

AI Lab 2

Aim: To implement Breadth First Search

Tools used: Python

Theory:

In **Breadth First Search**, expansion is proceeding by level by level. Initially in first step, it expands all the node of first level and then explore the nodes of second level and so on till the goal node is not found. It works on the principle of First In First Out (FIFO) on a queue arrangement. It is not used unless there is high memory demand [3].

About Breadth First Search:

- Find the solution of problem which consist of less curves.
- Find the best shortest path from many solutions.
- No constraint of Memory.
- Minimal spanning tree and shortest path in unweighted table. Shared network.
- GPS Navigation systems.
- To evaluate how this graph is inclusive or not.

Algorithm

Step 1: SET STATUS = 1 (ready state) for each node in G

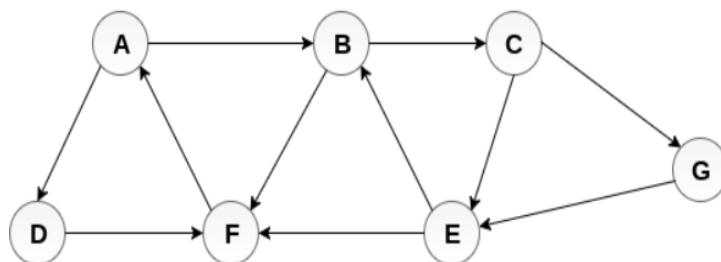
Step 2: Enqueue the starting node A and set its STATUS = 2(waiting state)

Step 3: Repeat Steps 4 and 5 until QUEUE is empty

Step 4: Dequeue a node N. Process it and set its STATUS = 3 (processed state).

Step 5: Enqueue all the neighbours of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state) [END OF LOOP]

Step 6: EXIT.



Adjacency Lists

A : B, D

B : C, F

C : E, G

G : E

E : B, F

F : A

D : F

Solution: A>B>C>E

Pseudocode for BFS

```
BFS(v){
    {add v to queue and mark it}
    Add(Q, v)
    Mark v as visited
    while (not IsEmpty(Q)) do
        begin
            w = QueueFront(Q)
            Remove(Q)
            {loop invariant : there is a path from vertex w to every vertex in the queue Q}
            for each unvisited vertex u adjacent to w do
                begin
                    Mark u as visited
                    Add(Q , u)
                end { for }
            end{ while
        }
    }
```

Code (BFS):

```
class Queue(object):
    def __init__(self):
        self.items = []

    def enqueue(self, item):
        self.items.insert(0, item)

    def dequeue(self):
        if not self.is_empty():
            return self.items.pop()

    def is_empty(self):
        return len(self.items) == 0

    def peek(self):
        if not self.is_empty():
            return self.items[-1].value

    def __len__(self):
        return self.size()

    def size(self):
        return len(self.items)

class Node(object):
    def __init__(self, value):
```

```

        self.value = value
        self.left = None
        self.right = None

class BinaryTree(object):
    def __init__(self, root):
        self.root = Node(root)

    def print_tree(self, traversal_type):
        if traversal_type == "bfs":
            return self.bfs(tree.root)

        else:
            print("Traversal type " + str(traversal_type) + " is not supported.")
            return False

    def bfs(self, start):
        if start is None:
            return

        queue = Queue()
        queue.enqueue(start)

        traversal = ""
        while len(queue) > 0:
            traversal += str(queue.peek()) + "-"
            node = queue.dequeue()

            if node.left:
                queue.enqueue(node.left)
            if node.right:
                queue.enqueue(node.right)

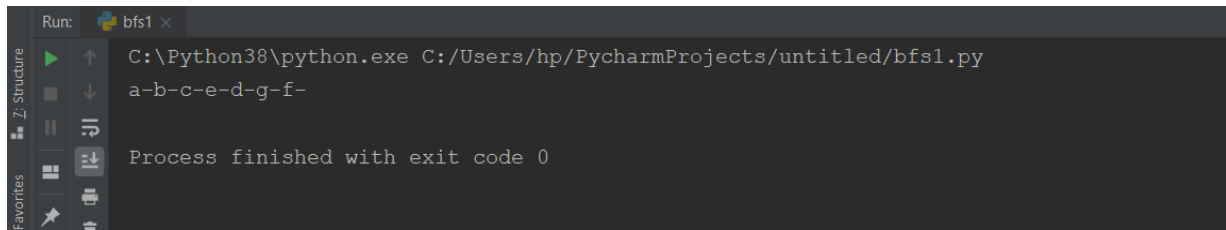
        return traversal

tree = BinaryTree('a')
tree.root.left = Node('b')
tree.root.right = Node('c')
tree.root.left.left = Node('e')
tree.root.left.right = Node('d')
tree.root.right.left = Node('g')
tree.root.right.right = Node('f')

print(tree.print_tree("bfs"))

```

Output:



```
Run: bfs1 ×  
C:\Python38\python.exe C:/Users/hp/PycharmProjects/untitled/bfs1.py  
a-b-c-e-d-g-f-  
Process finished with exit code 0
```

Conclusion:

In this lab we learnt how to implement Breadth First Search in Python using a binary tree structure and queue data structure. This method is much efficient than the Depth first Search and helps in finding the element in a quicker way. This algorithm is widely deployed for various applications such as GPS and others which involve finding the shortest path.