Week 7 — Elementary Graph Algorithms

Félix Chavelli felix.chavelli@inria.fr

April 8, 2026 · Semester 2

PSL UNIVERSITÉ PARIS

Today's Agenda

Motivation & Applications

Graph Representations

Breadth-First Search

Depth-First Search

Topological Sort

Strongly Connected Components

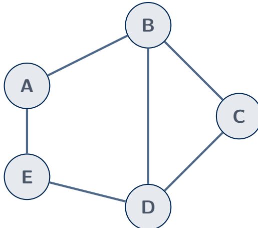
Summary & What's Next

Part 1

Motivation & Applications

Why Graphs?

- Graphs model relationships between objects
- One of the most versatile data structures in CS
- Appear *everywhere* in AI and beyond:
  - Knowledge representation
  - State-space search
  - Neural network architectures
  - Optimization problems



Graphs in the Real World

Navigation & Maps

- Google Maps, Waze, GPS routing
- Cities = vertices, roads = edges
- Shortest path → Dijkstra, A*

Social Networks

- Facebook, LinkedIn, Twitter/X
- Users = vertices, friendships/follows = edges
- Community detection → connected components, SCC

The Web

- Pages = vertices, hyperlinks = directed edges
- PageRank (Google) walks this graph
- Web crawlers = BFS/DFS on the link graph

Video Games

- Game maps, pathfinding (A*, navmeshes)
- State-space graphs for AI decisions
- Scene graphs, dependency graphs

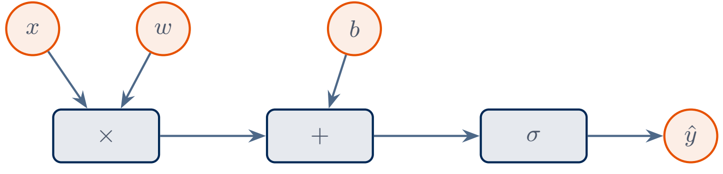
Puzzle Solving

- Rubik's cube, 15-puzzle, Sudoku
- States = vertices, moves = edges
- BFS → shortest solution

Autograd (Deep Learning)

- Computation graph: nodes = operations, edges = data flow
- Forward pass = topological-order evaluation
- Backward pass = reverse topological order
- We will code this later in the course!

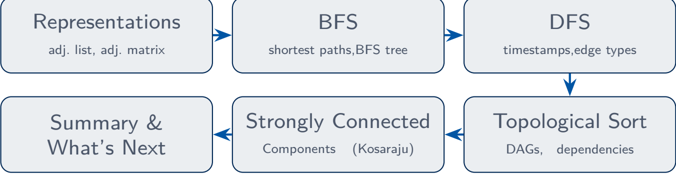
The Autograd Computation Graph



Forward pass: evaluate in topological order (left $\to$ right).
Backward pass (backprop): compute gradients in reverse topological order (right $\to$ left).

PyTorch's `autograd` builds exactly this DAG at runtime. Understanding graphs = understanding how your neural network trains!

Today's Roadmap



Part 2

Graph Representations

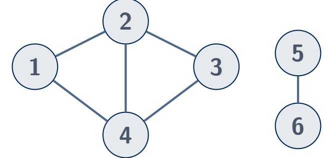
Definitions

Graph

A **graph** $G = (V, E)$ consists of a finite set V of **vertices** (nodes) and a set E of **edges** (pairs of vertices).

Undirected graph:

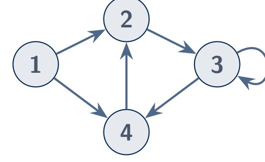
- $E \subseteq \binom{V}{2}$ (unordered pairs)
- Edge $\{u, v\}$: u and v are *adjacent*
- **Degree** of v: $\#$ edges incident to v



Components: $\{1, 2, 3, 4\}$ and $\{5, 6\}$

Directed graph (digraph):

- $E \subseteq V \times V$ (ordered pairs)
- Edge (u, v): from u to v
- **Out-degree**, **in-degree**

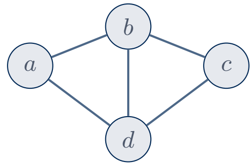


- Graphs may have **multiple connected components** (disconnected graph)
- Graphs may have **self-loops** (v, v)

Degree Sum & Handshaking Lemma

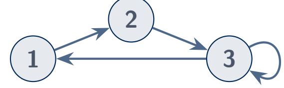
Handshaking Lemma

Undirected: $\sum_{v \in V} \deg(v) = 2\,|E|$ **Directed:** $\sum_{v \in V} \text{in-deg}(v) = \sum_{v \in V} \text{out-deg}(v) = |E|$



$\deg(a) = 2,\ \deg(b) = 3,\ \deg(c) = 2,\ \deg(d) = 3$

$\sum \deg = 10 = 2 \times 5 = 2|E|$ ✓



$|E| = 4$ (3 arcs + 1 self-loop)

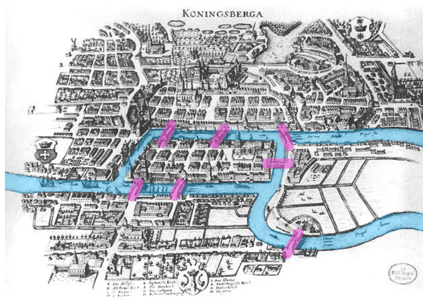
$\sum$ out-deg $= 1 + 1 + 2 = 4$ ✓

$\sum$ in-deg $= 1 + 1 + 2 = 4$ ✓

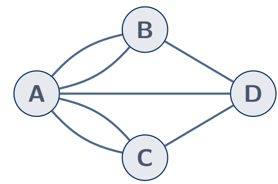
Warning

A self-loop (v, v) contributes $+1$ to *both* in-deg and out-deg (one arc).
Degree of vertex 3 is 4 although it is involved in only 3 edges (one is a self-loop).

The Bridges of Königsberg (1736)



Königsberg (now Kaliningrad), 7 bridges.



deg(A)=5, deg(B)=3, deg(C)=3, deg(D)=3

The problem: Can one cross each bridge *exactly once* and return to the start?

Euler's insight (1736):

- Model: landmasses = vertices, bridges = edges
- Such a closed walk exists iff every vertex has **even degree**
- All 4 vertices have odd degree ⇒ impossible!

Key Idea

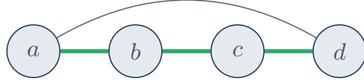
This is considered the **birth of graph theory** (1736). Euler abstracted the map into a mathematical structure and proved a general theorem about **Eulerian circuits**.

Paths

Path

A **path** from u to v: sequence $\langle v_0, v_1, \ldots, v_k \rangle$ with $v_0 = u$, $v_k = v$, $(v_{i-1}, v_i) \in E$.
Length: number of edges k. **Simple path**: no repeated vertices.

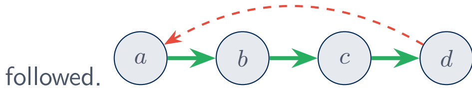
Undirected path:



Simple path $\langle a, b, c, d \rangle$, length 3.

Non-simple: $\langle a, d \rangle$ (direct edge) also valid.

Directed path: direction of edges must be followed.



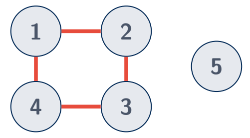
Valid directed path $\langle a \to b \to c \to d \rangle$.

$d \to a$: only if that arc exists.

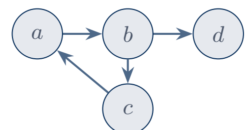
Cycles & Connectivity

Cycles

A **cycle** is a path with $v_0 = v_k$ and $k \geqslant 1$. **Simple cycle**: no repeated vertices (except $v_0 = v_k$).



Cycle $\langle 1, 2, 3, 4, 1 \rangle$. Node 5: isolated.



Connectivity

Undirected:

- **Connected**: $\exists$ path between every pair
- **Connected Component**: maximal connected subgraph

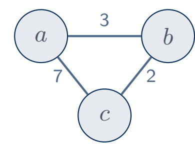
Directed:

- **Strongly connected**: $\exists$ path $u \rightsquigarrow v$ *and* $v \rightsquigarrow u$ for all pairs
- **Weakly connected**: connected ignoring directions
- **Strongly Connected Component**: maximal strongly connected subgraph

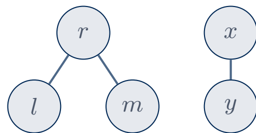
Strongly Connected Component $= \{a, b, c\}$. The whole graph is weakly connected.

Special Graphs

Weighted graph:
Edges carry weights $w \colon E \to \mathbb{R}$

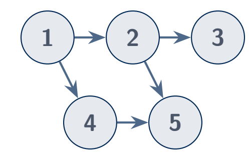


Tree & Forest:
Tree: connected acyclic undirected graph
Forest: union of trees



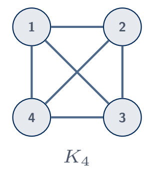
A forest with two trees

DAG (directed acyclic graph):

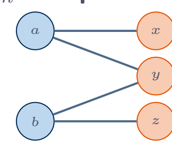


No cycle $\Rightarrow$ topological ordering exists

Complete K_n **& Bipartite:**



K_4



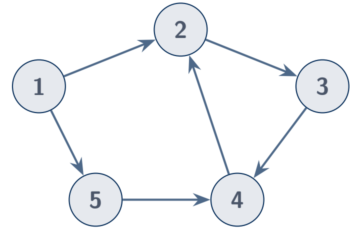
Bipartite

Key Idea

Tree on n vertices $\Rightarrow$ $n-1$ edges. Connected $\Rightarrow$ $|E| \geqslant |V|-1$. K_n: $\binom{n}{2} = \Theta(n^2)$ edges.

Adjacency List Representation

Directed graph:



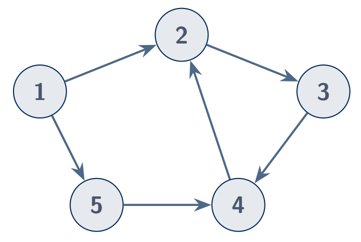
Adjacency list Adj$[\cdot]$:

Vertex	Neighbors
1	$[2, 5]$
2	$[3]$
3	$[4]$
4	$[2]$
5	$[4]$

- **Space:** $\Theta(V + E)$
- **Check edge** (u, v)**:** $\mathcal{O}(\deg(u))$ — must scan the list
- **Iterate neighbors of** u**:** $\mathcal{O}(\deg(u))$ — perfect!
- Best for sparse graphs ($|E| \ll |V|^2$)

Adjacency Matrix Representation

Same directed graph:



Adjacency matrix A:

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

$a_{ij} = 1$ iff $(i, j) \in E$

- **Space:** $\Theta(V^2)$
- **Check edge** (u, v)**:** $\mathcal{O}(1)$ — just look up $A[u][v]$!
- **Iterate neighbors:** $\mathcal{O}(V)$ — must scan entire row
- Best for dense graphs ($|E| \approx |V|^2$)
- For undirected graphs: $A = A^T$ (symmetric)

Comparison: List vs. Matrix

Operation	Adjacency List	Adjacency Matrix
Space	$\Theta(V+E)$	$\Theta(V^2)$
Check edge (u,v)	$\mathcal{O}(\deg(u))$	$\mathcal{O}(1)$
List neighbors of u	$\mathcal{O}(\deg(u))$	$\mathcal{O}(V)$
Add edge	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Add vertex	$\mathcal{O}(1)$	$\mathcal{O}(V^2)$ (rebuild)
Iterate all edges	$\Theta(V+E)$	$\Theta(V^2)$

Key Idea

Most algorithms (BFS, DFS, Dijkstra, etc.) use **adjacency lists**. Use an adjacency matrix when:

- The graph is dense
- You need $\mathcal{O}(1)$ edge queries (e.g., Floyd-Warshall)

Python & C++ Representations

Python — adjacency list:

Python

```python
# dict of lists
graph = {
    1: [2, 5],
    2: [3],
    3: [4],
    4: [2],
    5: [4],
}
# or: list of lists (0-indexed)
adj = [[], [2,5], [3], [4], [2], [4]]
```

C++ — adjacency list:

Cpp

```cpp
#include <vector>
using namespace std;
// 1-indexed, n vertices
int n = 5;
vector<vector<int>> adj(n+1);
adj[1] = {2, 5};
adj[2] = {3};
adj[3] = {4};
adj[4] = {2};
adj[5] = {4};
```

Part 3

Breadth-First Search

BFS — Intuition

- Start from a **source vertex** s
- Explore vertices in **waves**: first all neighbors of s, then neighbors of neighbors, etc.
- Uses a **FIFO queue**
- Colors: **white** $\rightarrow$ **gray** (discovered, in queue) $\rightarrow$ **black** (finished)
- Computes **shortest distances** $v.d = \delta(s, v)$ (in # edges)
- Builds a **BFS tree** via predecessor pointers $v.\pi$

Waves emanate from s.
Wave k = all vertices at distance k from s.

BFS — Pseudocode

```
Input: Graph G = (V, E), source s
 1: for each u ∈ V \ {s} do
 2:     u.color ← WHITE;  u.d ← ∞;  u.π ← NIL
 3: end for
 4: s.color ← GRAY;  s.d ← 0;  s.π ← NIL
 5: Q ← ∅;  ENQUEUE(Q, s)
 6: while Q ≠ ∅ do
 7:     u ← DEQUEUE(Q)
 8:     for each v ∈ Adj[u] do
 9:         if v.color = WHITE then
10:             v.color ← GRAY
11:             v.d ← u.d + 1
12:             v.π ← u
13:             ENQUEUE(Q, v)
14:         end if
15:     end for
16:     u.color ← BLACK
17: end while
```

Complexity

$\mathcal{O}(V + E)$

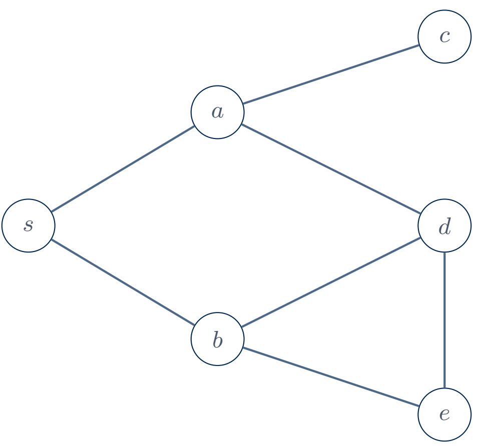
Each vertex enqueued/dequeued at most once. Each adjacency list scanned once.

Correctness (CLRS 20.5)

Upon termination:
$v.d = \delta(s, v)$ for all $v \in V$.
The BFS tree gives shortest paths.

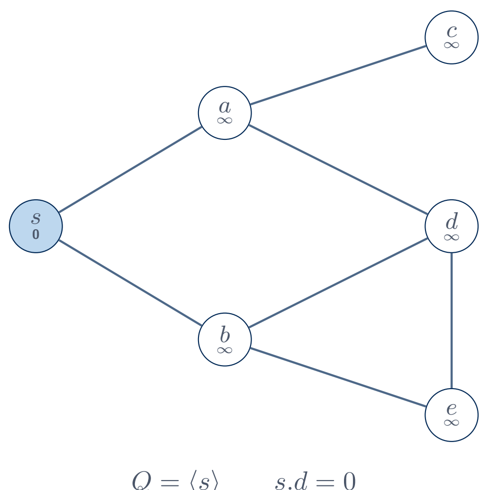
BFS — Step-by-Step: The Graph

Undirected graph — source: s



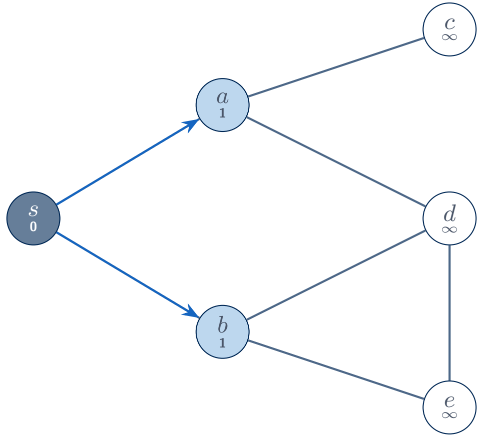
All vertices white, $d = \infty$, $Q = \emptyset$

BFS — Step 0: Initialize source



$Q = \langle s \rangle \qquad s.d = 0$

BFS — Step 1: Dequeue s, discover a and b

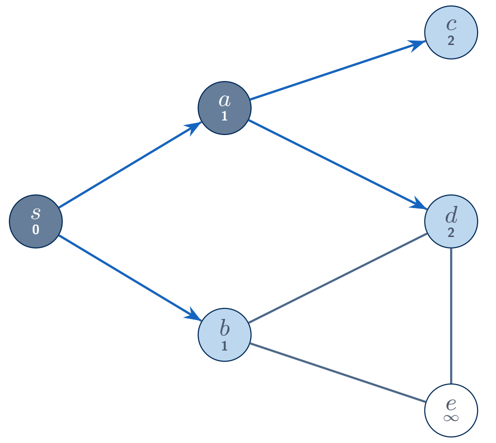


Dequeue $s \to$ scan $\text{Adj}[s] = \{a, b\}$

$a.d = 1,\ a.\pi = s \qquad b.d = 1,\ b.\pi = s$

$Q = \langle a, b \rangle$

BFS — Step 2: Dequeue a, discover c and d

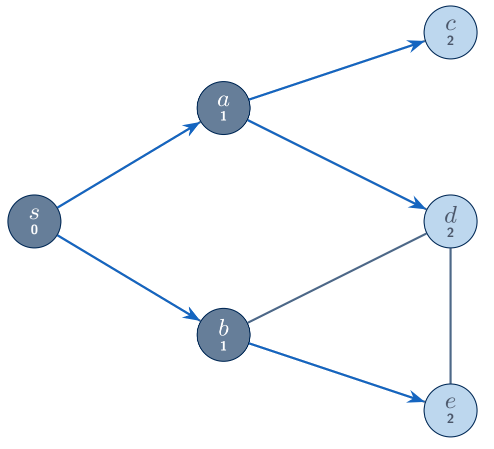


Dequeue $a \to \text{Adj}[a] = \{s, c, d\}$; s already visited

$c.d = 2,\ c.\pi = a \quad d.d = 2,\ d.\pi = a$

$Q = \langle b, c, d \rangle$

BFS — Step 3: Dequeue b, discover e

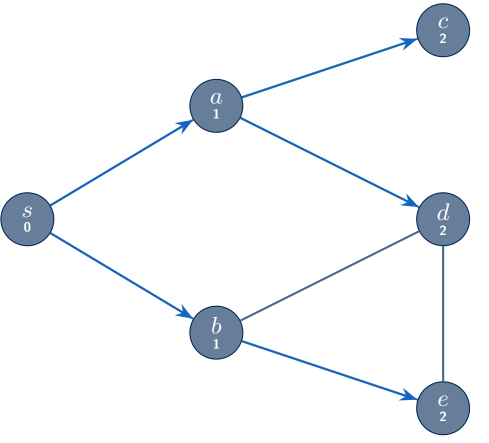


Dequeue $b \to \text{Adj}[b] = \{s, d, e\}$; s, d already visited

$e.d = 2,\ e.\pi = b$

$Q = \langle c, d, e \rangle$

BFS — Steps 4–6: Dequeue c, d, e (no new discoveries)



All remaining vertices dequeued $\rightarrow$ all neighbors already visited
$Q = \langle\rangle$ (empty) $\rightarrow$ **BFS complete!**

BFS Tree (blue edges)

Tree edges: $(s, a), (s, b), (a, c), (a, d), (b, e)$. Path $s \rightarrow d$: $s \rightarrow a \rightarrow d$ with distance 2.

BFS in Python

Python

```
from collections import deque

def bfs(graph, s):
    dist = {v: float('inf') for v in graph}
    parent = {v: None for v in graph}
    dist[s] = 0
    queue = deque([s])
    while queue:
        u = queue.popleft()
        for v in graph[u]:
            if dist[v] == float('inf'):    # white
                dist[v] = dist[u] + 1
                parent[v] = u
                queue.append(v)
    return dist, parent
```

Part 4

Depth-First Search

DFS — Intuition

- Explore as **deep as possible** before backtracking
- Uses a **stack** (implicit via recursion)
- Colors: **white** → **gray** (discovered) → **black** (finished)
- Two **timestamps** per vertex:
  - $u.d$ = discovery time (turn gray)
  - $u.f$ = finish time (turn black)
- May start from **multiple sources** → produces a **DFS forest**
- Classifies edges into 4 types

Always go deeper.
Hit a dead end? Backtrack to the last unexplored branch.

Key property:
$u.d < u.f$ always.
Timestamps: 1 to $2|V|$.

DFS — Pseudocode

DFS(G):

1: **for** each $u \in G.V$ **do**
2: $\quad u$.color $\leftarrow$ WHITE; $u.\pi \leftarrow \text{NIL}$
3: **end for**
4: time $\leftarrow 0$
5: **for** each $u \in G.V$ **do**
6: $\quad$ **if** u.color $=$ WHITE **then**
7: $\quad\quad \text{DFS-Visit}(G, u)$
8: $\quad$ **end if**
9: **end for**

DFS-Visit(G, u):

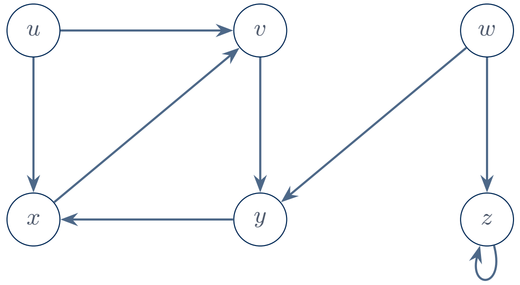
1: time $\leftarrow$ time $+ 1$
2: $u.d \leftarrow$ time; u.color $\leftarrow$ GRAY
3: **for** each $v \in$ Adj$[u]$ **do**
4: $\quad$ **if** v.color $=$ WHITE **then**
5: $\quad\quad v.\pi \leftarrow u$
6: $\quad\quad \text{DFS-Visit}(G, v)$
7: $\quad$ **end if**
8: **end for**
9: time $\leftarrow$ time $+ 1$
10: $u.f \leftarrow$ time; u.color $\leftarrow$ BLACK

Complexity

$\Theta(V + E)$

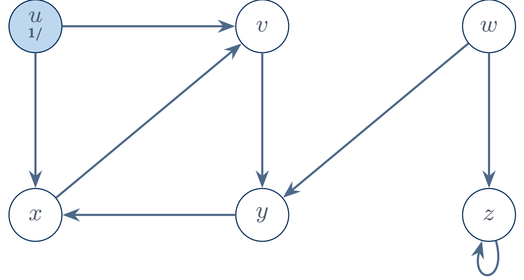
DFS — Step-by-Step: The Graph

Directed graph (vertex order: u, v, w, x, y, z)



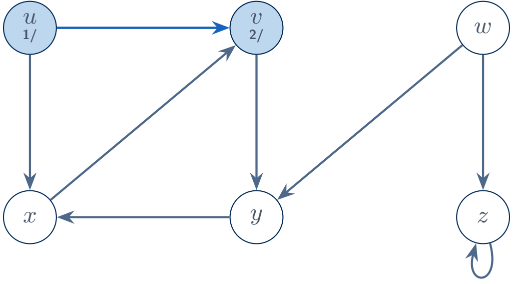
All white, time $= 0$

DFS — Step 1: Visit u (discover u)



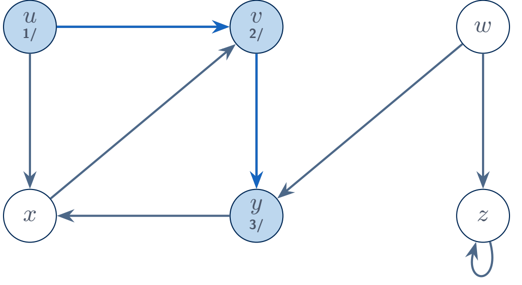
time $= 1$, $u.d = 1$, u turns gray
Explore Adj$[u]$: first neighbor v is white $\rightarrow$ recurse

DFS — Step 2: Visit v (from u)



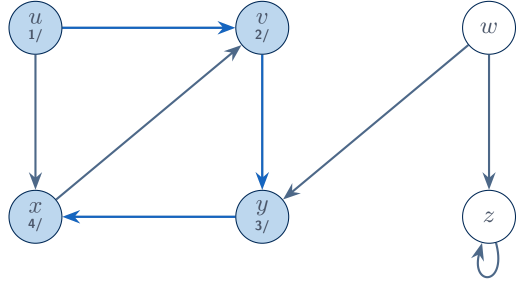
time $= 2$, $v.d = 2$, edge (u, v): **Tree**
Explore Adj[v]: neighbor y is white $\rightarrow$ recurse

DFS — Step 3: Visit y (from v)



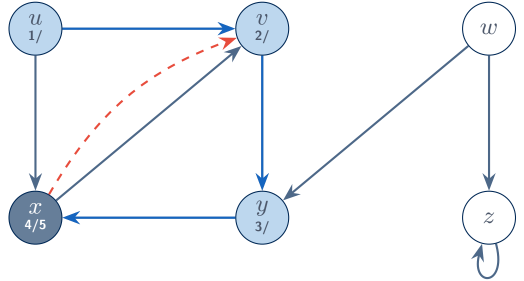
time $= 3$, $y.d = 3$, edge (v, y): **Tree**
Explore Adj$[y]$: neighbor x is white $\rightarrow$ recurse

DFS — Step 4: Visit x (from y)



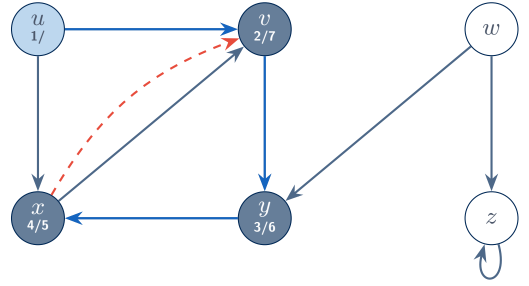
time $= 4$, $x.d = 4$, edge (y, x): **Tree**
Explore Adj$[x]$: neighbor v is gray $\rightarrow$ **Back edge!**

DFS — Step 5: Finish x, backtrack



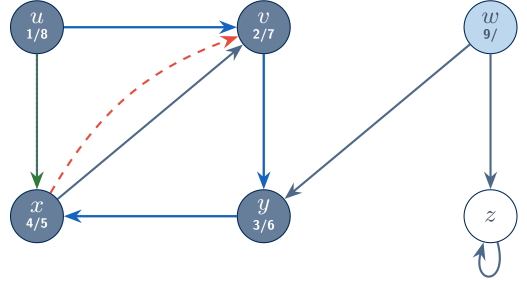
$x.f = 5$, x turns **black**. Edge (x, v): **Back** (v is gray = ancestor).
Backtrack to y: y has no more white neighbors.

DFS — Step 6: Finish y, finish v



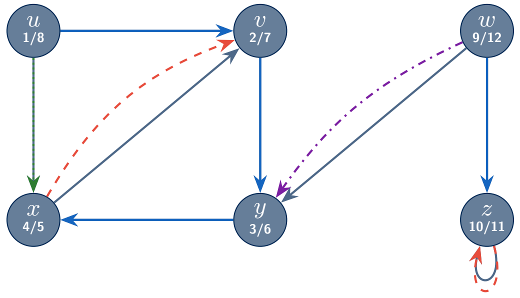
$y.f = 6$, $v.f = 7$. Backtrack to u.
Adj$[u]$: next neighbor x is **black** $\rightarrow$ edge (u, x): **Forward** $(u.d < x.d)$

DFS — Step 7: Finish u, start new tree at w



$u.f = 8$. Back to DFS main loop: w is white $\rightarrow$ new tree.
$w.d = 9$. Adj[w]: y is **black** $\rightarrow$ **Cross** ($w.d > y.f$). z is white $\rightarrow$ recurse.

DFS — Steps 8–9: Visit z, finish z, finish w



$z.d = 10$, (w, z): **Tree**. (z, z): **Back** (self-loop). $z.f = 11$, $w.f = 12$.
DFS complete! Two DFS trees: $\{u, v, y, x\}$ and $\{w, z\}$.

Edge Classification Summary

Edge type	Color of v	Condition	Example
Tree	WHITE	v discovered via (u, v)	$(u, v), (v, y)$
Back	GRAY	v is ancestor of u	$(x, v), (z, z)$
Forward	BLACK	$u.d < v.d$	(u, x)
Cross	BLACK	$u.d > v.d$	(w, y)

Theorem (CLRS 20.10)

In a DFS of an **undirected** graph, every edge is either a **tree edge** or a **back edge**.
(No forward or cross edges.)

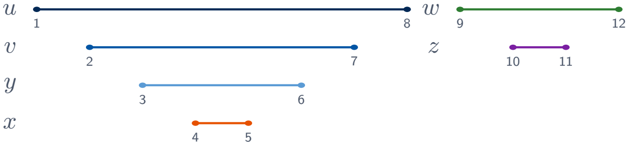
Key Idea

Back edge $\Leftrightarrow$ **cycle**! A directed graph has a cycle iff DFS finds a back edge.

Parenthesis Theorem (CLRS 20.7)

For any two vertices u, v, exactly one holds:

1. $[u.d, u.f]$ and $[v.d, v.f]$ are **disjoint**: neither is a descendant of the other
2. $[u.d, u.f] \subseteq [v.d, v.f]$: u is a descendant of v
3. $[v.d, v.f] \subseteq [u.d, u.f]$: v is a descendant of u



Nested intervals = ancestor/descendant. Disjoint intervals = no relationship.

DFS in Python

Python

```python
def dfs(graph):
    color = {v: 'W' for v in graph}
    d, f = {}, {}
    parent = {v: None for v in graph}
    time = [0]

    for u in graph:
        if color[u] == 'W':
            dfs_visit(u, graph, color,
                      d, f, parent, time)
    return d, f, parent
```

Python

```python
def dfs_visit(u, graph, color,
              d, f, parent, time):
    time[0] += 1
    d[u] = time[0]
    color[u] = 'G'
    for v in graph[u]:
        if color[v] == 'W':
            parent[v] = u
            dfs_visit(v, graph, color,
                      d, f, parent,
                      ↪  time)
    time[0] += 1
    f[u] = time[0]
    color[u] = 'B'
```

Part 5

Topological Sort

Topological Sort — What and Why

Topological Sort

A **topological sort** of a DAG $G = (V, E)$ is a linear ordering of all vertices such that if $(u, v) \in E$, then u appears before v.

Only defined for **directed acyclic graphs (DAGs)**.

Applications:

- **Build systems:** compile files in dependency order (Makefile, CMake)
- **Task scheduling:** prerequisites before dependent tasks
- **Autograd:** evaluate computation graph in forward order
- **Course prerequisites:** which course to take first?
- **Spreadsheet:** evaluate cells in dependency order

Lemma (CLRS 20.11)

A directed graph G is acyclic $\iff$ DFS on G yields **no back edges**.

Topological Sort — Algorithm

Algorithm TOPOLOGICAL-SORT(G):

1. Run DFS(G) to compute finish times $v.f$
2. As each vertex finishes, prepend it to a linked list
3. Return the list

Theorem (CLRS 20.12)

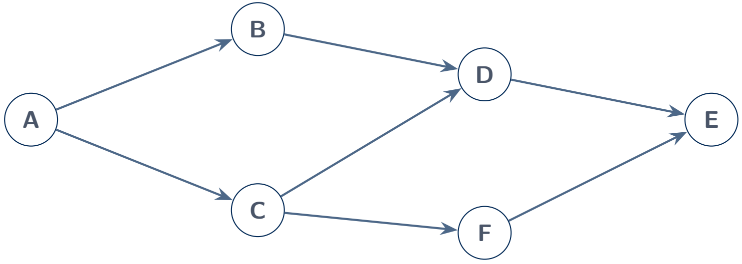
If $(u, v) \in E$, then $v.f < u.f$.
So **decreasing f-order** = topological order.

Complexity: $\Theta(V + E)$ (just a DFS)

Warning

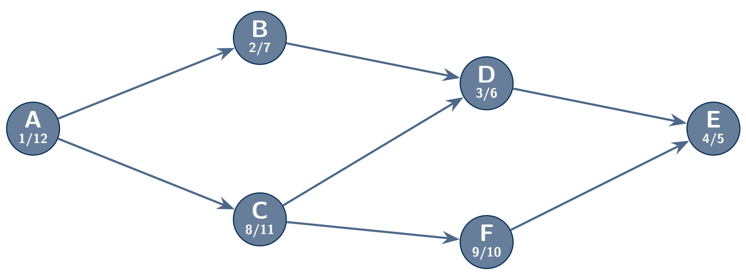
If the graph has a cycle, topological sort is **undefined**. DFS will find a back edge $\rightarrow$ report error.

Topological Sort — Step-by-Step: The DAG



Think: A = "Calculus", B = "Probability", C = "Programming",
D = "Machine Learning", E = "Deep Learning", F = "Databases"

Topological Sort — DFS with Finish Times



DFS from A: $A \to B \to D \to E$ (finish E:5, D:6) $\to$ back to B (finish B:7) $\to$ back to $A \to C$ (D already visited) $\to F$ (E already visited, finish F:10, C:11, A:12.)

Decreasing finish times: $A(12) \to C(11) \to F(10) \to B(7) \to D(6) \to E(5)$

Topological Order

$A \to C \to F \to B \to D \to E$ (one valid ordering — not unique!)

Topological Sort — Kahn's Algorithm (Alternative)

Idea: repeatedly remove vertices with in-degree 0.

Input: DAG $G = (V, E)$

1: Compute in-degree for all vertices
2: $Q \leftarrow$ queue of all vertices with in-degree 0
3: $L \leftarrow$ empty list
4: **while** $Q \neq \emptyset$ **do**
5: $\quad u \leftarrow \textsc{Dequeue}(Q)$
6: $\quad$ Append u to L
7: $\quad$ **for** each $v \in \text{Adj}[u]$ **do**
8: $\quad\quad$ Decrement in-degree of v
9: $\quad\quad$ **if** in-degree of $v = 0$ **then**
10: $\quad\quad\quad \textsc{Enqueue}(Q, v)$
11: $\quad\quad$ **end if**
12: $\quad$ **end for**
13: **end while**
14: **if** $|L| \neq |V|$ **then**
15: $\quad$ **Error:** graph has a cycle!
16: **end if**
17: **return** L

Key Idea

Also $\Theta(V + E)$.

Advantage: no recursion, can detect cycles, good for parallel scheduling.

Both approaches produce valid topological orderings.

Topological Sort in Python (DFS-based)

Python

```
def topological_sort(graph):
    color = {v: 'W' for v in graph}
    order = []

    def dfs_visit(u):
        color[u] = 'G'
        for v in graph[u]:
            if color[v] == 'G':
                raise ValueError("Cycle!")
            if color[v] == 'W':
                dfs_visit(v)
        color[u] = 'B'
        order.append(u)  # append on finish

    for u in graph:
        if color[u] == 'W':
            dfs_visit(u)
    return order[::-1]  # reverse finish order
```

Part 6

Strongly Connected Components

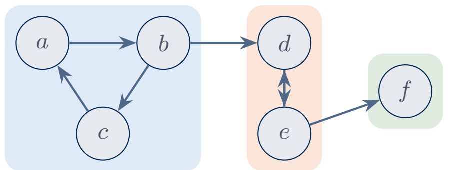
Strongly Connected Components — Definition

Strongly Connected Component (SCC)

A **strongly connected component** of a directed graph $G = (V, E)$ is a **maximal** set of vertices $C \subseteq V$ such that for every pair $u, v \in C$:

$$u \rightsquigarrow v \quad \text{and} \quad v \rightsquigarrow u$$

(every vertex is reachable from every other vertex in C.)



3 SCCs:

- $\{a, b, c\}$: cycle $a \to b \to c \to a$
- $\{d, e\}$: cycle $d \to e \to d$
- $\{f\}$: singleton (no cycle through f)

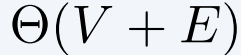
Component graph G^{SCC}:
Always a DAG!

Kosaraju's Algorithm

STRONGLY-CONNECTED-COMPONENTS(G)**:**

1. Run DFS(G) to compute finish times $u.f$
2. Compute G^T (transpose: reverse all edges)
3. Run DFS(G^T), processing vertices in **decreasing** $u.f$ order
4. Each DFS tree in step 3 = one SCC

Complexity



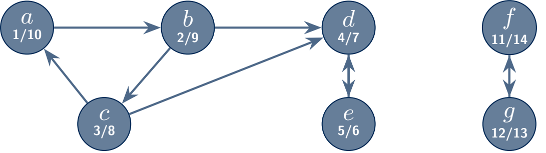
$\Theta(V + E)$

Two DFS passes + transpose.

Key Idea

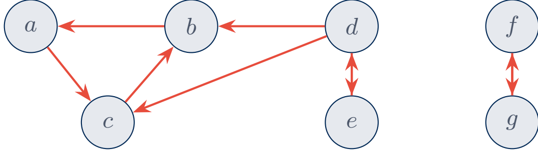
The 2nd DFS visits components in reverse topological order of G^{SCC}. Each tree captures exactly one SCC.

Kosaraju — Step 1: First DFS on G



Finish order (decreasing): $f(14), g(13), a(10), b(9), c(8), d(7), e(6)$

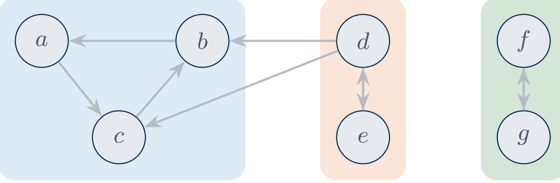
Kosaraju — Step 2: Transpose G^T



All edges reversed (red). Same SCCs as G!

Kosaraju — Step 3: Second DFS on G^T

Process vertices in decreasing finish time: f, g, a, b, c, d, e



Tree 1: start at $f \to g$. So tree: $\{f, g\}$
Tree 2: start at $a \to c \to b$: $\{a, b, c\}$
Tree 3: start at $d \to e$: $\{d, e\}$

Result: 3 SCCs

$\{a, b, c\}$, $\{d, e\}$, $\{f, g\}$ — component graph is a DAG.

Kosaraju's Algorithm in Python (1/2)

Python

```python
def kosaraju(graph):
    # 1. First DFS: compute finish order
    visited, finish_order = set(), []
    def dfs1(u):
        visited.add(u)
        for v in graph[u]:
            if v not in visited:
                dfs1(v)
        finish_order.append(u)
    for u in graph:
        if u not in visited:
            dfs1(u)

    # 2. Build transpose
    transpose = {u: [] for u in graph}
    for u in graph:
        for v in graph[u]:
            transpose[v].append(u)
```

Kosaraju's Algorithm in Python (2/2)

Python

```python
# kosaraju(graph) continued...
# 3. Second DFS on transpose
visited.clear()
sccs = []
def dfs2(u, component):
    visited.add(u)
    component.append(u)
    for v in transpose[u]:
        if v not in visited:
            dfs2(v, component)
for u in reversed(finish_order):
    if u not in visited:
        comp = []
        dfs2(u, comp)
        sccs.append(comp)
return sccs
```

Part 7

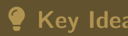
Summary & What's Next

BFS vs. DFS — Comparison

	BFS	DFS
Data structure	Queue (FIFO)	Stack / recursion
Exploration	Level by level	As deep as possible
Complexity	$\mathcal{O}(V + E)$	$\Theta(V + E)$
Shortest paths (unweighted)	✓	×
Cycle detection	only undirected	✓(back edges)
Topological sort	Kahn's variant	DFS finish order
Connected components	✓	✓
Strongly connected comp.	×	✓(Kosaraju)

Complexity Summary

Algorithm	Time	Key result
BFS	$\mathcal{O}(V + E)$	Shortest paths (# edges)
DFS	$\Theta(V + E)$	Timestamps, edge types
Topological Sort	$\Theta(V + E)$	Linear order of DAG
Kosaraju (SCC)	$\Theta(V + E)$	All strongly connected components
Transpose G^T	$\Theta(V + E)$	Reverse all edges



Key Idea

All algorithms this week run in **linear time** $\mathcal{O}(V + E)$.
This is optimal: you need to read the entire input at least once!

What's Next

- **Week 8:** Shortest paths & Minimum Spanning Trees
  - Dijkstra, Bellman-Ford, Kruskal, Prim
  - Union-Find (disjoint sets)
- **Week 9:** A* search & All-pairs shortest paths
  - Floyd-Warshall, heuristic search

Key Idea

Graphs are the **foundation** of most algorithms you'll use in AI: from training neural networks (computation graphs) to knowledge graphs, from planning (state-space search) to optimization (network flows).

Master BFS, DFS, and their applications — everything else builds on them.