Task 5: Trees and Graphs

Tree basics

Definition

A tree is a non-linear, hierarchical data structure consisting of nodes connected by edges

Basic terminologies

1) Root: The starting node of the tree from which all other nodes originate
2) Leaf: The last or deepest node in a tree. This node has no children
3) Parent: The node which has children, ie, other nodes connected directly to it
4) Child: The node that is connected to some other node above it
5) Sibling: When two child nodes share the same parent, they are called sibling nodes to each other
6) Subtree: A smaller unit of a tree which itself is a tree
7) Height of a tree: The maximum distance from the root to the deepest leaf node. It is 0 indexed
8) Depth of a node: The number of edges to traverse from the root to reach that particular node. It is 0 indexed

Binary Tree v/s N-ary Tree

- Binary tree: Each node can have at most 2 children [0, 1 or 2 children].
- N-ary tree: A node can have n-number of children [0…N children]. Binary trees are a special type of N-ary trees.

Other types of trees

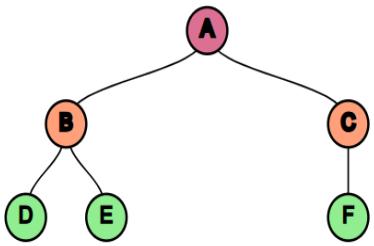
Task 5: Trees and Graphs

1) Full binary tree: A binary tree in which a node can have either 0 or 2 children. A single child is not allowed.
2) Complete binary tree: A binary tree in which all levels except possibly the last one have two children, and all the nodes are as left aligned as possible.
3) Perfect binary tree: A binary tree in which all internal nodes have two children, and all the leaf nodes are at the same level
4) Balanced binary tree: A binary tree the height of which can be expressed in $\log_2 n$

Source code to build a sample binary tree:

```
# Define the initial class for Nodes of the tree
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

# Building the sample binary tree
def build_sample_tree():
    root = Node('A')
    root.left = Node('B')
    root.right = Node('C')
    root.left.left = Node('D')
    root.left.right = Node('E')
    root.right.right = Node('F')
```



The tree will look like:

Task 5: Trees and Graphs

Graphs Basics

Definition

A graph is a non-linear data structure which consists of a set of vertices (nodes) and edges (links).
It is used to represent relationships between entities in a program or a system.

Directed v/s Undirected Graph

Feature	**Directed Graph**	**Undirected Graph**
Edge Direction	Edges have direction (A → B)	Edges have no direction (A — B)
Representation	Ordered pair of nodes (u, v)	Unordered pair of nodes {u, v}
Reverse Edge	(A → B) ≠ (B → A)	(A — B) = (B — A)
Use Case	Web links, one-way roads	Social networks, mutual friendships

Weighted v/s Unweighted Graph

Feature	**Weighted Graph**	**Unweighted Graph**
Edge Values	Each edge has a weight (cost, time)	All edges are treated equally
Path Cost	Path cost is sum of weights	Path cost is number of edges

Task 5: Trees and Graphs

Use Case	Maps, networks with costs	Simple connections, basic social graphs

Cyclic v/s Acyclic Graph

Feature	**Cyclic Graph**	**Acyclic Graph**
Cycle Presence	Contains at least one cycle	No cycles present
Repetition	Nodes/edges can be revisited	No node is revisited in a simple path
Use Case	Transport loops, circuit designs	Task scheduling, dependency graphs

Connected v/s Disconnected Graph

Feature	**Connected Graph**	**Disconnected Graph**
Reachability	All nodes reachable from any other	Some nodes are not reachable
Components	One connected component	Multiple separate components
Use Case	Fully functioning networks	Broken or isolated network systems

---

Tree Traversal

Types

1) DFS (Depth first search):

Task 5: Trees and Graphs

- Pre-order: Root, Left, Right
- In-order: Left, Root, Right
- Post-order: Left, Right, Root

2) BFS (Breadth first search)

DFS

1) Pre-order(Root-Left-Right):
   - Recursive source code:

```
# Pre-order traversal
def preorder(node):
    if node:
        print(node.value, end=' ')
        preorder(node.left)
        preorder(node.right)
```

   - Iterative source code:

```
def iterative_preorder(root):
    if not root:
        return
    stack = [root]
    while stack:
        node = stack.pop()
        print(node.value, end=' ')
        if node.right:
            stack.append(node.right)
        if node.left:
            stack.append(node.left)
```

   Output: A B D E C F

2) In-order(Left-Right-Root):
   - Recursive source code:

Task 5: Trees and Graphs

```
def inorder(node):
    if node:
        inorder(node.left)
        print(node.value, end=' ')
        inorder(node.right)
```

- Iterative source code:

```
def iterative_inorder(root):
    stack = []
    current = root
    while stack or current:
        while current:
            stack.append(current)
            current = current.left
        current = stack.pop()
        print(current.value, end=' ')
        current = current.right
```

Output: D B E A C F

3) Post-order(Left-Right-Root):

- Recursive source code:

```
def inorder(node):
    if node:
        inorder(node.left)
        print(node.value, end=' ')
        inorder(node.right)
```

- Iterative source code:

```
def iterative_postorder(root):
    if not root:
        return
    stack1 = [root]
    stack2 = []
    while stack1:
        node = stack1.pop()
```

Task 5: Trees and Graphs

```
            stack2.append(node)
            if node.left:
                stack1.append(node.left)
        if node.right:
            stack1.append(node.right)
    while stack2:
        print(stack2.pop().value, end=' ')
```

Output: D E B F C A

BFS (Level-by-Level or Level-Order Search)

Source code:

```
def bfs(root):
    if not root:
        return
    queue = deque([root])
    while queue:
        node = queue.popleft()
        print(node.value, end=' ')
        if node.left:
            queue.append(node.left)
        if node.right:
            queue.append(node.right)
```

Output: A B C D E F

Graph Traversal

1) Node class creation

```
class Graph:
    def __init__(self, directed=False):
```

Task 5: Trees and Graphs

```
        self.graph = defaultdict(list)
        self.directed = directed
```

2) *To add edge*

```
def add_edge(self, u, v):
        self.graph[u].append(v)
        if not self.directed:
            self.graph[v].append(u)
```

3) DFS Traversal (Recursive)

```
def dfs_recursive(self, node, visited=None):
        if visited is None:
            visited = set()
        visited.add(node)
        print(node, end=' ')
        for neighbor in self.graph[node]:
            if neighbor not in visited:
                self.dfs_recursive(neighbor, visited)
```

Output: A B D E C

4) DFS Traversal (Iterative)

```
def dfs_iterative(self, start):
        visited = set()
        stack = [start]
        while stack:
            node = stack.pop()
            if node not in visited:
                print(node, end=' ')
                visited.add(node)
                for neighbor in reversed(self.graph[node]):
                    if neighbor not in visited:
                        stack.append(neighbor)
```

Output: A C D E B

5) BFS

```
def bfs(self, start):
        visited = set()
```

Task 5: Trees and Graphs

```
        queue = deque([start])
        visited.add(start)
        while queue:
            node = queue.popleft()
            print(node, end=' ')
            for neighbor in self.graph[node]:
                if neighbor not in visited:
                    visited.add(neighbor)
                    queue.append(neighbor)
```

Output: A B C D E

6) Cycle detection

```
def detect_cycle_dfs(self):
    visited = set()
    pathVisited = set()

    def dfs(node):
        visited.add(node)
        pathVisited.add(node)

        for neighbor in self.graph[node]:
            if neighbor not in visited:
                if dfs(neighbor):
                    return True
            elif neighbor in pathVisited:
                return True

        pathVisited.remove(node)
        return False

    for node in list(self.graph):
        if node not in visited:
            if dfs(node):
                return True
    return False
```

Output: False

Task 5: Trees and Graphs

7) Topological sort with Kahn's

```
def topological_sort_kahn(self):
        indegree = defaultdict(int)
        for u in self.graph:
            for v in self.graph[u]:
                indegree[v] += 1

        queue = deque([node for node in self.graph if indegree[node]
== 0])
        topo_order = []

        while queue:
            node = queue.popleft()
            topo_order.append(node)
            for neighbor in self.graph[node]:
                indegree[neighbor] -= 1
                if indegree[neighbor] == 0:
                    queue.append(neighbor)

        if len(topo_order) != len(self.graph):
            print("Cycle detected! No topological ordering
possible.")
        else:
            print("Topological Order (Kahn's):", topo_order)
```

Output: A B C D E

8) Topological sort with DFS

```
def topological_sort_dfs(self):
        visited = set()
        stack = []

        def dfs(node):
            visited.add(node)
            for neighbor in self.graph[node]:
                if neighbor not in visited:
                    dfs(neighbor)
            stack.append(node)

        for node in self.graph:
```

Task 5: Trees and Graphs

```
            if node not in visited:
                dfs(node)
        print("Topological Order (DFS):", stack[::-1])
```

Output: A C B D E

Creation of the graph

```
g = Graph(directed=True)
g.add_edge('A', 'B')
g.add_edge('A', 'C')
g.add_edge('B', 'D')
g.add_edge('C', 'D')
g.add_edge('D', 'E')
```

Function calls

```
print("DFS Recursive:")
g.dfs_recursive('A')
print("\nDFS Iterative:")
g.dfs_iterative('A')
print("\nBFS:")
g.bfs('A')

print("\nCycle Detected:" if g.detect_cycle_dfs() else "\nNo Cycle
Detected")

print("\nTopological Sort (Kahn):")
g.topological_sort_kahn()
print("\nTopological Sort (DFS):")
g.topological_sort_dfs()
```

Tree Algorithms

1) Lowest Common Ancestor

Definition of BST Node

Task 5: Trees and Graphs

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
```

Explanation:
LCA of two nodes p and q is their common ancestor at the lowest possible level. A node can also be its own ancestor. Since we are working with a BST here (values on right are greater, those on left are smaller), this code moves rightwards if both p and q have a value more than the cur node (iterator node), left if lower. In case of a split (such as p less and q great) or both values being equal, we break the loop since we found the ancestor at the lowest possible level from both p and q and also due to the possibility that a node can be its own ancestor.

Source code:

```
def lowestCommonAncestor(self, root: TreeNode, p: TreeNode, q:
TreeNode) -> TreeNode:
    cur = root
    while cur:
        if p.val > cur.val and q.val > cur.val:
            cur = cur.right
        elif p.val < cur.val and q.val < cur.val:
            cur = cur.left
        else:
            return cur
```

2) *Diameter of Tree*

Explanation:
The diameter of a binary tree is the length of the longest possible path that can be traced in the tree. It may or may not pass through the root. To find this we recursively call a dfs function that returns the length of the path obtained in each

call, and then return the maximum possible length, either on the left or the right side.

```
def diameterOfBinaryTree(self, root: Optional[TreeNode]) -> int:
        res = 0

        def dfs(root):
            nonlocal res

            if not root:
                return 0
            left = dfs(root.left)
            right = dfs(root.right)
            res = max(res, left + right)

            return 1 + max(left, right)

        dfs(root)
        return res
```

3) Invert Tree

Explanation:
Since the tree only needs to be laterally inverted, all we need to do is change the values in the nodes, not the actual nodes themselves. We could do that but the result would be the same and that would just take more time.

```
def invertTree(self, root: Optional[TreeNode]) ->
Optional[TreeNode]:
        if not root: return None

        root.left, root.right = root.right, root.left

        self.invertTree(root.left)
        self.invertTree(root.right)

        return root
```

Task 5: Trees and Graphs

4) Max. Depth

Explanation:
Since left and right subtrees can be of different depths, we search each side of the tree to find the depth, and return the maximum one found.

```
def maxDepth(self, root: Optional[TreeNode]) -> int:
    if not root:
        return 0

    return 1 + max(self.maxDepth(root.left),
self.maxDepth(root.right))
```

5) Balanced Tree

Explanation:
A binary tree is said to be balanced if the height of any of the left and right subtrees contained in it doesn't differ by more than | 1 |. It's similar to the condition all AVL trees follow.

Here we recursively calculate depth using dfs method, and check for left and right differences in each call. If false at any one, the control exits.

```
def isBalanced(self, root: Optional[TreeNode]) -> bool:
        def dfs(root):
            if not root:
                return [True, 0]

            left, right = dfs(root.left), dfs(root.right)
            balanced = (left[0] and right[0] and abs(left[1] -
right[1]) <= 1)

            return [balanced, 1 + max(left[1], right[1])]

        return dfs(root)[0]
```

Task 5: Trees and Graphs

6) Serialize / Deserialize Binary Tree
Explanation:

Serialization:
We perform DFS on the tree, and at each place if the node exists we append its value to the resultant array, else we just add an “N”, indicating that no value was found.

Deserialization:
We scan the array, and then run a DFS function. At each point where there’s an “N”, we exit that call and move to the next. Else we simply create a new node, and call the DFS function on its left and right connections.

```
def serialize(self, root: Optional[TreeNode]) -> str:
        res = []

        def dfs(node):
            if not node:
                res.append("N")
                return
            res.append(str(node.val))
            dfs(node.left)
            dfs(node.right)

        dfs(root)
        print(res)
        return ",".join(res)

def deserialize(self, data: str) -> Optional[TreeNode]:
        vals = data.split(",")
        self.i = 0

        def dfs():
            if vals[self.i] == "N":
                self.i += 1
```

Task 5: Trees and Graphs

```
                return None
            node = TreeNode(int(vals[self.i]))
            self.i += 1
            node.left = dfs()
            node.right = dfs()
            return node

        return dfs()
```

Graph Algorithms

```
import heapq
from collections import defaultdict

# ---------- Graph Class ----------
class Graph:
    def __init__(self, directed=False):
        self.graph = defaultdict(list)
        self.directed = directed

    def add_edge(self, u, v, w=1):
        self.graph[u].append((v, w))
        if not self.directed:
            self.graph[v].append((u, w))

    def nodes(self):
        return list(set(self.graph.keys()) | {v for edges in
self.graph.values() for v, _ in edges})
```

1) *Djikstra's Algorithm:*
   Initialize a table with distances to all nodes set as infinity and the starting node distance as 0.
   Then explore each of the starting node's connections, and update the total distance value it takes to reach those nodes

Task 5: Trees and Graphs

from the starting node iff its lesser than what is there in the table.
Then move to the connected node for which the obtained value is least in each step.
Finally, the table with optimal values will be obtained.

```
def dijkstra(graph, start):
    dist = {node: float('inf') for node in graph}
    dist[start] = 0

    visited = set()

    while len(visited) < len(graph):
        curr = None
        min_dist = float('inf')
        for node in graph:
            if node not in visited and dist[node] < min_dist:
                min_dist = dist[node]
                curr = node

        if curr is None:
            break
        visited.add(curr)

        for neighbor, weight in graph[curr]:
            if neighbor not in visited:
                new_dist = dist[curr] + weight
                if new_dist < dist[neighbor]:
                    dist[neighbor] = new_dist

    return dist
```

2) Floyd Warshall
   Explanation:
   Visit every node one by one and then each one of its connections. Analyze the distances to each of the other nodes

from there. Update the table if the distance obtained turns out to be shorter.

```
def floyd_warshall(graph):

    nodes = list(graph.keys())

    dist = {u: {v: float('inf') for v in nodes} for u in
nodes}

    for u in nodes:
        dist[u][u] = 0
        for v, w in graph[u]:
            dist[u][v] = w

    for k in nodes:
        for i in nodes:
            for j in nodes:
                if dist[i][k] + dist[k][j] < dist[i][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]

    return dist
```

3) Bellman Ford

Explanation:
Initialize a table with distances to all nodes set as infinity and the starting node distance as 0.
Then, for V - 1 times, go through all the edges and update the distance to the destination node iff the new total distance is lesser than the one currently in the table.
This simulates exploring all possible paths by relaxing the edges repeatedly.
Finally, check once more to detect if any edge can still reduce a value — if so, a negative weight cycle exists.

Task 5: Trees and Graphs

The final table will contain the optimal distances from the start node to all others.

```
def bellman_ford(graph, start):

    dist = {node: float('inf') for node in graph}
    dist[start] = 0

    for _ in range(len(graph) - 1):
        for u in graph:
            for v, w in graph[u]:
                if dist[u] + w < dist[v]:
                    dist[v] = dist[u] + w

    for u in graph:
        for v, w in graph[u]:
            if dist[u] + w < dist[v]:
                raise ValueError("Graph contains a negative
weight cycle")

    return dist
```

4) Prim's Algorithm

Explanation:

Initialize a table with all node costs set to infinity, and the starting node's cost as 0.

At each step, pick the node with the lowest cost that hasn't been visited yet.

Then update the costs of all its neighbors, but only if the edge weight is smaller than what's already in the table.

Repeat this process until all nodes are included.

The final table will show the minimum edge cost used to bring each node into the spanning tree, and the sum of these values is the minimum total cost.

```
def bellman_ford(graph, start):
```

Task 5: Trees and Graphs

```
    dist = {node: float('inf') for node in graph}
    dist[start] = 0

    for _ in range(len(graph) - 1):
        for u in graph:
            for v, w in graph[u]:
                if dist[u] + w < dist[v]:
                    dist[v] = dist[u] + w

    for u in graph:
        for v, w in graph[u]:
            if dist[u] + w < dist[v]:
                raise ValueError("Graph contains a negative
weight cycle")

    return dist
```

5) Kruskal's Algorithm

Explanation:

First, make a list of all edges in the graph along with their weights. To avoid duplicates (since the graph is undirected), store only one copy of each edge by sorting its endpoints.
Then, sort all the edges in ascending order of their weights using a helper function.
Start picking edges one by one from this sorted list. Since we assume the graph has no cycles,we just pick the smallest edges.
Keep adding edges until you've selected (V - 1) of them, where V is the number of nodes.
These edges form the minimum spanning tree (MST), and their total weight is the answer.

Task 5: Trees and Graphs

```python
def kruskal_no_cycle(graph):
    edges = []
    seen = set()
    for u in graph:
        for v, w in graph[u]:
            edge = tuple(sorted((u, v)) + [w])
            if edge not in seen:
                seen.add(edge)
                edges.append((u, v, w))


    def get_weight(edge):
        return edge[2]
    edges.sort(key=get_weight)
    mst_cost = 0
    mst_edges = []
    for u, v, w in edges:
        mst_cost += w
        mst_edges.append((u, v))
        if len(mst_edges) == len(graph) - 1:
            break
    return mst_cost, mst_edges
```

Task 5: Trees and Graphs

6) Union Find / Disjoint Set

Explanation:

Create a structure where each node initially belongs to its own separate set.

To check if two nodes are in the same set, use the find( ) function which recursively goes up to the parent of each node until the root is found.

To merge two different sets, use the union( ) function. This joins the root of one set to the root of the other.

We also apply path compression in find( ) so that future lookups are faster. Every time we find the root of a node, we update its direct parent to that root.

This structure is mainly used in Kruskal's Algorithm to detect if adding an edge will form a cycle. If two nodes are already in the same set, adding an edge between them creates a cycle, so we skip it.

```
class UnionFind:

    def __init__(self, nodes):

        self.parent = {node: node for node in nodes}



    def find(self, x):

        if self.parent[x] != x:

            self.parent[x] = self.find(self.parent[x])

        return self.parent[x]
```

Task 5: Trees and Graphs

```
def union(self, x, y):
    root_x = self.find(x)
    root_y = self.find(y)
    if root_x == root_y:
        return False
    self.parent[root_y] = root_x
    return True
```

7) Tarjan's Algorithm
   Explanation:
   An SCC (Strongly Connected Component) is a group of nodes in a directed graph such that every node is reachable from every other node in that group. Tarjan's algorithm helps identify these groups efficiently.

   We use DFS to visit each node. For every node, we assign:

   - An index (when it was discovered),
   - A lowlink (the lowest index reachable from this node).

   We push nodes onto a stack during traversal. If we come back to a node that's already in the stack, we update the lowlink of the current node to reflect that a cycle might be forming.

   When a node's lowlink equals its index, we've found the start of a strongly connected component — pop all nodes from the

stack up to that point.
Repeat this for every unvisited node. The result is a list of all SCCs.

```
ef tarjans_scc(graph):
    index = 0
    index_map = {}
    lowlink = {}
    stack = []
    on_stack = set()
    sccs = []

    def dfs(node):
        nonlocal index
        index_map[node] = lowlink[node] = index
        index += 1
        stack.append(node)
        on_stack.add(node)

        for neighbor, _ in graph.get(node, []):
            if neighbor not in index_map:
                dfs(neighbor)
                lowlink[node] = min(lowlink[node],
lowlink[neighbor])
            elif neighbor in on_stack:
                lowlink[node] = min(lowlink[node],
index_map[neighbor])

        if lowlink[node] == index_map[node]:
            component = []
            while True:
                w = stack.pop()
                on_stack.remove(w)
                component.append(w)
                if w == node:
                    break
            sccs.append(component)
```

```
    for node in graph:
        if node not in index_map:
            dfs(node)

    return sccs
```

8) Kosaraju's Algorithm:

Kosaraju's algorithm finds all SCCs in three steps:

1. Run DFS on the original graph and store the finishing order of each node.
2. Reverse all edges in the graph (i.e. flip direction of every arrow).
3. Run DFS again, this time in the reverse of the finishing order from step 1.
4. Each DFS now gives one strongly connected component.

It works because once edges are reversed, the components become isolated, and we can discover them cleanly.

```
def kosaraju_scc(graph):

    visited = set()

    finish_order = []



    def dfs1(node):

        visited.add(node)

        for neighbor, _ in graph.get(node, []):

            if neighbor not in visited:

                dfs1(neighbor)
```

Task 5: Trees and Graphs

```
        finish_order.append(node)


    for node in graph:
        if node not in visited:
            dfs1(node)


    reversed_graph = {node: [] for node in graph}
    for u in graph:
        for v, w in graph[u]:
            reversed_graph[v].append((u, w))
    visited.clear()
    sccs = []
    def dfs2(node, component):
        visited.add(node)
        component.append(node)
        for neighbor, _ in reversed_graph.get(node, []):
            if neighbor not in visited:
                dfs2(neighbor, component)


    for node in reversed(finish_order):
        if node not in visited:
            component = []
```

Task 5: Trees and Graphs

```
            dfs2(node, component)
            sccs.append(component)
    return sccs
```

9) A* Algorithm

Explanation:

A* (A-star) is a shortest path algorithm used to find the least-cost path from a start node to a goal node

Start by setting the cost of the start node as 0.
At each step, pick the node with the lowest total estimated cost (actual distance so far + heuristic estimate to the goal).
Then explore its neighbors and update their costs if a shorter path is found.
Repeat this until the goal node is reached.

The final path will be the optimal one, guided by both actual cost and guesses.

```
mport heapq


def a_star_simple(graph, start, goal, heuristic):
    open_set = [(0 + heuristic(start), 0, start)]
    came_from = {}
    g_score = {node: float('inf') for node in graph}
    g_score[start] = 0
```

Task 5: Trees and Graphs

```
    while open_set:

        _, current_g, current = heapq.heappop(open_set)


        if current == goal:

            path = []

            while current in came_from:

                path.append(current)

                current = came_from[current]

            path.append(start)

            return list(reversed(path))


        for neighbor, weight in graph[current]:

            tentative_g = current_g + weight

            if tentative_g < g_score[neighbor]:

                g_score[neighbor] = tentative_g

                f_score = tentative_g + heuristic(neighbor)

                came_from[neighbor] = current

                heapq.heappush(open_set, (f_score,
tentative_g, neighbor))


    return None
```

Task 5: Trees and Graphs

Sources:

1) TakeUForward on YouTube
2) Neetcode on YouTube
3) Abdul Bari on YouTube