

# Breadth-first Search & Depth-first Search

For use in CSE6010 only

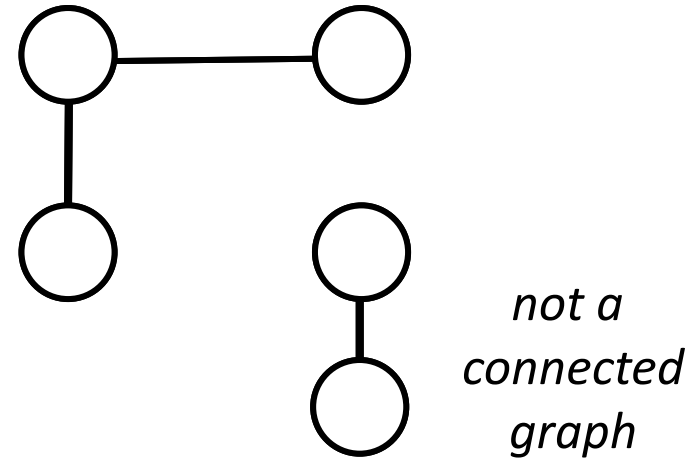
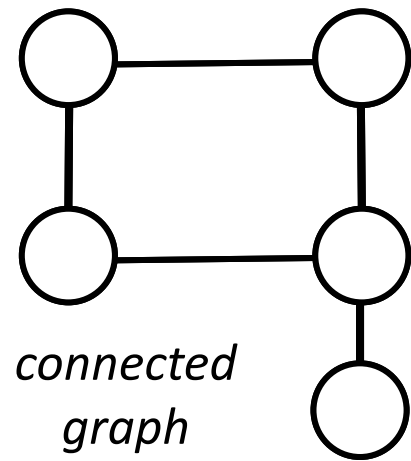
Not for distribution

# Learning Goals

- Explain the difference between breadth-first and depth-first search algorithms on graphs.
- Perform breadth-first and depth-first search algorithms on a given graph.
- Express the execution time scaling of BFS and DFS in terms of the numbers of vertices and edges in a graph.
- Define a topological sort and the graph conditions necessary to guarantee it exists.
- Use depth-first search for topological sorting.

# Statement of Motivating Problem

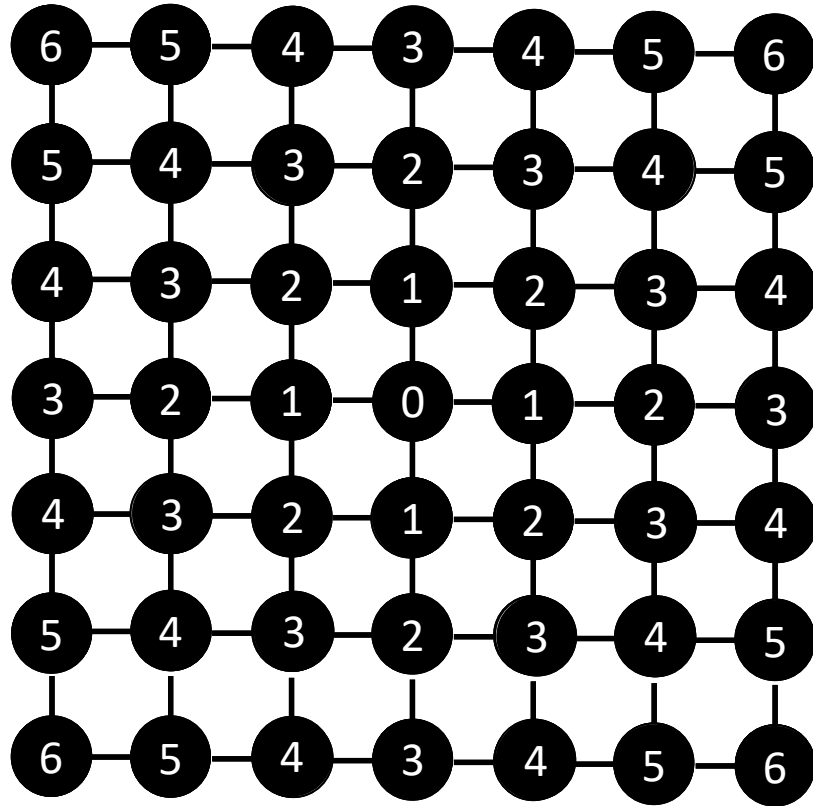
Definition: A *connected* undirected graph is one where there exists a path between every pair of vertices



Problem: Given a connected undirected graph  $G$  (vertex set  $V$ , edge set  $E$ ) and a source node  $S$

- Visit every node in the graph starting from  $S$
- Find the minimum distance from  $S$  to every other vertex assuming edges have unit weight.

# Basic Idea



**Done!**

Note: traditionally, color is used to differentiate vertices we've visited already from those we have not visited yet (alt: condition)

- Initially white (unvis)
- Gray (vis) when a vertex is first visited to “mark” it, but its neighbors have not yet been explored
- Black (done) when done exploring neighbors of vertex

- Starting from S, mark all vertices 1 hop from S (neighbors of S)
- Mark all vertices 2 hops from S
  - The “2 hop” vertices are all unmarked neighbors of the “1 hop” vertices
  - So... mark neighbors of “1 hop vertices” that are not already marked as “2 hop vertices”
- Mark all vertices 3 hops from S by marking those neighbors of “2 hop” vertices that have not been marked yet, ...

# Breadth-first Algorithm

Vertices numbered 0, 1, 2, ..., N-1

$D[i]$  = number of hops (distance) from S to i

$C[i]$  = color of vertex i differentiates three “categories” of nodes:

- White/unvis: vertex has not been visited (marked) yet
- Gray/vis: vertex has been visited and its distance computed, but its neighbor vertices still need to be examined
- Black/done: done with vertex (neighbors examined)

## Initially

- Source node S:  $D[S]=0$ ;  $C[S]=\text{vis}$  // gray
- $C[i] = \text{unvis}$ ,  $D[i]$  undefined for all  $i \neq S$

# Breadth-first Algorithm (cont.)

Introduce FIFO queue

- Holds all vis vertices (still to be processed)
- Initially holds only source vertex S

## Algorithm

While (FIFO queue not empty)

$V$  = Next vertex from FIFO queue (remove it from queue)

    For each neighbor  $i$  of  $V$

        If ( $C[i] == \text{unvis}$ )

$D[i] = D[V] + 1$

$C[i] = \text{vis}$

            Add  $i$  to FIFO queue

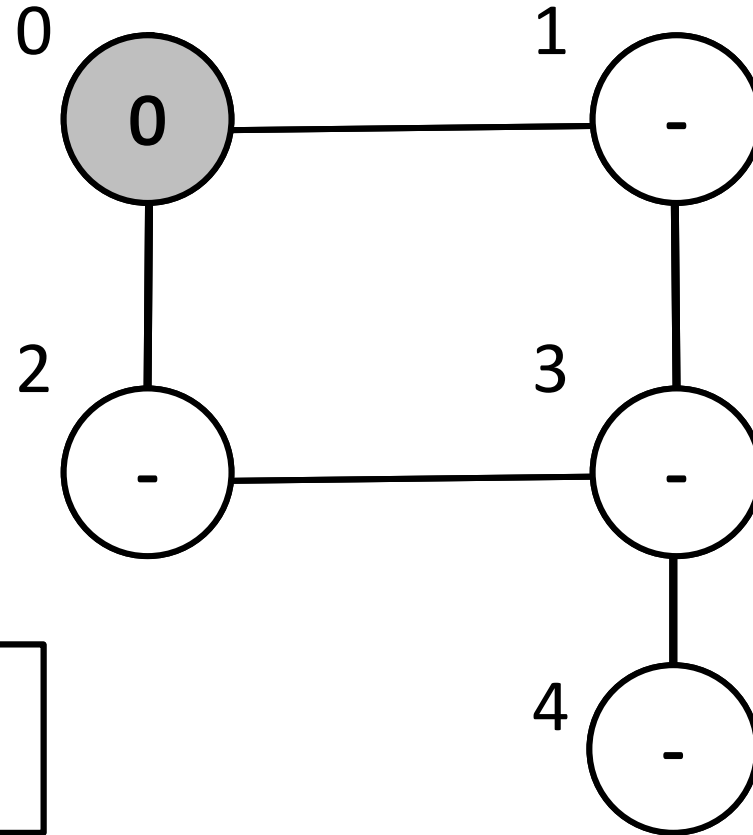
$C[V] = \text{done}$



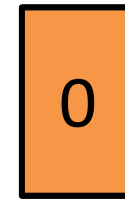
*Why do we only need to worry about neighbors of  $V$  that are unvisited?*

# Example

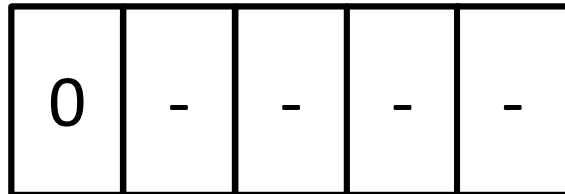
$S=0$



FIFO Queue



D:

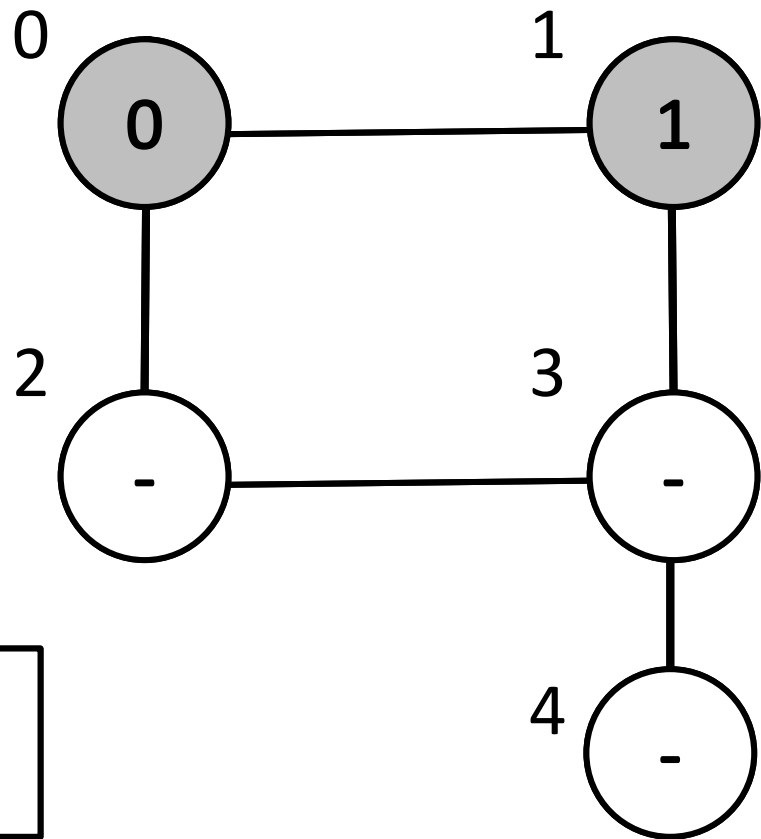


C:



# Example

S=0



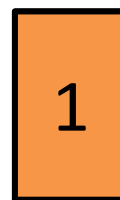
D:

0	1	-	-	-
---	---	---	---	---

C:

--	--	--	--	--

FIFO Queue



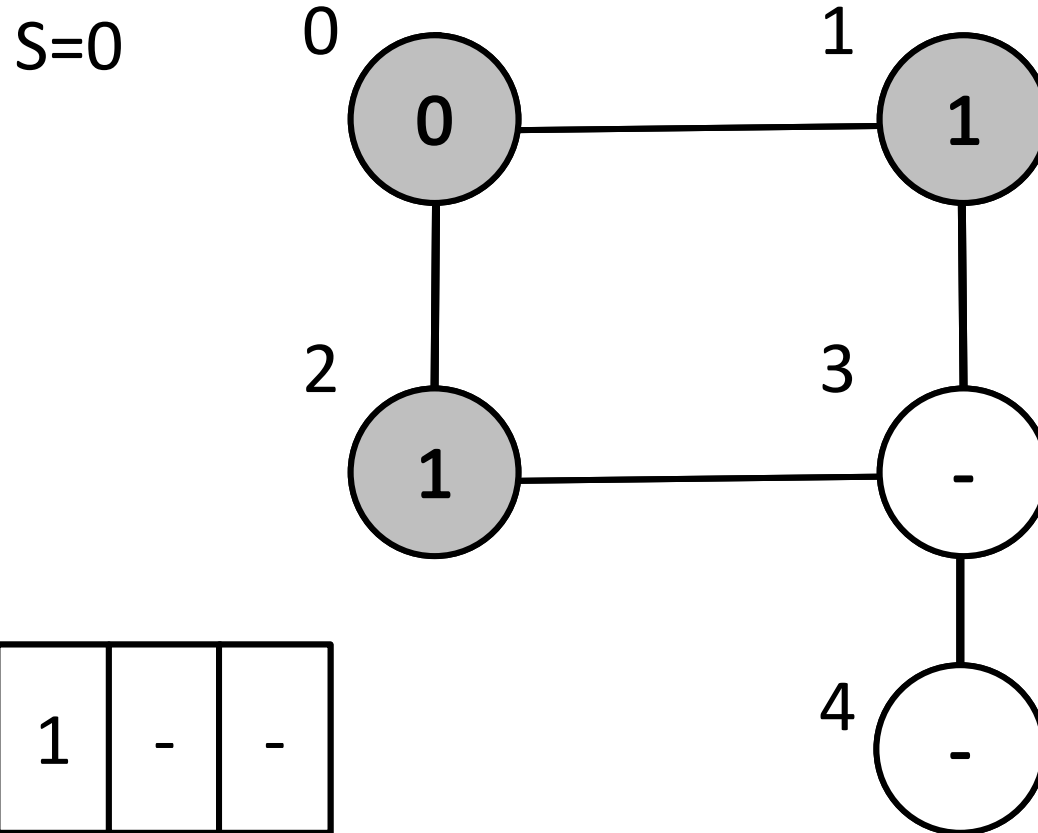
Process V=**0** (remove fr/Q):

Neighbor i=1

- D[1]=1
- C[1]=vis
- Add 1 to queue



# Example



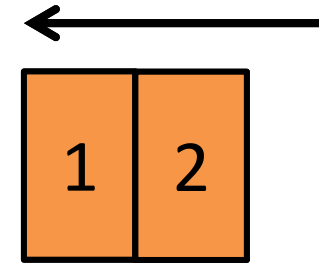
D:

0	1	1	-	-
---	---	---	---	---

C:

--	--	--	--	--

FIFO Queue



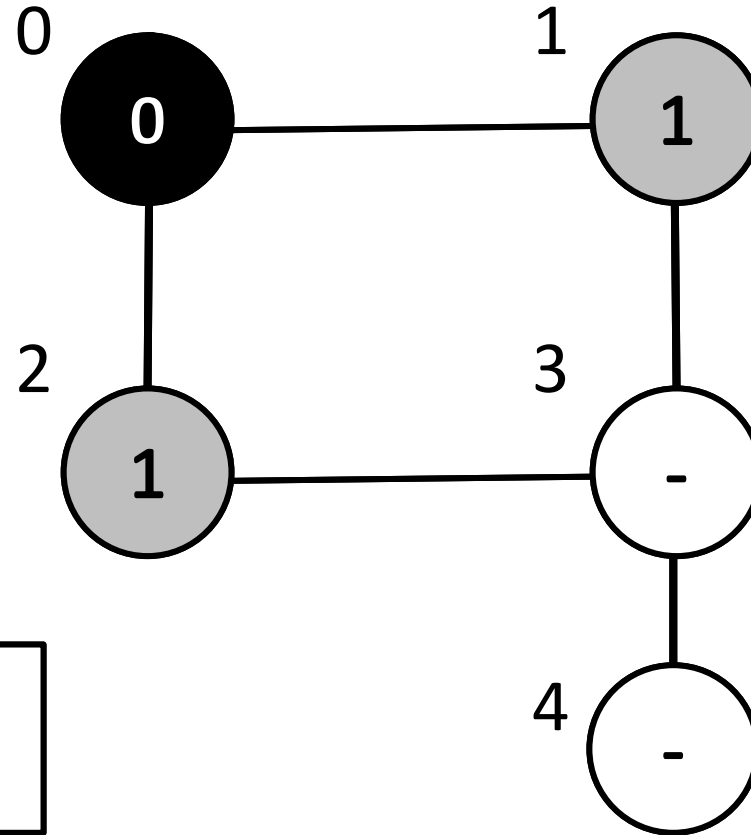
Process  $V=0$ :

Neighbor  $i=2$

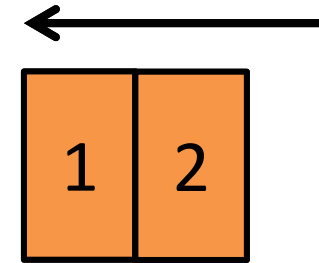
- $D[2]=1$
- $C[2]=vis$
- Add 2 to queue

# Example

$S=0$



FIFO Queue



Process  $V=0$ :

- $C[0]=\text{done}$

D:

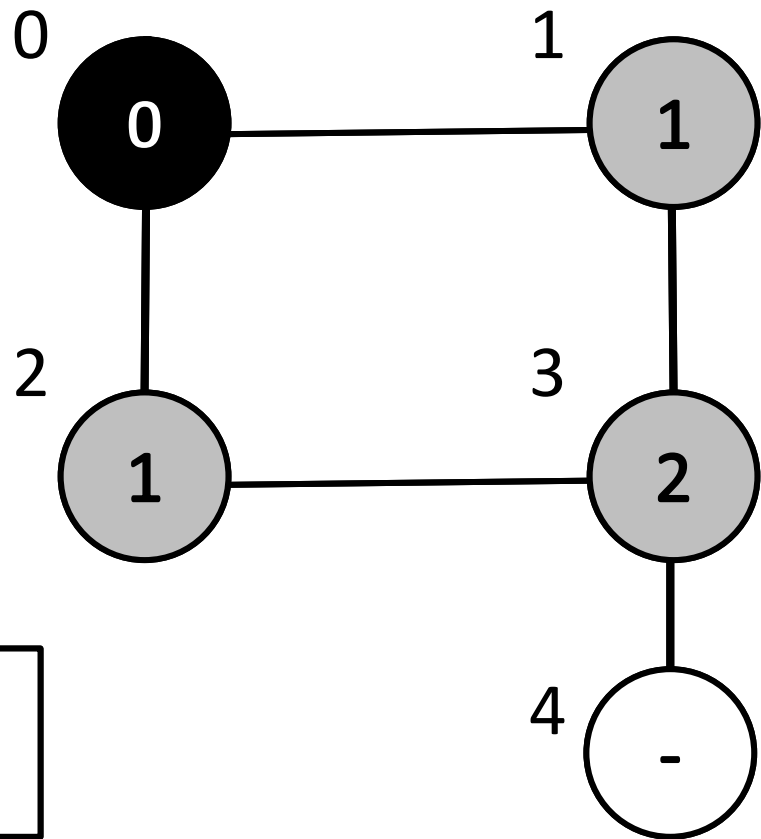
0	1	1	-	-
---	---	---	---	---

C:

--	--	--	--	--

# Example

S=0



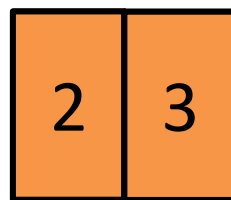
D:

0	1	1	2	-
---	---	---	---	---

C:

--	--	--	--	--

FIFO Queue



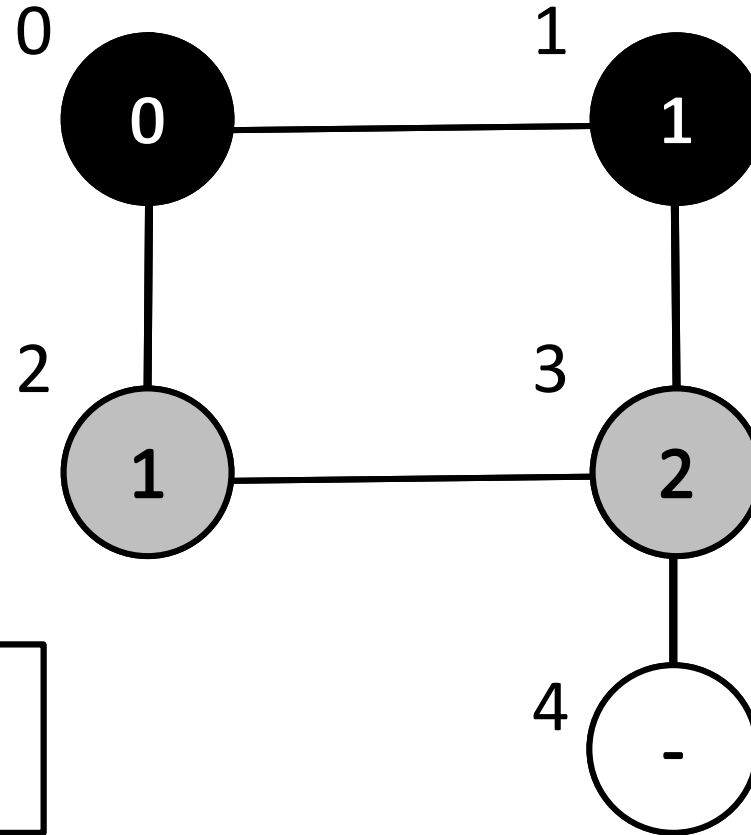
Process V=**1** (remove fr/Q):

Neighbor i=3

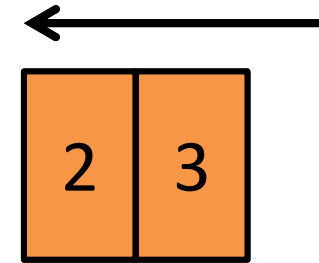
- D[3]=2
- C[3]=vis
- Add 3 to queue

# Example

$S=0$



FIFO Queue



Process  $V=1$ :

- $C[1]=\text{done}$

D:

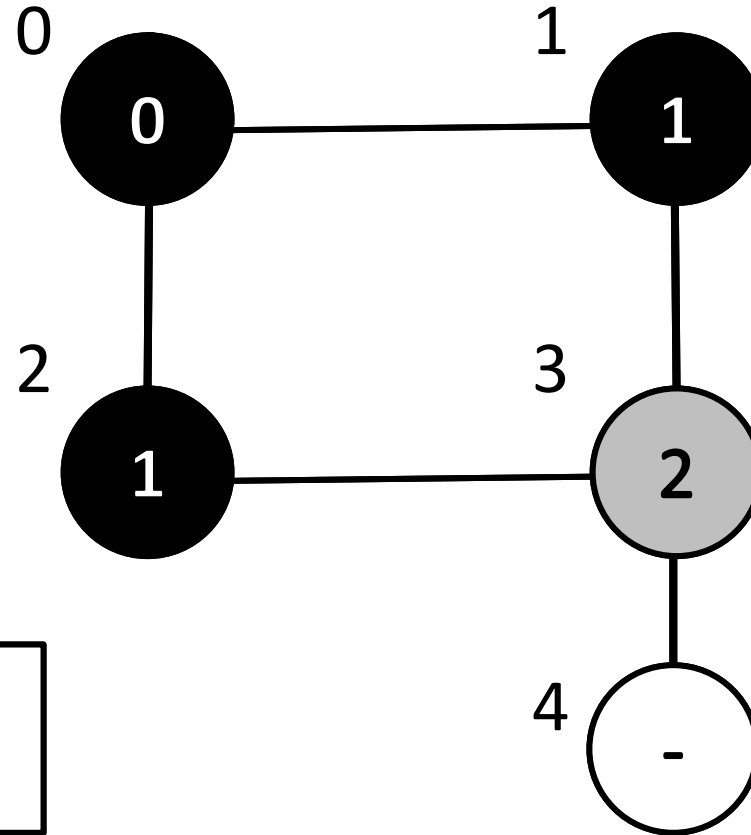
0	1	1	2	-
---	---	---	---	---

C:

--	--	--	--	--

# Example

S=0



D:

0	1	1	2	-
---	---	---	---	---

C:

--	--	--	--	--

FIFO Queue

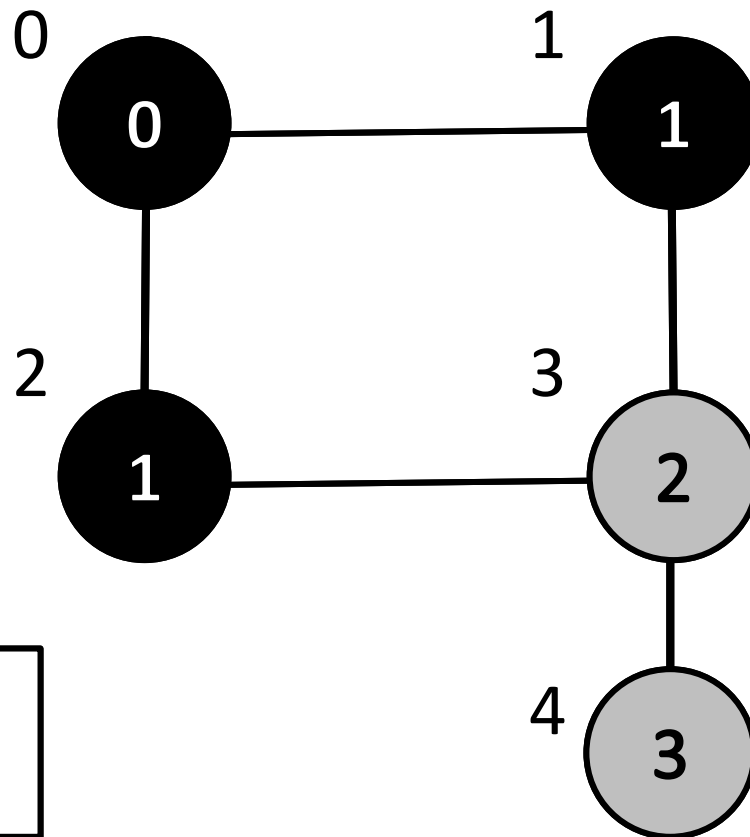


Process V=**2** (remove fr/Q):

- C[2]=done

# Example

S=0



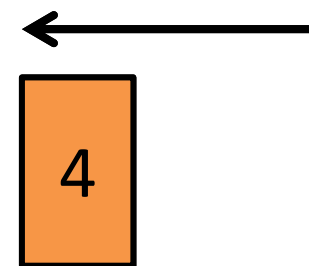
D:

0	1	1	2	3
---	---	---	---	---

C:

--	--	--	--	--

FIFO Queue

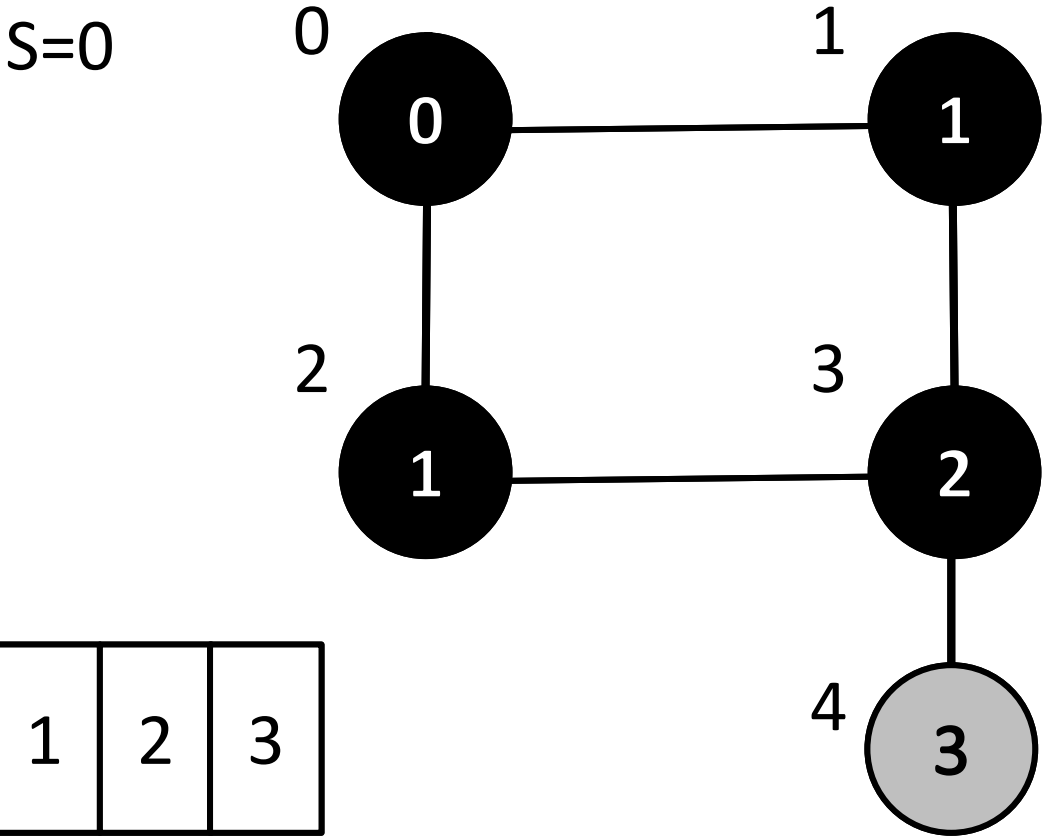


Process V=**3** (remove fr/Q):

Neighbor i=4

- D[4]=3
- C[4]=vis
- Add 4 to queue

# Example



**D:**

0

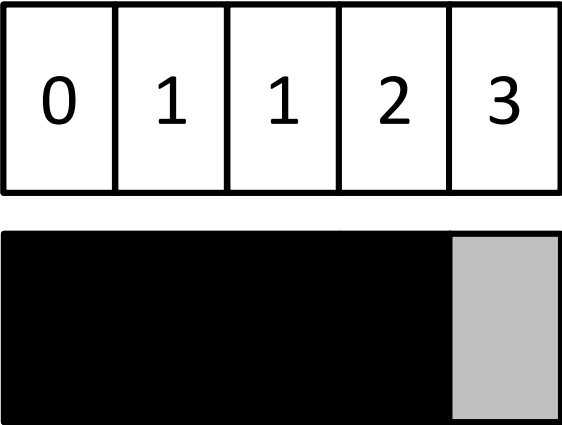
1

1

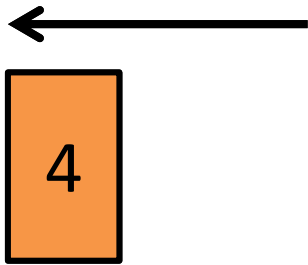
2

3

C:



# FIFO Queue

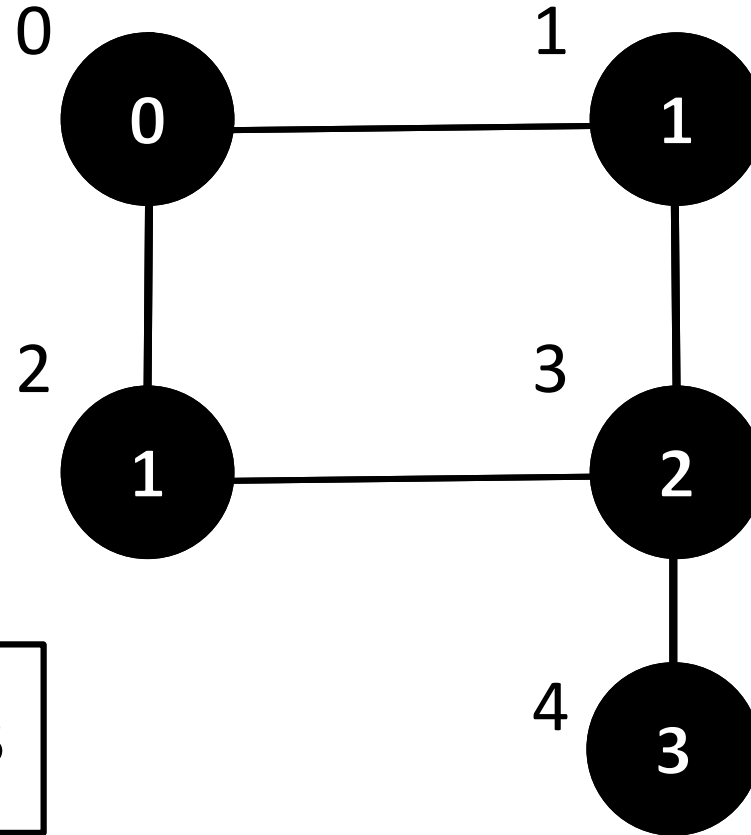


## Process V=3:

- C[3]=done

# Example

$S=0$



FIFO Queue  
←

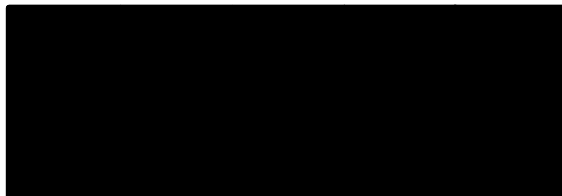
Process  $V=4$  (remove fr/Q):

- $C[4]=\text{done}$

D:

0	1	1	2	3
---	---	---	---	---

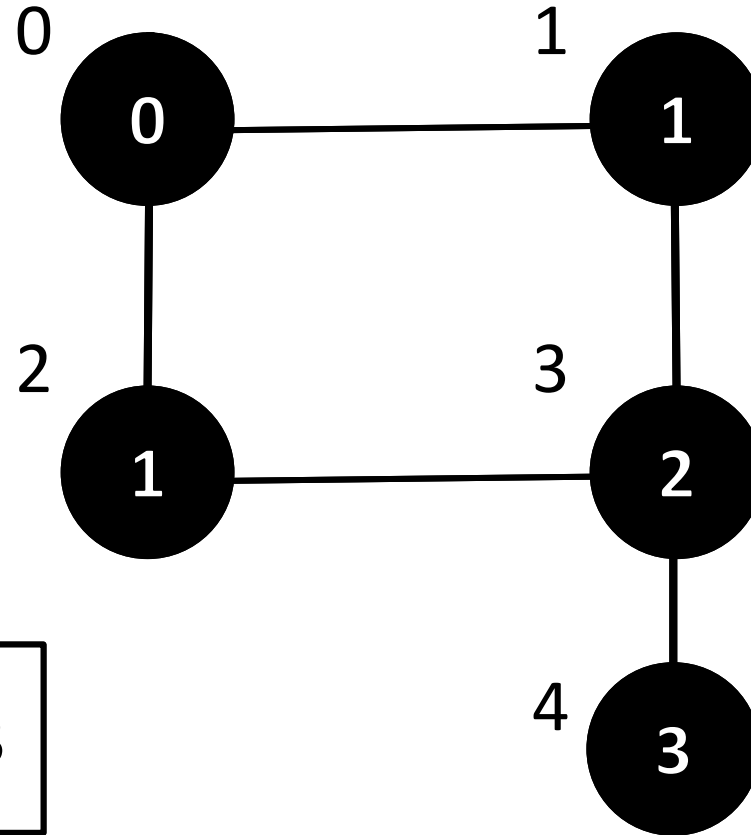
C:





# Example

S=0

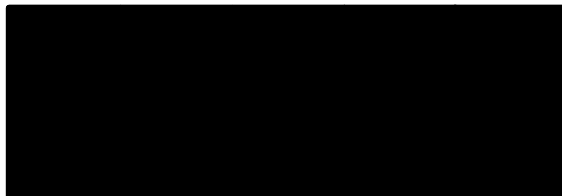


FIFO Queue  
←

D:

0	1	1	2	3
---	---	---	---	---

C:



DONE!

D[0]=0; D[1]=1; D[2]=1; D[3]=2; D[4]=3

# Execution Time

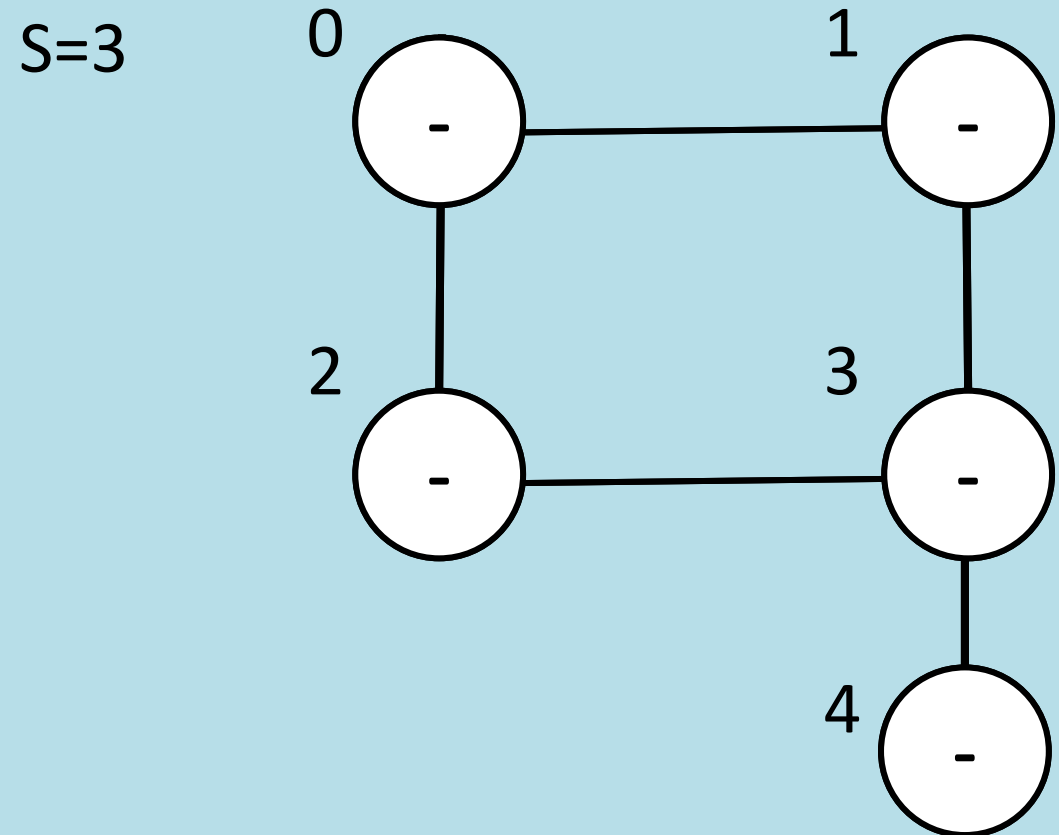
While (FIFO queue not empty)

- $V$  = Remove next vertex from FIFO queue
  - For each neighbor  $i$  of  $V$ 
    - If ( $C[i] == \text{unvis}$ )
      - $D[i] = D[V] + 1$
      - $C[i] = \text{vis}$
      - Add  $i$  to FIFO queue
  - $C[V] = \text{done}$
- 
- Outer loop (While): each vertex added to/removed from FIFO one time: time proportional to the number of nodes
  - Inner loop (For): time proportional to number of edges (e.g., all adjacency list entries scanned)

Total time proportional to sum of number of vertices and number of edges in graph:  $O(V+E)$

# Breadth-first search example

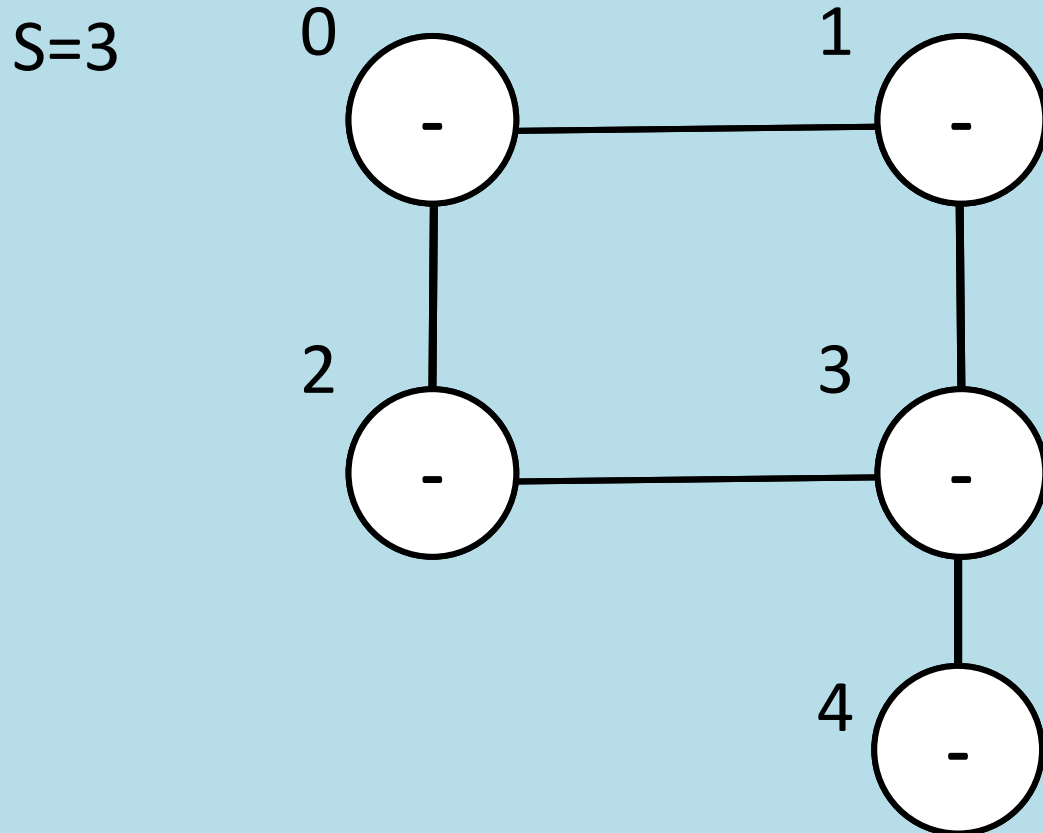
- Run through BFS on the same graph using  $S=3$  as the starting point! Show how the D and C arrays are updated (can be combined for ease of display) as well as the FIFO queue.



- Thought question: how many different colors/conditions are really needed to execute this algorithm?

# Breadth-first search example

- Run through BFS on the same graph using S=3 as the starting point!



3

D:

-	-	-	0	-
---	---	---	---	---

1 2 4

D:

-	1	1	0	1
---	---	---	---	---

2 4 0

D:

2	1	1	0	1
---	---	---	---	---

4 0

D:

2	1	1	0	1
---	---	---	---	---

0

D:

2	1	1	0	1
---	---	---	---	---

D:

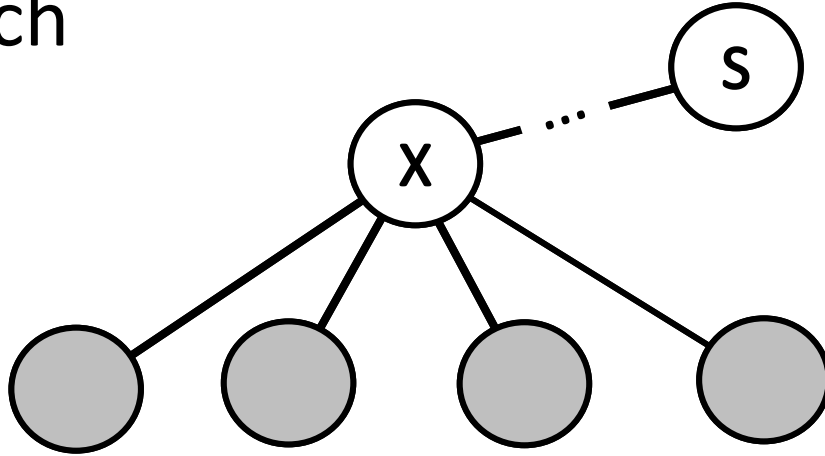
2	1	1	0	1
---	---	---	---	---

# Breadth-first search example

- How many different colors/conditions are really needed to execute this algorithm?
  - Really only two colors/conditions are needed, to differentiate between nodes that have been added to the queue already (at some point) and those that have not. There is no risk of re-adding a node to the queue once it has been processed.

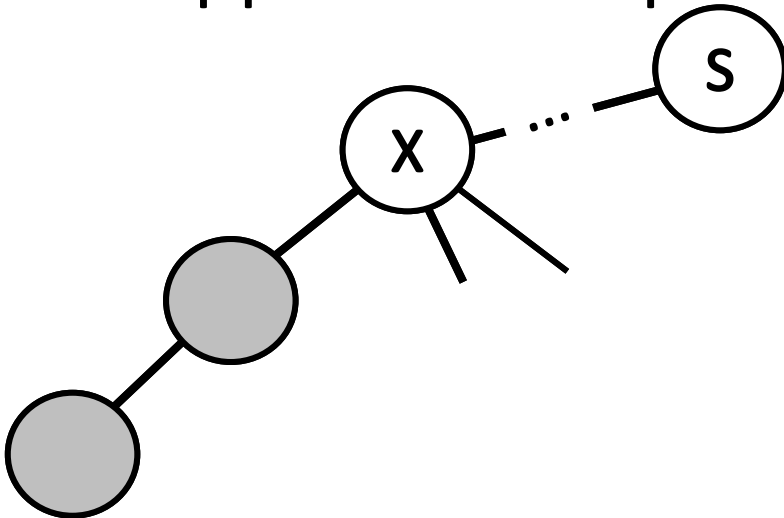
# Search Approaches

- Previous algorithm uses an approach called “breadth-first” search



Visit neighbors of X  
**before** exploring  
nodes further from S

- Another approach: “depth-first” search

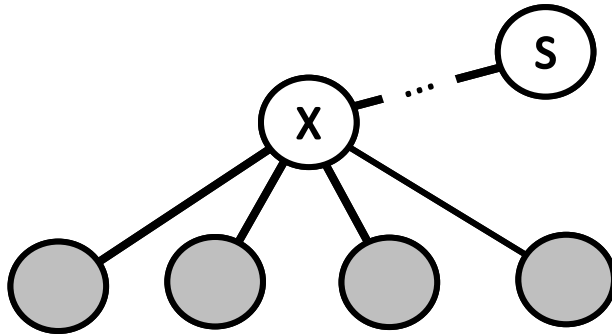


Visit nodes further  
from S before  
visiting all of X's  
neighbors

# Analogy: Literature Survey (or Web Crawler)

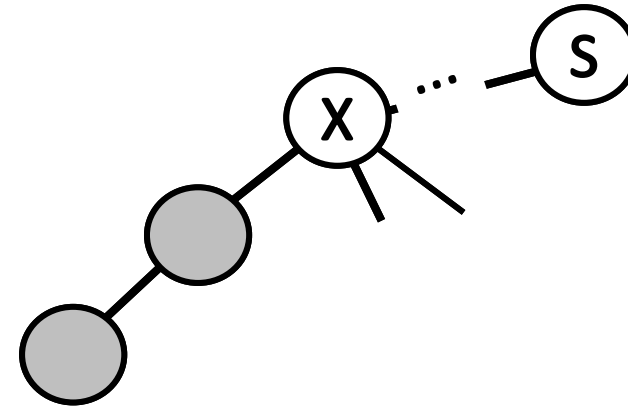
Paper citations can be viewed as a graph

- Node: a single paper
- Link: reference another paper in the bibliography



## Breadth-first search (BFS)

- Read one paper
- Read all papers cited in the bibliography of this paper
- For the set of papers you just read, compile list of the papers they cite that you have not already read; read these
- Repeat previous step until every paper that was cited has been read

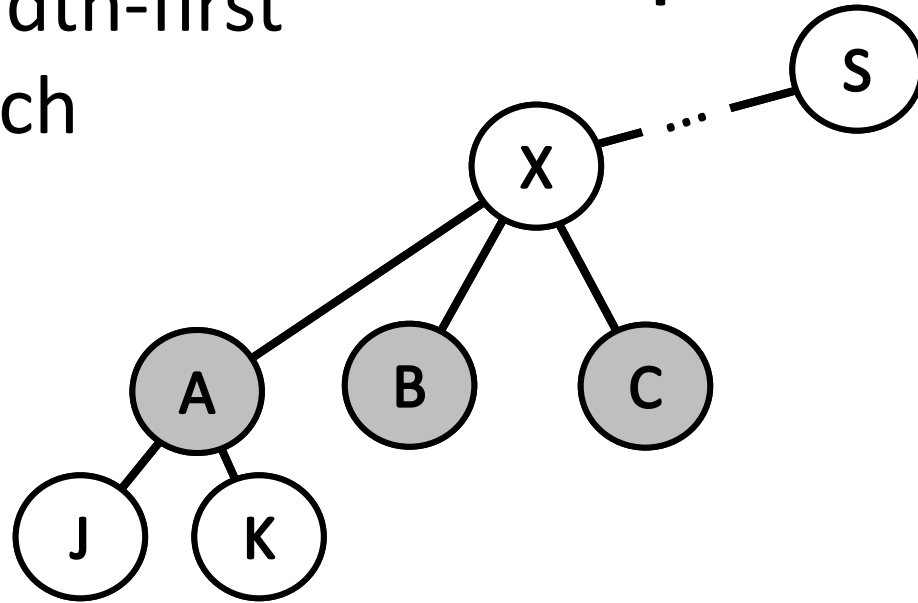


## Depth-first search (DFS)

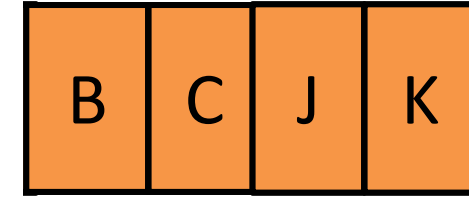
- Read one paper
- Read one paper cited by the paper you just read
- Repeat previous step until you reach a paper that only cites papers you've already read
- **Backtrack** to the last previously read paper that cites a paper you haven't read, and repeat DFS procedure starting from this paper

# Implementation

## Breadth-first Search



FIFO Queue  
(delete left; insert right)



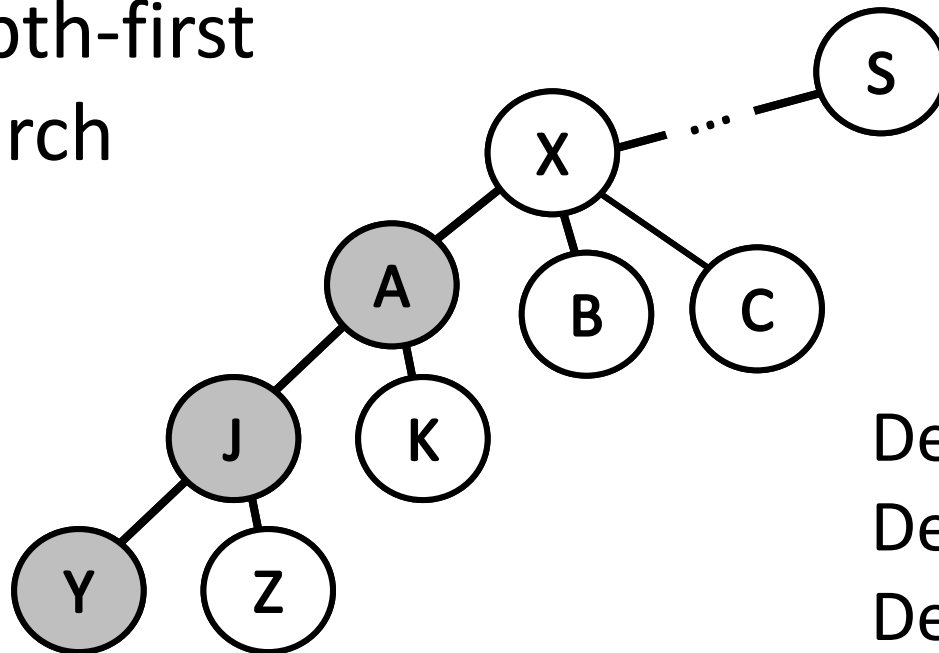
Delete (visit) B

Delete (visit) C

Delete (visit) J

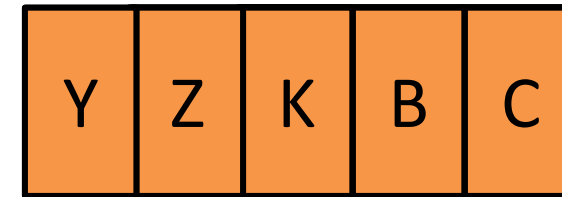
Delete (visit) K

## Depth-first Search



LIFO Stack

(delete left; insert left)



Delete (visit) Y

Delete (visit) Z

Delete (visit) K (backtracking)

Add (push) B

Add (push) C



# Depth ~~Breadth~~-first Search

Initially (neglecting distance calculation)

- $C[S] = \text{vis}; C[i] = \text{unvis}, i \neq S$
- ~~FIFO Queue~~ holds S

## Algorithm

While (~~FIFO queue~~ not empty)

V = Remove next vertex from ~~FIFO queue~~

For each neighbor i of V

If ( $C[i] == \text{unvis}$ )

$C[i] = \text{vis}$

Add i to ~~FIFO queue~~

$C[V] = \text{done}$

# Depth-first Search (Recursion)

// LIFO implemented via recursion

// two colors/conditions: visited, unvisited

for all vertices  $v$

if ( $C[v] == \text{unvis}$ ) DFS ( $v$ ); // perform DFS from vertex  $v$

**DFS ( $v$ )**

$C[v] = \text{vis}$

for each neighbor  $u$  of  $v$

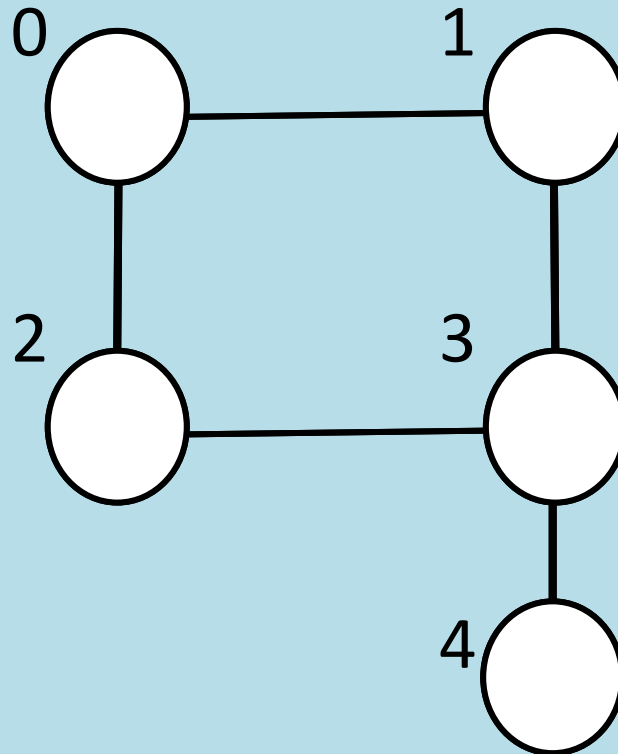
if ( $C[u] == \text{unvis}$ ) DFS ( $u$ );

# Things You Should Know

- Breadth-first search
  - Approach to visit all nodes of a connected graph
  - Visitation, in effect, builds a tree (aka spanning tree; no loops)
  - Visit nodes in level  $i$  of tree before exploring level  $i+1$
  - Implemented using a FIFO queue
  - Can be used to compute minimum path from source  $S$  (no edge weights)
- Depth-first search
  - Also visits all nodes in a connected graph and builds spanning tree
  - Go down tree, level by level, as quickly as possible, then backtrack to visit others in each level
  - Implemented using a LIFO stack
  - Simple, elegant description using recursion
  - Computes minimum path length?

# Depth-first search example

- Run through DFS on the same graph as before using  $S=0$  as the starting point using the stack approach. In what order are the nodes visited, assuming that when we add multiple nodes to the stack we push the highest-numbered node first?
- What if the starting node is changed to  $S=3$ ?
- Analyze the execution time for DFS.



## Initialization

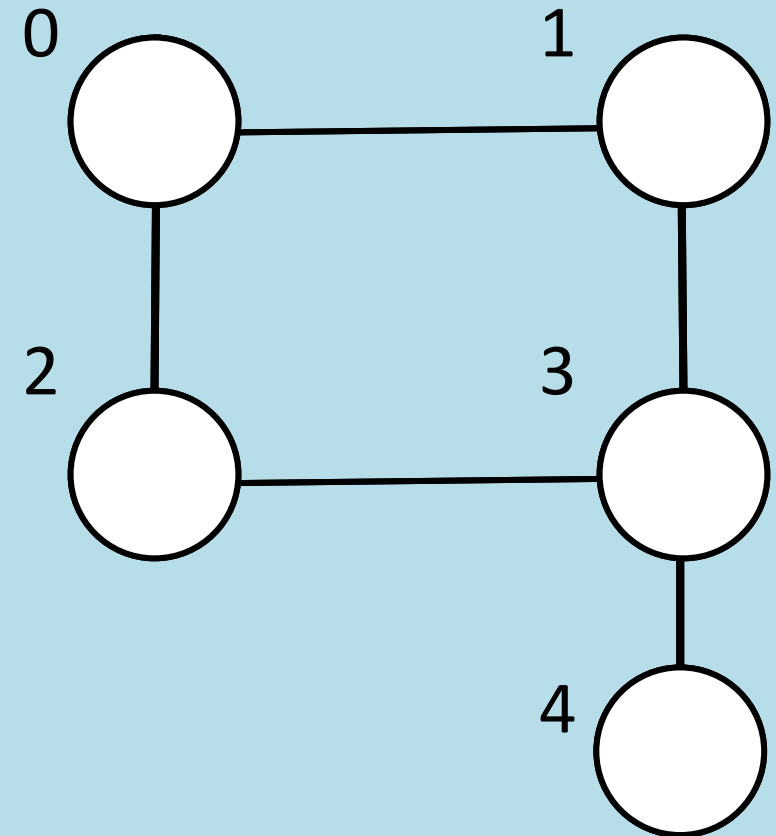
- $C[S]=vis$ ;  $C[i]=unvis$ ,  $i \neq S$
- Stack holds  $S$

## Algorithm

```
While (stack not empty)
  V = next vertex from stack
  For each neighbor i of V
    If ( $C[i] == unvis$ )
       $C[i] = vis$ 
      Add i to stack
   $C[V] = done$ 
```

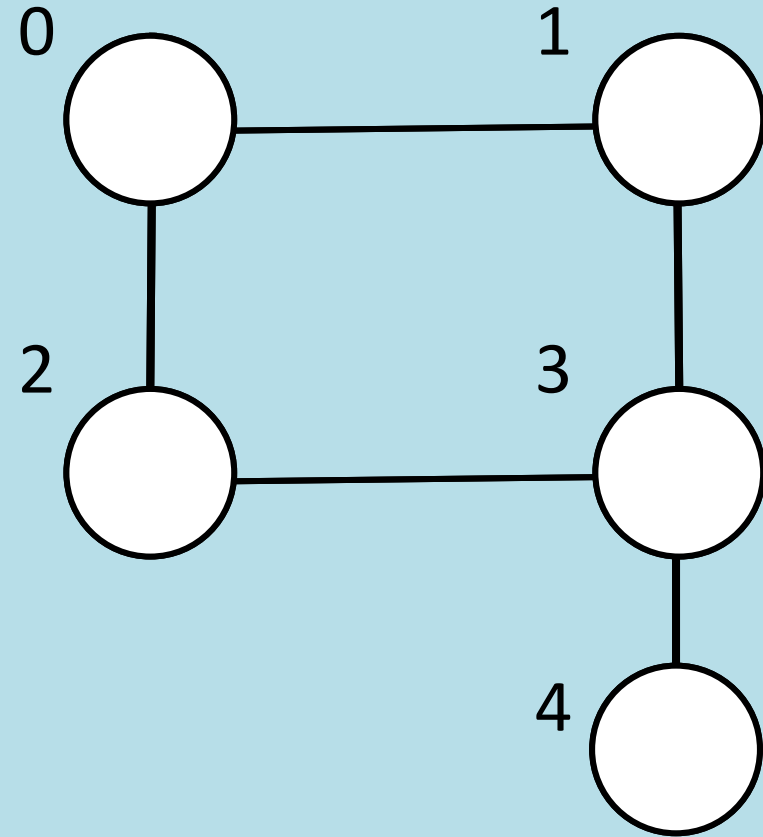
# Depth-first search example

- DFS using  $S=0$  as the starting point. In what order are the nodes visited, assuming that when we add multiple nodes to the stack we push the highest-numbered node first?
- Nodes are visited as follows:
  - 0; neighbors are 1 and 2, push to stack: 1 2
  - 1; neighbors are 0 and 3, push 3 to stack: 3 2
  - 3; neighbors are 1, 2, 4, push 4 to stack: 4 2
  - 4; neighbor is 3, time to backtrack
  - 2



# Depth-first search example

- DFS using  $S=3$  as the starting point.
- Nodes are visited as follows:
  - 3; neighbors are 1, 2, 4, push to stack: 1 2 4
  - 1; neighbors are 0 and 3, push 0 to stack: 0 2 4
  - 0; neighbors are 1 and 2, time to backtrack: 2 4
  - 2; neighbors are 0 and 3, backtrack again: 4
  - 4



# Depth-first search example

- Analyze the execution time for DFS.


The execution time is the **same as for BFS**:  $O(V+E)$ .

**Initially (neglecting distance calculation)**


$C[S]=vis$ ;  $C[i]=unvis$ ,  $i \neq S$

LIFO stack holds  $S$

## Algorithm

While (LIFO stack not empty)  *Loop over all vertices*

$V$  = Remove next vertex from LIFO stack

For each neighbor  $i$  of  $V$   *Sweep through entries of adjacency list; cumulatively all edges will be traversed*

    If ( $C[i]==unvis$ )

$C[i] = vis$

        Add  $i$  to LIFO stack

$C[V] = done$

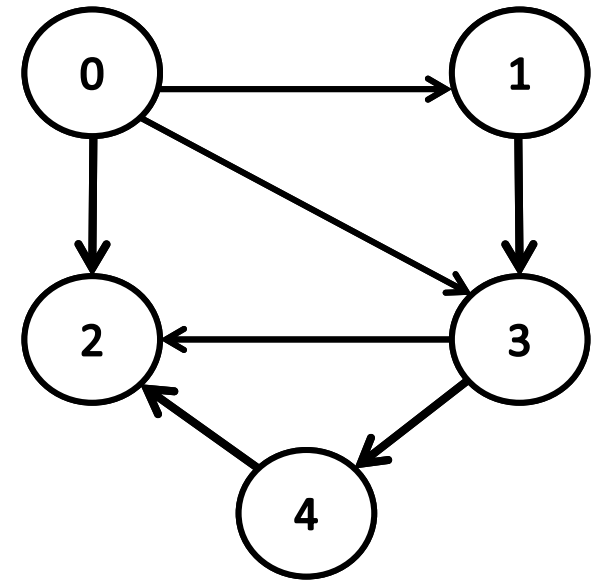
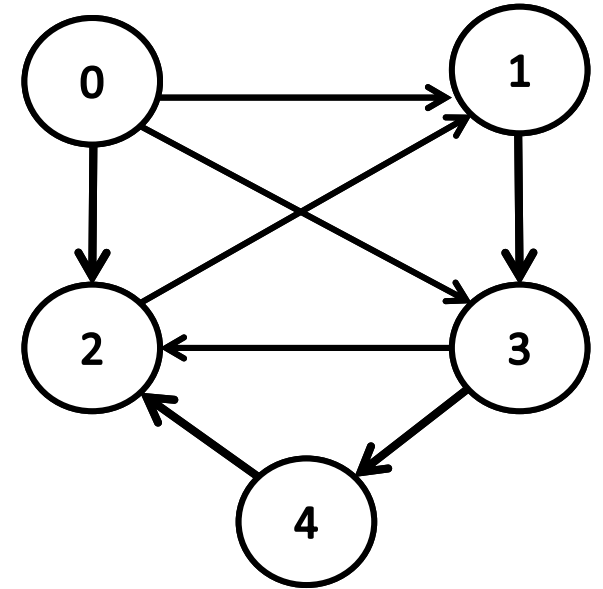
# Topological Sorting

- Sometimes it is useful to perform a ***topological sort*** of a graph to indicate precedences among events.
- We will do this only for a directed acyclic graph (“dag”).
  - Edges are directed (arrows).
  - Graph does not contain any cycle: in a series of ordered vertices with each consecutive pair of vertices connected by an edge, no vertices repeat.
- A topological sort is a linear ordering of all the vertices of a dag such that if  $G$  contains an edge  $(u,v)$ ,  $u$  appears before  $v$  in the ordering.
  - Essentially, a listing of vertices along a horizontal line such that all directed edges go from left to right.



# Topological Sorting

- Examples:
  - Top graph has cycles (e.g., 1321, 34213)
  - Bottom graph does not and thus is a dag.
- Topological sorts provide a potential order of events.
  - For a curriculum, vertices could be courses and edges could indicate prerequisites.
  - A topological sort could list an ordering for taking those courses.
- Topological sorts are not necessarily unique.



# Topological Sorting

- To do topological sorting, we can use recursive DFS with one modification: add a stack to store each completed vertex (returning to 3 conditions/ colors for explanation).
- Then pop from the stack in order to obtain the topological sort.

$C[i] = \text{unvis};$

for all vertices  $v$

if ( $C[v] = \text{unvis}$ ) DFS ( $v$ ) // perform DFS from vertex  $v$

**DFS ( $v$ )**

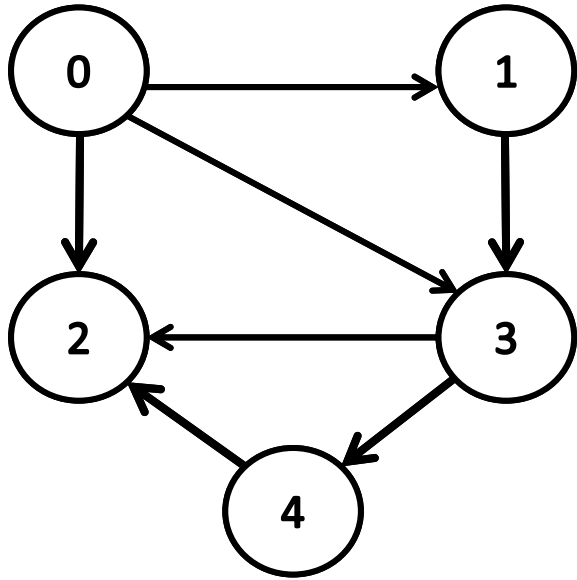
$C[v] = \text{vis}$

for each neighbor  $u$  of  $v$

if ( $C[u] = \text{unvis}$ ) DFS ( $u$ )

$C[v] = \text{done}$ , Add  $v$  to output stack

# Topological Sort



$C[i] = \text{unvis}$   
for all vertices  $v$   
if ( $C[v] = \text{unvis}$ ) DFS ( $v$ )

## DFS ( $v$ )

$C[v] = \text{vis}$

for each neighbor  $u$  of  $v$   
if ( $C[u] == \text{unvis}$ ) DFS ( $u$ )

$C[v] = \text{done}$ , Add  $v$  to output stack

$C = [u \ u \ u \ u \ u]$

DFS[0]

$C = [v \ u \ u \ u \ u]$

DFS[1]

$C = [v \ v \ u \ u \ u]$

DFS[3]

$C = [v \ v \ u \ v \ u]$

DFS[2]

$C = [v \ v \ v \ v \ u]$

No unvisited neighbors

$C = [v \ v \ d \ v \ u]$ , output = [2]

DFS[4]

$C = [v \ v \ d \ v \ v]$

No unvisited neighbors

$C = [v \ v \ d \ v \ d]$ , output = [4 2]

$C = [v \ v \ d \ d \ d]$ , output = [3 4 2]

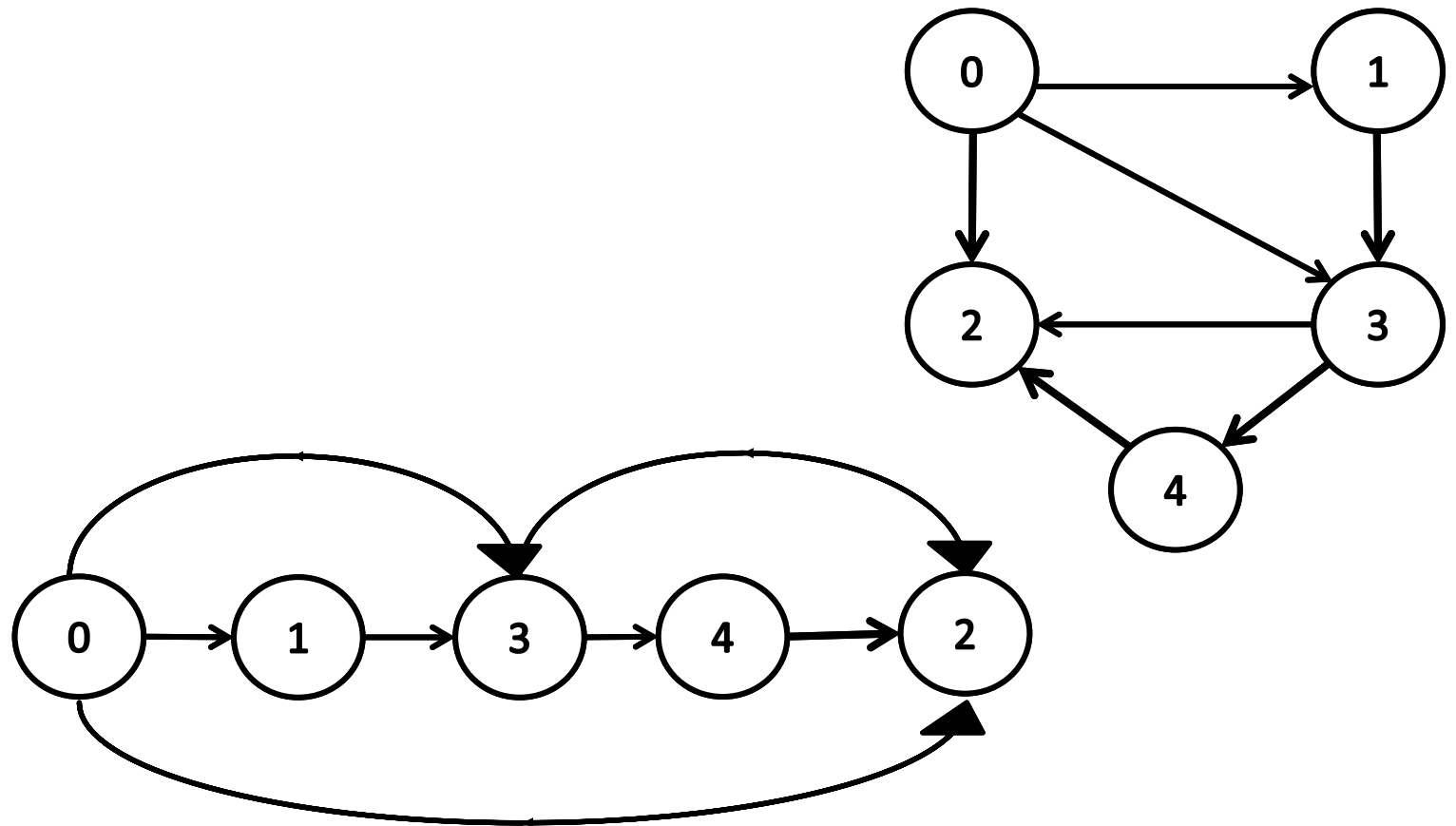
$C = [v \ d \ d \ d \ d]$ , output = [1 3 4 2]

$C = [d \ d \ d \ d \ d]$ , output = [0 1 3 4 2]

# Confirming the Topological Sort

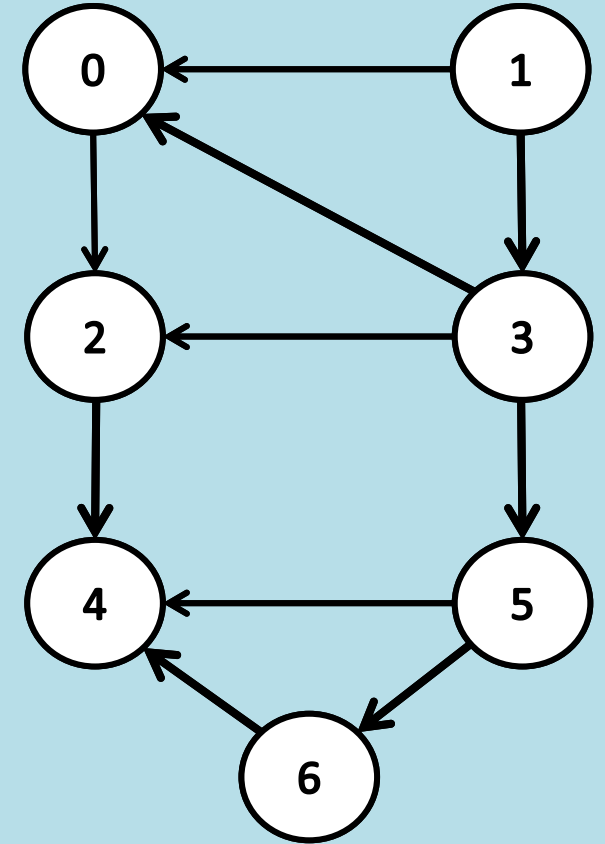
- We can confirm the topological sort result: [0 1 3 4 2]
- Check that for all edges  $[u, v]$ ,  $u$  appears before  $v$
- List of edges:

- (0, 1)
- (0, 2)
- (0, 3)
- (1, 3)
- (3, 2)
- (3, 4)
- (4, 2)



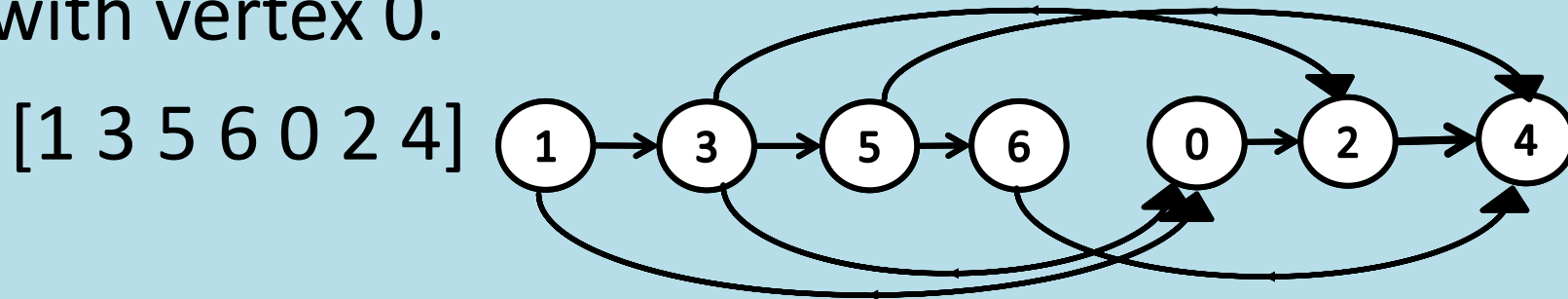
# Topological Sort Example

- Use the extended DFS algorithm to perform a topological sort of the graph, starting with vertex 0.
- Repeat, but assume we process vertices in the order 3 4 5 6 0 1 2 (essentially this is a relabeling).

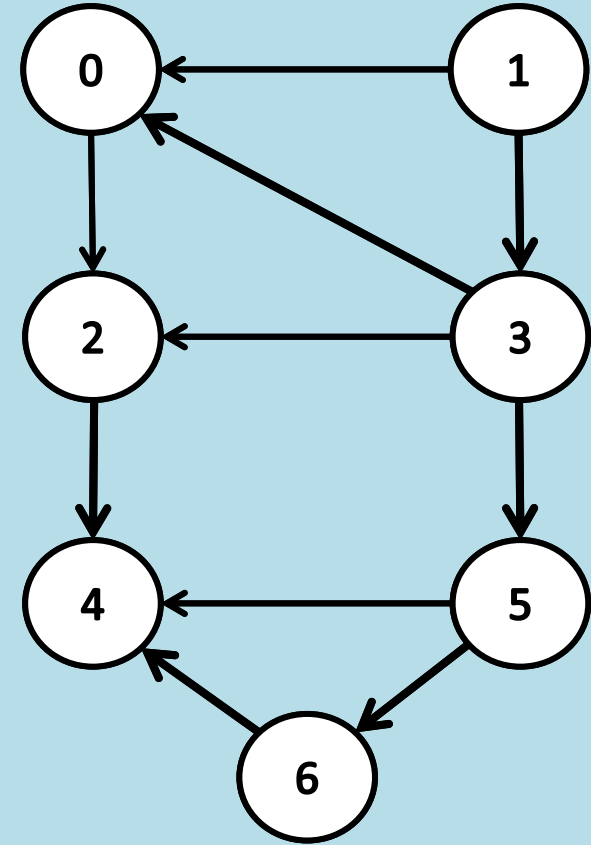
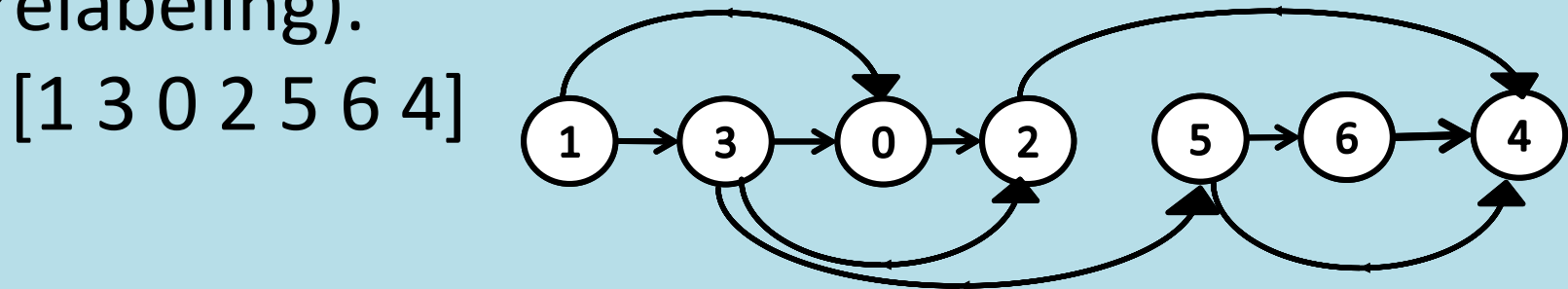


# Topological Sort Example

- Use the extended DFS algorithm to perform a topological sort of the graph below, starting with vertex 0.



- Repeat, but assume we process vertices in the order 3 4 5 6 0 1 2 (essentially this is a relabeling).



# Topological Sorting Complexity

- Topological sorting is just DFS with additional storage.
- New storage is  $O(V)$ .
- Thus, topological sorting remains  $O(V + E)$ .
- This is one approach for topological sorting; multiple topological sorts can exist, and not all topological sorts can be produced by this algorithm.