Data Structures & Algorithms

Prepared by: Mohamed Ayman

Algorithm Engineer at Valeo

Deep Learning Researcher and Teaching Assistant

at The American University in Cairo (AUC)

spring 2020

sw.eng.MohamedAyman@gmail.com

facebook.com/cs.MohamedAyman

linkedin.com/in/cs-MohamedAyman

github.com/cs-MohamedAyman

codeforces.com/profile/Mohamed_Ayman

Lecture 13

Graph

Course Roadmap

Part 2: Non-Linear Data Structures

Lecture 8: Binary Tree

Lecture 9: Binary Search Tree

Lecture 10: Self Balancing Binary Search Tree

Lecture 11: Binary Heap Tree

Lecture 12: Hash Table

Lecture 13: Graph

Lecture 14: STL in C++ (Non-Linear Data Structures)

Lecture Agenda

We will discuss in this lecture the following topics

1- Introduction to Graphs

2- Directed vs. Undirected Graph

3- Breadth First Traverse

4- Depth First Traverse

5- Cyclic vs. Acyclic Graph

6- Connected vs. Disconnected Graph

7- Time Complexity & Space Complexity


Let's
START UP

Lecture Agenda

Section 1: Introduction to Graphs

Section 2: Directed vs. Undirected Graph

Section 3: Breadth First Traverse

Section 4: Depth First Traverse

Section 5: Cyclic vs. Acyclic Graph

Section 6: Connected vs. Disconnected Graph

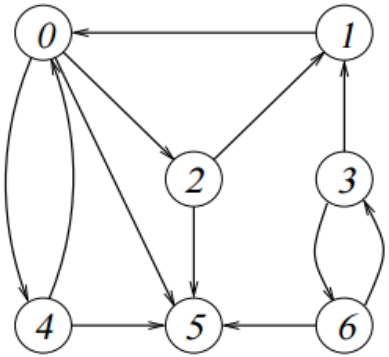
Section 7: Time Complexity & Space Complexity

Introduction to Graphs

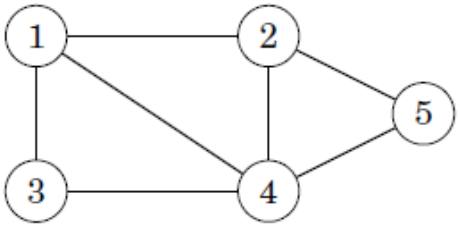
- Graph is a data structure that consists of following two components:
  1. A finite set of vertices also called as nodes.
  2. A finite set of ordered pair of the form (u, v) called as edge.
- The pair is ordered because (u, v) is not same as (v, u) in case of directed graph (di-graph). The pair of form (u, v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight/value/cost.
- Many programming problems can be solved by modeling the problem as a graph problem and using an appropriate graph algorithm. A typical example of a graph is a network of roads and cities in a country. Sometimes, though, the graph is hidden in the problem and it may be difficult to detect it.
- Formally, a graph is an ordered pair, G = (V, E), of two sets representing the nodes or vertices of the graph and the edges of the graph. An edge specifies which nodes have a connection between them.
- Graphs are used to represent many real life applications: it used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like LinkedIn, Facebook. For example, in Facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender and locale.

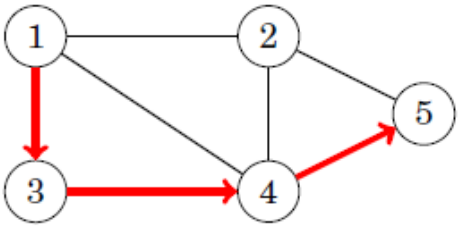
Introduction to Graphs

- G = (V, E) with
- Vertex Set V = {0, 1, 2, 3, 4, 5, 6}
- Edge Set E = {(0, 2), (0, 4), (0, 5), (1, 0), (2, 1), (2, 5), (3, 1), (3, 6), (4, 0), (4, 5), (6, 3), (6, 5)}



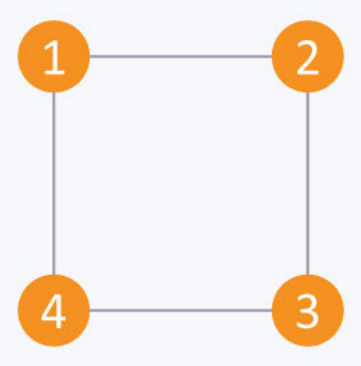
- A graph consists of nodes and edges. the variable n denotes the number of nodes in a graph, and the variable m denotes the number of edges. The nodes are numbered using integers 1,2,...,n. For example, the following graph consists of 5 nodes and 7 edges:
- A path leads from node a to node b through edges of the graph. The length of a path is the number of edges in it. For example, the above graph contains a path 1 -> 3 -> 4 -> 5 of length 3 from node 1 to node 5:
- A path is a cycle if the first and last node is the same. this graph contains a cycle 1 -> 3 -> 4 -> 1. A path is simple if each node appears at most once in the path.

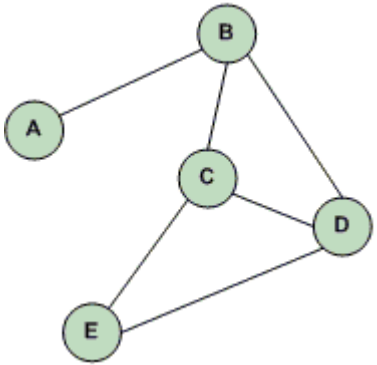




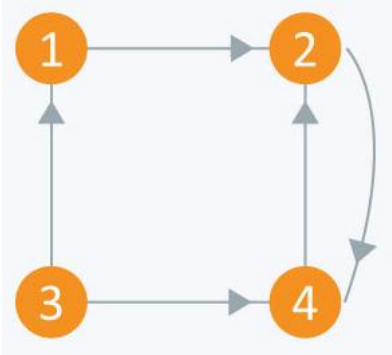
Types of Graph

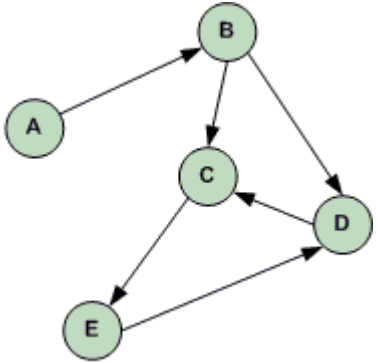
- **Undirected Graph:** A graph in which all the edges are undirected is called as a non-directed graph. In other words, edges of an undirected graph do not contain any direction.





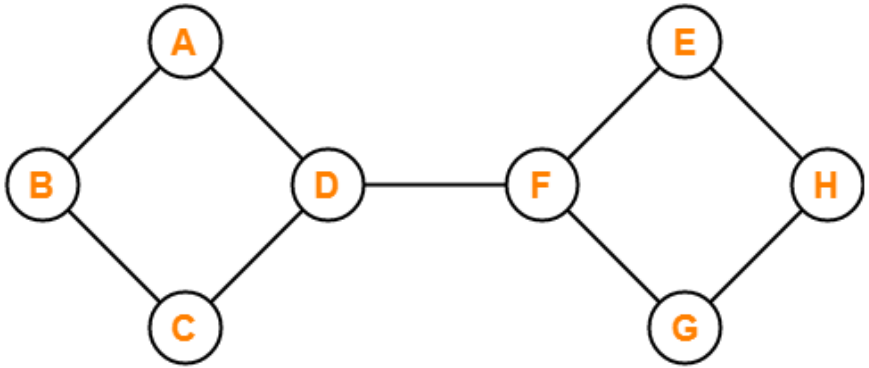
- **Directed Graph:** A graph in which all the edges are directed is called as a directed graph. In other words, all the edges of a directed graph contain some direction. Directed graphs are also called as digraphs.



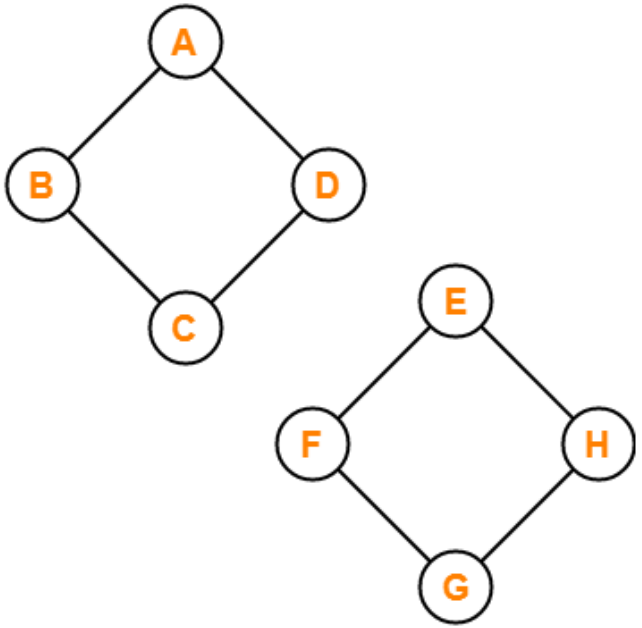


Types of Graph

- **Connected Graph:** A graph in which we can visit from any one vertex to any other vertex is called as a connected graph. In connected graph, at least one path exists between every pair of vertices.

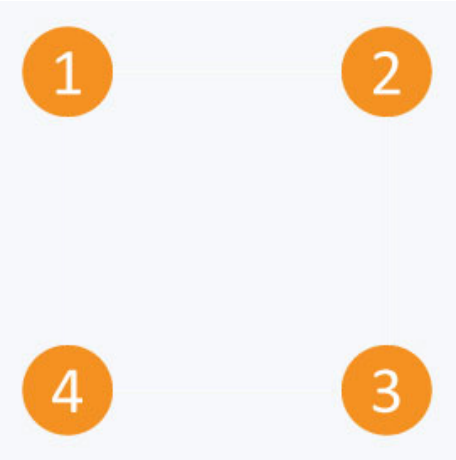


- **Disconnected Graph:** A graph in which there does not exist any path between at least one pair of vertices is called as a disconnected graph.

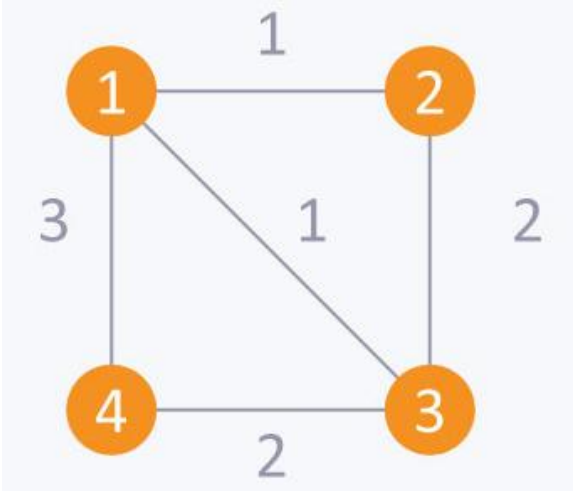


Types of Graph

- **Null Graph:** A graph whose edge set is empty is called as a null graph. In other words, a null graph does not contain any edges in it.

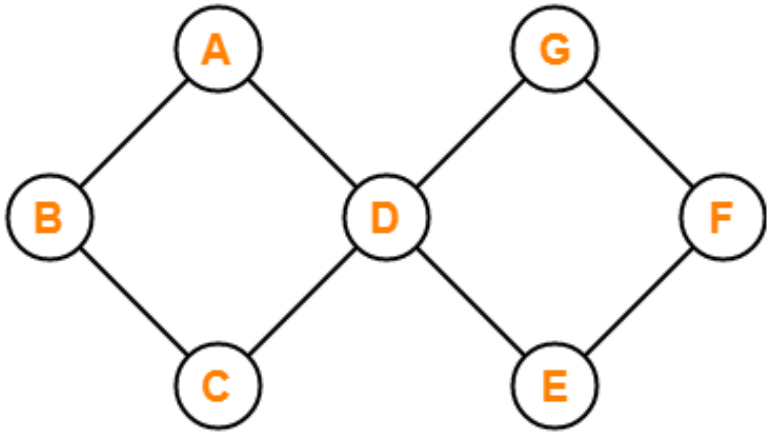


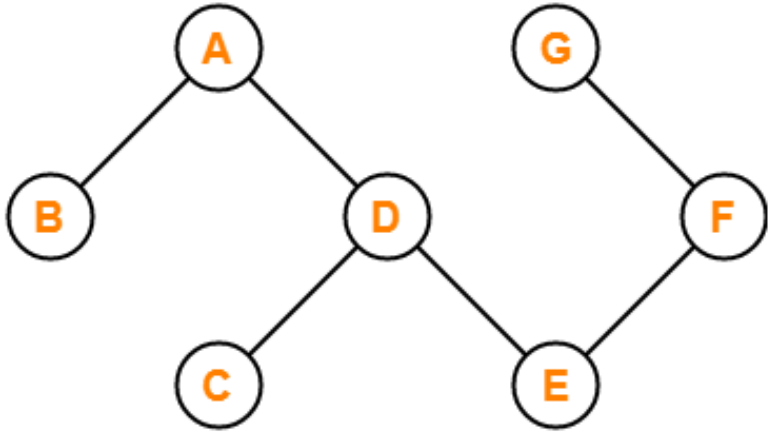
- **Weighted Graph:** A graph that each edge is assigned a weight or cost. Consider a graph of 4 nodes as in the diagram below. As you can see each edge has a weight/cost assigned to it.



Types of Graph

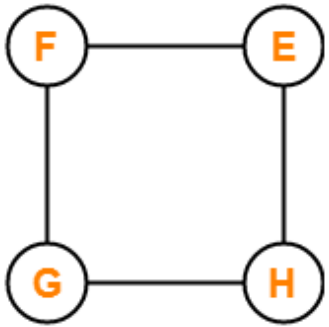
- Cyclic Graph: A graph containing at least one cycle in it is called as a cyclic graph.
- Acyclic Graph: A graph not containing any cycle in it is called as an acyclic graph.

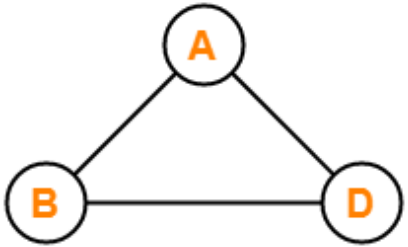




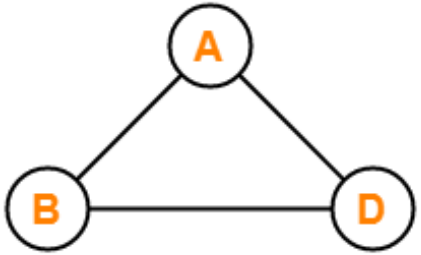
Types of Graph

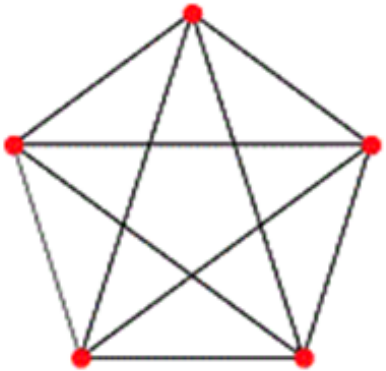
- **Regular Graph:** A graph in which degree of all the vertices is same is called as a regular graph. If all the vertices in a graph are of degree k, then it is called as a "k-regular graph".

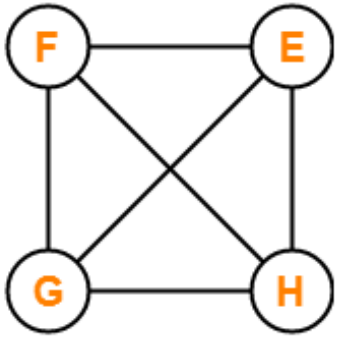




- **Complete Graph:** A graph in which exactly one edge is present between every pair of vertices is called as a complete graph.

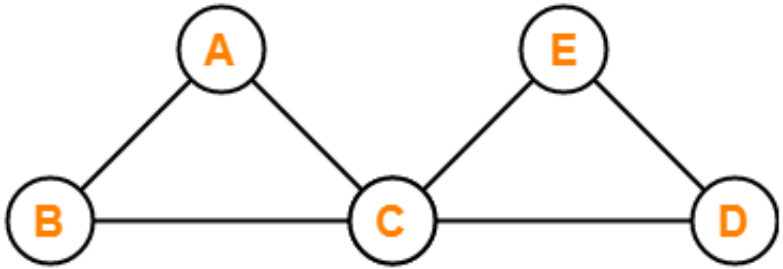




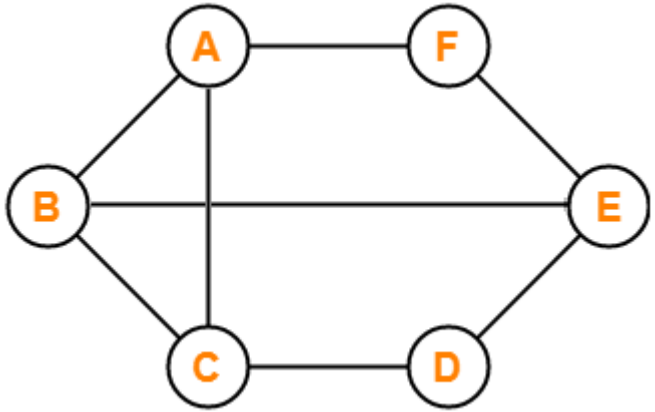


Types of Graph

- **Euler Graph:** Euler Graph is a connected graph in which all the vertices are even degree.



- **Hamiltonian Graph:** If there exists a closed walk in the connected graph that visits every vertex of the graph exactly once (except starting vertex) without repeating the edges, then such a graph is called as a Hamiltonian graph.



Lecture Agenda

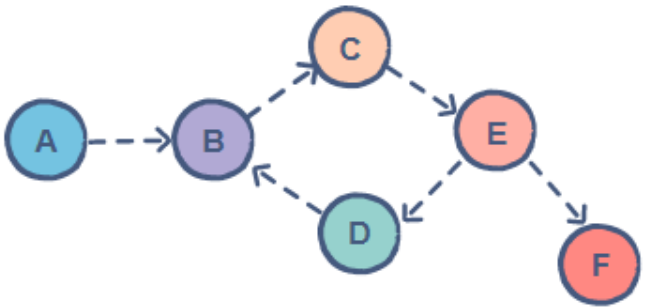
- ✓ Section 1: Introduction to Graphs
- **Section 2: Directed vs. Undirected Graph**
- Section 3: Breadth First Traverse
- Section 4: Depth First Traverse
- Section 5: Cyclic vs. Acyclic Graph
- Section 6: Connected vs. Disconnected Graph
- Section 7: Time Complexity & Space Complexity

Directed vs. Undirected Graph

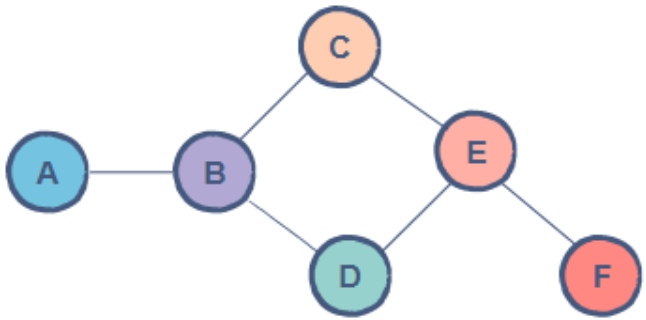
- Definition: A directed graph is a type of graph that contains ordered pairs of vertices while an undirected graph is a type of graph that contains unordered pairs of vertices. Thus, this is the main difference between directed and undirected graph.
- Direction: Furthermore, in directed graphs, the edges represent the direction of vertexes. However, in undirected graphs, the edges do not represent the direction of vertexes. Hence, this is another difference between directed and undirected graph.
- Representation: Moreover, the symbol of representation is a major difference between directed and undirected graph. In directed graphs, arrows represent the edges, while in undirected graphs, undirected arcs represent the edges.
- Conclusion: There are two types of graphs as directed and undirected graphs. The main difference between directed and undirected graph is that a directed graph contains an ordered pair of vertices whereas an undirected graph contains an unordered pair of vertices.

Directed vs. Undirected Graph

- **Directed Graphs:** A directed graph is a set of vertices (nodes) connected by edges, with each node having a direction associated with it. Edges are usually represented by arrows pointing in the direction the graph can be traversed.
- **Undirected Graphs:** In an undirected graph the edges are bidirectional, with no direction associated with them. Hence, the graph can be traversed in either direction. The absence of an arrow tells us that the graph is undirected.



Directed Graph



Undirected Graph

Adjacency Matrix vs. Adjacency List

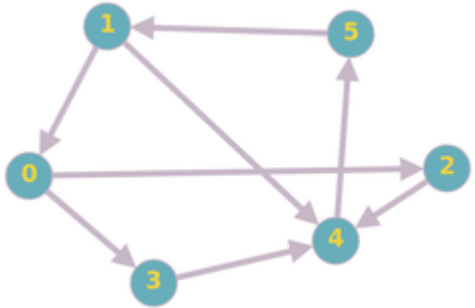
- Adjacency matrices: For a graph with |V| vertices, an adjacency matrix is a |V| × |V| matrix of 0s and 1s, where the entry in row i and column j is 1 if and only if the edge (i, j) is in the graph. If you want to indicate an edge weight, put it in the row i, column j entry, and reserve a special value (perhaps null) to indicate an absent edge. Here's the adjacency matrix for the social network graph:
- With an adjacency matrix, we can find out whether an edge is present in constant time, by just looking up the corresponding entry in the matrix. For example, if the adjacency matrix is named graph, then we can query whether edge (i, j) is in the graph by looking at graph[i][j]. So what's the disadvantage of an adjacency matrix?
- Two things, actually. First, it takes Θ(V^2) space, even if the graph is sparse: relatively few edges. In other words, for a sparse graph, the adjacency matrix is mostly 0s, and we use lots of space to represent only a few edges. Second, if you want to find out which vertices are adjacent to a given vertex i, you have to look at all |V| entries in row i, even if only a small number of vertices are adjacent to vertex i.
- For an undirected graph, the adjacency matrix is symmetric: the row i, column j entry is 1 if and only if the row j, column i entry is 1. For a directed graph, the adjacency matrix need not be symmetric.

Adjacency Matrix vs. Adjacency List

- Adjacency lists: Representing a graph with adjacency lists combines adjacency matrices with edge lists. For each vertex i, store an array of the vertices adjacent to it. We typically have an array of |V| adjacency lists, one adjacency list per vertex. Here's an adjacency-list representation of the social network graph:
- Vertex numbers in an adjacency list are not required to appear in any particular order, though it is often convenient to list them in increasing order, as in this example. We can get to each vertex's adjacency list in constant time, because we just have to index into an array.
- To find out whether an edge (i,j) is present in the graph, we go to i's adjacency list in constant time and then look for j in i's adjacency list. How long does that take in the worst case? The answer is $\Theta(d)$ where d is the degree of vertex i, because that's how long i's adjacency list is. The degree of vertex i could be as high as |V|-1 (if i is adjacent to all the other |V|-1 vertices) or as low as 0 (if i is isolated, with no incident edges).
- In an undirected graph, vertex j is in vertex i's adjacency list if and only if i is in j's adjacency list. If the graph is weighted, then each item in each adjacency list is either a two-item array or an object, giving the vertex number and the edge weight.

Adjacency Matrix vs. Adjacency List

- Directed Graph



- Adjacency Matrix

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

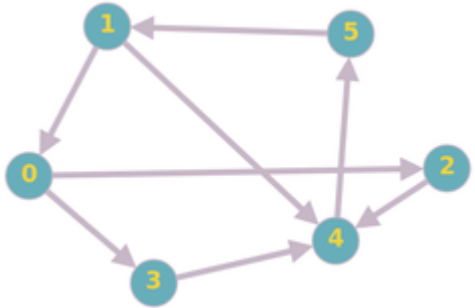
- Undirected Graph

- Adjacency Matrix

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

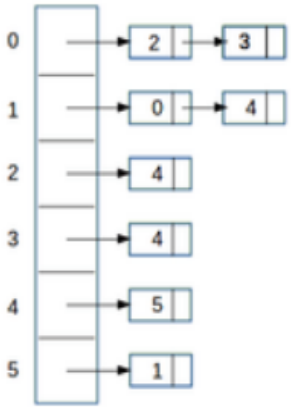
Adjacency Matrix vs. Adjacency List

- Directed Graph

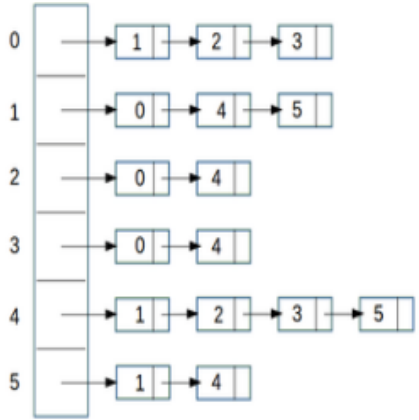


- Undirected Graph

- Adjacency List



- Adjacency List



Adjacency Matrix vs. Adjacency List - Edges

- Adjacency Matrix

```
// Initialize an adjacency Matrix with static length
const int N = 1e3+3;
int n;
bool adj[N][N];

// This function adds edges between given vertices (undirected edge)
void add_undirected_edge(int u, int v) {
    adj[u][v] = true;
    adj[v][u] = true;
}
// This function adds edges between given vertices (directed edge)
void add_directed_edge(int u, int v) {
    adj[u][v] = true;
}
```

Graphs-Adjacency-Matrix.cpp

Adjacency Matrix vs. Adjacency List - Edges

- Adjacency List

```
// Initialize an adjacency List with static length
const int N = 1e5+3;
int n;
vector<int> adj[N];

// This function adds edges between given vertices (undirected edge)
void add_undirected_edge(int u, int v) {
    adj[u].push_back(v);
    adj[v].push_back(u);
}
// This function adds edges between given vertices (directed edge)
void add_directed_edge(int u, int v) {
    adj[u].push_back(v);
}
```

Graphs-Adjacency-List.cpp

Lecture Agenda

- ✔ Section 1: Introduction to Graphs
- ✔ Section 2: Directed vs. Undirected Graph
- **Section 3: Breadth First Traverse**
- Section 4: Depth First Traverse
- Section 5: Cyclic vs. Acyclic Graph
- Section 6: Connected vs. Disconnected Graph
- Section 7: Time Complexity & Space Complexity

Breadth First Traverse

- Graph traversal means visiting every vertex and edge exactly once in a well-defined order. While using certain graph algorithms, you must ensure that each vertex of the graph is visited exactly once. The order in which the vertices are visited are important and may depend upon the algorithm or question that you are solving. During a traversal, it is important that you track which vertices have been visited. The most common way of tracking vertices is to mark them.
- Breadth-first search visits the nodes in increasing order of their distance from the starting node. Thus, we can calculate the distance from the starting node to all other nodes using breadth-first search. However, breadth-first search is more difficult to implement than depth-first search. Breadth-first search goes through the nodes one level after another. First the search explores the nodes whose distance from the starting node is 1, then the nodes whose distance is 2, and so on. This process continues until all nodes have been visited.
- In a breadth-first traversal, we visit the starting node and then on the first pass visit all of the nodes directly connected to it. In the second pass, we visit nodes that are two edges "away" from the starting node. With each new pass, we visit nodes that are one more edge away. Because there might be cycles in the graph, it is possible for a node to be on two paths of different lengths from the starting node. Because we will visit that node for the first time along the shortest path from the starting node, we will not need to consider it again. We will, therefore, either need to keep a list of the nodes we have visited or we will need to use a variable in the node to mark it as visited to prevent multiple visits.

Breadth First Traverse Algorithm

Algorithm bfsFromVertex(G, v)

1. $visited[v] = \text{TRUE}$
2. Q.enqueue(v)
3. **while not** Q.isEmpty$()$ **do**
4. $\quad v \leftarrow Q$.dequeue$()$
5. $\quad$ **for all** w adjacent to v **do**
6. $\quad\quad$ **if** $visited[w] = \text{FALSE}$ **then**
7. $\quad\quad\quad visited[w] = \text{TRUE}$
8. $\quad\quad\quad Q$.enqueue(w)

Breadth First Traverse - Adjacency Matrix

```cpp
// Initialize an array to mark visited vertices
bool vis_bfs[N];
// This function traverses the graph in breadth first search
void bfs(int src) {
    queue<int> q;
    q.push(src);  // push and mark the source node as visited
    vis_bfs[src] = true;
    // loop to print all vertices in the graph
    while (!q.empty()) {
        int u = q.front();
        q.pop();  // pop and print the front node
        cout << u << ' ';
        // loop for all neighbors of the vertex u
        for (int v = 1 ; v <= n ; v++) {
            if(adj[u][v] && vis_bfs[v] == false) {
                q.push(v);  // push and mark the current node as visited
                vis_bfs[v] = true;
            }
        }
    }
}
```

Graphs-Adjacency-Matrix.cpp

Breadth First Traverse - Adjacency List

```cpp
// Initialize an array to mark visited vertices
bool vis_bfs[N];
// This function traverses the graph in breadth first search
void bfs(int src) {
    queue<int> q;
    q.push(src);  // push and mark the source node as visited
    vis_bfs[src] = true;
    // loop to print all vertices in the graph
    while (!q.empty()) {
        int u = q.front();
        q.pop();  // pop and print the front node
        cout << u << ' ';
        // loop for all neighbors of the vertex u
        for (int v : adj[u]) {
            if (vis_bfs[v] == false) {
                q.push(v);  // push and mark the current node as visited
                vis_bfs[v] = true;
            }
        }
    }
}
```

Graphs-Adjacency-List.cpp

Lecture Agenda

- ✔ Section 1: Introduction to Graphs
- ✔ Section 2: Directed vs. Undirected Graph
- ✔ Section 3: Breadth First Traverse
- **Section 4: Depth First Traverse**
- Section 5: Cyclic vs. Acyclic Graph
- Section 6: Connected vs. Disconnected Graph
- Section 7: Time Complexity & Space Complexity

Depth First Traverse

- Graph traversal means visiting every vertex and edge exactly once in a well-defined order. While using certain graph algorithms, you must ensure that each vertex of the graph is visited exactly once. The order in which the vertices are visited are important and may depend upon the algorithm or question that you are solving. During a traversal, it is important that you track which vertices have been visited. The most common way of tracking vertices is to mark them.
- Depth-first search is a straightforward graph traversal technique. The algorithm begins at a starting node, and proceeds to all other nodes that are reachable from the starting node using the edges of the graph. Depth-first search always follows a single path in the graph as long as it finds new nodes. After this, it returns to previous nodes and begins to explore other parts of the graph. The algorithm keeps track of visited nodes, so that it processes each node only once.
- In depth-first traversal, we visit the starting node and then proceed to follow links through the graph until we reach a dead end. In an undirected graph, a node is a dead end if all of the nodes adjacent to it have already been visited. In a directed graph, if a node has no outgoing edges, we also have a dead end. When we reach a dead end, we back up along our path until we find an unvisited adjacent node and then continue in that new direction. The process will have completed when we back up to the starting node and all the nodes adjacent to it have been visited. When this algorithm is implemented, that choice will depend on how the edges of the graph are stored.

Depth First Traverse Algorithm

Algorithm dfsFromVertex(G, v)

1. $visited[v] = \text{TRUE}$
2. S.push(v)
3. **while not** S.isEmpty$()$ **do**
4. $\quad v \leftarrow S$.pop$()$
5. $\quad$ **for all** w adjacent to v **do**
6. $\quad\quad$ **if** $visited[w] = \text{FALSE}$ **then**
7. $\quad\quad\quad visited[w] = \text{TRUE}$
8. $\quad\quad\quad S$.push(w)

Depth First Traverse - Adjacency Matrix (Iterative)

```cpp
// Initialize an array to mark visited vertices
bool vis_dfs[N];
// This function traverses the graph in depth first search
void dfs(int src) {
    stack<int> stk;
    stk.push(src);  // push and mark the source node as visited
    vis_dfs[src] = true;
    // loop to print all vertices in the graph
    while (!stk.empty()) {
        int u = stk.top();
        stk.pop();  // pop and print the top node
        cout << u << ' ';
        // loop for all neighbors of the vertex u
        for (int v = 1; v <= n; v++) {
            if(adj[u][v] && vis_dfs[v] == false) {
                stk.push(v);  // push and mark the source node as visited
                vis_dfs[v] = true;
            }
        }
    }
}
```

Graphs-Adjacency-Matrix.cpp

Depth First Traverse - Adjacency List (Iterative)

```cpp
// Initialize an array to mark visited vertices
bool vis_dfs[N];
// This function traverses the graph in depth first search
void dfs(int src) {
    stack<int> stk;
    stk.push(src);  // push and mark the source node as visited
    vis_dfs[src] = true;
    // loop to print all vertices in the graph
    while (!stk.empty()) {
        int u = stk.top();
        stk.pop(); // pop and print the top node
        cout << u << ' ';
        // loop for all neighbors of the vertex u
        for (int v : adj[u]) {
            if (vis_dfs[v] == false) {
                stk.push(v);  // push and mark the source node as visited
                vis_dfs[v] = true;
            }
        }
    }
}
```

Graphs-Adjacency-List.cpp

Depth First Traverse - Adjacency Matrix (Recursive)

```cpp
// Initialize an array to mark visited vertices
bool vis[N];
// This function traverses the graph in depth first search
void dfs_util(int u) {
    // mark the current node as visited and print it
    vis[u] = true;
    cout << u << ' ';
    // recursive for all the vertices adjacent to this vertex
    for (int v = 1 ; v <= n ; v++) {
        // check if this vertex not visited yet
        if(adj[u][v] && vis[v] == false)
            dfs_util(v);
    }
}
// This function traverses the graph in depth first search
void dfs() {
    // Call the recursive helper function to print dfs traversal
    // starting from all vertices one by one
    for (int i = 1 ; i <= n ; i++)
        // check if this vertex not visited yet
        if (vis[i] == false)
            dfs_util(i);
}
```

Graphs-Adjacency-Matrix.cpp

Depth First Traverse - Adjacency List (Recursive)

```cpp
// Initialize an array to mark visited vertices
bool vis[N];
// This function traverses the graph in depth first search
void dfs_util(int u) {
    // mark the current node as visited and print it
    vis[u] = true;
    cout << u << ' ';
    // recursive for all the vertices adjacent to this vertex
    for (int v : adj[u]) {
        // check if this vertex not visited yet
        if(vis[v] == false)
            dfs_util(v);
    }
}
// This function traverses the graph in depth first search
void dfs() {
    // Call the recursive helper function to print dfs traversal
    // starting from all vertices one by one
    for (int i = 1 ; i <= n ; i++)
        // check if this vertex not visited yet
        if (vis[i] == false)
            dfs_util(i);
}
```

Graphs-Adjacency-List.cpp

Lecture Agenda

- ✔ Section 1: Introduction to Graphs
- ✔ Section 2: Directed vs. Undirected Graph
- ✔ Section 3: Breadth First Traverse
- ✔ Section 4: Depth First Traverse
- **Section 5: Cyclic vs. Acyclic Graph**
- Section 6: Connected vs. Disconnected Graph
- Section 7: Time Complexity & Space Complexity

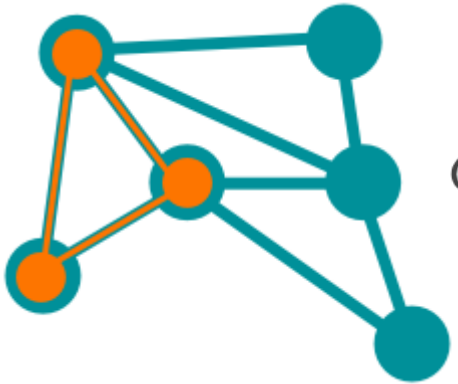
Cyclic vs. Acyclic Graph

- Graph contains cycle if there are any back edges. There are two types of back edges. Use DFS (Depth-First Search) to detect the back edge, Do the DFS from each vertex.

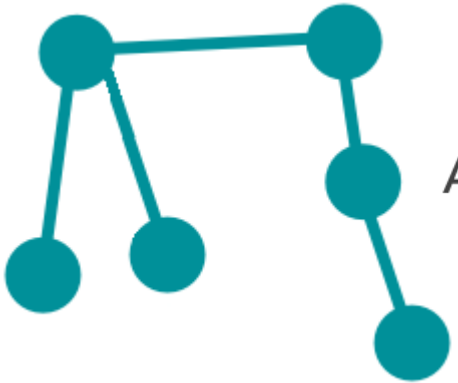
1- Edge from a vertex to itself. (Self loop)

2- Edge from any descendent back to vertex.

- For DFS from each vertex, keep track of visiting vertices in a recursion stack (array). If you encounter a vertex which already present in recursion stack then we have found a cycle. Use visited [] for DFS to keep track of already visited vertices.
- How different is recursion stack [] from visitied []. Visited[] is used to keep track of already visited vertices during the DFS is never gets, Recursion stack[] is used from keep track of visiting vertices during DFS from particular vertex and gets reset once cycle is not found from that vertex and will try DFS from other vertices.



Cyclic Graph

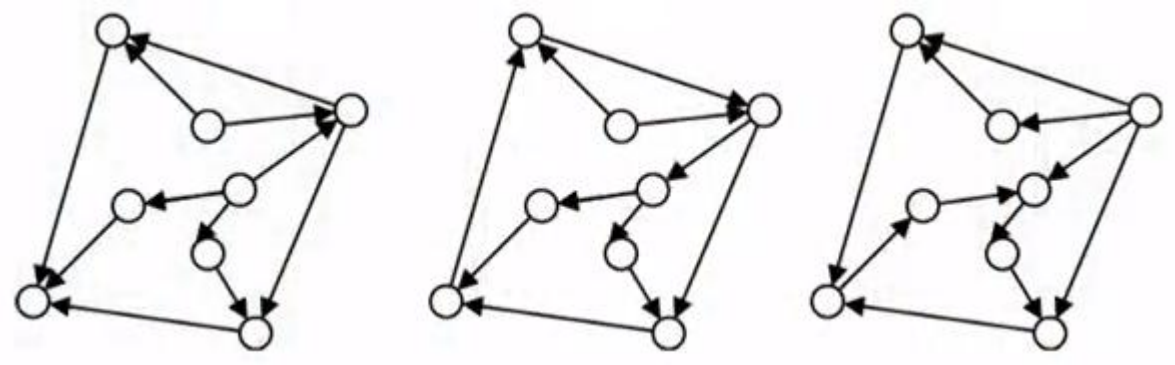


Acyclic Graph

Cyclic vs. Acyclic Graph

- Which of the following graphs is Cyclic, Acyclic Graph?

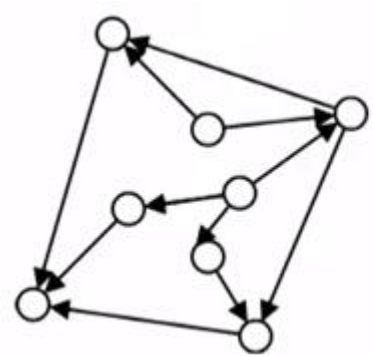
Graph 1 Graph 2 Graph 3



Cyclic vs. Acyclic Graph

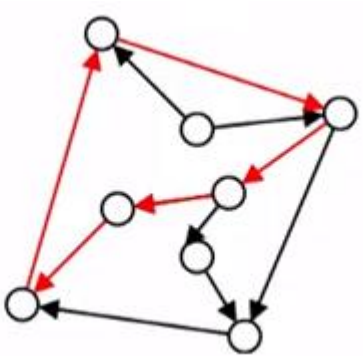
- Which of the following graphs is Cyclic, Acyclic Graph?

Graph 1



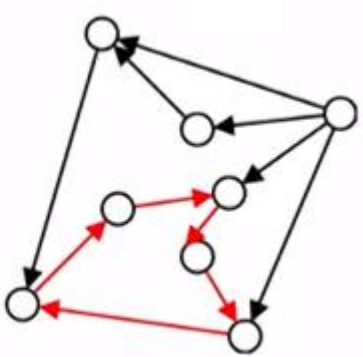
ACyclic Graph

Graph 2



Cyclic Graph

Graph 3



Cyclic Graph

Cyclic vs. Acyclic Graph

- Approach: Run a DFS from every unvisited node. Depth First Traversal can be used to detect a cycle in a Graph. DFS for a connected graph produces a tree. There is a cycle in a graph only if there is a back edge present in the graph. A back edge is an edge that is joining a node to itself (self-loop) or one of its ancestor in the tree produced by DFS. To find the back edge to any of its ancestor keep a visited array and if there is a back edge to any visited node then there is a loop and return true.
- Algorithm:

1- Create the graph using the given number of edges and vertices.

2- Create a recursive function that current index or vertex, visited and recursion stack.

3- Mark the current node as visited and also mark the index in recursion stack.

4- Find all the vertices which are not visited and are adjacent to the current node. Recursively call the function for those vertices, If the recursive function returns true then return true.

5- If the adjacent vertices are already marked in the recursion stack then return true.

6- Create a wrapper class, that calls the recursive function for all the vertices and if any function returns true, then return true. Else if for all vertices the function returns false then return false.

Cyclic vs. Acyclic Graph - Adjacency Matrix

```
bool dfs_cycle(int u, int par) {
    vis_cycle[u] = 1;
    for (int v = 1 ; v <= n ; v++) {
        if (adj[u][v] == false || v == par)
            continue;
        if (vis_cycle[v] == 0) {
            parent[v] = u;
            if (dfs_cycle(v, parent[v]))
                return true;
        }
        else if (vis_cycle[v] == 1) {
            cycle_end = u;
            cycle_start = v;
            return true;
        }
    }
    vis_cycle[u] = 2;
    return false;
}
```

- Initialize a global variables

```
int vis_cycle[N];
int parent[N];
int cycle_start, cycle_end;
```

Graphs-Adjacency-Matrix.cpp

Cyclic vs. Acyclic Graph - Adjacency Matrix

```
void find_cycle() {
    memset(parent, -1, sizeof parent);
    cycle_start = -1;
    for (int v = 0; v < n; v++) {
        if (vis_cycle[v] == 0 && dfs_cycle(v, parent[v]))
            break;
    }
    if (cycle_start == -1) {
        cout << "Acyclic Graph";
        return;
    }
    vector<int> cycle;
    cycle.push_back(cycle_start);
    for (int v = cycle_end; v != cycle_start; v = parent[v])
        cycle.push_back(v);
    cycle.push_back(cycle_start);
    reverse(cycle.begin(), cycle.end());
    cout << "Cycle found: ";
    for (int v : cycle)
        cout << v << " ";
}
```

Graphs-Adjacency-Matrix.cpp

Cyclic vs. Acyclic Graph - Adjacency List

```
bool dfs_cycle(int u, int p) {
    vis_cycle[u] = 1;
    for (int v : adj[u]) {
        if (v == p)
            continue;
        if (vis_cycle[v] == 0) {
            parent[v] = u;
            if (dfs_cycle(v, parent[v]))
                return true;
        }
        else if (vis_cycle[v] == 1) {
            cycle_end = u;
            cycle_start = v;
            return true;
        }
    }
    vis_cycle[u] = 2;
    return false;
}
```

- Initialize a global variables

```
int vis_cycle[N];
int parent[N];
int cycle_start, cycle_end;
```

Graphs-Adjacency-List.cpp

Cyclic vs. Acyclic Graph - Adjacency List

```
void find_cycle() {
    memset(parent, -1, sizeof parent);
    cycle_start = -1;
    for (int v = 0; v < n; v++) {
        if (vis_cycle[v] == 0 && dfs_cycle(v, parent[v]))
            break;
    }
    if (cycle_start == -1) {
        cout << "Acyclic Graph";
        return;
    }
    vector<int> cycle;
    cycle.push_back(cycle_start);
    for (int v = cycle_end; v != cycle_start; v = parent[v])
        cycle.push_back(v);
    cycle.push_back(cycle_start);
    reverse(cycle.begin(), cycle.end());
    cout << "Cycle found: ";
    for (int v : cycle)
        cout << v << " ";
}
```

Graphs-Adjacency-List.cpp

Lecture Agenda

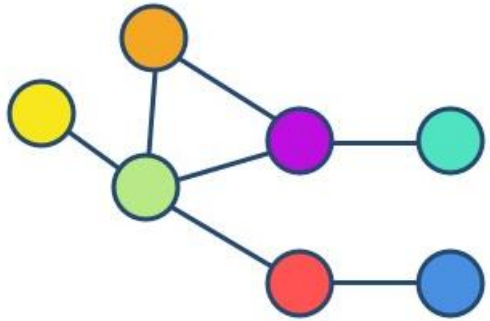
- ✓ Section 1: Introduction to Graphs
- ✓ Section 2: Directed vs. Undirected Graph
- ✓ Section 3: Breadth First Traverse
- ✓ Section 4: Depth First Traverse
- ✓ Section 5: Cyclic vs. Acyclic Graph
- **Section 6: Connected vs. Disconnected Graph**
- Section 7: Time Complexity & Space Complexity

Connected vs. Disconnected Graph

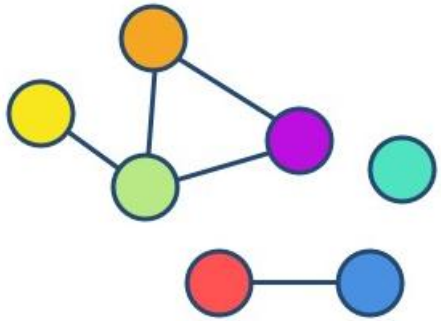
- Connectivity is a basic concept of graph theory. It defines whether a graph is connected or disconnected. Without connectivity, it is not possible to traverse a graph from one vertex to another vertex. A graph is said to be connected graph if there is a path between every pair of vertex. From every vertex to any other vertex there must be some path to traverse. This is called the connectivity of a graph. A graph is said to be disconnected, if there exists multiple disconnected vertices and edges.
- Graph connectivity theories are essential in network applications, routing transportation networks, network tolerance etc. In an undirected graph G, two vertices u and v are called connected if G contains a path from u to v. Otherwise, they are called disconnected. If the two vertices are additionally connected by a path of length 1, i.e. by a single edge, the vertices are called adjacent.
- A graph is said to be connected if every pair of vertices in the graph is connected. This means that there is a path between every pair of vertices. An undirected graph that is not connected is called disconnected. An undirected graph G is therefore disconnected if there exist two vertices in G such that no path in G has these vertices as endpoints. A graph with just one vertex is connected. An edgeless graph with two or more vertices is disconnected.

Connected vs. Disconnected Graph

- A directed graph is called weakly connected if replacing all of its directed edges with undirected edges produces a connected (undirected) graph. It is unilaterally connected or unilateral (also called semiconnected) if it contains a directed path from u to v or a directed path from v to u for every pair of vertices u, v. It is strongly connected, or simply strong, if it contains a directed path from u to v and a directed path from v to u for every pair of vertices u, v.
- When we traverse graph from single source that is mean traverses only the vertices reachable from a given source vertex. All the vertices may not be reachable from a given vertex (example Disconnected graph). To do complete DFS traversal of such graphs, we must call DFSUtil for every vertex. Also, before calling DFSUtil, we should check if it is already printed by some other call of DFSUtil. Following implementation does the complete graph traversal even if the nodes are unreachable.



Connected

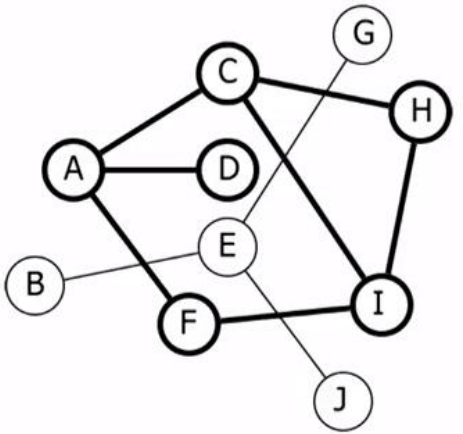


Disconnected

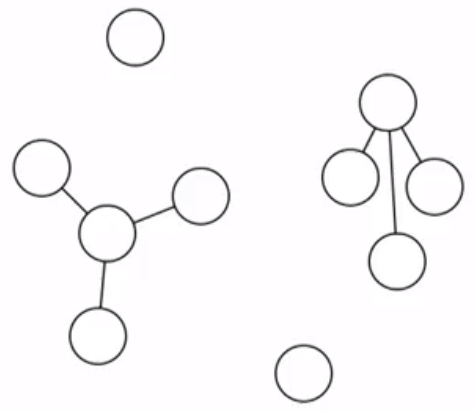
Connected vs. Disconnected Graph

- How many connected components does the graphs below have?

Graph 1



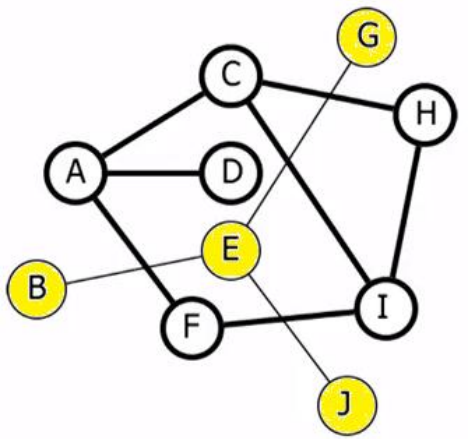
Graph 2



Connected vs. Disconnected Graph

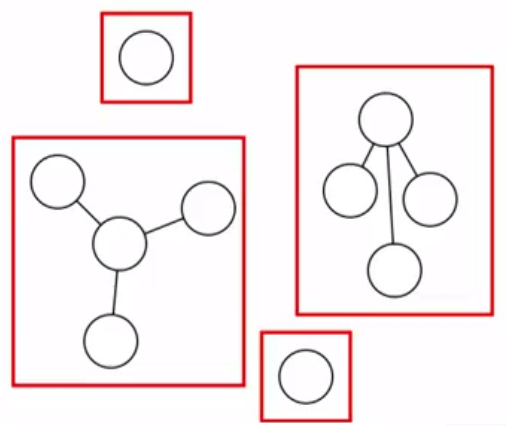
- How many connected components does the graphs below have?

Graph 1



2 components

Graph 2



4 components

Connected vs. Disconnected Graph - Adjacency Matrix

```
bool vis_connected[N];

void dfs_connected(int u) {
    vis_connected[u] = true;
    for (int v = 1 ; v <= n ; v++) {
        if (adj[u][v] && vis_connected[v] == false)
            dfs_connected(v);
    }
}
bool is_connected() {
    dfs_connected(1);
    for (int i = 1 ; i <= n ; i++) {
        if (vis_connected[i] == false)
            return false;
    }
    return true;
}
```

Graphs-Adjacency-Matrix.cpp

Connected vs. Disconnected Graph - Adjacency List

```
bool vis_connected[N];

void dfs_connected(int u) {
    vis_connected[u] = true;
    for (int v : adj[u]) {
        if (vis_connected[v] == false)
            dfs_connected(v);
    }
}
bool is_connected() {
    dfs_connected(1);
    for (int i = 1 ; i <= n ; i++) {
        if (vis_connected[i] == false)
            return false;
    }
    return true;
}
```

Graphs-Adjacency-List.cpp

Functionality Testing - Adjacency Matrix

- In the Main function:

```
n = 5;
add_undirected_edge(1, 2);
add_undirected_edge(1, 3);
add_undirected_edge(2, 4);
add_undirected_edge(5, 3);
add_undirected_edge(2, 3);
add_undirected_edge(1, 5);

cout << "Graph representation in bfs : \n";
bfs(1);
cout << "\n-----------------------------------------------------\n";
cout << "Graph representation in dfs iterative : \n";
dfs(1);
cout << "\n-----------------------------------------------------\n";
cout << "Graph representation in dfs recursion : \n";
dfs();
cout << "\n-----------------------------------------------------\n";
```

- Expected Output:

```
Graph representation in bfs :
1 2 3 5 4
---------------------------------------
Graph representation in dfs iterative :
1 5 3 2 4
---------------------------------------
Graph representation in dfs recursion :
1 2 4 3 5
---------------------------------------
```

Graphs-Adjacency-Matrix.cpp

Functionality Testing - Adjacency Matrix

- In the Main function:

```
cout << (is_connected()? "Connected Graph" : "Disconnected Graph");
cout << "\n-------------------------------------------------\n";
find_cycle();
cout << "\n-------------------------------------------------\n";
```

- Expected Output:

```
Connected Graph
------------------------
Cycle found: 1 2 3 1
------------------------
```

Graphs-Adjacency-Matrix.cpp

Functionality Testing - Adjacency List

➢ In the Main function:

```
n = 5;
add_undirected_edge(1, 2);
add_undirected_edge(1, 3);
add_undirected_edge(2, 4);
add_undirected_edge(5, 3);
add_undirected_edge(2, 3);
add_undirected_edge(1, 5);

cout << "Graph representation in bfs : \n";
bfs(1);
cout << "\n-----------------------------------------------------\n";
cout << "Graph representation in dfs iterative : \n";
dfs(1);
cout << "\n-----------------------------------------------------\n";
cout << "Graph representation in dfs recursion : \n";
dfs();
cout << "\n-----------------------------------------------------\n";
```

➢ Expected Output:

```
Graph representation in bfs :
1 2 3 5 4
---------------------------------------
Graph representation in dfs iterative :
1 5 3 2 4
---------------------------------------
Graph representation in dfs recursion :
1 2 4 3 5
---------------------------------------
```

Graphs-Adjacency-List.cpp

Functionality Testing - Adjacency List

- In the Main function:

```
cout << (is_connected()? "Connected Graph" : "Disconnected Graph");
cout << "\n-------------------------------------------------\n";
find_cycle();
cout << "\n-------------------------------------------------\n";
```

- Expected Output:

```
Connected Graph
------------------------
Cycle found: 1 2 3 1
------------------------
```

Graphs-Adjacency-List.cpp

Lecture Agenda

- ✓ Section 1: Introduction to Graphs
- ✓ Section 2: Directed vs. Undirected Graph
- ✓ Section 3: Breadth First Traverse
- ✓ Section 4: Depth First Traverse
- ✓ Section 5: Cyclic vs. Acyclic Graph
- ✓ Section 6: Connected vs. Disconnected Graph

Section 7: Time Complexity & Space Complexity

Time Complexity & Space Complexity

- Time & Space Analysis

	Worst Case (Adjacency Matrix)	Worst Case (Adjacency List)
• Space	$\Theta(V^2)$	$\Theta(V + E)$
• Connectivity of (i, j)	$\Theta(1)$	$\Theta(\text{out-degree}(V))$
• Connectivity of i	$\Theta(V)$	$\Theta(\text{out-degree}(V))$
• Traverse all Edges	$\Theta(V^2)$	$\Theta(V + E)$

Lecture Agenda

- ✓ Section 1: Introduction to Graphs
- ✓ Section 2: Directed vs. Undirected Graph
- ✓ Section 3: Breadth First Traverse
- ✓ Section 4: Depth First Traverse
- ✓ Section 5: Cyclic vs. Acyclic Graph
- ✓ Section 6: Connected vs. Disconnected Graph
- ✓ Section 7: Time Complexity & Space Complexity

Practice

Practice

1- Check if two nodes are on same path in a tree
2- Count all possible paths between two vertices
3- Count nodes within K-distance from all nodes in a set
4- Count number of trees in a forest
5- Count the number of nodes at given level in a tree using BFS
6- Delete edge to minimize subtree sum difference
7- Find a mother vertex in a Graph
8- Find k-cores of an undirected graph
9- Find length of the largest region in boolean matrix
10- Find the minimum number of moves needed to move from one cell of matrix to another
11- Find the smallest binary digit multiple of given number
12- Calculate height of a generic tree from parent array
13- Maximum product of two non-intersecting paths in a tree
14- Minimum edge reversals to make a root
15- Minimum initial vertices to traverse whole matrix with given conditions

Practice

16- Minimum number of edges between two vertices of a graph
17- Minimum number of operation required to convert number x into y
18- Minimum steps to reach end of array under constraints
19- Minimum steps to reach target by a knight
20- Move weighting scale alternate under given constraints
21- Number of pair of positions in matrix which are not accessible
22- Print all paths from a given source to a destination using BFS
23- Roots of a tree which give minimum height
24- Shortest path to reach one prime to other by changing single digit at a time
25- Stepping Numbers
26- Sum of the minimum elements in all connected components of an undirected graph
27- Check if a graphs has a cycle of odd length
28- Check if there is a cycle with odd weight sum in an undirected graph
29- Check loop in array according to given constraints
30- Cycles of length n in an undirected and connected graph

Practice

31- Detect cycle in a directed graph
32- Detect cycle in an undirected graph
33- Check if a given directed graph is strongly connected using BFS
34- Check if a given graph is tree or not
35- Check if a graph is strongly connected using DFS
36- Check if removing a given edge disconnects a graph
37- Count all possible walks from a source to a destination with exactly k edges
38- Count single node isolated sub-graphs in a disconnected graph
39- Count the number of non-reachable nodes
40- Find all reachable nodes from every node present in a given set
41- Find if there is a path between two vertices in a directed graph
42- Find if there is a path of more than k length from a source
43- Find the Degree of a Particular vertex in a Graph
44- Find the minimum cost to reach destination using a train
45- Find the number of islands using DFS

Practice

46- Find k'th heaviest adjacent node in a graph where each vertex has weight

47- Minimum cost to connect weighted nodes represented as array

48- Number of cyclic elements in an array where we can jump according to value

49- Number of groups formed in a graph of friends

50- Number of loops of size k starting from a specific node


DO
MORE.