Graphs

CSE 2320 – Algorithms and Data Structures

Alexandra Stefan
and Vassilis Athitsos
University of Texas at Arlington

References and Recommended Review

Recommended Student Review from CSE 2315

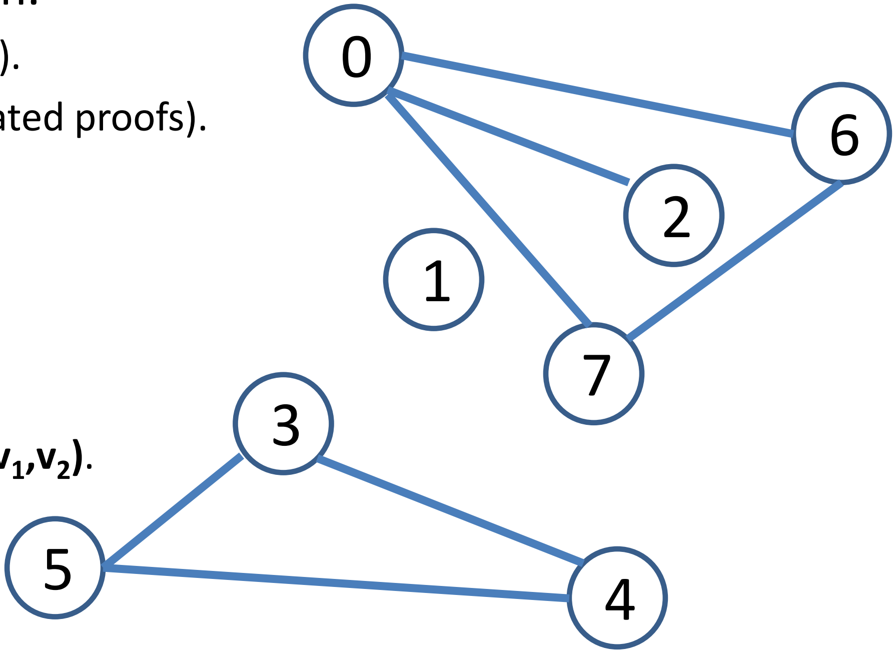
- Representation
  - Adjacency matrix
  - Adjacency lists
- Concepts:
  - vertex, edge, path, cycle, connected, bipartite.
- Search:
  - Breadth-first
  - Depth-first

- Recommended: CLRS
- Graph definition and representations
  - CLRS (3rd edition) - Chapter 22.1 (pg 589)
  - Sedgewick - Ch 3.7 (pg 120)
    - 115-120: 2D arrays and lists
- Graph traversal
  - CLRS: BFS - 22.2, DFS-22.3
  - Sedgewick, Ch 5.8
- The code used in slides is from Sedgewick.
- See other links on the Code page.

Graphs

- Graphs are representations of structures, set relations and states and state-transitions.
  - Direct representation of a real-world structures
    - Networks (roads, computers, social)
  - States and state transitions of a problem.
    - Game-playing algorithms (e.g., Rubik's cube).
    - Problem-solving algorithms (e.g., for automated proofs).

- A graph is defined as **G = (V,E)** where:
  - **V** : set of vertices (or nodes).
  - **E** : set of edges.
    - Each edge is a pair of two vertices in V: $\mathbf{e = (v_1,v_2)}$.

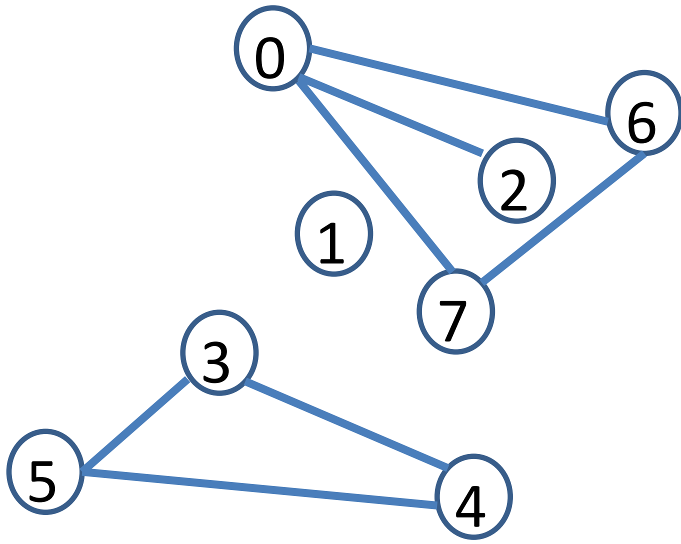


Graphs

- G = (V,E)
  - How many graphs are here?: __1
  - |V| = _8_ , V: { 0, 1, 2, 3, 4, 5, 6, 7 }
  - |E| = _7_ , E: { (0,2), (0,6), (0, 7), (3, 4), (3, 5), (4, 5), (6,7)}.
- Paths
  - Are 2 and 7 connected? Yes: paths 2-0-6-7 or 2-0-7
  - Are 1 and 3 connected? No.
- Cycle
  - A path from a node back to itself.
  - Any cycles here? 3-5-4-3, 0-6-7-0
- Directed / undirected
- Connected component (in undirected graphs)
  - A set of vertices s.t. for any two vertices, u and v, there is a path from u to v.
  - Here: Maximal: {1}, {3,4,5}, {2,0,6,7}. Non-maximal {0,6,7}, {3,5},…
  - In directed graphs: strongly connected components.
- Degree of a vertex
  - Number of edges incident to the vertex (for undirected graphs).
  - Here: degree(0) = 3, degree(1) = 0 , degree(5) = 2
- Sparse /dense
- Representation: adjacency matrix, adjacency list

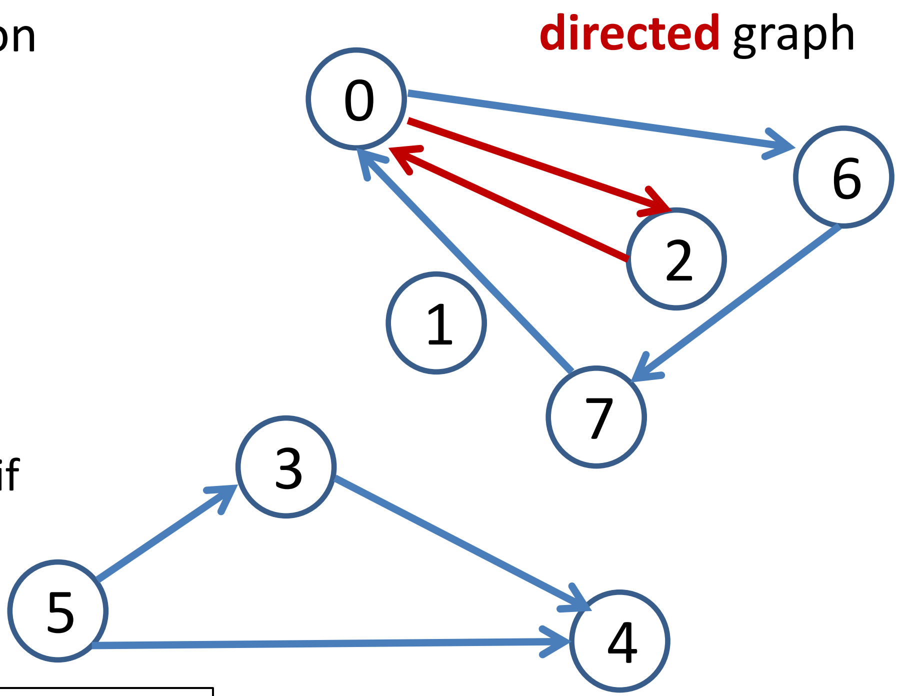
Note: A tree is a graph that: is connected and has no cycles

Undirected graph



Directed vs Undirected Graphs

- Graphs can be directed or undirected.
- Undirected graph: edges have no direction
  - edge (A, B) means that we can go (on that edge) from **both A to B and B to A.**
- Directed graph: edges have direction
  - edge (A, B) means that we can go (on that edge) **from A to B, but not from B to A.**
  - will have both edge (A, B) and edge (B, A) if A and B are linked in both directions.

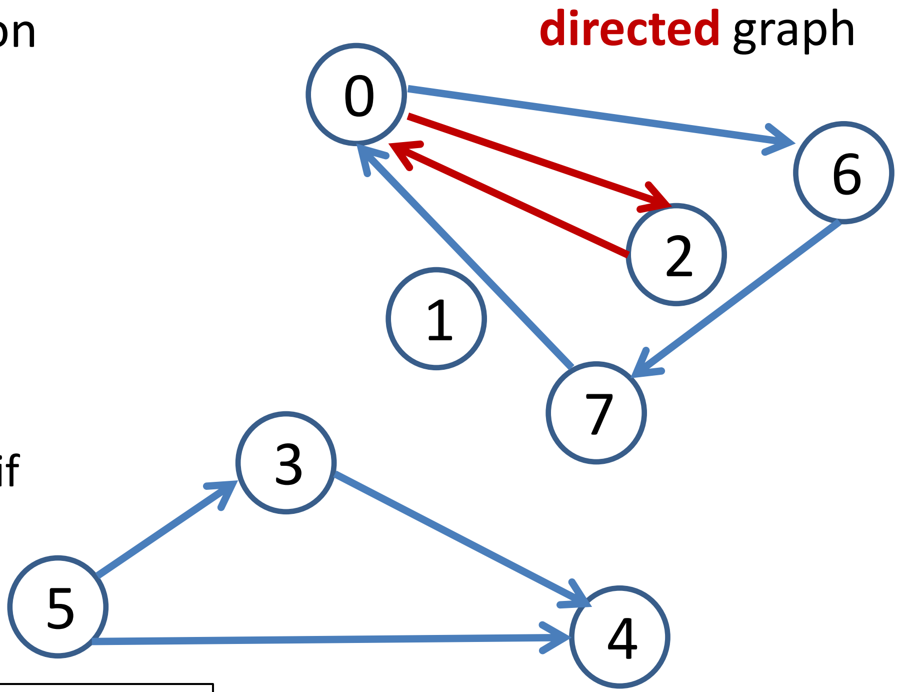


Degree of a vertex of a directed graph:
- **In-degree** – number of edges arriving at this vertex
- **Out-degree** – number of edges leaving from this vertex

Vertex	0	4	5	1	7
In degree					
Out-degree					

Directed vs Undirected Graphs

- Graphs can be directed or undirected.
- Undirected graph: edges have no direction
  - edge (A, B) means that we can go (on that edge) from **both A to B and B to A.**
- Directed graph: edges have direction
  - edge (A, B) means that we can go (on that edge) **from A to B, but not from B to A.**
  - will have both edge (A, B) and edge (B, A) if A and B are linked in both directions.

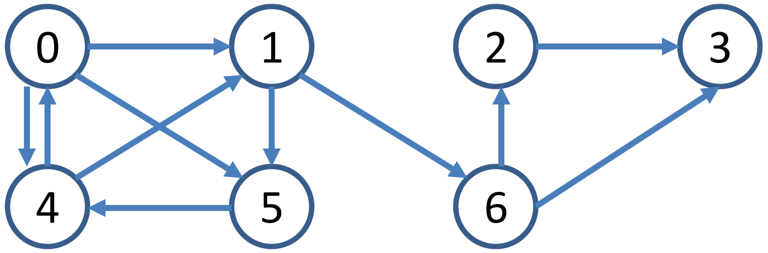


Degree of a vertex of a directed graph:
- **In-degree** – number of edges arriving at this vertex
- **Out-degree** – number of edges leaving from this vertex

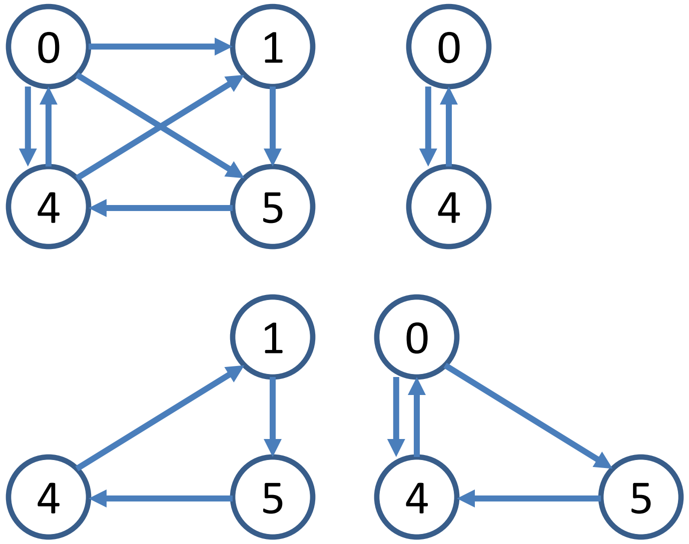
Vertex	0	4	5	1	7
In degree	**2**	**2**	**0**	**0**	**1**
Out-degree	**2**	**0**	**2**	**0**	**1**

Strongly Connected Components (Directed Graphs)

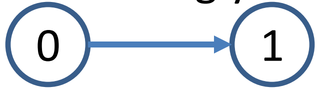
- How many "connected components" does this graph have?
  1. Can you get from 0 to every other vertex?
  2. Can you get from 3 to 6?
  3. Connected components do not apply to directed graphs.
- For directed graphs we define **strongly** connected components: a subset of vertices, V_s, and the edges between them , E_s, such that for any two vertices u,v in V_s we can get from u to v (and from v to u) with only edges from E_s.
  - Strongly connected components in this graph:
    {0,1,4,5}, {0,4}, {1,5,4}, {0,5,4}
  - NOT strongly connected components.
    {6,2,3}, {0,1} Why?



Strongly connected components:



NOT strongly connected



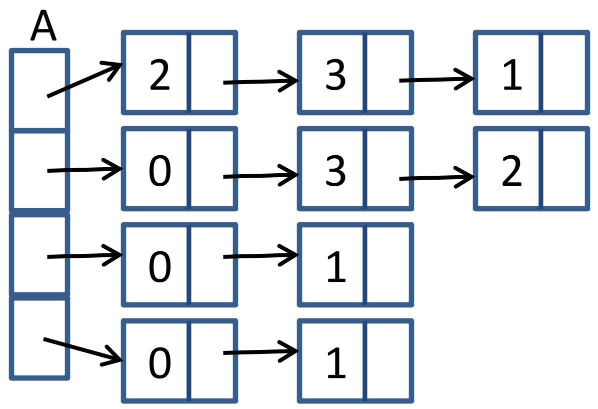
Graph Representations

- G = (V,E). Let |V| = N and |E| = M.
  - |V| is the size of set V, i.e. number of vertices in the graph. Similar for |E|.
  - <u>Notation abuse: V (and E) instead of |V| (and |E|).</u>
- Vertices: store N
  - E.g.: If graph G has N=8 vertices, those vertices will be: 0, 1, 2, 3, 4, 5, 6, 7.
  - Excludes case where additional labels are needed for vertices (e.g. city names).
- Edges: 2 representations:

Adjacency matrix:
A is a 2D matrix of size NxN

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

Adjacency lists:
A is a 1D array of N linked lists



Adjacency Matrix

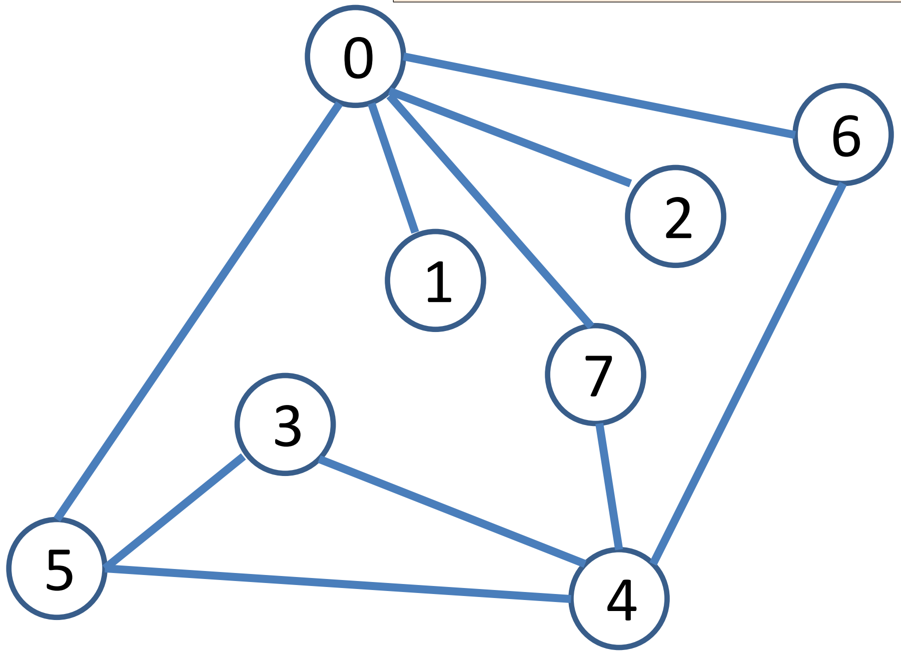
Let |V| = N vertices: 0,1, . . . , N-1.
Represent edges using a 2D binary matrix, M, of size |V|*|V|.

M[v_1][v_2] = 1 if and only if there is an edge from v_1 to v_2.

M[v_1][v_2] = 0 otherwise (there is no edge from v_1 to v_2).

- Space complexity: Θ(|V|*|V|)
- Time complexity for add/remove/check edge: Θ(1)
- Time complexity to find neighbors: Θ(V)

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



Note: the adjacency matrix of non-directed graphs is symmetric.

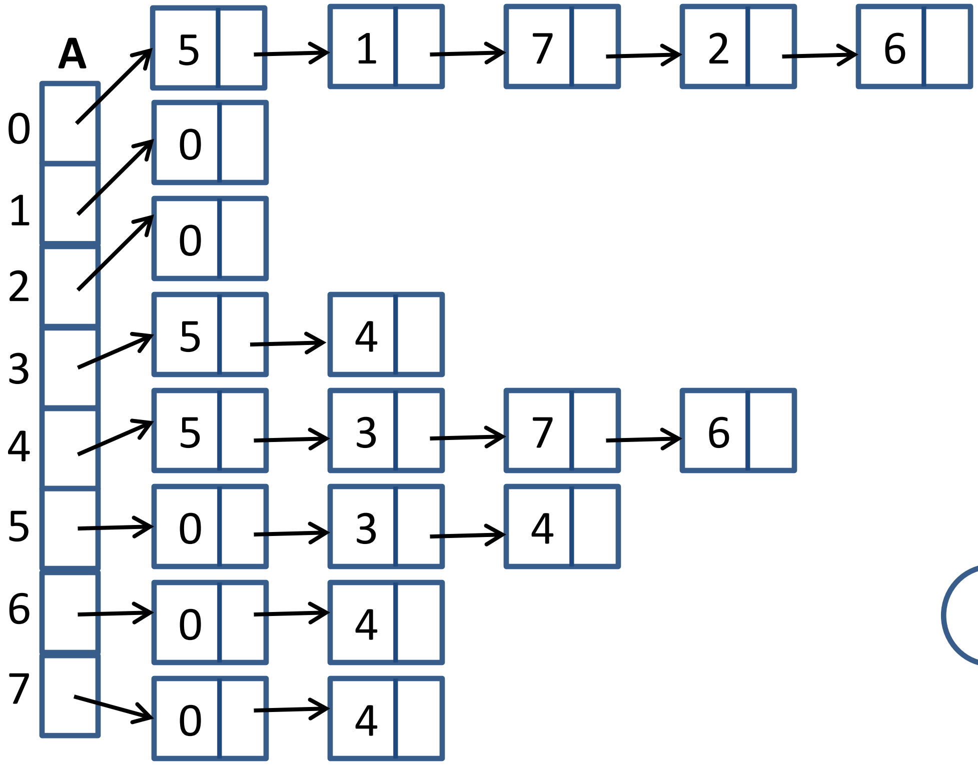
C implementation for Adjacency Matrix

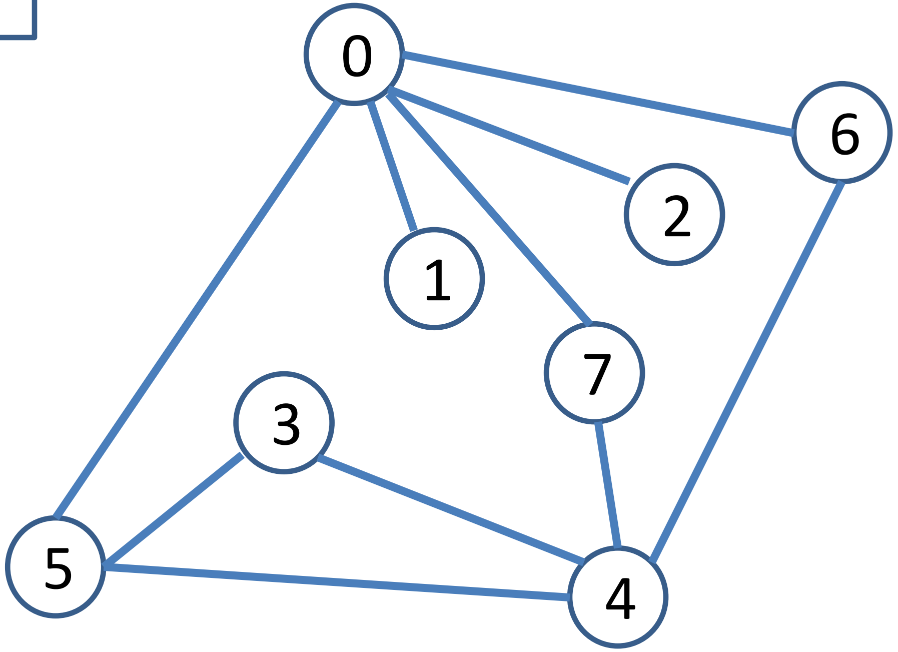
```
typedef struct struct_graph * graph;
struct struct_graph {
   int number_of_vertices;
   int ** adjacencies;
};

int edgeExists(graph g, int v1, int v2){
   return g->adjacencies[v1][v2];
}

void addEdge(graph g, int v1, int v2){
   g->adjacencies[v1][v2] = 1;
   g->adjacencies[v2][v1] = 1;
}

void removeEdge(graph g, int v1, int v2){
   g->adjacencies[v1][v2] = 0;
   g->adjacencies[v2][v1] = 0;
}
```

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

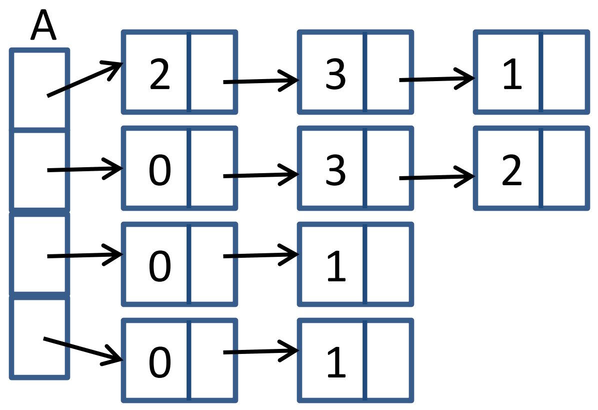
Adjacency Lists

- Represent the edges of the graph by an **array of linked lists**.
  - Let's name that array A
  - $A[v_1]$ is a list containing the neighbors of vertex v_1.





Adjacency Lists



- **Space**
  - for A
  - For nodes:

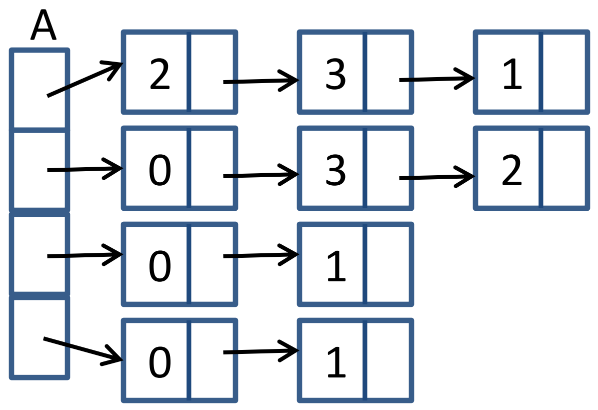
- **Time** to check if an edge exists or not
  - Worst case:

- **Time** to remove an edge?

- **Time** to add an edge?

Adjacency Lists

- **Space:** **Θ(E + V)**
  - For A: Θ(V)
  - For nodes: Θ(E)
  - If the graph is relatively sparse, $|E| << |V|^2$, this can be a significant advantage.



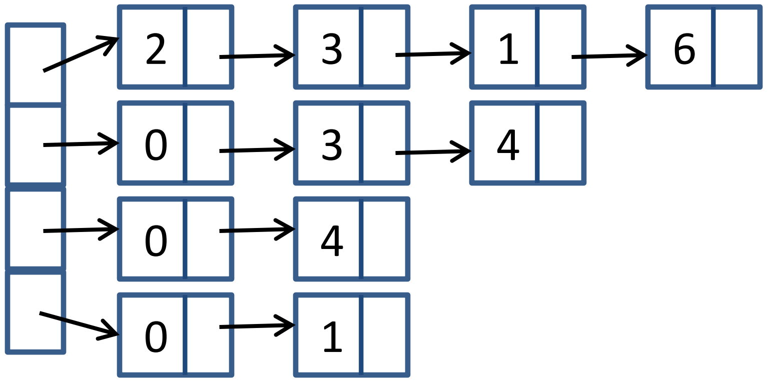
- **Time** to check if an edge exists or not: **O(V)**
  - Worst case: Θ(V).
    - Each vertex can have up to V-1 neighbors, and we may need to go through all of them to see if an edge exists.
  - Slower than using adjacency matrices.

- **Time** to remove an edge: **O(V)**
  - If must check if the edge exists.
- **Time** to add an edge: **O(V)**
  - If must check if the edge exists.
    - Why? Because if the edge already exists, we should not duplicate it.

C implementation of Adjacency Lists

Defining the object type:

```
typedef struct struct_graph * graph;
struct struct_graph{
   int number_of_vertices;
   list * adjacencies;
};
```



Time: _____   Space: ______

```
int edgeExists(graph g, int v1, int v2) {
   link n;
   for(n=getFirst(g->adjacencies[v1]); n!=NULL; n=getLinkNext(n)) {
      if (getLinkItem(n) == v2)
          return 1;
   }
   return 0;
}
```

Time: _____   Space: ______

```
void addEdge(graph g, int v1, int v2){
   if !(edgeExists(g, v1, v2)){
      insertAtBeginning(g->adjacencies[v1], newLink(v2));
      insertAtBeginning(g->adjacencies[v2], newLink(v1));
   }
}
```

// Similar for remove edge: iterate through lists of v1 and v2 to find the other and remove it.

Sparse Graphs

- Sparse graph
  - A graph with $|E| << |V|^2$.

  - E.g. consider an undirected graph with 10^6 nodes
    - Number of edges if 20 edges per node: $10^6*20/2$
    - Max possible edges $10^6*(10^6-1)/2$

    => 10^5 factor between max possible and actual number of edges

    => Use adjacency lists
  - Can you think of real-world data that may be represented as sparse graphs?

Space Analysis: Adjacency Matrices vs. Adjacency Lists

- Suppose we have an undirected graph with:
  - 10 million vertices.
  - Each vertex has at most 20 neighbors.

- Individual practice: Calculate the minimum space needed to store this graph in each representation. Use/assume:
  - A matrix of BITS for the matrix representation
  - An int is storred on 8 bytes and a memory address is stored on 8 bytes as well.

  Calculate the space requirement (actual number, not Θ) for each representation.

  Compare your result with the numbers below.

  Check your solution against the posted one. Clarify next lecture any questions you may have.

- Adjacency matrices: we need at least 100 trillion bits of memory, so at least 12.5TB of memory.
- Adjacency lists: in total, they would store at most 200 million nodes. With 16 bytes per node (as an example), this takes at most 3.28 Gigabytes.
- We'll see next how to compute/verify such answers.

Steps for Solving This Problem: understand all terms and numbers

- Suppose we have an undirected graph with:
  - 10 million vertices.
  - Each vertex has at most 20 neighbors.
- Adjacency matrices: we need at least 100 trillion bits of memory, so at least 12.5TB of memory.
- Adjacency lists: in total, they would store at most 100 million nodes. With 16 bytes per node (as an example), this takes 1.68 Gigabytes.

- Find ‘keywords’, understand numbers:
  - 10 million vertices => 10 * 10^6
  - Trillion = 10^{12}
  - 1 TB (terra bytes) = 10^{12} bytes
  - 1GB = 10^9 bytes
  - 100 Trillion <u>bits</u> vs 12.5 TB (terra <u>bytes</u>)

Solving: Adjacency Matrix

- Suppose we have a graph with:
  - 10 million vertices. => |V| = $10*10^6$ = 10^7
  - Each vertex has at most 20 neighbors.

- Adjacency matrix representation for the graph:
  - The smallest possible matrix: a 2D array of **bits** =>
  - The matrix size will be: |V| x |V| x 1bit =>

  $10^7 * 10^7 * 1\text{bit} = 10^{14}$ bits

  - Bits => bytes:

  1byte = 8bits => 10^{14}bits = $10^{14}/8$ bytes = $100/8*10^{12}$bytes = $12.5*10^{12}$bytes

  - $12.5*10^{12}$bytes = **12.5 TB (final result)**

$\boxed{10^{12}\text{bytes} = 1\text{TB}}$

Solving: Adjacency List

- Suppose we have an undirected graph with:
  - 10 million vertices. => **V = 10^7**
  - Each vertex has **at most 20 neighbors**.

- Adjacency lists representation of graphs:
  - For each vertex, keep a list of edges (a list of neighboring vertices)
  - Space for the adjacency list array:

    = 10 million vertices*8 bytes (memory address) = 8*10^7 bytes = 0.08 GB
  - Space for all the lists for all the vertices:

    ≤ 10^7vertices * (20 neighbors/vertex) = **20*10^7 nodes** = 2*10^8 nodes

    Assume 16 bytes per node: 8 bytes for the *next* pointer, and 8 bytes for the data (vertex):

    2*10^8 nodes * 16byte/node = 32 * 10^8 bytes = 3.2 * 10^9 bytes = 3.2GB

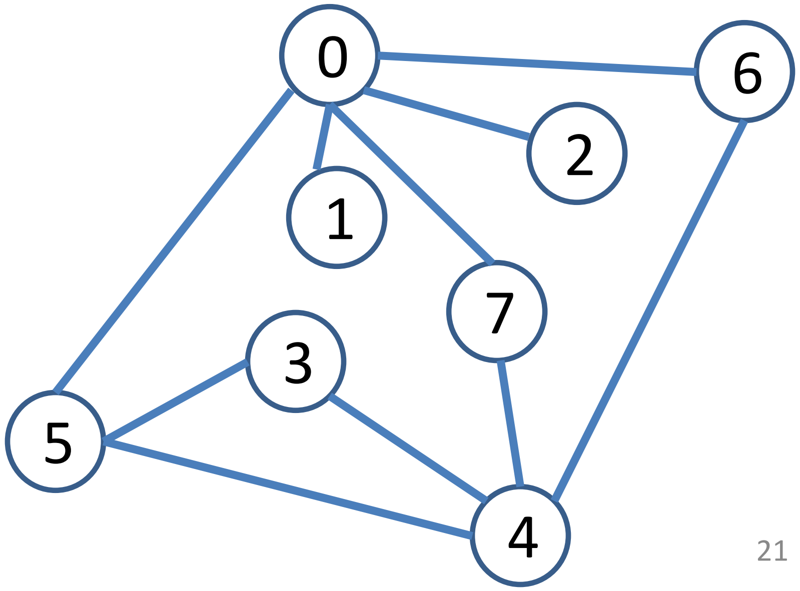
    Total: **3.2GB + 0.08 GB = 3.28GB** (10^9 bytes = 1GB (GigaByte))

Check Out Posted Code

- **graph.h**: defines an abstract interface for basic graph functions.
- **graph_matrix.c**: implements the abstract interface of graph.h, using an adjacency matrix.
- **graph_list.c**: also implements the abstract interface of graph.h, using adjacency lists.
- **graph_main:** a test program, that can be compiled with **either** graph_matrix.c or graphs_list.c.

Graph Traversal - Graph Search

- We will use **"graph traversal"** and **"graph search"** almost interchangeably.
  - However, there is a small difference:
    - "Traversal": visit every node in the graph.
    - "Search": visit nodes until find what we are looking for. E.g.:
      - A node labeled "New York".
      - A node containing integer 2014.
- Graph traversal:
  - Input: start/source vertex.
  - Output: a sequence of nodes resulting from graph traversal.



Graph Traversals

Breath-First Search (**BFS**)

- O(V+E) (when Adj list repr)

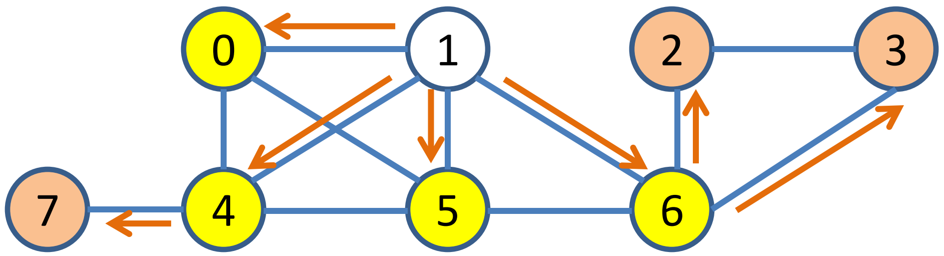
- Explores vertices in the order:
  - root, (no color)
  - nodes 1 edge away from the root, (yellow)
  - nodes 2 edges away from the root, (orange)
  - … and so on until all nodes are visited

- If graph is a tree, gives a level-order traversal.

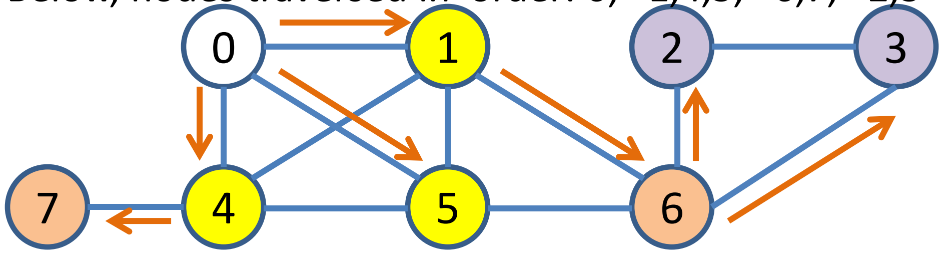
- Finds shortest paths from a source vertex.

*Length of the path is **the number of edges** on it.

E.g. Flight route with fewest connections.



Below, nodes traversed in order: 0, 1,4,5, 6,7, 2,3

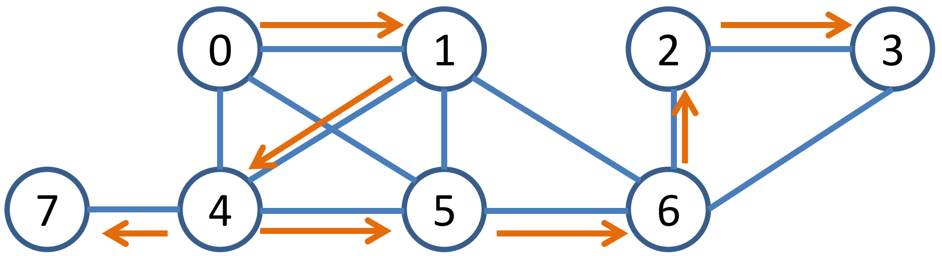


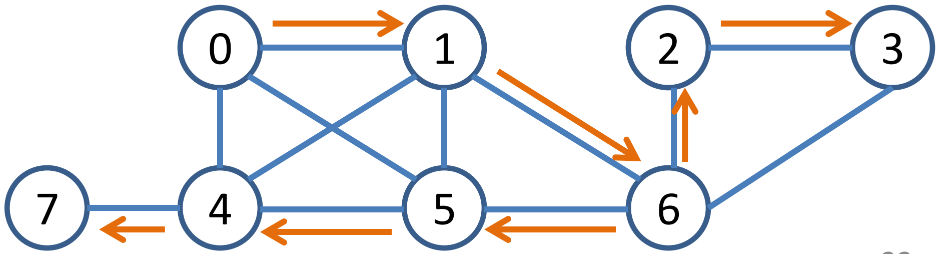
Depth-First Search (**DFS**)

- O(V+E) (when Adj list repr)

- Explores the vertices by following down a path as much as possible, backtracking and continuing from the last discovered node.

- Useful for
  - Finding and labelling connected components (easy to implement)
  - Finding cycles
  - Topological sorting of DAGs (Directed Acyclic Graphs).





For both DFS and BFS the resulting trees depend on the order in which neighbors are visited.

Vertex coloring while searching

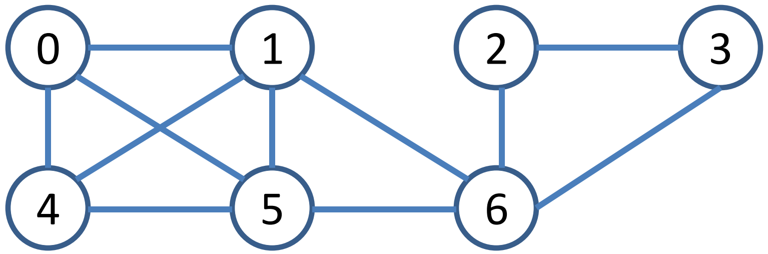
- Vertices will be in one of 3 states while searching and we will assign a color for each state:
  - White – undiscovered
  - Gray – discovered, but the algorithm is not done processing it
  - Black – discovered and the algorithm finished processing it.

Breadth-First Search (BFS)

CLRS 22.2

```
BFS(G,s)  // search graph G starting from vertex s.
    1.  For each vertex u of G
        1.  color[u] = WHITE
        2.  dist[u] = inf       // distance from s to u
        3.  pred[u] = NIL       // predecessor of u on the path from s to u
    2.  color[s] = GRAY
    3.  dist[s] = 0
    4.  pred[s] = NIL
    5.  Initialize empty queue Q
    6.  Enqueue(Q,s)      // s goes to the end of Q
    7.  While Q is not empty
        1.  u = Dequeue(Q)  // removes u from the front of Q
        2.  For each v adjacent to u   //explore edge (u,v)
            1.  If color[v] == WHITE
                1.  color[v] = GRAY
                2.  dist[v] = dist[u]+1
                3.  pred[v] = u
                4.  Enqueue(Q,v)
        3.  color[u] = BLACK
```

Repr\runtime	BFS
Adj LIST	$O(\|V\|+\|E\|)$
Adj MATRIX	$O(\|V\|^2)$



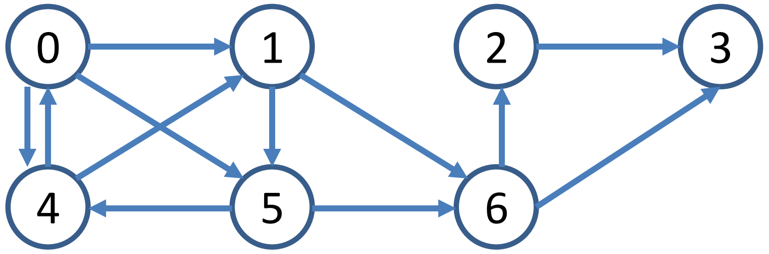
Aggregate time analysis: for each vertex, for each edge

Breadth-First Search (BFS):

- Note that the code above, CLRS22.2 algorithm, assumes that you will only call BFS(G,s) once for s, and not attempt to find other connected components by calling it again for unvisited nodes. =>
- If the graph is NOT connected, you will not reach all vertices when starting from s => time complexity is O, not Θ.

Depth-First Search (DFS) – simple version

```
DFS(G)
    1. For each vertex u of G
        a. color[u] = WHITE
        b. pred[u] = NIL
    2. for (u = 0; u<G.N; u++) // for each vertex u of G
        a. If color[u] == WHITE
            1. DFS_visit(G,u)

DFS_visit(G,u)
    1. color[u] = GRAY
    2. For each v adjacent to u    // explore edge (u,v)
        a. If color[v]==WHITE
            1. pred[v] = u
            2. DFS_visit(G,v)
    3. color[u] = BLACK
```



DFS with time stamps

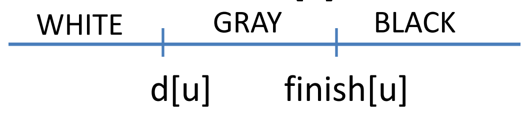
- Next, DFS with ‘time stamps’ of when a node *u* was first discovered (*d[u]*) and the time when the algorithm finished processing that node (*finish[u]*).
- The time is needed for edge labeling (to distinguish between forward and cross edges)
  - tree edge
  - backward edge
  - forward edge
  - cross edge
- The following pseudo-code does not specify the details of the implementation (e.g. if *time* is a global variable or if it will be passed as an argument). The *time* should never decrement (in C it should be passed as a pointer).

Depth-First Search (DFS) (with time stamps) - CLRS

```
DFS(G)
1.  For each vertex u of G
    1.  color[u] = WHITE
    2.  pred[u] = NIL
    3.  d[u] = -1
2.  time = 0
3.  For (u=0;u<G.N;u++) //each vertex u of G –Updated 11/26/18
    1.  If color[u] == WHITE  // u is in undiscovered
        1.  DFS_visit(G,u)

DFS_visit(G,u) // Search graph G starting from vertex u. u must be WHITE.
1.  time = time + 1
2.  d[u] = time        // time when u was discovered
3.  color[u] = GRAY
4.  For each v adjacent to u    // explore edge (u,v)
    1.  If color[v]==WHITE
        1.  pred[v] = u
        2.  DFS_visit(G,v)
5.  color[u] = BLACK
6.  time = time + 1 //no vertex can start and end at the same time
7.  finish[u] = time
```

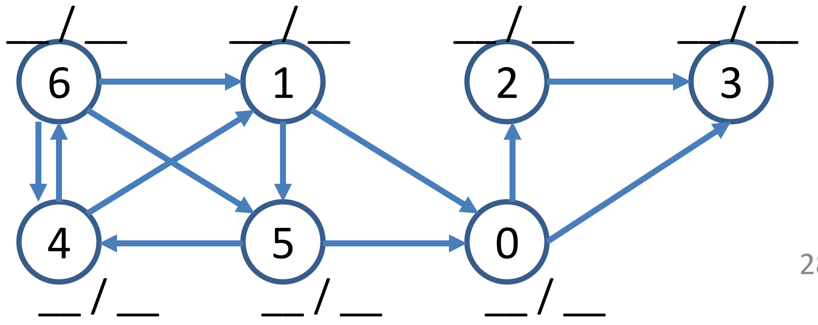
(See CLRS page 605 for step-by-step example.)

Node u will be:
- WHITE before time d[u],
- GRAY between d[u] and finish[u],
- BLACK after finish[u]



Repr\runtime	DFS	DFS-Visit
Adj LIST	$\Theta(\lvert V\rvert+\lvert E\rvert)$	Θ(#neighbors)
Adj MATRIX	$\Theta(\lvert V\rvert^2)$	$\Theta(\lvert V\rvert)$

Order from 1st to last	Visited vertex	Start	Finish	Pred



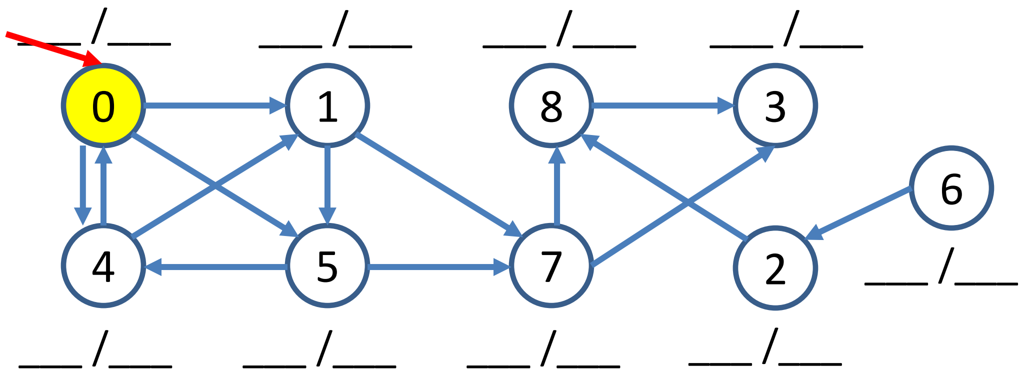
Worksheet

Convention:
<u>start</u> /<u>end</u>
pred

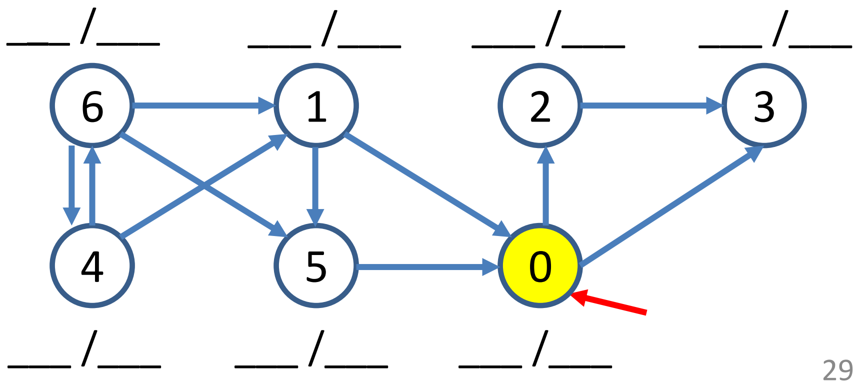
Run DFS on the graphs below. Visit neighbors of u in increasing order.
For each node, write the start and finish times and the predecessor.
Do edge classification as well.

Order from 1st to last	Visited vertex	Start	Finish	Pred

Order from 1st to last	Visited vertex	Start	Finish	Pred



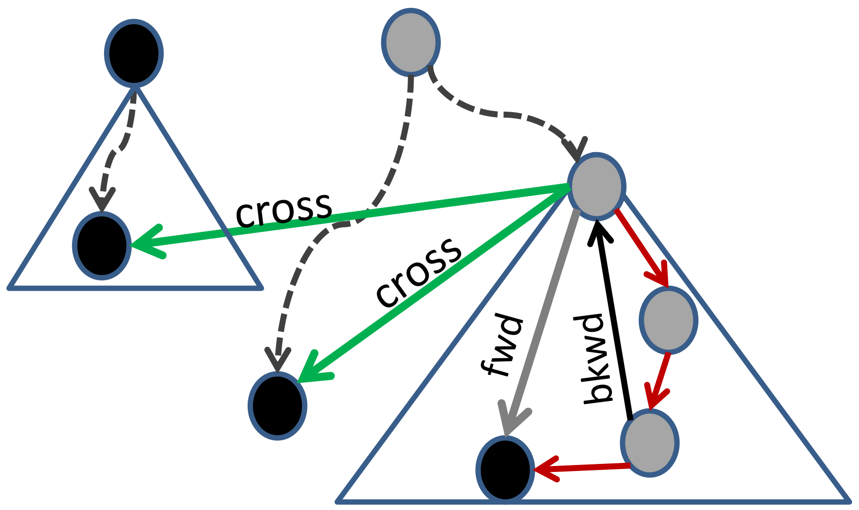
DFS(G):



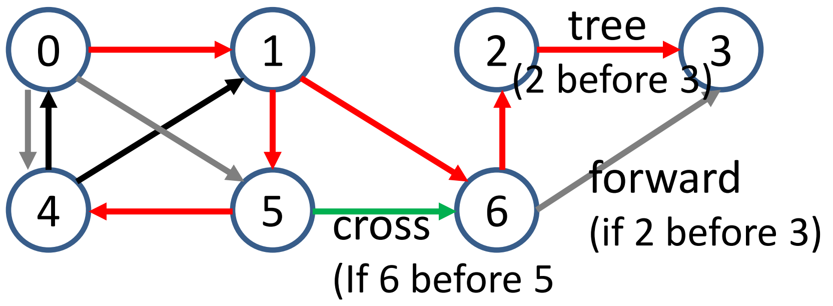
DFS(G) (note that 0 moved)

Edge Classification

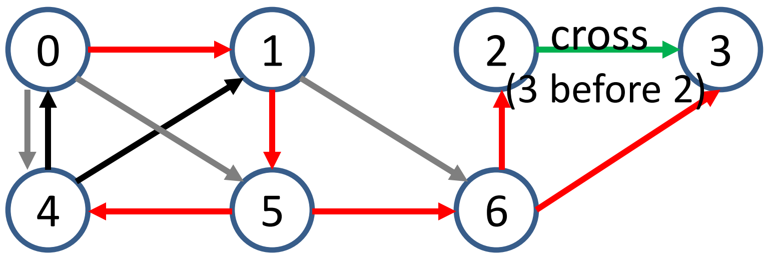
Edge type for (u,v)	**Color of v**	**Arrow**	**Comments**
Tree	White	→	v is discovered
Backward	Gray	→	There is a cycle
Forward	Black	→	Shortcut. v is a descendant of u v started after u started
Cross	Black	→	v is not a descendant of u v started before u started



The edge classification depends on the order in which vertices are discovered, which depends on the order by which neighbors are visited.



DFS(G,0): 0, 1, 6, 2, 3, 5, 4
Visit neighbors clockwise,
starting at 3 o'clock.



DFS(G,0): 0, 1, 5, 4, 6, 3, 2
Visit neighbors in increasing order, but at 6 visit 3 before 2.

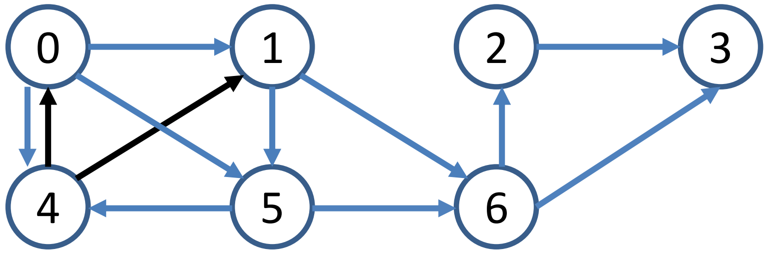
Edge Classification for Undirected Graphs

- An undirected graph will only have:
  - Tree edges
  - Back edges

  - As there is no direction on the edges (can go both ways), what could be a *forward* or a *cross* edge will already have been explored in the other direction, as either a *backward* or a *tree* edge.
    - Forward (u,v) => backward (v,u)
    - Cross (u,v) => tree (v,u)

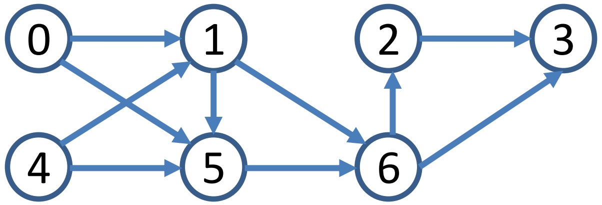
DAGs & Detecting Cycles in a Graph

- A graph has a cycle if a DFS traversal finds a backward edge.
  - Applies to both directed and undirected graphs.
- A Directed Acyclic Graph (DAG) is a directed graph that has no cycles.



Topological Sorting

- Topological sort of a directed **acyclic** graph (DAG), G, is a linear ordering of its vertices s.t. if (u,v) is an edge in G, then u will be listed before v (all edges point from left to right).
  - If a graph has a cycle, it CANNOT have a topological sorting.
- Application:
  - Task ordering (e.g. for an assembly line)
    - Vertices represent tasks
    - Edge (u,v) indicates that task u must finish before task v starts.
    - Topological sorting gives a feasible order for completing the tasks.
  - Identify strongly connected components in directed graphs.



Algorithm version 1:

1. Run DFS and return time finish time data.

 - If cycle found, quit => NO topological order

2. Return array with vertices in reversed order of DFS finish time.

Algorithm version 2 (CLRS):

1. Initialize an empty list L.

2. Call DFS(G) with modification:

 When a vertex finishes (black) add it at the beginning of list L.

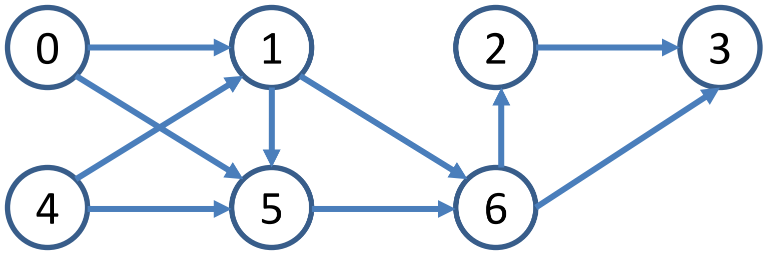
 NOTE: If a cycle is detected (backward edge), return null. => No topological order.

3. Return L

Topological Sorting - Worksheet

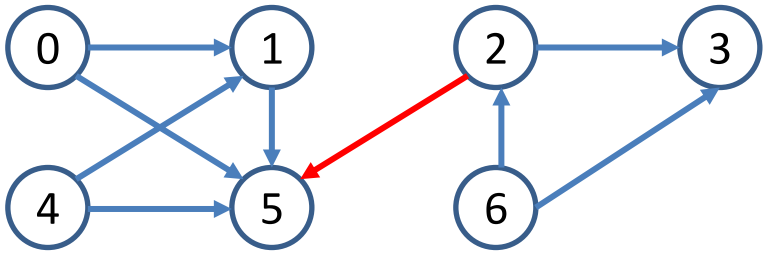
- There may be more than one topological order for a DAG. In that case, any one of those is good.
- Red arrows show what is different from the graph in Example 1.

Example 1



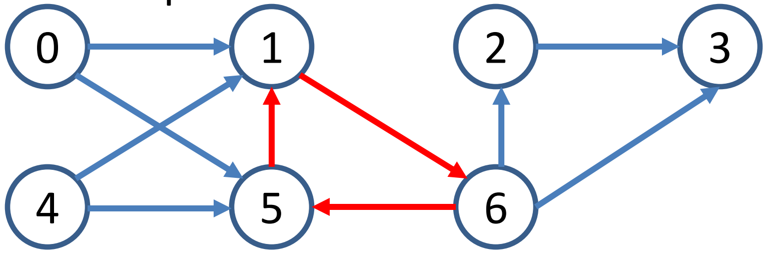
Topological order:

Example 2



Topological order:

Example 3

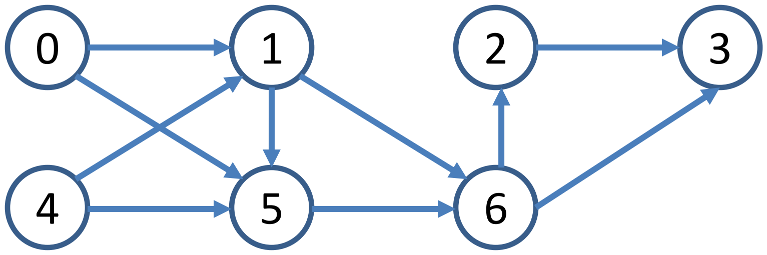


Topological order: none. It has a cycle.

Topological Sorting - Answer

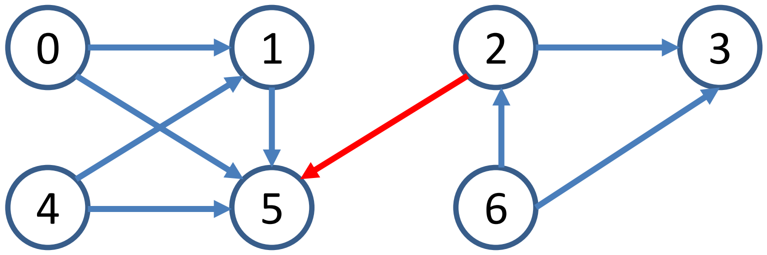
- There may be more than one topological order for a DAG.
  In that case, any one of those is good.
- Red arrows show what is different from the graph in Example 1.

Example 1



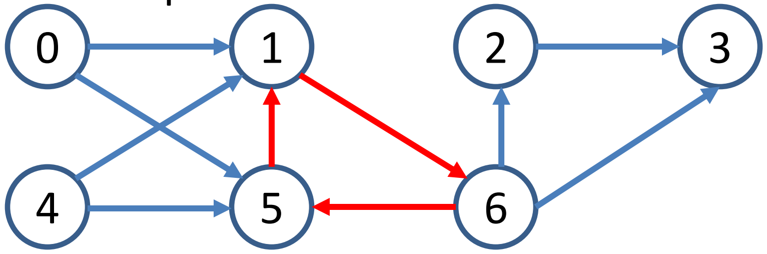
Topological order:
4, 0, 1, 5, 6, 2, 3
(0, 4, 1, 5, 6, 2, 3)

Example 2



Topological order:
6, 4, 2, 3, 0, 1, 5
(0, 4, 1, 6, 2, 3, 5)
(0, 4, 1, 6, 2, 5, 3)

Example 3



Topological order: none. It has a cycle.

Simple pseudocode:

Run DFS and return time finish time data.

- If cycle found, quit => NO topological order

Return array with vertices in reversed order of finish time.

Identifying Strongly Connected Components in a Directed Graph

Strongly_Connected_Components(G)

1. `finish1`= DFS(G) //Call DFS and return the finish time data, `finish1`

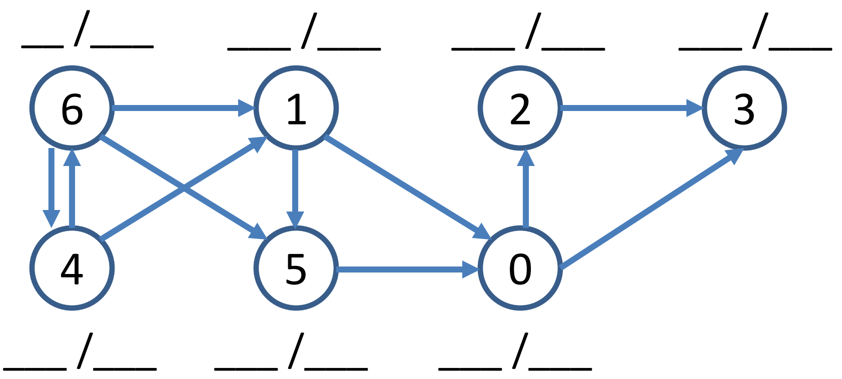
2. Compute G^T

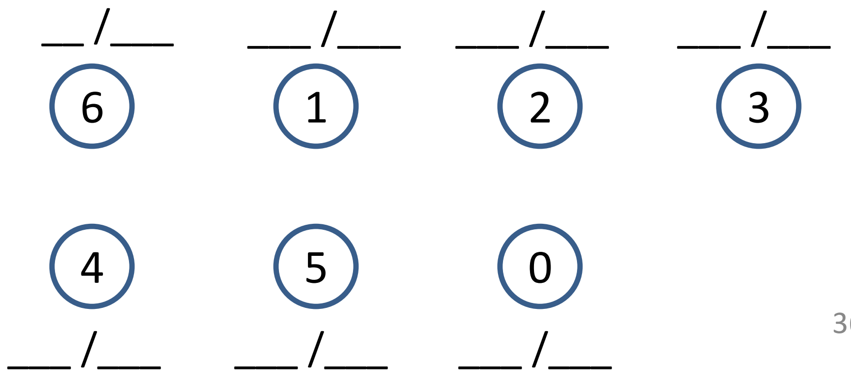
3. Call DFS(G^T), but in its main loop consider the vertices in order of decreasing finish time (`finish1`)

4. Output the vertices of each tree from line 3 as a separate strongly connected component.

Where: $G^T = (V, E^T)$, with $E^T = \{(v,u) : (u,v) \in E\}$

- the *transpose of G*: a graph with the same vertices as G, but with edges in reverse order.





Strongly Connected Components in a Directed Graph Worksheet 1

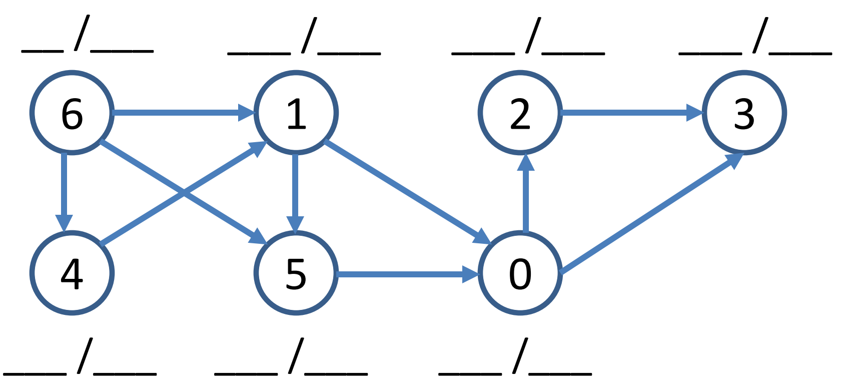
Strongly_Connected_Components(G)

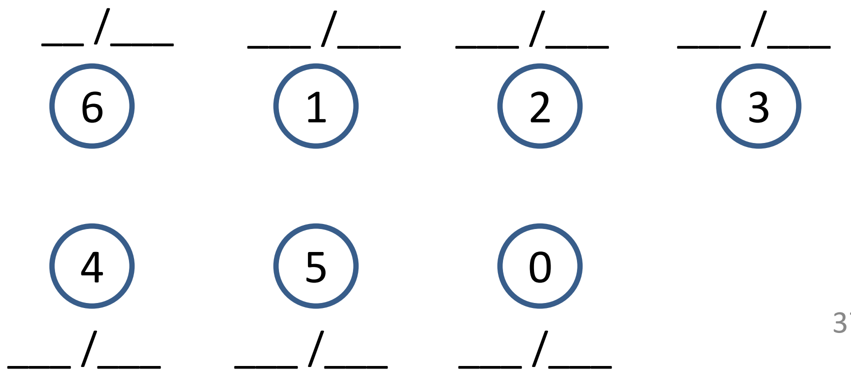
1. `finish1`= DFS(G)
2. Compute G^T
3. Call DFS(G^T), but in its main loop consider the vertices in order of decreasing finish time (`finish1`)
4. Output the vertices of each tree from line 3 as a separate strongly connected component.

Where: $G^T = (V, E^T)$, with $E^T = \{(v,u) : (u,v) \in E\}$

- the *transpose of G*: a graph with the same vertices as G, but with edges in reverse order.

Visited vertex	Start	Finish	Pred





Strongly Connected Components in a Directed Graph Worksheet 2

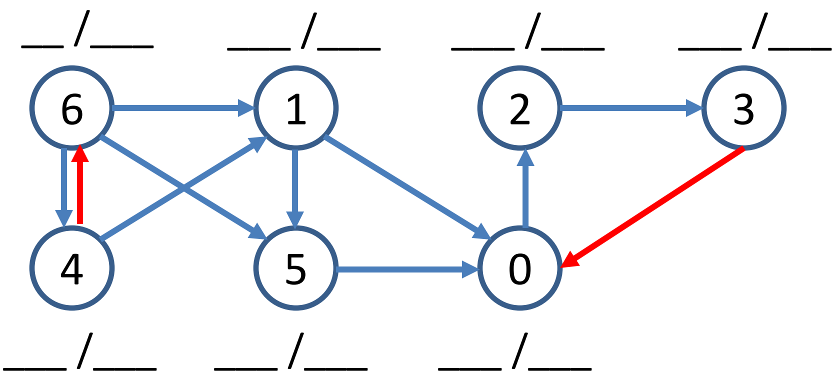
Strongly_Connected_Components(G)

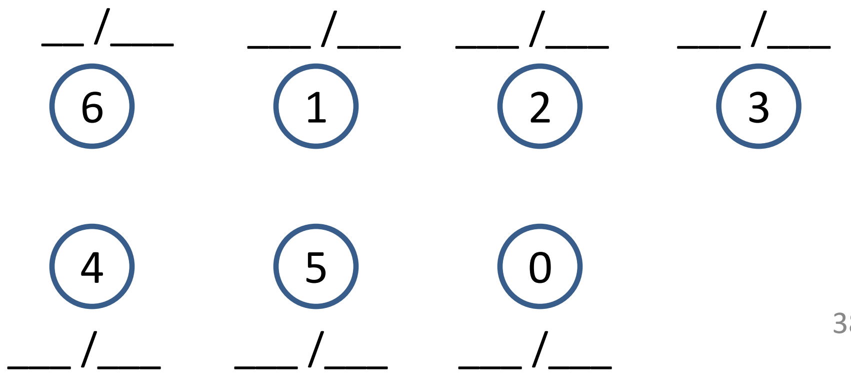
1. `finish1`= DFS(G)
2. Compute G^T
3. Call DFS(G^T), but in its main loop consider the vertices in order of decreasing finish time (`finish1`)
4. Output the vertices of each tree from line 3 as a separate strongly connected component.

Where: $G^T = (V, E^T)$, with $E^T = \{(v,u) : (u,v) \in E\}$

- the *transpose of G*: a graph with the same vertices as G, but with edges in reverse order.

Visited vertex	Start	Finish	Pred

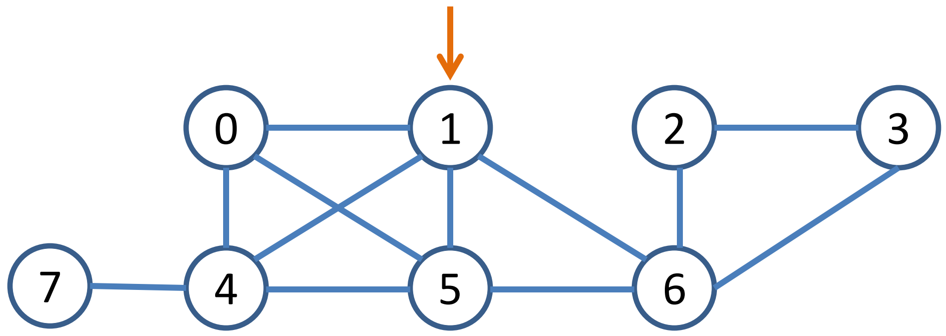


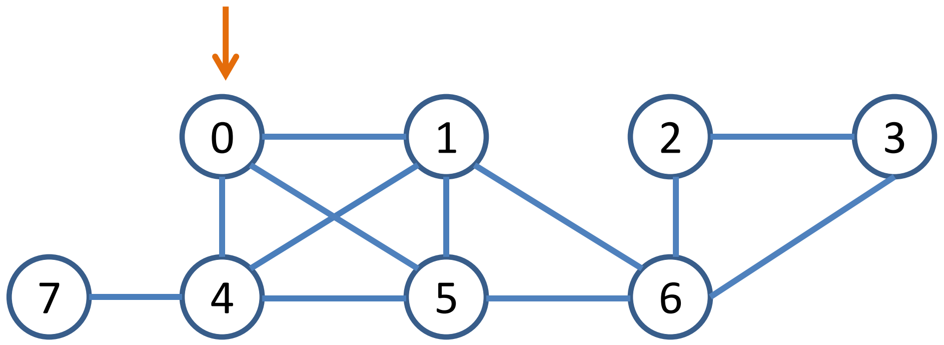


Extra Slides

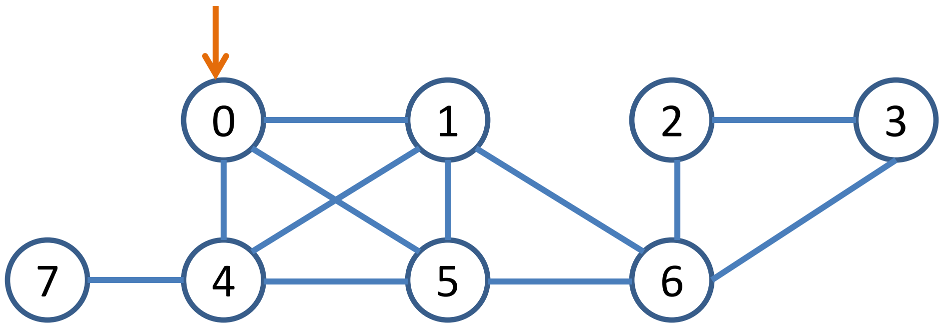
Graph Traversal - Practice

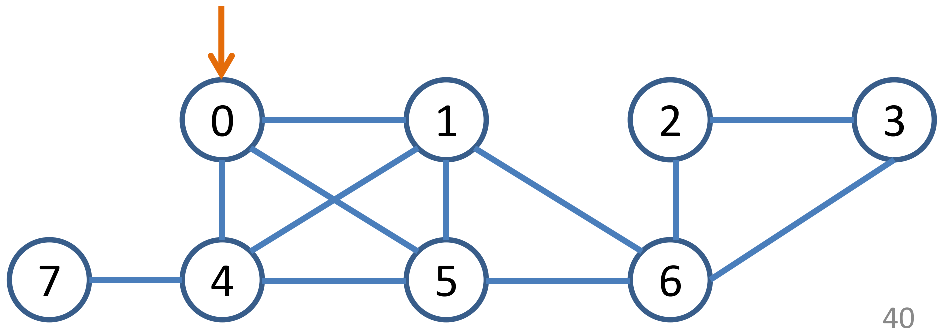
- BFS :





- DFS :

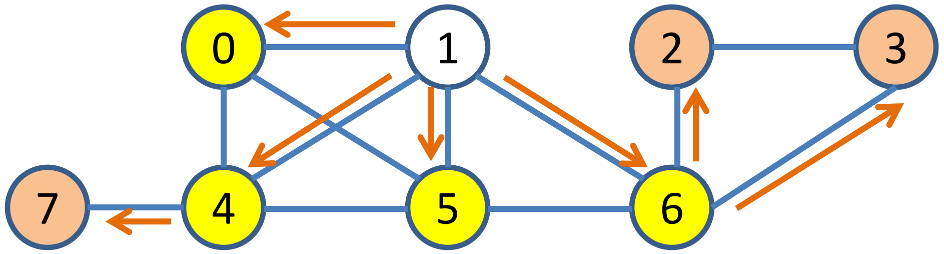


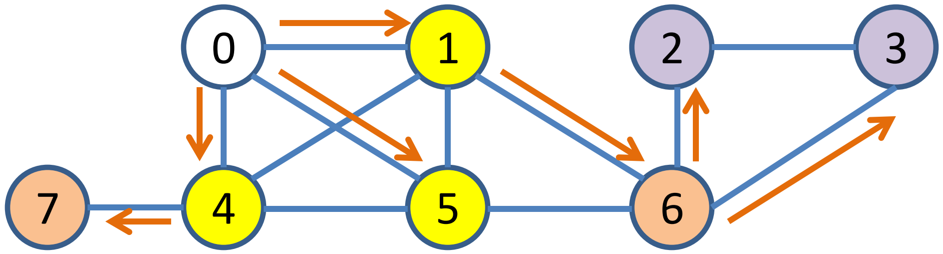


Graph Traversal

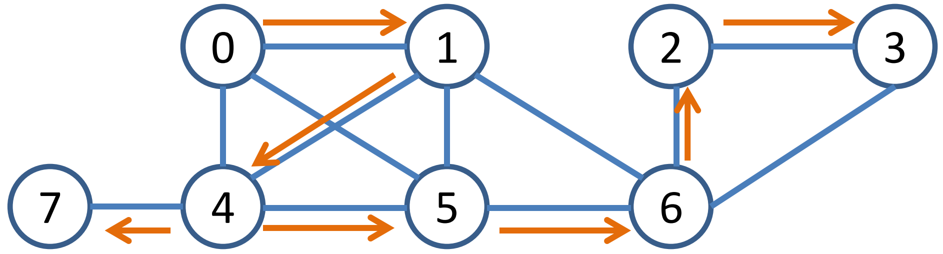
For both DFS and BFS the resulting trees depend on the order in which neighbors are visited.

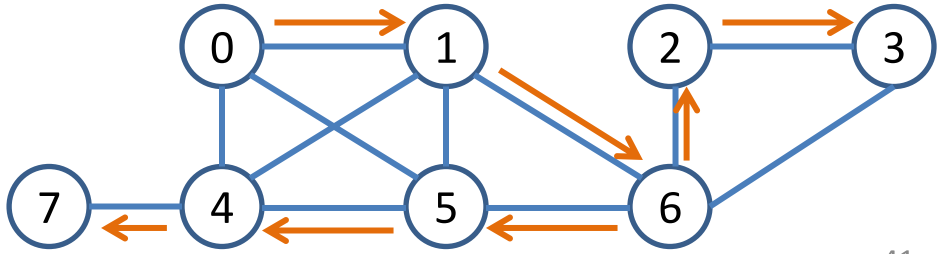
- BFS traversal examples:





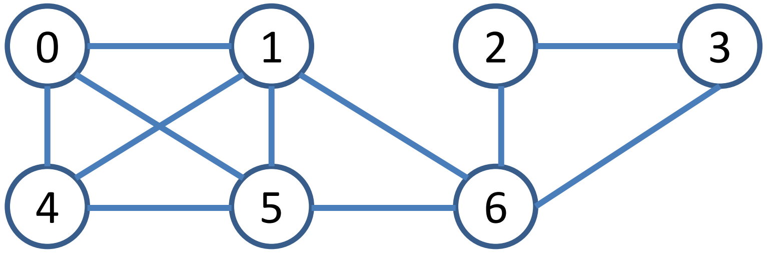
- DFS traversal examples.





DFS Practice

Try various directions for arrows.



Extra Material – Not required

DFS – Non-Recursive

- Sedgewick , Figure 5.34, page 244
- Use a stack instead of recursion
  - Visit the nodes in the same order as the recursive version.
    - Put nodes on the stack in reverse order
    - Mark node as visited when you start processing them from the stack, not when you put them on the stack
    - 2 versions based on what they put on the stack:
      - only the vertex
      - the vertex and a node reference in the adjacency list for that vertex
  - Visit the nodes in different order, but still depth-first.