PAGE :
DATE : / /

1. Iterative deepening Search depth first search :-

Algorithm :-

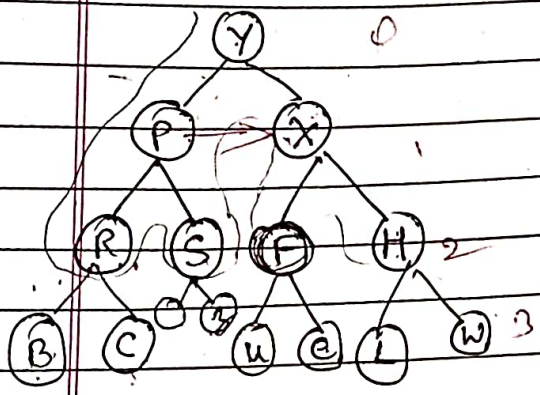
1. Create an empty graph using an adjacency list
2. Start at the root node until you find the goal node
3. Perform Depth limited search at that depth
   - If goal node is found at the current depth,
     return "Success"
   - else:
     increment the depth & continue searching
4. If the depth limit is reached return failure for that branch.
   - for the child node of the current node
     - if the neighbor hasn't been visited yet, perform a recursive DLS on that node with the depth reduced by 1
     - if any recursive DLS call finds the goal node, return success.
     - else :- return "failure".

Terminate :

- if the goal node is found at any depth, "return 'Success'"
- else:
  if the search reaches the maximum depth without finding the goal, "return failure".

15/10/24

PAGE :
DATE : / /



Y

Y P X

Y P R S X F

PAGE :
DATE : / /

1)
```
from collections import defaultdict
class Graph:
    def __init__(self):
        self.graph = defaultdict(list)
    def add_edge(self, u, v):
        self.graph[u].append(v)
    def idfs(self, start, goal, max_depth):
        for depth in range(max_depth+1):
            visited = set()
            if self.dls(start, goal, depth, visited):
                return True
        return False
    def dls(self, node, goal, depth, visited):
        if node == goal:
            return True
        if depth == 0:
            return False
        visited.add(node)
        for neighbor in self.graph[node]:
            if neighbor not in visited:
                if self.dls(neighbor, goal, depth-1,
                            visited):
                    return True
        return False
g = Graph()
g.add_edge(0, 1)
g.add_edge(0, 2)
g.add_edge(1, 3)
g.add_edge(1, 4)
g.add_edge(2, 5)
g.add_edge(2, 6)
start = 0
goal = 5
max_depth = 3
```

PAGE :
DATE :

```
if g.iddfs(start, goal, max_depth):
        print("path found")
else:
        print("path not found")
```

o/p :-
path found.
Y -> P -> R -> S -> X -> F

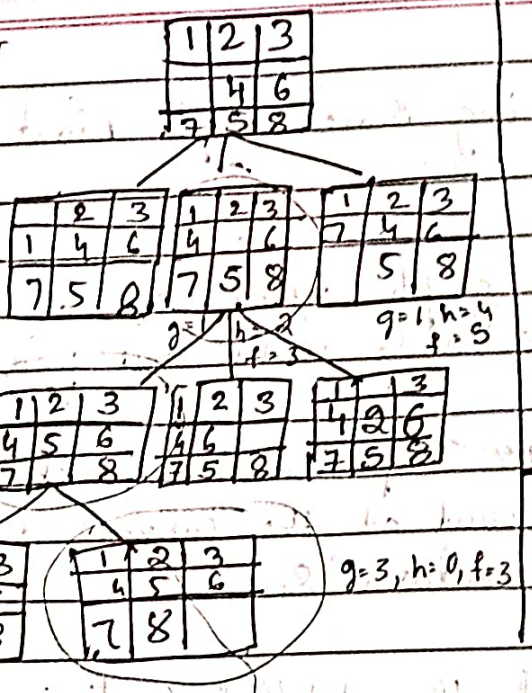
13/12/24

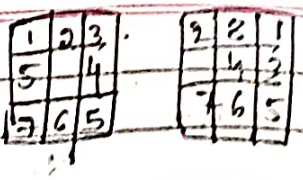
```
Path found: A -> C -> G
Target is reachable from source within max depth

=== Code Execution Successful ===
```

PAGE :
DATE : / /

2. A*





moves :- up down left right

Use $f(x) = g(x) + h(x)$

'g' represents the number of moves it took to reach a particular node from initial configuration

'h' is the heuristic function which basically estimates how far the current state is from the goal state

Step 1 Create 3x3 matrix

Step 2 Initialize open & closed lists

Step 3 create the start node
-> Add the start node to open list

Select the node with the lowest 'f' value from open list

Step 4 check if the current node is the goal node if not check with the child nodes. w.r.t 'f' value

Proceed

```
[4, 2, 6]
[7, 5, 8]

Current State:
[1, 0, 3]
[4, 2, 6]
[7, 5, 8]

Current State:
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]

Current State:
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]

Current State:
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]

Found with 12 iterations

=== Code Execution Successful ===
```

```
Current State:
[4, 1, 3]
[7, 2, 6]
[0, 5, 8]

Current State:
[4, 1, 3]
[0, 2, 6]
[7, 5, 8]

Current State:
[0, 1, 3]
[4, 2, 6]
[7, 5, 8]

Current State:
[1, 0, 3]
[4, 2, 6]
[7, 5, 8]

Current State:
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]

Current State:
```

```
Current State:
[4, 1, 3]
[7, 2, 6]
[5, 8, 0]

Current State:
[4, 1, 3]
[7, 2, 0]
[5, 8, 6]

Current State:
[4, 1, 3]
[7, 2, 6]
[5, 0, 8]

Current State:
[4, 1, 3]
[7, 0, 2]
[5, 8, 6]

Current State:
[4, 1, 3]
[7, 0, 6]
[5, 2, 8]

Current State:
```