CS2040S – Data Structures and Algorithms

Lecture 16 – Graph Traversal

chongket@comp.nus.edu.sg

NUS National University of Singapore

School of Computing

Outline

Two algorithms to traverse a graph

- Depth First Search (DFS) and Breadth First Search (BFS)
- Plus some of their interesting applications

https://visualgo.net/en/dfsbfs

Reference: Mostly from CP4 Section 4.2

GRAPH TRAVERSAL ALGORITHMS

Review – **Binary Tree** Traversal

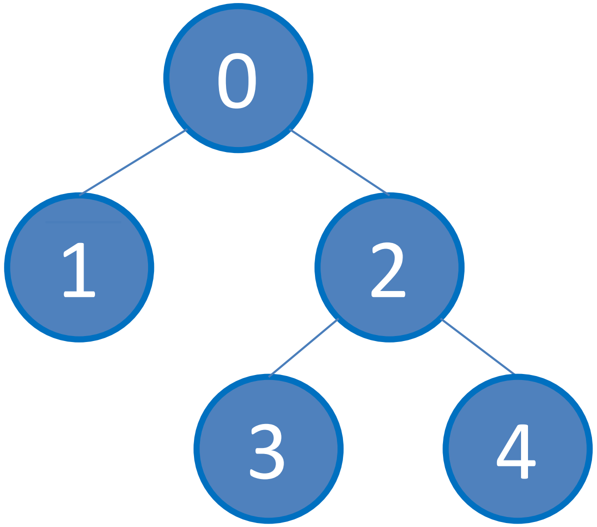
In a binary tree, there are three standard traversal:

- 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);`

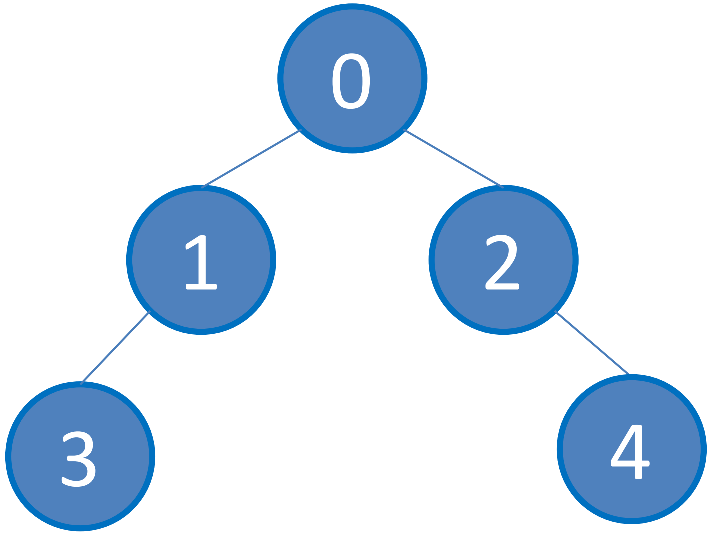
We start binary tree traversal from root:

- pre(root)/in(root)/post(root)
  - pre = 0, 1, 2, 3, 4
  - in = 1, 0, 3, 2, 4
  - post = 1, 3, 4, 2, 0



What is the **Post**Order Traversal of this Binary Tree?

1. 0 1 2 3 4
2. 0 1 3 2 4
3. 3 4 1 2 0
4. 3 1 4 2 0



Traversing a Graph (1)

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

1. The start
2. The movement

Defining the start ("source")

- In tree, we *normally* start from root
  - Note: Not all tree are rooted though!
    - In that case, we have to select one vertex as the "source", see below
- In general graph, we do not have the notion of root
  - Instead, we start from a distinguished vertex
    - We call this vertex as the **"source" s**

Traversing a Graph (2)

Defining the movement:

- In (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**)
- In (binary) tree, there is **no cycle**
- In general graph, we **may have (trivial/non trivial) cycles**
  - We need a way to avoid revisiting **u** → **v** → **w** → **u** → **v** … indefinitely

Traversing a Graph (3)

Solution: BFS and DFS ☺

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

(recursive definition)

Breadth First Search (BFS) – Ideas

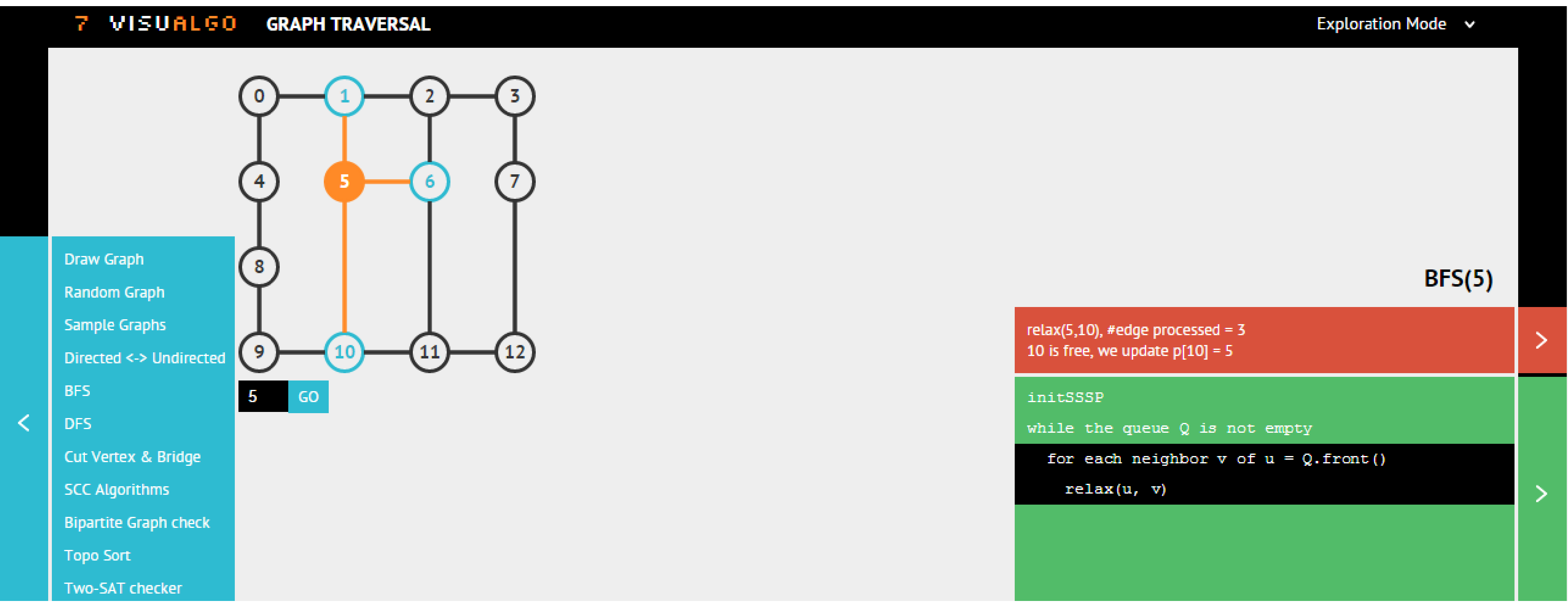
data structures needed by bfs
1. Queue to q in vertices that have been visited so that we can visit their verticies
2. visited array: keep track of visited and visited vertices so that we only visit each vertex once
3. predecessor array to keep track of parent and memorise the path

- Start from **s**
- BFS visits vertices of G in *breadth-first* manner (when viewed from source vertex s)
  - Q: How to maintain such order?
    - A: Use queue **Q**, initially, it contains only **s**
  - Q: How to differentiate visited vs unvisited vertices (to avoid cycle)?
    - A: 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?
    - A: 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 of G (tree that includes all vertices of G) stored in **p**

Graph Traversal: BFS(s)

Ask VisuAlgo to perform various Breadth-First Search operations on the sample Graph

In the screen shot below, we show the start of **BFS(5)**



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 neighbor
    if visited[v] = 0 // influences BFS
      visited[v] ← true // visitation sequence
      p[v] ← u
      Q.enqueue(v)

// after BFS stops, we can use info stored in visited/p
```

O(v)
Initialization phase
of the arrays being used

disconnected graph: 0(v) because each of the v vertices only runs O(1) because no neighbour
connected graph: num of edges = O(v) to O(V^2
WHILE LOOP: loops O(v) times because we only visit each vertex once, so it is only queued and dequeued once

INNER FOR LOOP: depends on which data structure we use

its adj matrix -> O(v)
so total time complexity = O(v) + O(v^2) = O(v^2)

if its adj list - O(2E)
so total time complexity = O(v) + O(2E) = O(v + E)

Main loop

go to row u, scan k neighbours of u
every vertex dequeue and look through all neighbours meaning
we will process edge at most twice

graph traversal most appropriate to use adj list

BFS Analysis

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**)

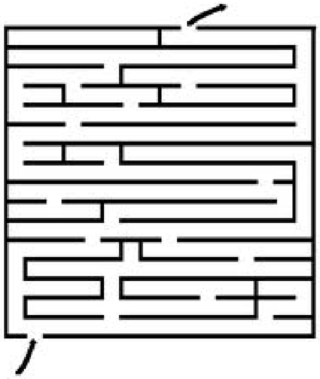
neighbours gotten from either adjacency list or adjacency matrix

```
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 neighbor
    if visited[v] = 0 //  influences BFS
      visited[v] ← true // visitation sequence
      p[v] ← u
      Q.enqueue(v)

// we can then use information stored in visited/p
```

Depth First Search (DFS) – Ideas

data structures needed by dfs

1. visited array: keep track of visited and visited vertices so that we only visit each vertex once
2. predecessor array to keep track of parent and memorise the path

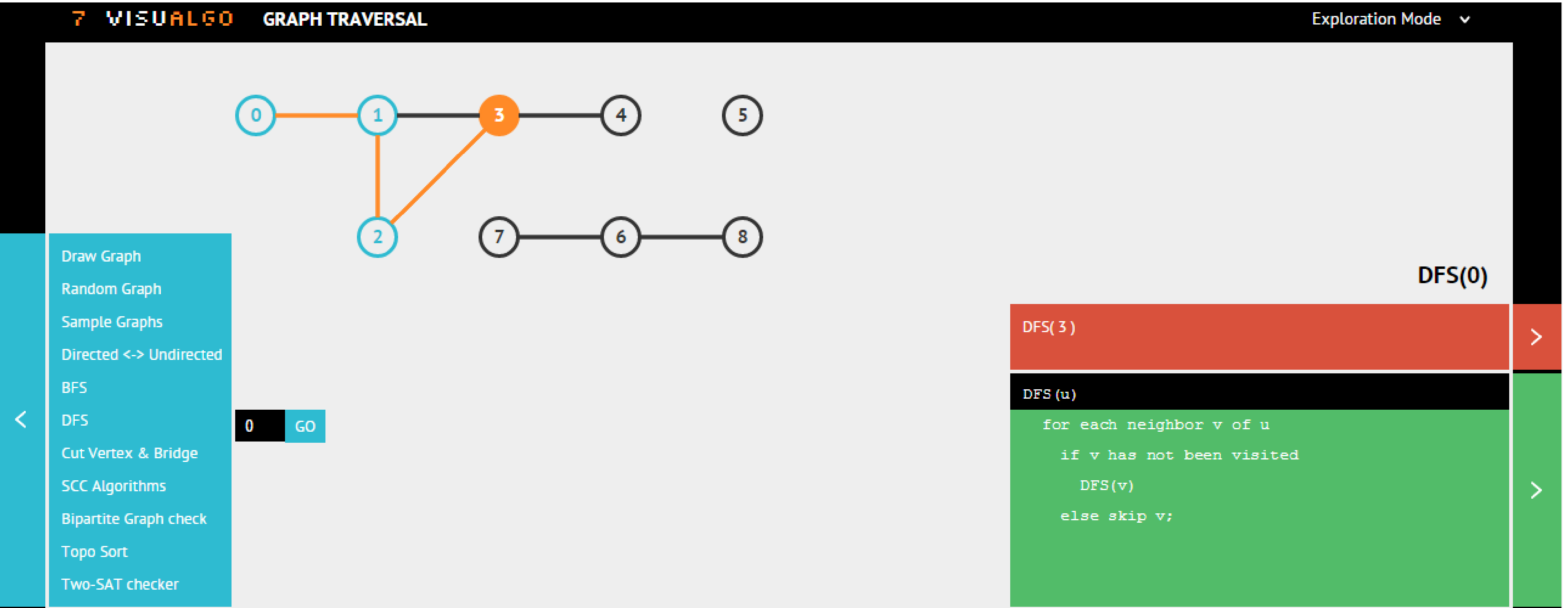


- Start from **s**
- DFS visits vertices of G in *depth-first* manner (when viewed from source vertex s)
  - Q: How to maintain such order?
    - A: Stack **S**, but we will simply use recursion (an implicit stack)
  - Q: How to differentiate visited vs unvisited vertices (to avoid cycle)?
    - A: 1D array/Vector **visited** of size V,
      visited[v] = 0 initially, and **visited[v] = 1** when **v** is visited
  - Q: How to memorize the path?
    - A: 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 of G (tree that includes all vertices of G) stored in **p**

Graph Traversal: DFS(s)

Ask VisuAlgo to perform various Depth-First Search operations on the sample Graph

In the screen shot below, we show the start of **DFS(0)**



DFS Pseudo Code

O(k)

```
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)
```

Recursive phase

for each vertex we make a recursive call
-> O(v) recurisve calls

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

O(V)

Initialization phase, same as with BFS

if adjacency matrix, we do O(v) O(V) times so it becomes O(v^2)
TOTAL: O(V^2)

if adjacency list, same as BFS, each edge is traversed at most twice because we look at all V vertices in the list, so recursion takes O(2E)
TOTAL: O(V + E)

DFS Analysis

when we pick source vertex
it will only visit everybody in same component as it

so if it is disconnect graph, we only form spanning tree
of the component that the source vertex belongs to
and not of the entire graph

can only get spanning tree of entire graph if it is connected

```
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)
```

recursively go to neighbour v and check all its neighbours

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

the way to get the path after doing dfs/bfs

```
// 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
```

we want path source to vertex 0

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

4

start at index 4, its predecessor is 3, 3's is 2 and so on until we reach the source which is 0(cause its pred is -1)

4, 3, 2, ,1 0 -> path is listed in reverse order

use recursion to output in correct order

Path Reconstruction Algorithm (2)

```
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
```

post order processing -> only after the recursive call, output it

```
// 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}
```

SOME GRAPH TRAVERSAL APPLICATIONS

What can we do with BFS/DFS? (1)

Lots of stuffs, let’s look at ***some of them***:

1. Reachability Test
2. Find Shortest Path between 2 vertices in an unweighted graph
3. Identifying/Counting Component(s)
4. Topological Sort
5. Identifying/Counting Strongly Connected Component(s)

Reachability Test

- Test whether vertex **v** is reachable from vertex **u**

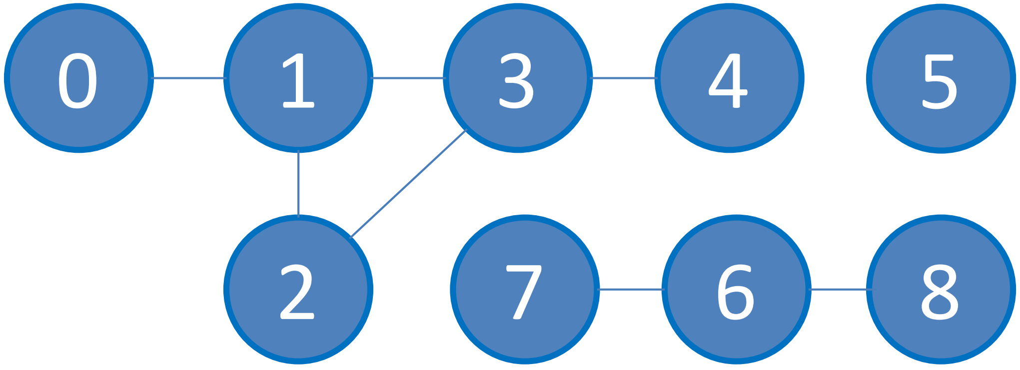
make u the source vertex and apply dfs/bfs

  – Start BFS/DFS from **s = u**

if visited[v] = 1 means we visited it from u so it is reachable -> there is path from u to v

  – If **visited[v] = 1** after BFS/DFS terminates, then **v** is *reachable* from **u**; otherwise, **v** is *not reachable* from **u**

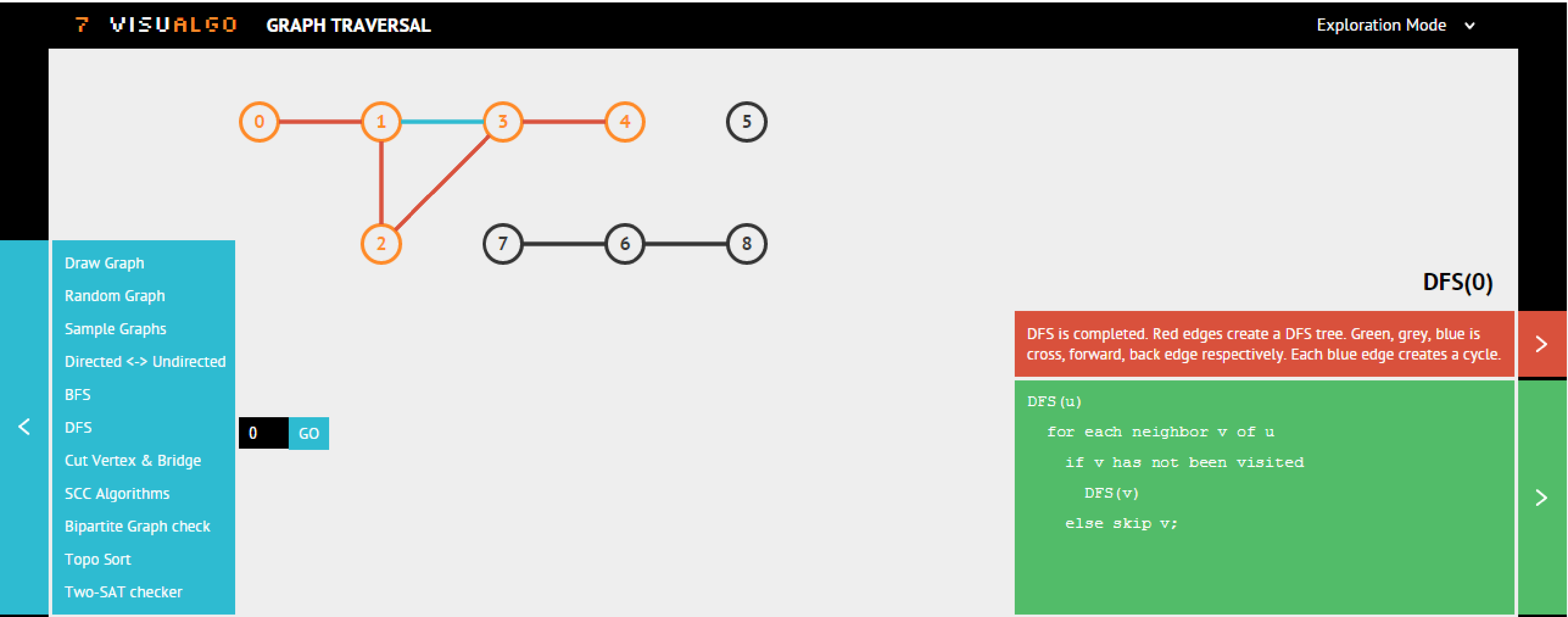
```
BFS(u) // DFSrec(u)
if visited[v] == 1
  Output "Yes"
else
  Output "No"
```



Reachability Test

Ask VisuAlgo to perform various DFS (or BFS) operations on the sample Graph

Below, we show vertices that are reachable from vertex 0



unweighted graph path length is no of edges in the path from a to b

Find Shortest Path between 2 vertices in an unweighted graph

- When the graph is **unweighted*/edges have same weight**, shortest path between any 2 vertices **u,v** is finding the **least number of edges** traversed from u to v

  if it is weighted, find shortest cost

- The O(**V+E**) Breadth First Search (BFS) traversal algorithm precisely gives such a path

  bfs property: it always explores nodes closest to the source node first, so if we look at it level by level, at each level it chooses closest node to it, so thats why it will definitely give us the shortest path from source to a vetex

- *Will cover this in more detail when we come to Shortest Path problems (last few lectures)*

* Can treat the edge weight as 1

components is only defined for undirected graphs

Identifying/Counting component(s)

- Component is sub graph containing 1 or more vertices in which any 2 vertices are connected to each other by at least one path, and is connected to no additional vertices

maximal subgraph

- With BFS/DFS, we can identify components by labeling/counting them in graph G

O(V+E) because if visited[v] == 0 prevents us from processing the vertex multiple times and calling dfs on it

imagine if it was a singular connected component, we do dfs once and all vertices are visited so time complexity is only V + E

so we can imagine that for all components, it is at most O(V + E) + O(V) = O(V + E)

- Algorithm:

```
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
```

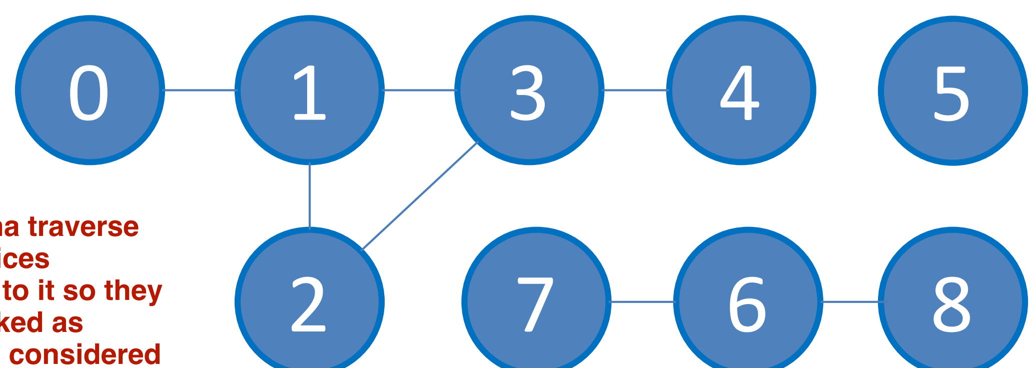
initialise to 0

for loop will check all the vertices in the vertex set

if not visited, it must be a new component

only does dfs if not visited

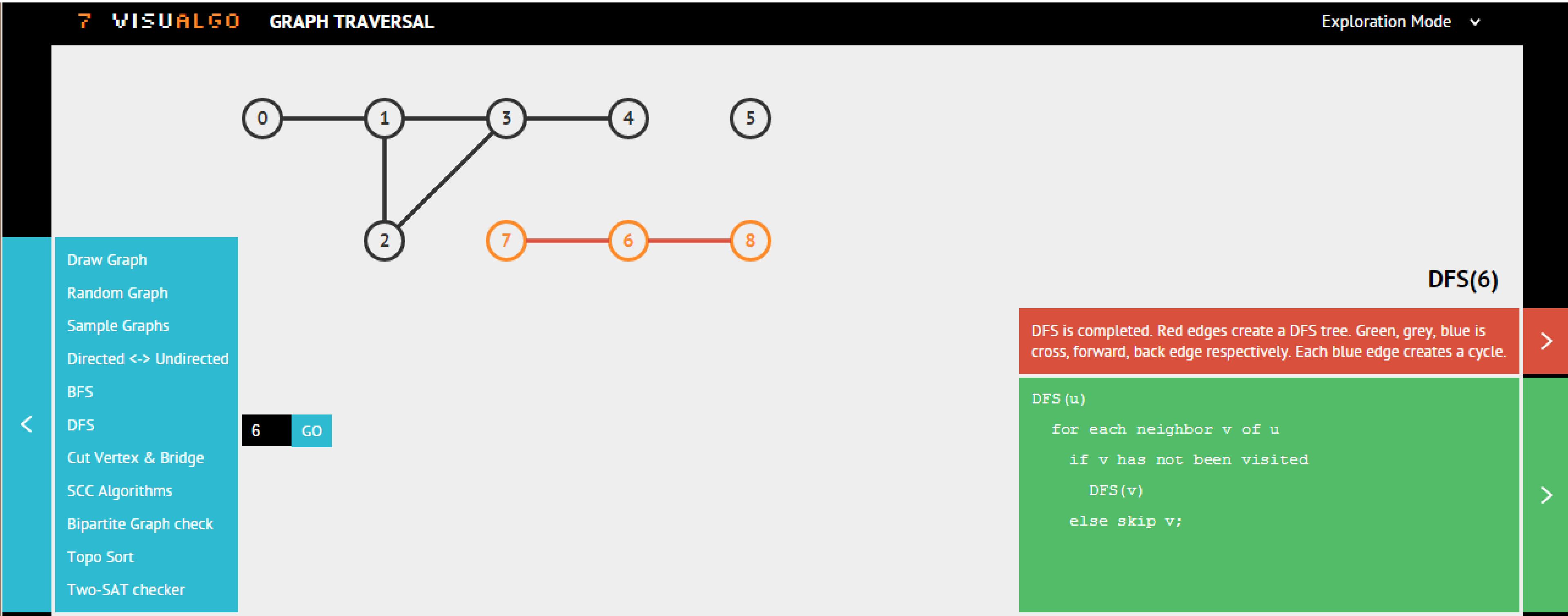
dfs is gonna traverse all the vertices connected to it so they are all marked as visited and considered one component



Identifying/Counting Component(s)

Ask VisuAlgo to perform various DFS (or BFS) operations on the sample Graph

Call **DFS(0)/BFS(0)**, **DFS(5)/BFS(5)**, then **DFS(6)/BFS(6)**

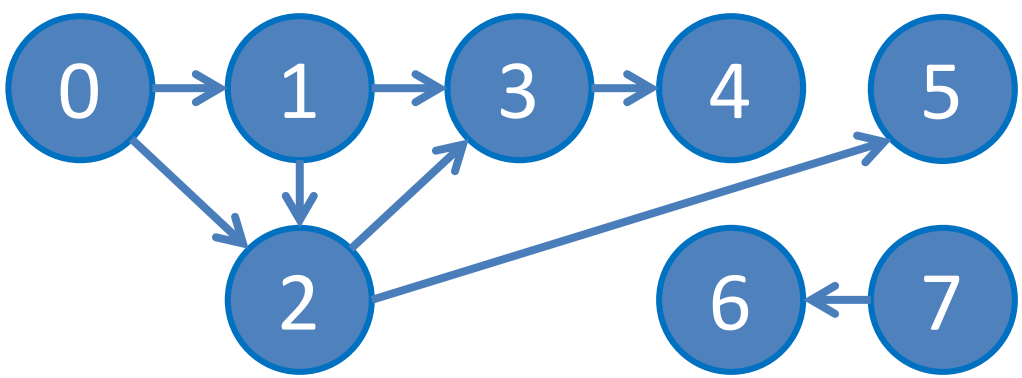


What is the time complexity for "identifying/counting component(s)"?

1. Hm… you can call O(**V**+**E**) DFS/BFS up to **V** times… I think it is O(**V***(**V**+**E**)) = O(**V^2** + **VE**)
2. It is O(**V**+**E**)…
3. Maybe some other time complexity, it is O(______)

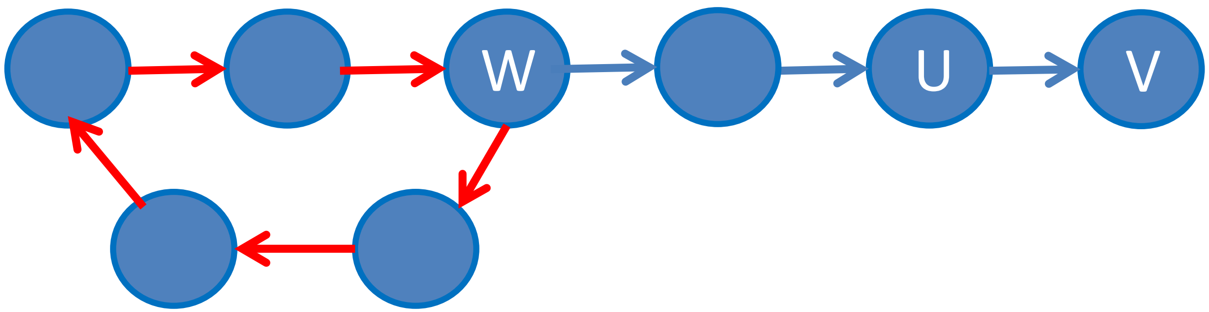
Topological Sort

- 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
- Every DAG has one *or more* topological sorts



Proof that every DAG has a Topological ordering (1)

- Lemma: If G is a DAG, it has a node with no incoming edges
- Proof by contradiction:
  - Assume every node in G has an incoming edge
  - Pick a node **V** and follow one of it's 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
  - Every vertex encountered between successive visits to **W** will form a cycle (contradiction that G is a DAG)



Proof that every DAG has a Topological ordering (2)

- 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

- This constructive proof is the basis for the BFS based algorithm (Kahn's algorithm) to compute topological ordering of a DAG

Topological Sort – Kahn’s algorithm

- If graph is a DAG, then running a modified version of BFS (Kahn’s algorithm) on it will give us a 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
- See pseudo code in the next slide

Kahn's Algorithm Pseudo Code

modifications from BFS in red

```
  for all v in V
O(V) indeg[v] ← 0          init in degree of all vertices to be 0 at the start
    p[v] ← -1
                           if there is edge u -> v, we will increment
                           indegree of v by 1
  for each edge (u,v) // get in-degree of vertices
    indeg[v] ← indeg[v] + 1   O(V + E) -> adj list
  for all v' where indeg[v'] = 0
O(V) Q ← {v'} // enqueue v'    check which vertices have
                               0 in degree and eq them
                               into q
```

O(V + E) Initialization phase

```
  while Q is not empty
    u ← Q.dequeue()
    append u to back of toposort
    for all v adjacent to u // order of neighbor
      indeg[v] ← indeg[v] - 1   since we have placed u in the topological ordering
                                we decrement the in deg count of all its
                                neighbours by 1
      if indeg[v] = 0 // add to queue
        p[v] ← u          if 0 add to q, it should be added to the toposort
        Q.enqueue(v)
```

O(V + E) Main loop

```
  Output Toposort as the topological order
```

Topological Sort – DFS based algorithm

- Running a slightly modified **DFS** on the DAG (and at the same time record the vertices in "post-order" manner) will also give us one valid topological order
  - "Post-order" = process vertex **u** after all **neighbors** of **u** have been visited
  - Use an ArrayList **toposort** to record the vertices
  - After running the algorithm, all vertices reachable by any vertex v will be placed before v in **toposort**
- See pseudo code in the next slide

for any vertex in toposort array, it will be placed only after every body reachable from it is placed to the left of it/before it

DFS Topological Sort – Pseudo Code

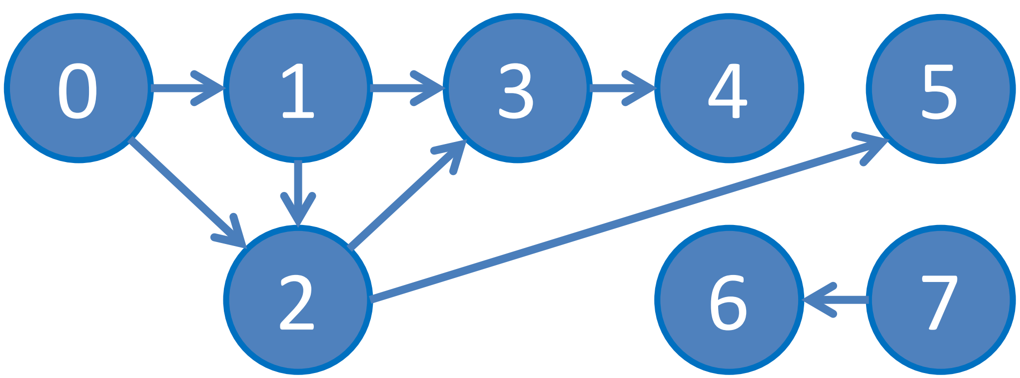
Simply look at the codes in red/underlined

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

O(V + E)

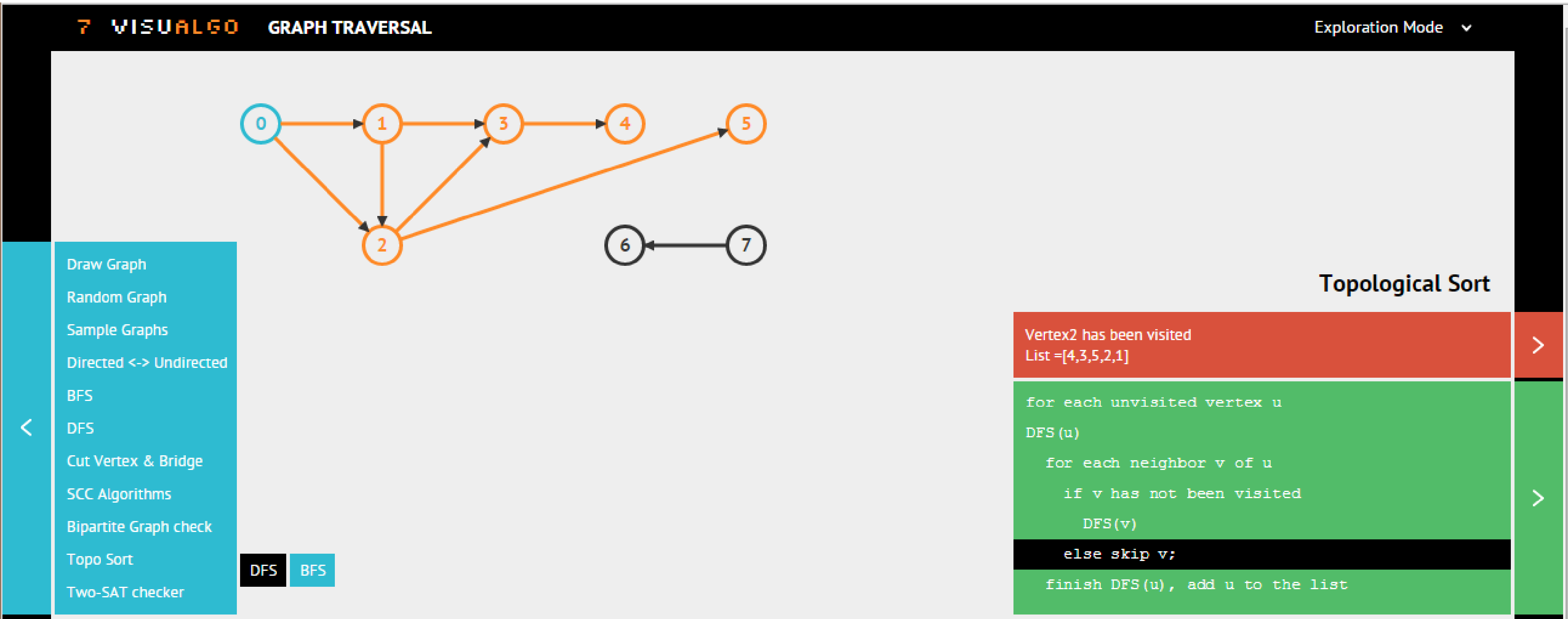
DFS Topological Sort – How it works

- Suppose we have visited all neighbors of 0 recursively with DFS
- toposort list = [[list of vertices reachable from 0], vertex 0]
  - Suppose we have visited all neighbors of 1 recursively with DFS
  - toposort list = [[[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]



Topological Sort

Ask VisuAlgo to perform Topo Sort (Kahn's/DFS) operation on the sample Graph



scc of directed graphs is equivalent to connected component of undirected graphs

Identifying/Counting Strongly Connected Component(s) (SCCs)

- A **strongly connected component** (SCC) is a sub graph of a directed graph containing 1 or more vertices in which any 2 vertices (if there are >= 2 vertices) are connected to each other by at least one path, and is connected to no additional vertices (maximal subgraph)

  every vertex inside strongly connected component can visit every other vertex

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

  SCC: if start from any node in a component, can reach every other node

- Identifying SCCs is harder than identifying components due to the direction of the edges.
- One algorithm to do this is Kosaraju's algorithm which makes use of DFS

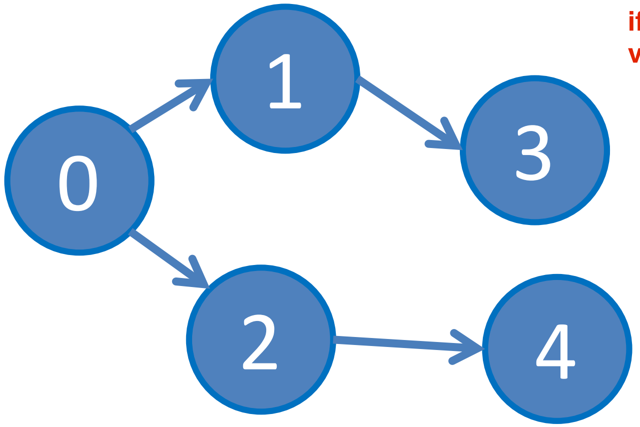
How many SCCs does the graph below have? (1)

this is a DAG -> directed acyclic graph
so every vertex is a scc

if there is a no cycle, the no of vertices =
no of SCC

if there is a cycle, the no of sccs < no of
vertices

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



how to check if directed graph has cycle?
run kosaraju's algorithm

if there is a no cycle, the no of vertices =
no of SCC

if there is a cycle, the no of sccs < no of
vertices

if have scc of more than one vertex, it has
to be a cycle if not we cannot have every vertex
connecting every other vertex

How many SCCs does the graph below have? (2)

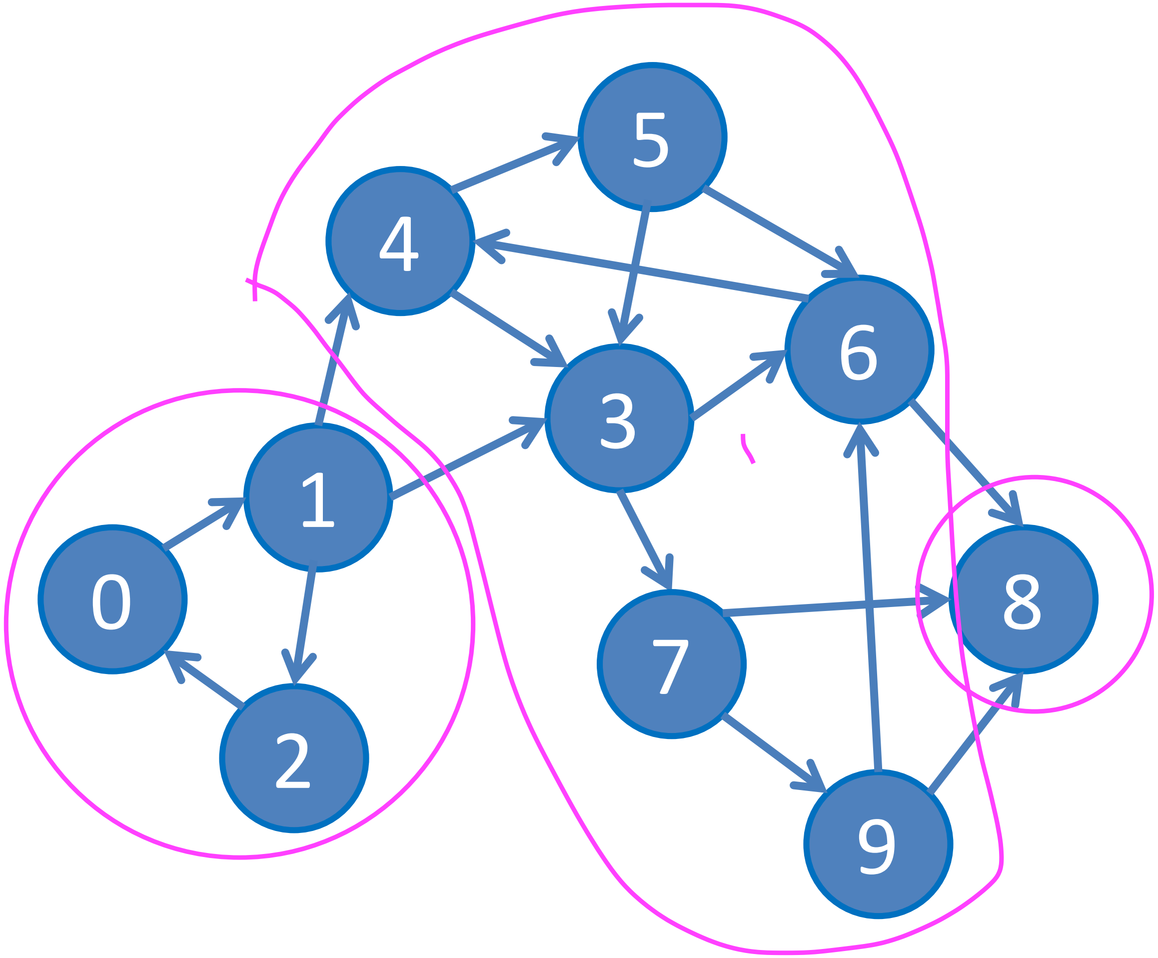
a) 1

b) 2

c) 3

d) 4

e) 5



Kosaraju's Algorithm to identify SCCs

topological sort is only fro DAGs if the graph is not a DAG, we just want a post order sequence

1. Perform DFS topological sort algo on the given directed graph G
   – i.e post-order processing of the vertices into an array **K** O(V + E)
2. Create transpose graph G' of G

   in new graph u is the vertex and v is the neighbour

   – i.e create a graph where the direction of all edges in G is reversed
     • for each vertex v in adj. list of G and for each neighbor u of v, add edge u->v to G' O(V + E)

   advantage of this is that we will not be able to go anywhere apart from the scc

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)
```

O(V + E)

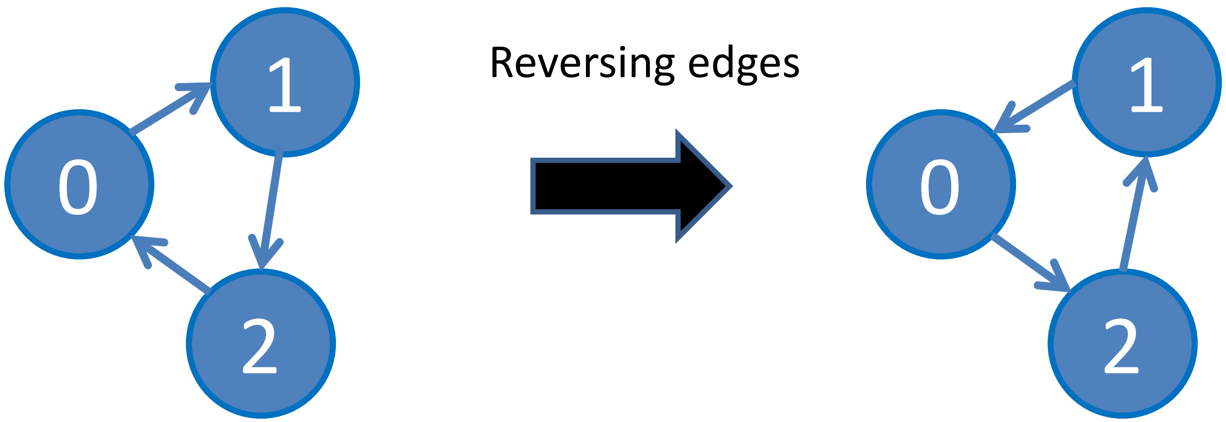
same algo as counting components for undirected graph

What is time complexity of Kosaraju's Algorithm?

O(V + E)

Why does Kosaraju's algorithm work? (1)

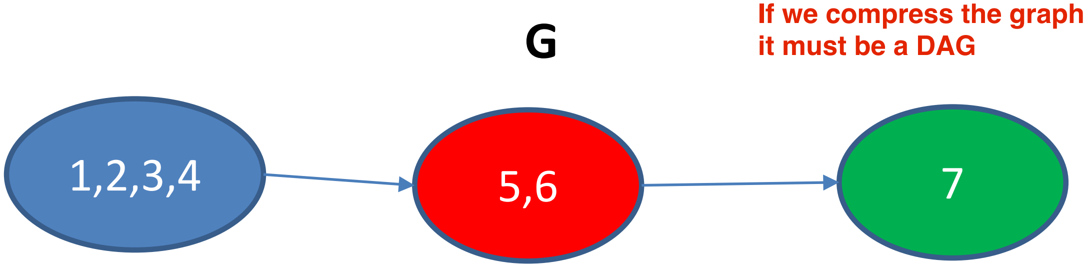
- Given any SCC, reversing all the edges in the SCC will still result in the same SCC



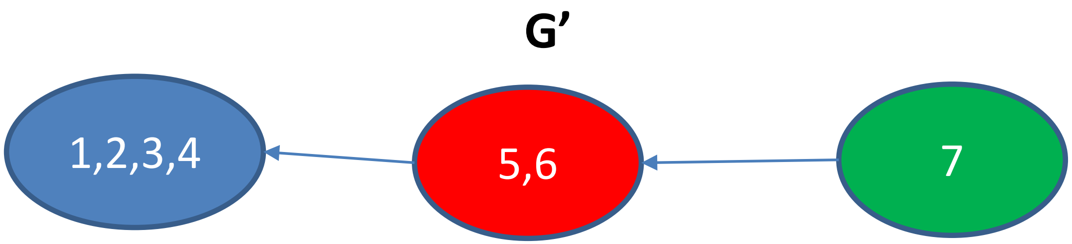
Why does Kosaraju's algorithm work? (2)

- If we have the following SCCs in a directed graph

If we compress the graph to make big vertices, we get a DAG
it must be a DAG



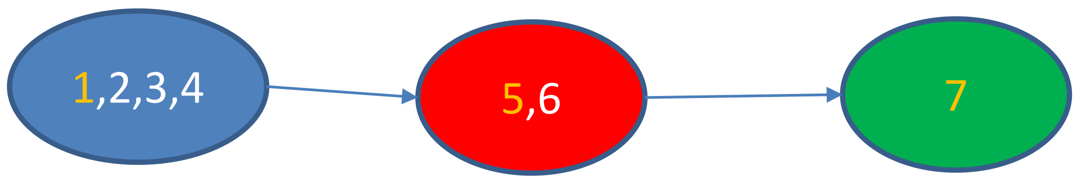
- If we flip the graph we will still get the same SCCs but with the edges linking them flipped (if there are such edges)



Why does Kosaraju's algorithm work? (3)

- Now if we view each SCC in G or G' as a vertex, then G or G' is actually a DAG!
- Let v' be the 1st vertex visited in each SCC when we perform DFS toposort algo 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

7 put first because of post order processing will only process a vertex after finishing its right and left

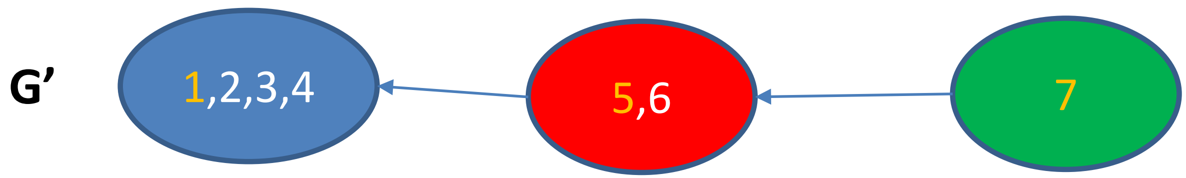
{2,3,4} may be in these 3 segments

K = ... 7.... 5 1

6 may be in these 2 segments

Why does Kosaraju's algorithm work? (4)

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



Process **K** from back to front

K = ...7 .... 5 1

- 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 SCCs

Trade-Off

O(V+E) DFS

- Pros:
  - Required for counting SCCs
- Cons:
  - Cannot solve SSSP on unweighted graphs

O(V+E) BFS

- Pros:
  - Can solve SSSP on unweighted graphs (revisited in later lectures)
- Cons:
  - Cannot be used to count SCCs

the other stuff like reachability test, counting components and topo sort can use with dfs or bfs

Summary

In this lecture, we have looked at:

- 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, SP in unweighted/same weight graph, Counting Components, Topological sort, Counting SCCs