

# CS2010 – Data Structures and Algorithms II

## Lecture 06 – Maze Exploration

[chongket@comp.nus.edu.sg](mailto:chongket@comp.nus.edu.sg)



# Admin Stuff (1)

- PS2 deadline today at 23:59
- PS3 will be opened on Saturday 23<sup>rd</sup> Sep, 12 noon.
- PS3 Deadline is Friday, 6<sup>th</sup> Oct 23:59

# Admin Stuff (2)

- Final PSA for midterms
  - Online Quiz 1 this Thursday during your lab session
  - Written Quiz 1 this Friday 7pm to 8:30pm
  - Both online and written quiz is open book (but not open internet so no electronic device)

# Admin Stuff (3) – Written Quiz 1 venues

Room	Starting Name	Ending Name
SR1	A, AARON SEAH YUHAO	T, TAN SHENG YANG JERALD
SR10	T, TAN WEI HAO	Y, YANG YUQING
SR8	Y, YAP NI	Z, ZOU YUTONG

Names are as listed in IVLE class roster

If you go to wrong venue, I will ask you to move over as I will prepare exact number of copies of exam papers in these 3 venues

# Outline

Continue Week 05 stuffs (Graph DS Applications)

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/dfsbfbs>

Reference: Mostly from CP3 Section 4.2

- Not all sections in CP3 chapter 4 are used in CS2010!
  - Some are quite advanced :O

# **SOME GRAPH DATA STRUCTURE APPLICATIONS**

# So, what can we do so far? (1)

With just graph DS, not much, but here are some:

- Counting **V** (or **|V|**) (the number of vertices)
  - Very trivial for both **AdjMatrix** and **AdjList**: **V** = number of rows!
  - Sometimes this number is stored in separate variable so that we do not have to re-compute this every time, that is,  $O(1)$ , *especially if the graph never changes after it is created*
  - To think about: How about **EdgeList**?

Adjacency Matrix

	0	1	2	3	4	5	6
0	0	1	1	0	0	0	0
1	1	0	1	1	0	0	0
2	1	1	0	0	1	0	0
3	0	1	0	0	1	0	0
4	0	0	1	1	0	1	0
5	0	0	0	0	1	0	1
6	0	0	0	0	0	1	0

Adjacency List

0:	1	2	
1:	0	2	3
2:	0	1	4
3:	1	4	
4:	2	3	5
5:	4	6	
6:	5		

Edge List

0:	0	1
1:	0	2
2:	1	2
3:	1	3
4:	2	4
5:	3	4
6:	4	5
7:	5	6

# So, what can we do so far? (2)

- *See this during live lecture*

**Adjacency Matrix**

	0	1	2	3	4	5	6
0	0	1	1	0	0	0	0
1	1	0	1	1	0	0	0
2	1	1	0	0	1	0	0
3	0	1	0	0	1	0	0
4	0	0	1	1	0	1	0
5	0	0	0	0	1	0	1
6	0	0	0	0	0	1	0

**Adjacency List**

0:	1	2	
1:	0	2	3
2:	0	1	4
3:	1	4	
4:	2	3	5
5:	4	6	
6:	5		

**Edge List**

0:	0	1
1:	0	2
2:	1	2
3:	1	3
4:	2	4
5:	3	4
6:	4	5
7:	5	6



# So, what can we do so far? (3)

- *See this during live lecture*

**Adjacency Matrix**

	0	1	2	3	4	5	6
0	0	1	1	0	0	0	0
1	1	0	1	1	0	0	0
2	1	1	0	0	1	0	0
3	0	1	0	0	1	0	0
4	0	0	1	1	0	1	0
5	0	0	0	0	1	0	1
6	0	0	0	0	0	1	0

**Adjacency List**

0:	1	2	
1:	0	2	3
2:	0	1	4
3:	1	4	
4:	2	3	5
5:	4	6	
6:	5		

**Edge List**

0:	0	1
1:	0	2
2:	1	2
3:	1	3
4:	2	4
5:	3	4
6:	4	5
7:	5	6

# So, what can we do so far? (4)

- *See this during live lecture*

**Adjacency Matrix**

	0	1	2	3	4	5	6
0	0	1	1	0	0	0	0
1	1	0	1	1	0	0	0
2	1	1	0	0	1	0	0
3	0	1	0	0	1	0	0
4	0	0	1	1	0	1	0
5	0	0	0	0	1	0	1
6	0	0	0	0	0	1	0

**Adjacency List**

0:	1	2	
1:	0	2	3
2:	0	1	4
3:	1	4	
4:	2	3	5
5:	4	6	
6:	5		

**Edge List**

0:	0	1
1:	0	2
2:	1	2
3:	1	3
4:	2	4
5:	3	4
6:	4	5
7:	5	6

# Trade-Off

## Adjacency Matrix

### Pros:

- Existence of edge  $i$ - $j$  can be found in  $O(1)$
- Good for dense graph/  
Floyd Warshall's (Lecture 12)

### Cons:

- $O(V)$  to enumerate neighbors of a vertex
- $O(V^2)$  space

## Adjacency List

### Pros:

- $O(k)$  to enumerate  $k$  neighbors of a vertex
- Good for sparse graph/Dijkstra's/  
DFS/BFS,  $O(V+E)$  space

### Cons:

- $O(k)$  to check the existence of edge  $i$ - $j$
- A small overhead in maintaining the list (for sparse graph)

# VisuAlgo Graph DS Exploration (1)

Click each of the sample graphs one by one and verify the content of the corresponding **Adjacency Matrix**, **Adjacency List**, and **Edge List**

7 VISUALGO UNDIRECTED / UNWEIGHTED UNDIRECTED / WEIGHTED DIRECTED / UNWEIGHTED DIRECTED / WEIGHTED Exploration Mode ▾

Tree Graph  
Star Graph  
K5 Graph  
CP2.2  
CP2.5A - Grid Graph  
CP2.5B - Grid Graph  
CP2.5C - Knight Jump Graph  
CP4.3.1 - MST  
CP4.2  
CP4.2.5 - DAG  
CP4.5  
CP4.6.1 - Flow Graph  
CP4.8

• Is tree: No • Is complete: No • Is bipartite: No • Is DAG: No

Adjacency matrix							
	0	1	2	3	4	5	6
0	0	1	1	0	0	0	0
1	1	0	1	1	0	0	0
2	1	1	0	0	1	0	0
3	0	1	0	0	1	0	0
4	0	0	1	1	0	1	0
5	0	0	0	0	1	0	1
6	0	0	0	0	0	1	0

Adjacency list			
0:	1	2	
1:	0	2	3
2:	1	4	0
3:	1	4	
4:	3	2	5
5:	4	6	
6:	5		

Edge list		
0:	0	1
1:	1	2
2:	3	1
3:	3	4
4:	4	2
5:	4	5
6:	5	6
7:	2	0

# VisuAlgo Graph DS Exploration (2)

Now, use your mouse over the currently displayed graph and **start drawing some new vertices and/or edges** and see the updates in AdjMatrix/AdjList/EdgeList structures

7 VISUALGO    UNDIRECTED / UNWEIGHTED    UNDIRECTED / WEIGHTED    DIRECTED / UNWEIGHTED    DIRECTED / WEIGHTED    Exploration Mode ▾

• Click on empty space to add vertex  
• Drag from vertex to vertex to add edge  
• Select + Delete to delete vertex/edge  
• Select Edge + Enter to change edge's weight  
• Press Ctrl to Drag vertex around

• Click anywhere to switch to our default mode undirected/unweighted or choose another graph drawing mode

• Is tree: Yes    • Is complete: No    • Is bipartite: Yes    • Is DAG: No

	0	1	2
0	0	1	0
1	1	0	1
2	0	1	0

	0	1	2
0 :		1	
1 :	0		2
2 :	1		

	0	1	2
0 :	0	1	
1 :	1		2

# **GRAPH TRAVERSAL ALGORITHMS**

# Review – Binary Tree Traversal

In a binary tree, there are three standard traversal:

- Preorder
- **Inorder**
- Postorder

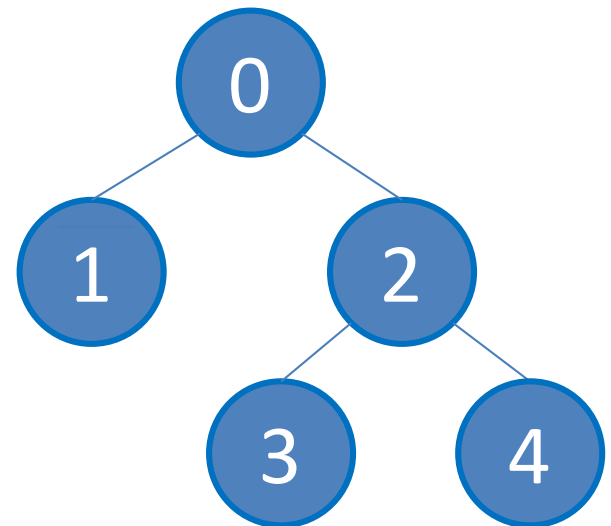
```
pre(u)
  visit(u);
  pre(u->left);
  pre(u->right);
```

```
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



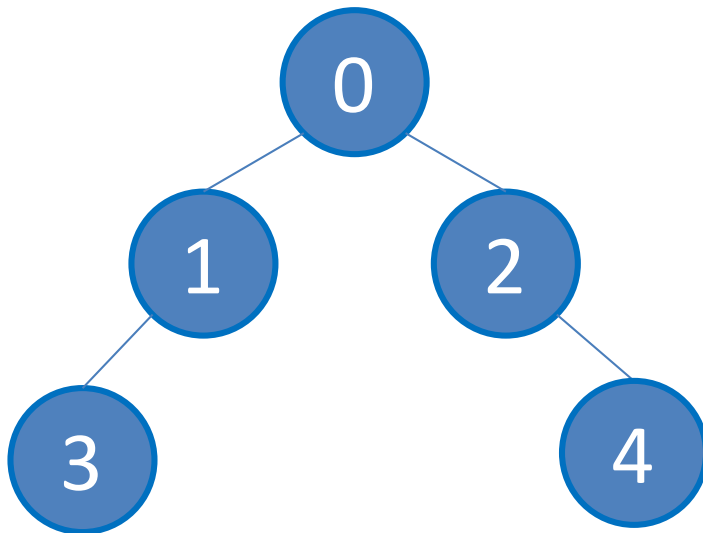
# What is the **PostOrder** Traversal of this Binary Tree?

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  **$u \rightarrow v \rightarrow w \rightarrow u \rightarrow v \dots$**  indefinitely

# Traversing a Graph (2)

**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 predecessor (or parent) of  $v$



# Graph Traversal: BFS(s)

Ask VisuAlgo to perform various Breadth-First Search operations on the sample Graph (CP3 4.3, Undirected)

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

The screenshot displays the VisuAlgo Graph Traversal interface. The top bar shows the logo and "GRAPH TRAVERSAL" with a dropdown menu set to "Exploration Mode". The main area features a 4x4 grid graph with 13 nodes (0-12). Node 5 is highlighted in orange, and its neighbors (1, 6, 10) are highlighted in light blue. The left sidebar contains a menu with options: Draw Graph, Random Graph, Sample Graphs, Directed <-> Undirected, BFS (selected), DFS, Cut Vertex & Bridge, SCC Algorithms, Bipartite Graph check, Topo Sort, and Two-SAT checker. Below the menu, the input "5" is entered, and the "GO" button is visible. On the right, the text "BFS(5)" is displayed above a log of operations: "relax(5,10), #edge processed = 3" and "10 is free, we update p[10] = 5". Below the log, a code block shows the pseudocode for the relaxation step: "initSSSP", "while the queue Q is not empty", "for each neighbor v of u = Q.front()", and "relax(u, v)".

7 VISUALGO GRAPH TRAVERSAL Exploration Mode ▾

0 1 2 3  
4 5 6 7  
8  
9 10 11 12

5 GO

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

```
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)
```

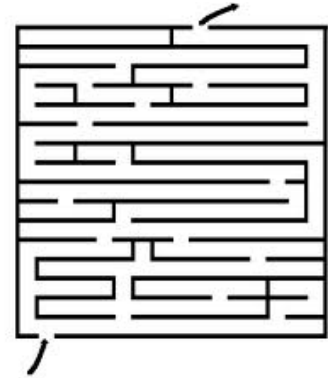
```
// we can then use information stored in visited/p
```

Time Complexity:  $O(V+E)$

- Each vertex is only in the queue once  $\sim O(V)$
- Every time a vertex is dequeued, all its **k** neighbors are scanned; After all vertices are dequeued, all **E** edges are examined  $\sim O(E)$   
→ assuming that we use **Adjacency List!**
- Overall:  $O(V+E)$

# 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 predecessor (or parent) of  $v$





# Graph Traversal: DFS(s)

Ask VisuAlgo to perform various Depth-First Search operations on the sample Graph (CP3 4.1, Undirected)

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

The screenshot displays the VisuAlgo Graph Traversal interface. The top bar shows the VisuAlgo logo and the title "GRAPH TRAVERSAL". The right side of the top bar indicates "Exploration Mode".

The main area shows a graph with 9 nodes (0-8) and several edges. Nodes 0, 1, and 2 are highlighted in light blue, indicating they are part of the current search path. Node 3 is highlighted in orange, indicating it is the current node being processed. Nodes 4, 5, 6, 7, and 8 are in grey, indicating they have not been visited yet.

On the left side, there is a menu with the following options: Draw Graph, Random Graph, Sample Graphs, Directed <-> Undirected, BFS, DFS, Cut Vertex & Bridge, SCC Algorithms, Bipartite Graph check, Topo Sort, and Two-SAT checker. The "DFS" option is selected.

Below the menu, there is a "GO" button and a "0" button, indicating the starting node for the DFS operation.

On the right side, there is a stack of function calls. The top call is "DFS(0)". Below it is "DFS(3)". Below that is "DFS(u)". The stack is currently empty, indicating that the DFS operation has just started at node 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]  $\leftarrow$  1 // to avoid cycle
```

```
    for all v adjacent to u // order of neighbor
```

```
        if visited[v] = 0 // influences DFS
```

```
            p[v]  $\leftarrow$  u // visitation sequence
```

```
            DFSrec(v) // recursive (implicit stack)
```

Recursive  
phase

```
// in the main method
```

```
for all v in V
```

```
    visited[v]  $\leftarrow$  0
```

```
    p[v]  $\leftarrow$  -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:  $O(V+E)$

- Each vertex is only visited once  $O(V)$ , then it is flagged to avoid cycle
- Every time a vertex is visited, all its  $k$  neighbors are scanned; Thus after all vertices are visited, we have examined all  $E$  edges  $\sim O(E) \rightarrow$  assuming that we use **Adjacency List!**
- Overall:  $O(V+E)$

# Path Reconstruction Algorithm (1)

```
// iterative version (will produce reversed output)
```

```
Output "(Reversed) Path:"
```

```
i  $\leftarrow$  t // start from end of path: suppose vertex t
```

```
while i  $\neq$  s
```

```
    Output i
```

```
    i  $\leftarrow$  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}
```

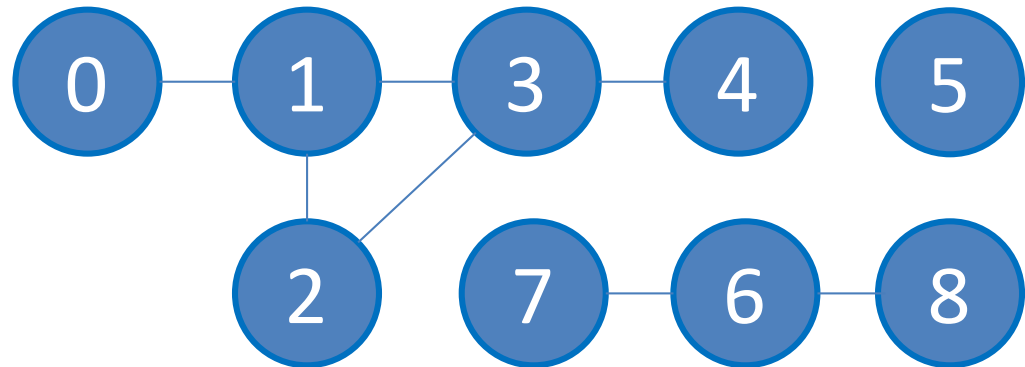
# **SOME GRAPH TRAVERSAL APPLICATIONS**

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

Several stuffs, let's see *some of them*:

- 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 (CP3 4.1, Undirected)

Below, we show vertices that are reachable from vertex 0

**VISUALGO** GRAPH TRAVERSAL Exploration Mode

```
graph LR; 0 ---|red| 1; 1 ---|red| 2; 1 ---|blue| 3; 2 ---|red| 3; 3 ---|red| 4; 4 ---|grey| 5; 5 ---|grey| 6; 6 ---|grey| 7; 7 ---|grey| 8;
```

DFS

0

GO

Draw Graph

Random Graph

Sample Graphs

Directed <-> Undirected

BFS

Cut Vertex & Bridge

SCC Algorithms

Bipartite Graph check

Topo Sort

Two-SAT checker

DFS(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.

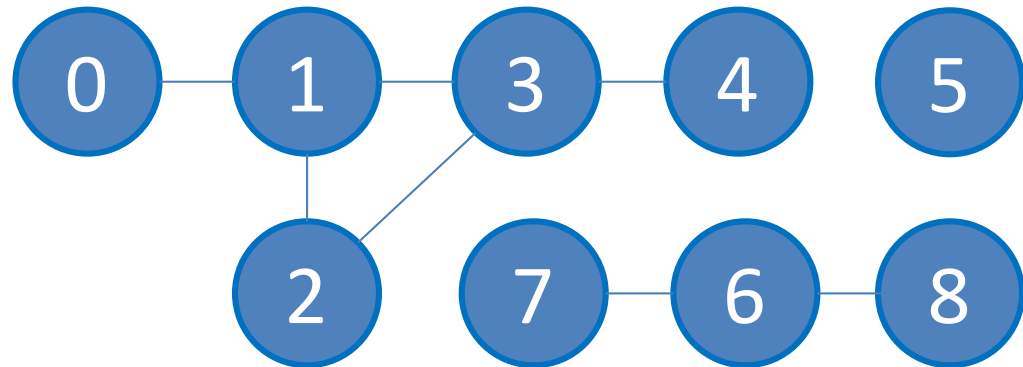
```
DFS(u)
  for each neighbor v of u
    if v has not been visited
      DFS(v)
    else skip v;
```



# What can we do with BFS/DFS? (2)

- Identifying component(s)
  - Component is sub graph in which any 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
  - Solution:

```
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 Components

Ask VisuAlgo to perform various DFS (or BFS) operations on the sample Graph (CP3 4.1, Undirected)

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

**7 VISUALGO** GRAPH TRAVERSAL Exploration Mode ▾

<

Draw Graph

Random Graph

Sample Graphs

Directed <-> Undirected

BFS

DFS

Cut Vertex & Bridge

SCC Algorithms

Bipartite Graph check

Topo Sort

Two-SAT checker

6 GO

DFS(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.

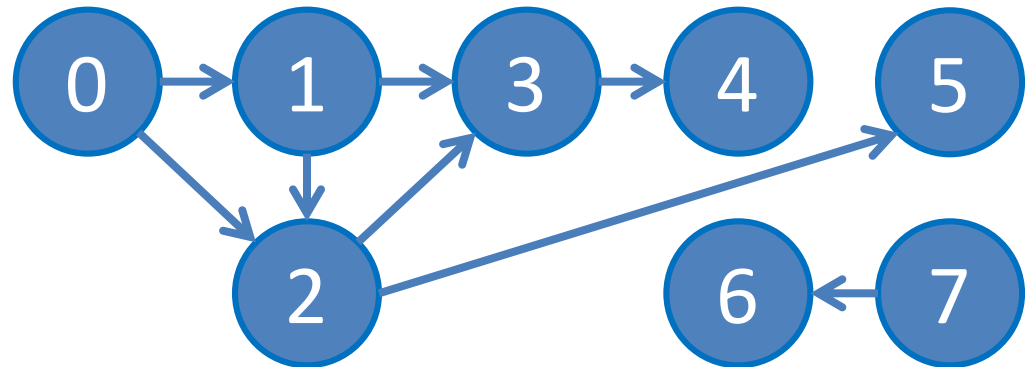
```
DFS(u)
  for each neighbor v of u
    if v has not been visited
      DFS(v)
    else skip v;
```

# What is the time complexity for “counting connected component”?

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  $O(\underline{\hspace{2cm}})$

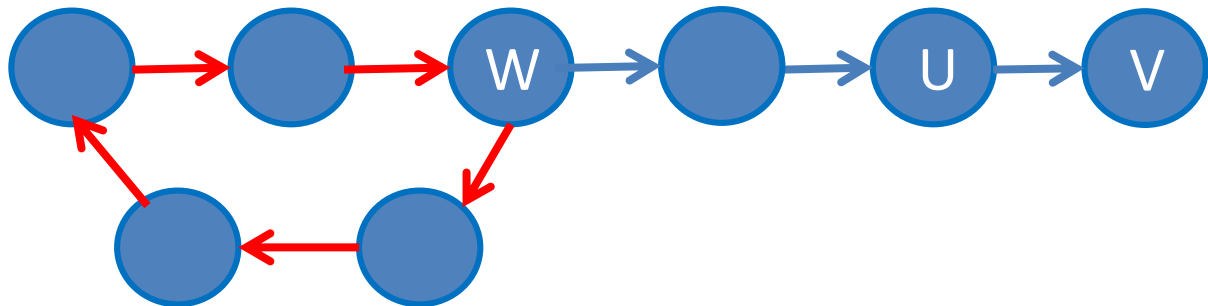
# What can we do with BFS/DFS? (3)

- 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
  - One of the main purpose of finding topological sort: for Dynamic Programming (DP) on DAG (will be discussed a few weeks later...)



# 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 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
  - 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

# What can we do with BFS/DFS? (4)

- Topological Sort
  - If the graph is a DAG, then simply running **DFS** on it (and at the same time record the vertices in “post-order” manner) will 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
  - See pseudo code in the next slide

# DFS for TopoSort – 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(s) // start the recursive call from s
reverse toposort and output it
```



# Topological Sort

Ask VisuAlgo to perform Topo Sort (DFS) operation on the sample Graph (CP3 4.4, Directed)

Below, we show execution of the DFS variant

7 VISUALGO

GRAPH TRAVERSAL

Exploration Mode

```
graph LR; 0((0)) --> 1((1)); 0((0)) --> 2((2)); 1((1)) --> 3((3)); 1((1)) --> 2((2)); 2((2)) --> 3((3)); 2((2)) --> 5((5)); 3((3)) --> 4((4)); 4((4)) --> 5((5)); 6((6)) --> 7((7));
```

Draw Graph

Random Graph

Sample Graphs

Directed <-> Undirected

BFS

DFS

Cut Vertex & Bridge

SCC Algorithms

Bipartite Graph check

Topo Sort

Two-SAT checker

DFS

BFS

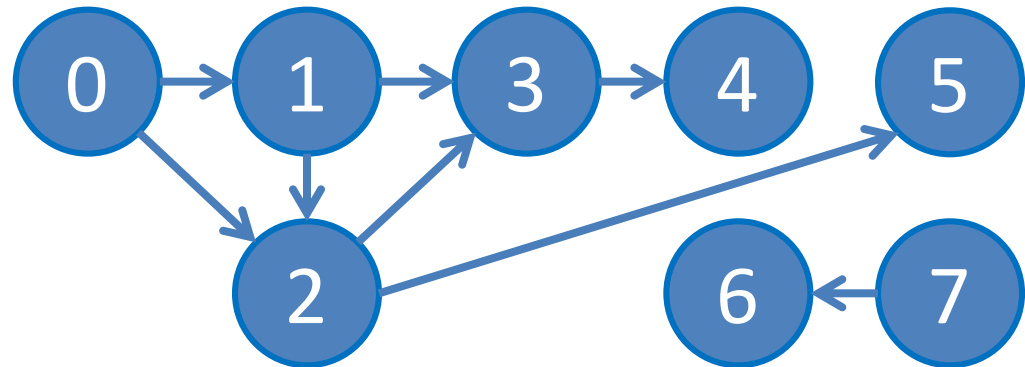
Topological Sort

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
```

# What can we do with BFS/DFS? (5)

- Topological Sort
  - 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]



# Trade-Off

## **$O(V+E)$ DFS**

- Pros:
  - Slightly easier? to code (this one depends)
  - Use less memory
- Cons:
  - Cannot solve SSSP on unweighted graphs

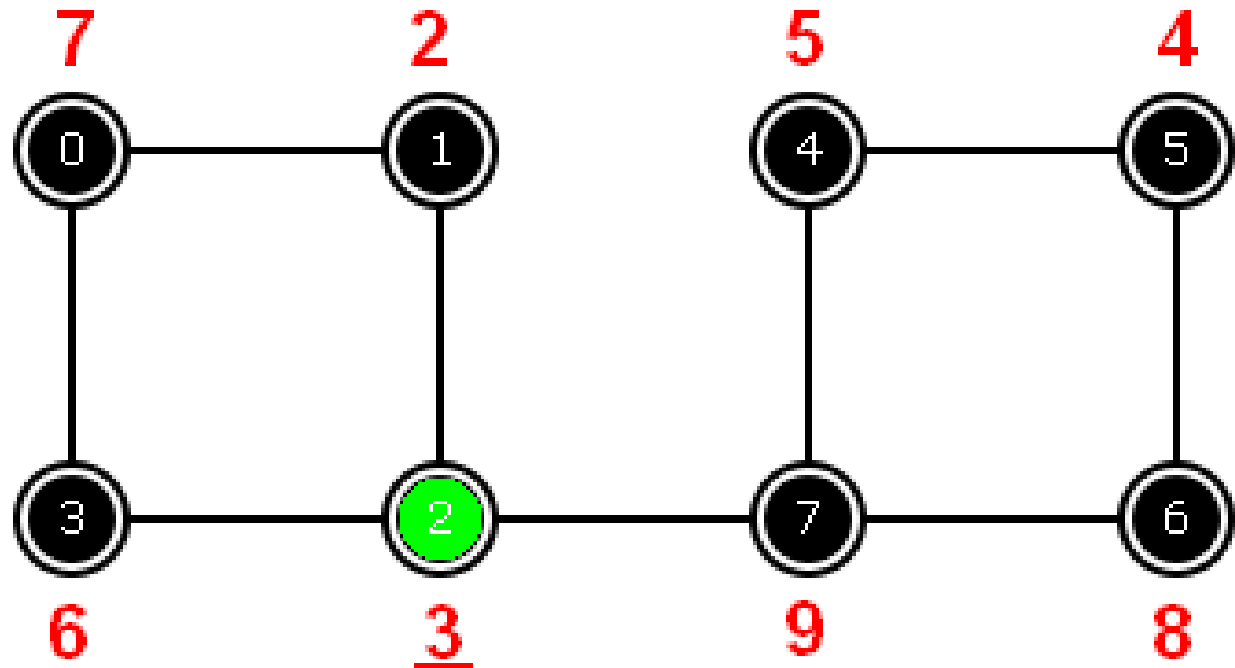
## **$O(V+E)$ BFS**

- Pros:
  - Can solve SSSP on unweighted graphs (revisited in latter lectures)
- Cons:
  - Slightly longer? to code (this one depends)
  - Use more memory (especially for the queue)

# Hospital Renovation Problem (PS3) – open next Wednesday 12 noon

Given a layout of a hospital...

- Determine which room(s) is/are the ‘important room(s)’ that have the potential to be renovated
- Among those room(s), pick one with the lowest rating score to renovate



# Summary

In this lecture, we have looked at:

- Some applications of Graph Data Structures
  - Continuation from Lecture 05
- 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, CC, Toposort