CS2040 – Data Structures and Algorithms

Lecture 13 – 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

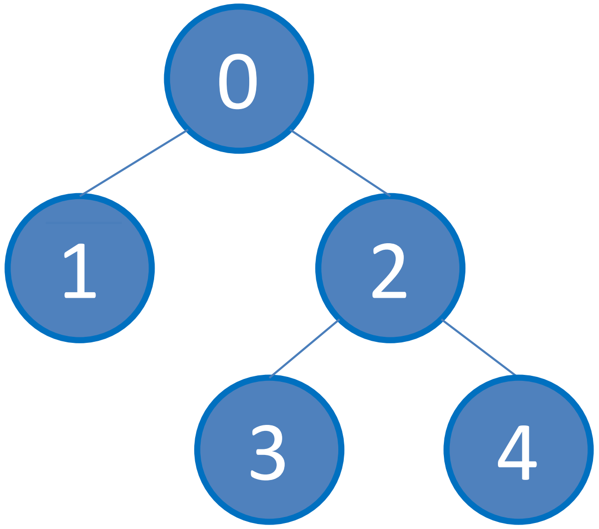
```
pre(u)
  visit(u);
  pre(u->left);
  pre(u->right);
```

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

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

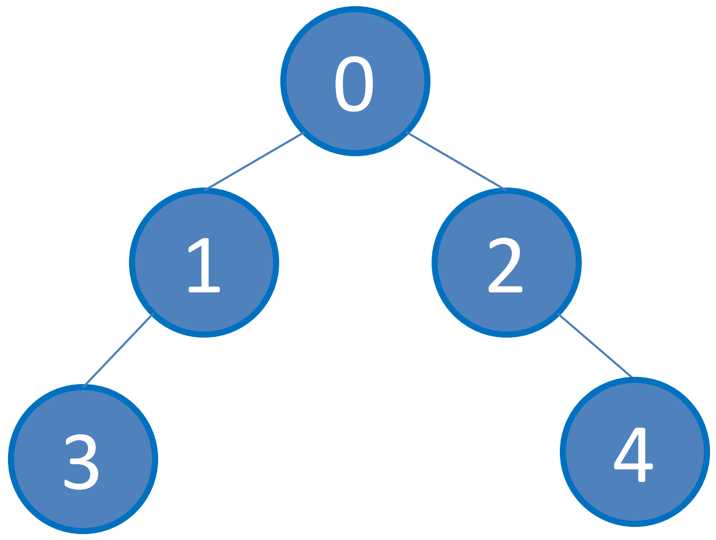
We start binary tree traversal from root:

- 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

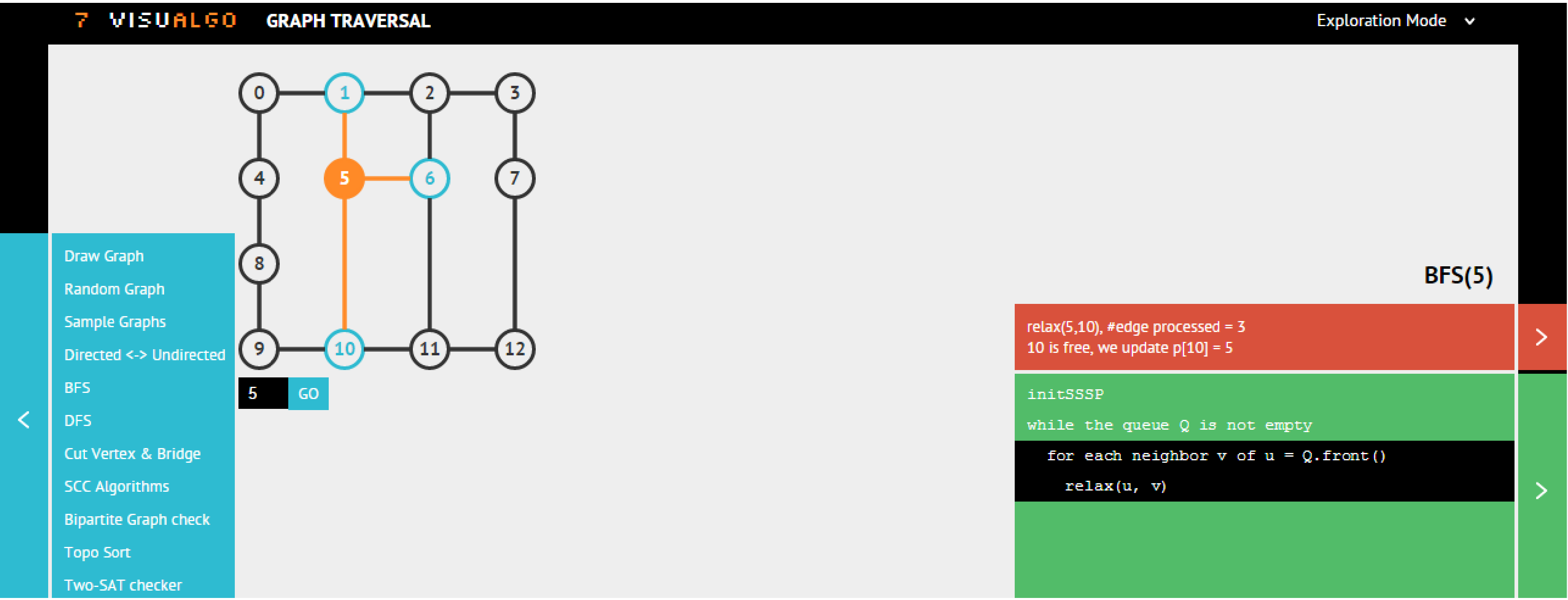
- Start from **s**
- BFS visits vertices of G in *breadth-first* manner (when viewed from source vertex s)
  - Q: How to maintain such order?
    - A: Use queue **Q**, initially, it contains only **s**
  - Q: How to differentiate visited vs unvisited vertices (to avoid cycle)?
    - A: 1D array/Vector **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)**



relax(5,10), #edge processed = 3
10 is free, we update p[10] = 5

```
initSSSP
while the queue Q is not empty
  for each neighbor v of u = Q.front()
    relax(u, v)
```

BFS Pseudo Code

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

Initialization phase

```
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)
```

Main loop

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

BFS Analysis

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

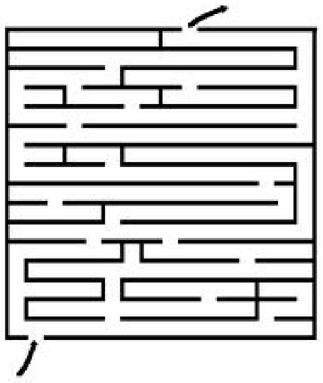
- Initialization is O(**V**)
- For the while loop
    - Case 1 : disconnected graph **E** = 0, takes O(**E**)
    - Case 2: connected graph
        - Each vertex is in the queue once (visited will be flagged to avoid cycle)
        - When a vertex is dequeued, all its neighbors are scanned (for loop);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

```
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



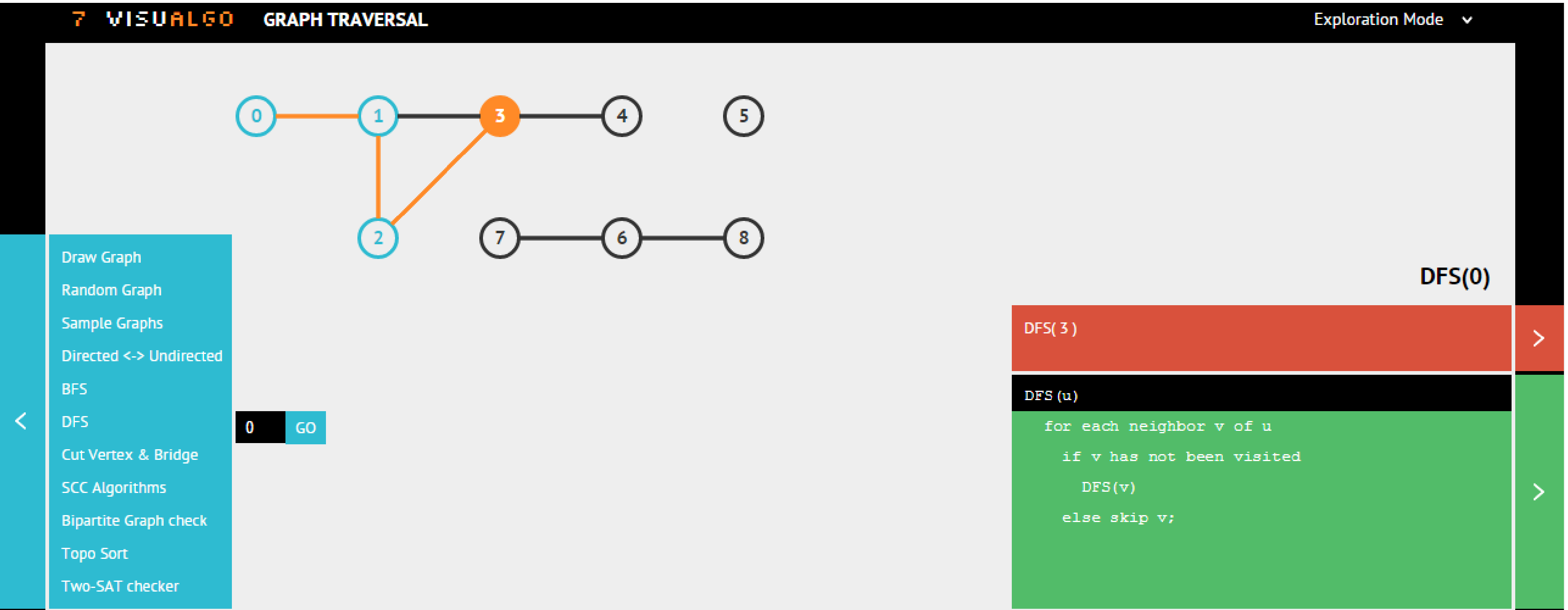
- Start from **s**
- DFS visits vertices of G in *depth-first* manner (when viewed from source vertex s)
  - Q: How to maintain such order?
    - A: Stack **S**, but we will simply use recursion (an implicit stack)
  - Q: How to differentiate visited vs unvisited vertices (to avoid cycle)?
    - A: 1D 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(0)

DFS(3)

```
DFS(u)
  for each neighbor v of u
    if v has not been visited
      DFS(v)
    else skip v;
```

DFS 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)
```

Recursive phase

```
// 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 // 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)

// 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)

```
// 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 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

// 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)

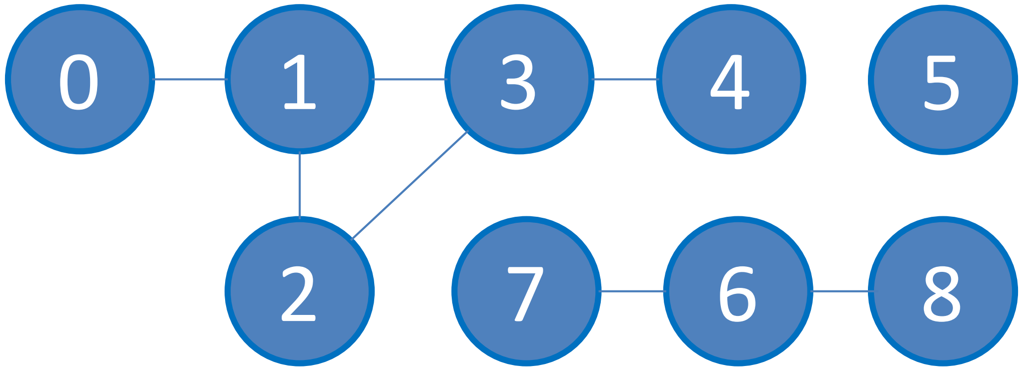
Lots of stuffs, let’s look at ***some of them***:

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

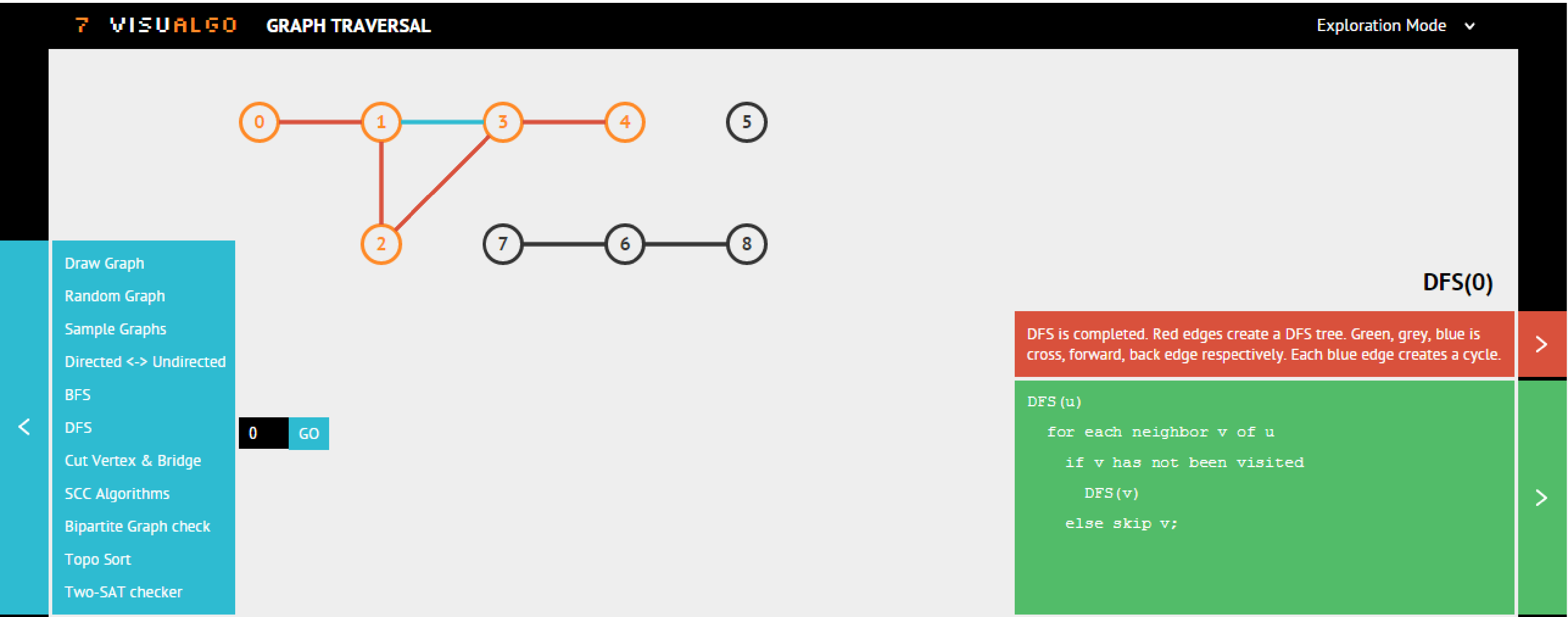
```
BFS(u) // DFSrec(u)
if visited[v] == 1
  Output "Yes"
else
  Output "No"
```



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



DFS is completed. Red edges create a DFS tree. Green, grey, blue is cross, forward, back edge respectively. Each blue edge creates a cycle.

```
DFS(u)
  for each neighbor v of u
    if v has not been visited
      DFS(v)
    else skip v;
```

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

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

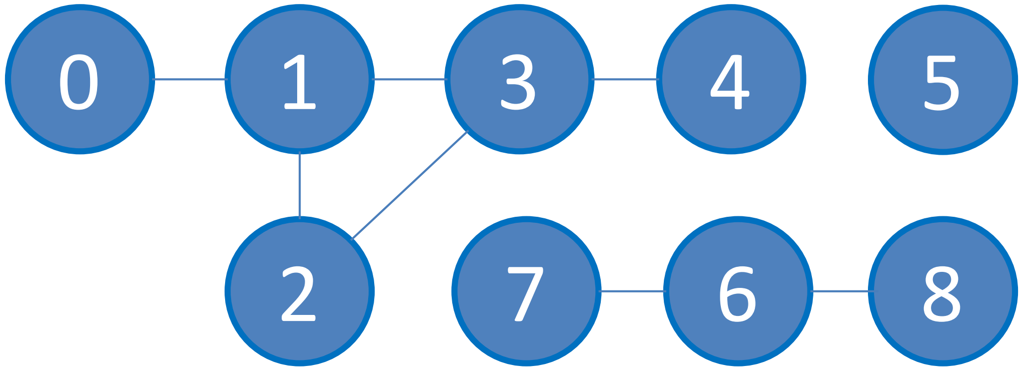
- *Will cover this in more detail when we come to Shortest Path problems (last few lectures)*

* Can treat the edge weight as 1

Identifying/Counting component(s)

- Component is sub 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
- With BFS/DFS, we can identify components by labeling/counting them in graph G
- Algorithm:

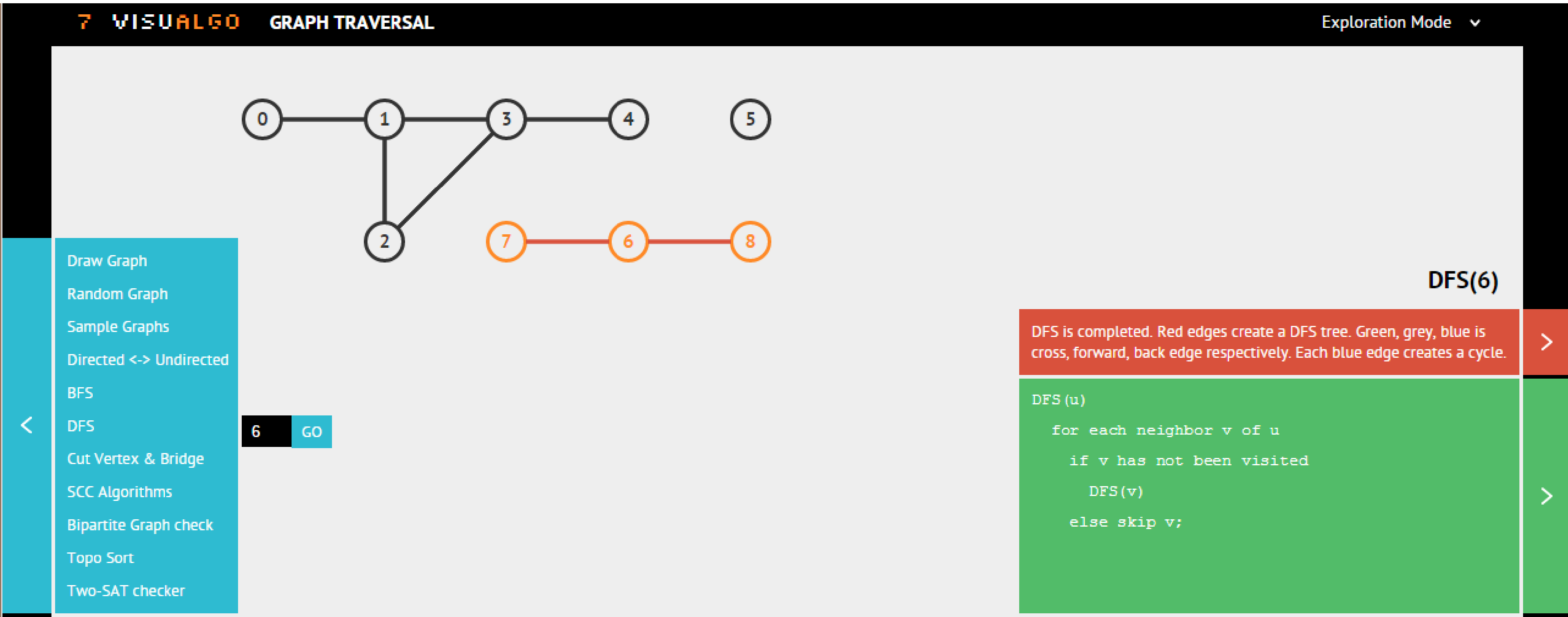
```
CC ← 0
for all v in V
  visited[v] ← 0
for all v in V // O(V)?
  if visited[v] == 0
    CC ← CC + 1
    DFSrec(v)//O(V+E)?
    // BFS from v
    // is also OK
```



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)**



DFS is completed. Red edges create a DFS tree. Green, grey, blue is cross, forward, back edge respectively. Each blue edge creates a cycle.

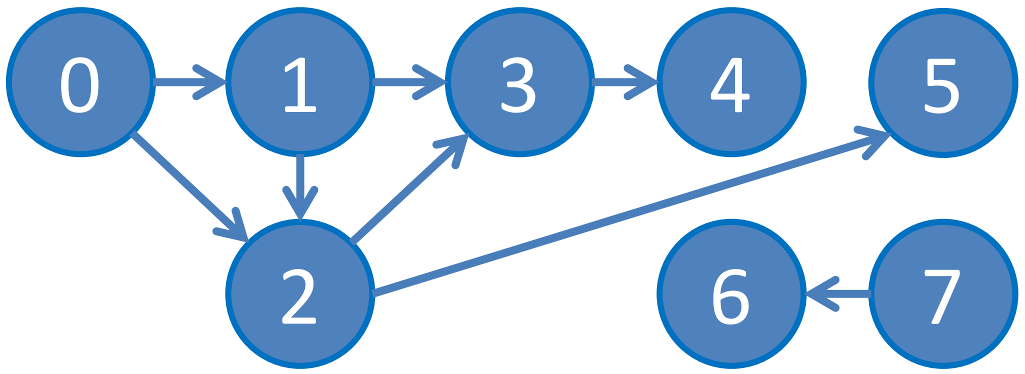
```
DFS(u)
  for each neighbor v of u
    if v has not been visited
      DFS(v)
    else skip v;
```

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(_______)

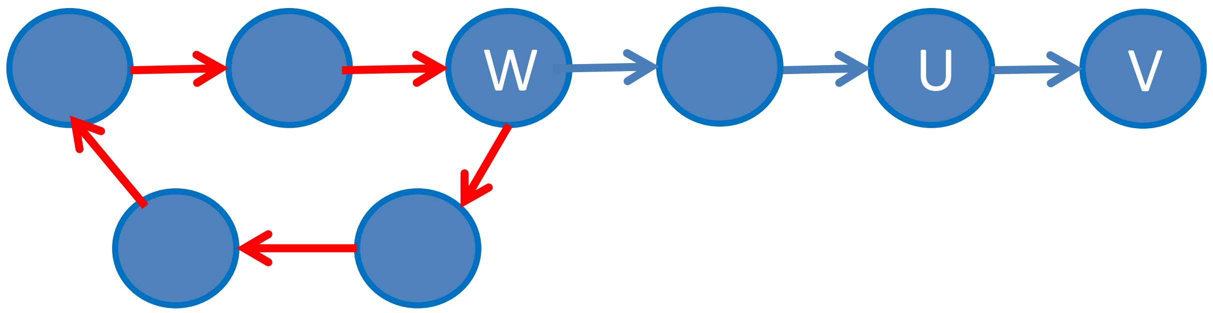
Topological Sort

- 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
  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 neighbor
    indeg[v] ← indeg[v] - 1
    if indeg[v] = 0 // add to queue
      p[v] ← u
      Q.enqueue(v)

Output Toposort as the topological order
```

Initialization phase

Main loop

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
  - Use an ArrayList **toposort** to record the vertices
  - “Post-order” processing = process vertex **u** (i.e put **u** in **toposort**) after all **neighbors** of **u** have been visited
  - 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

DFS Topological Sort – Pseudo Code

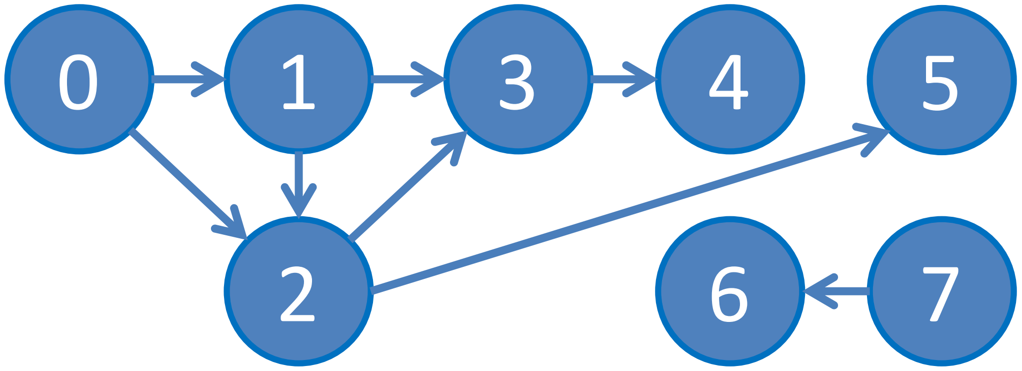
Simply look at the codes in red/underlined

```
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
```

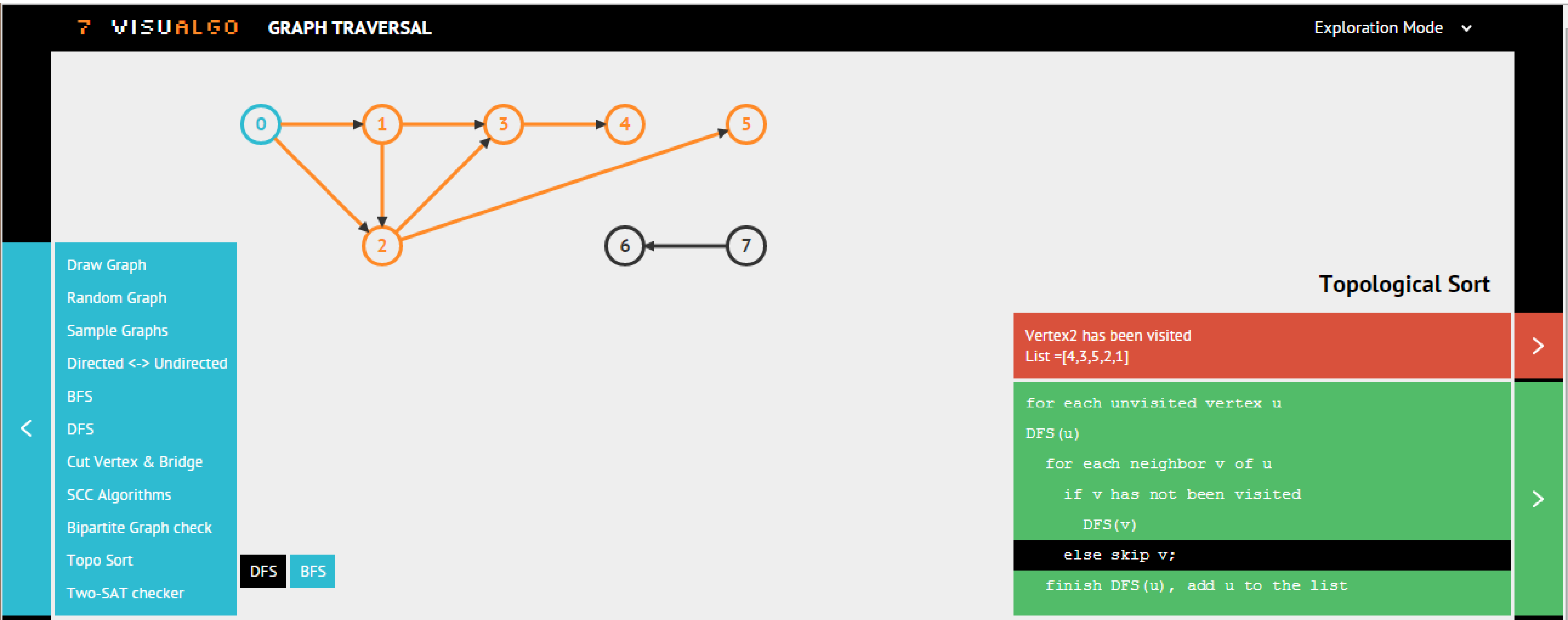
DFS Topological Sort – How it works

- 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$\underline{, 3, 5, 2, 1, 0}$, 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



Vertex2 has been visited
List =[4,3,5,2,1]

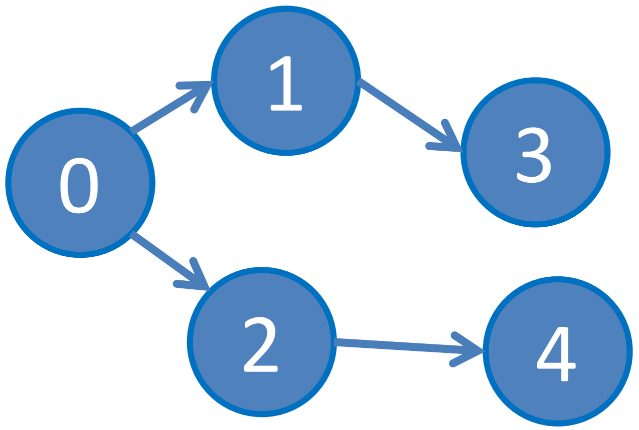
```
for each unvisited vertex u
DFS(u)
  for each neighbor v of u
    if v has not been visited
      DFS(v)
    else skip v;
  finish DFS(u), add u to the list
```

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)
- 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 which makes use of DFS

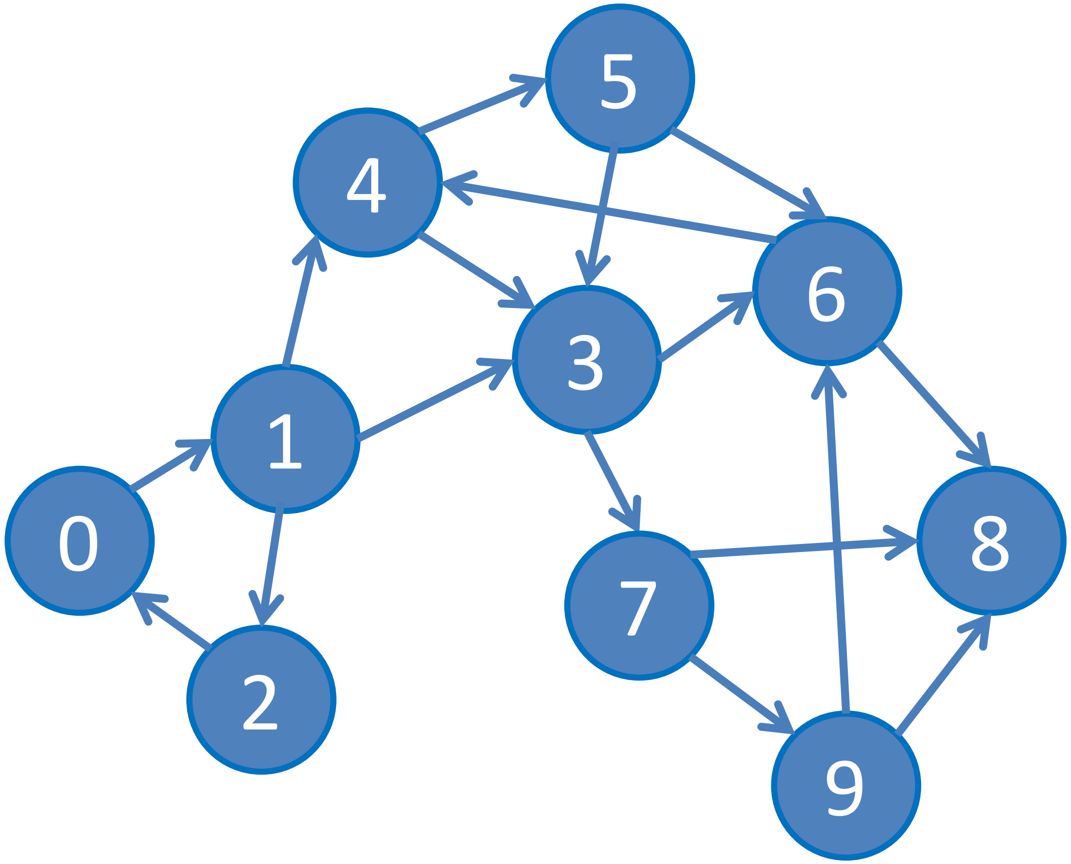
How many SCCs does the graph below have? (1)

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



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

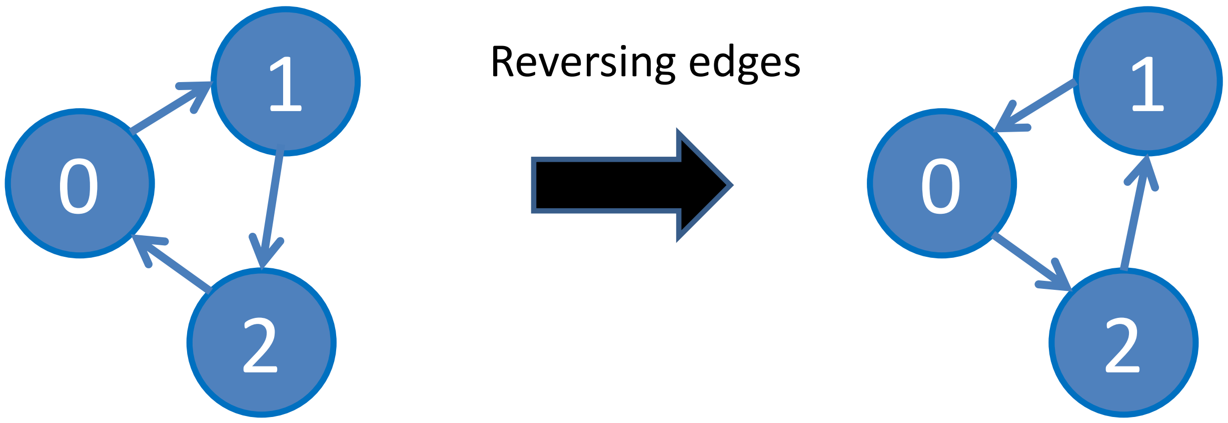
1. Perform DFS topological sort algo on the given directed graph G
   - i.e post-order processing of the vertices into an array **K**
2. Create transpose graph G’ of G
   - i.e create a graph where the direction of all edges in G is reversed
     - for each vertex v in adj. list of G and for each neighbor u of v, add edge u->v to G’
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)
```

What is time complexity of Kosaraju’s Algorithm?

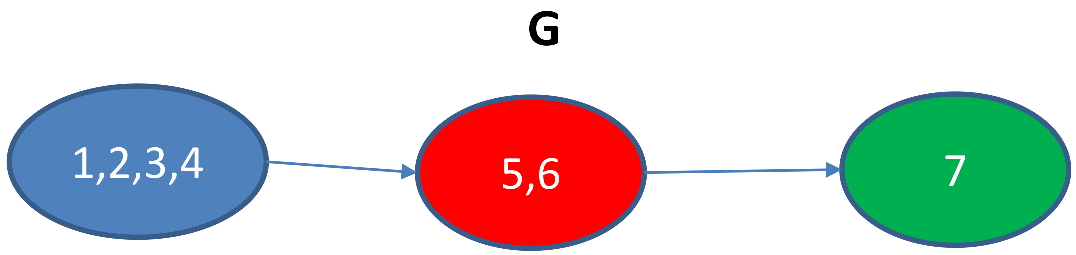
Why does Kosaraju’s algorithm work? (1)

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

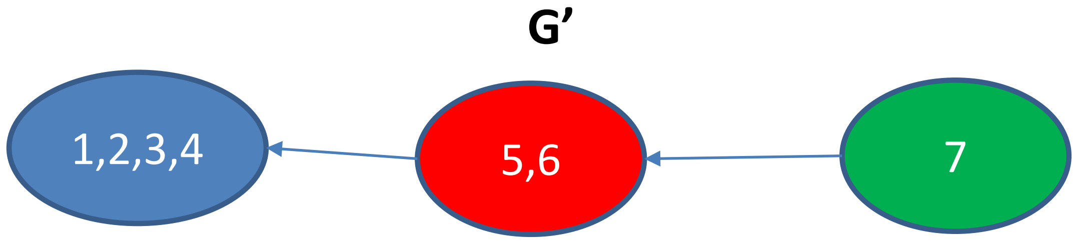


Why does Kosaraju's algorithm work? (2)

- If we have the following SCCs in a directed graph

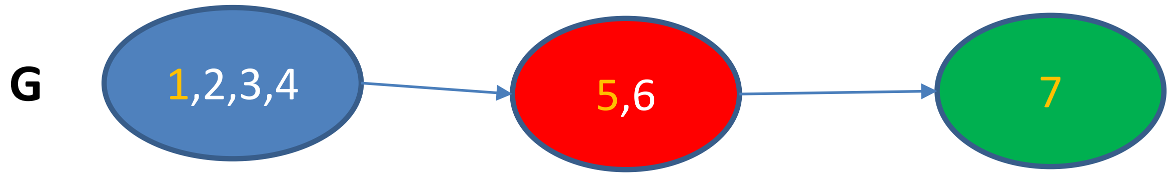


- If we flip the graph we will still get the same SCCs but with the edges linking them flipped (if there are such edges)

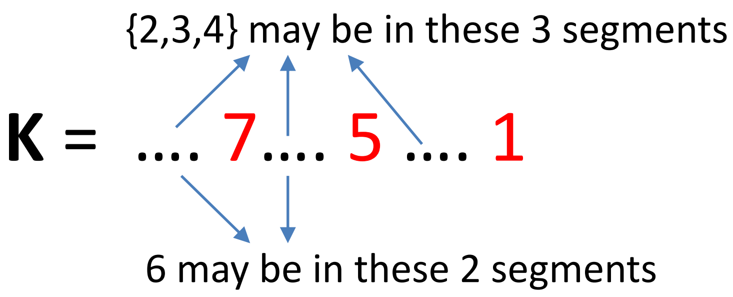


Why does Kosaraju's algorithm work? (3)

- 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**

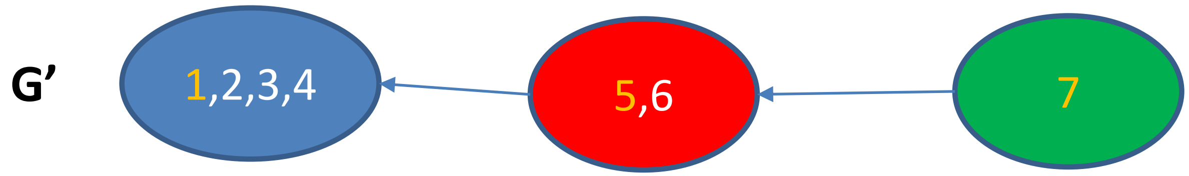


Assuming the colored vertex is v' (the first one visited) in its respective SCC


{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 <u>unvisited</u> 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

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