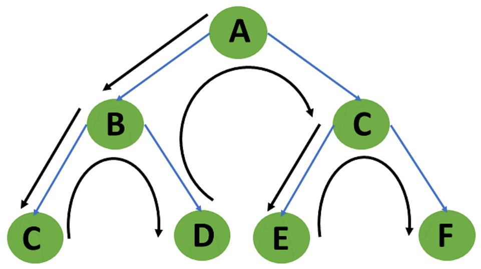
Lab 3 - Implementation of Depth First Search

Depth First Search

Depth-First Search or DFS algorithm is a recursive algorithm that uses the backtracking principle. It entails conducting exhaustive searches of all nodes by moving forward if possible and backtracking, if necessary. To visit the next node, pop the top node from the stack and push all of its nearby nodes into a stack. Topological sorting, scheduling problems, graph cycle detection, and solving puzzles with just one solution, such as a maze or a sudoku puzzle, all employ depth-first search algorithms. Other applications include network analysis, such as determining if a graph is bipartite.

What is a Depth-First Search Algorithm?

The depth-first search or DFS algorithm traverses or explores data structures, such as trees and graphs. The algorithm starts at the root node (in the case of a graph, you can use any random node as the root node) and examines each branch as far as possible before backtracking.



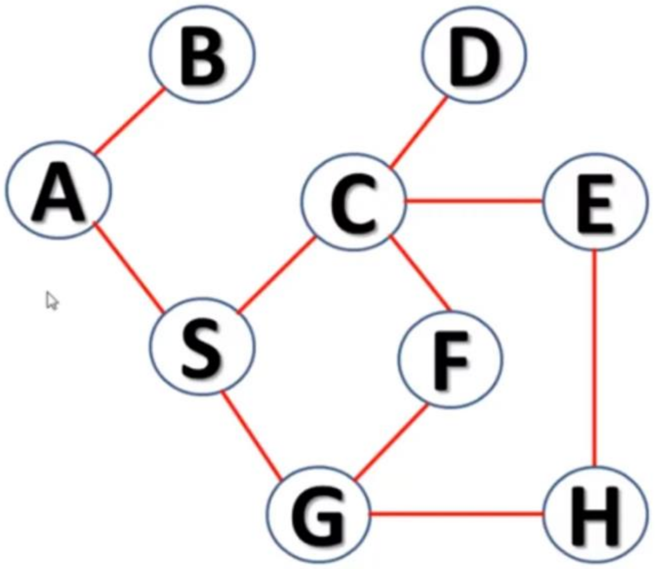
When a dead-end occurs in any iteration, the Depth First Search (DFS) method traverses a network in a deathward motion and uses a stack data structure to remember to acquire the next vertex to start a search.

Following the definition of the dfs algorithm, you will look at an example of a depth-first search method for a better understanding.

Example of Depth-First Search Algorithm

The outcome of a DFS traversal of a graph is a spanning tree. A spanning tree is a graph that is devoid of loops. To implement DFS traversal, you need to utilize a stack data structure with a maximum size equal to the total number of vertices in the graph.

To implement DFS traversal, you need to take the following stages.



Step 1: A is the root node. So since A is visited, we push this onto the stack.
Stack : A
Step 2: Let's go to the branch A-B. B is not visited, so we go to B and push B onto the stack.
Stack : A B
Step 3: Now, we have come to the end of our A-B branch and we move to the n-1th node which is A. We will now look at the adjacent node of A which is S. Visit S and push it onto the stack. Now you have to traverse the S-C-D branch, up to the depth ie upto D and mark S, C, D as visited.
Stack: A B S C D
Step 4: Since D has no other adjacent nodes, move back to C and traverse its adjacent branch E-H-G to the depth and push them onto the stack.
Stack : A B S C D E H G
Step 5: On reaching D, there is only one adjacent node ie F which is not visited. Push F onto the stack as well.

Stack : A B S C D E H G F

This stack itself is the traversal of the DFS.

Pseudocode

```
1.  DFS(G,v)   ( v is the vertex where the search starts )
2.          Stack S := {};   ( start with an empty stack )
3.          for each vertex u, set visited[u] := false;
4.          push S, v;
5.          while (S is not empty) do
6.             u := pop S;
7.             if (not visited[u]) then
8.                visited[u] := true;
```

```
9.           for each unvisited neighbour w of uu
10.            push S, w;
11.        end if
12.      end while
13.    END DFS()
```

Code:

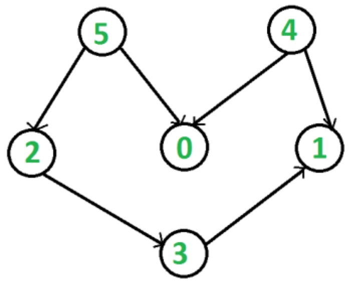
```
graph1 = {
    'A' : ['B','S'],
    'B' : ['A'],
    'C' : ['D','E','F','S'],
    'D' : ['C'],
    'E' : ['C','H'],
    'F' : ['C','G'],
    'G' : ['F','S'],
    'H' : ['E','G'],
    'S' : ['A','C','G']
}

def dfs(graph, node, visited):
    if node not in visited:
        visited.append(node)
        for k in graph[node]:
            dfs(graph,k, visited)
    return visited

visited = dfs(graph1,'A', [])
print(visited)
```

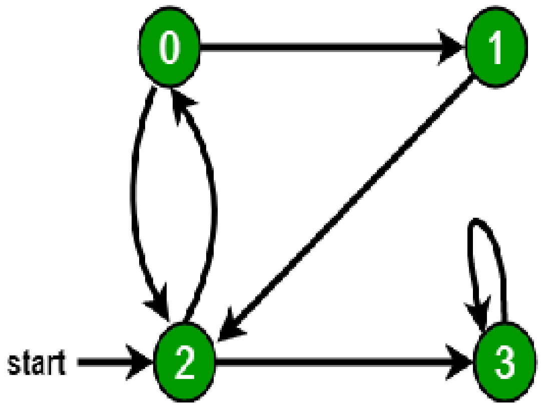
Lab exercises:

Q.1) Implement topological sorting using DFS algorithm for the following graph.



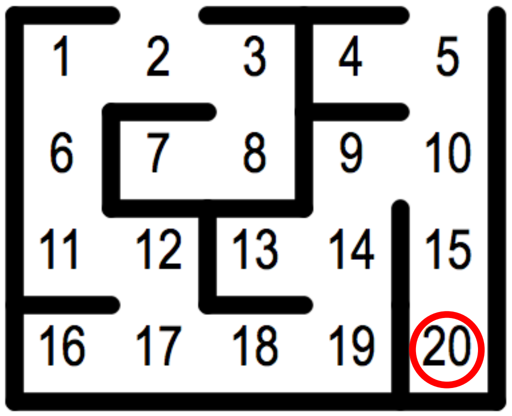
Note: Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge u v, vertex u comes before v in the ordering.
For example, a topological sorting of the following graph is "5 4 2 3 1 0". There can be more than one topological sorting for a graph. Another topological sorting of the following graph is "4 5 2 3 1 0". The first vertex in topological sorting is always a vertex with an in-degree of 0 (a vertex with no incoming edges).

Q.2) Consider the following directed graph for detecting cycles in the graph using DFS algorithm using Python.



Q.3) Write a Python program to solve the maze problem using DFS algorithm. The following is the problem statement and algorithm for the maze problem. Con

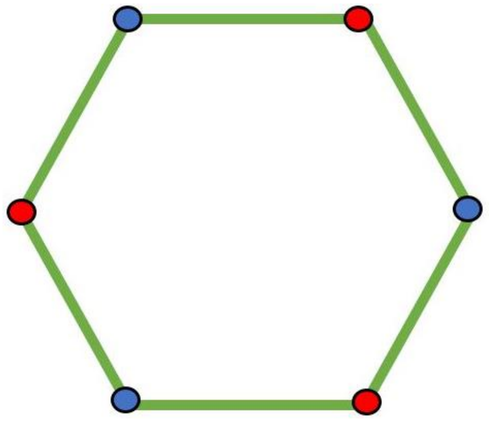
1. Enter the maze
2. If you have multiple ways, choose anyone and move forward
3. Keep choosing a way which was not seen so far till you exit the maze or reach dead end
4. If you exit maze, you are done.
5. If you reach dead end, this is wrong path, so take one step back, choose different path. If all paths are seen in this, take one step back and repeat



Additional questions:

Q.1) Write a Python program to solve 3x3 sudoku with Depth First Search algorithm.

Q.2) Write a Python code to check if a given graph is Bipartite using DFS.



Note: A bipartite graph is possible if the graph coloring is possible using two colors such that vertices in a set are colored with the same color.

Lab 4 – Implementation of Breadth First Search

Breadth First Search

The breadth-first search (BFS) algorithm is used to search a tree or graph data structure for a node that meets a set of criteria. It starts at the tree's root or graph and searches/visits all nodes at the current depth level before moving on to the nodes at the next depth level.

Breadth- First -Search:

Consider the state space of a problem that takes the form of a tree. Now, if we search the goal along each breadth of the tree, starting from the root and continuing up to the largest depth, we call it *breadth first search*.

Algorithm:

1. Create a variable called NODE-LIST and set it to initial state
2. Until a goal state is found or NODE-LIST is empty do
    a. Remove the first element from NODE-LIST and call it E. If NODE-LIST was empty,quit
    b. For each way that each rule can match the state described in E do:
        i. Apply the rule to generate a new state
        ii. If the new state is a goal state, quit and return this state
        iii. Otherwise, add the new state to the end of NODE-LIST

BFS illustrated:

Step 1: Initially fringe contains only one node corresponding to the source state A.

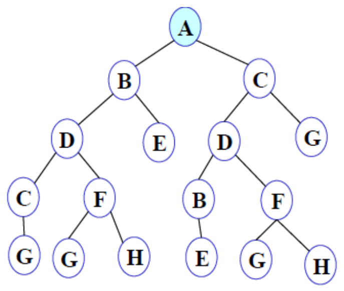


Figure 1

FRINGE: A

Step 2: A is removed from fringe. The node is expanded, and its children B and C are generated. They are placed at the back of fringe.

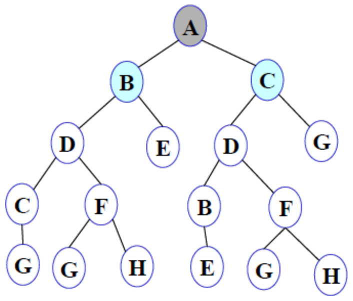


Figure 2

FRINGE: B C

Step 3: Node B is removed from fringe and is expanded. Its children D, E are generated and put at the back of fringe.

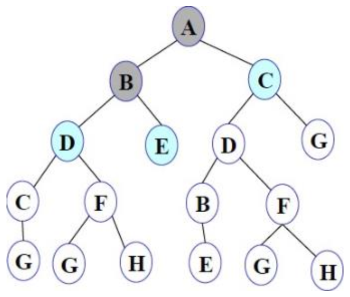


Figure 3

FRINGE: C D E

Step 4: Node C is removed from fringe and is expanded. Its children D and G are added to the back of fringe.

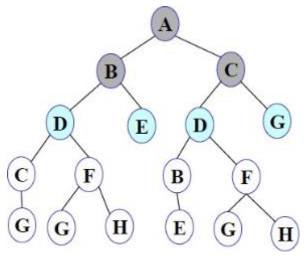


Figure 4

FRINGE: D E D G

Step 5: Node D is removed from fringe. Its children C and F are generated and added to the backof fringe.

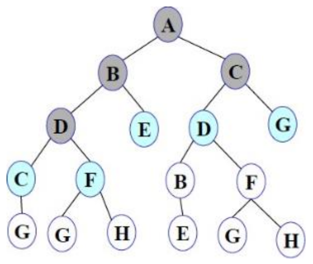


Figure 5

FRINGE: E D G C F

Step 6: Node E is removed from fringe. It has no children.

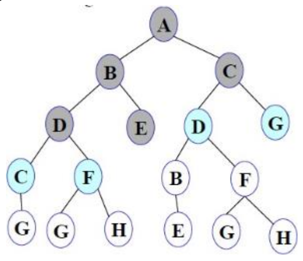


Figure 6

FRINGE: D G C F

Step 7: D is expanded; B and F are put in OPEN.

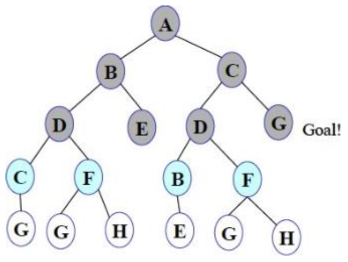
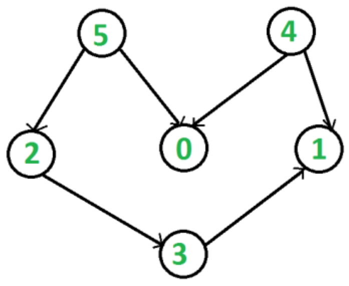


Figure 7

FRINGE: G C F B F

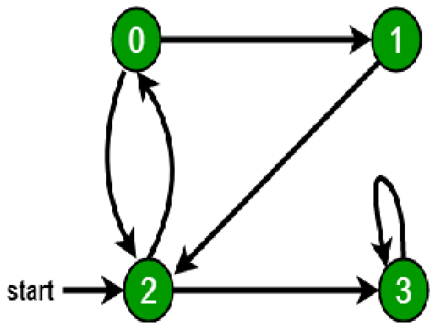
Lab Exercise 1:

Q.1) Implement topological sorting using BFS algorithm for the following graph.



Note: Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge u v, vertex u comes before v in the ordering.
For example, a topological sorting of the following graph is "5 4 2 3 1 0". There can be more than one topological sorting for a graph. Another topological sorting of the following graph is "4 5 2 3 1 0". The first vertex in topological sorting is always a vertex with an in-degree of 0 (a vertex with no incoming edges).

Q.2) Consider the following directed graph for detecting cycles in the graph using BFS algorithm using Python.



Q3. Write python code for Traveling Salesman Problem (TSP) using Breadth First Search (BFS). Graph Given Below.

```
graph = {
   'A': {'B': 2, 'C': 3, 'D': 1},
   'B': {'A': 2, 'C': 4, 'D': 2},
   'C': {'A': 3, 'B': 4, 'D': 3},
   'D': {'A': 1, 'B': 2, 'C': 3}
}
```

Additional Exercise: Write a Python program to solve 3x3 sudoku with Depth First Search algorithm