Week 9: Topological Sort & DFS

Algorithms & Data Structures

Thorben Klabunde

th-kl.ch

November 17, 2025

Agenda

Mini-Quiz

Assignment

Q3

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

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

Q3

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.

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

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

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.

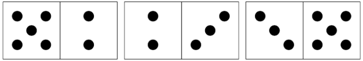
(b) Prove that every tree with n vertices has exactly $n - 1$ edges.

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.*

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?

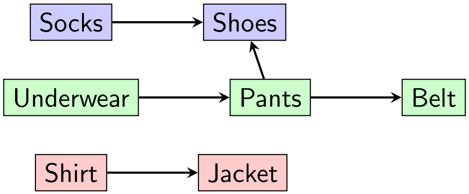
Directed Graphs

Motivation: Task Scheduling with Dependencies

Real-world problem: Getting dressed in the morning!

Dependencies:

- Socks → Shoes
- Underwear → Pants → Belt
- Shirt → Jacket

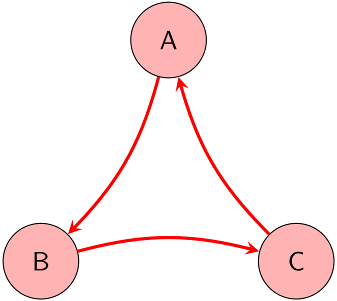


Valid order:
Underwear, Shirt, Socks, Pants, Belt, Shoes, Jacket

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

When Ordering is Impossible

Circular dependencies create problems!



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

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)

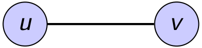
A **directed graph** (or *digraph*) $G = (V, E)$ consists of:

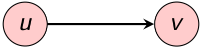
- V: a finite set of **vertices**
- $E \subseteq V \times V$: a set of **ordered pairs** called **directed edges**

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 $(u, v) \in E$:

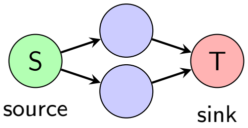
- v is a **successor** (Nachfolger) of u
- u is a **predecessor** (Vorgänger) of v

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)

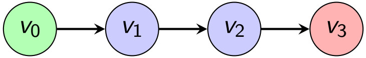
A **directed path** from v_0 to v_k is a sequence of vertices $(v_0, v_1, \ldots, v_k)$ where $(v_i, v_{i+1}) \in E$ for all $0 \leq i < k$.

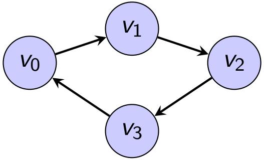
Definition 2.5 (Directed Cycle)

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



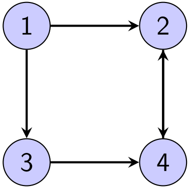


Directed Cycle

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

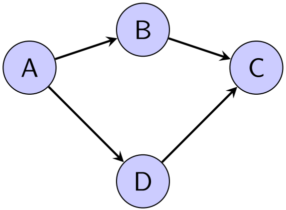
Topological Sorting: The Problem

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 $\quad \triangleright\ \deg_{out}(v) = 0$
  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 $v_j \to v_{k_1} \to v_{k_2} \to \cdots \to v_{k_m} \to v_i$

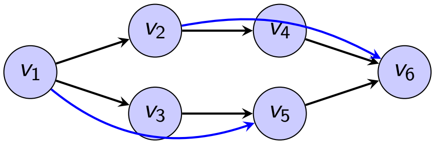
Time: 3-4 minutes

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:

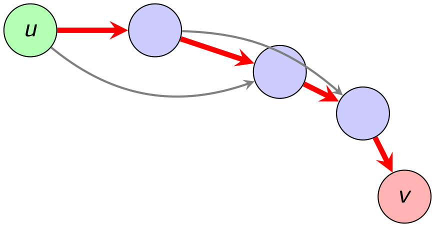
- Can process vertices left-to-right
- When processing v_i, **all vertices that v_i depends on have been processed**
- **Enables efficient algorithms** (e.g., shortest paths in $O(|V| + |E|)$)

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!

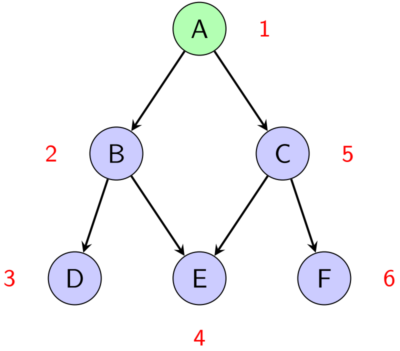
Depth-First Search (DFS)

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
```

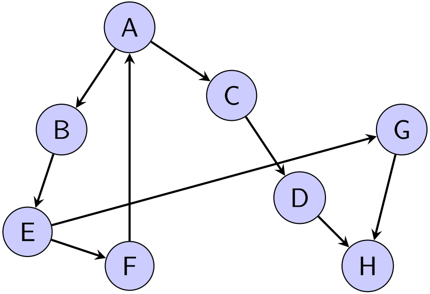
```
function DFS(G = (V, E))
    Mark all vertices as unmarked
    for each u ∈ V do
        if u is unmarked then
            VISIT(u)
        end if
    end for
end function
```

Why the outer loop in DFS?

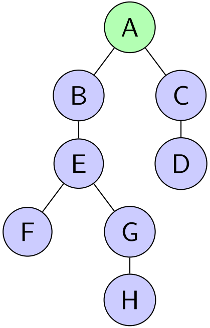
- The graph might be disconnected
- Starting from one vertex might not reach all vertices
- We need to ensure we visit every vertex

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 $\text{DFS}(G)$
    $T \leftarrow 1$
    (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

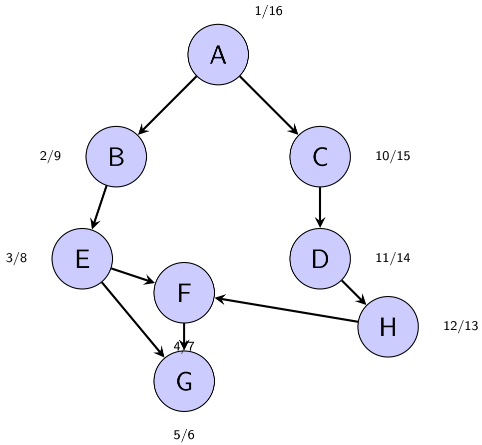
Interval notation:

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

These intervals reveal important structural properties!

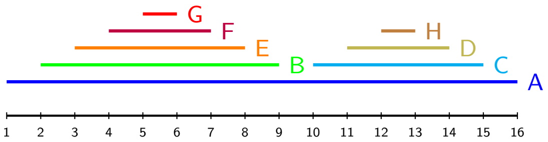
DFS Timestamp Example

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 $\text{post}[v] < \text{post}[u]$
- Reversing post-order puts u before v, as required!

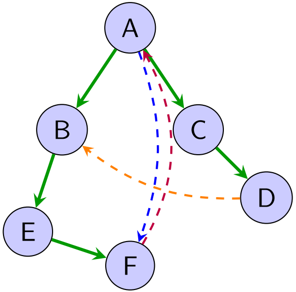
```
function TopologicalSort(G)
    Run DFS on G, computing post-order numbers
    return vertices in reverse post-order
end function
```

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



— Tree, - - Back, - - Forward, - - Cross

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

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.