Graph
Data Structure

Graph Data Structure

Graph data structure is a collection of vertices (nodes) and edges

A vertex represents an entity (object)

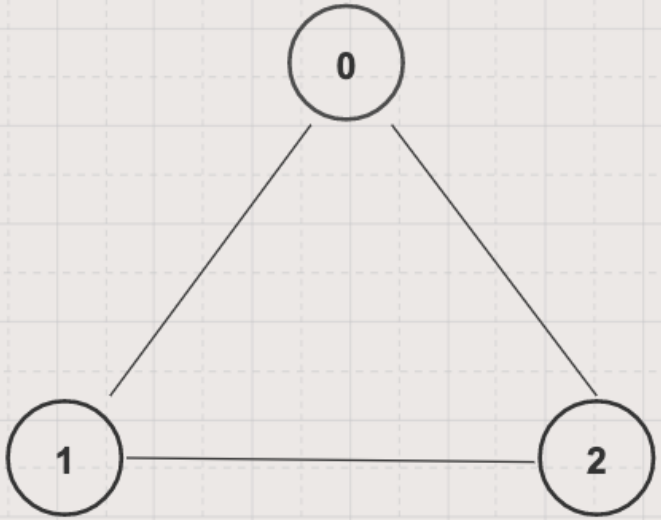
An edge is a line or arc that connects a pair of vertices in the graph, represents the relationship between entities

Examples

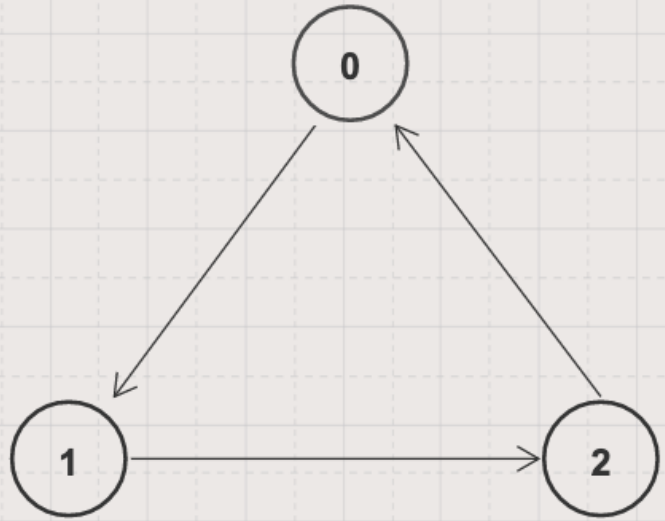
- A computer network is a graph with computers are vertices and network connections between them are edges

- The World Wide Web is a graph with web pages are vertices and hyperlinks are edges

Graphs Types and Terminology



Undirected Graph

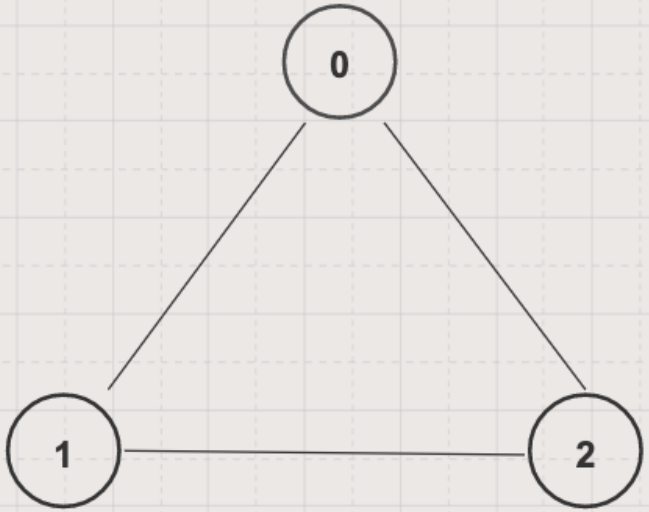


Directed Graph

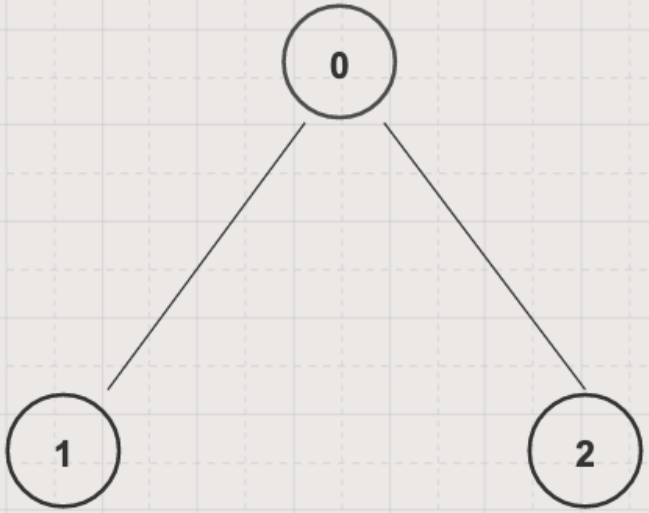
Undirected vs Directed Graph

- An undirected graph has no directed edges.
- Every edge in the undirected graph can be travel in both directions (two-way relationships)

- A directed graph has no undirected edges.
- Every edge in the directed graph can be traveled only in a single direction (one-way relationship)



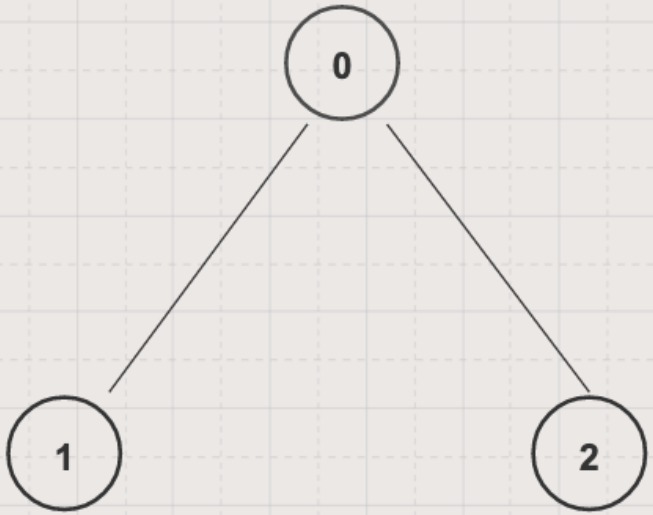
Cyclic Graph



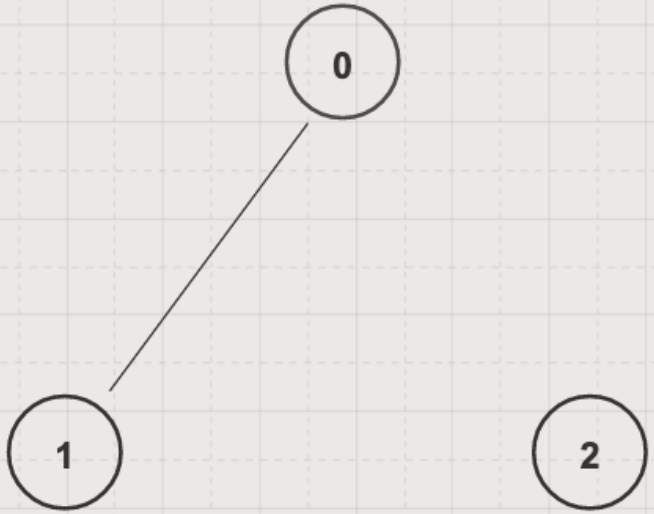
Acyclic Graph

Cyclic vs Acyclic Graph

- A cyclic graph has at least a cycle (existing a path from at least one node back to itself)
- An acyclic graph has no cycles



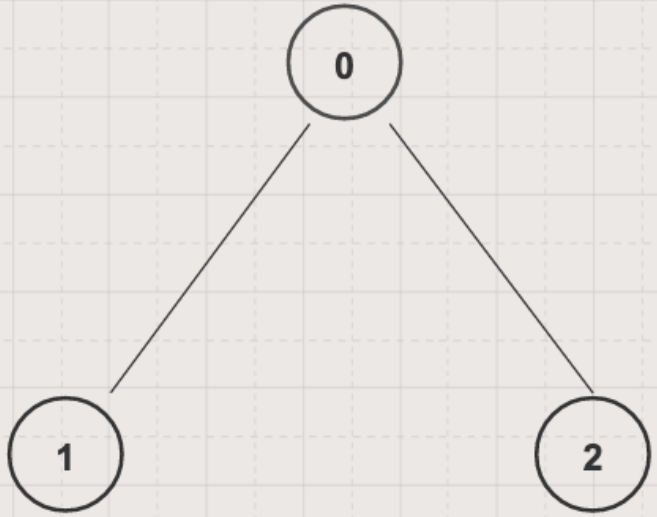
Connected Graph



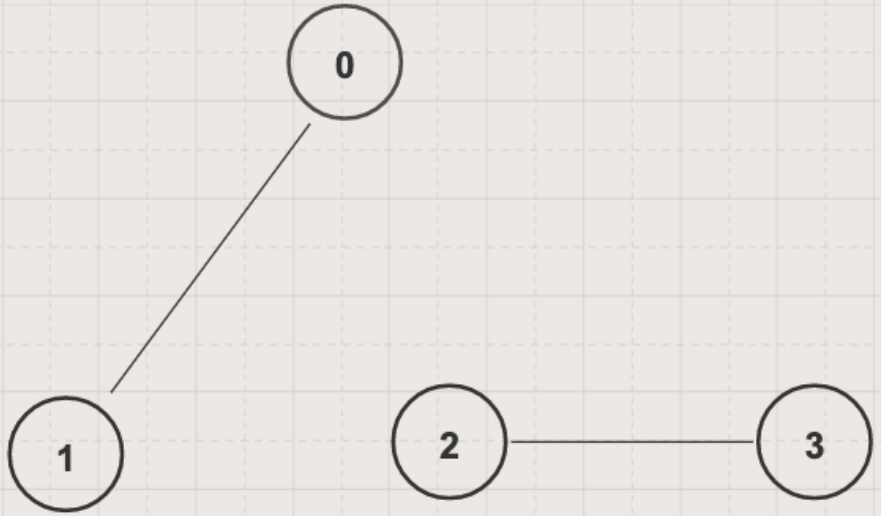
Disconnected Graph

Connected vs Disconnected Graph

- A connected graph has no unreachable vertices (existing a path between every pair of vertices)
- A disconnected graph has at least an unreachable vertex



Tree

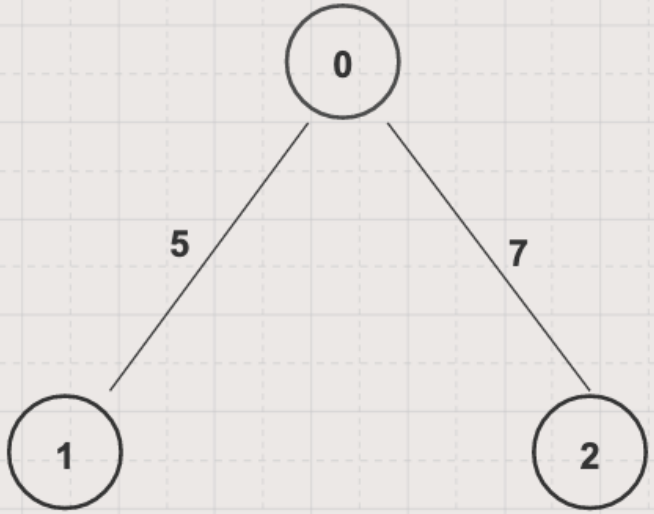


Forest

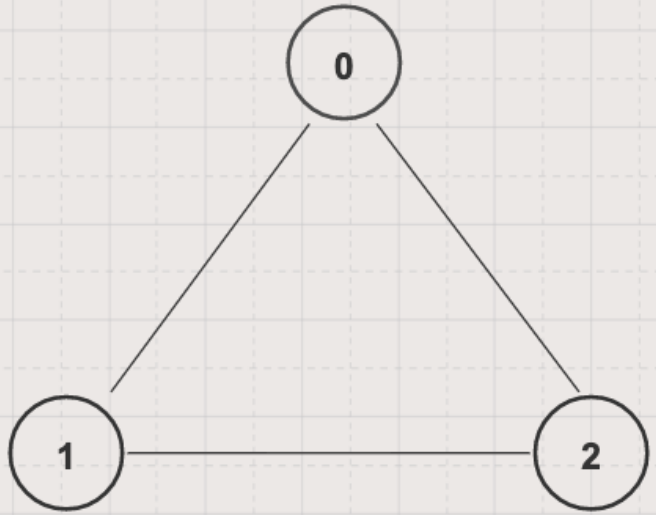
Tree vs Forrest

- A tree is a connected acyclic undirected graph

- A forest is a graph with each connected component a tree



Weighted Graph



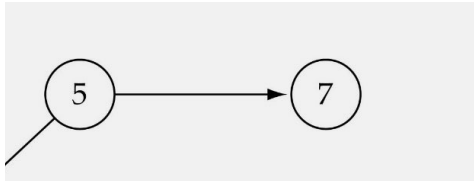
Unweighted Graph

Weighted vs Unweighted Graph

- A weighted graph has a weight attached to each edge (for example, the distance between two vertices)
- An unweighted graph has no weight attached to each edge

Graph Terminology

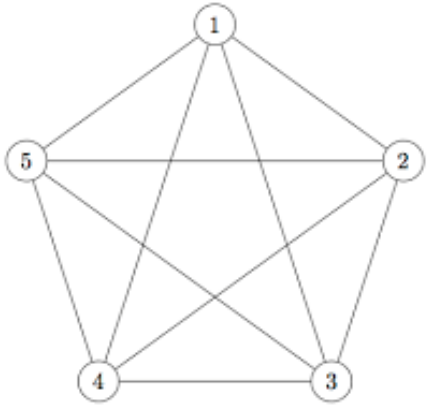
Adjacent nodes: two nodes are adjacent if they are connected by an edge



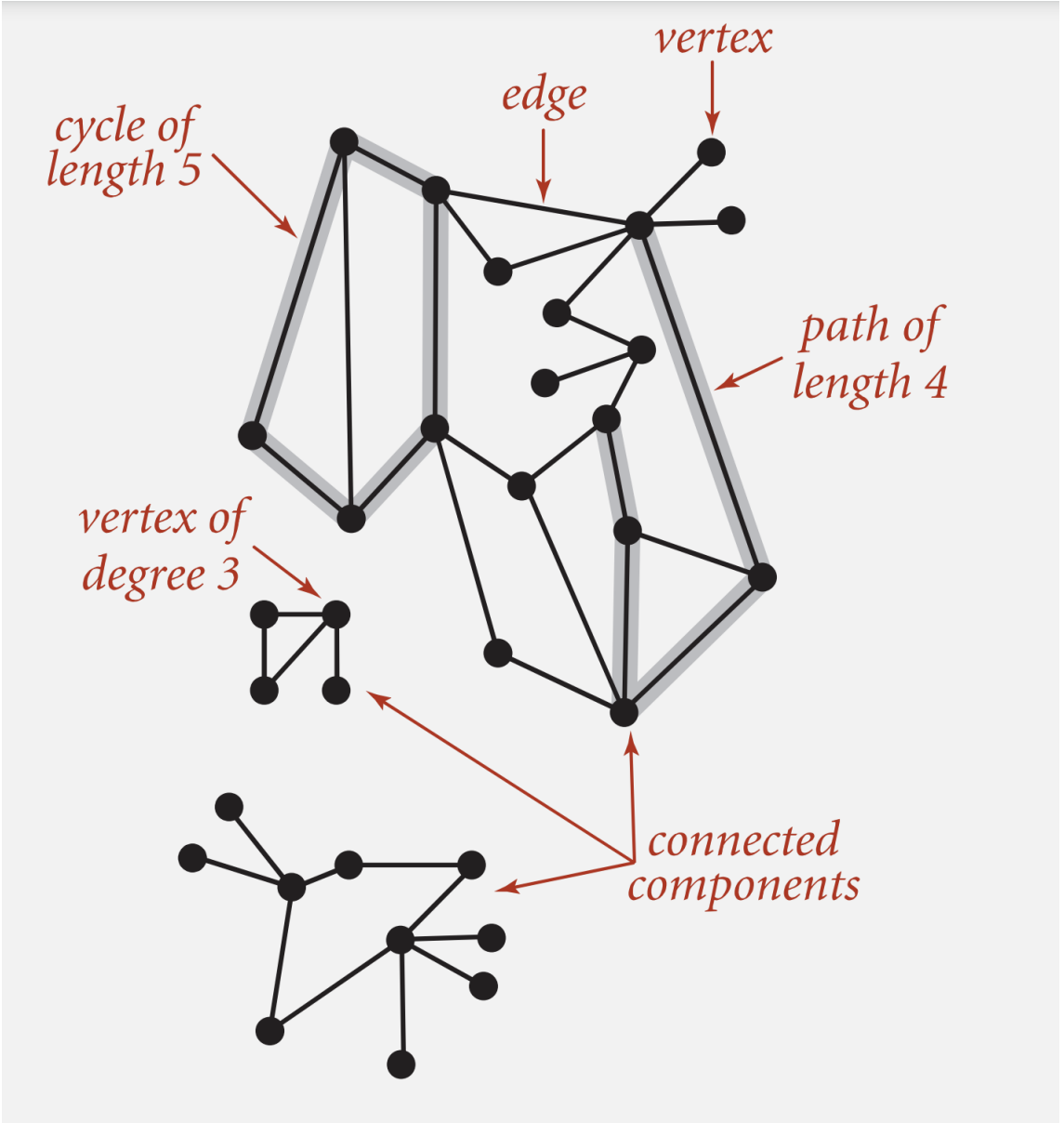
5 is adjacent to 7
7 is adjacent from 5

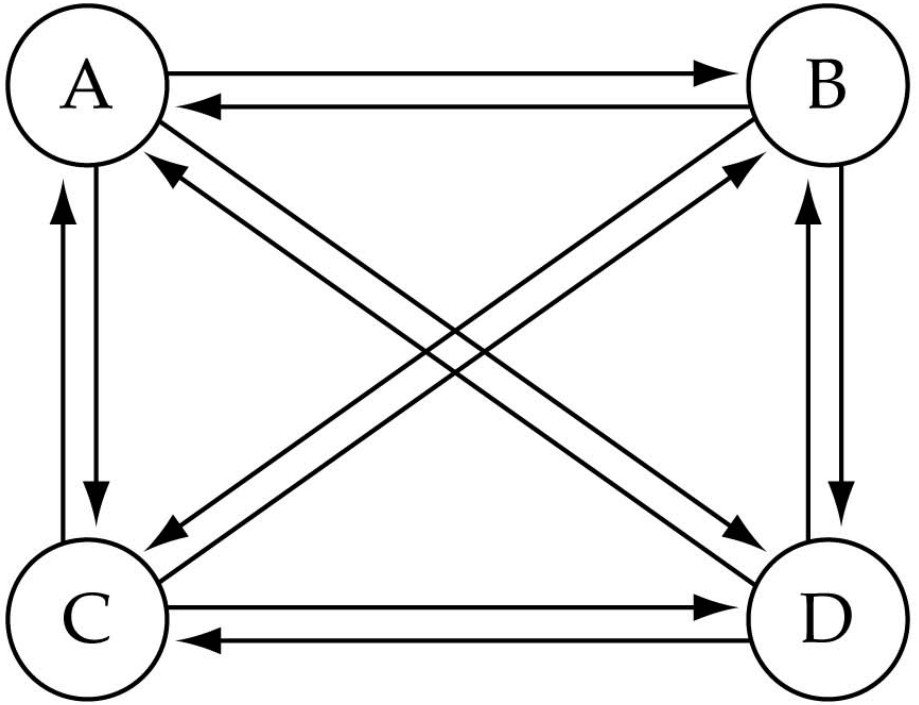
Path: a sequence of vertices that connect two nodes in a graph

Complete graph: a graph in which every vertex is directly connected to every other vertex



Graph Terminology Example



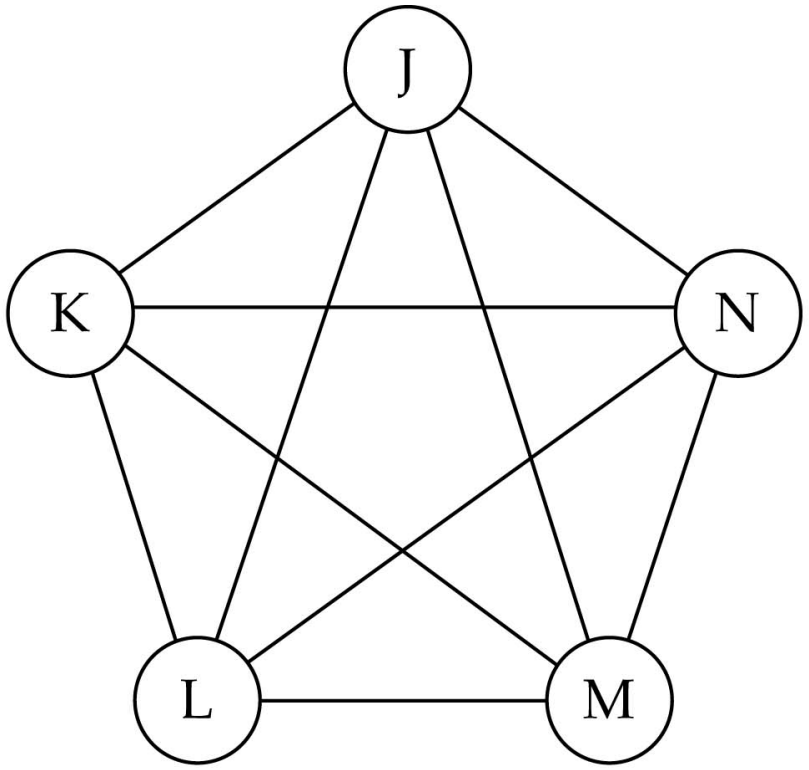


(a) Complete directed graph.

What is the number of edges in a complete directed graph with N vertices?

N * (N-1)

$O(N^2)$



(b) Complete undirected graph.

What is the number of edges in a complete undirected graph with N vertices?

N * (N-1) / 2

$O(N^2)$

Graph Representations

Graph Representations

Incidence (Matrix): Most useful when information about edges is more desirable than information about vertices.

Adjacency (Matrix/List): Most useful when information about the vertices is more desirable than information about the edges. These two representations are also most popular since information about the vertices is often more desirable than edges in most applications

Incidence Matrix (Edge list)

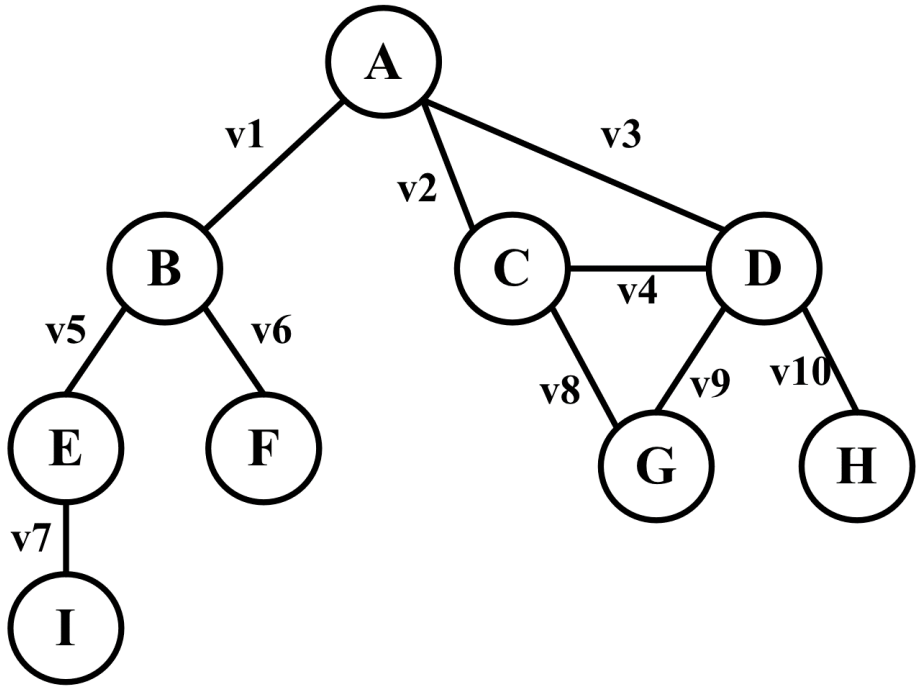
- G = (V, E) be an undirected graph. Suppose that v_1, v_2, v_3, …, v_n are the vertices and e_1, e_2, …, e_m are the edges of G. Then the incidence matrix with respect to this ordering of V and E is the nx m matrix M = [m_{ij}], where

$$m_{ij} = \begin{cases} 1 & \text{when edge } e_j \text{ is incident with } v_i \\ 0 & \text{otherwise} \end{cases}$$

Can also be used to represent :

Multiple edges: by using columns with identical entries, since these edges are incident with the same pair of vertices

Loops: by using a column with exactly one entry equal to 1, corresponding to the vertex that is incident with the loop



	v1	v2	v3	v4	v5	v6	v7	v8	v9	v10
A	1	1	1	0	0	0	0	0	0	0
B	1	0	0	0	1	1	0	0	0	0
C	0	1	0	1	0	0	0	1	0	0
D	0	0	1	1	0	0	0	0	1	1
E	0	0	0	0	1	0	1	0	0	0
F	0	0	0	0	0	1	0	0	0	0
G	0	0	0	0	0	0	0	1	1	0
H	0	0	0	0	0	0	0	0	0	1
I	0	0	0	0	0	0	1	0	0	0

Representation Example: G = (V, E)

Incidence Matrix

Adjacency Matrix

- There is an N x N matrix, where |V| = N , the Adjacent Matrix (NxN) A = $[a_{ij}]$

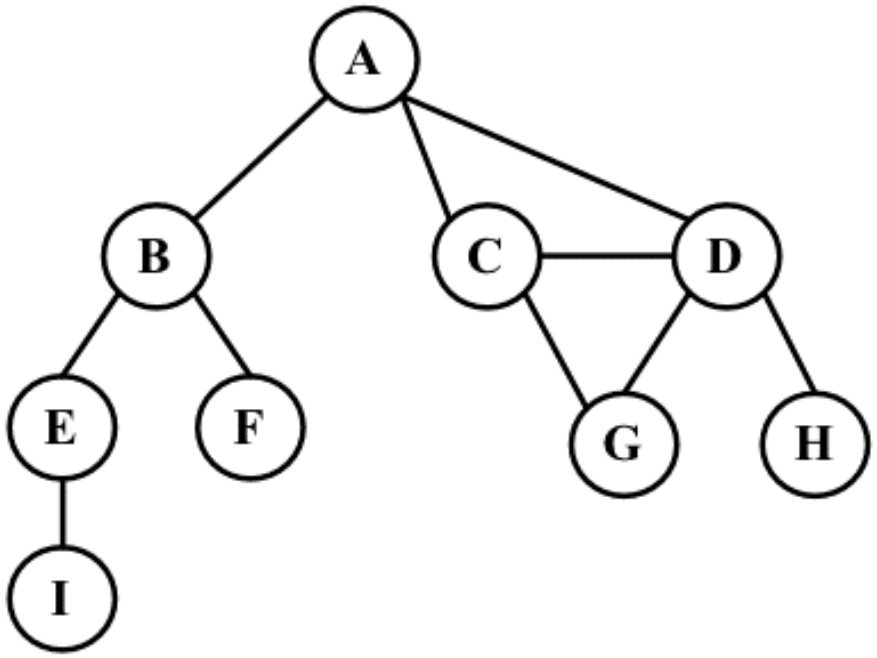
For undirected graph

$$a_{ij} = \begin{cases} 1 & \text{if } \{v_i, v_j\} \text{ is an edge of } G \\ 0 & \text{otherwise} \end{cases}$$

For directed graph

$$a_{ij} = \begin{cases} 1 & \text{if } (v_i, v_j) \text{ is an edge of } G \\ 0 & \text{otherwise} \end{cases}$$

- This makes it easier to find subgraphs, and to reverse graphs if needed.



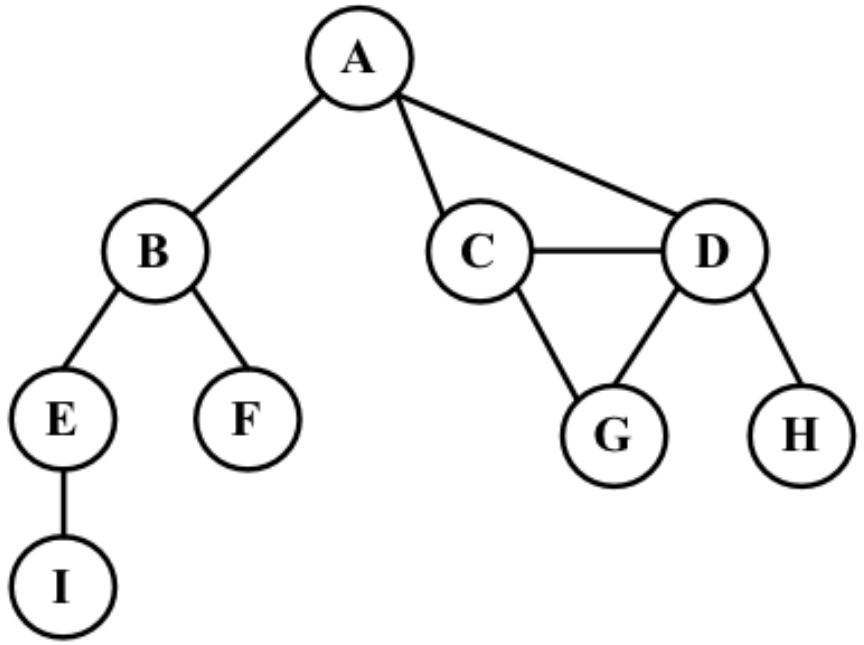
	A	B	C	D	E	F	G	H	I
A	0	1	1	1	0	0	0	0	0
B	1	0	0	0	1	1	0	0	0
C	1	0	0	1	0	0	1	0	0
D	1	0	1	0	0	0	1	1	0
E	0	1	0	0	0	0	0	0	1
F	0	1	0	0	0	0	0	0	0
G	0	0	1	1	0	0	0	0	0
H	0	0	0	1	0	0	0	0	0
I	0	0	0	0	1	0	0	0	0

Example: Undirected Graph G (V, E)

Adjacency Matrix

Adjacency List

- Each node (vertex) has a list of which nodes (vertex) it is adjacent



A	B	C	D		
B	A	E	F		
C	A	D	G		
D	A	C	G	H	
E	B	I			
F	B				
G	C	D			
H	D				
I	E				

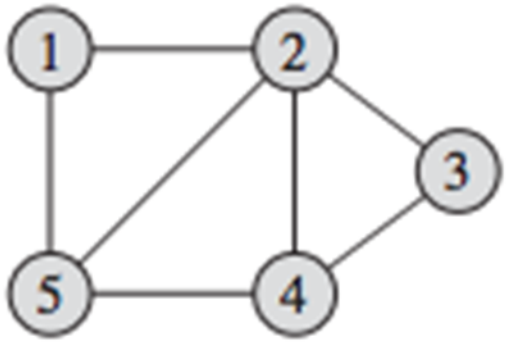
Example: undirectd graph G (V, E)

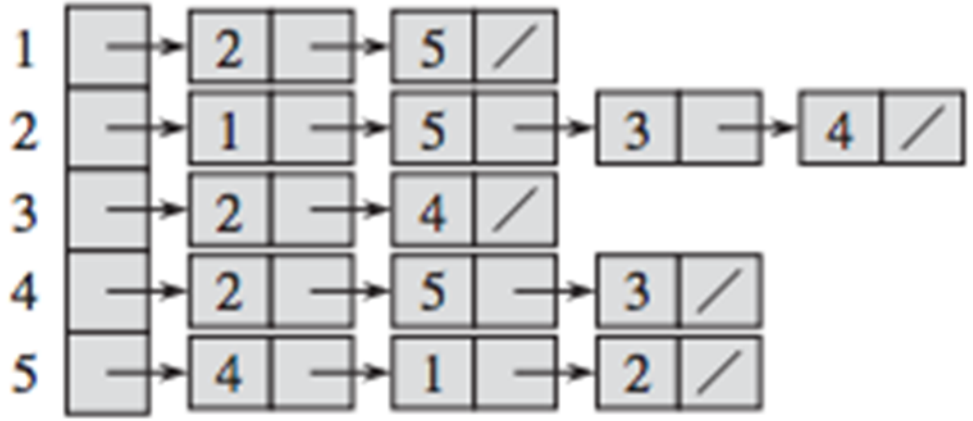
Undirected graph implementation example with an adjacency list in Java

Q: What could be the possible changes to implement the directed graph?

A: addEdge in an undirected graph adds two-ways direction while in a directed graph adds one-way direction

```java
import java.util.ArrayList;
import java.util.List;

public class GraphUndirectedByAdjacencyList {
    private int V;
    private List<List<Integer>> adjacencyList;

    public GraphUndirectedByAdjacencyList(int V) {
        this.V = V;

        adjacencyList = new ArrayList<>(V);
        for (int i = 0; i < V; i++) {
            adjacencyList.add(new ArrayList<>());
        }
    }

    public Integer getV() {
        return this.V;
    }

    public List<List<Integer>> getAdjacencyList() {
        return this.adjacencyList;
    }

    public void addEdge(int source, int dest) {
        adjacencyList.get(source).add(dest);
        adjacencyList.get(dest).add(source);
    }

    public void printAdjacencyList() {
        for (int i = 0; i < V; i++) {
            System.out.printf("Adjacency list of vertex %d is %s %s", i,
                adjacencyList.get(i), System.lineSeparator());
        }
    }

    public static void main(String[] args) {
        GraphUndirectedByAdjacencyList graph = new GraphUndirectedByAdjacencyList(3);
        graph.addEdge(0, 1);
        graph.addEdge(1, 2);
        graph.addEdge(2, 0);
        graph.printAdjacencyList();
    }
}
```





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

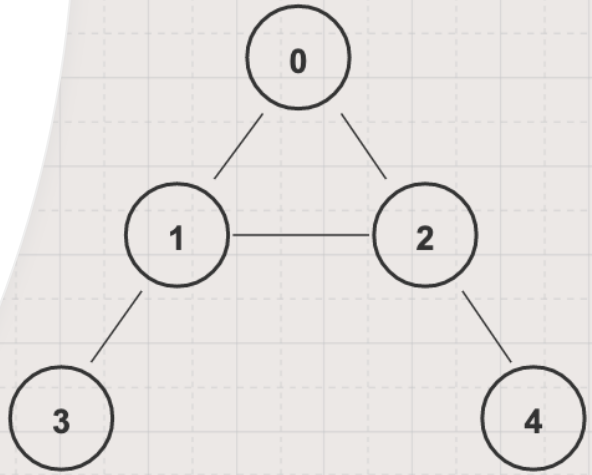
Exercise: Discover both the adjacency list and matrix for the above graph?

Graph Traversal and Searching

Depth-First and Breadth-First Search

- Depth-First Search (DFS) and Breadth-First Search (BFS) are algorithms for traversing or searching a graph
- DFS starts at an arbitrary node and explores as far as possible along each branch before backtracking
- BFS starts at an arbitrary node and explores all of the neighbor nodes at the current level before moving on to the next level

- DFS is better when the target is far from the source while BFS is better when the target is close to the source
- DFS uses a Stack while BFS uses a Queue data structure to backtrack vertices on the traversal path
- DFS consumes less memory than BFS as BFS has to store all the children at each level



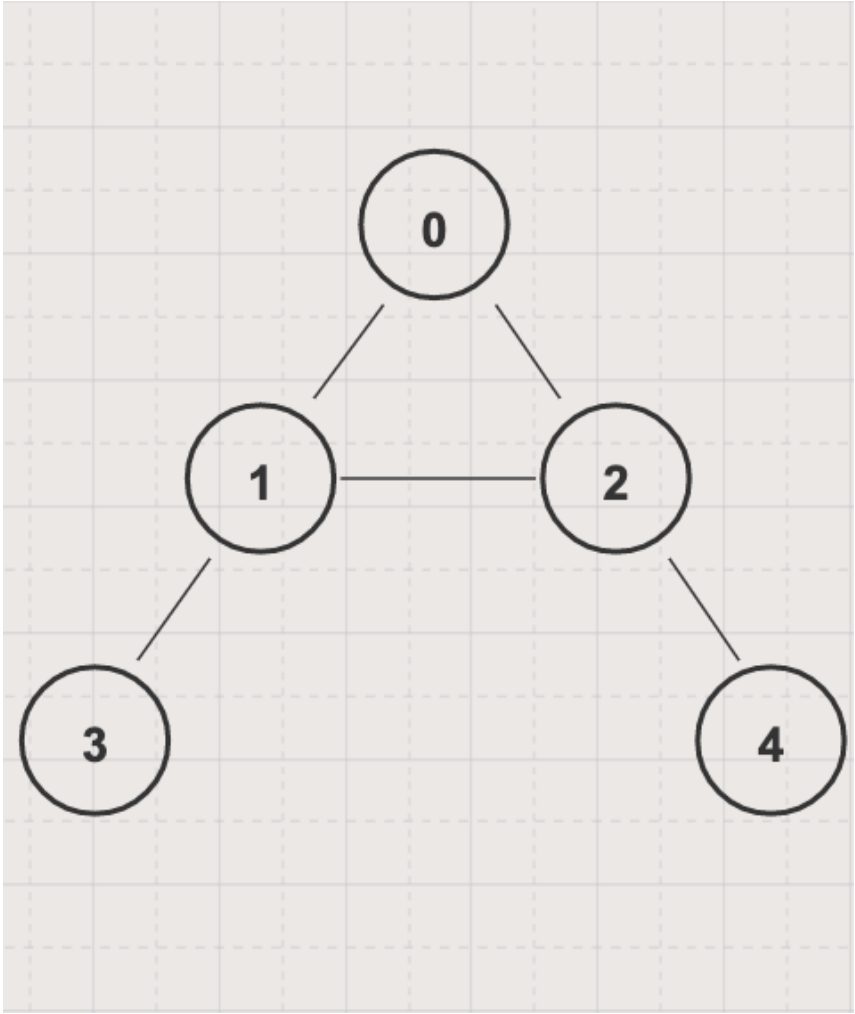
DFS Traversal: 0 -> 1 -> 3 -> 2 -> 4

BFS Traversal: 0 -> 1 -> 2 -> 3 -> 4

Depth First Search on Graph with Iterative and Recursive in Java

Problem

- Give an undirected/directed graph G(V, E)
- Write a Depth-First search algorithm to print out each vertex value exactly once
- For this graph with 0 as the starting vertex, assuming that the left edges are chosen before the right edges, the DFS traversal order will be 0 -> 1 -> 3 -> 2 -> 4



DFS Approach 1: Iterative

- Use an array to track visited nodes to avoid processing a node more than once
- Use a stack to track which nodes to visit next
- Time complexity: O(V+E)
- Space complexity: O(V)

```java
import java.util.ArrayDeque;
import java.util.Deque;

public class DFSByIterative {
    static void dfsByIterative(GraphUndirectedByAdjacencyList g, int v) {
        boolean[] visited = new boolean[g.getV()];

        Deque<Integer> stack = new ArrayDeque<>();
        stack.push(v);

        while (!stack.isEmpty()) {
            v = stack.pop();

            if (!visited[v]) {
                visited[v] = true;
                System.out.printf("%d ", v);

                for(Integer w : g.getAdjacencyList().get(v)) {
                    stack.push(w);
                }
            }
        }
    }

    public static void main(String[] args) {
        GraphUndirectedByAdjacencyList g = new GraphUndirectedByAdjacencyList(5);
        g.addEdge(0, 2);
        g.addEdge(1, 2);
        g.addEdge(1, 0);
        g.addEdge(1, 3);
        g.addEdge(2, 4);

        dfsByIterative(g, 0);
    }
}
```

DFS Approach 2: Iterative with Color

- Use a color array to track vertex states. Each vertex can have 3 states marked by color
  - White represents unvisited
  - Gray represents a visit in progress
  - Black represents visited
- Use a stack to track which nodes to visit next

- Time complexity: O(V+E)
- Space complexity: O(V)

```java
import java.util.ArrayDeque;
import java.util.Deque;

public class DFSByIterativeWithColor {
    static final int WHITE = 0, GRAY = 1, BLACK = 2;

    static void dfsByIterativeWithColor(GraphUndirectedByAdjacencyList g, int v) {
        int[] color = new int[g.getV()];

        Deque<Integer> stack = new ArrayDeque<>();
        stack.push(v);

        while (!stack.isEmpty()) {
            v = stack.pop();

            if (color[v] == WHITE) {
                color[v] = GRAY;
                System.out.printf("%d ", v);

                for(Integer w : g.getAdjacencyList().get(v)) {
                    stack.push(w);
                }

                color[v] = BLACK;
            }
        }
    }

    public static void main(String[] args) {
        GraphUndirectedByAdjacencyList g = new GraphUndirectedByAdjacencyList(5);
        g.addEdge(0, 2);
        g.addEdge(1, 2);
        g.addEdge(1, 0);
        g.addEdge(1, 3);
        g.addEdge(2, 4);

        dfsByIterativeWithColor(g, 0);
    }
}
```

DFS Approach 3: Recursive

- Use an array to track visited nodes to avoid processing a node more than once
- Instead of using a stack, the DFS algorithm calls to itself to explore unvisited vertices
- Time complexity: O(V+E)
- Space complexity: O(V)

```java
public class DFSByRecursive {

    static void dfs(GraphUndirectedByAdjacencyList g, int v, boolean[] visited) {
        visited[v] = true;
        System.out.printf("%d ", v);

        for (Integer w : g.getAdjacencyList().get(v)) {
            if (!visited[w]) {
                dfs(g, w, visited);
            }
        }
    }

    static void traversal(GraphUndirectedByAdjacencyList g) {
        boolean[] visited = new boolean[g.getV()];

        for (int i = 0; i < g.getV(); i++) {
            if (!visited[i]) {
                dfs(g, i, visited);
            }
        }
    }

    public static void main(String[] args) {
        GraphUndirectedByAdjacencyList g = new GraphUndirectedByAdjacencyList(5);
        g.addEdge(0, 2);
        g.addEdge(1, 2);
        g.addEdge(1, 0);
        g.addEdge(1, 3);
        g.addEdge(2, 4);

        traversal(g);
    }
}
```

DFS Approach 4: Recursive with Color

- Use a color array to track vertex states. Each vertex can have 3 states marked by color
  - White represents unvisited
  - Gray represents a visit in progress
  - Black represents visited
- Instead of using a stack, the DFS algorithm calls to itself to explore White vertices

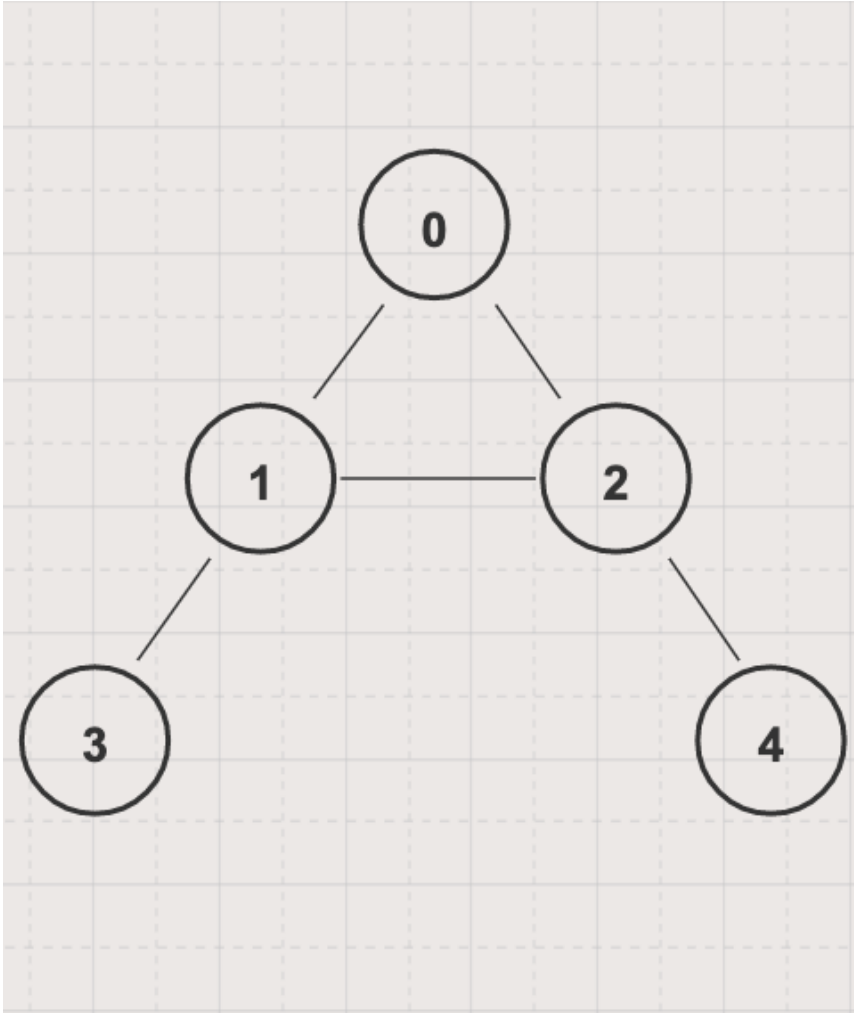
- Time complexity: O(V+E)
- Space complexity: O(V)

```java
public class DFSByRecursiveWithColor {
    static final int WHITE = 0, GRAY = 1, BLACK = 2;

    static void dfsByRecursiveWithColor(GraphUndirectedByAdjacencyList g, int v, int[] color) {
        color[v] = GRAY;
        System.out.printf("%d ", v);

        for (Integer w : g.getAdjacencyList().get(v)) {
            if (color[w] == WHITE) {
                dfsByRecursiveWithColor(g, w, color);
            }
        }
        color[v] = BLACK;
    }

    static void traversal(GraphUndirectedByAdjacencyList g) {
        int[] color = new int[g.getV()];

        for (int i = 0; i < g.getV(); i++) {
            if (color[i] == WHITE) {
                dfsByRecursiveWithColor(g, i, color);
            }
        }
    }

    public static void main(String[] args) {
        GraphUndirectedByAdjacencyList g = new GraphUndirectedByAdjacencyList(5);
        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(1, 2);
        g.addEdge(1, 3);
        g.addEdge(2, 4);

        traversal(g);
    }
}
```

Breadth First Search Algorithm on Graph in Java

Problem

- Give an undirected/directed graph G(V, E) where V is the list of vertex and E is the list of edges
- Write a Breadth-First search algorithm to print out each vertex value exactly once
- For this graph with 0 as the starting vertexassuming that the left edges are chosen before the right edges, the BFS traversal order will be 0 -> 1 -> 2 -> 3 -> 4



BFS Approach 1: Boolean Array and FIFO Queue

- Create a Boolean array to track visited nodes to avoid processing a node more than once. Each node can have 2 states: visited or unvisited
- Create a FIFO queue to track which nodes to visit next on the traversal path. Add the starting node into the queue
- Do the following while the queue is still containing items
  - Retrieves and removes a node from the queue
  - If the node has yet to be visited, change its state to visited in the Boolean array, print out its value, and add all of its neighbors' nodes into the queue

```java
import java.util.ArrayDeque;
import java.util.Deque;

public class BFSByIterative {
    public static void bfsByIterative(GraphUndirectedByAdjacencyList g, int v) {
        boolean[] visited = new boolean[g.getV()];
        Deque<Integer> queue = new ArrayDeque<>();
        queue.offer(v);

        while (!queue.isEmpty()) {
            v = queue.poll();

            if (!visited[v]) {
                visited[v] = true;
                System.out.printf("%d ", v);

                for (Integer w : g.getAdjacencyList().get(v)) {
                    queue.offer(w);
                }
            }
        }
    }

    public static void main(String[] args) {
        GraphUndirectedByAdjacencyList g = new GraphUndirectedByAdjacencyList(5);
        g.addEdge(0, 1);
        g.addEdge(1, 2);
        g.addEdge(2, 0);
        g.addEdge(1, 3);
        g.addEdge(2, 4);

        bfsByIterative(g, 0);
    }
}
```

Time complexity is O(V+E) and space complexity is O(V)

BFS Approach 2: Color Array and FIFO Queue

- Create a color array to track vertex states. Each vertex can have 3 states marked by color
  - White represents unvisited
  - Gray represents a visit in progress
  - Black represents visited
- Create a FIFO queue to track which nodes to visit next on the traversal path
- Do the following while the queue is still containing items
  - Retrieves and removes a node from the queue
  - If the node has white color, change it to gray, print out its value, add all of its neighbors' nodes into the queue, and change the node color to black

```java
import java.util.ArrayDeque;
import java.util.Deque;

public class BFSByIterativeWithColor {
    static final int WHITE = 0, GRAY = 1, BLACK = 2;

    public static void bfsByIterativeWithColor(GraphUndirectedByAdjacencyList g, int v) {
        int[] color = new int[g.getV()];
        Deque<Integer> queue = new ArrayDeque<>();
        queue.offer(v);

        while (!queue.isEmpty()) {
            v = queue.poll();

            if (color[v] == WHITE) {
                color[v] = GRAY;
                System.out.printf("%d ", v);

                for (Integer w : g.getAdjacencyList().get(v)) {
                    queue.offer(w);
                }

                color[v] = BLACK;
            }
        }
    }

    public static void main(String[] args) {
        GraphUndirectedByAdjacencyList g = new GraphUndirectedByAdjacencyList(5);
        g.addEdge(0, 1);
        g.addEdge(1, 2);
        g.addEdge(2, 0);
        g.addEdge(1, 3);
        g.addEdge(2, 4);

        bfsByIterativeWithColor(g, 0);
    }
}
```

Time complexity is O(V+E) and space complexity is O(V)

- This Java program, to perform the BFS traversal of a given undirected graph in the form of the adjacency matrix and check for the connectivity of the graph. The BFS traversal makes use of a queue.

```
$javac UndirectedConnectivityBFS.java
$java UndirectedConnectivityBFS
Enter the number of nodes in the graph
5
Enter the adjacency matrix
0 1 1 1 0
0 0 1 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
Enter the source for the graph
1
The graph is disconnected
```

Practical Exercise #1: Write Java Program to Check whether Undirected Graph is Connected using BFS?

- This Java program, to perform the DFS traversal on the given undirected graph represented by a adjacency matrix to check connectivity. The DFS traversal makes use of a stack.

```
$javac UndirectedConnectivityDfs.java
$java UndirectedConnectivityDfs
Enter the number of nodes in the graph
5
Enter the adjacency matrix
0 1 0 1 0
0 0 1 0 0
0 0 0 0 0
0 1 0 0 0
0 0 0 0 0
Enter the source for the graph
1
The graph is disconnected
```

Practical Exercise #2: Write Java Program to Check whether Undirected Graph is Connected using DFS?

- This Java program is to check whether graph is Biconnected. In graph theory, a biconnected graph is a connected and “non-separable” graph, meaning that if any vertex were to be removed, the graph will remain connected. Therefore a biconnected graph has no articulation vertices.

```
$javac BiConnectedGraph.java
$java BiConnectedGraph
Enter the number of nodes in the graph
5
Enter the adjacency matrix
0 1 1 1 0
1 0 1 0 0
1 1 0 0 1
1 0 0 0 1
0 0 1 1 0
Enter the source for the graph
1
The Given Graph is BiConnected
```

Practical Exercise #3: Write Java Program to Check whether Graph is Biconnected?

Practical Exercise #4: Write Java Program to Check if an Undirected Graph is a Tree or Not Using DFS?

- This is a java program to check if graph is tree or not. Graph is tree if,
  - It has number of edges one less than number of vertices.
  - Graph is connected.
  - There are no cycles.

```
Enter the number of vertices:
4
Enter the number of edges:
3
Enter the edges: <from> <to>
1 2
1 3
2 4
1:  3 2
2:  4 1
3:  1
4:  2
Graph is a Tree, as graph is connected and Euler's criterion is satisfied.
```

```
$ javac CheckUndirectedGraphisTree.java
$ java CheckUndirectedGraphisTree

Enter the number of vertices:
6
Enter the number of edges:
7
Enter the edges: <from> <to>
1 2
2 3
2 4
4 5
5 6
6 4
6 3
1:  2
2:  4 3 1
3:  6 2
4:  6 5 2
5:  6 4
6:  3 4 5
Graph is not a Tree, as Euler's criterion is not satisfied
```

Practical Exercise #5: Write Java Program to Find the Connected Components of an Undirected Graph?

- This is a java program In graph theory, a connected component (or just component) of an undirected graph is a subgraph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices in the super graph.

- A graph that is itself connected has exactly one connected component, consisting of the whole graph.

```
$ javac ConnectedComponents.java
$ java ConnectedComponents

Enter the number of vertices:
6
Enter the number of edges:
7
Enter the edges: <from> <to>
1 2
2 3
2 4
4 5
5 6
6 3
6 4
1:  2
2:  4 3 1
3:  6 2
4:  6 5 2
5:  6 4
6:  4 3 5
Component 1: 1 2 4 3 6 5
```

```
Enter the number of vertices:
6
Enter the number of edges:
7
Enter the edges: <from> <to>
1 2
1 4
1 3
2 3
5 6
6 5
4 3
1:  3 4 2
2:  3 1
3:  4 2 1
4:  3 1
5:  6 6
6:  5 5
Component 1: 1 3 4 2
Component 2: 5 6
```

Any Questions

GitHub https://github.com/MustafaDaraghmeh/COEN-352/tree/master/src/coen352/tut11