Graph Traversal

10.020 Data Driven World

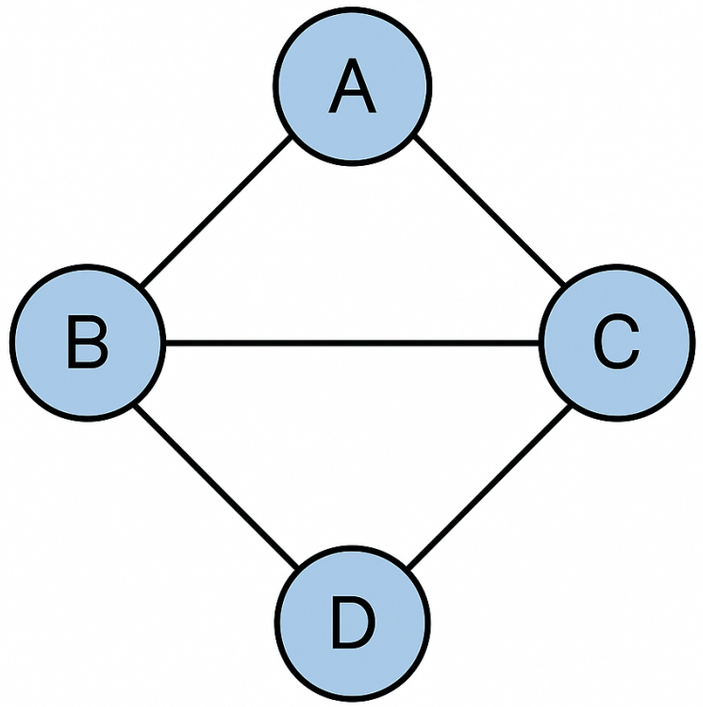
Natalie Agus

Learning Objectives

- Define graph, vertices, edges and weights.
- Use **Dictionary** and **OOP** to represent graph.
    - Represent graphs using **adjacency-list** representation or adjacency-matrix representation.
    - Differentiate **directed** and **undirected** graphs.
- Define paths.
- Create a **Vertex** class and a **Graph** class.
- **Extend** class Vertex and Graph for **graph traversal algorithm**
- Explain and implement **breadth first search**
- Explain and implement **depth first search.**

Graph

Basic Elements

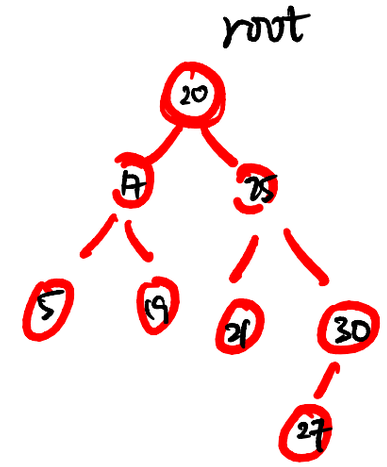


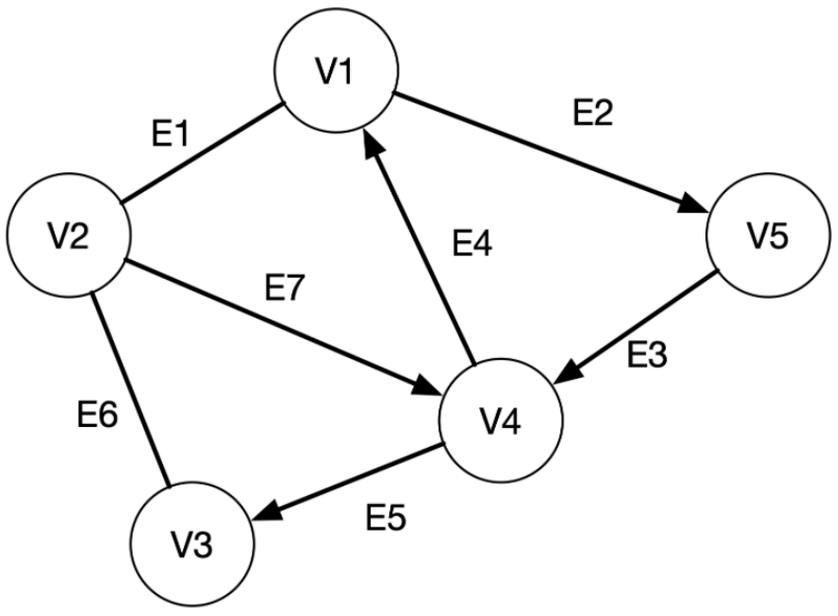
- Graph is a **data structure:**
  - Just like queue, stack, deque, heap
  - Non-linear data structure
- A graph has **nodes** **(vertices)** connected by **edges**
- Application:
  - Model networks (nodes are entities, edges are connections): social network, network routing, recommendation system
  - Game level maps: **pathfinding**
  - **Navigation**

Graph

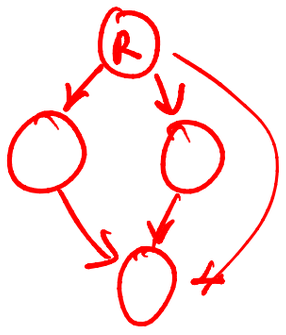
Directions

- It can be **unidirectional** or **bidirectional**
  - **Directed vs undirected graphs**
- Tree is a form of graph that does not form a **cycle**





↳ acyclic ≠ tree



* There's only one path to reach any node from the root

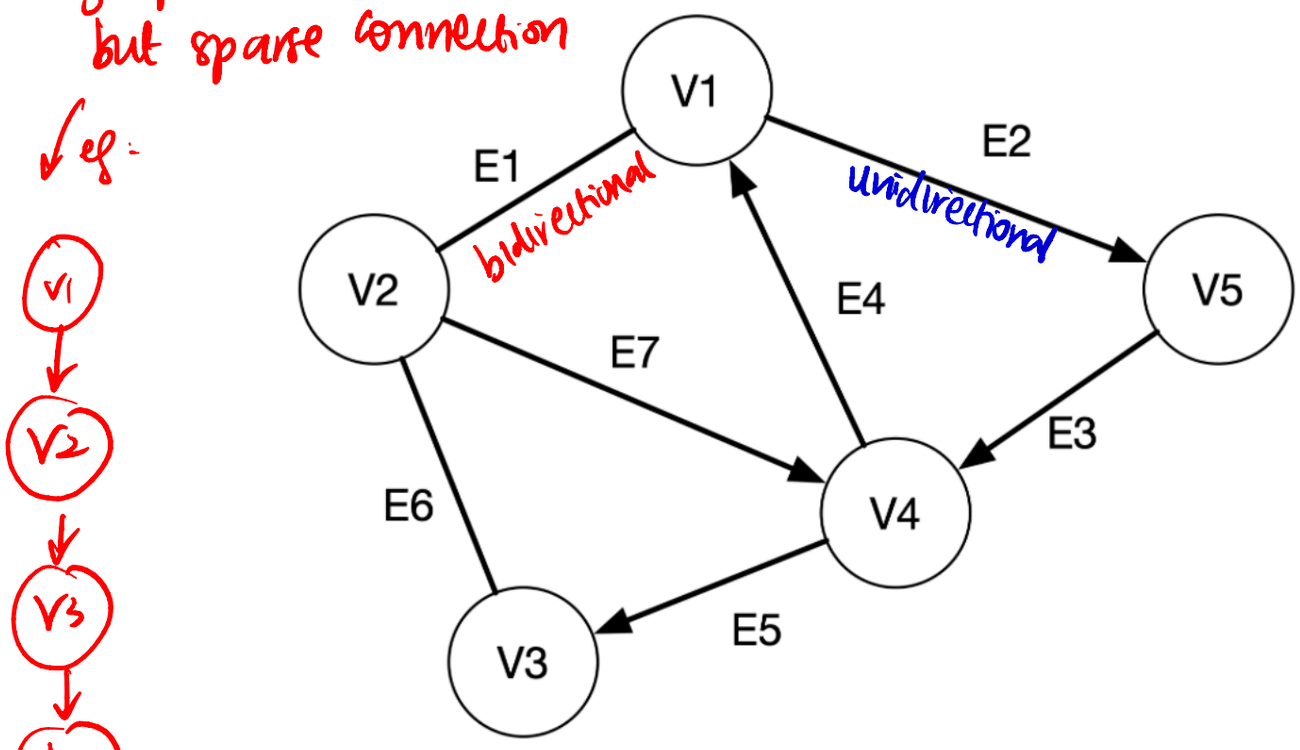
Represent Graph in Code

Adjacency Matrix

- **Cons**: might result in sparse matrix

graph with a lot of nodes but sparse connection

↳ waste space



eg:

a "chain"

* 99 edges
* 100 nodes
* 100^2 of spaces in the matrix
* $100^2 - 99 \rightarrow 0$s.

```
graph = [[0,1,0,0,1],
         [1,0,1,0,0],
         [0,1,0,0,0],
         [1,0,1,0,0],
         [0,0,0,1,0]]
```

to 0 1 2 3 4

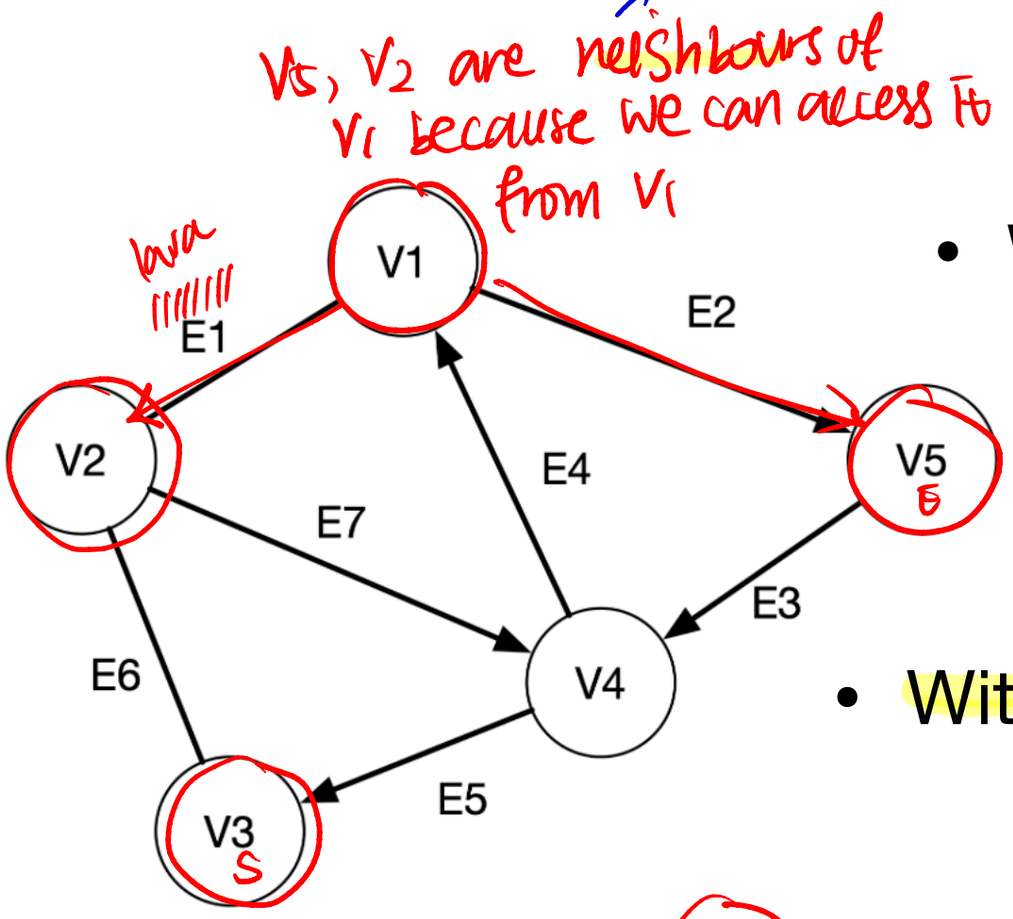
from

0	V1		1		1
1	V2	1		1	1
2	V3		1		
3	V4	1		1	
4	V5				1

Represent Graph in Code

Adjacency List

- Suitable if the number of edges is not large



- Without cost

```
graph1 = {'V1': ['V2', 'V5'],
          'V2': ['V1', 'V3', 'V4'],
          'V3': ['V2'],
          'V4': ['V1', 'V3'],
          'V5': ['V4']}
```

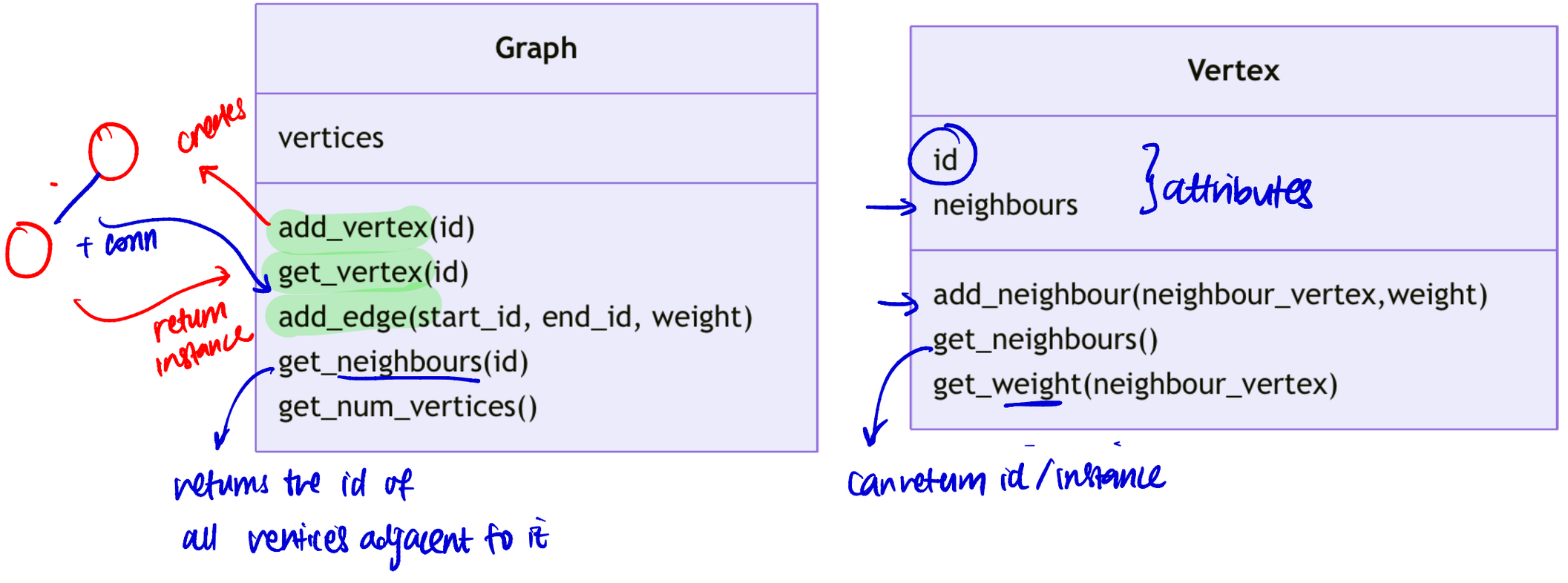
- With cost

```
graph1 = {'V1': {'V2': 1, 'V5': 1},
          'V2': {'V1': 1, 'V3': 1, 'V4': 1},
          'V3': {'V2': 1 },
          'V4': {'V1': 1, 'V3': 1},
          'V5': {'V4': 1}}
```

Represent Graph in Code

Using OOP

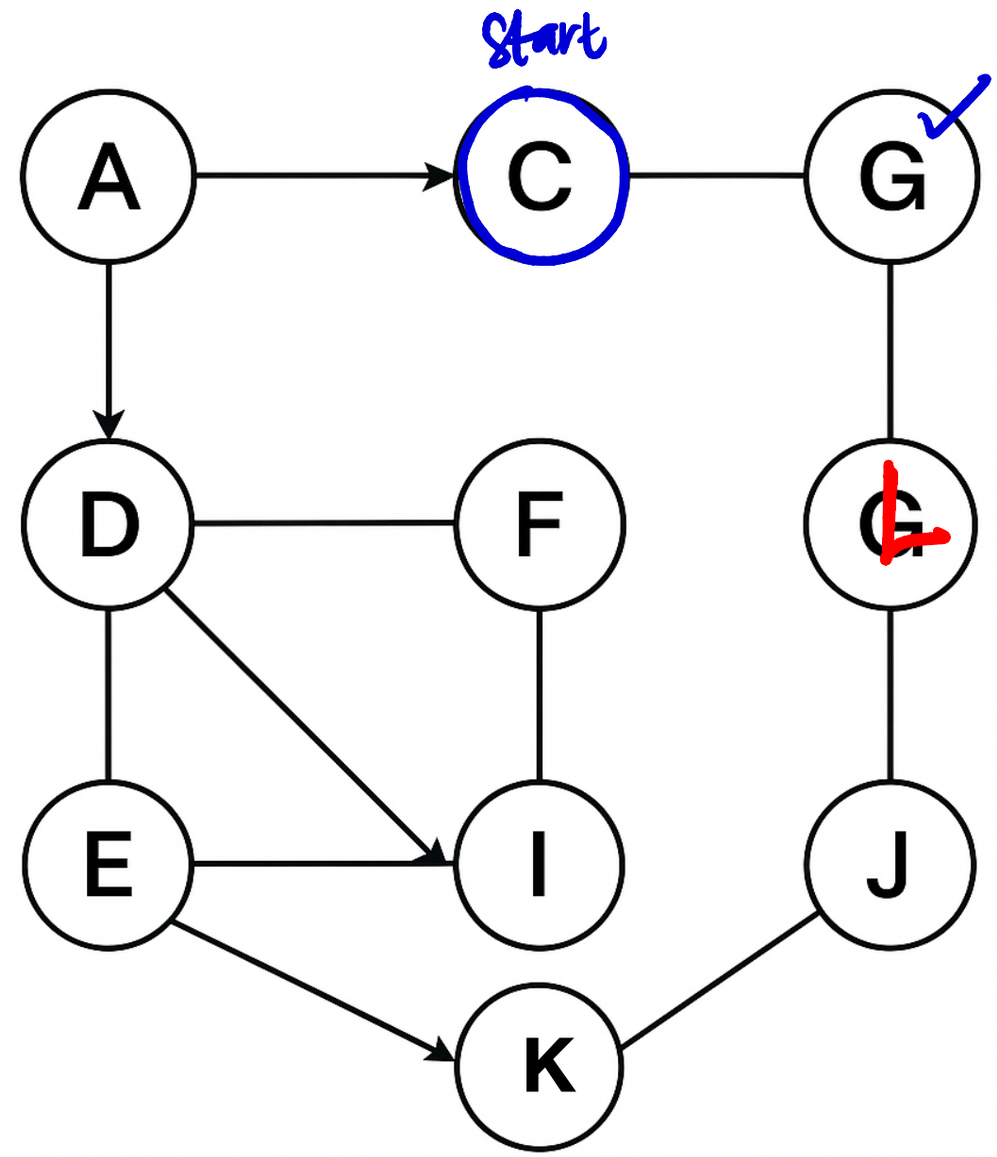
- It's a **has-a** relationship (composition)
  - A graph has a list of **Vertices**



Graph Traversal

Given a starting node, how do we "walk" the graph?

- **Breadth-first** search
- **Depth-first** search
- You need to know:
  - **Starting** node
  - **Neighbours** of each node (directional)
  - Edge costs (optional)

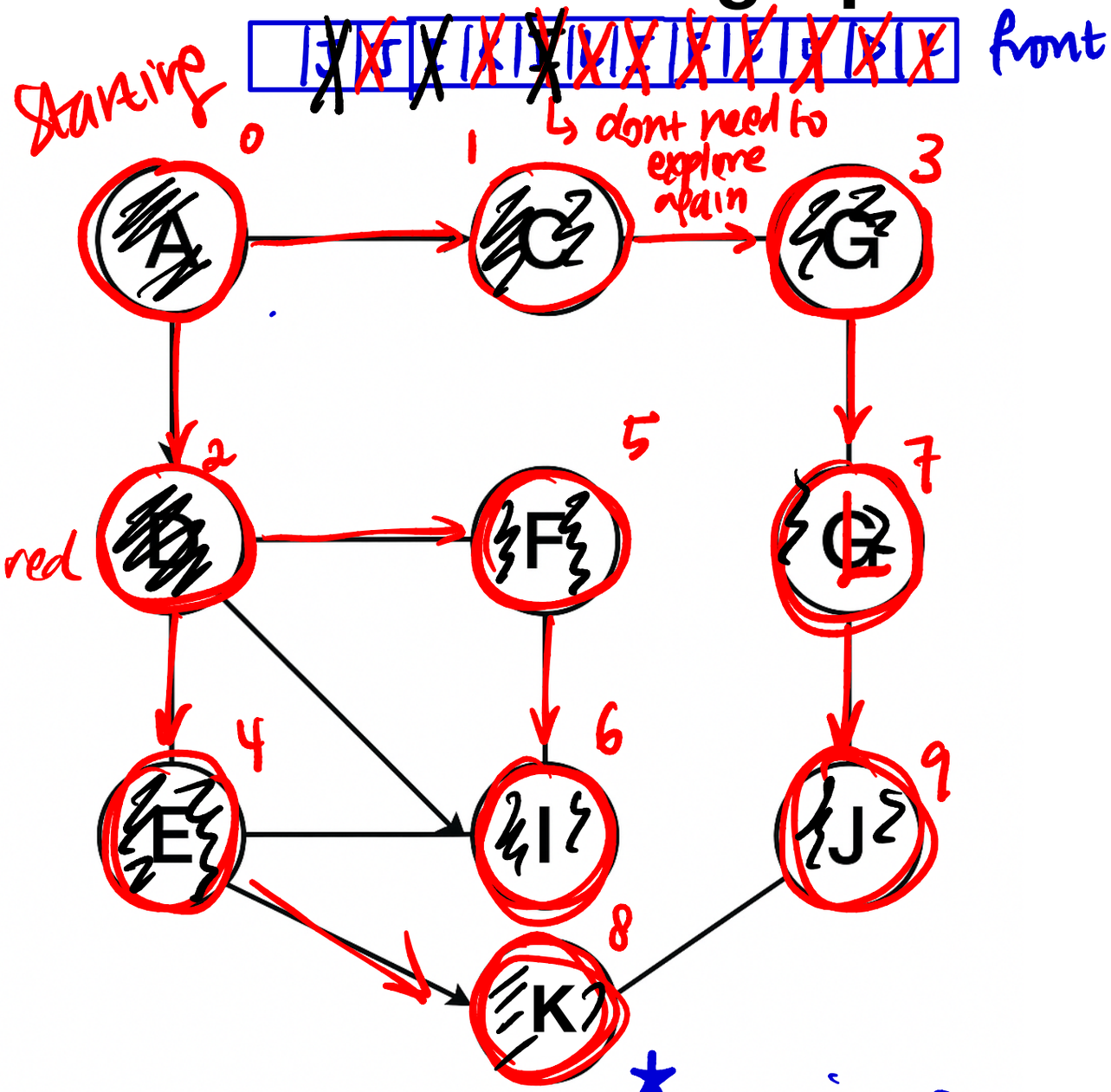


Breadth-first Search

(black) pop()
① Entered → explored
② find all the neighbours, put them in a queue if they're not black

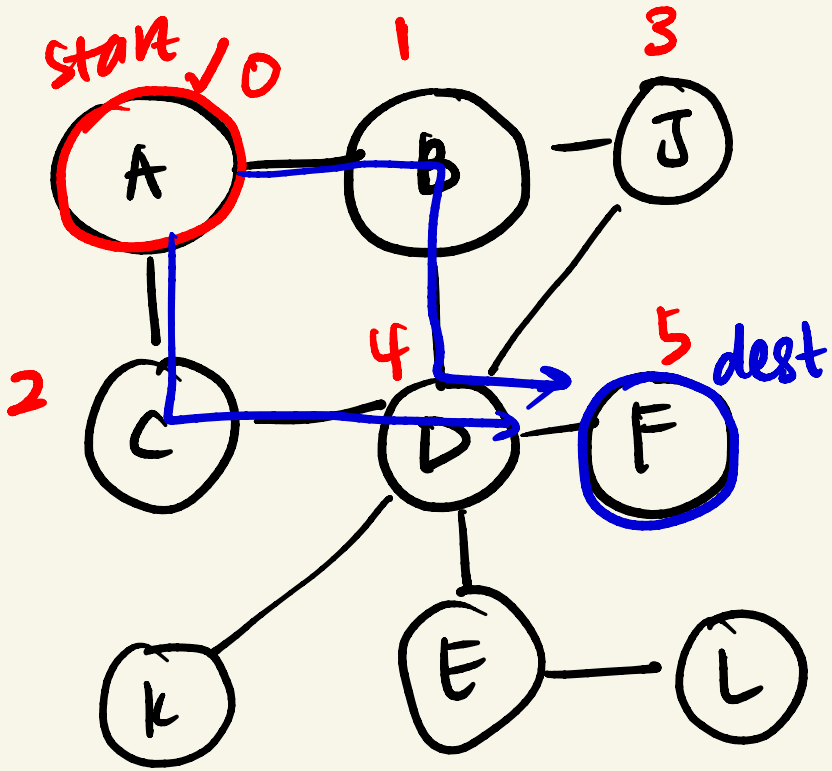
Given a starting node, how do we "walk" the graph?

- **Always** queue nodes in the order they are discovered
  - **Useful data structure:** Queue
  - **2-Colour concept**: white (new), black (done)
    ↓explored
  - **If weight is constant,** BFS can be used to find **shortest path**



BFS order of traversal : A, C, D, G, E, F, I, L, K, J

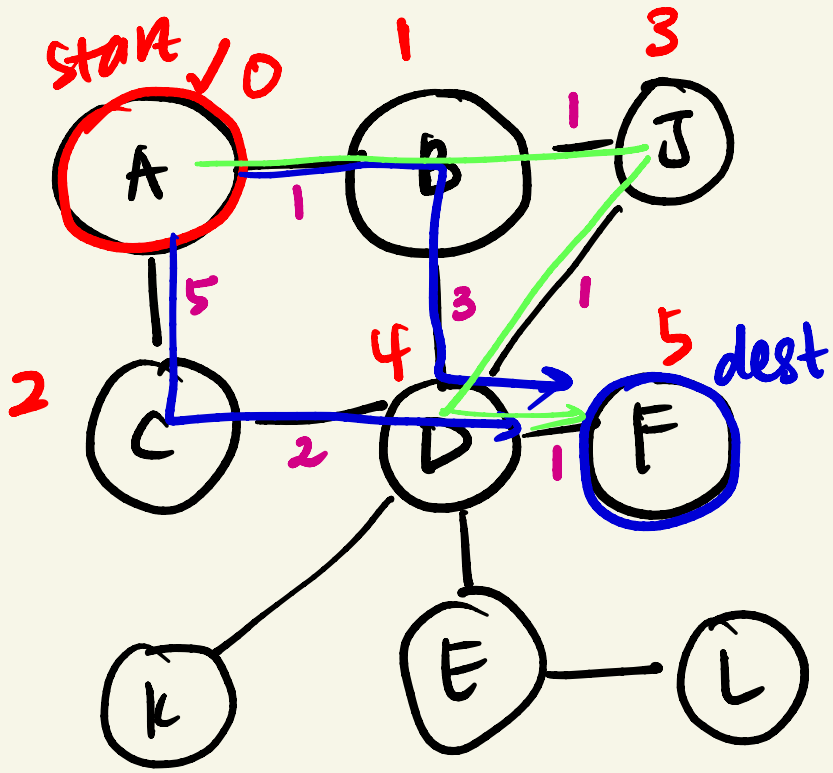
* not unique. It depends on which neighbour to explore first.



shortest # edges to traverse
↳ (3).

can be discovered by BFS if the weight (cost) of each edge is the same.

↓else

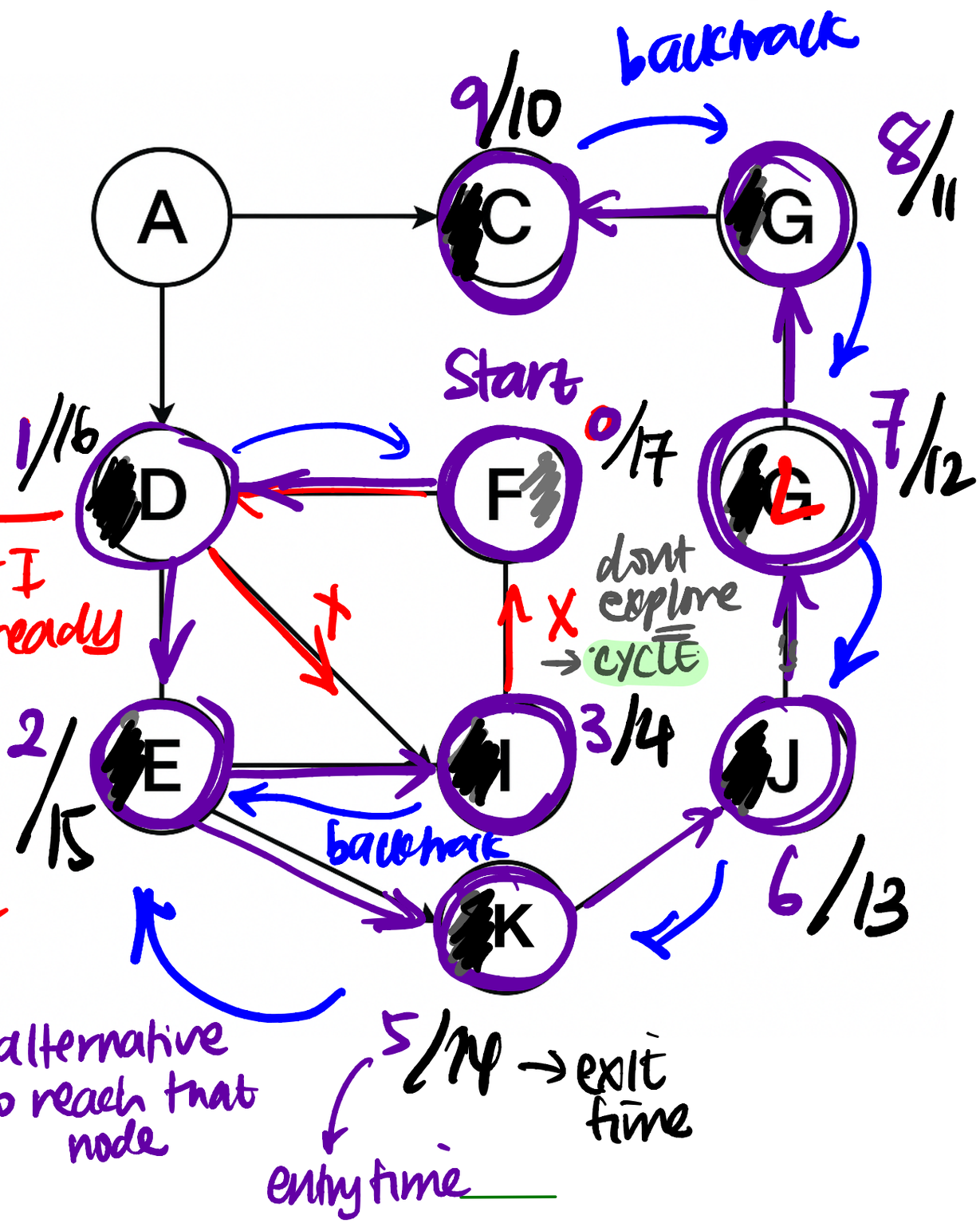


Depth-first Search

Backtrack : only done if all neighbours are already explored (black)

Given a starting node, how do we "walk" the graph?

- **Always** stack nodes to explore as deep as possible before **backtracking**
- **Useful data structure:** Stack
- **3-Colour concept:** white (new), grey (in-transit, visiting), black (visited, left for good)



Start : F → sequence of DFS is:

F, D, E, I, K, J, G, C } not unique, depends on which neighbour to enter first

(by entry time)

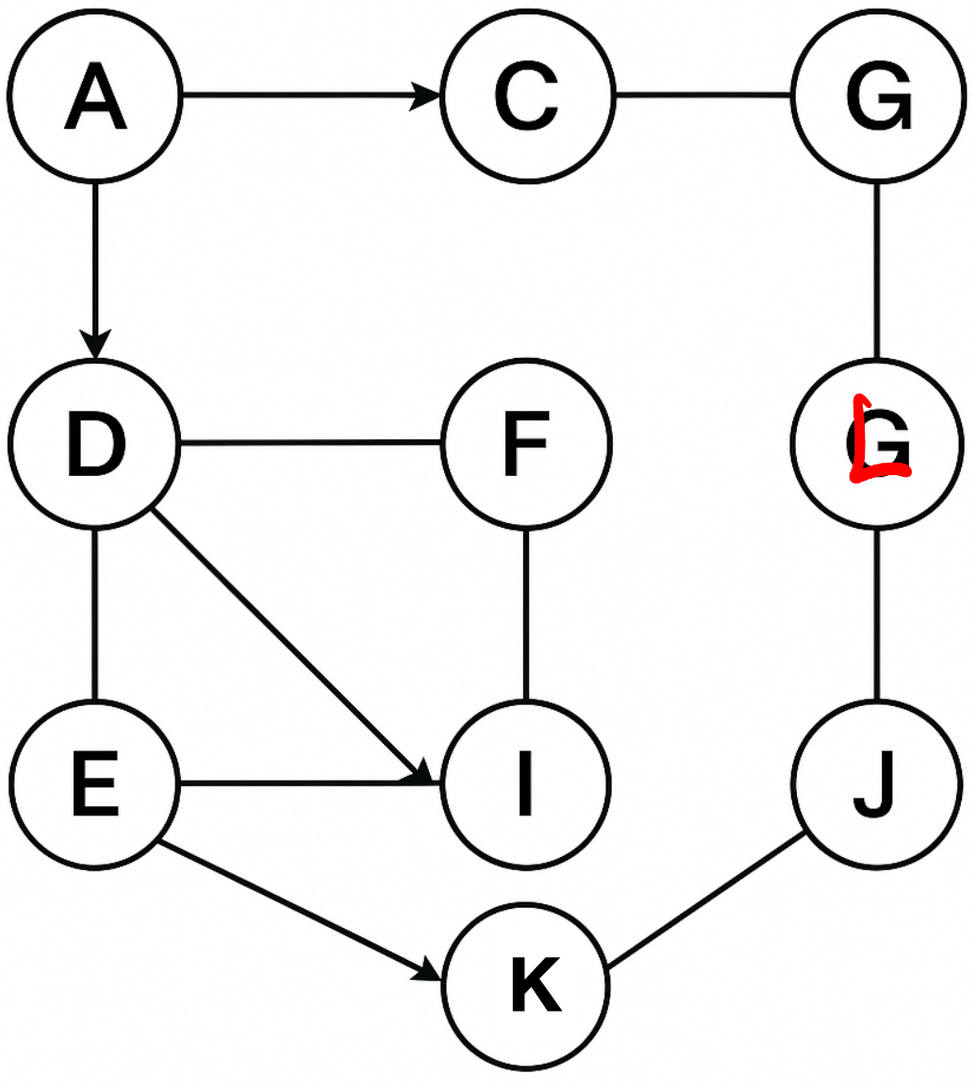
it's an alternative path to reach that node

Depth-first Search

Cycle Detection

grey node

- Meeting an *already explored* node means a cycle is present

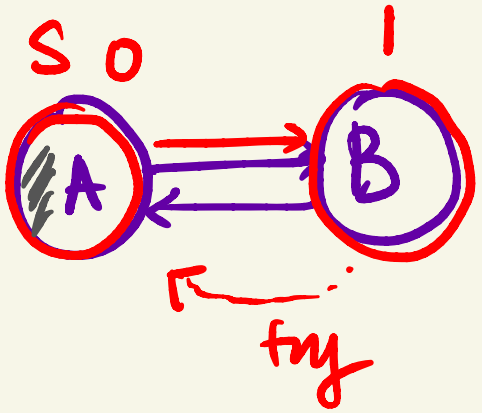


meeting a black node means that there's an alternative path to reach that node

⇓

* the graph is NOT a tree

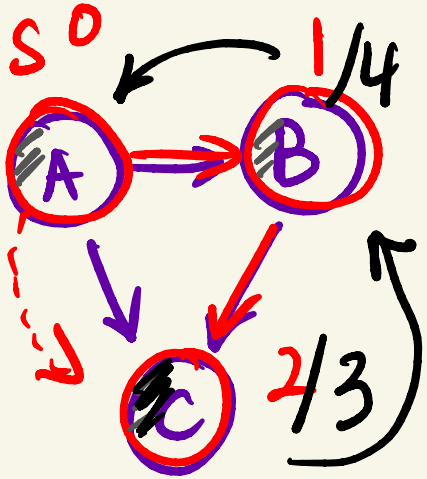
cycle detection



in B, we meet a grey neighbour (A)

↓

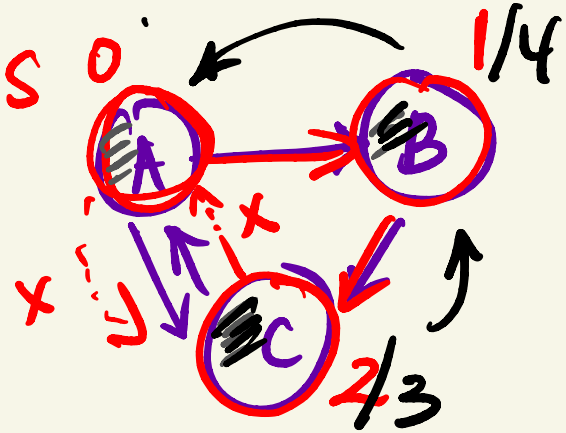
cycle



In A, we meet a black neighbour (C)

↓

but its not a cycle



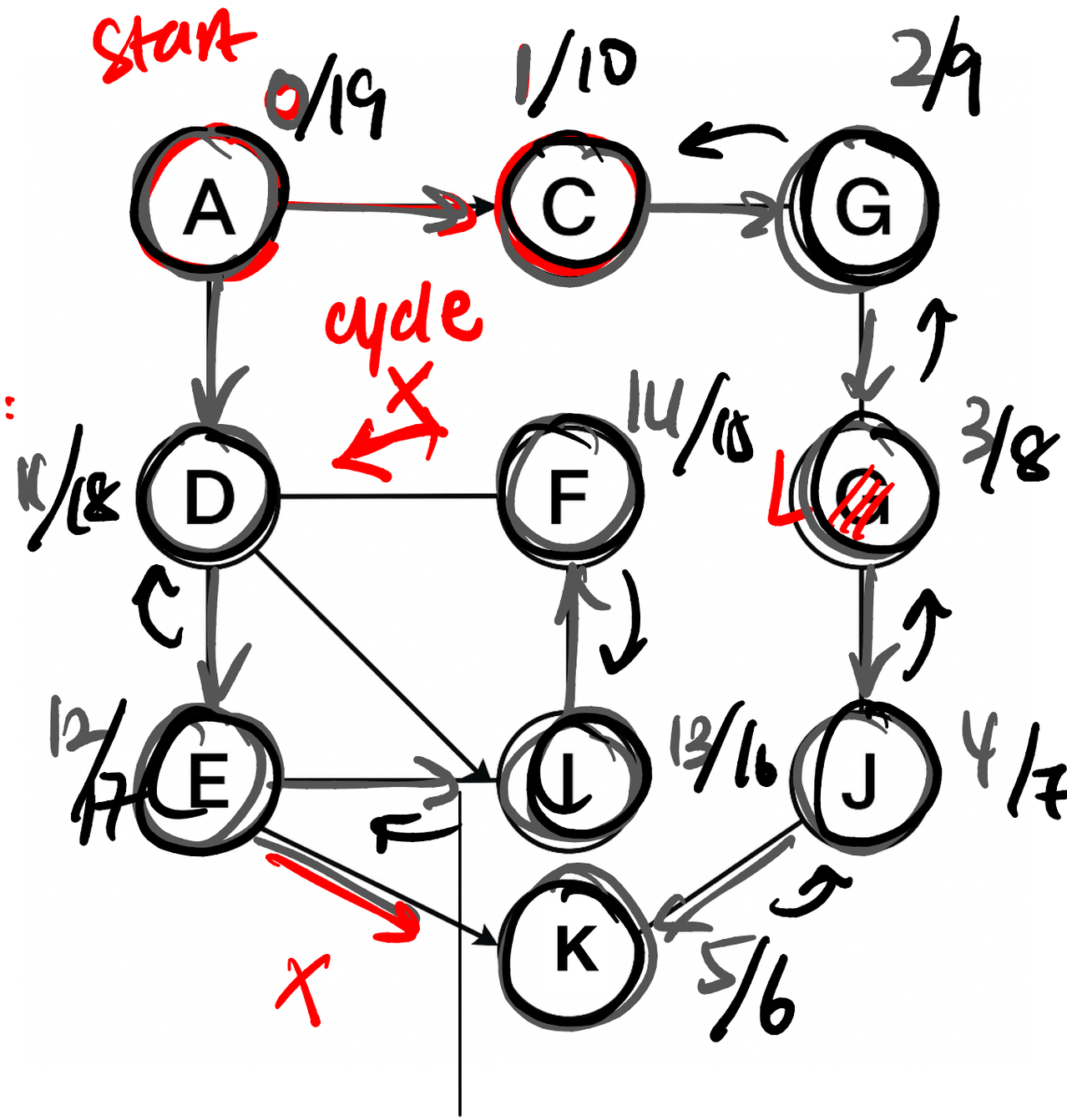
cycle as C meets grey A ✓ cycle

A meets a black node C ? doesn't mean its part of the cycle.

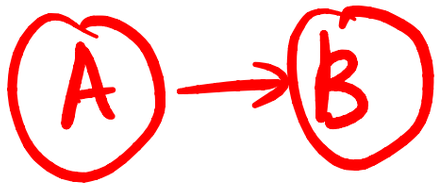
Depth-first Search

Topological Sort

- Do DFS, then get order by **finishing** time from largest to smallest



some graph describes dependency:

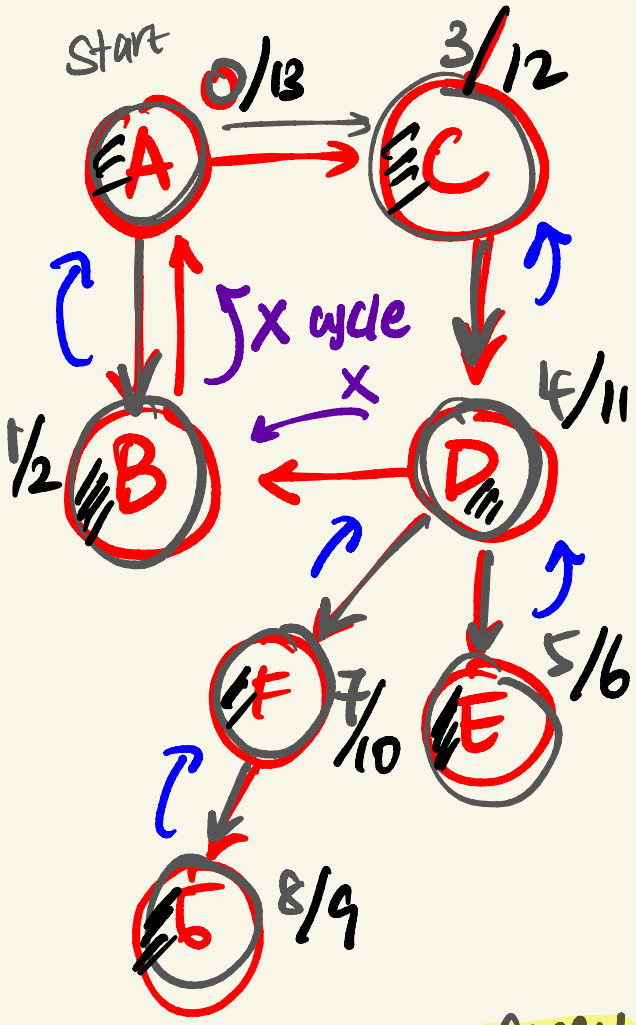


do A first, then B

A, D, E, I, F, C, G, L, J, K

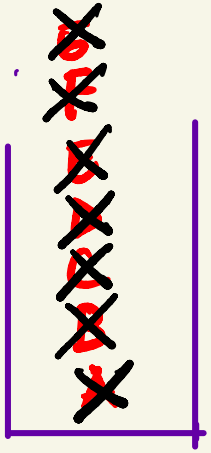
19 18 17 16 15 10 9 8 7 6

eg: F needs D & I to be done first ✓



note: explore neighbours alphabetically

stack



peek

⓪ put start node to stack,

① start time: the time we put the node into the stack

Explore neighbours, put white one into the stack,

② if no more new (white) neighbour, we pop the stack: exit time

③ stop when stack is empty

grey : when put to stack

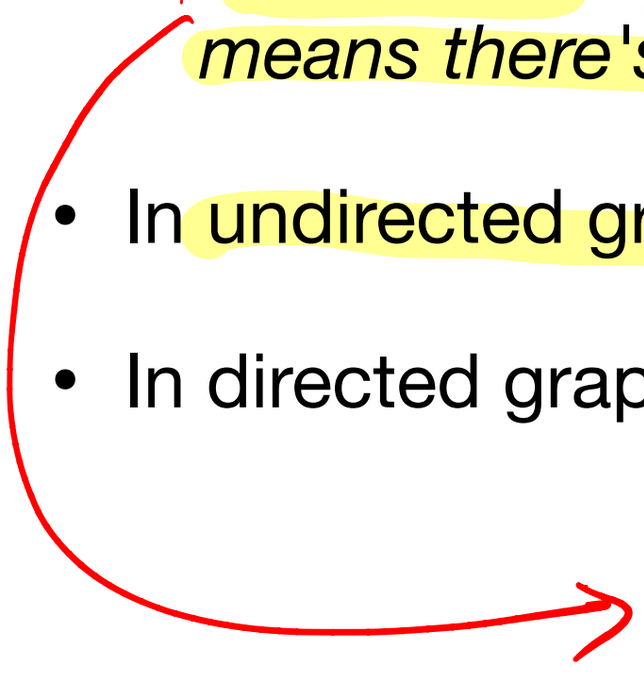
black : when popped from stack

Can BFS detect Cycles?

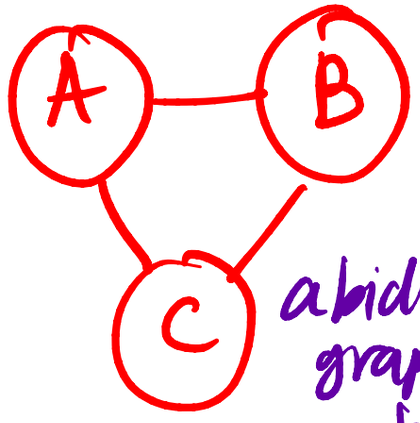
DFS can detect cycles, what about BFS?

→ no grey node

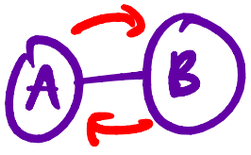
- BFS has no "in-visit" state, it's either *not visited* or *visited*
  - *What if we say: if I saw a vertex that's already visited, that means there's a cycle!* not true → may or may not be a cycle.
- In undirected graphs (**bidirectional**)?
- In directed graphs (**unidirectional**)?



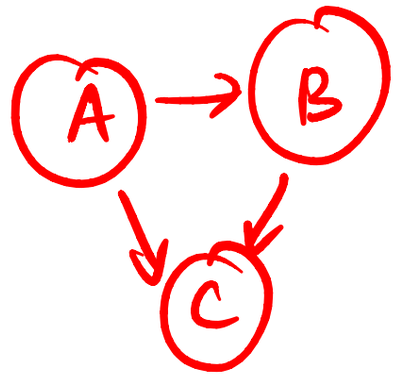
yes for this special case



a bidirectional graph always have a cycle.



> 1 path to reach a node



Graph Traversal

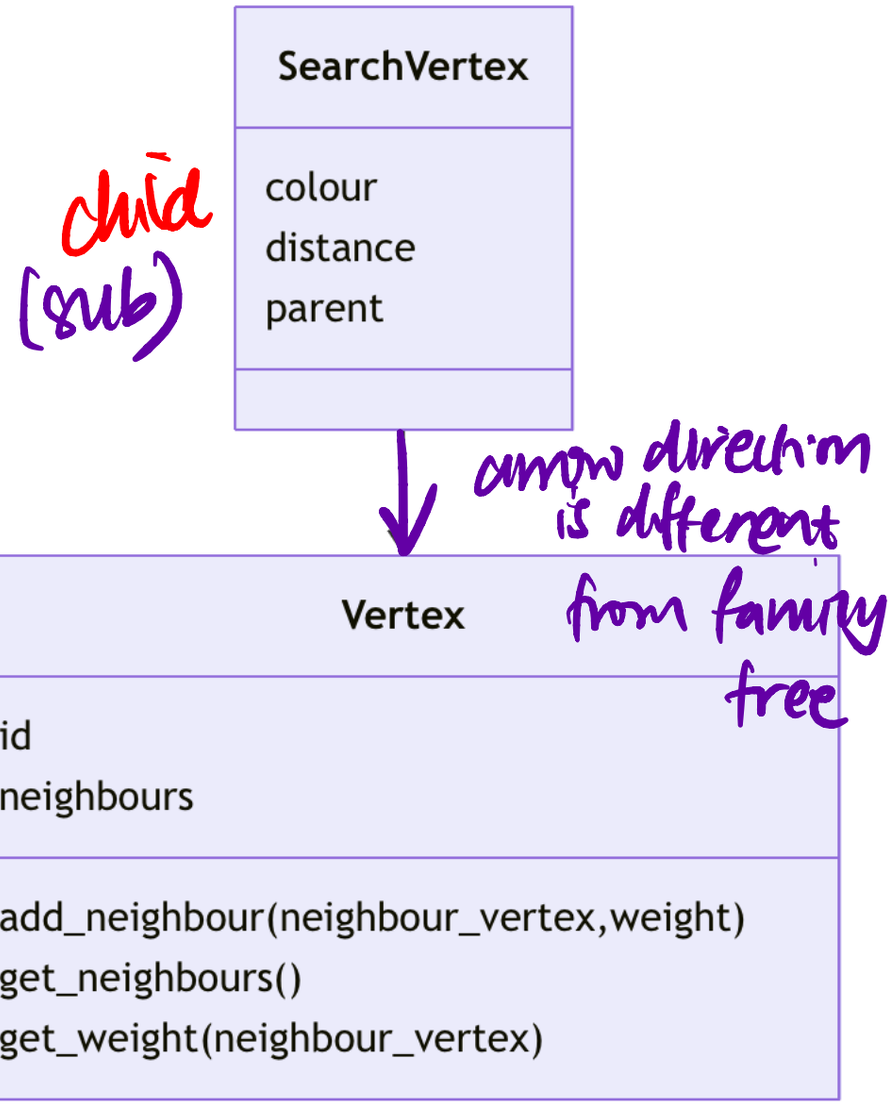
Applications

- **Breadth-first** search
  - Shortest path in public-transport system (edges of **equal** weights)
  - Web crawling: discover all reachable pages from a starting page (level-by-level)
- **Depth-first** search
  - Finding file in nested folders
  - Solving Sudoku or puzzles
  - Detecting **cycles** in dependencies
  - Course prereq solution (**topological sort**)

Inheritance

SearchVertex and Vertex

- Inheritance allows us to **create** a **new** class **without duplicating** all the other parts that is the same as classes we already have
- This is an **is-a** relationship
  - SearchVertex is-a Vertex



Learning Objectives

- Define graph, vertices, edges and weights.
- Use **Dictionary** and **OOP** to represent graph.
    - Represent graphs using **adjacency-list** representation or adjacency-matrix representation.
    - Differentiate **directed** and **undirected** graphs.
- Define paths.
- Create a **Vertex** class and a **Graph** class.
- **Extend** class Vertex and Graph for **graph traversal algorithm**
- Explain and implement **breadth first search**
- Explain and implement **depth first search.**