Lecture 17: Graph Introduction

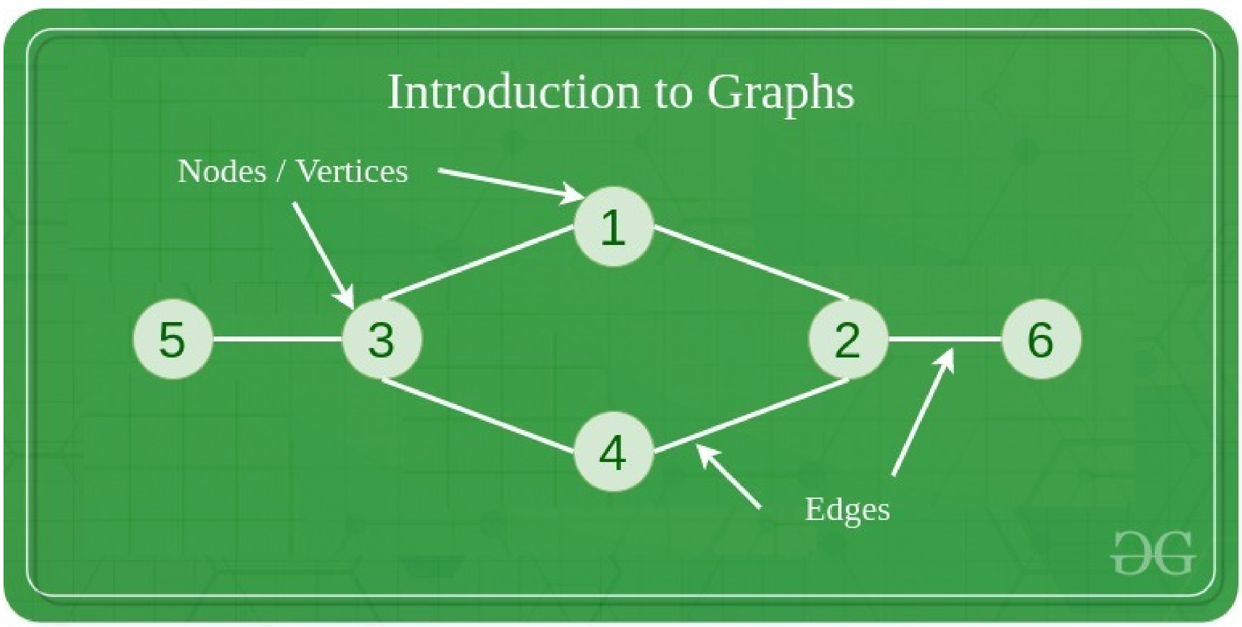
Date: 10/17/2023

Introduction

A **Graph** is a non-linear data structure consisting of vertices and edges.

- vertices: sometimes also referred to as nodes
- edges: lines or arcs that connect any two nodes in the graph.

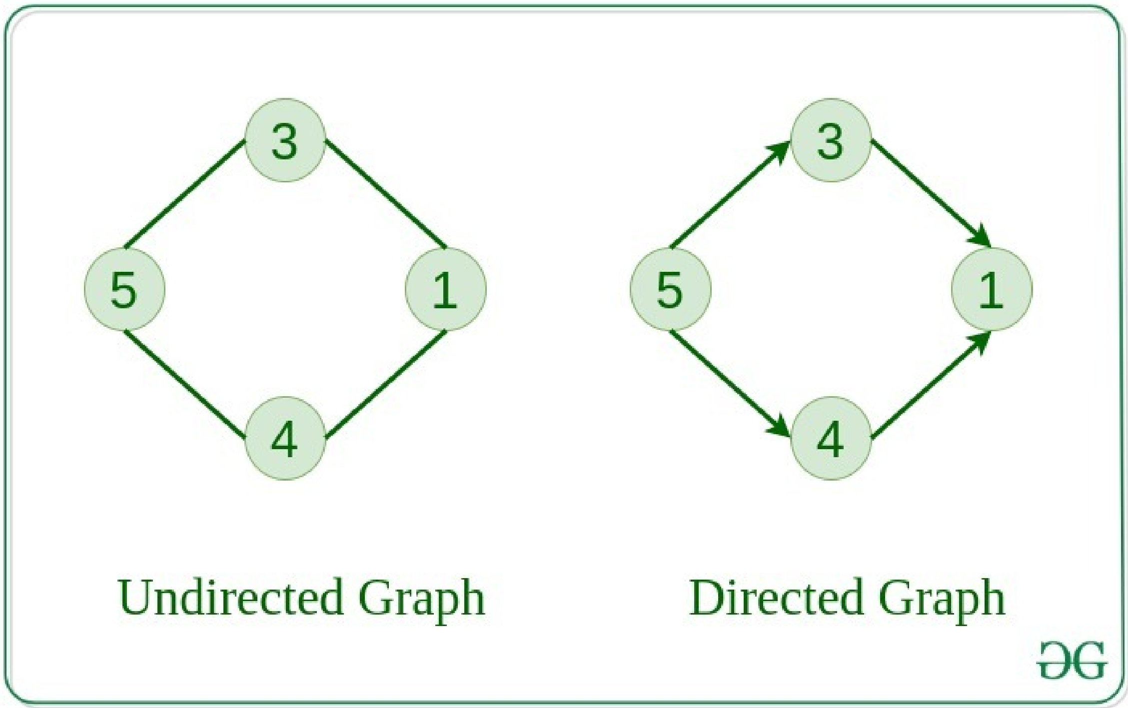
More formally a Graph is composed of a set of vertices V and a set of edges E. The graph is denoted by **G(V, E)**.

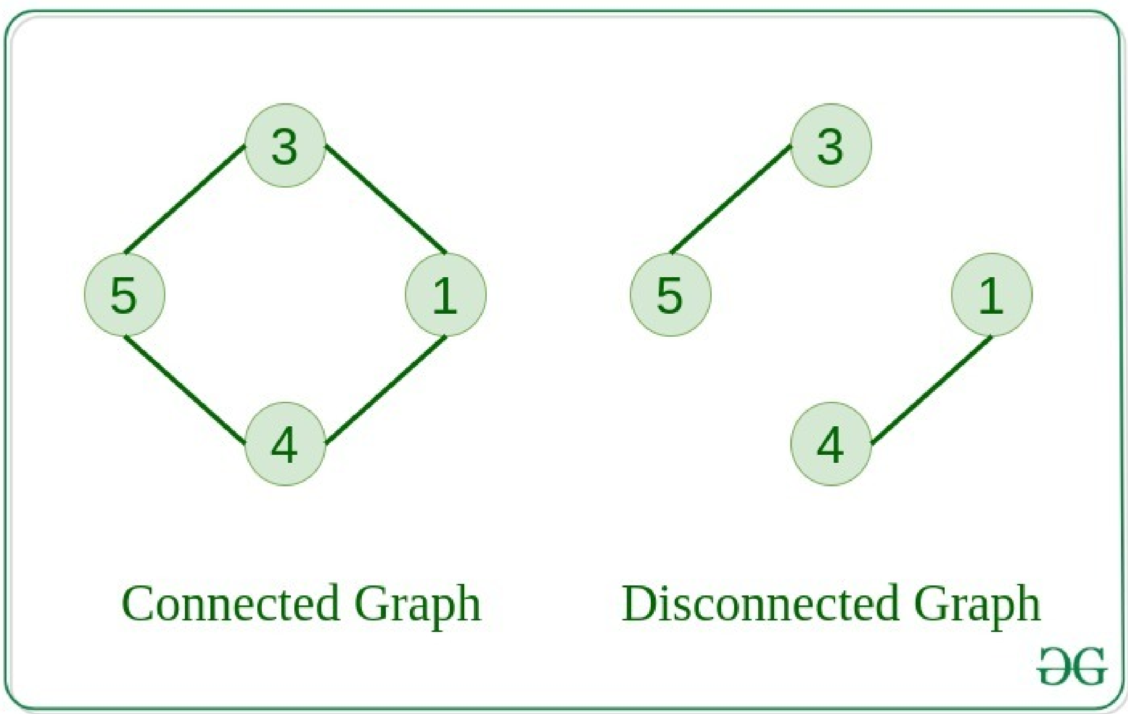


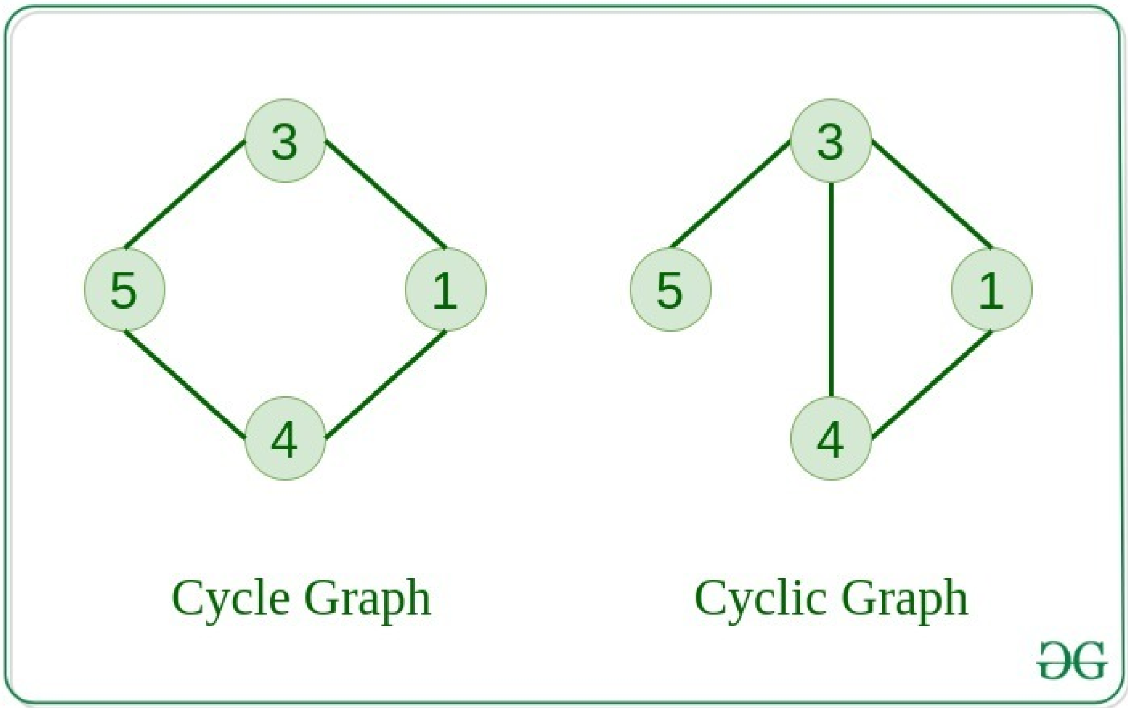
reference: https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/

For this graph,
V = {1, 2, 3, 4, 5, 6}
E = {(1,2), (1,3), (2,4), (2,6), (3,5), (3,4)}

Types


3
5
1
4
Undirected Graph
3
5
1
4
Directed Graph


3
5
1
4
Connected Graph
3
5
1
4
Disconnected Graph



reference:
https://www.geeksforgeeks.org/introduction-to-graphs-data-structure-and-algorithm-tutorials/

Representation

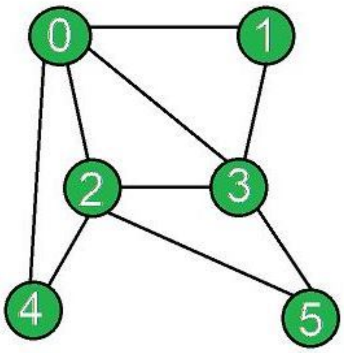
There are two most common ways to represent a graph.

1. Adjacency Matrix
2. Adjacency List

Adjacency Matrix

An adjacency matrix is a way of representing a graph as a matrix of booleans.
adjMat[n][n] having dimension n x n.

- If there is an edge from vertex i to j, mark adjMat[i][j] as 1.
- If there is no edge from vertex i to j, mark adjMat[i][j] as 0.



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

reference:www.geeksforgeeks.org/convert-adjacency-matrix-to-adjacency-list-representation-of-graph/

1_graph_adj_mat.cpp

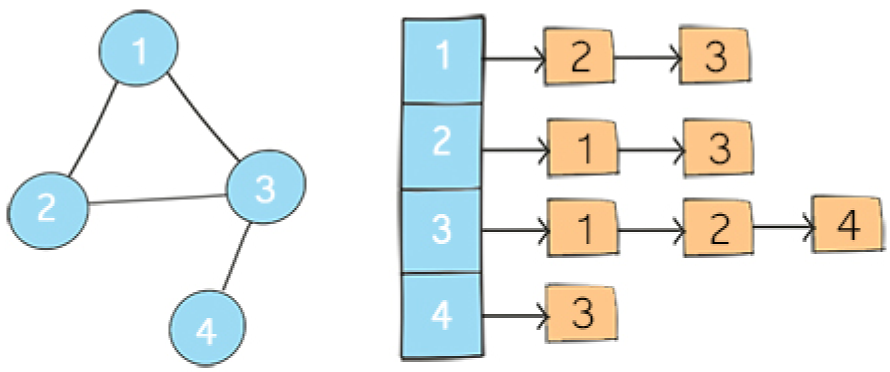
```
// vector version (node values are 0 to V)
class Graph{
public:
      vector<vector<bool>> adjMat;
      Graph(int V){
          adjMat.resize(V, vector<bool>(V));
      }

      void addEdge(int s, int t) {
          adjMat[s][t] = true;
      }

      bool isConnected(int s, int t){
          return adjMat[s][t];
      }
};

// map version (supports any data type)
class Graph{
public:
    map<int, map<int, bool> > adjMat;

    void addEdge(int s, int t) {
       adjMat[s][t] = true;
    }
```

```
    bool isConnected(int s, int t){
      adjMat[s][t];
    }
};
```

Adjacency List

An array of Lists is used to store edges between two vertices.
array of list of size n as adjList[n].

- adjList[0] will have all nodes which are connected to vertex 0.
- adjList[1] will have all nodes which are connected to vertex 1 and so on.



reference: https://www.lavivienpost.com/implement-graph-as-adjacency-list/

2_graph_adj_list.cpp

```
class Node{
public:
      int data;
      vector<Node*> connected;
};

class Graph{
public:
      // supports any node values
      map<int, Node*> nodes;
      void addEdge(int s, int t) {
           Node* nodeS = nodes[s];
           Node* nodeT = nodes[t];
```

```
        nodeS->connected.push_back(nodeT);
        nodeT->connected.push_back(nodeS);
    }

    bool isConnected(int s, int t){
        Node* nodeS = nodes[s];
        Node* nodeT = nodes[t];

        for(Node* connectedToS: nodeS->connected){
            if(connectedToS == nodeT){
                return true;
            }
        }

        return false;
    }
};
```

Comparison

	Adjacency Matrix	Adjacency List
Space Required	$O(V^2)$	$O(V + E)$
Adding Vertex	$O(V^2)$	$O(1)$
Removing Vertex	$O(V^2)$	$O(V + E)$
Adding Edge	$O(1)$	$O(1)$
Removing Edge	$O(1)$	$O(E)$
Checking if Edge Exist	$O(1)$	$O(V)$

Operations

- Graph Traversal
- Find Cycles

- Shortest Path
- Minimum Spanning Tree
- Topological Sorting
- Connectivity
- Maximum Flow

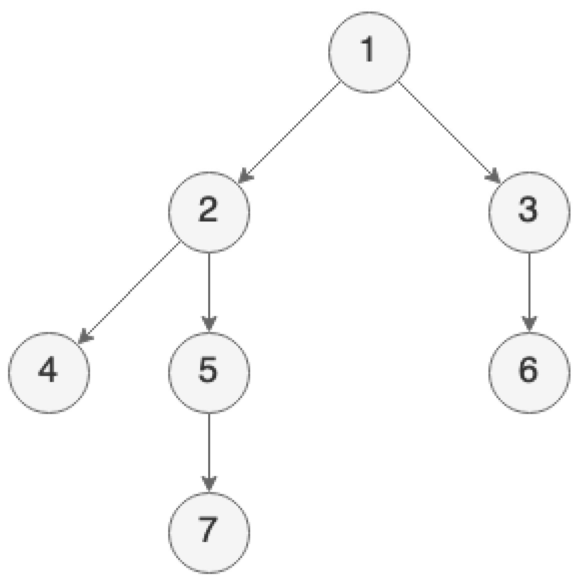
Depth First Traversal

Let's try to use the same algorithm we used for Trees.

3_tree_dfs.cpp

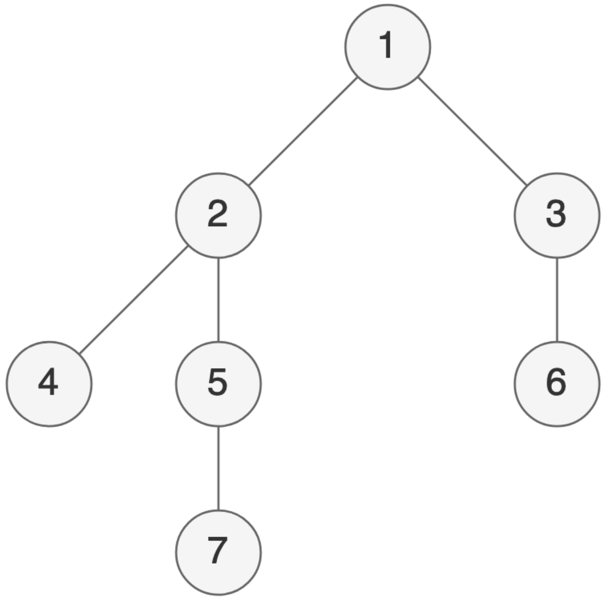
```
void dfs(Node* node){
    if(node == nullptr){
        return;
    }
    cout << node->data << "\n";
    for(Node* child: node->connected){
        dfs(child);
    }
}
```

Example:



That works just fine.

What about undirected graph?



We now have an infinite running algorithm.

We need to store all visited nodes, and everytime before we traverse a node, check if it has already been visited.

4_graph_dfs.cpp

```
map<Node*, bool> visited;
void dfs(Node* node){
      if(node == nullptr)
      return;

      if(visited[node])
      return;

      cout << node->data << "\n";
      visited[node] = true;
      for(Node* node: node->connected){
          dfs(node);
      }
}
```

⇒ Note: In Trees, we always start with a root node. But for graphs, we don't have any root nodes. So, we need to specify from which node we want to start the algorithm from.

Breadth First Traversal

5_graph_bfs.cpp

```
map<Node*, bool> visited;
void bfs(Node* source){
      // current level
      vector<Node*> level;
      level.push_back(source);

      // iterate until we have exhausted all nodes
      while(level.size()!=0){
      // to store next level nodes that will be
      // traversed in next iteration
      vector<Node*> newLevel;

      // for each node in current level, add all its
      // non-null and non-visited child to nextLevel
      for(Node* node: level){
            cout << node -> data << " ";

            for(Node* connected: node->connected)
            if(connected != nullptr && !visited[connected])
            newLevel.push_back(connected);
      }

      // building next level
      level.clear();
      for(Node* node: newLevel)
          level.push_back(node);
      }
}
```