

CS2040 – Data Structures and Algorithms

Lecture 13 – Maze Exploration ~ Graph Traversal

chongket@comp.nus.edu.sg



Annoucement

- PE on Saturday 6th April
 - Will be **harder** then SIL 1&2
 - There will be **4 problems**
 - Topics tested will be as follows
- Q1. Linear Data Structure Applications
- Applications of LinkedList, Stack and Queue
- Q2. Non-Linear Data Structure Applications
- Applications of HashSet/HashMap, TreeSet/TreeMap and PriorityQueue
- Q3. Graph Representation and Traversals Applications
- Applications of DFS and BFS
- Q4. Mix and Match Applications
- Applications of any topics covered up to Week 11 Lab Session and Week 10 Lectures (Thursday/Friday). Multiple topics can be combined.

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/dfsbfbs>

Reference: Mostly from CP3 Section 4.2

- Not all sections in CP3 chapter 4 are used in CS2040!
 - Some are quite advanced :O

GRAPH TRAVERSAL ALGORITHMS

Review – Binary Tree Traversal

In a binary tree, there are three standard traversal:

- Preorder
- **Inorder**
- Postorder

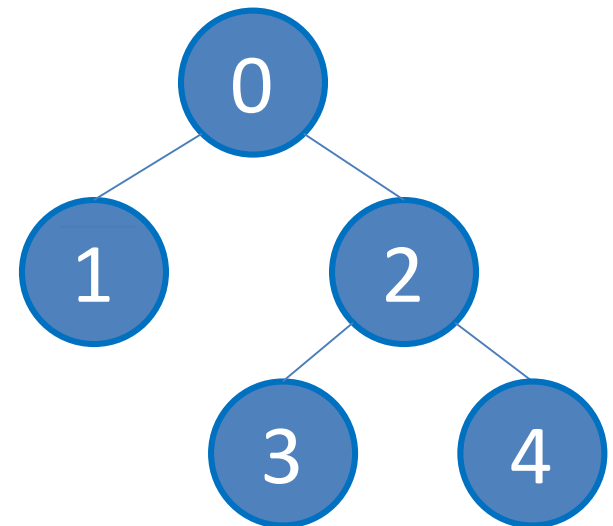
```
pre(u)
  visit(u);
  pre(u->left);
  pre(u->right);
```

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

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

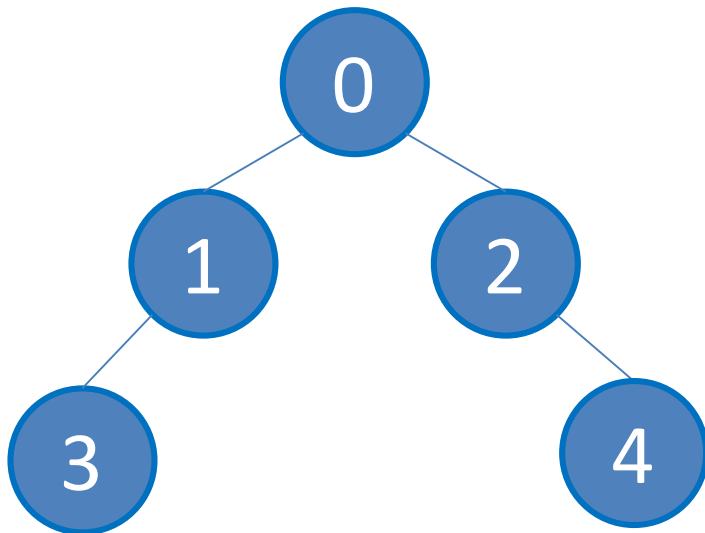
We start binary tree traversal from root:

- 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
- 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 (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 main window shows a 4x4 grid graph with 13 vertices (0-12) and 16 edges. Vertex 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, DFS, Cut Vertex & Bridge, SCC Algorithms, Bipartite Graph check, Topo Sort, and Two-SAT checker. The 'BFS' option is selected. Below the menu, the input '5' is entered in a black box, and a 'GO' button is visible. The right panel shows the execution of BFS(5) with a red status bar indicating 'relax(5,10), #edge processed = 3' and '10 is free, we update p[10] = 5'. Below this, a green box contains 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)'. The top right corner of the interface shows 'Exploration Mode' with a dropdown arrow.

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)
```

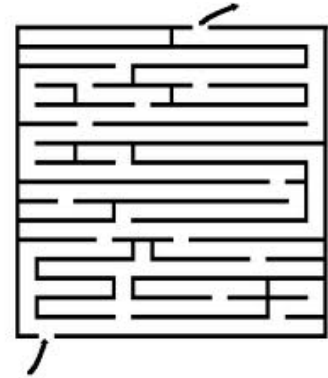
```
// we can then use information stored in visited/p
```

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
- 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 (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. The edges are: (0,1), (1,2), (1,3), (3,4), (3,2), (7,6), (6,8), and (5,4).

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.

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 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] ← 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 $\sim O(E) \rightarrow$ 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}
```

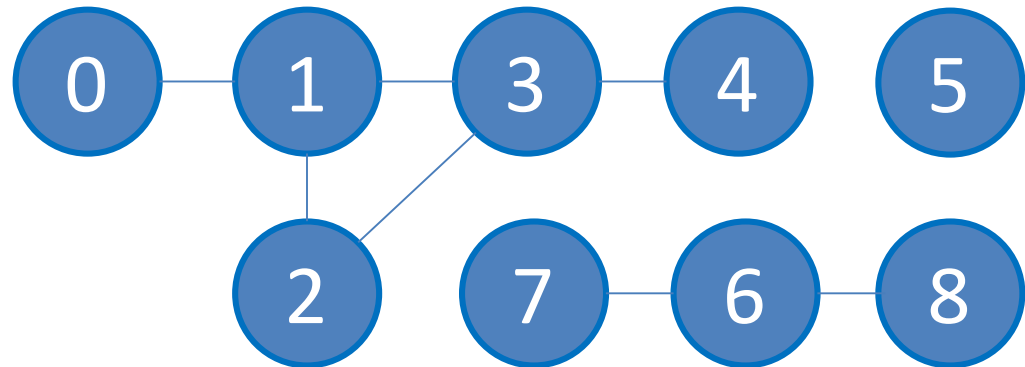
SOME GRAPH TRAVERSAL APPLICATIONS

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

7 VISUALGO GRAPH TRAVERSAL

Exploration Mode



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)

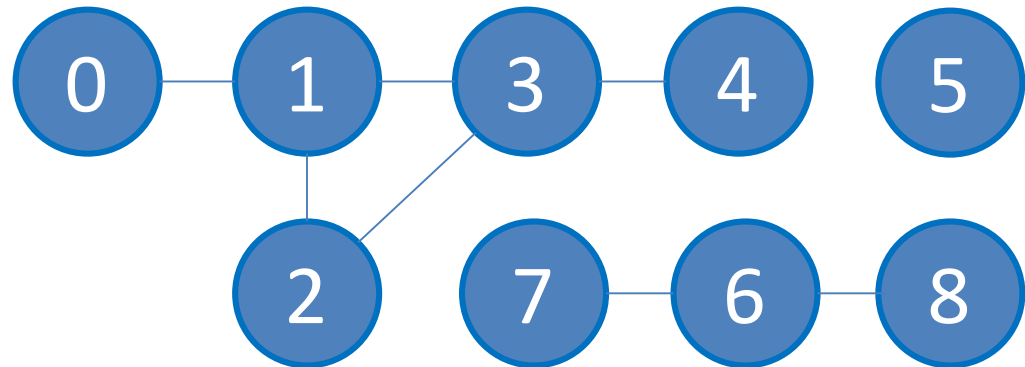
- Find Shortest Path between 2 vertices in an unweighted graph/graph where edges have same weight
 - When the graph is **unweighted*/edges have same weight**, shortest path between any 2 vertices **u,v** is finding the **least number of edges** traversed from u to v
 - The $O(V+E)$ Breadth First Search (BFS) traversal algorithm precisely measures this
 - Run BFS from u as source
 - Construct shortest path from u to v from **p** after BFS finishes
 - Cost of shortest path from u to v is
(number of edges in the path) \times (edge weight for weighted edges)

* Can treat the edge weight as 1

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

- 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

```
graph LR; 0 --- 1; 1 --- 2; 1 --- 3; 2 --- 3; 3 --- 4; 5 --- 6; 6 --- 7; 7 --- 8;
```

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;
```

Draw Graph

Random Graph

Sample Graphs

Directed <-> Undirected

BFS

DFS

Cut Vertex & Bridge

SCC Algorithms

Bipartite Graph check

Topo Sort

Two-SAT checker

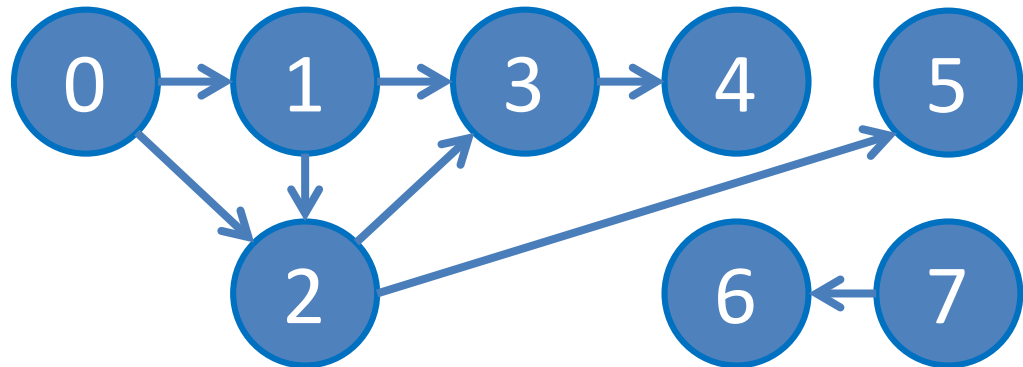
6 GO

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

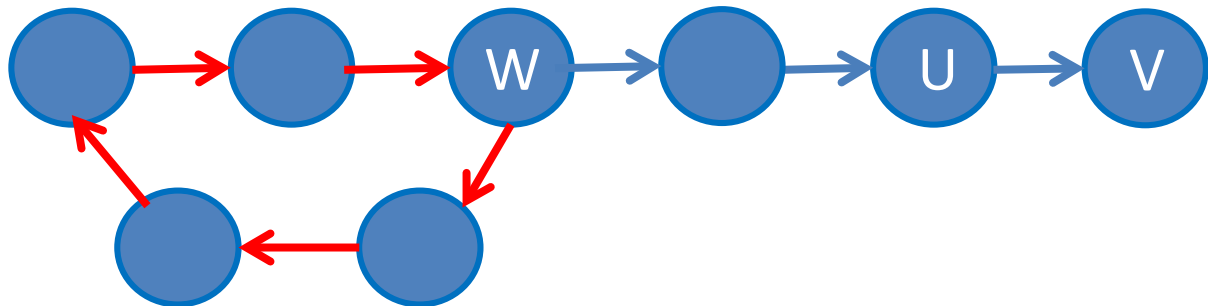
What can we do with BFS/DFS? (4)

- Topological Sort
 - Topological sort of a DAG is a linear ordering of its vertices in which each vertex comes before all vertices to which it has outbound edges
 - 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 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
- This constructive proof is the basis for the BFS based algorithm (Khan's algorithm) to compute topological ordering of a DAG

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

- Topological Sort – Khan's algorithm
 - If graph is a DAG, then running a modified version of BFS (Khan'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 visited
 - See pseudo code in the next slide

Khan'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'
```

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

Main loop

Output toposort as the topological order

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

- 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
 - See pseudo code in the next slide

DFS for TopoSort – Pseudo Code

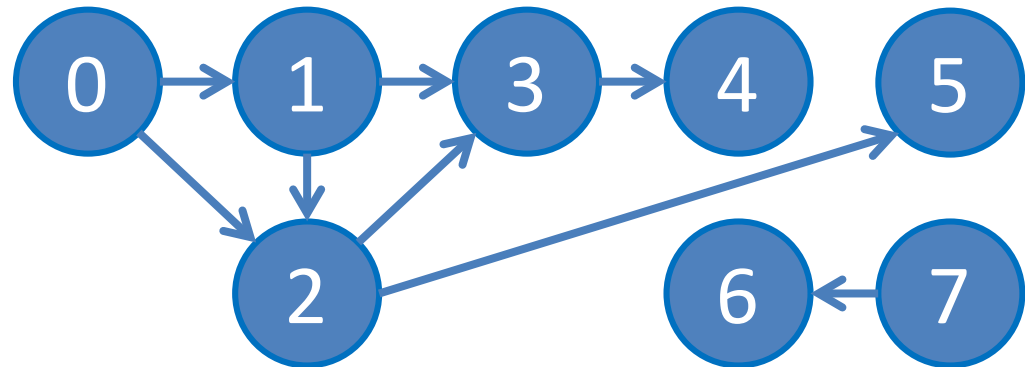
Simply look at the codes in red/underlined

```
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
```

What can we do with BFS/DFS? (6)

- Topological Sort with DFS
 - 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 (BFS/DFS) operation on the sample Graph (CP3 4.4, Directed)

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)); style 0 fill:#add8e6; style 1 fill:#ffa500; style 2 fill:#ffa500; style 3 fill:#ffa500; style 4 fill:#ffa500; style 5 fill:#ffa500; style 6 fill:#ccc; style 7 fill:#ccc;
```

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
```

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)

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, CC, Toposort