Graph Data Structure

A graph is a non-linear data structure consisting of nodes (or vertices) and edges (connections between nodes). Graphs are widely used to model relationships and interactions in various domains such as computer networks, social networks, transportation systems, and more.

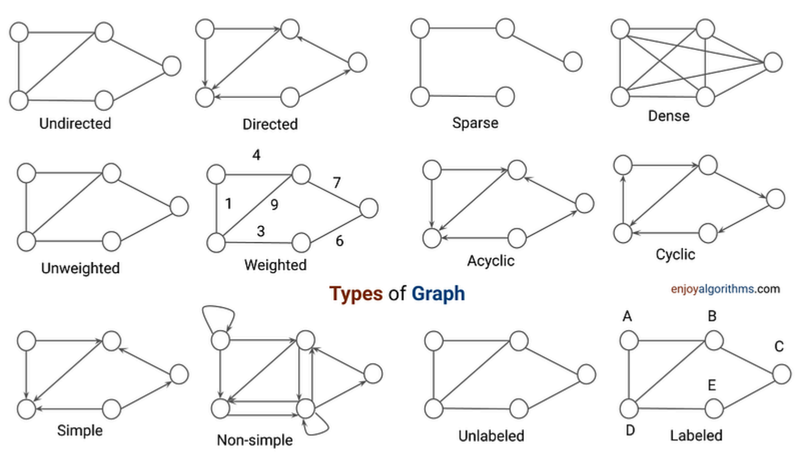
Key Concepts

1. **Vertices (Nodes)**: The fundamental units of the graph which represent entities.
2. **Edges (Links)**: The connections between vertices that represent relationships or interactions.

Terminology

Term	Definition
Degree	Number of edges connected to a vertex
In-degree	Number of edges coming into a vertex in a directed graph
Out-degree	Number of edges going out of a vertex in a directed graph
Simple path	Path between two vertices are simply if no vertex is repeated.
Path cost	The cumulative sum of all the weights of the edges in a path in a weighted graph
Reachability	A vertex U is reachable from V if there is a path from U to V
Length of a Path	Number of edges in the path
Subgraph	A subset of the vertices of a graph
Dense Graph	A graph is dense if the number of edges (|E|) is nearly equal to |V|^2
Sparse Graph	A graph is sparse if the number of edges |E| << |V|^2
Strongly Connected	There is a path between every pair of edges. Therefore |E| is greater than or equal to |V| - 1
Complete graph	This means that there are no isolated vertices, and each vertex is directly connected to every other vertex.

Types of Graphs



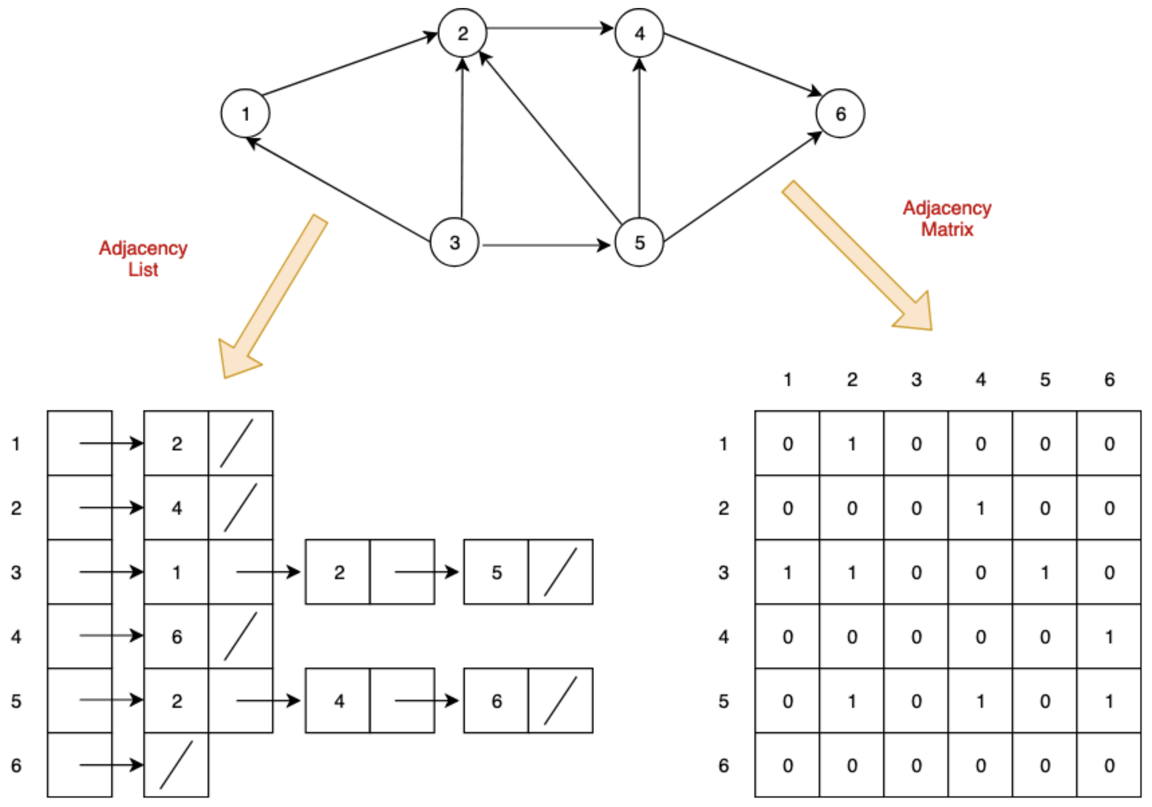
1. **Directed Graph (Digraph)**: A graph where edges have a direction, indicating a one-way relationship from one vertex to another. Represented as (u,v) where the edge points from u to v.
2. **Undirected Graph**: A graph where edges have no direction, indicating a bi-directional relationship. Represented as $\{u,v\}$ where the edge connects uu and vv without any direction.
3. **Weighted Graph**: A graph where edges have associated weights, representing the cost, distance, or any other metric between vertices.
4. **Unweighted Graph**: A graph where edges do not have weights.
5. **Cyclic Graph**: A graph containing at least one cycle, a path of edges and vertices wherein a vertex is reachable from itself.
6. **Acyclic Graph**: A graph without cycles.
7. **Connected Graph**: An undirected graph where there is a path between any two vertices.
8. **Disconnected Graph**: An undirected graph where at least two vertices do not have a path between them.

Relationship between number of edges and vertices in graphs

Type of Graph	Definition	Number of Edges (E)	Description
`Undirected`			
Complete Undirected Graph	Each pair of distinct vertices is connected by an edge. For a complete graph with V vertices:	$V \times (V-1)$ / 2	A graph with maximum possible edges.
Sparse Undirected Graph	In a sparse undirected graph (fewer edges relative to vertices), the number of edges is typically much smaller than in a complete graph.	Varies	A graph with relatively few edges.

Type of Graph	**Definition**	**Number of Edges (E)**	**Description**
`Directed`			
Complete Directed Graph	Each pair of distinct vertices is connected by both an outgoing and an incoming edge. For a complete directed graph with V vertices:	$V×(V−1)$	A directed graph with maximum edges.
Sparse Directed Graph	Similar to undirected sparse graphs, the relationship between edges and vertices in a sparse directed graph can vary based on the specific characteristics and application of the graph.	Varies	A directed graph with relatively few edges.

Graph Representation



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

Data Structure	Description	Pros	Cons	To Listing all adjacent vertices of u)	Checking availability of Edge (u,v)	Space Complexity
Adjacency List	An array of lists. Each index represents a vertex, and the list at each index contains the neighbors of that vertex.	Space-efficient for sparse graphs.	Slower edge lookup compared to adjacency matrix.	O(degree(u)) this turns to O(V) in worst case	$O(V)$ -Not efficient	$O(V+E)$
Adjacency Matrix	A 2D array where rows represent source vertices and columns represent destination vertices. Cell (i, j) indicates the presence (and possibly the weight) of an edge from vertex i to vertex j*.	Quick edge lookup.	Space-inefficient for sparse graphs.	$O(V)$	$O(1)$	$O(V$^2)

- `V` refers to the number of vertices in the graph.
- `E` refers to the number of edges in the graph.

Graph Traversal Algorithms

1. **Breadth-First Search (BFS)**: Explores the graph level by level starting from a source vertex. Uses a queue to manage the exploration process.
   - **Applications**: Shortest path in unweighted graphs, peer-to-peer networks, social networking sites.
2. **Depth-First Search (DFS)**: Explores as far along a branch as possible before backtracking. Uses a stack (often implemented via recursion).
   - **Applications**: Topological sorting, finding connected components, solving puzzles.

Advanced Concepts

1. **Topological Sort**: An ordering of vertices in a directed acyclic graph (DAG) such that for any directed edge uv, vertex u comes before v in the ordering.
2. **Strongly Connected Components (SCC)**: Maximal subgraphs in a directed graph where each vertex is reachable from every other vertex in the subgraph.
3. **`Minimum Spanning Tree (MST)`**: A subset of edges that connects all vertices in the graph without any cycles and with the minimum possible total edge weight.
   - **Algorithms**: Kruskal’s and Prim’s algorithms.

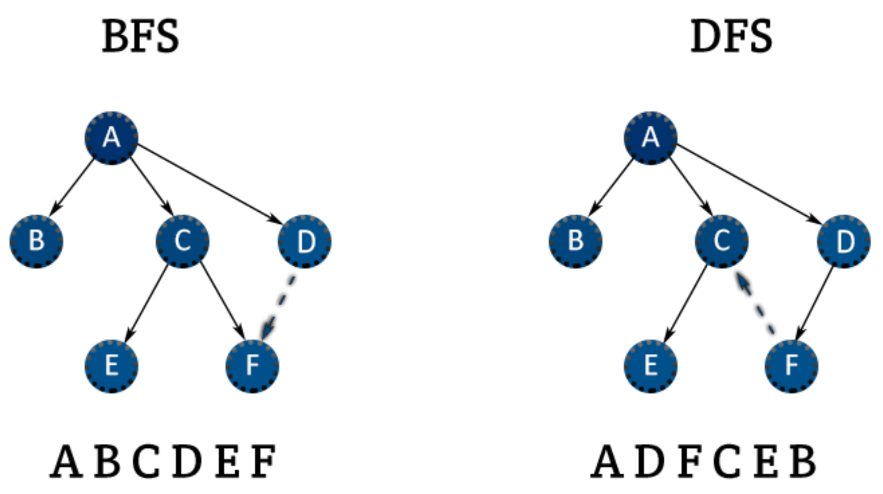
4. `Shortest Path Algorithms` : Algorithms to find the shortest path between vertices.
    - **Dijkstra's Algorithm**: Efficient for graphs with non-negative weights.
    - **Bellman-Ford Algorithm**: Handles graphs with negative weights and detects negative cycles.
    - **Floyd-Warshall Algorithm**: Computes shortest paths between all pairs of vertices.

Use Cases of Graphs

- **Social Networks**: Modeling relationships and interactions between users.
- **Transportation Networks**: Planning routes and analyzing connectivity.
- **Computer Networks**: Routing and network topology design.
- **Biological Networks**: Analyzing molecular interactions and gene regulation.
- **Recommendation Systems**: Identifying related items or users.

Graphs are versatile and powerful data structures that provide a robust framework for solving a wide range of computational problems. Understanding their properties and the various algorithms associated with them is crucial for effectively leveraging their potential in practical applications.

Traversing Graphs



1. ### Breadth-First Search (BFS)

   Breadth-First Search (BFS) is a fundamental graph traversal algorithm that explores a graph level by level. Starting from a designated source vertex, BFS systematically explores all the vertices reachable from that source vertex, visiting neighboring vertices before moving on to vertices at the next level. `BFS is widely used in various applications such as shortest path finding, connectivity analysis, and network flow algorithms`.

   #### Algorithm:

   1. **Initialization**: Start with a source vertex s and mark it as visited.
   2. **Queue Initialization**: Enqueue the source vertex s into a queue.
   3. Exploration Loop : Repeat until the queue is empty.
       - Dequeue a vertex v from the queue.

   - Visit vertex v and process it.
     - Enqueue all unvisited neighbors of v into the queue and mark them as visited.

4. **Termination**: The algorithm terminates when the queue becomes empty.

Key Features:

- **Level-by-Level Exploration**: BFS explores vertices in the order of their distance from the source vertex, ensuring that closer vertices are visited before farther ones.

- **Shortest Path Finding**: BFS can be used to find the shortest path from the source vertex to any other vertex in an unweighted graph.
  - **Connectivity Analysis**: BFS can determine whether a graph is connected and identify connected components.
  - **Avoiding Cycles**:`BFS naturally avoids revisiting already visited vertices, making it suitable for graphs with cycles.`

Applications:

1. **Shortest Path Finding**: `BFS can find the shortest path from a source vertex to all other vertices in an unweighted graph.`

2. **Connectivity Analysis**: BFS can determine whether a graph is connected and identify connected components.

3. **Network Routing**: BFS can be used to explore and find paths in computer networks.
4. **Web Crawling**: BFS is used in web crawlers to systematically explore and index web pages.

Time Complexity:

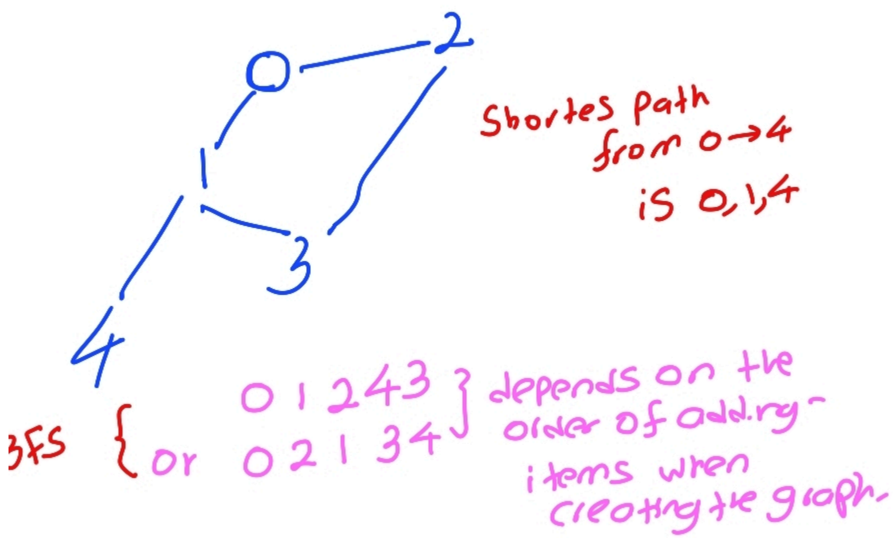
- `The time complexity of BFS is` $O(V+E)$, where V is the number of vertices and E is the number of edges. This is because each vertex and each edge is processed once.

Space Complexity:

- `The space complexity of BFS is` $O(V)$ where V is the number of vertices. This is because BFS typically requires storing a queue of vertices to be explored.

Breadth-First Search is a versatile and powerful algorithm with a wide range of applications in graph theory and real-world problems. Its simplicity and efficiency make it a popular choice for many graph-related tasks.

Example


0
2
1
3
4
Shortes Path from 0→4 is 0,1,4
BFS { 0 1 2 4 3 or 0 2 1 3 4 } depends on the order of adding items when creating the graph.

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <algorithm> // for std::reverse
using namespace std;

// Function to add an edge to the graph
void addEdge(vector<vector<int>>& graph, int u, int v) {
    graph[u].push_back(v);
    graph[v].push_back(u);
}

// Function to perform BFS on the graph and print vertices in BFS order
void bfs(vector<vector<int>> graph, int start) {
    queue<int> q;
    vector<bool> visited(graph.size(), false); // Keeps track of visited nodes
    q.push(start);
    visited[start] = true;

    while (!q.empty()) {
        int current = q.front();
        q.pop();
        cout << current << " "; // Print the current node

        // Explore all the neighbors of the current node
        for (auto neighbor : graph[current]) {
            if (!visited[neighbor]) {
                q.push(neighbor);
                visited[neighbor] = true;
            }
        }
    }
}

// Function to find the shortest path from start to end using BFS
vector<int> bfsShortestPath(vector<vector<int>> graph, int start, int end) {
    queue<int> q;
    vector<bool> visited(graph.size(), false); // Keeps track of visited
nodes
    vector<int> parent(graph.size(), -1); // Keeps track of the parent of
each node
    q.push(start);
    visited[start] = true;

    while (!q.empty()) {
        int current = q.front();
        q.pop();

        // If the end node is reached, reconstruct the path
        if (current == end) {
            vector<int> shortestPath;
            while (parent[current] != -1) {
                shortestPath.push_back(current);
                current = parent[current];
            }
            shortestPath.push_back(start); // Add the start node to the path
            reverse(shortestPath.begin(), shortestPath.end()); // Reverse to
get the correct order
            return shortestPath;
        }

        // Explore all the neighbors of the current node
        for (auto neighbor : graph[current]) {
            if (!visited[neighbor]) {
                q.push(neighbor);
                parent[neighbor] = current; // Set the parent of the neighbor
                visited[neighbor] = true;
            }
        }
    }

    return {}; // Return an empty path if there is no path from start to end
}

int main() {
    int vertices = 5; // Number of vertices in the graph
    vector<vector<int>> graph(vertices); // Initialize the graph with the
given number of vertices

    // Add edges to the graph
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 1, 3);
    addEdge(graph, 1, 4);
    addEdge(graph, 2, 4);

    // Perform BFS starting from vertex 0
    cout << "BFS order: ";
    bfs(graph, 0);
    cout << endl;

    // Find and print the shortest path from vertex 0 to vertex 4
    vector<int> shortestPath = bfsShortestPath(graph, 0, 4);
    cout << "Shortest path from 0 to 4: ";
    for (int x : shortestPath) {
        cout << x << " ";
    }
    cout << endl;

    return 0;
}
```

3. Depth-First Search (DFS)

Depth-First Search (DFS) is a fundamental graph traversal algorithm that explores a graph by going as deep as possible along each branch before backtracking. Starting from a designated source vertex, DFS systematically explores each branch of the graph to its deepest point before moving on to explore other branches. DFS is widely used in various applications such as `pathfinding, cycle detection, and topological sorting`.

Algorithm:

1. **Initialization**: Start with a source vertex s and mark it as visited.
2. **Stack Initialization**: Push the source vertex s onto a stack.
3. **Exploration Loop**: Repeat until the stack is empty.
   - Pop a vertex v from the stack.
   - Visit vertex v and process it.
   - Push all unvisited neighbors of v onto the stack and mark them as visited.
4. **Termination**: The algorithm terminates when the stack becomes empty.

Alternatively, DFS can be implemented recursively:

Recursive Call: For each unvisited neighbor of the current vertex, recursively perform DFS on that neighbor.

Key Features:

- **Deep Exploration**: DFS explores each branch of the graph to its deepest point before backtracking to explore other branches.
- **Pathfinding**: DFS can be used to find paths between vertices in a graph.
- **Cycle Detection**: DFS can detect cycles in both directed and undirected graphs by identifying back edges.
- **Topological Sorting**: DFS can be used to perform topological sorting in a Directed Acyclic Graph (DAG).

Applications:

1. **Pathfinding**: DFS can find paths between vertices in a graph.
2. **Cycle Detection**: DFS can detect cycles in both directed and undirected graphs.
3. **Topological Sorting**: DFS is used in topological sorting of a Directed Acyclic Graph (DAG).

4. **Strongly Connected Components**: DFS is used in algorithms like Kosaraju's and Tarjan's to find strongly connected components in a graph.
5. **Maze Solving**: DFS can be used to explore and solve mazes by exploring all possible paths.

Time Complexity:

- The time complexity of DFS is $O(V+E)$, where V V is the number of vertices and E E is the number of edges. This is because each vertex and each edge is processed once.

Space Complexity:

- The space complexity of DFS is $O(V)$, where V is the number of vertices. This is because DFS typically requires storing a stack (or the call stack in the case of the recursive implementation) of vertices to be explored.

Iterative vs. Recursive Implementation:

- **Iterative DFS**: Uses an explicit stack to manage vertices.
    - Pros: Avoids potential stack overflow issues.
    - Cons: Slightly more complex to implement compared to the recursive version.
- **Recursive DFS**: Uses the system's call stack for managing vertices.
    - Pros: Simple and elegant to implement.
    - Cons: Can lead to stack overflow for deep graphs.

Depth-First Search is a versatile and powerful algorithm with a wide range of applications in graph theory and computer science. Its deep exploration strategy makes it suitable for solving various complex problems involving graphs and networks.

Example

Recursive DFS

```cpp
#include <iostream>
#include <vector>
using namespace std;

// Function to add an edge to the graph
void addEdge(vector<vector<int>>& graph, int u, int v) {
    graph[u].push_back(v);
    graph[v].push_back(u);
}

// Helper function for recursive DFS
void dfsRecursiveUtil(const vector<vector<int>>& graph, int current,
vector<bool>& visited) {
    visited[current] = true;
    cout << current << " "; // Print the current node

    // Recur for all the vertices adjacent to this vertex
    for (auto neighbor : graph[current]) {
        if (!visited[neighbor]) {
            dfsRecursiveUtil(graph, neighbor, visited);
        }
    }
}

// Function to perform recursive DFS
void dfsRecursive(const vector<vector<int>>& graph, int start) {
    vector<bool> visited(graph.size(), false); // Keeps track of visited
nodes
    dfsRecursiveUtil(graph, start, visited);
}

int main() {
    int vertices = 5; // Number of vertices in the graph
    vector<vector<int>> graph(vertices); // Initialize the graph with the
given number of vertices

    // Add edges to the graph
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 1, 3);
    addEdge(graph, 1, 4);
    addEdge(graph, 2, 4);

    // Perform recursive DFS starting from vertex 0
    cout << "DFS (Recursive) order: ";
    dfsRecursive(graph, 0);
    cout << endl;

    return 0;
}
```

Iterative DFS

```
#include <iostream>
#include <vector>
#include <stack>
using namespace std;

// Function to add an edge to the graph
void addEdge(vector<vector<int>>& graph, int u, int v) {
    graph[u].push_back(v);
    graph[v].push_back(u);
}

// Function to perform iterative DFS
void dfsIterative(const vector<vector<int>>& graph, int start) {
    vector<bool> visited(graph.size(), false); // Keeps track of visited nodes
    stack<int> s;
    s.push(start);

    while (!s.empty()) {
        int current = s.top();
        s.pop();
        if (!visited[current]) {
            visited[current] = true;
            cout << current << " "; // Print the current node

            // Push all the adjacent vertices of the current node onto the stack
            for (auto it = graph[current].rbegin(); it != graph[current].rend();
++it) {
                if (!visited[*it]) {
                    s.push(*it);
                }
            }
        }
    }
}

int main() {
    int vertices = 5; // Number of vertices in the graph
    vector<vector<int>> graph(vertices); // Initialize the graph with the given
number of vertices

    // Add edges to the graph
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 1, 3);
    addEdge(graph, 1, 4);
    addEdge(graph, 2, 4);

    // Perform iterative DFS starting from vertex 0
    cout << "DFS (Iterative) order: ";
    dfsIterative(graph, 0);
    cout << endl;

    return 0;
}
```

When to Use BFS

1. **Finding the Shortest Path in an Unweighted Graph**:
    - BFS is ideal for finding the shortest path from a source vertex to a target vertex in an unweighted graph because it explores all neighbors at the present depth level before moving on to nodes at the next depth level.
    - Example: Finding the shortest route in a road network.
2. **Level-Order Traversal**:
    - BFS is useful when you need to explore nodes level by level.
    - Example: Printing nodes of a tree level by level.
3. **Connectivity and Component Detection**:
    - BFS can determine if a graph is connected and identify all connected components in an undirected graph.

  - Example: Checking if a network of computers is fully connected.

4. **Finding All Nodes within One Connected Component**:
   - BFS is efficient for finding all nodes reachable from a given node in an unweighted graph.
   - Example: Identifying all reachable friends in a social network starting from a particular user.

When to Use DFS

1. **Pathfinding in Specific Scenarios**:
   - DFS is useful when you need to explore all paths or if you need to perform exhaustive search.
   - Example: Finding a path in a maze.
2. **Cycle Detection**:
   - DFS can detect cycles in both directed and undirected graphs by identifying back edges.
   - Example: Detecting circular dependencies in a dependency graph.
3. **Topological Sorting**:
   - DFS is used to perform topological sorting in a Directed Acyclic Graph (DAG).
   - Example: Scheduling tasks with dependencies.
4. **Strongly Connected Components**:
   - DFS is used in algorithms like Kosaraju's and Tarjan's to find strongly connected components in directed graphs.
   - Example: Identifying strongly connected components in a web graph.
5. **Tree Traversals**:
   - DFS is used for tree traversals like in-order, pre-order, and post-order traversals.
   - Example: Evaluating expressions represented as binary trees.

Summary of Differences

Criteria	**BFS**	**DFS**
Exploration Strategy	Level-by-level	Depth-first
Shortest Path in Unweighted Graph	Yes	No
Time Complexity	$O(V + E)$	$O(V + E)$
Space Complexity	$O(V)$ (queue)	$O(V)$ (stack or recursion)
Cycle Detection	Possible (requires additional logic)	Yes
Finding All Paths	No	Yes
Tree Traversal	No	Yes
Topological Sorting	No	Yes

Criteria	BFS	DFS
Strongly Connected Components	No	Yes
Memory Usage	Generally higher due to storing all vertices at a particular level	Generally lower in sparse graphs

---

Parentheses theorem (For DFS)

The "parenthesis theorem" is typically associated with the concept of depth-first search (DFS) in graph theory, particularly in the context of trees and directed graphs. It provides a way to describe the entry and exit times of nodes during a DFS traversal.

Parenthesis Theorem in DFS

When performing a DFS on a graph, we can track the times when we first discover (or enter) a node and when we finish processing (or exit) a node. These times can be thought of as opening and closing parentheses in a well-formed expression. This analogy leads to the parenthesis theorem, which can be stated as follows:

1. **Discovery and Finish Times**: For each node uu in a DFS traversal, the discovery time $d[u]d[u]$ is recorded when uu is first encountered, and the finish time $f[u]f[u]$ is recorded when the DFS has finished processing all of uu's descendants.
2. Parenthesis Structure

   : These times form a well-parenthesized structure. Specifically, for any two nodes u and v, one of the following three conditions holds:
   - The intervals [$[d[u], f[u]$][*d*[*u*] and $[d[v], f[v]]$[*d*[*v*] are entirely disjoint, meaning neither u nor v is a descendant of the other.
   - The interval $[d[u], f[u]]$[*d*[*u*] is entirely contained within the interval $[d[v], f[v]]$[*d*[*v*] meaning u is a descendant of v.
   - The interval $[d[v], f[v]]$[*d*[*v*] is entirely contained within the interval $[d[u], f[u]]$[*d*[*u*] meaning v is a descendant of u.

Implications of the Parenthesis Theorem

The theorem helps in understanding the hierarchical structure of a graph as discovered by DFS. It is particularly useful in algorithms that rely on DFS, such as finding strongly connected components, topological sorting, and detecting cycles in directed graphs.

Example

Consider the following DFS traversal of a graph:

```
    1
   / \
  2   3
 / \
4   5
```

Let's say the DFS starts at node 1 and proceeds as follows:

- Visits node 1 (discovery time d[1]=1).
- Visits node 2 (discovery time d[2]=2).
- Visits node 4 (discovery time d[4]=3, finish time f[4]=4).
- Backtracks to node 2 and visits node 5 (discovery time d[5]=5, finish time f[5]=6).
- Backtracks to node 2 (finish time f[2]=7).
- Backtracks to node 1 and visits node 3 (discovery time d[3]=8, finish time f[3]=9).
- Backtracks to node 1 (finish time f[1]=10).

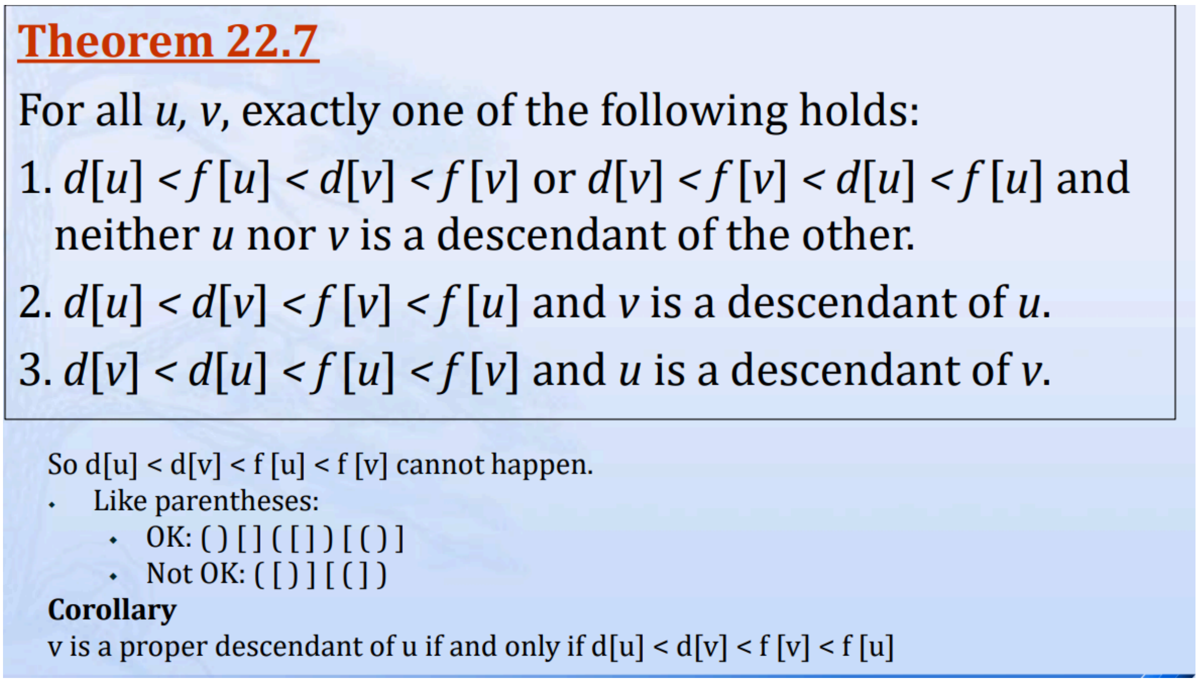
The discovery and finish times will be:

- Node 1: d[1]=1, f[1]=10
- Node 2: d[2]=2, f[2]=7
- Node 3: d[3]]=8, f[3]=9
- Node 4: d[4]=3, f[4]=4
- Node 5: d[5]=5, f[5]=6

Using the parenthesis theorem, we can interpret these intervals:

- The interval [1,10][1,10] contains `[2,7]`, `[3,9]`, `[3,4]`, and `[5,6]`.
- The interval [2,7][2,7] contains `[3,4]` and `[5,6]`.
- The intervals [3,4][3,4] and `[5,6]` are disjoint.
- The interval [8,9][8,9] is disjoint from `[2,7]`.

This hierarchical containment reflects the tree structure of the DFS traversal.



Theorem 22.7

For all *u*, *v*, exactly one of the following holds:

1. $d[u] < f[u] < d[v] < f[v]$ or $d[v] < f[v] < d[u] < f[u]$ and neither *u* nor *v* is a descendant of the other.
2. $d[u] < d[v] < f[v] < f[u]$ and *v* is a descendant of *u*.
3. $d[v] < d[u] < f[u] < f[v]$ and *u* is a descendant of *v*.

So d[u] < d[v] < f [u] < f [v] cannot happen.

- Like parentheses:
  - OK: ( ) [ ] ([ ]) [( )]
  - Not OK: ([ )] [( ])

Corollary

v is a proper descendant of u if and only if d[u] < d[v] < f [v] < f [u]

Questions

Adding a vertex in adjacency list representation is harder than adjacency matrix representation.

Select one:

- ○ True
- [✓ False]

No, it is not true that adding a vertex in adjacency list representation is harder than in adjacency matrix representation. In fact, adding a vertex is typically easier in an adjacency list representation than in an adjacency matrix representation. Here's why:

Adjacency List Representation

- **Adding a Vertex**: To add a vertex, you simply add a new list (or an empty data structure) for the new vertex. This operation is $O(1)$ in terms of time complexity because it involves a single addition to the list of vertices.
- **Adding an Edge**: This involves adding an entry to the list corresponding to the start vertex (and possibly the end vertex in an undirected graph). The time complexity for this is also $O(1)$ per edge addition.

Adjacency Matrix Representation

- **Adding a Vertex**: To add a vertex in an adjacency matrix, you need to resize the matrix to accommodate the new vertex. This involves creating a new matrix with dimensions $(V+1) \times (V+1)$ (where V is the number of vertices before addition) and copying over the old matrix values. The time complexity for this operation is $O(V^2)$ because you need to copy all the existing entries into the new matrix.
- **Adding an Edge**: This operation is $O(1)$ since you just set the appropriate entry in the matrix.

Summary

- **Adjacency List**: Easier to add a vertex (O(1) operation).
- **Adjacency Matrix**: Harder to add a vertex (O(V^2) operation).

Therefore, the statement that adding a vertex in adjacency list representation is harder than in adjacency matrix representation is not true. It is, in fact, easier to add a vertex in the adjacency list representation.

---

a and b are nodes in a depth first search tree. Which of the following statements are true regarding these?

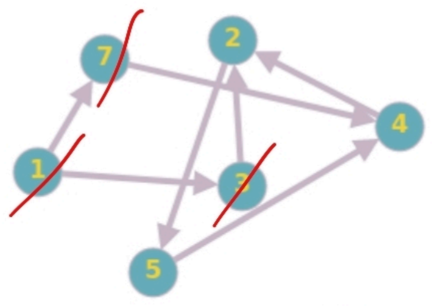
discover time

b is a descendant of a

descendant කෙනෙක් වුවම දෙපැත්තෙන් bound විය යුතුයි

- ☐ If a is a descendant of b, then d[a] < d[b] < f[b] < f[a] ← finishing time
- ☐ There cannot exist a case where d[a] < d[b] < f[a] < f[b]
- ☑ If d[a] < d[b] < f[b] < f[a], then b is a descendant of a
- ☐ If neither a nor b is a descendant of each other, then it has to be that d[a] < f[a] < d[b] < f[b]   or d[b]<f[b]<d[a]<f[a]
- ☐ If neither a nor b is a descendant of each other, then it has to be that d[b] < d[a] < f[b] < f[a]   This cannot happen.

Find an output of a Breadth-First Traversal of the following graph starting from Node 1:



137245

or 173425

Select one:

- 1 3 2 5 4 7
- 1 7 3 4 2 5
- 1 3 7 4 2 5
- [✓ 1 3 7 2 4 5]