Jay Menon

H041

AI Lab 1

Aim: To Implement Depth First Search (Un Informed Search technique).

Tools used: Python

Theory:

Search algorithms is one of the important part of Artificial Intelligence. The search techniques are the first step to start with AI problem and based on these techniques, AI system with work in the subsequent steps. These techniques will help to find the path to reach the goal. Using search techniques, many computer sciences practical problem of various domain like robotics, database system, an expert system can be solved. Search techniques are used to overcome particular problems. They are categorized in two type i.e.

- Un-informed search algorithm – In this search algorithm, no information is available other than their interpretation of the problem. Using this algorithm, problem can be solved but none can solve the problem effectively. Uninformed search is absolutely deaf search and the methods are Breadth First Search, Depth First Search, Iterative Deepening etc. In this search technique, there is not idea about number step to reach the goal state from the initial state. These techniques are time consuming and not able to solve the problem in large environment.

- Informed search algorithm – In this search algorithm, information is available other than their interpretation of the problem. This algorithm can do very well. Heuristic search is also called as Informed search, which uses heuristic function to solve a problem (i.e. estimate the steps involved to reach the goal state from present state). It uses of knowledgeable search produce method and check. In this search, various algorithm is involved like Hill climbing, A*, AO*, steepest climbing, Best First Search. Informed searches are more successful and efficient than uninformed searches. Heuristic role in informed search is used as a guide that will take us to target state. The Informed search will not automatically explore the search tree. [2].

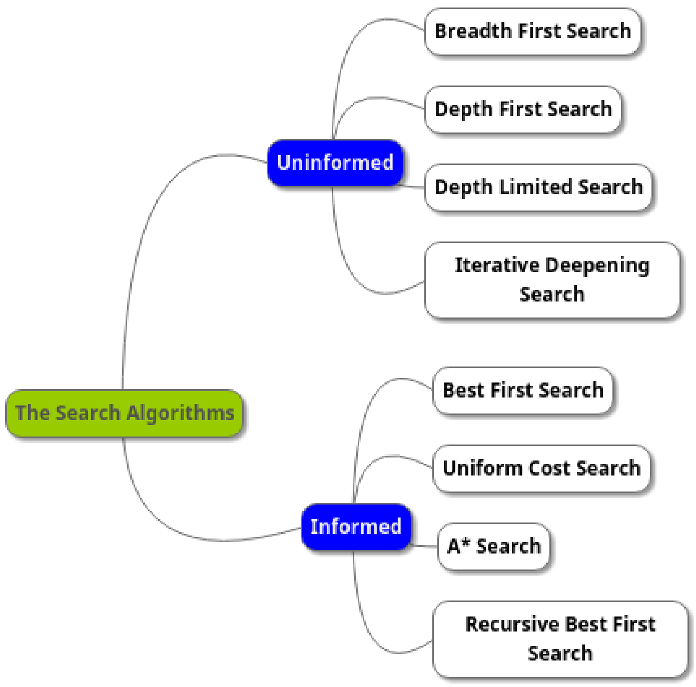


Fig.1 Type of search algorithms

In **Depth First Search**, expansion begins at the root node and moves up to the highest unexpanded node in a target node quest. It works on the principle of Last In First Out (LIFO). It is also known as recursive algorithm since it works as stack concept [2].

About Depth First Search:

- In this, limitation of Memory.
- Add the order of the node's neighbours to the stack such that solutions can be sought at the first attempt.
- It's a not a good strategy because collecting in infinite forms becomes possible, this occurs when the graph becomes limitless, or where loops are in the graph.
- This algorithm will identify loops in graph and various components which are connected strongly in graph.
- Seeking a mystery solution, in which there is just one answer.

Algorithm

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

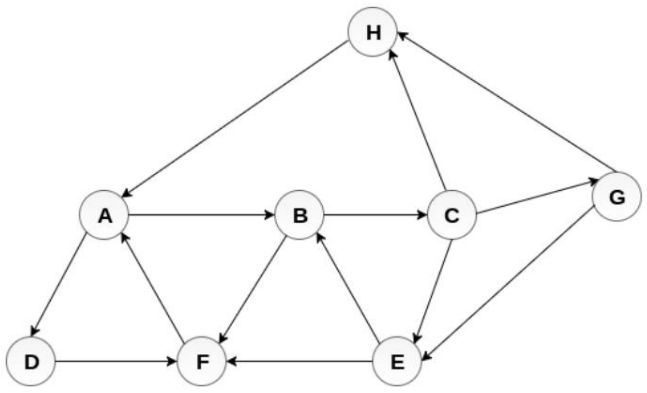
Step 2: Push the starting node A on the stack and set its STATUS = 2 (waiting state)

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

Step 4: Pop the top node N. Process it and set its STATUS = 3 (processed state)

Step 5: Push on the stack 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, H
G : E, H
E : B, F
F : A
D : F
H : A

Solution: H>A>D>>F>B>C>G>E

Pseudocode for DFS

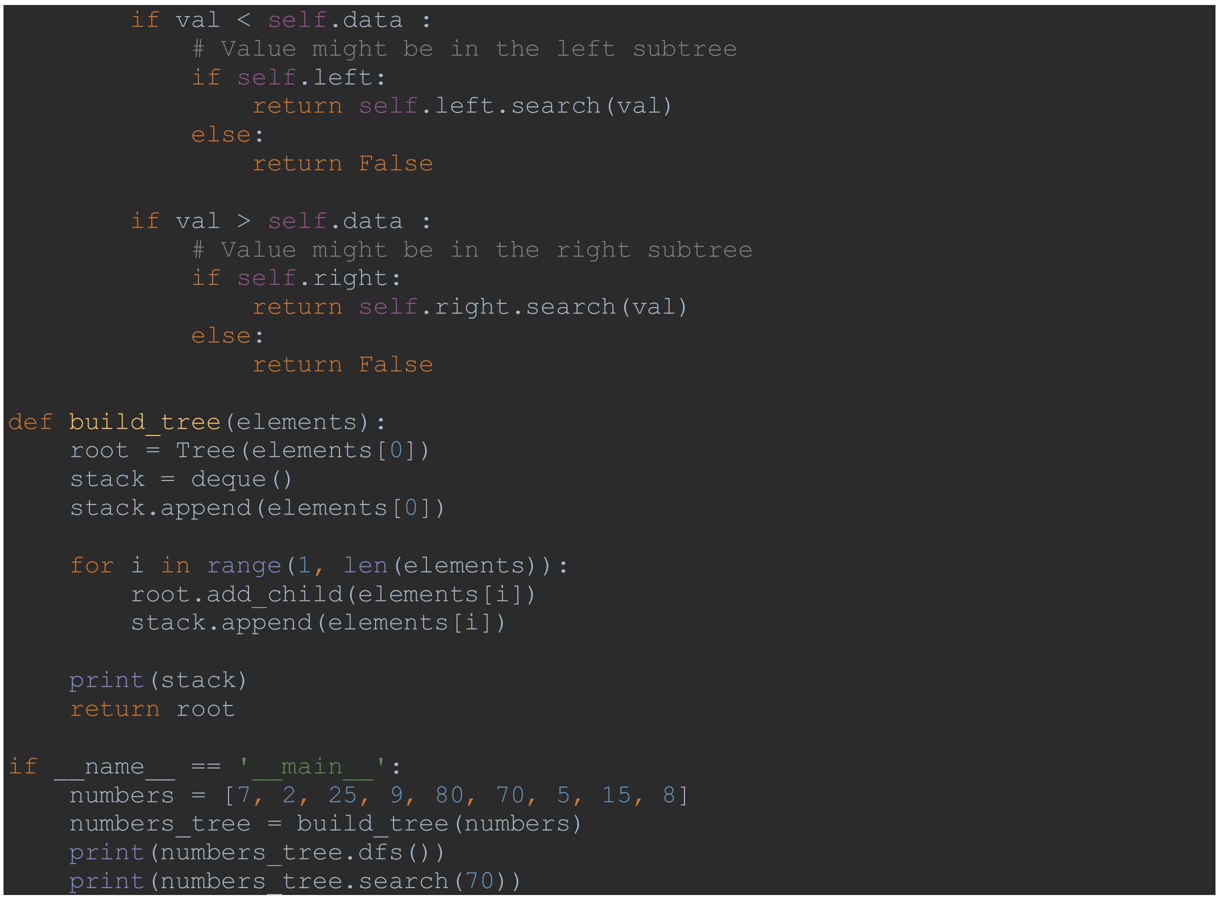
```
Initialize an empty stack for storage of nodes, S.
For each vertex u, define u.visited to be false.
Push the root (first node to be visited) onto S.
While S is not empty:
    Pop the first element in S, u.
    If u.visited = false, then:
        U.visited = true
        for each unvisited neighbor w of u:
            Push w into S.
End process when all nodes have been visited.
```

Python Implementation without Recursion (DFS)

```python
def depth_first_search(graph):
    visited, stack = set(), [root]
    while stack:
        vertex = stack.pop()
        if vertex not in visited:
            visited.add(vertex)
            stack.extend(graph[vertex] - visited)
    return visited
```

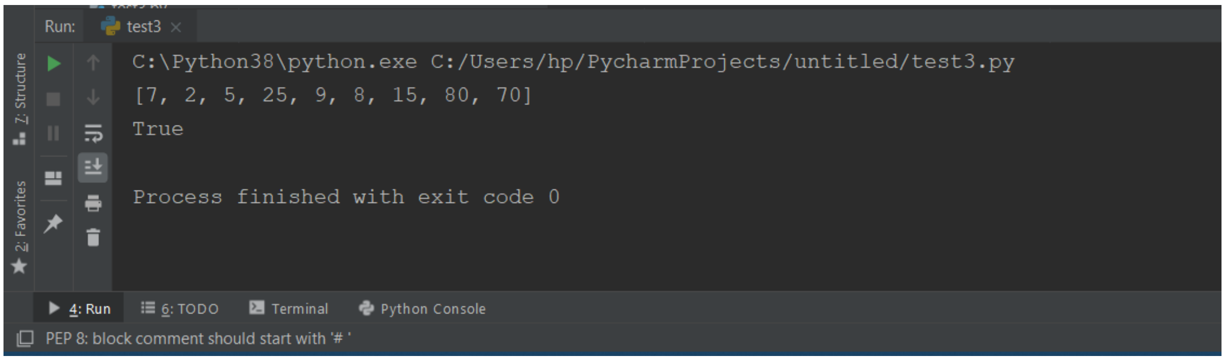
Code (DFS) :

```python
from collections import deque

class Tree:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

    def push(self, val):
        self.container.append(val)

    def pop(self):
        return self.container.pop()

    def peek(self):
        return self.container[-1]

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

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

    def add_child(self, data):
        if data == self.data:
            print("Found")
            return

        if data < self.data:
            # add data in left subtree
            if self.left:
                self.left.add_child(data)
            else:
                self.left = Tree(data)
        else:
            # add data in right subtree
            if self.right:
                self.right.add_child(data)
            else:
                self.right = Tree(data)

    def dfs(self):
        elements = []
        # visit base node
        # visit left tree
        # visit left child
        # visit left's left child
        elements.append(self.data)

        if self.left:
            elements += self.left.dfs()

        # Visit right tree
        if self.right:
            elements += self.right.dfs()

        return elements

    def search(self, val):
        if self.data == val:
            return True
```



```python
        if val < self.data :
            # Value might be in the left subtree
            if self.left:
                return self.left.search(val)
            else:
                return False

        if val > self.data :
            # Value might be in the right subtree
            if self.right:
                return self.right.search(val)
            else:
                return False

def build_tree(elements):
    root = Tree(elements[0])
    stack = deque()
    stack.append(elements[0])

    for i in range(1, len(elements)):
        root.add_child(elements[i])
        stack.append(elements[i])

    print(stack)
    return root

if __name__ == '__main__':
    numbers = [7, 2, 25, 9, 80, 70, 5, 15, 8]
    numbers_tree = build_tree(numbers)
    print(numbers_tree.dfs())
    print(numbers_tree.search(70))
```

<u>Output</u>



```
C:\Python38\python.exe C:/Users/hp/PycharmProjects/untitled/test3.py
[7, 2, 5, 25, 9, 8, 15, 80, 70]
True

Process finished with exit code 0
```

PEP 8: block comment should start with '# '

<u>Conclusion</u>

In this lab we learnt how to implement Depth First Search in Python using a binary tree structure and stacks. We followed a left-first approach throughout the algorithm and searched for our desired element in the tree.