Parallel And Sequential Data Structures and Algorithms

Graphs and Depth-First Search

Learning Objectives

- Review **graph** definitions and representations
  - Undirected, direction, weighted, unweighted graphs
  - Neighbours, paths, cycles, components
  - Adjacency lists, adjacency matrices
- Understand **depth-first search** and its applications
  - Explore DFS in **more depth** than in 15-122
  - Develop a reusable algorithmic template for many graph problems

Definitions

Definition (Graph): A **graph** consists of a collection of vertices (things) and **edges** (connections between those things)

Formally, we write $G = (V, E)$ where

- V is the set of vertices
- E is the set of edges, each connecting two vertices

- Edges can be **directed** or **undirected**
- Edges may have associated **weights**
- By convention, we use $n = |V|$ for the number of vertices and $m = |E|$ for the number of edges

Definitions

Definition (Graph): A **graph** consists of a collection of vertices (things) and **edges** (connections between those things)

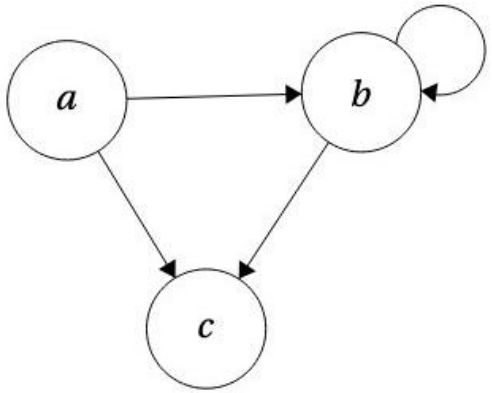
Formally, we write $G = (V, E)$ where

- V is the set of vertices
- E is the set of edges, each connecting a pair of vertices (u, v)

Example: This **directed graph** has:

$$V = \{a, b, c\}, \qquad E = \{(a, b), (b, b), (b, c), (a, c)\}$$

Since the graph is directed, the edge from a to b is represented by the ordered pair (a, b)



Graph Classification

- The graph we saw on the last slide was **directed** (every edge has a **source** and a **destination** vertex)

Definition (Undirected Graph): A graph is **undirected** if its edges have no direction, meaning (u, v) and (v, u) represent the same edge.

Definition (Simple Graph): A **simple graph** is an **undirected** graph with no **self-loops** or **multi-edges** (an edge that appears multiple times)

Graph Substructures

Definition (Neighbour): A vertex is a **neighbour** of another vertex in an undirected graph if the two vertices are connected by an edge

- The neighbourhood of v is the set of its neighbours:

$$N_G(v) \coloneqq \{u : (u, v) \in E\}$$

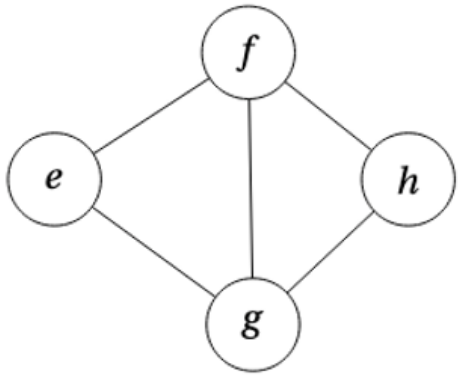
- For a directed graph, we say u is an **in-neighbour** of v if there is an edge from u to v. Then, v is an **out-neighbour** of u
- The set of all in-neighbours of v in graph G is denoted $N_G^-(v)$ and the set of all out-neighbours as $N_G^+(v)$

Graph Substructures

Definition (Degree): In a simple undirected graph G, the **degree** of a vertex v is the number of neighbours it has. That is,

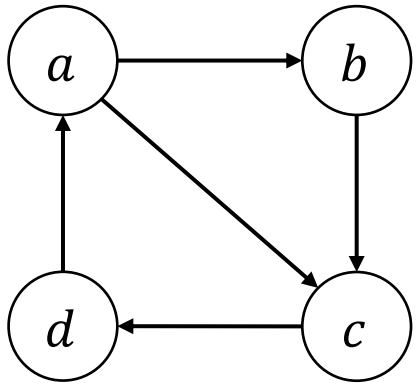
$$d_G(v) = |N_G(v)|$$

- Similarly, the **in-degree** and **out-degree** of a vertex v in a directed graph G, are denoted by $d_G^-(v)$ and $d_G^+(v)$ respectively



$d_G(g) = \mathbf{3}$

$d_G(h) = \mathbf{2}$



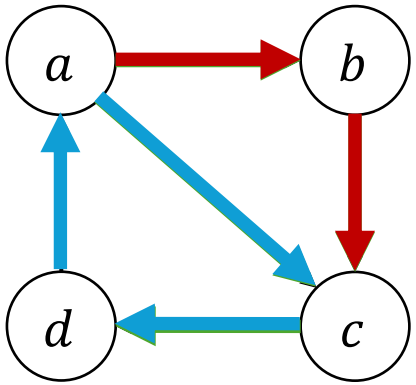
$d_G^+(a) = \mathbf{2}$

$d_G^-(b) = \mathbf{1}$

Graph Substructures

Definition (Path): A **path** is a sequence of vertices such that each adjacent pair in the sequence has an edge between them in the graph. The length of a path is the number of edges in the path.

Definition (Simple Path): A path that does not reuse any vertices



- $[a, b, c, d]$ is a path
- $[a, c, d]$ is a path
- $[c, b, a]$ is **not** a path (uses edge backwards)
- Is $[a, c, d, a]$ a path?
  Yes, but not **simple**

Graph Substructures

Definition (Path): A **path** is a sequence of vertices such that each adjacent pair in the sequence has an edge between them in the graph. The length of a path is the number of edges in the path.

Definition (Simple Path): A path that does not reuse any vertices

Terminology: Graph theorists and computer scientists cannot agree on terminology!

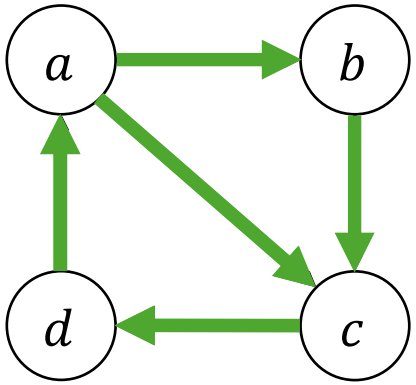
- Many graph theory books use path == simple path and call a non-simple path a **walk**
- A path that can repeat vertices but not repeat edges is then called a **trail**

We will select the convention where a path can repeat

Graph Substructures

Definition (Cycle): A path that starts and ends at the same vertex
- In an undirected graph, must have at least three edges
- In a directed graph, must have at least one edge

Definition (Simple Cycle): a cycle that does not reuse any vertices or edges (excluding the starting/ending vertex)



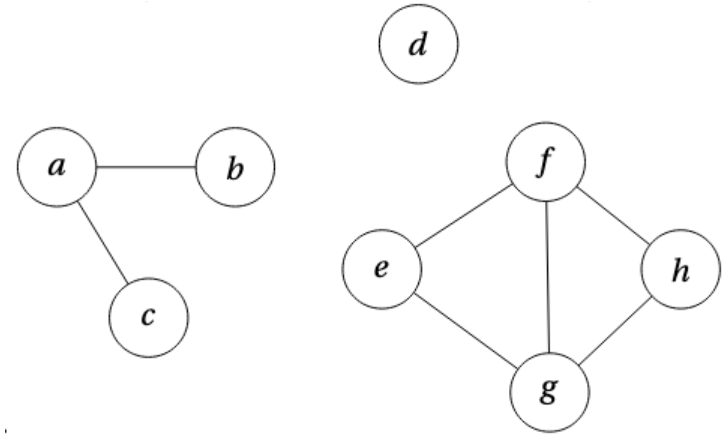
- $[a, c, d, a]$ is a cycle
- Is $[c, d, a, b, c, d, a, c]$ a cycle?
  Yes, but not **simple**

Graph Substructures

Definition (Reachability): A vertex v is reachable from a vertex u if there exists a path that starts at u and ends at v.

Definition (Connected Component): a connected component of an undirected graph is a maximal subset C of the vertices such that, given any vertex v in C, all other vertices are reachable from v

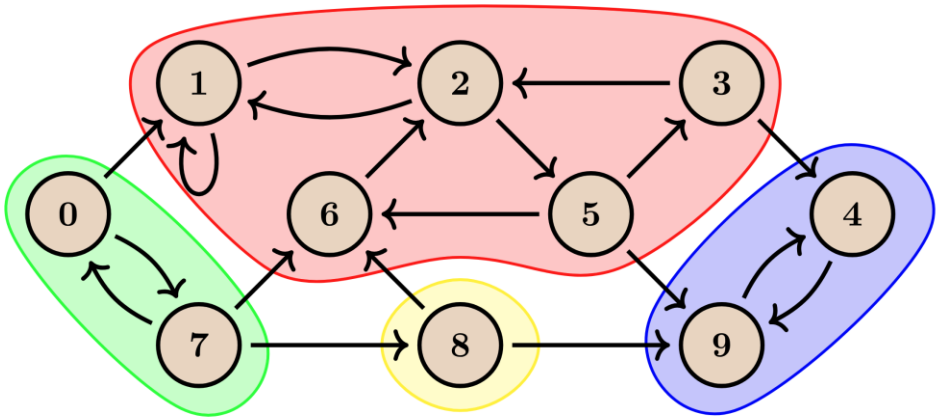
Example: in this graph, the connected components are $\{a, b, c\}, \{d\}, \text{and } \{e, f, g, h\}$



Graph Substructures

Definition (Strongly Connected Component): In a directed graph, two vertices u and v are in the same **strongly connected component** (SCC) if and only if u is reachable from v and v is reachable from u.

Example: in this graph, the strongly connected components are $\{0,7\}, \{8\}, \{4,9\}$, and $\{1,2,3,6,5\}$



Representing Graphs

Graph Representation

- How should we represent a graph for use in algorithms?
- By default, we focus on **enumerable graphs**, where the n vertices of the graph are labeled with the integers 0 to $n-1$
- Different representations may support different operations more efficiently

The key operation for many graph algorithms: given a vertex v, obtain a list/sequence of its neighbors

Adjacency Lists (Enumerable Graphs)

Definition (Adjacency List): for each vertex v, store the edges to the **out-neighbours** of v (for an undirected graph, the neighbours)

- If the graph is **enumerable**, an adjacency list is a type satisfying

```
sequence<sequence<Edge>>
```

- Given an adjacency list `adj`, the outgoing edges of u are `adj[u]`
- Allows for $O(1)$ access to a vertex's adjacency list

Terminology: Turns out its not just graph theorists; computer scientists can't agree with each other! Some textbooks use adjacency lists strictly to mean **linked lists**. We will use the convention that an adjacency list is any per-vertex collection of edges.

Adjacency Lists (Enumerable Graphs)

Definition (Adjacency List): for each vertex v, store the edges to the **out-neighbours** of v (for an undirected graph, the neighbours)

- If the graph is **enumerable**, an adjacency list is a type satisfying

```
sequence<sequence<Edge>>
```

- Given an adjacency list `adj`, the outgoing edges of u are `adj[u]`
- Allows for $O(1)$ access to a vertex's adjacency list

- **Unweighted Graphs**: `type Edge = int`
- **Weighted Graphs**: `type Edge = { to : int, weight : W }`

Adjacency Lists (Other Vertex Types)

Definition (Adjacency List): for each vertex v, store the edges to the **out-neighbours** of v (for an undirected graph, the neighbours)

- For non-enumerable graphs, vertices have arbitrary labels of type `V`, rather than necessarily being indexed $0, \ldots, n-1$
- The only change to our representation is how we locate each vertex's corresponding adjacency list:

```
dictionary<sequence<Edge>>
```

- **Unweighted Graphs**: `type Edge = V`
- **Weighted Graphs**: `type Edge = { to : V, weight : W }`

Adjacency Lists Costs

Claim (Cost of Adjacency Lists): For both the enumerable and non-enumerable version, an adjacency list uses $\Theta(n + m)$ space.

For obtaining the neighbors of a vertex u, the cost is $O(1)$ for an enumerable graph. If the graph is not enumerable, we can either use:

- **A hash table dictionary**: lookup has $O(1)$ worst-case expected cost
- **A balanced-BST dictionary**: lookup has $O(\log n)$ worst-case cost

Adjacency Matrices

Definition (Adjacency Matrix): For an **enumerable graph**, we store an $n \times n$ matrix

```
sequence<sequence<W>>
```

where entry (u, v) indicates the weight of the edge from u to v. In an unweighted graph, the weights can simply be `bool`s.

- Allow constant-time edge existence queries
- However, finding all the neighbors of a vertex requires searching an entire row and takes $\Theta(n)$ time
- The space usage is $\Theta(n^2)$, which is inefficient for sparse graphs but sometimes the best choice for dense graphs

Adjacency Matrices

Definition (Adjacency Matrix): For an **enumerable graph**, we store an $n \times n$ matrix

```
sequence<sequence<W>>
```

where entry (u, v) indicates the weight of the edge from u to v. In an unweighted graph, the weights can simply be `bool`s.

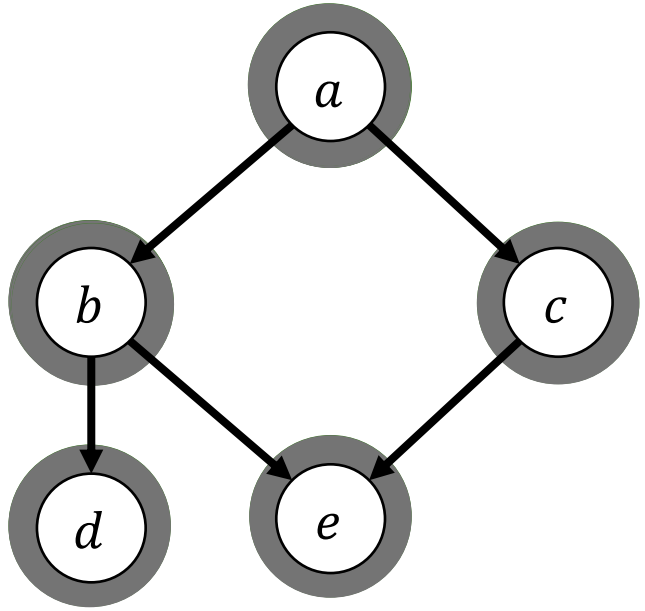
Absent edges: The value used to represent an absent edge depends on the application (e.g., ∞ for shortest paths, 0 for network capacities, etc).

Depth-First Search

Depth-First Search (DFS)

- Recall **depth-first search (DFS)** from 15-122

- Start at a given vertex, marking it as **visited**.
- Repeatedly explore an **unvisited neighbour**, continuing as **deep** as possible.
- When there are no unvisited neighbours, **backtrack** to the most recent vertex with unexplored alternatives.
- Recursion naturally expresses this process: **fully explore one neighbour before moving to the next**.



DFS as a Reusable Template

- We've seen how DFS explores a graph. But how do we actually **compute something** while exploring?
- Rather than designing a new algorithm for each problem, we can start with the same DFS code and **customize it**
- There are initially two places to customize it:
  - **what happens when we discover a vertex**?
  - **which vertices we start DFS from**?

Key idea: DFS is a **reusable** traversal pattern. Different graph algorithms customize what happens during traversal and where searches begin.

DFS as a Reusable Template

```
fun dfs_application(adj : UnweightedAdjList, ...):
  n = |adj|
  visited = [False for _ in 0...n-1]

  fun dfs(u : int):
    visited[u] ← True
    // DISCOVER(u)
    for v in adj[u]:
      if not visited[v]:
        dfs(v)

  // Run DFS from appropriate vertices
```

Red = Application-specific code goes here

Property: `dfs(u)` visits **every unvisited vertex reachable** from u.

Application 1: Reachability

Problem: Given vertices s and t, determine whether t is reachable from s.

```
fun reachable(adj : UnweightedAdjList, s : int, t : int):
  n = |adj|
  visited = [False for _ in 0...n-1]

  fun dfs(u : int):
    visited[u] ← True
    // DISCOVER(u): nothing to do
    for v in adj[u]:
      if not visited[v]:
        dfs(v)

  dfs(s); return visited[t]
```

Cost: costs $O(n + m)$ work (and span; it's sequential!)

Application 2: Connectivty

Problem: Given an undirected graph, assign each vertex an integer label identifying its **connected component**.

```
fun connected_components(adj : UnweightedAdjList):
  n = |adj|
  visited = [False for _ in 0...n-1]
  component = [0 for _ in 0...n-1]
  id = 0

  fun dfs(u : int):
    visited[u] ← True
    component[u] ← id  // DISCOVER(u)
    for v in adj[u]:
      if not visited[v]:
        dfs(v)

  for u in 0...n-1:
    if not visited[u]:
      dfs(u)
      id ← id + 1

  return component
```

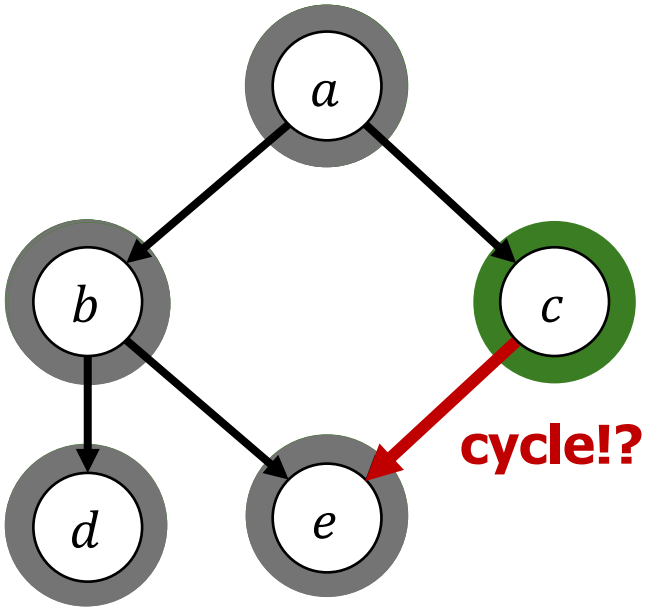
"DFS All" is a very common pattern

More General DFS

Cycle Finding

Problem: Given a directed graph, does it contain a cycle?

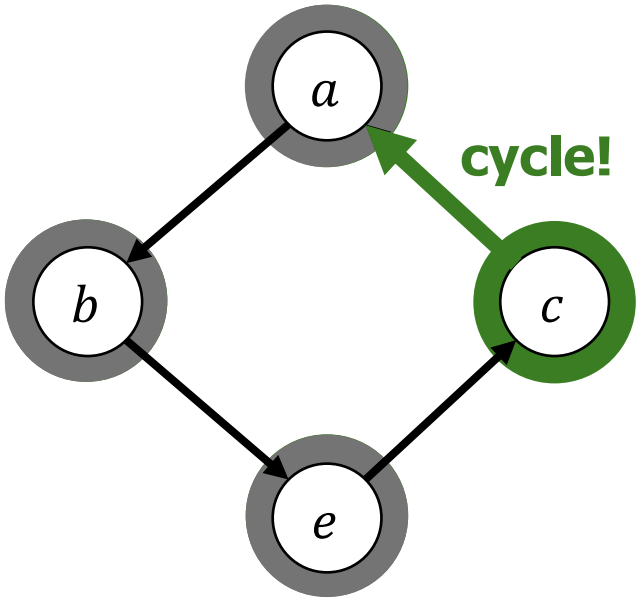
- **(Almost correct) idea**: If DFS encounters an edge to an already-visited vertex, we've found a cycle!



Cycle Finding

Problem: Given a directed graph, does it contain a cycle?

- **(Almost correct) idea**: If DFS encounters an edge to an already-visited vertex, we've found a cycle!



Key Distinction:
- A vertex might be **currently being explored**.
- Or its exploration might have **already finished**.

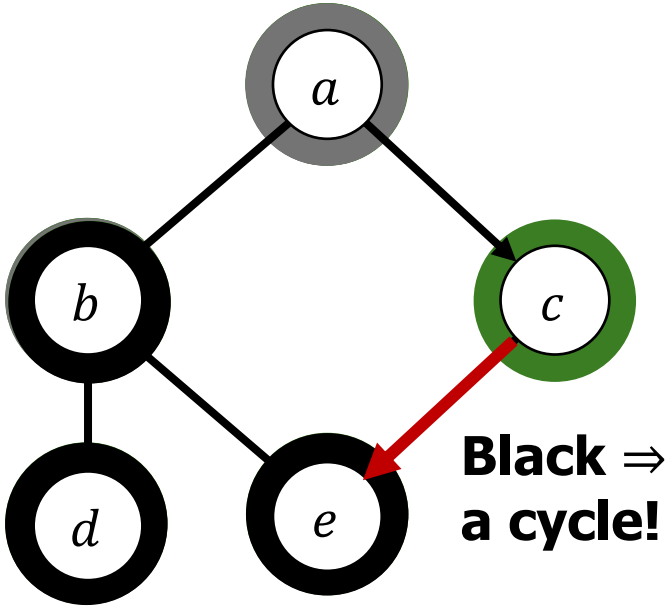
DFS finds a cycle when it encounters an edge to a vertex that is **still being explored**.

Three-Colour DFS

Motivation: We need to distinguish vertices that are **currently being explored** from those whose **exploration has finished**.

- We can **generalize** our DFS algorithm
  - Instead of a Boolean visited
  - **WHITE**: Not yet discovered
  - **GRAY**: Discovered, but still exploring
  - **BLACK**: Exploration has finished

Property: The **GRAY** vertices are exactly those who's `dfs` calls are currently active.

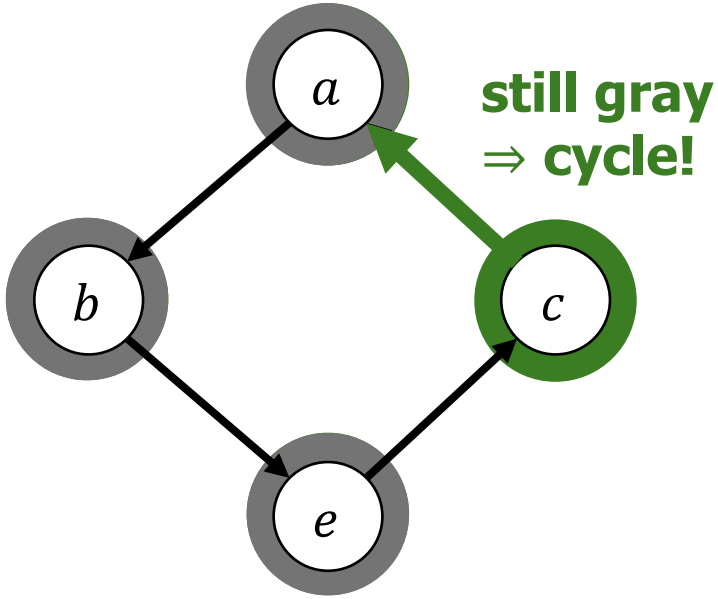


Black ⇒ not a cycle!

Three-Colour DFS

Motivation: We need to distinguish vertices that are **currently being explored** from those whose **exploration has finished**.

- We can **generalize** our DFS algorithm
  - Instead of a Boolean visited
  - **WHITE**: Not yet discovered
  - **GRAY**: Discovered, but still exploring
  - **BLACK**: Exploration has finished



Property: The **GRAY** vertices are exactly those who's `dfs` calls are currently active.

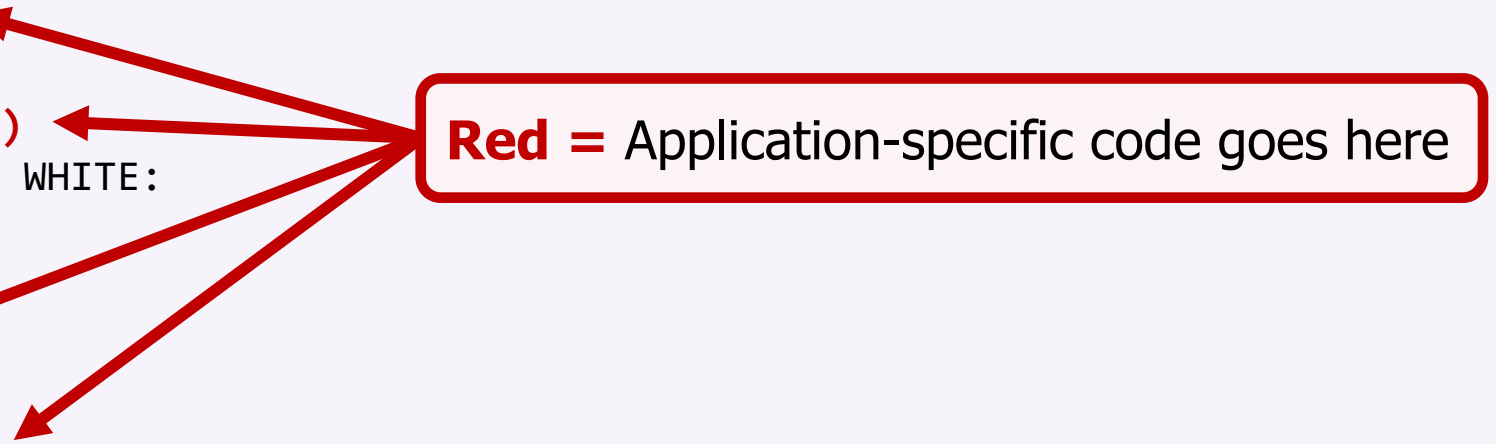
Three-Colour DFS Template

```
fun dfs_application(adj : UnweightedAdjList, ...):
  n = |adj|
  color = [WHITE for _ in 0...n-1]

  fun dfs(u : int):
    color[u] ← GRAY
    // DISCOVER(u)
    for v in adj[u]:
      // EXAMINE(u,v)
      if color[v] == WHITE:
        dfs(v)

    // FINISH(u)
    color[u] ← BLACK

  // Run DFS from appropriate vertices
```



Application 3: Cycle Detection

Problem: Given a directed graph, does it contain a cycle?

```
fun has_cycle(adj : UnweightedAdjList):
  n = |adj|
  color = [WHITE for _ in 0...n-1]
  found = False

  fun dfs(u : int):
    color[u] ← GRAY
    for v in adj[u]:
      if color[v] == GRAY: found ← True
      if color[v] == WHITE:
        dfs(v)

    color[u] ← BLACK

  for u in 0...n-1:
    if color[u] == WHITE:
      dfs(u)

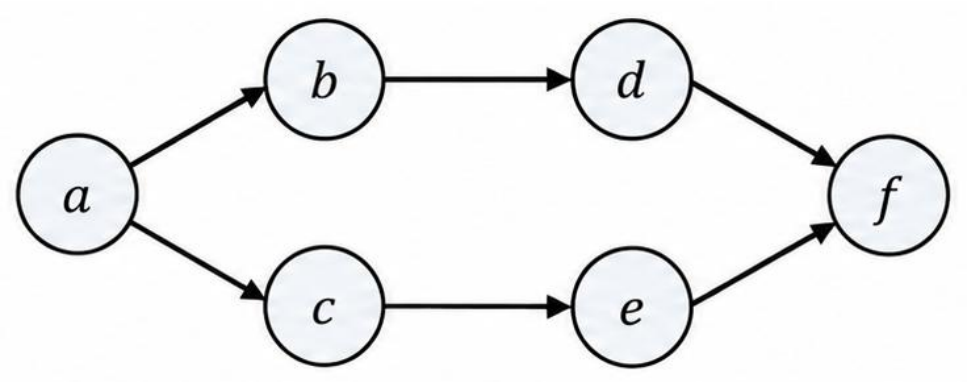
  return found
```

"DFS All" pattern again!

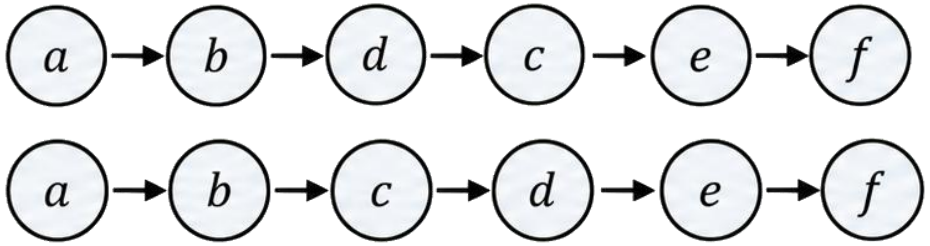
Topological Sorting

Problem: Given a directed acyclic graph (DAG), find an **ordering of its vertices** such that for every edge (u, v), u appears before v

Example: Course prerequisites



Two example topological orders



for every edge (u, v), u appears before v

Key Idea: In a DAG, for every edge (u, v), v finishes before u in a DFS. Therefore, **reversing the order in which vertices finish gives a topological ordering**

Application 4: Topological Sorting

Problem: Given a DAG, return a topological ordering of its vertices

```
fun topological_sort(adj : UnweightedAdjList):
  n = |adj|
  color = [WHITE for _ in 0...n-1]
  order = []

  fun dfs(u : int):
    color[u] ← GRAY
    for v in adj[u]:
      if color[v] == WHITE:
        dfs(v)

    order.append(u)  // FINISH(U)
    color[u] ← BLACK

  for u in 0...n-1:
    if color[u] == WHITE:
      dfs(u)

  return reverse(order)
```

Topological Sorting: Correctness

Theorem: For every edge (u, v) in a DAG, v finishes before u

- Proof: Consider what happens when DFS examines an edge (u, v). There are three possibilities:
  1. **WHITE**: DFS recursively explores v, so v finishes before u.
  2. **GRAY**: Impossible! An edge to a gray vertex would imply a cycle.
  3. **BLACK**: v has already finished, so v finishes before u.
- Therefore, for every edge (u, v), v finishes before u.

Corollary: Reverse finish order is a valid topological ordering.

Even More General DFS

DFS Start and Finish Times

Motivation: Three-colour DFS tells us which vertices are currently being explored, but after DFS finishes, every vertex is BLACK. What if we want to **preserve information** about the structure and nesting of recursive explorations?

- We can generalize the colours by recording two **timestamps**:
  - `start[u]`: When DFS first discovered vertex u
  - `finish[u]`: When DFS finished exploring u
- This generalizes the three-colour scheme:
  - **WHITE**: `start[u] == NONE`
  - **GRAY**: `start[u] != NONE` and `finish[u] == NONE`
  - **BLACK**: `finish[u] != NONE`

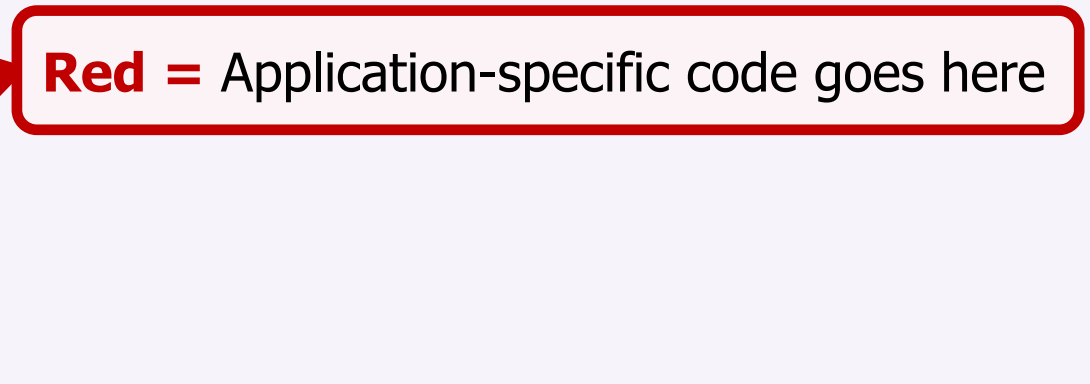
DFS Start and Finish Times

```
fun dfs_application(adj : UnweightedAdjList, ...):
  n = |adj|
  start = [NONE for _ in 0...n-1]
  finish = [NONE for _ in 0...n-1]
  time = 0

  fun dfs(u : int):
    start[u] ← time++
    // DISCOVER(u)
    for v in adj[u]:
      // EXAMINE(u,v)
      if start[v] == NONE:
        dfs(v)
        // RETURN(u,v)

    // FINISH(u)
    finish[u] ← time++

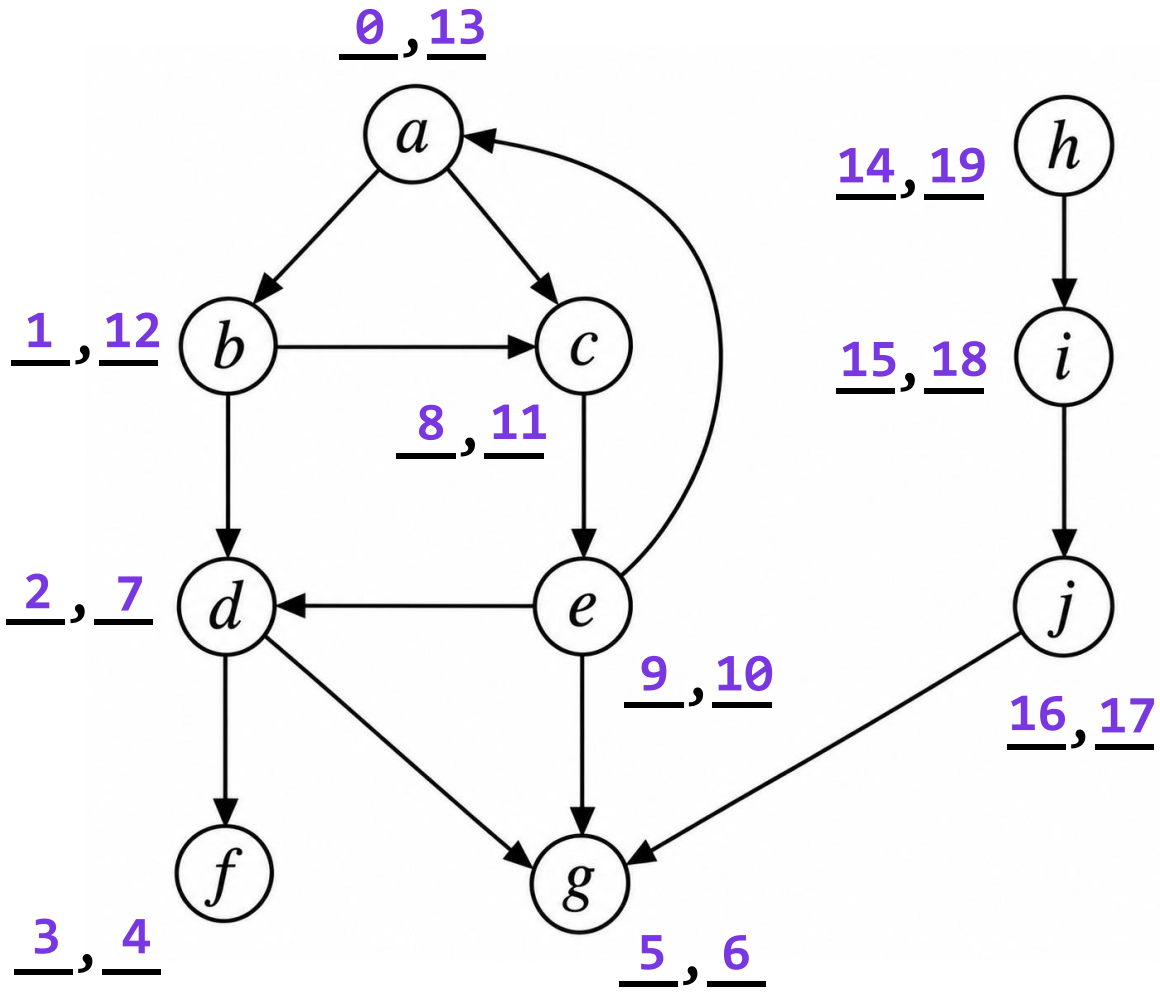
  // Initiate DFS from appropriate vertices
```



Vertex Numbering

Record two **timestamps**:

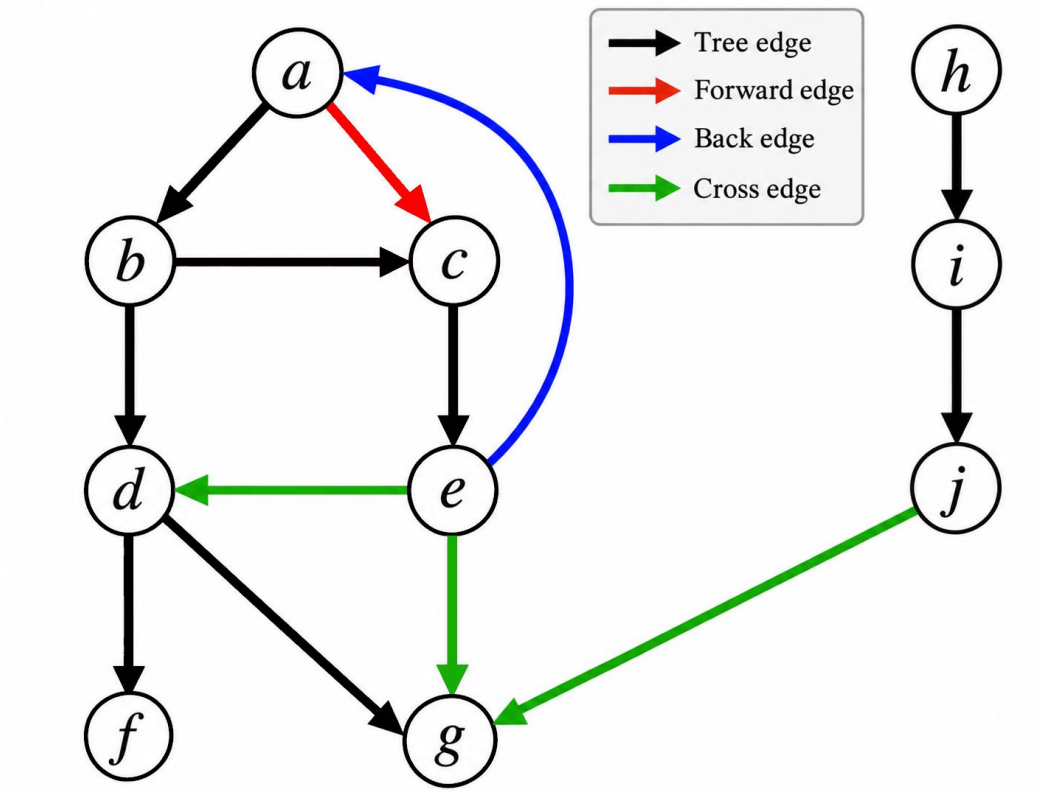
- `start[u]`: When DFS first discovered vertex u
- `finish[u]`: When DFS finished exploring u



Classification of DFS Edges

Definition (DFS Edges):

- **Tree edges** occur when a vertex is first visited.
- **Forward edges** are non-tree edges that go from a vertex to one of its descendants in the tree.
- **Back edges** go from a vertex to one of its ancestors in the tree.
- **Cross edges** connect two vertices neither of which is an ancestor of the other



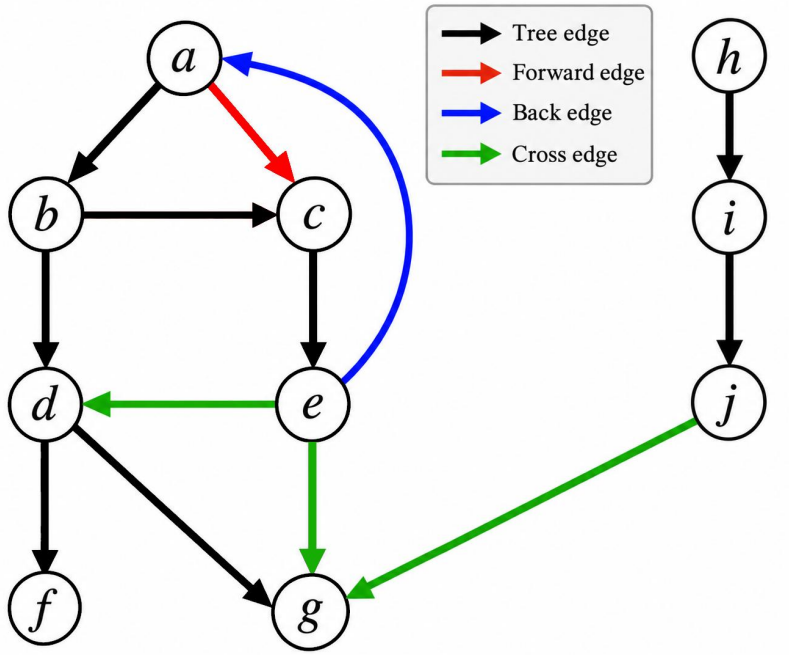
Classification via Timestamps

Theorem (DFS Interval Property): For any two vertices, their [start, finish] intervals are either disjoint or one contains the other

For an edge (u, v), the timestamps give us three possible relationships:

- **Tree or Forward**: $s_u < s_v < f_v < f_u$
- **Back**: $s_v < s_u < f_u < f_v$
- **Cross**: $f_v < s_u$

Note: Timestamps alone cannot distinguish tree edges from forward edges



Summary

- **DFS as a Reusable Template**
  - **Visited**: Reachability and connected components.
  - **Three colors**: Distinguish vertices currently being explored from those already finished, enabling directed cycle detection.
  - **Finish events**: Topological sorting by reversing the finish order
  - **Start/finish timestamps**: Records DFS history, enabling ancestry queries and edge classification.

Key takeaway: The underlying DFS traversal stays the same. Different applications use different information and events during the traversal.