Data Structures And Algorithms

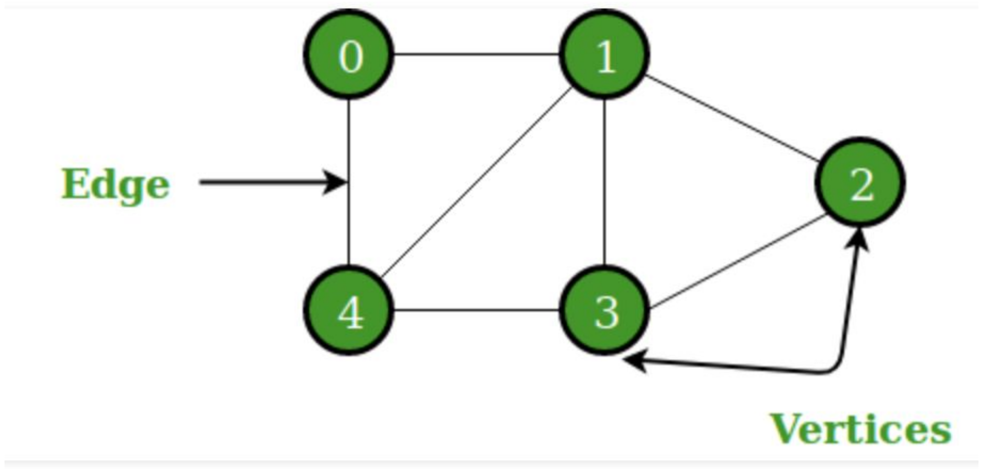
Lecture 10

Introduction to Graph Theory

- Graph theory, a major branch of mathematics, has been studied intensively for hundreds of years. Many important and useful properties of graphs have been discovered, many important algorithms have been developed, and many difficult problems are still actively being studied.
- In this lecture, we will introduce a variety of fundamental graph algorithms that are important in diverse applications.

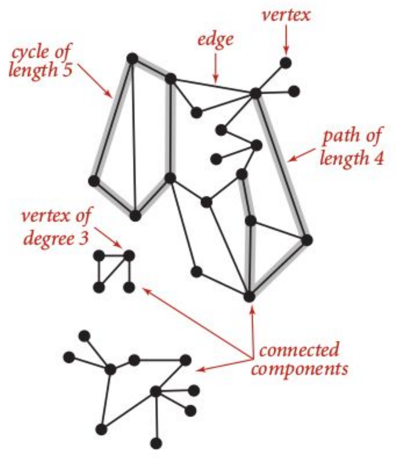
Graph Fundamentals

- A graph is a set of vertices and a collection of edges that each connect a pair of vertices.



Anatomy of a Graph

- A path in a graph is a sequence of vertices connected by edges. A simple path is one with no repeated vertices.
- A cycle is a path with at least one edge whose first and last vertices are the same. A simple cycle is a cycle with no repeated edges or vertices.
- The length of a path or a cycle is its number of edges.



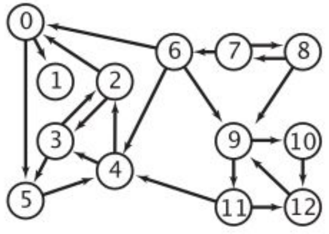
Anatomy of a graph

Typical Graph Application

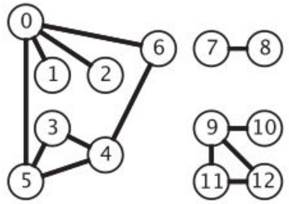
- Maps
  - A person who is planning a trip may need to answer questions such as “What is the shortest route from Providence to Princeton?” A seasoned traveler who has experienced traffic delays andon the shortest route may ask the question “What is the fastest way to get from Providence to Princeton?” To answer such questions, we process information about connections (roads) between items (intersections).
- Web content
  - When we browse the web, we encounter pages that contain references (links) to other pages we move from page to page by clicking on the links. The entire web is a graph, where the items are pages and the connections are links. Graph-processing algorithms are essential components of the search engines that help us locate information on the web.

Undirected and Directed graph

- Graphs can be either
  - directed



  - Undirected

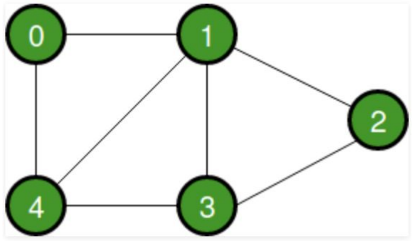


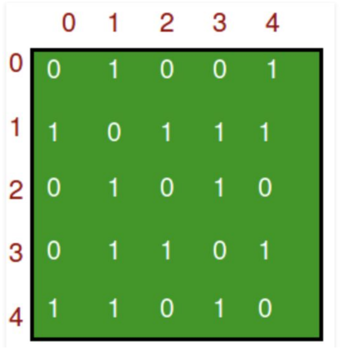
- While directed edges are like a one-way street, undirected edges are like a two-way street.

Edge Weight

- An edge-weighted graph is a graph model where we associate weights or costs with each edge.
- Such graphs are natural models for many applications. In an airline map where edges represent flight routes, these weights might represent distances or fares.
- In an electric circuit where edges represent wires, the weights might represent the length of the wire, its cost, or the time that it takes a signal to propagate through it.
- Minimizing cost is naturally of interest in such situations.

Representation of Graph

- Graph is a data structure that consists of following two components:
  - A finite set of vertices also called as nodes.
  - A finite set of ordered pair of the form (u, v) called as edge. The pair is ordered because (u, v) is not same as (v, u) in case of a directed graph(di-graph). The pair of the form (u, v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight/value/cost. Following two are the most commonly used representations of a graph.
- Following two are the most commonly used representations of a graph.
  - Adjacency Matrix
  - Adjacency list
- Adjacency Matrix
  - 

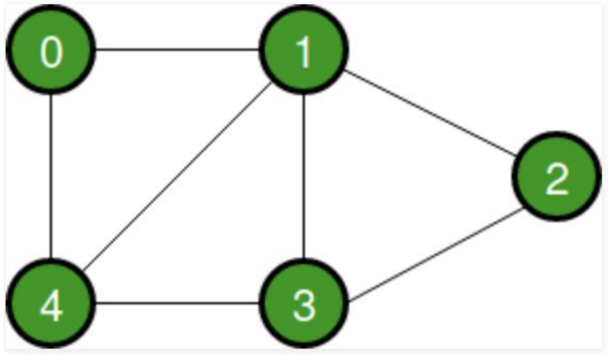


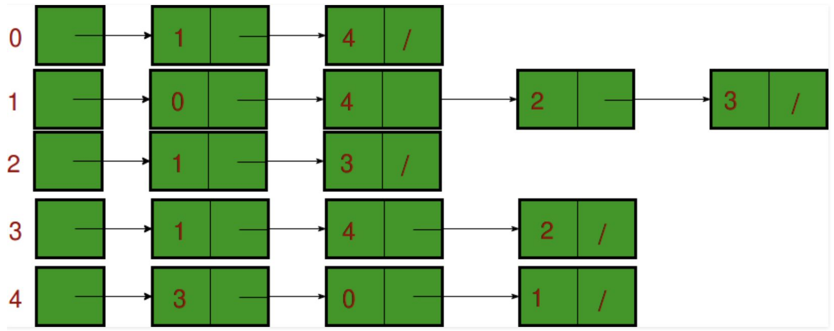
	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

Adjacency Matrix

- Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j. Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.
- Pros:
  - Representation is easier to implement and follow. Removing an edge takes O(1) time. Queries like whether there is an edge from vertex ‘u’ to vertex ‘v’ are efficient and can be done O(1).
- Cons:
  - Consumes more space O(V^2). Even if the graph is sparse(contains less number of edges), it consumes the same space. Adding a vertex is O(V^2) time.

Adjacency List





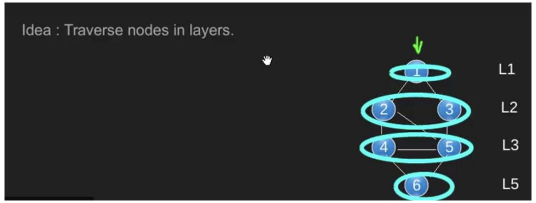
- An array of lists is used. Size of the array is equal to the number of vertices. Let the array be array[]. An entry array[i] represents the list of vertices adjacent to the *i*th vertex. This representation can also be used to represent a weighted graph.

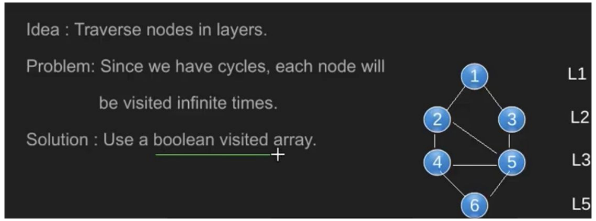
Representation of Graph using adjacency list

- https://repl.it/@RahulKumar40/graph

Graph Traversal

- The two most common ways to search a graph are
  - depth-first search and
  - breadth-first search.
- BFS:


Idea : Traverse nodes in layers.

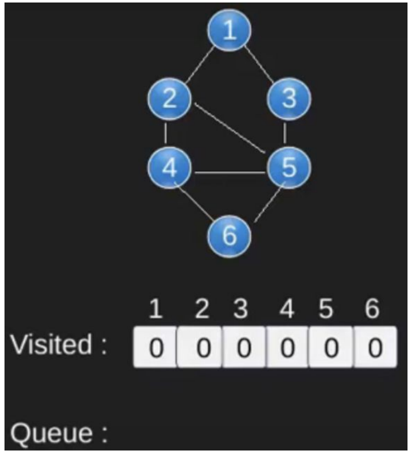

Idea : Traverse nodes in layers.

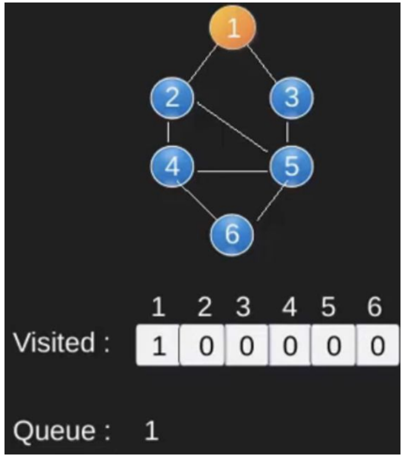
Problem: Since we have cycles, each node will be visited infinite times.

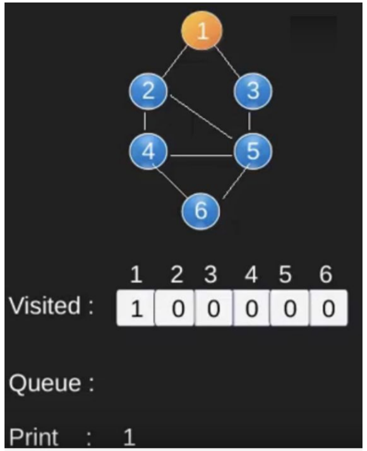
Solution : Use a boolean visited array.

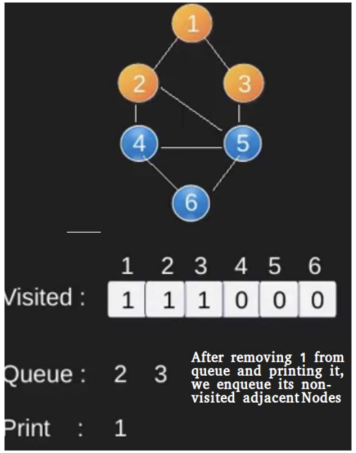
- In breadth-first search (BFS), we start at the root (or another arbitrarily selected node) and explore each neighbor before going on to any of their children. That is, we go wide (hence breadth-first search) before we go deep.

BFS Traversal



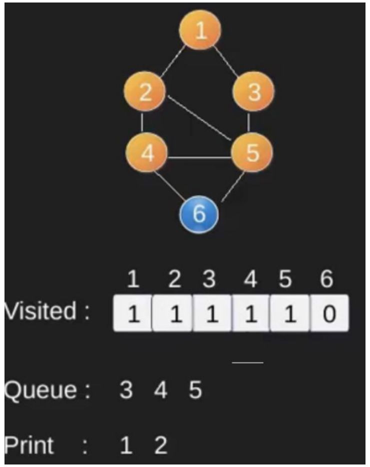






After removing 1 from queue and printing it, we enqueue its non-visited adjacent Nodes

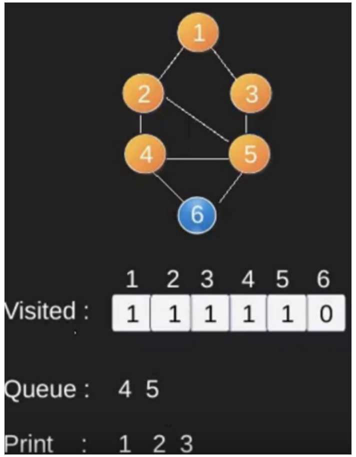
BFS Traversal



	1	2	3	4	5	6
Visited :	1	1	1	1	1	0

Queue : 3 4 5

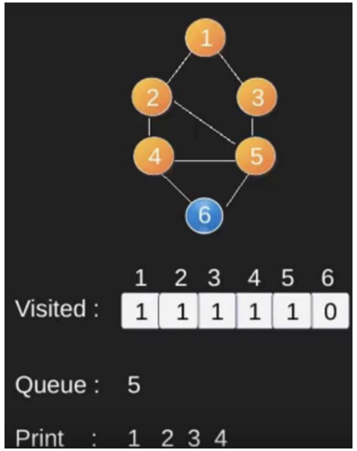
Print : 1 2



	1	2	3	4	5	6
Visited :	1	1	1	1	1	0

Queue : 4 5

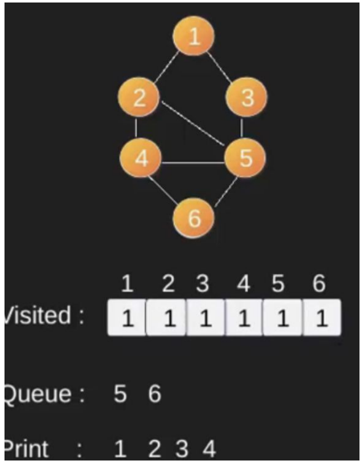
Print : 1 2 3



	1	2	3	4	5	6
Visited :	1	1	1	1	1	0

Queue : 5

Print : 1 2 3 4

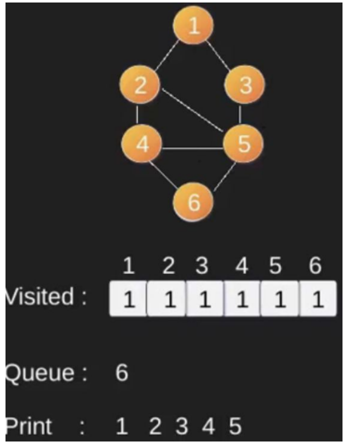


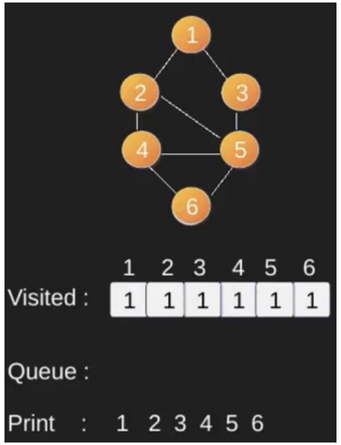
	1	2	3	4	5	6
Visited :	1	1	1	1	1	1

Queue : 5 6

Print : 1 2 3 4

BFS Traversal





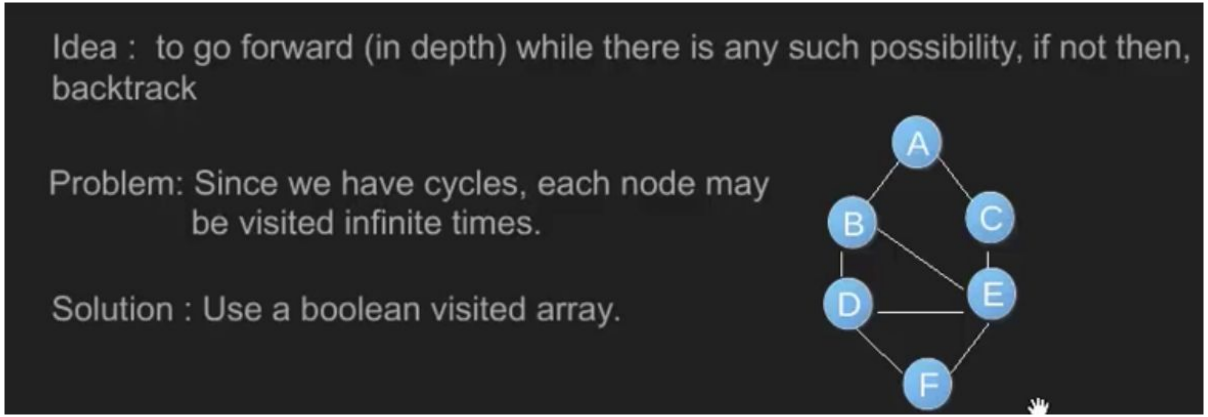
BFS Traversal

- Pseudocode

```
BFS(Node root)
    Initialize empty array visited(having length of number of nodes in graph) to false
    Create an empty queue q
    Mark visited[root] = true
    Push root to q
    Add root to queue q
    Loop while queue is not empty
        Dequeue a node from q and assign it's value to temp_node
        Print temp_node value
        Get the adjacent list of temp_node.
        Loop around the adjacent list
            If node is not marked as false in visited[node]
                Mark visited[node] = true
                Add node to queue q
```

DFS Traversal

- In depth-first search (DFS), we start at the root (or another arbitrarily selected node) and explore each branch completely before moving on to the next branch. That is, we go deep first (hence the name depth-first search) before we go wide.

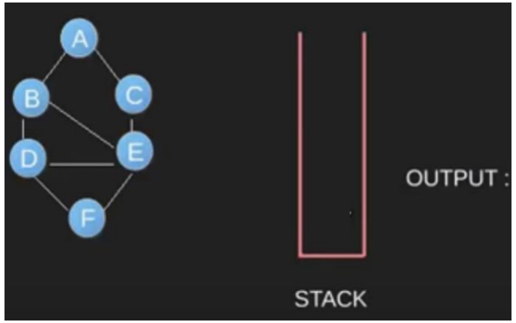


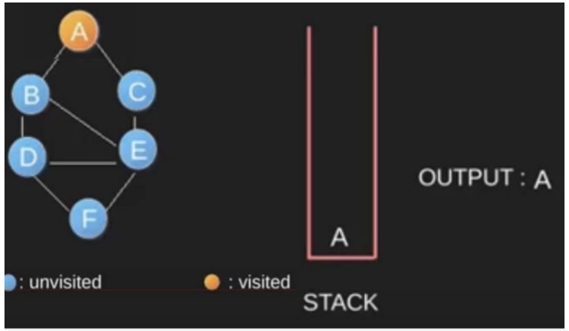
Idea : to go forward (in depth) while there is any such possibility, if not then, backtrack

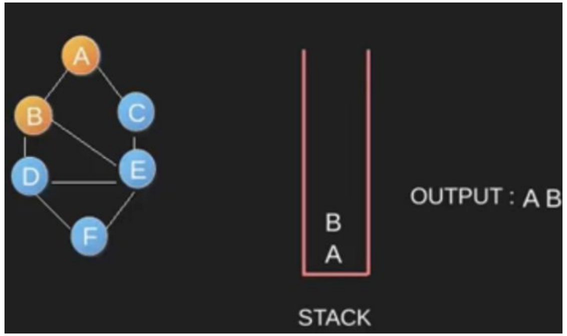
Problem: Since we have cycles, each node may be visited infinite times.

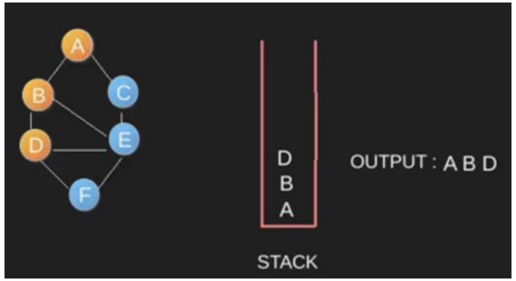
Solution : Use a boolean visited array.

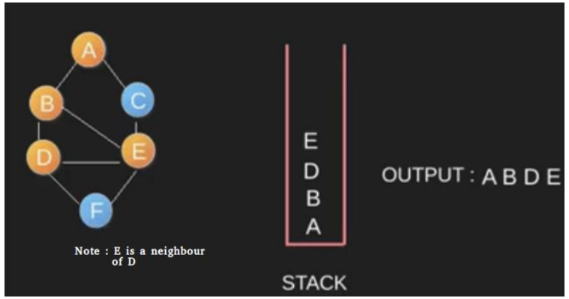
DFS Traversal

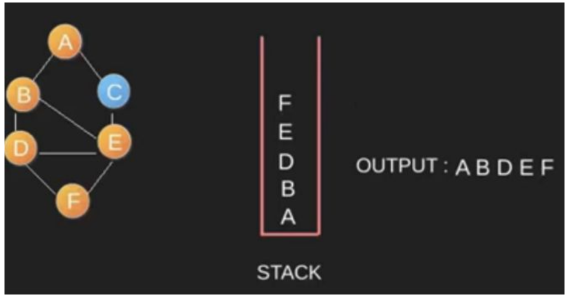




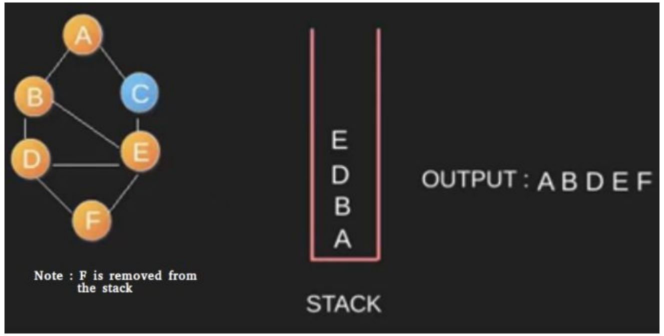


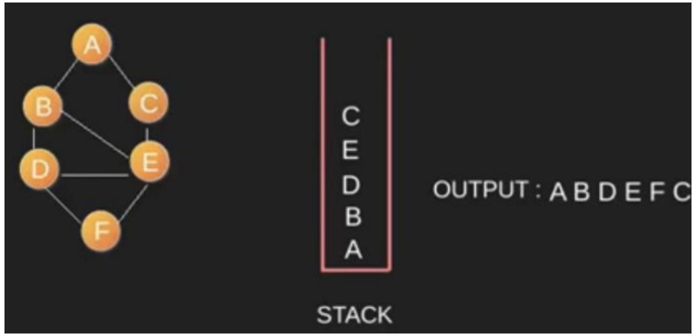


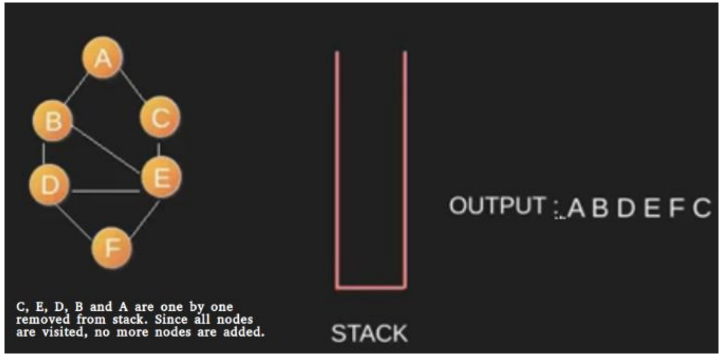




DFS Traversal






A
B
C
D
E
F
OUTPUT : A B D E F C
C, E, D, B and A are one by one removed from stack. Since all nodes are visited, no more nodes are added.
STACK

DFS Traversal

- Pseudocode
  - DFS(Node root)
    - Initialize empty array visited(having length of number of nodes in graph) to false
    - Call DFSUtil(root, visited)
  - DFSUtil(Node node)
    - Mark visited[node] = true
    - Print node value
    - Get the adjacent list of node
    - Loop around the adjacent list
      - If node in adjacent list is marked as false in visited[node]
      - Call DFSUtil(node)
- https://repl.it/@RahulKumar40/graph

Application of DFS And BFS

- Breadth-first search and depth-first search tend to be used in different scenarios.
- DFS is often preferred if we want to visit every node in the graph. Both will work just fine, but depth-first search is a bit simpler. However, if we want to find the shortest path (or just any path) between two nodes, BFS is generally better.

SCQ/MCQ

- Is A tree is actually a type of graph, but not all graphs are trees?
    - Yes
    - No
        - Ans - Yes
- Is A tree a connected graph without cycles?
    - Yes
    - No
        - Ans - Yes