 2.5 (Directed Cycle)

A **directed cycle** is a directed path where:

- $k \geq 1$ (no self-loops)
- $v_0 = v_k$ (starts and ends at same vertex)
- All other vertices are distinct

Directed Path

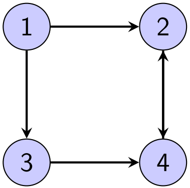




Directed Cycle

Representing Directed Graphs

Both adjacency matrix and adjacency list work for directed graphs!



Example Graph

Adjacency Matrix A

$$A[u][v] = \begin{cases} 1 & \text{if } (u, v) \in E \\ 0 & \text{otherwise} \end{cases}$$

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

Adjacency List

1: [2, 3]
2: [4]
3: [4]
4: [2]

Note: Matrix is generally **not symmetric** for directed graphs!

Topological Sorting

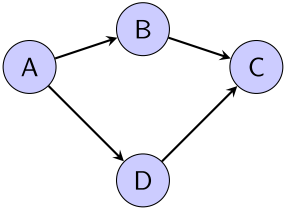
Topological Sorting: The Problem

Definition 2.6 (Topological Sorting)

A **topological ordering** of a directed graph $G = (V, E)$ is a linear (total) ordering of vertices $v_1, v_2, \ldots, v_n$ such that:

$$(v_i, v_j) \in E \implies i < j$$

In other words: all edges point "forward" in the ordering.



Valid: A, D, B, C
Also valid: A, B, D, C

Applications:

- Task scheduling
- Build systems (Makefiles)
- Course prerequisites
- Compiler dependency resolution

When Does a Topological Ordering Exist?

Theorem 2.7

A directed graph G has a topological ordering **if and only if** G is acyclic (has no directed cycles).

Intuition for "$\Leftarrow$" (acyclic $\implies$ topological ordering exists):

- If there are no cycles, we can find a **sink** (vertex with no outgoing edges)
- Place the sink at the *end* of the ordering
- Remove it and repeat recursively
- This always works because each step maintains acyclicity

Question: Why must an acyclic graph have a sink?

Algorithm 1: Recursive Sink-Finding

function $\textsc{TopSort}(G = (V, E))$
  if $V = \emptyset$ **then**
    return empty list
  end if
  Find a sink v in G $\triangleright\ \deg_{\text{out}}(v) = 0$
  Let $G' = (V \setminus \{v\}, E')$ where $E' = E \setminus \{(u, v) \mid u \in V\}$
  $L \leftarrow \textsc{TopSort}(G')$
  return L with v appended at the end
end function

Correctness:

- Each recursive call places a sink at the end
- All edges from other vertices point to sinks we've already placed
- By induction, produces a valid topological ordering

Runtime: $O(|V|^2)$ if finding a sink takes $O(|V|)$ time each call.

Can we do better?

Your Turn: No Backward Paths in Topological Orderings

Claim: Let $G = (V, E)$ be a DAG with topological ordering $v_1, v_2, \ldots, v_n$. If $i < j$, then there is **no** directed path from v_j to v_i.

In other words: you cannot go "backward" in a topological ordering by following edges.

Prove this claim:

Strategy: Use proof by contradiction and assume $i < j$ but there exists a directed path $v_j \to v_{k_1} \to v_{k_2} \to \cdots \to v_{k_m} \to v_i$

Time: 3-4 minutes

Solution: Edges in Topological Orderings

Proof.

We prove this by contradiction.

Suppose there exists an edge $(v_i, v_j) \in E$ and a directed path P from v_j to v_i.

From the topological ordering:

- Since $(v_i, v_j) \in E$, we have $i < j$ (by definition of topological ordering)

From the path:

- Let $P : v_j \to v_{k_1} \to v_{k_2} \to \cdots \to v_{k_m} \to v_i$
- By the topological ordering property, each edge goes "forward":
  - $(v_j, v_{k_1}) \in E \implies j < k_1$
  - $(v_{k_1}, v_{k_2}) \in E \implies k_1 < k_2$
  - $\vdots$
  - $(v_{k_m}, v_i) \in E \implies k_m < i$
- By transitivity: $j < k_1 < k_2 < \cdots < k_m < i$, so $j < i$

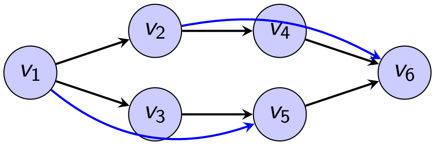
But we have both $i < j$ and $j < i$, a contradiction. No such path can exist. $\square$

Understanding DAG Structure

What does this tell us?

Key Insight

In a topologically sorted DAG, all edges point "forward" in the ordering, and there are **no paths that go backward**.



Blue edges "skip ahead" but still go forward

Consequences:

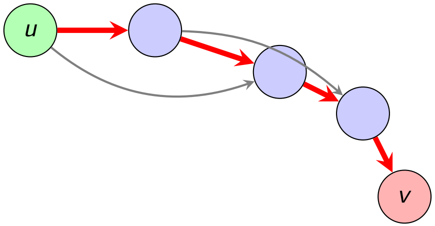
- Can process vertices left-to-right
- When processing v_i, **all vertices that v_i depends on have been processed**
- **Enables efficient algorithms** (e.g., shortest paths in $O(|V| + |E|)$)

Application: This structure is why we can compute shortest and even longest paths in DAGs in **linear time**—we just process vertices in topological order!

A Better Approach: Using Depth-First Search

Key insight: Instead of repeatedly finding sinks, we can use DFS!

Observation: When we start at a vertex u and follow a path as far as possible (without repeating vertices), where do we end up?



Path from u to v

Claim: If we follow an unmarked path as far as possible, we must end at a vertex v where:

- All successors of v have been visited, OR
- v has no successors (is a sink)

If the graph is acyclic, v must be a sink!

This gives us an efficient way to find vertices to place at the end of our ordering!

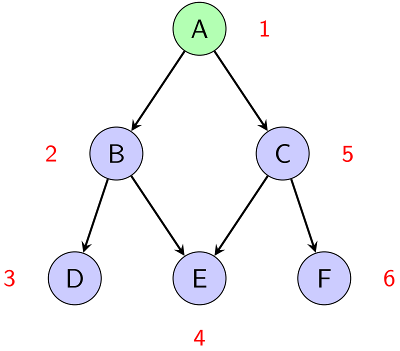
Depth-First Search (DFS)

Depth-First Search: The Idea

Strategy: Explore as deeply as possible before backtracking

Algorithm sketch:

1. Start at a vertex u
2. Mark u as visited
3. For each unmarked successor v:
   - Recursively visit v
4. When no unmarked successors remain, backtrack



Visit order: A, B, D, E, C, F

Key property: DFS explores one branch completely before moving to the next!

DFS: Formal Algorithm

```
function VISIT(u)
    mark u
    for each successor v of u do
        if v is unmarked then
            VISIT(v)
        end if
    end for
end function
```

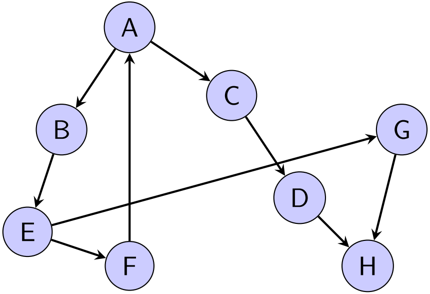
```
function DFS(G = (V, E))
    Mark all vertices as unmarked
    for each u ∈ V do
        if u is unmarked then
            VISIT(u)
        end if
    end for
end function
```

Why the outer loop in DFS?

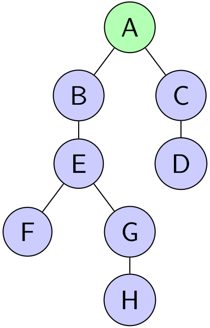
- The graph might be disconnected
- Starting from one vertex might not reach all vertices
- We need to ensure we visit every vertex

Result: Creates a **DFS forest** (collection of DFS trees)

DFS: Visual Example



Graph



DFS Tree (starting at A)

Tree edges shown above

Observation: The recursion naturally creates a tree structure!

DFS Runtime Analysis

Theorem 2.8

DFS runs in $O(|V| + |E|)$ time when using an adjacency list representation.

Proof Idea.

Work done per vertex:

- Each vertex u is marked exactly once
- When visiting u, we iterate through all its successors
- Time for visiting u: $O(1 + \deg_{\text{out}}(u))$

Total time:

$$\sum_{u \in V} O(1 + \deg_{\text{out}}(u)) = O\left(\sum_{u \in V} 1 + \sum_{u \in V} \deg_{\text{out}}(u)\right)$$
$$= O(|V| + |E|)$$

$\square$

DFS with Timestamps

Enhancement: Record when we enter and exit each vertex

function $\textsc{Visit}(u)$
  $\text{pre}[u] \leftarrow T$; $T \leftarrow T + 1$
  mark u
  for each successor v of u **do**
    if v is unmarked **then**
      $\textsc{Visit}(v)$
    end if
  end for
  $\text{post}[u] \leftarrow T$; $T \leftarrow T + 1$
end function

function $\textsc{DFS}(G)$
  $T \leftarrow 1$
  (mark all unmarked, loop over vertices)
end function

Timestamps:

- $\text{pre}[u]$: when we first visit u
- $\text{post}[u]$: when we finish exploring from u

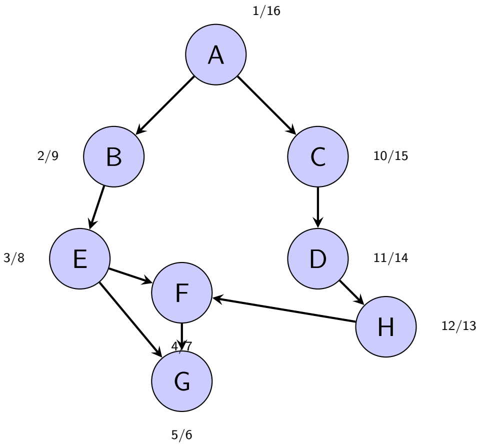
Interval notation:

$$I_u = [\text{pre}[u], \text{post}[u]]$$

These intervals reveal important structural properties!

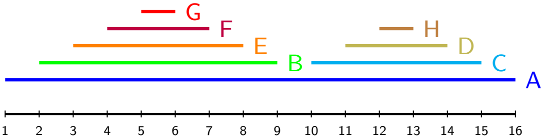
DFS Timestamp Example

Graph



Pre/Post orders:
Pre-order: A, B, E, F, G, C, D, H
Post-order: G, F, E, B, H, D, C, A

Interval representation:



Nesting property: Intervals either nest (one contains the other) or are disjoint!

DFS and Topological Sorting

Theorem 2.9

If G is acyclic (a DAG), then the **reverse post-order** from DFS gives a topological ordering.

Why does this work?

- When we finish exploring from vertex u, all vertices reachable from u have already been finished
- So post[u] comes after all descendants of u
- If $(u, v) \in E$, then v is explored before we finish u, so post[v] $<$ post[u]
- Reversing post-order puts u before v, as required!

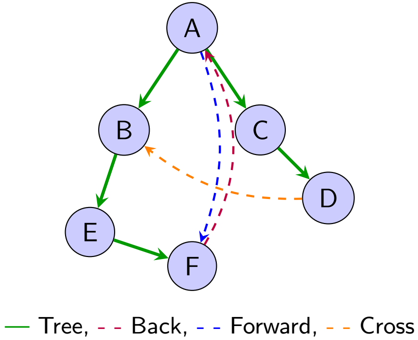
```
function TopologicalSort(G)
    Run DFS on G, computing post-order numbers
    return vertices in reverse post-order
end function
```

Runtime: $O(|V| + |E|)$

Edge Classification in DFS

For edge (u, v):

- **Tree edge:** v is unmarked when explored from u
  - Forms the DFS tree/forest
- **Back edge:** $I_u \subseteq I_v$
  - Points to an ancestor
- **Forward edge:** $I_v \subseteq I_u$
  - Points to a descendant (not via tree edges)
- **Cross edge:** $I_u \cap I_v = \emptyset$
  - Connects different branches



DFS for Cycle Detection: A directed graph has a cycle $\iff$ DFS finds a back edge!

Cycle Detection Using DFS

Theorem 2.10

A directed graph G contains a cycle if and only if DFS discovers a back edge.

Proof Sketch.

(⇒): Suppose G has a cycle $v_1 \to v_2 \to \cdots \to v_k \to v_1$.

- Let v_i be the first vertex in the cycle visited by DFS
- When exploring from v_i, we must eventually reach v_{i-1} (the predecessor in the cycle)
- But v_i is still "active" (not finished), so $I_{v_{i-1}} \subseteq I_{v_i}$
- Therefore (v_{i-1}, v_i) is a back edge

(⇐): Suppose DFS finds back edge (u, v) where $I_u \subseteq I_v$.

- v is an ancestor of u in the DFS tree
- There's a tree path from v to u
- Adding edge (u, v) creates a cycle

□

Your Turn: DFS Interval Property

Claim: In any DFS of a directed graph G, for any two vertices u and v, exactly one of the following holds:

1. $I_u \cap I_v = \emptyset$ (intervals are disjoint), OR
2. $I_u \subseteq I_v$ (interval of u is contained in interval of v), OR
3. $I_v \subseteq I_u$ (interval of v is contained in interval of u)

where $I_u = [\text{pre}[u], \text{post}[u]]$ and $I_v = [\text{pre}[v], \text{post}[v]]$.

Prove this claim:

Hints:

- Without loss of generality, assume $\text{pre}[u] < \text{pre}[v]$ (i.e., we visit u first)
- There are two cases to consider: Is v visited before we finish exploring from u?

Time: 4-5 minutes

Solution: DFS Interval Property

Claim: For any two vertices u and v, exactly one of the following holds: $I_u \cap I_v = \emptyset$, OR $I_u \subseteq I_v$, OR $I_v \subseteq I_u$

Proof.

Without loss of generality, assume $\text{pre}[u] < \text{pre}[v]$ (we visit u first).

Case 1: $\text{pre}[v] < \text{post}[u]$ (we visit v before finishing u)

- This means v was discovered during the exploration from u
- We must finish exploring from v before we can finish u
- Therefore: $\text{pre}[u] < \text{pre}[v] < \text{post}[v] < \text{post}[u]$
- Thus $I_v \subseteq I_u$

Case 2: $\text{pre}[v] > \text{post}[u]$ (we visit v after finishing u)

- The intervals don't overlap at all
- Thus $I_u \cap I_v = \emptyset$

These are the only two possibilities, and they're mutually exclusive. $\square$

DFS - Your Friend and Helper

You are given an undirected graph as an adjacency list. Implement:

1. `hasCycle(n, E)`: Detect if the graph contains a cycle
2. `isReachable(E, u, v)`: Check if there's a path from u to v
3. `countComponents(E)`: Count the number of connected components

Setup:

- Download the template from my website
- Copy `Main.java` to CodeExpert's "Welcome" exercise
- Copy `custom.in` and `custom.out` to corresponding files in "Welcome" exercise (in the `public/` directory)

Questions?

Peer Grading

Peer-Grading Exercise

This week's peer-grading exercise is **Exercise 8.3d-f**

Please follow the usual process:

- Grade the assigned group's submission.
- Give constructive feedback and upload your feedback to Moodle by 23.59 tonight, at the latest.
- Contact me if you don't receive a submission to grade.