DFS VS BFS

Ying Qiao

KEY CONCEPTS

- Tree & Graph
- DFS vs BFS
- Stack vs Queue

TREE & GRAPH TRAVERSAL

- Linear data structures have only one logical way to traverse
- Trees & graph can be traversed in different ways
- By the order in which the nodes are visited:
  - depth-first search
  - breadth-first search

TREE TRAVERSAL

- Depth First Traversals:

  (a) Preorder (Root, Left, Right) : 1 2 4 5 3

  (b) Inorder (Left, Root, Right) : 4 2 5 1 3

  (c) Postorder (Left, Right, Root) : 4 5 2 3 1

TREE TRAVERSAL

- Depth First Traversals:

  (a) Preorder (Root, Left, Right) : 1 2 4 5 3

  (b) Inorder (Left, Root, Right) : 4 2 5 1 3

  (c) Postorder (Left, Right, Root) : 4 5 2 3 1

TREE TRAVERSAL

```
def printPreorder(root):

    if root:

        # First print the data of node
        print(root.val)

        # Then recur on left child
        printPreorder(root.left)

        # Finally recur on right child
        printPreorder(root.right)
```

Preorder (Root, Left, Right) : 1 2 4 5 3

https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/

TREE TRAVERSAL

```
def printInorder(root):

    if root:

        # First recur on left child
        printInorder(root.left)

        # then print the data of node
        print(root.val)

        # now recur on right child
        printInorder(root.right)
```

Inorder (Left, Root, Right) : 4 2 5 1 3

https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/

TREE TRAVERSAL

```
def printPostorder(root):

    if root:

        # First recur on left child
        printPostorder(root.left)

        # the recur on right child
        printPostorder(root.right)

        # now print the data of node
        print(root.val)
```

Postorder (Left, Right, Root) : 4 5 2 3 1

https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/

TREE TRAVERSAL

- Breadth First Traversal:

  Level Order: 1 2 3 4 5

TREE TRAVERSAL

```
def printLevelOrder(root):
    if root is None:
        return
    queue = deque([])
    queue.append(root)
    while(len(queue) > 0):
        print (queue[0].data)
        node = queue.popleft()
        if node.left:
            queue.append(node.left)
        if node.right:
            queue.append(node.right)
```

TREE TRAVERSAL

- Cover most of binary tree related problems
- DFS
  - Related to recursion/divide and conquer, can be implemented with stack
  - Information from parent or children
  - Backtrack, Inorder traversal of BST
- BFS
  - Level/height related problems
- Can easily extend to multinode tree

GRAPH TRAVERSAL

- G = {V, E}
- V : set of vertices/nodes, no root
- E : set of edges
- Can have cycles
- Can be undirected/directed, weighted etc.

DFS FOR GRAPH

```
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
    print(start) # 01243
    for next in graph[start]:
        if next not in visited:
            dfs(graph, next, visited)

dfs(graph, '0')
```

```
graph = {'0': set(['1', ‘2’, ‘3’]),
         '1': set([‘0’,‘2']),
         '2': set(['0’,’1’,’4’]),
         '3': set([‘0']),
         '4': set(['2'])}
```

BFS FOR GRAPH

```
visited = [ ]
queue = deque([])
def bfs(graph, node, visited):
    visited.append(node)
    queue.append(node)
    while queue:
        s = queue.popleft()
        print (s)  # 01234
        for neighbour in graph[s]:
            if neighbour not in visited:
                visited.append(neighbour)
                queue.append(neighbour)
bfs(graph, '0', visited)
```

```
graph = {'0': set(['1', ‘2’, ‘3’]),
         '1': set([‘0’,‘2']),
         '2': set(['0’,’1’,’4’]),
         '3': set([‘0']),
         '4': set(['2'])}
```

DFS OR BFS ?

- Is a question ....
- both used to traverse graphs
- The time complexity of DFS and BFS:
  - $O(V + E)$ when Adjacency List is used
  - $O(V^2)$ when Adjacency Matrix is used
- DFS is more memory efficient than BFS
  - Binary tree: Stack: $O(h)$ (= $O(\log n)$) Queue: $O(n)$
  - DFS can backtrack sooner, but may cause stack overflow

DFS OR BFS ?

BASIS FOR COMPARISON	BFS	DFS
Basic	Vertex-based algorithm	Edge-based algorithm
Data structure used to store the nodes	Queue	Stack
Memory consumption	Inefficient	Efficient
source	Better when target is closer to source	Better when target is far from source
Traversing fashion	Oldest unvisited vertices are explored at first.	Vertices along the edge are explored in the beginning.
Optimality	Optimal for finding the shortest distance	More suitable for decision tree
Application	Examines bipartite graph, connected component and shortest path present in a graph.	Examines two-edge connected graph, strongly connected graph, acyclic graph

104. MAXIMUM DEPTH OF BINARY TREE

- Given the root of a binary tree, return *its maximum depth*.
- A binary tree's **maximum depth** is the number of nodes along the longest path from the root node down to the farthest leaf node.

```
Input: root = [3,9,20,null,null,15,7]
Output: 3
```

DFS

- Postorder traversal

```
class Solution:
    def maxDepth(self, root: TreeNode):
        if not root:
            return 0
        left = self.maxDepth(root.left)
        right = self.maxDepth(root.right)
        depth = max(left, right) +1
        return depth
```

111. MINIMUM DEPTH OF BINARY TREE

- Given a binary tree, find its *minimum depth*.
- The **minimum depth** is the number of nodes along the shortest path from the root node down to the nearest leaf node.

```
Input: root = [3,9,20,null,null,15,7]
Output: 2
```

BFS

```
class Solution:
    def minDepth(self, root: TreeNode) -> int:
        if not root:
            return 0
        queue = deque([(root,1)])

        while queue:
            node, depth = queue.popleft()
            if node.left:
                queue.append((node.left, depth+1))
            if node.right:
                queue.append((node.right,depth+1))
            if not node.left and not node.right:
                return depth
```

323. NUMBER OF CONNECTED COMPONENTS IN AN UNDIRECTED GRAPH

- Given n nodes labeled from 0 to n - 1 and a list of undirected edges (each edge is a pair of nodes), write a function to find the number of connected components in an undirected graph.
- Subgraph in which any two vertices are connected with each other via a path

```
Input: n = 5 and edges = [[0, 1], [1, 2], [2, 3], [3,
4]]

     0           4
     |           |
     1 --- 2 --- 3

Output:  1
```

```
Input: n = 5 and edges = [[0, 1], [1, 2], [3, 4]]

     0           3
     |           |
     1 --- 2     4

Output: 2
```

DFS

```
def countComponents(n, edges):
    g = {i:[] for i in range(n)}
    for u,v in edges:
        g[u].append(v)
        g[v].append(u)
    visited = []
    nums = 0
     for i in range(n):
        if i not in visited:
            dfs(i)
            nums += 1
    return nums
```

```
def dfs(node):
    if node in visited:
        return
    visited.append(node)
    for neighbor in g[node]:
        dfs(neighbor)
```

BFS

```
def countComponents(n, edges):
    queue = deque([])
    for i in range(n):
        if i not in visited:
            queue.append(i)
            while queue:
                node = queue.popleft()
                visited.append(node)
                for neighbor in g[node]:
                    if neighbor not in visited:
                        queue.append(neighbor)
            nums += 1
    return nums
```

NOTES

- Tricky Questions
  - Permutation/combination (DFS), shortest path/distance (BFS)
  - Realize it is a graph/tree searching problem (decision tree, minimum steps, etc.)
- Classic Graph Algorithms
  - Prim's Algorithm for minimum spanning tree (subset of edges connecting all vertices)
  - Dijkstra's Algorithm for Shortest Paths (weighted graph)
  - Floyd's all-pairs shortest-path algorithm
- Practice Questions:
  - Leetcode 98, 199, 130, 200