CS2040 – Data Structures and Algorithms

Lecture 13 – Graph Traversal

axgopala@comp.nus.edu.sg

NUS National University of Singapore

School of Computing

Outline

- Two algorithms to traverse a graph
- Breadth First Search (BFS) and Depth First Search (DFS)
- And some of their interesting applications ☺

https://visualgo.net/en/dfsbfs

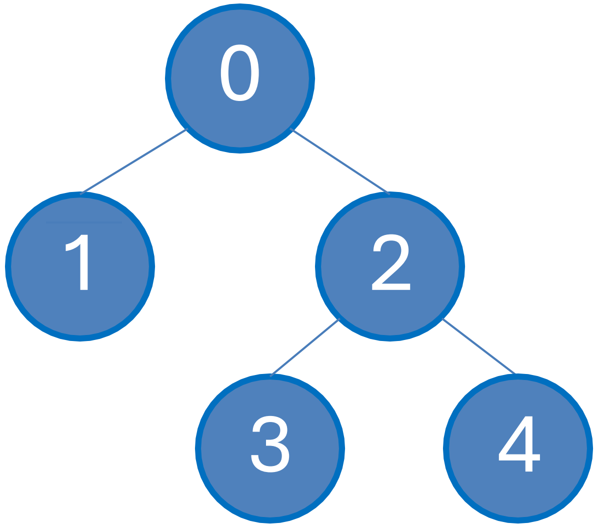
- Reference: Mostly from CP4 Section 4.2

Review: Binary Tree Traversal

- In a binary tree, there are three standard traversals:
  - Preorder
  - Inorder
  - Postorder

`pre(u)`
`  visit(u);`
`  pre(u->left);`
`  pre(u->right);`
	`in(u)`
	` in(u->left);`
	` visit(u);`
	` in(u->right);`
		`post(u)`
		` post(u->left);`
		` post(u->right);`
		` visit(u);`

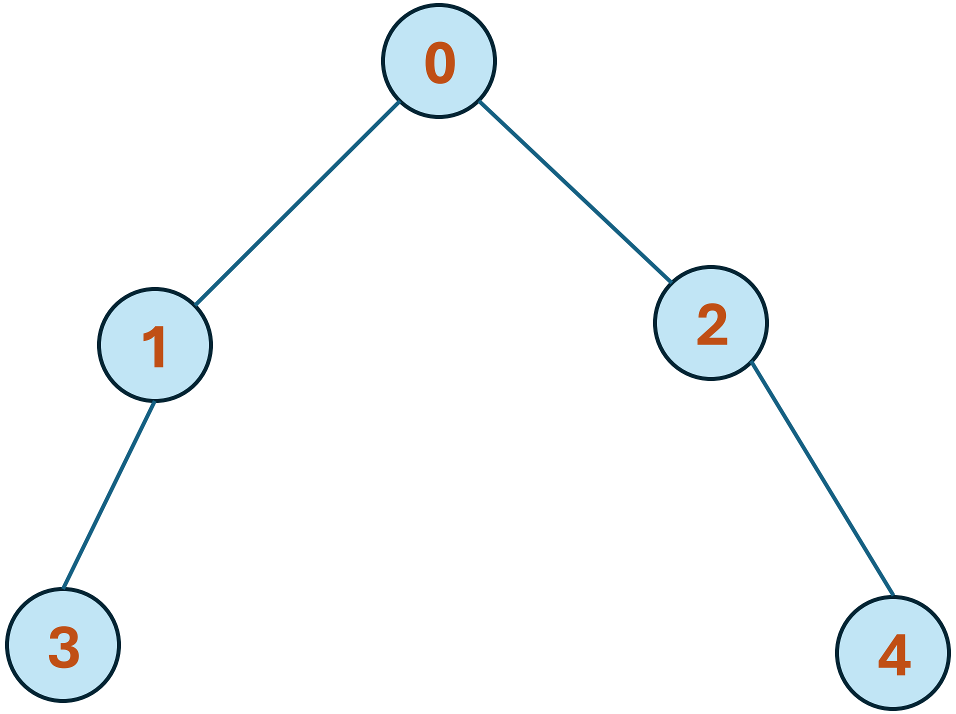
- We start binary tree traversal from root:
  - pre(root)/in(root)/post(root)
    - pre: 0 – 1 – 2 – 3 – 4
    - in: 1 – 0 – 3 – 2 – 4
    - post: 1 – 3 – 4 – 2 – 0



Live Quiz

- What is the result of **post(0)**?
  1. 0 – 1 – 2 – 3 – 4
  2. 0 – 1 – 3 – 2 – 4
  3. 3 – 4 – 1 – 2 – 0
  4. **3 – 1 – 4 – 2 – 0**

```
post(u)
 post(u->left);
 post(u->right);
 visit(u);
```



Traversing a Graph

- Two ingredients are needed for a **traversal:**

1. Where do we start? (The start)

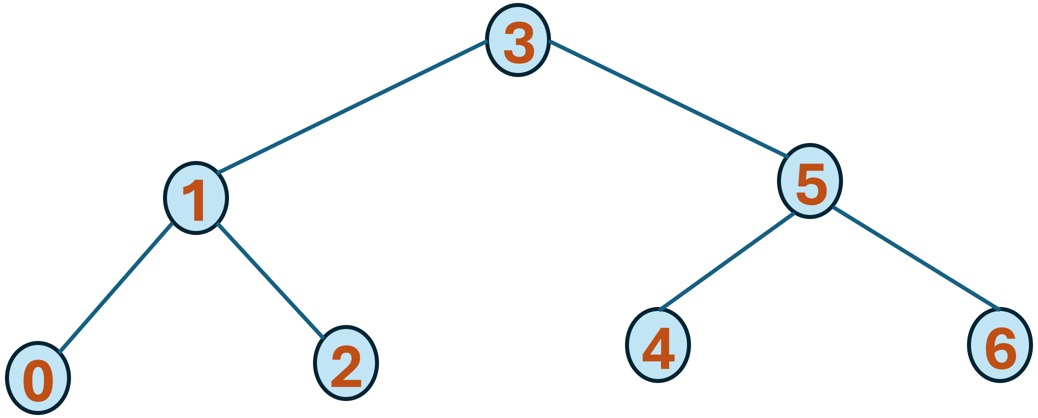
2. Which nodes are next? (The movement)

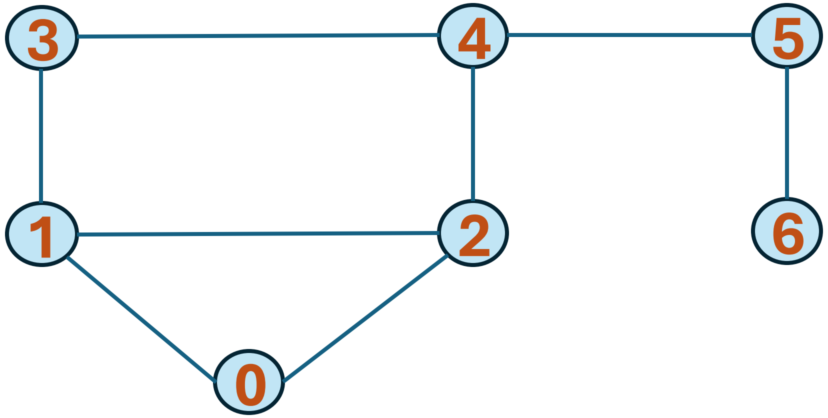
Traversing a Graph – Source

- In a tree, we *normally* start from root
    - Note: Not all trees are rooted though – we have to select one vertex as the "source"
- In a general graph, we do not have the notion of root
    - Instead, we start from a distinguished vertex – called **"source"** (**s**)

Traversing a Graph – Movement

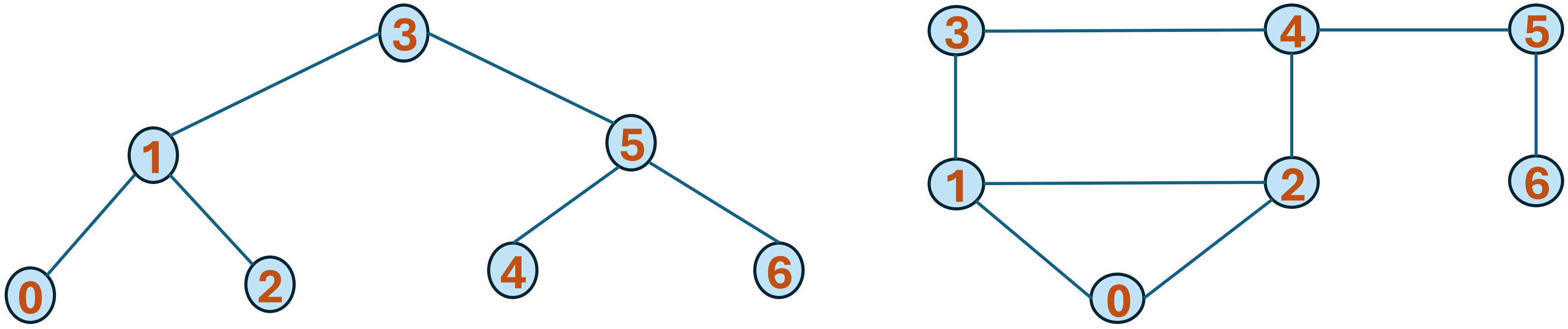
- In a (binary) tree, we only have (at most) two choices:
  - Go to the **left subtree** or to the **right subtree**
- In general graph, we can have more choices:
  - If **vertex u** and **vertex v** are adjacent/connected with edge (**u**, **v**); and we are now in **vertex u**; then we can also go to **vertex v** by traversing that edge (**u**, **v**)





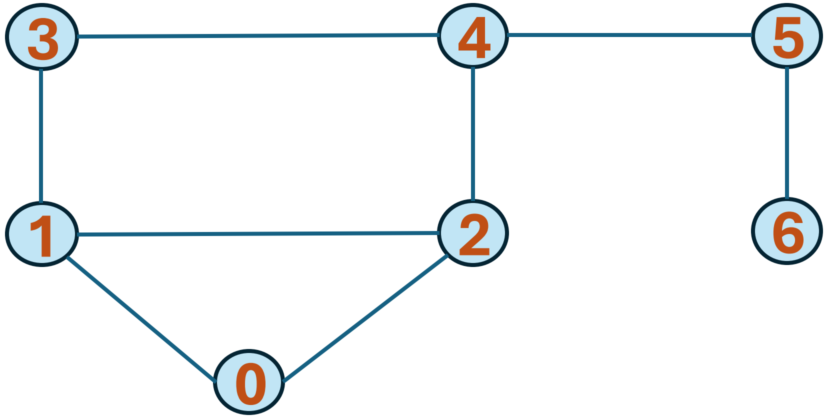
Traversing a Graph – Cycle

- In (binary) tree, there is **no cycle**
- In general graph, we **may have (trivial/non-trivial) cycles**
  - We need a way to avoid revisiting 0 ➔ 1 ➔ 2 ➔ 0 ➔ 1 … indefinitely



Traversing a Graph – Algorithms ☺

- **Breadth** First Search (BFS)
  - What's the heuristic?
  - Visit nodes in a **breadth first** manner (0 → 1 → 2 → 3 → 4 → 5 → 6)



- **Depth** First Search (DFS)
  - What's the heuristic?
  - Visit nodes in a **depth first** manner (0 → 1 → 2 → 4 → 3 → 5 → 6)

Idea: If a vertex **v** is reachable from **s**, then all neighbors of **v** will also be reachable from **s** (recursive definition)

BFS and DFS – Main questions

Q: How to maintain the order?

Q: How to differentiate visited vs unvisited vertices (to avoid cycle)?

Q: How to memorize the path?

BFS

Q: How to maintain the order?

- Use queue **Q** – initially, it contains only **s**

Q: How to differentiate visited vs unvisited vertices (to avoid cycle)?

- 1D array/Vector **visited** of size V – **visited[v] = 0** initially, and **visited[v] = 1** when **v** is visited

Q: How to memorize the path?

- 1D array/Vector **p** of size V – p**[v]** denotes the **p**redecessor (or **p**arent) of **v**

Edges used by BFS in the traversal will form a BFS “spanning” tree (tree that includes all vertices) of G that is stored in **p**

BFS – Pseudo Code

```
for all v in V
  visited[v] ← 0
  p[v] ← -1
Q ← {s} // start from s
visited[s] ← 1

while Q is not empty
  u ← Q.dequeue()
  for all v adjacent to u // order of neighbour
    if visited[v] = 0    // influences BFS
      visited[v] ← 1    // visitation sequence
      p[v] ← u
      Q.enqueue(v)
```

Initialization phase

Main loop

BFS – Analysis

```
for all v in V
  visited[v] ← 0
  p[v] ← -1
Q ← {s} // start from s
visited[s] ← 1

while Q is not empty
  u ← Q.dequeue()
  for all v adjacent to u // order of neighbour
    if visited[v] = 0    // influences BFS
      visited[v] ← 1    // visitation sequence
      p[v] ← u
      Q.enqueue(v)
```

Time Complexity: O(**V**+**E**)
- Initialization is O(**V**)
- For the while loop
  - Case 1: disconnected graph E = 0, takes O(**E**)
  - Case 2: connected graph
    - Each vertex is in the queue once (visited will be flagged to avoid cycle)
    - When a vertex is dequeued, all its neighbors are scanned (for loop); when queue is empty, all **E** edges are examined ~ O(**E**) ➔ if we use **Adjacency List**!
- Overall: O(**V**+**E**)

DFS

Q: How to maintain the order?

- Use stack **S** – can implicitly use one through recursion

Q: How to differentiate visited vs unvisited vertices (to avoid cycle)?

- 1D array/Vector **visited** of size V – **visited[v] = 0** initially, and **visited[v] = 1** when **v** is visited

Q: How to memorize the path?

- 1D array/Vector **p** of size V – p**[v]** denotes the **p**redecessor (or **p**arent) of **v**

Edges used by DFS in the traversal will form a DFS "spanning" tree (tree that includes all vertices) of G that is stored in **p**

DFS – Pseudo Code

```
DFSrec(u)
  visited[u] ← 1 // to avoid cycle
  for all v adjacent to u // order of neighbour
    if visited[v] = 0 // influences DFS
      p[v] ← u // visitation sequence
      DFSrec(v) // recursive (implicit stack)

// in the main method
for all v in V
  visited[v] ← 0
  p[v] ← -1
DFSrec(s) // start the recursive call from s
```

Initialization phase, same as with BFS

DFS – Analysis

```
DFSrec(u)
  visited[u] ← 1
  for all v adjacent to u
    if visited[v] = 0
      p[v] ← u
      DFSrec(v)

// in the main method
for all v in V
  visited[v] ← 0
  p[v] ← -1
DFSrec(s)
```

Time Complexity: O(**V**+**E**)

- Initialization is O(**V**)
- For the recursion:
  - Case 1: disconnected graph, E = 0, takes O(**E**)
  - Case 2: connected graph,
    - Each vertex is visited (i.e. call DFSrec on it) once (visited flagged to avoid cycle)
    - When a vertex is visited, all its neighbors are scanned (for loop); after all vertices are visited, we have examined all **E** edges ~ O(**E**) → if we use **Adjacency List**!
- Overall: O(**V**+**E**)

Path Reconstruction – Iterative Version

```
// will produce reversed output
Output "(Reversed) Path:"
i ←t // start from end of path: suppose vertex t
while i != s
  Output i
  i ←p[i] // go back to predecessor of i
Output s

         // try it on this array p, t = 4
         // p = {-1, 0, 1, 2, 3, -1, -1, -1}
```

Path Reconstruction – Recursive Version

```
void backtrack(u)
  if (u == -1) // recall: predecessor of s is -1
    stop
  backtrack(p[u]) // go back to predecessor of u
  Output u // recursion like this reverses the order

// in main method
// recursive version (normal path)
Output "Path:"
backtrack(t); // start from end of path (vertex t)
        // try it on this array p, t = 4
        // p = {-1, 0, 1, 2, 3, -1, -1, -1}
```

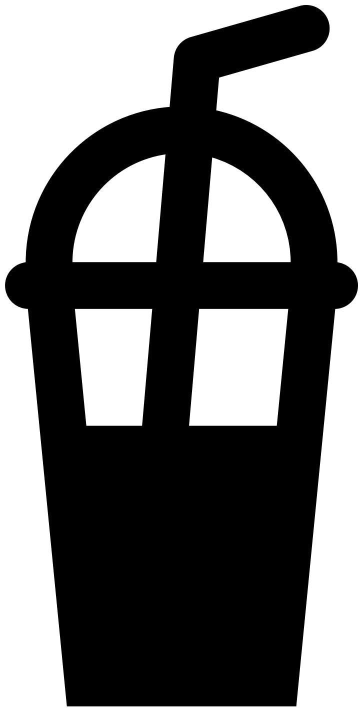
Graph Traversal Applications

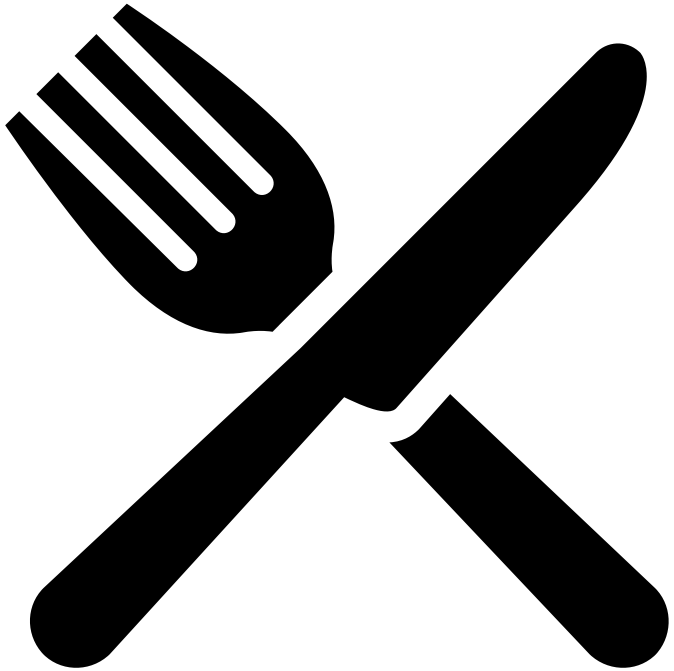
What can we do with BFS and DFS?

Some Applications of BFS and DFS ☺

1. Reachability Test
2. Find Shortest Path (multiple lectures dedicated to it 😎)
3. Identifying/Counting Component(s)
4. Topological Sort
5. Identifying/Counting Strongly Connected Component(s)

Take a Break



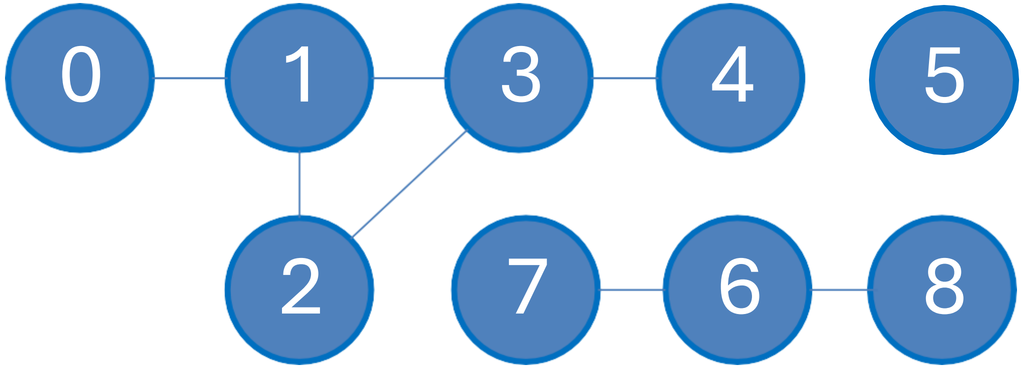


1. Reachability Test

- Check whether vertex **u** can reach vertex **v**

- Idea: Start BFS/DFS from **u** and after it terminates, check if **visited[v] = 1**

```
BFS(u) // DFSrec(u)
if visited[v] == 1
  Output "Yes"
else
  Output "No"
```



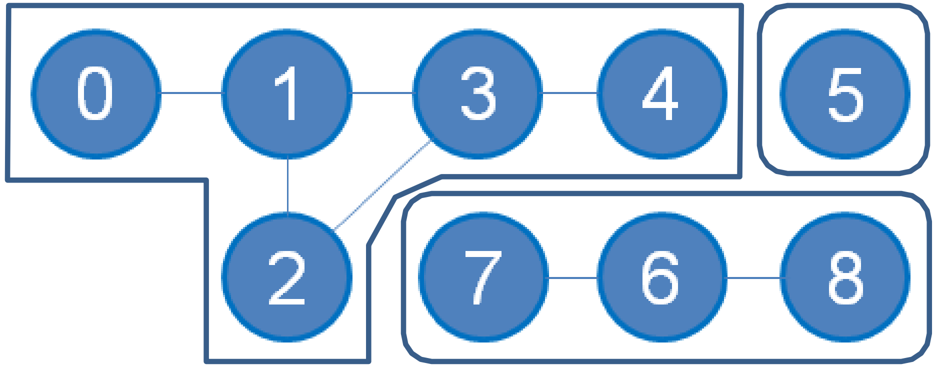
2. Finding Shortest Path

- For now, just look at unweighted graphs ⇔ edges have no weight
- Shortest path between any 2 vertices u, v ⇔ least number of edges traversed from u to v
- Algorithm?
  - BFS ☺ (Complexity = O(**V**+**E**))

3. Identifying/Counting Component(s)

- Component → A maximal group of vertices in an undirected graph that can visit each other via some path
- Use BFS/DFS to identify components by labeling/counting them

```
CC ← 0
for all v in V
  visited[v] ← 0
for all v in V // O(V)?
  if visited[v] == 0
    CC ← CC + 1
    DFSrec(v) //O(V+E)?
// BFS from v is also OK
```



Live Quiz

- What is the time complexity of identifying/counting component(s)?

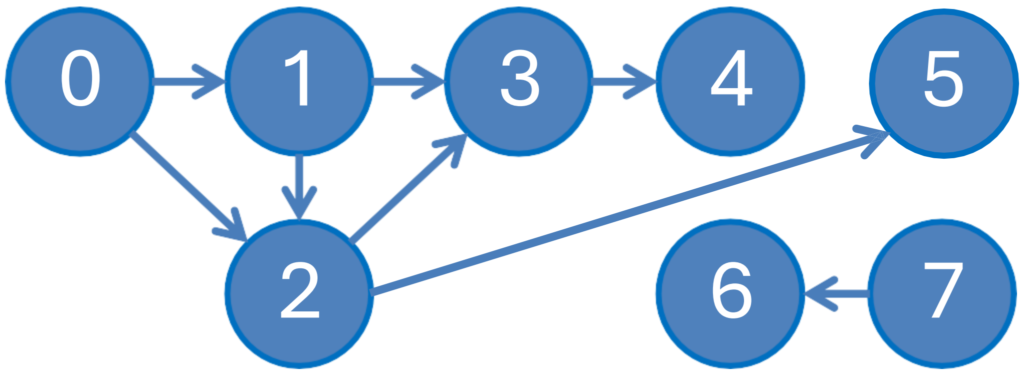
1. **Maybe O(V+E)**

2. Something else

3. Maybe O(**V***(**V**+**E**)) = O(**V^2** + **VE**)

4. Topological Sort

- Topological sort of a DAG is a linear ordering of its vertices in which each vertex comes before all vertices to which it has outbound edges



0 1 2 3 4 5 7 6

0 1 2 3 7 4 5 6

7 6 0 1 2 5 3 4

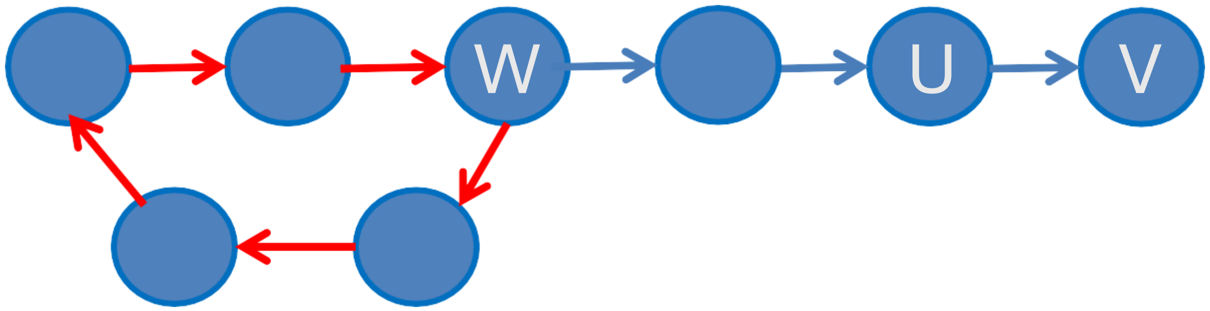
4. Topological Sort

- Every DAG has one *or more* topological sorts

  - Proof is next! 😯

Always a Lemma! ☺

- Lemma: If G is a DAG, it has a node with no incoming edges
- Proof by contradiction:
  - Assume every node in G (G is a DAG) has an incoming edge
  - Pick a node **V** and follow one of its incoming edge backwards e.g. (**U**,**V**) which will visit **U**
  - Do the same thing with **U**, and keep repeating this process
  - Since every node has an incoming edge, at some point you will visit a node **W** 2 times. Stop at this point as you have a cycle (Contradiction!)



Another Lemma! (Well – the proof actually ☺)

- Lemma: If G is a DAG, then it has a topological ordering
- Constructive proof
  - Pick node V with no incoming edge (must exist according to previous lemma)
  - Remove V from G and number it 1
  - G-{V} must still be a DAG since removing V cannot create a cycle
  - Pick the next node with no incoming edge W and number it 2
  - Repeat the above with increasing numbering until G is empty
  - For any node it cannot have incoming edges from nodes with a higher numbering
  - Thus, ordering the nodes from lowest to highest number will result in a topological ordering

Basis for the BFS based algorithm (Kahn's algorithm) to compute topological ordering of a DAG

4. Topological Sort – Kahn's Algorithm

- If graph is a DAG, then run a modified version of BFS (Kahn's algorithm) on it → valid topological order

  - Replace **visited** array with an integer array **indeg** that keeps track of the in-degree of each vertex in the DAG

  - Use an ArrayList **toposort** to record the vertices

Kahn's Algorithm – Pseudo Code

```
for all v in V
  indeg[v] ← 0
  p[v] ← -1
for each edge (u,v) // get in-degree of vertices
  indeg[v] ← indeg[v] + 1
for all v' where indeg[v'] = 0
  Q ← {v'} // enqueue v'

while Q is not empty
  u ← Q.dequeue()
  append u to back of toposort
  for all v adjacent to u // order of neighbour
    indeg[v] ← indeg[v] - 1
    if indeg[v] = 0 // add to queue
      p[v] ← u
      Q.enqueue(v)
Output toposort as the topological order
```

$\left.\right\}$ Initialization phase

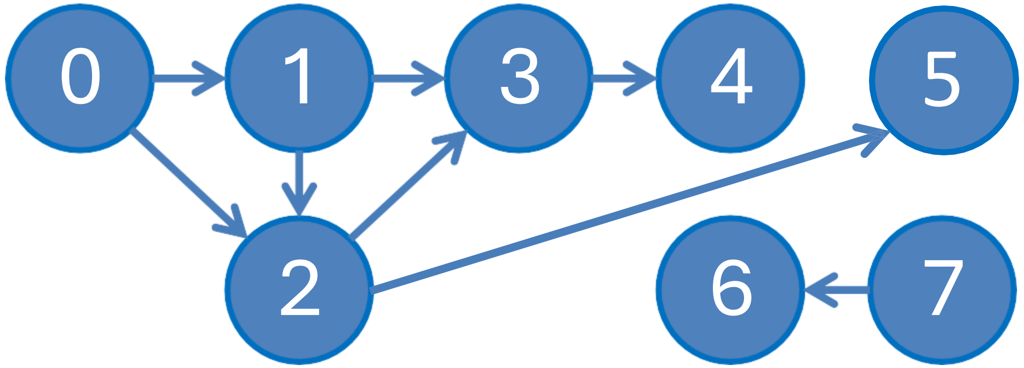
$\left.\right\}$ Main loop

4. Topological Sort – DFS version

- Running a slightly modified **DFS** on the DAG and recording the vertices in "post-order" manner → valid topological order
  - Use an ArrayList **toposort** to record the vertices
  - "Post-order" processing = process vertex **u** (i.e. put **u** in **toposort**) after all **neighbours** of **u** have been visited
  - After running the algorithm, all vertices reachable by any vertex v will be placed before v in **toposort**

4. Topological Sort – DFS version

- Suppose we have visited all neighbors of 0 recursively with DFS
- toposort = [[list of vertices reachable from 0], vertex 0]
  - Then, suppose we have visited all neighbors of 1 recursively with DFS
  - toposort = [[[list of vertices reachable from 1], vertex 1], vertex 0]
  - and so on…
- We will eventually have = [4, 3, 5, 2, 1, 0, 6, 7]
- Reversing it, we will have = [7, 6, 0, 1, 2, 5, 3, 4]



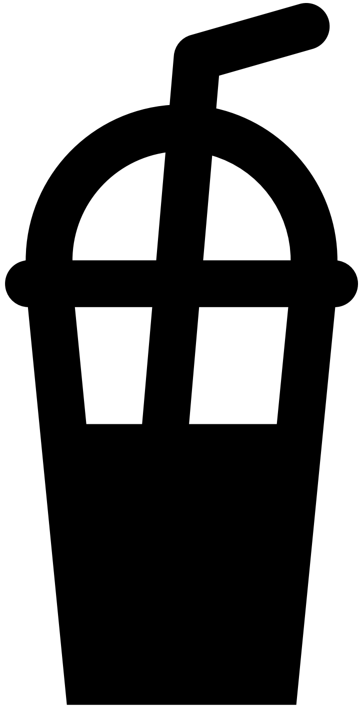
DFS version – Pseudo Code

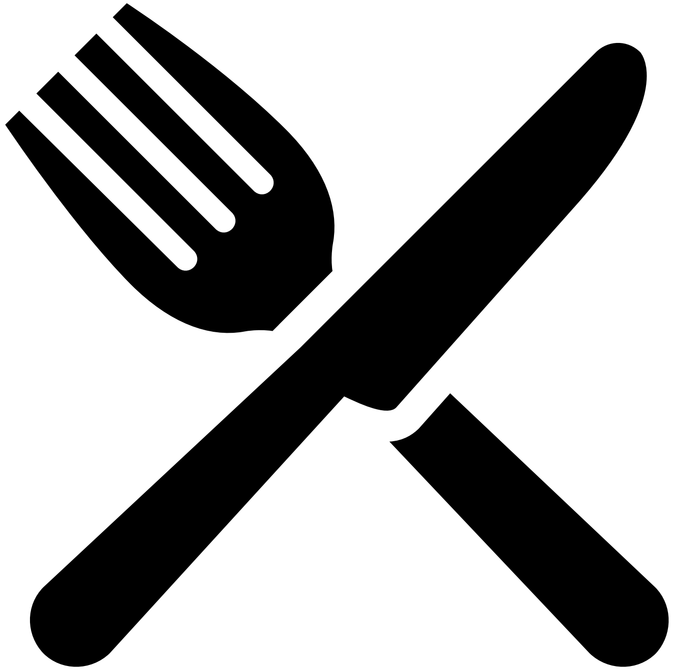
```
DFSrec(u)
  visited[u] ← 1 // to avoid cycle
  for all v adjacent to u // order of neighbor
    if visited[v] = 0 //    influences DFS
      p[v] ← u // visitation sequence
      DFSrec(v) // recursive (implicit stack)
  append u to the back of toposort // "post-order"

// in the main method
for all v in V
  visited[v] ← 0
  p[v] ← -1

for all v in V
  if visited[v] == 0
    DFSrec(v) // start the recursive call
reverse toposort and output it
```

Take a Break





5. Identifying/Counting Strongly Connected Component(s)

- A **strongly connected component** (SCC) ➔ A maximal group (subgraph) of vertices (>= 1) in a **directed graph** where every vertex is reachable from every other vertex

- A directed graph with 1 SCC is called a **strongly connected graph**

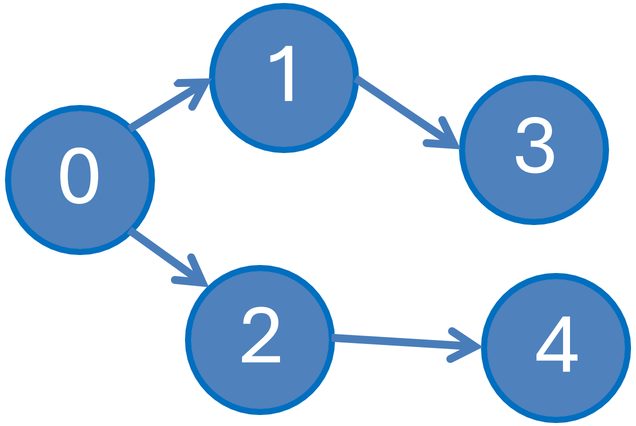
- Identifying SCCs is harder than identifying components due to the direction of the edges

- One algorithm to do this is Kosaraju's algorithm ⇔ uses DFS

Live Quiz

- How many SSCs does the graph have?

a) 0
b) 1
c) 2
d) 3
e) 4
f) 5

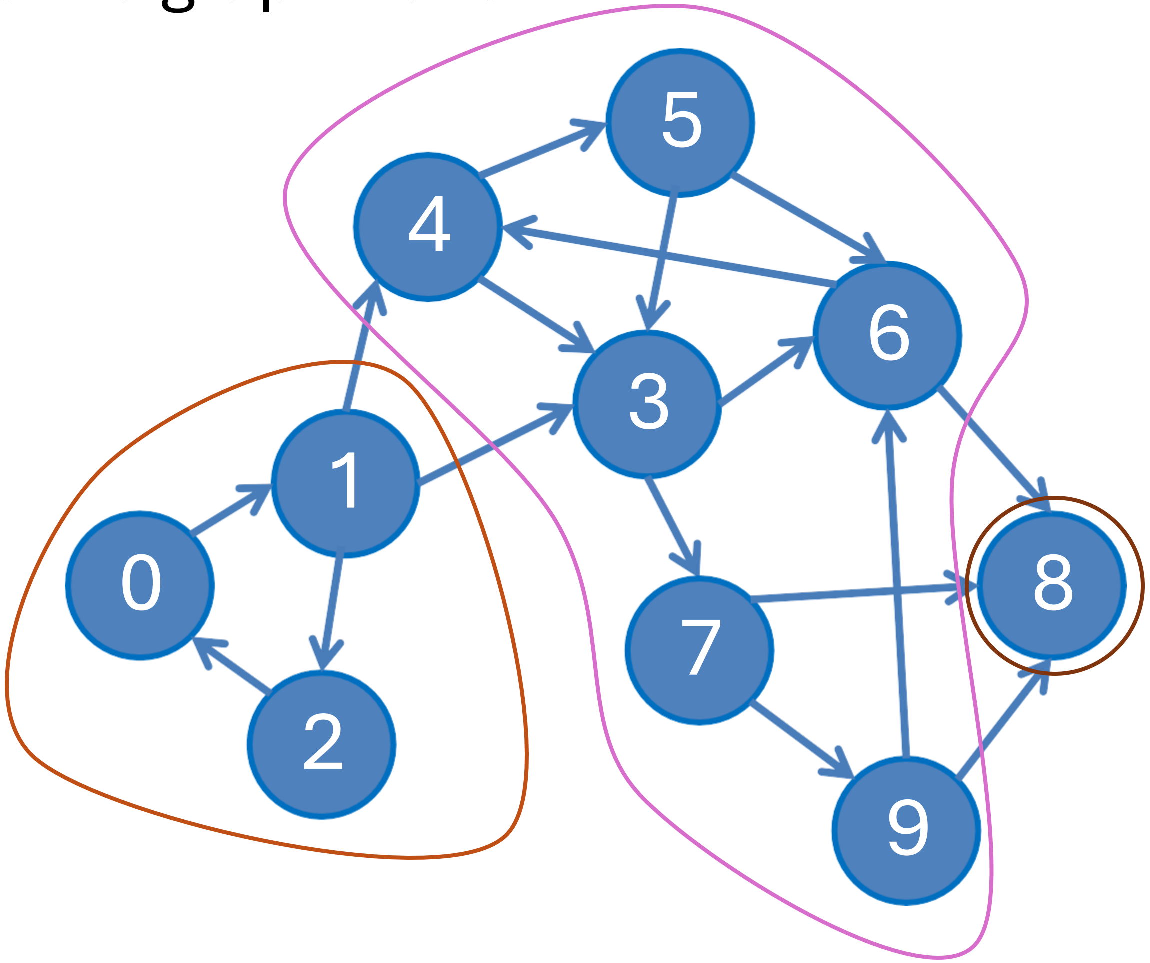


Each Vertex is a SCC

Live Quiz

- How many SSCs does the graph have?

a) 0
b) 1
c) 2
d) 3
e) 4
f) 5



5. Identifying SSCs – Kosaraju's Algorithm

1. Perform DFS **post-order** traversal on the given directed graph G and store the vertices into an array **K**

2. Create transpose G' from G (G' has direction of all edges reversed)
   - for each vertex v in adj. list of G and for each neighbour u of v, add edge u → v to G'

3. Perform counting strongly connected component algo on G', as follows

```
SCC ← 0
for all v in V
  visited[v] ← 0
for all v in K from last to first vertex
  if visited[v] == 0
    SCC ← SCC + 1
    DFSrec(v)
```

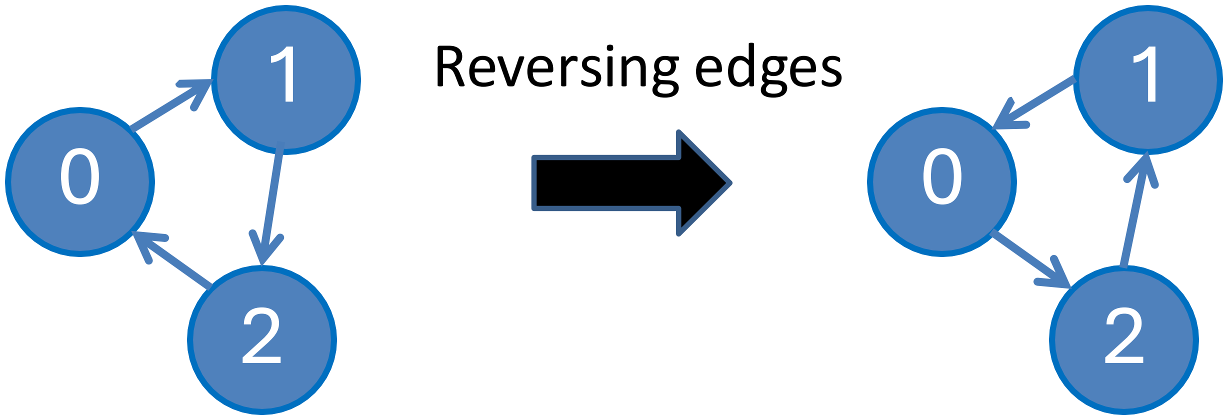
Time Complexity

Adjacency List O(**V+E**)

Adjacency Matrix?

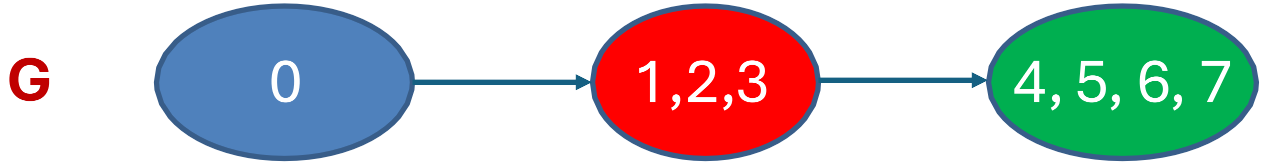
Why does Kosaraju's Algorithm work?

- Important property: given any SCC, reversing all the edges in the SCC will still result in the same SCC

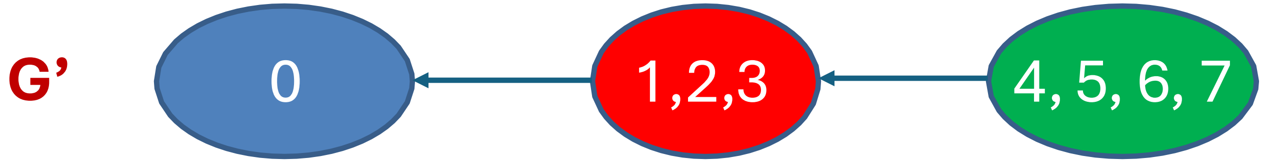


Why does Kosaraju's Algorithm work?

- If we have the following SCCs in a directed graph



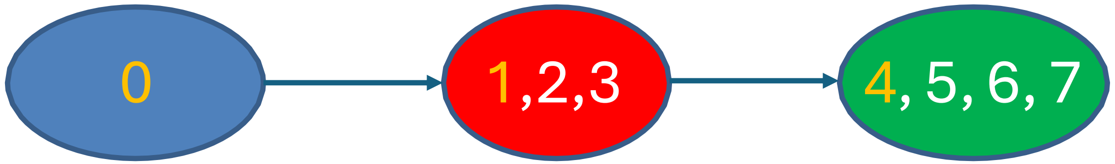
- If we flip the graph, we will still get the same SCCs but with the edges linking them flipped (if there are such edges)



Why does Kosaraju's Algorithm work?

- Now if we view each SCC in G or G' as a vertex, then G or G' is a DAG!
- Let v' be the 1^{st} vertex visited in each SCC when we perform DFS topological sort on G
  - For any SCC x, all reachable SCCs from x have their v' placed in **K** before the v' of x
  - Also, all vertices in same SCC as any v' must come before that v' in **K**

G



- Assuming the colored vertex is v' (the first one visited) in its respective SCC

{2,3} may be in these 2 segments

K = ... 4.... 1 0

{5,6,7} may be in this segment

Why does Kosaraju's Algorithm work?

- If we then perform counting SCC using **K** on the transpose graph **G'**

G'

Process **K** from back to front

K = … 4 … 1 … 0

- Essentially, we are visiting the SCCs in topological ordering of G
- The v' of each SCC must be 1st unvisited vertex encountered for that SCC, performing `DFSrec(v')`
  - Will only visit all vertices in the SCC of v'
  - Reversed edges will prevent us from visiting unvisited vertices in other SCC

Summary

- Graph Traversal Algorithms: Start+Movement
  - Breadth-First Search: uses queue, breadth-first
  - Depth-First Search: uses stack/recursion, depth-first
  - Both BFS/DFS uses “flag” technique to avoid cycling
  - Both BFS/DFS generates BFS/DFS “Spanning Tree”
  - Some applications: Reachability, Shortest Path in unweighted graph, Counting Components, Topological sort, Counting SCCs

Live Quiz

- What is the time complexity of BFS/DFS?

   A) O (V + E)

   B) O (V * E)

   C) O (V)

   D) None of the above

Live Quiz

- Which algorithm do you use for finding/counting connected components?

  A) Kosaraju's Algorithm

  B) Kahn's Algorithm

  C) Shortest Path Algorithm

  D) BFS/DFS

  E) Reachability Test

Live Quiz

- Which algorithm do you use for Topological Sort?

  A) Kosaraju's Algorithm

  B) Kahn's Algorithm

  C) Shortest Path Algorithm

  D) Reachability Test

Live Quiz

- Which algorithm do you use for finding/counting strongly connected components?

    A) **Kosaraju’s Algorithm**

    B) Kahn’s Algorithm

    C) Shortest Path Algorithm

    D) Reachability Test

Next Week

- Minimum Spanning Tree

Continuous Feedback

https://forms.office.com/r/KsNwmTUD0q