N- Queens

Exp No : 1

1. N- Queens Problem using Backtracking in python.

Aim :- Write a program to solve N-Queens problem with python code.

Algorithm:

i) Start

ii) Placing empty space - Create an N×N board filled.

iii) ~~Call~~ Check, if placing Queen is safe.
No Queen, is in the same row.
No other is in the same diagonal.
No other queen is in the same lower diagonal.

iv) If a safe position is found, place the queen ('1') and move to next column using recursion.

v) If all Queen are successfully placed the algorithm, return 'True' indicating that the solution is found.

vi) If placing the Queen in the current column does not lead to solution Algorithm Backtracks to the previous column and tries next row.

Program:

```
def isSafe(board, row, col, n):
    for i in range(col):
        if board[row][i] == 1:
            return False

    for i, j in zip(range(row, -1, -1), range(col-1, -1, -1)):
        if board[i][j] == 1:
            return False

    for i, j in zip(range(row, n, -1), range(col, -1, -1)):
        if board[i][j] == 1:
            return True

def solveNQUtil(board, col, n):
    if col >= n:
        return True

    for i in range(n):
        if isSafe(board, i, col, n):
            board[i][col] = 1
            if solveNQUtil(board, col+1, n) == True
                return True
            board[i][col] = 0
    return False

def solveNQ(n):
    board = [[0]*n for _ in range(n)]
    if solveNQUtil(board, 0, n) == false:
        print("Solution does not exists")
    return false
```

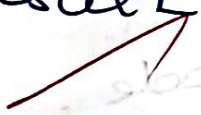
```
for i in board:
    print(i)
return True
n = int(input("enter n value : "))
solve NQ(n)
```

Output:

```
Enter n value : 5

[1, 0, 0, 0, 0]
[0, 0, 0, 1, 0]
[0, 1, 0, 0, 0]
[0, 0, 0, 0, 1]
[0, 0, 1, 0, 0]
```

Result:



Exp No: 2 Depth First Search

Aim:

To implement depth first search (DFS) to traverse a graph & explore all vertices by visiting as far along a branch as possible before backtracking.

Algorithm:

i) Start
ii) Initialize an empty stack and a list to keep track of visited nodes.
iii) Push the starting node onto stack & mas
iv) While the stack is not empty, repeat 5 to 7.
v) Pop the top node from the stack.
vi) Print or process the popped node.
vii) For each adjacent unvisited neighb of the popped node.
viii) Mark the neighbour as visited
xi) Push the unvisited neighbour onto the stack.
xii) Repeat until all reachable node are visited.
ix) Stop.

Program:

```
def dfs (graph, start):
    stack = [start]
    visited = set()
while stack:
    node = stack.pop()
    if node not in visited:
        print (node, end = " ")
        visited.add(node)
    for neighbor in graph[node]:
        if neighbor not in visited:
            stack.append(neighbor)
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': [F]
    'F': []
}
print("DFS Traversal Starting from node 'A':")
```

Output: DFS Traversal Start from node 'A'

~~A B E~~ A C F B E D