CS2010 – Data Structures and Algorithms II

Lecture 06 – Maze Exploration

chongket@comp.nus.edu.sg

NUS
National University of Singapore
School of Computing

Admin Stuff (1)

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

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($\mathbf{V}^2$) space

Adjacency List

Pros:

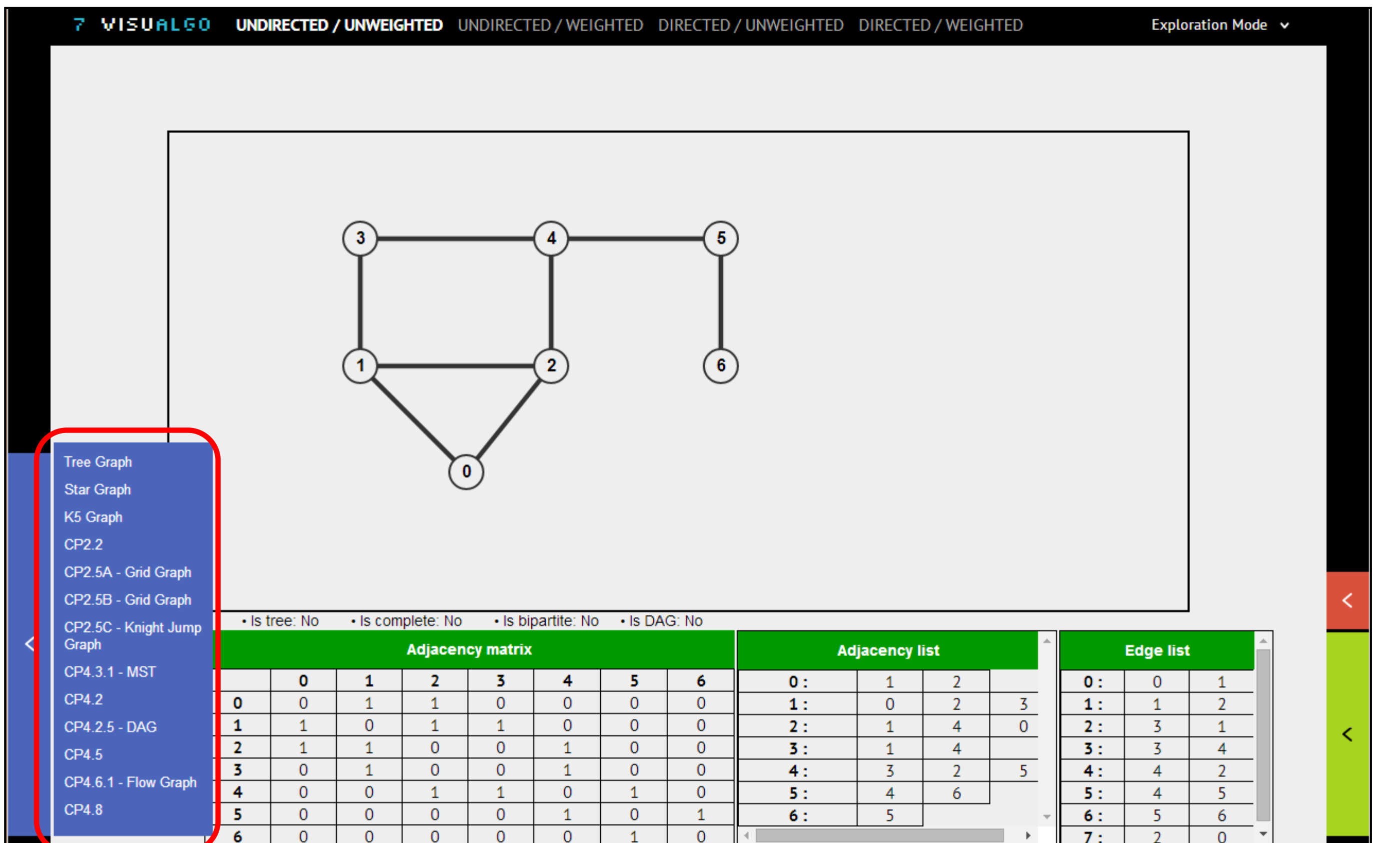
- O(**k**) to enumerate k neighbors of a vertex
- Good for sparse graph/Dijkstra's/ DFS/BFS, O(**V**+**E**) space

Cons:

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

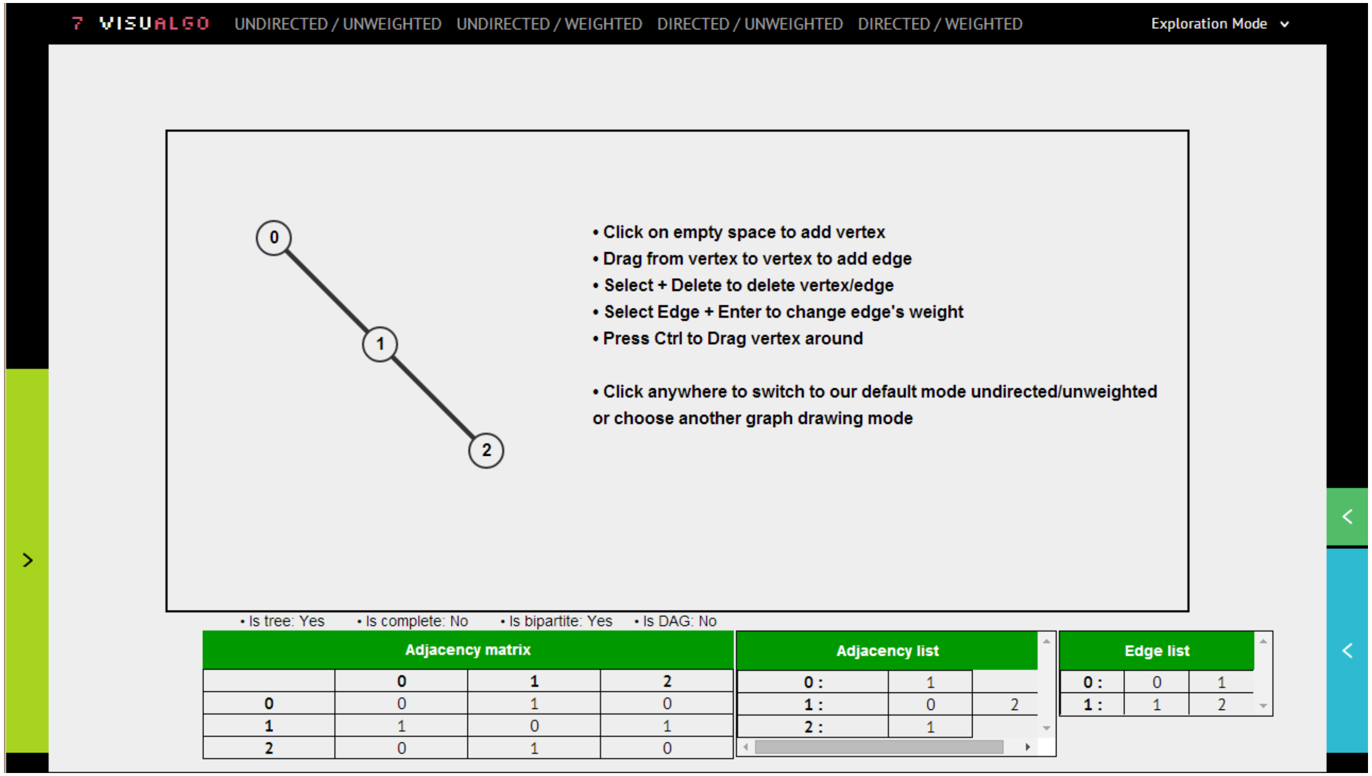
VisuAlgo Graph DS Exploration (1)

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



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



Adjacency matrix

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

Adjacency list

0 :	1	
1 :	0	2
2 :	1	

Edge list

0 :	0	1
1 :	1	2

GRAPH TRAVERSAL ALGORITHMS

Review – **Binary Tree** Traversal

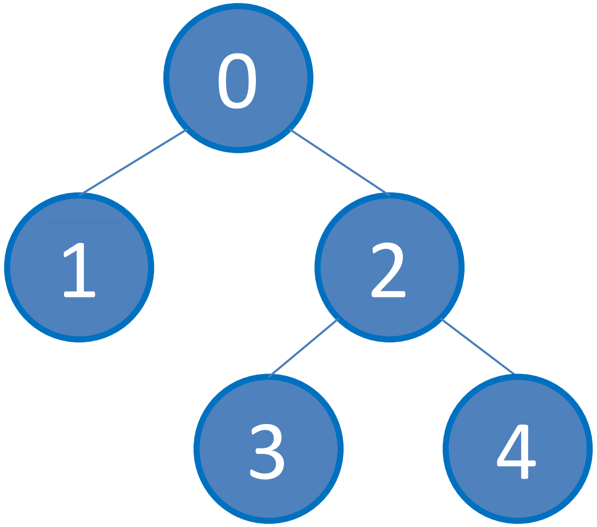
In a binary tree, there are three standard traversal:

- 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);`

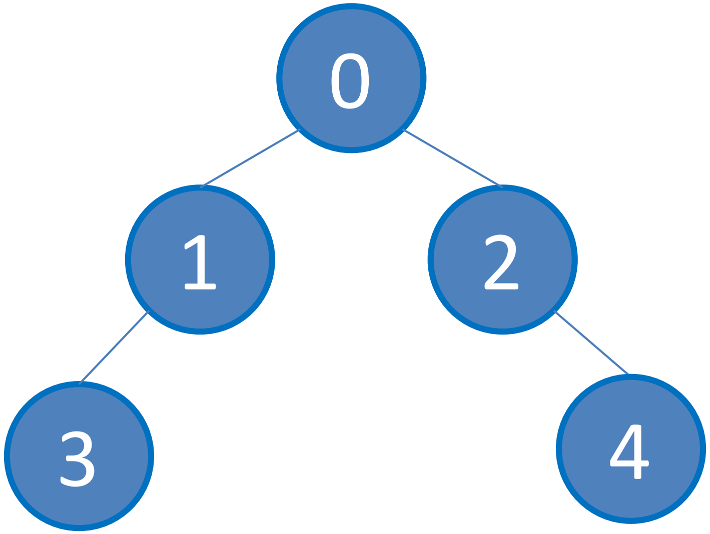
We start binary tree traversal from root:

- 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 **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 (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)

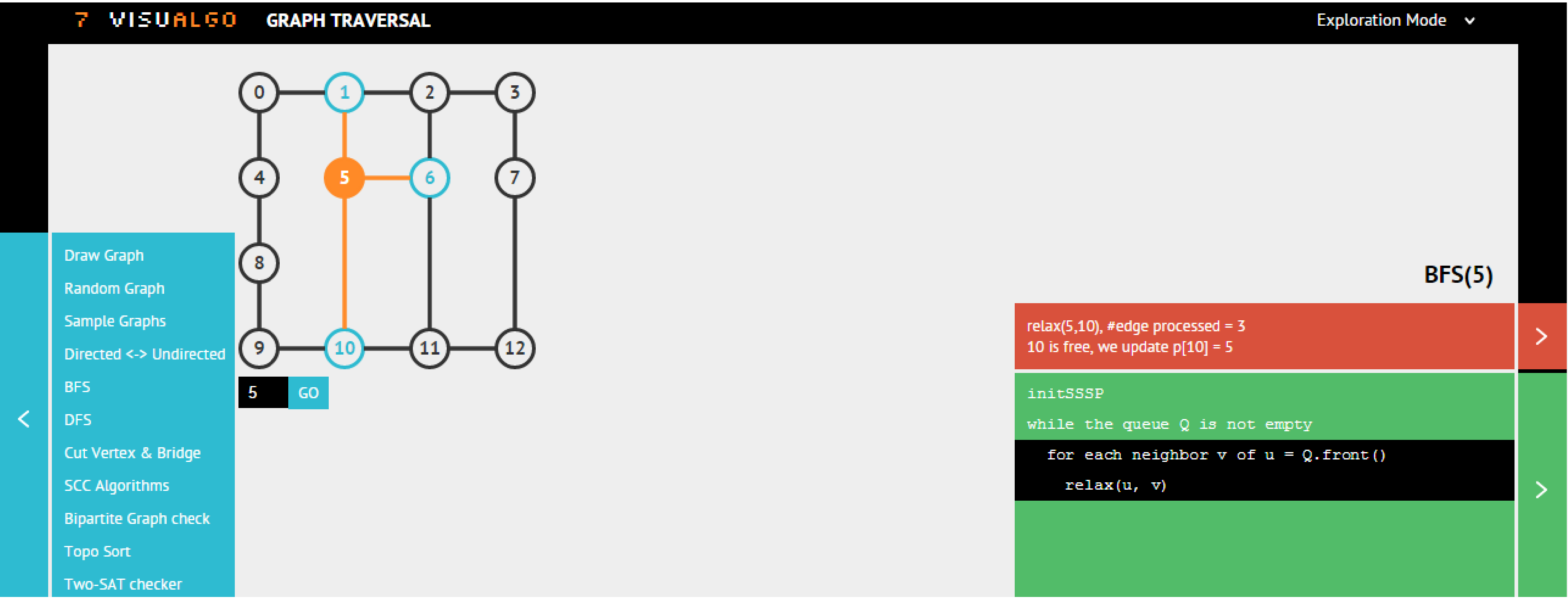
Breadth First Search (BFS) – Ideas

- 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 **p**redecessor (or **p**arent) 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)**



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
```

Initialization phase

Main loop

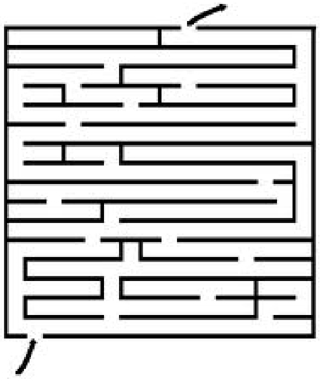
BFS Analysis

```
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 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 ~ O(**V**)
- Every time a vertex is dequeued, all its **k** neighbors are scanned; After all vertices are dequeued, all **E** edges are examined ~ O(**E**) → assuming that we use **Adjacency List**!
- Overall: O(**V**+**E**)

Depth First Search (DFS) – Ideas

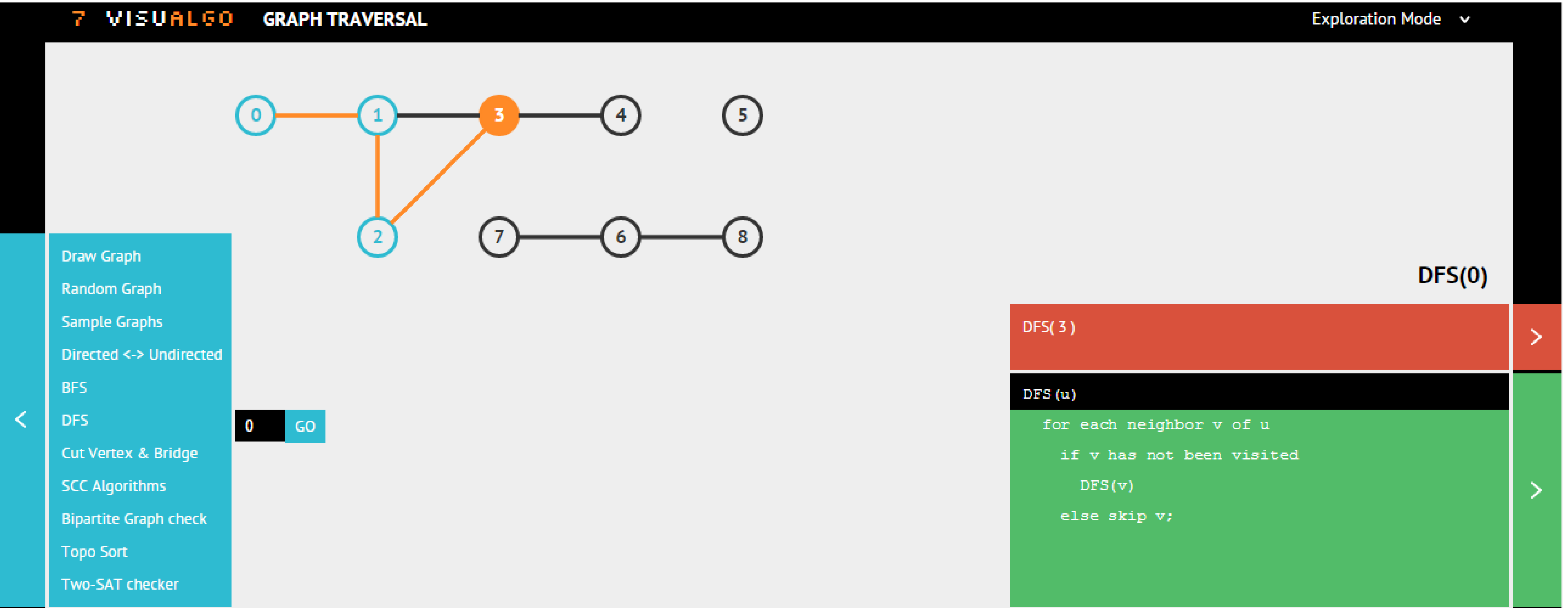


- Start from **s**
- DFS visits vertices of G in *depth-first* manner (when viewed from source vertex s)
  - Q: How to maintain such order?
    - A: Stack **S**, but we will simply use recursion (an implicit stack)
  - Q: How to differentiate visited vs unvisited vertices (to avoid cycle)?
    - A: 1D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



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**)

- 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 ~ O(**E**) → assuming that 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}
```

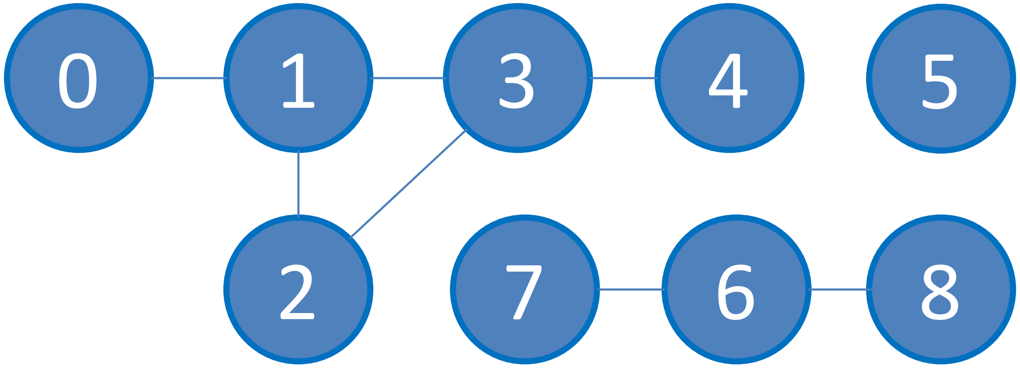
SOME GRAPH TRAVERSAL APPLICATIONS

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

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**

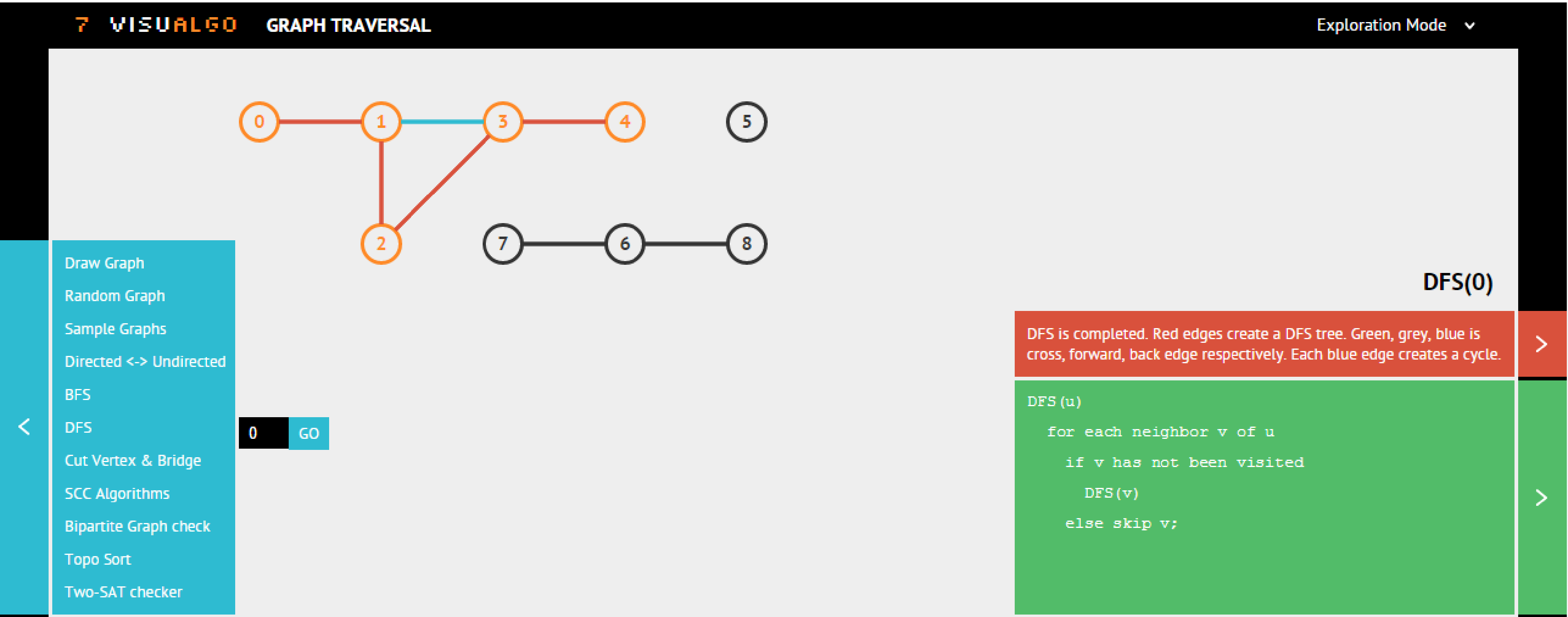
```
BFS(u) // DFSrec(u)
if visited[v] == 1
  Output "Yes"
else
  Output "No"
```



Reachability Test

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

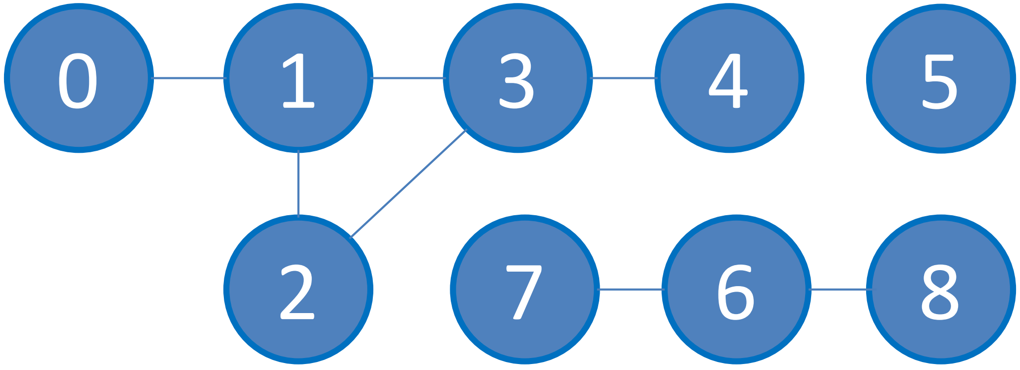
Below, we show vertices that are reachable from vertex 0



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:

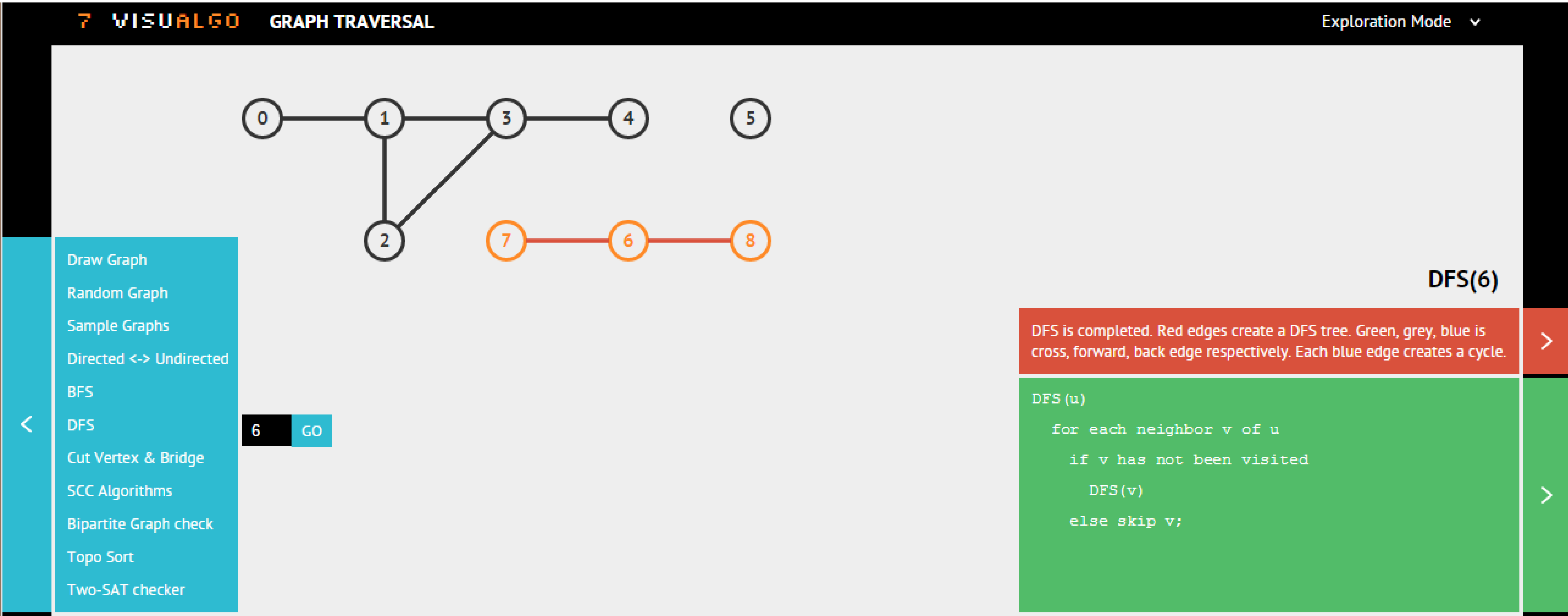
```
CC ← 0
for all v in V
  visited[v] ← 0
for all v in V // O(V)?
  if visited[v] == 0
    CC ← CC + 1
    DFSrec(v)//O(V+E)?
    // BFS from v
    // is also OK
```



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)**

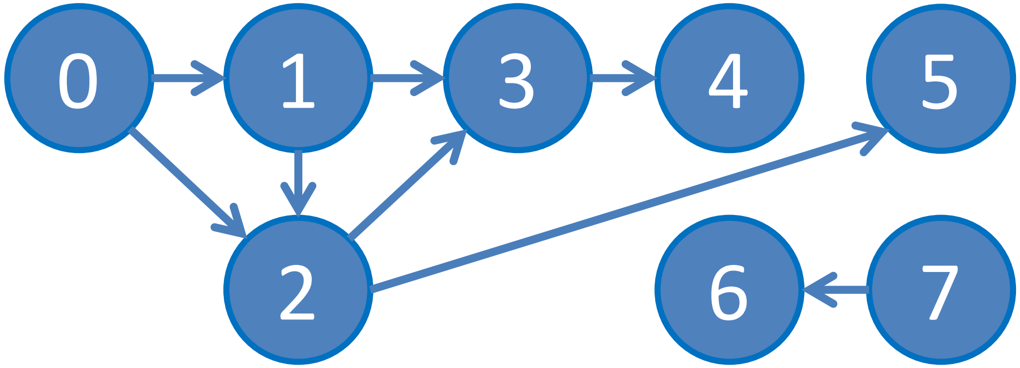


What is the time complexity for "counting connected component"?

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(______)

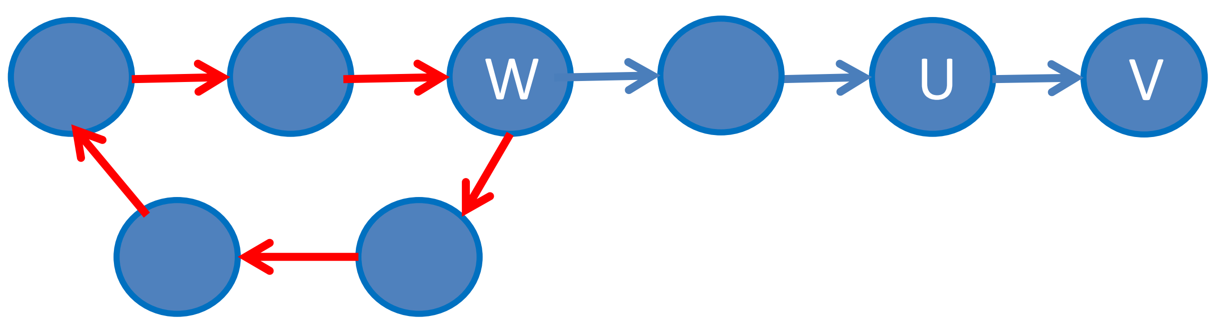
What can we do with BFS/DFS? (3)

- 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 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

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

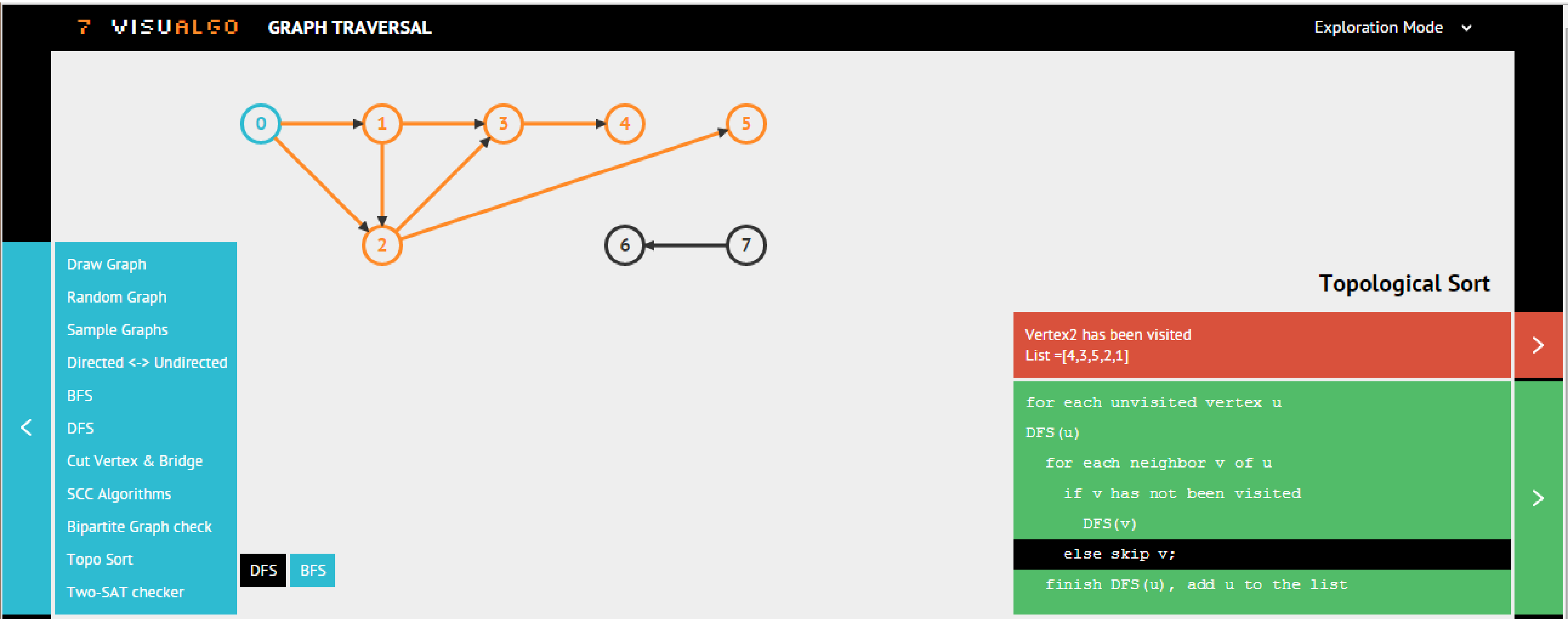
Simply look at the codes in red/underlined

```
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

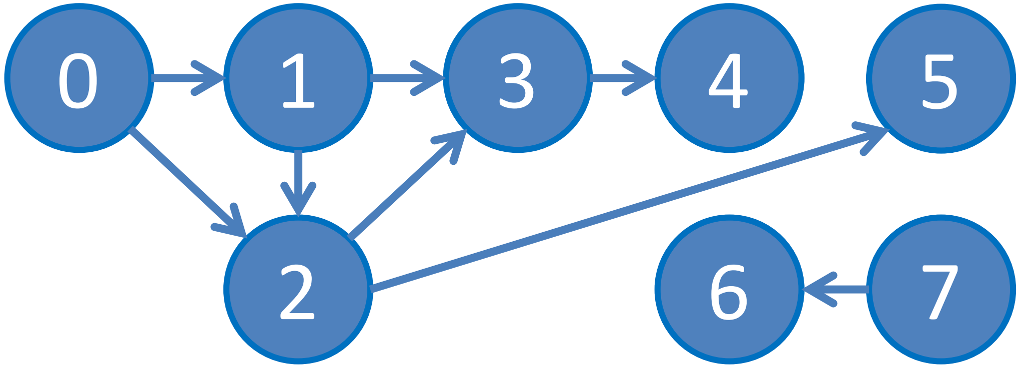
Ask VisuAlgo to perform Topo Sort (DFS) operation on the sample Graph (CP3 4.4, Directed)

Below, we show execution of the DFS variant



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

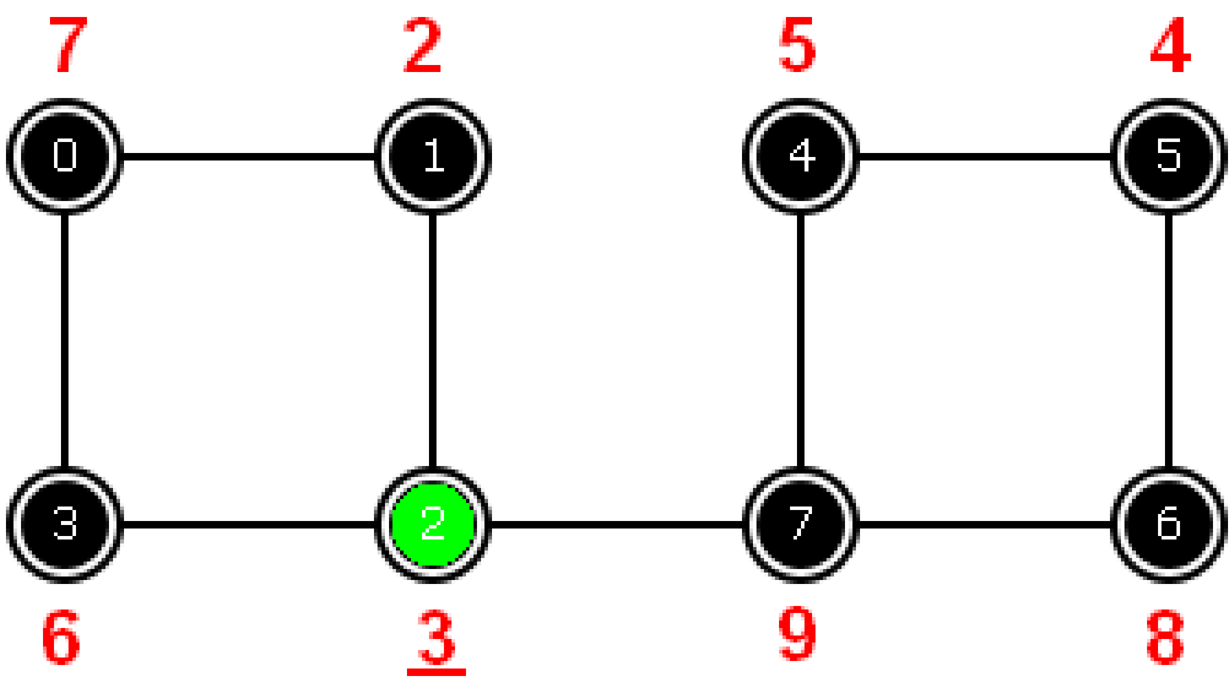
O(V+E) BFS

- 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