Department of Information and Communication Engineering
Pabna University of Science and Technology
B.Sc. (Engineering) 3rd Year 1st Semester Examination-2020
Session: 2017-2018 Course Code: ICE-3102
Course Title: Artificial Intelligence and Robotics Sessional

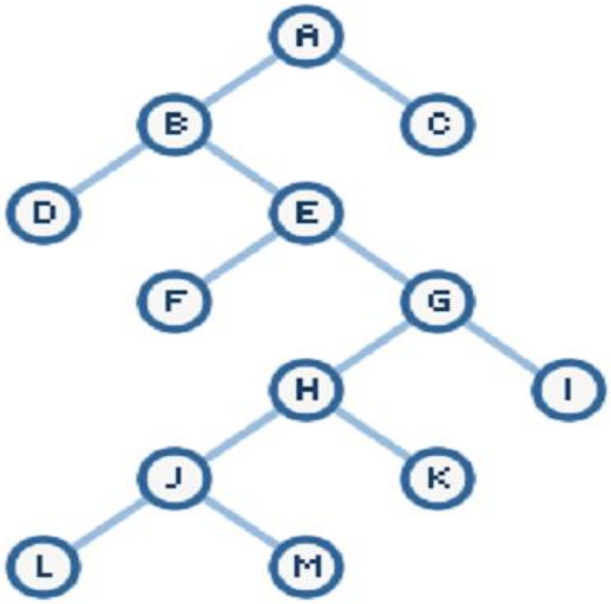
Write a program to implement depth-first search (DFS).

Theory

Depth First Search works by taking a node, checking its neighbors, expanding the first node it finds among the neighbors, checking if that expanded node is our destination, and if not, continue exploring more nodes.

A major goal of AI is to give computers the ability to think, or in other words, mimic human behavior. The problem is, unfortunately, computers don't function in the same way our minds do. They require a series of well-reasoned out steps before finding a solution. Your goal, then, is to take a complicated task and convert it into simpler steps that your computer can handle.

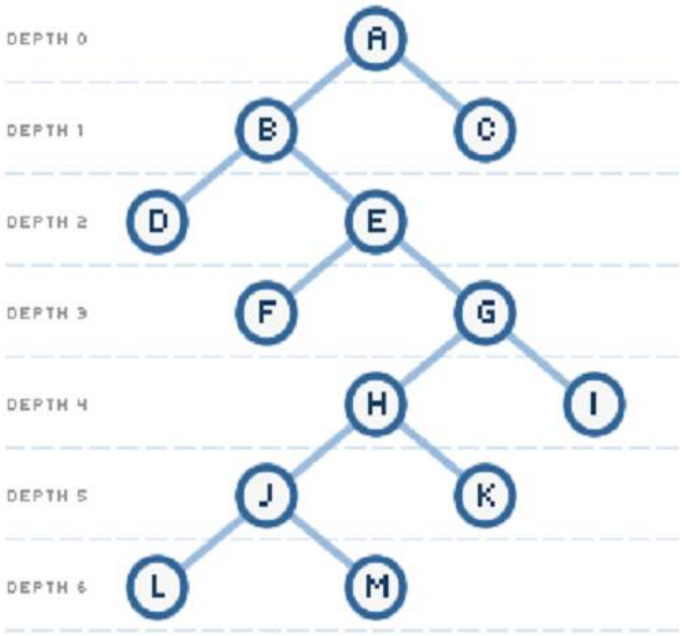
First, we need a representation of how our search problem will exist. The following is an example of our search tree. It is a series of interconnected nodes that we will be searching through:



In our above graph, the path connections are not two-way. All paths go only from top to bottom. In other words, A has a path to B and C, but B and C do not have a path to A. It is basically like a one-way street.

Each lettered circle in our graph is a node. A node can be connected to other via our edge/path, and those nodes that it connects to are called neighbors. B and C are neighbors of A. E and D are neighbors of B, and B is not a neighbor of D or E because B cannot be reached using either D or E.

Our search graph also contains depth:



We now have a way of describing location in our graph. We know how the various nodes (the lettered circles) are related to each other (neighbors), and we have a way of characterizing the depth each belongs in. Knowing this information isn't directly relevant in creating our search algorithm, but they do help us to better understand the problem.

Using depth first search tree, let's find a path between nodes A and F:

Step 0

Let’s start with our root/goal node:

I will be using two lists to keep track of what we are doing - an **Open List** and a **Closed List**. An Open list keeps track of what you need to do, and the Closed List keeps track of what you have already done. Right now, we only have our starting point, node A. We haven't done anything to it yet, so let's add it to our Open list.

- Open List: A
- Closed List: <empty>

Step 1

Now, let's explore the neighbors of our A node. To put another way, let's take the first item from our Open list and explore its neighbors:

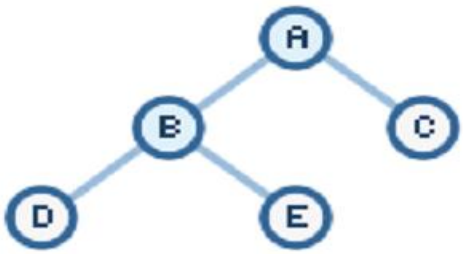
Node A's neighbors are the B and C nodes. Because we are now done with our A node, we can remove it from our Open list and add it to our Closed List. You aren't done with this step though. You now have two new nodes B and C that need exploring. Add those two nodes to our Open list.

Our current Open and Closed Lists contain the following data:

- **Open List:** B, C
- **Closed List:** A

Step 2

Our Open list contains two items. For depth first search and breadth first search, you always explore the first item from our Open list. The first item in our Open list is the B node. B is not our destination, so let's explore its neighbors:



Because I have now expanded B, I am going to remove it from the Open list and add it to the Closed List. Our new nodes are D and E, and we add these nodes to the beginning of our Open list:

- **Open List:** D, E, C
- **Closed List:** A, B

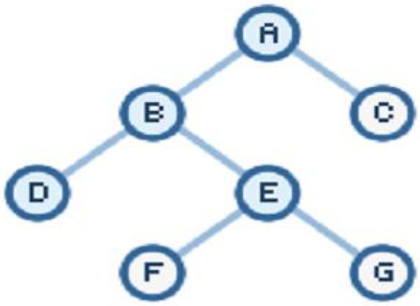
Step 3

We should start to see a pattern forming. Because D is at the beginning of our Open List, we expand it. D isn't our destination, and it does not contain any neighbors. All you do in this step is remove D from our Open List and add it to our Closed List:

- **Open List:** E, C
- **Closed List:** A, B, D

Step 4

We now expand the E node from our Open list. E is not our destination, so we explore its neighbors and find out that it contains the neighbors F and G. Remember, F is our target, but we don't stop here though. Despite F being on our path, we only end when we are about to expand our target Node - F in this case:

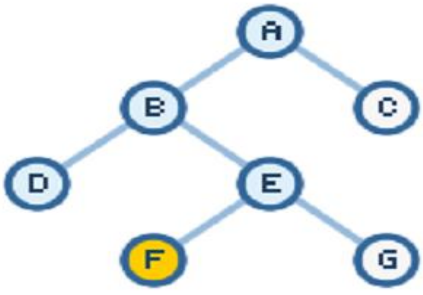


Our Open list will have the E node removed and the F and G nodes added. The removed E node will be added to our Closed List:

- **Open List:** F, G, C
- **Closed List:** A, B, D, E

Step 5

We now expand the F node. Since it is our intended destination, we stop:



We remove F from our Open list and add it to our Closed List. Since we are at our destination, there is no need to expand F in order to find its neighbors. Our final Open and Closed Lists contain the following data:

- **Open List**: G, C
- **Closed List:** A, B, D, E, F

The final path taken by our depth first search method is what the final value of our Closed List is: A, B, D, E, F.

Pseudocode / Informal Algorithms

The pseudocode for depth first search is:

i. Declare two empty lists: Open and Closed.
ii. Add Start node to our Open list.
iii. While our Open list is not empty, loop the following:
    a. Remove the first node from our Open List.
    b. Check to see if the removed node is our destination.
        i. If the removed node is our destination, break out of the loop, add the node to our closed list, and return the value of our Closed list.
        ii. If the removed node is not our destination, continue the loop (go to Step c).
    c. Extract the neighbors of our above removed node.
    d. Add the neighbors to the beginning of our Open list, and add the removed node to our Closed list. Continue looping.

Source Code:

```
graph = {
  'A': ['B', 'S'],
  'B': ['A'],
  'C': ['D', 'E', 'F', 'S'],
  'D': ['C'],
  'E': ['C', 'H'],
  'F': ['C', 'G'],
  'G': ['F', 'S'],
```

```
    'H': ['E', 'G'],
    'S': ['A', 'C', 'G']
}
visited = []  # Array to keep track of visited nodes.
def dfs(visited, graph, node):
    if node not in visited:
        print(node, end=' ')
        visited.append(node)
        for neighbour in graph[node]:
            dfs(visited, graph, neighbour)
# Driver Code
dfs(visited, graph, 'A')
```

BFS Algorithm Complexity

The time complexity of the BFS algorithm is represented in the form of O(V + E), where V is the number of nodes and E is the number of edges. The space complexity of the algorithm is O(V).

BFS Algorithm Applications

1. To build index by search index
2. For GPS navigation
3. Path finding algorithms
4. In Ford-Fulkerson algorithm to find maximum flow in a network
5. Cycle detection in an undirected graph
6. In minimum spanning tree