Backtracking

Introduction

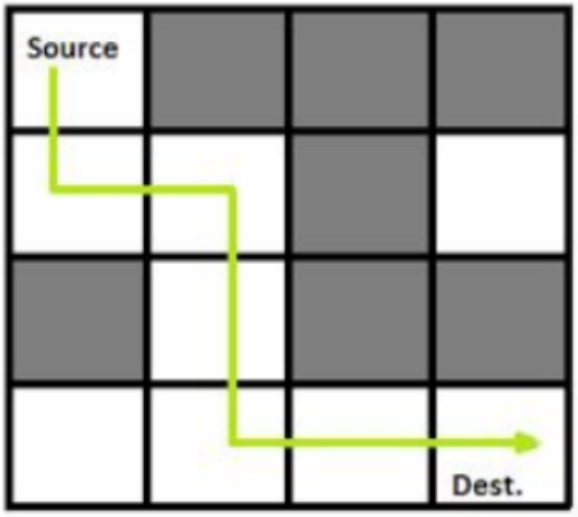
A backtracking algorithm is a problem-solving algorithm that uses a brute-force approach to find the desired output. The Brute force approach tries out all the possible solutions and chooses the desired/best solutions. The term backtracking suggests that if the current solution is not suitable, then backtrack and try other solutions. Thus, recursion is used in this approach. This approach is used to solve problems that have multiple solutions. Backtracking is thus a form of recursion. We begin by choosing an option and backtrack from it, if we reach a state where we conclude that this specified option does not give the required solution. We repeat these steps by going across each available option until we get the desired solution.

Difference between Recursion and Backtracking

In recursion, the function calls itself until it reaches a base case. In backtracking, we use recursion to explore all the possibilities until we get the best result for the problem.

Problem Statement: Rat in a Maze

You are given a starting position for a rat that is stuck in a maze at an initial point (0, 0) (the maze can be thought of as a 2-dimensional plane). The maze would be given in the form of a square matrix of order 'N' * 'N' where the cells with value 0 represent the maze's blocked locations while value 1 is the open/available path that the rat can take to reach its destination. The rat's destination is at ('N' - 1, 'N' - 1). Your task is to find all the possible paths that the rat can take to reach from source to destination in the maze. The possible directions that it can take to move in the maze are 'U'(up) i.e. (x, y - 1), 'D'(down) i.e. (x, y + 1) , 'L' (left) i.e. (x - 1, y), 'R' (right) i.e. (x + 1, y).



Algorithm to solve a rat in a maze

- Initialize Variables:
  - Initialize an empty list to store all possible paths.
  - Create a 2D visited array of size N x N to mark visited cells.
  - Mark the starting cell (0, 0) as visited.
- Define Helper Functions:
  - is_valid: Function to check if a given cell is a valid move. A move is valid if it is within the bounds of the maze and corresponds to an open path.
  - explore(): Recursive function to explore all possible paths from the current cell. It moves in all four directions and updates the path accordingly.

Implementation of Rat Maze in Python

```
# Import necessary modules
from os import *
from sys import *
from collections import *
from math import *

# Global list to store the paths found
ans = []
```

```
# Function to check if a given position is valid to move to
def isvalid(row, col, arr, visited, n):
    # Check if the position is out of bounds or already
visited
    if row < 0 or row >= n or col < 0 or col >= n or
arr[row][col] == 0 or visited[row][col]:
        return False
    return True

# Function to explore all possible paths in the maze
def explore(row, col, arr, visited, n, pathtaken):
    global ans
    # Check if the current position is empty
    if arr[row][col] == 0:
        return
    # Check if we have reached the destination
    if row == (n - 1) and col == (n - 1):
        ans.append(pathtaken)
        return
    # Define directions: Down, Left, Right, Up
    drow = [1, 0, 0, -1]
    dcol = [0, -1, 1, 0]
    dpath = ["D", "L", "R", "U"]
    # Explore in each direction
    for i in range(4):
        next_row = row + drow[i]
        next_col = col + dcol[i]
        # Check if the next position is valid
        if isvalid(next_row, next_col, arr, visited, n):
            visited[next_row][next_col] = True
            pathtaken += dpath[i]  # Add direction to the
current path
            # Recursively explore from the next position
            explore(next_row, next_col, arr, visited, n,
pathtaken)
            # Backtrack: Mark the position as unvisited and
remove the last direction from the path
            visited[next_row][next_col] = False
            pathtaken = pathtaken[:-1]
```

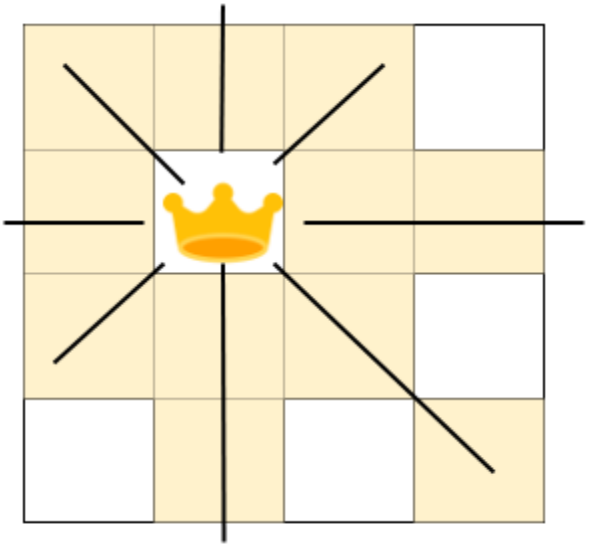
```
# Function to search the maze for all possible paths
def searchMaze(arr, n):
    global ans
    ans = []
    visited = [[False for _ in range(n)] for _ in range(n)]
# Initialize visited array
    visited[0][0] = True  # Mark the starting position as
visited
    explore(0, 0, arr, visited, n, "")  # Start exploring
from the starting position
    return ans

# Sample usage
# arr is a 2D array representing the maze, n is the size of
the maze
# searchMaze(arr, n) will return a list of all possible paths
through the maze
```

The time complexity of the rat in a maze is O(2^(n^2)). The recursion can run upper bound 2^(n^2) times. The space complexity is O(n^2) because an output matrix is required, so an extra space of size n*n is needed.

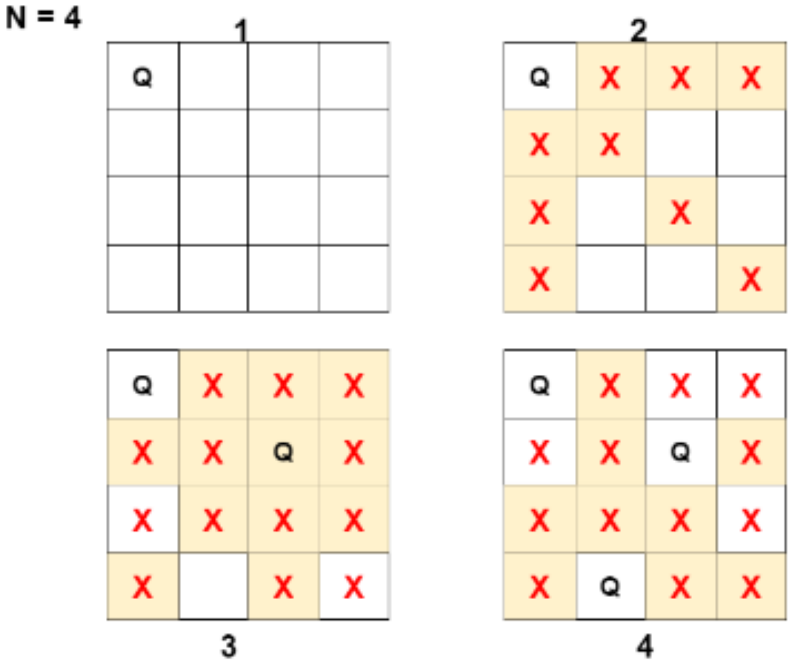
Problem Statement: N-Queen

You are given an integer **N**. For a given **N** x **N** chessboard, find a way to place **'N'** queens such that no queen can attack any other queen on the chessboard. A queen can be attacked when it lies in the same row, column, or the same diagonal as any of the other queens. You have to print one such configuration.

Print a binary matrix as output that has 1s for the cells where queens are placed.
Cells that can be attacked by a queen are shown below -



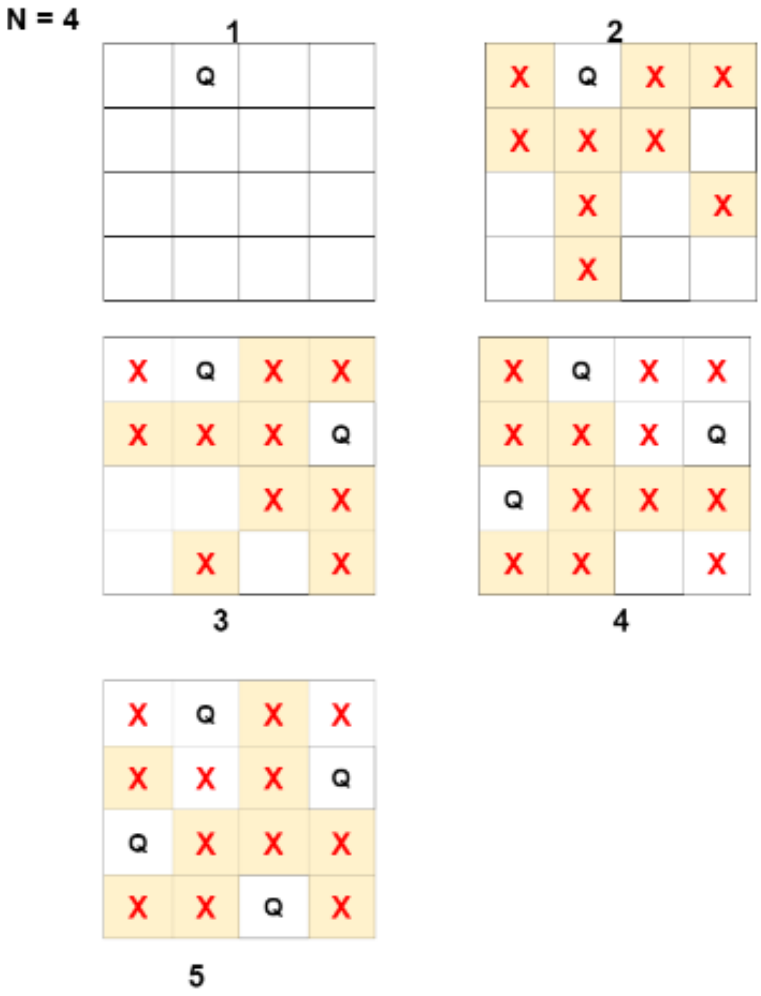
Let’s take an example to understand the requirement. Say, N=4. Now, we will try to place the 4 queens such that each of them is safe from every other queen.



In this figure, the red cross mark represents the cells that the queens can attack.

Observe that with this arrangement, we can only place 3 queens, and the 4th queen can't be placed as any of the positions left is safe. So, this is not the desired configuration.

Let’s try to place the queens in some other way -



Hence, the valid configuration is -
0 1 0 0
0 0 0 1
1 0 0 0
0 0 1 0
Note: There can be more than one such valid configuration. In this problem, our goal is just to print one of those.

The key idea is that no two queens can be placed in:

- Same Row
- Same Column
- Same Diagonal

There are **N** rows, so we will place one queen in each row(as no two queens can be in the same column) such that they all are safe from each other.

We will use backtracking to solve the problem. It is a recursive approach in which we place the queen in a safe position and then recur for the remaining queens. If at any step the number of queens to be placed is not zero and there are no safe cells left, then we change the position of the previously placed queen and try another safe position.

Let's see backtracking the approach step by step -

1. Start with the first row.
2. Check the number of queens placed. If it is **N**, then return true.
3. Check all the columns for the current row one by one.
    1. If the cell [row, column] is safe then mark this cell and recursively call the function for the remaining queens to find if it leads to the solution.

    If it leads to the solution, return true, else unmark the cell [row, column] (that is, backtrack), and check for the next column.

1. If the cell [row, column] is not safe, skip the current column and check for the next one.
2. If none of the cells in the current row is safe, then return false.

Implementation of N-Queen in Python

```
def is_safe(i, j, board, N):
    # checking for column j
    for k in range(i):
        if board[k][j] == 1:
            return False

    # Checking upper right diagonal
    k = i - 1
    l = j + 1
    while k >= 0 and l < N:
        if board[k][l] == 1:
            return False
        k -= 1
        l += 1
```

```
    # checking upper left diagonal
    k = i - 1
    l = j - 1
    while k >= 0 and l >= 0:
        if board[k][l] == 1:
            return False
        k -= 1
        l -= 1

    return True  # the cell[i][j] is safe


def n_queen(row, number_of_queens, N, board):
    if number_of_queens == 0:
        return True

    for j in range(N):
        if is_safe(row, j, board, N):
            board[row][j] = 1

            if n_queen(row + 1, number_of_queens - 1, N,
board):
                return True

            board[row][j] = 0  # backtracking

    return False


def main():
    N = 4
    board = [[0 for _ in range(N)] for _ in range(N)]

    n_queen(0, 4, N, board)

    # Printing the board
    for row in board:
        print(' '.join(map(str, row)))
```

```
if __name__ == "__main__":
    main()
```

Problem Statement: Word Search

You are given an M * N grid named ‘board’, and you are also given a string named ‘word’. Now you have to find out whether the given string can be constructed from the sequentially adjacent cells, where adjacent cells are horizontally or vertically neighboring. The same cell of a grid cannot be used twice.
Let’s understand this with the help of an example:

You are given a 2-D grid named ‘board’ like this:

A	B	C	E
S	F	C	S
A	D	E	E

Let you be given a string, ‘word’ as “ABCCED". Now you have to return true or false based on whether that string can be constructed inside the grid or not. Let’s try to form the string inside this grid:-

A	B	C	E
S	F	C	S
A	D	E	E

Now you can see that boxes colored in green show that this 2-D grid can accommodate the string. So the output that you have to print is true.

Approach to Solve Word Search

To solve a word search, we have a step-by-step plan. We look at each box on the grid, make sure it's okay to use, and then search in different directions to find the words we're looking for.

- isvalid(): This function checks whether a given position (row, col) on the board is valid for placing the next character of the word. It checks for boundary conditions, whether the position has been visited before, and whether the character at that position matches the next character in the word.

- findword(): This function recursively searches for the word starting from a given position (row, col) on the board. It iterates over all possible directions (up, down, left, right) from the current position. For each direction, it checks if the next position is valid. If it is valid, it marks that position as visited, makes a recursive call to search for the next character of the word, and if successful, returns True. If not successful, it marks the position as unvisited and continues to explore other directions. If no direction leads to a valid word, it returns False.

- present(): This function iterates over all positions on the board. For each position, if the character matches the first character of the word, it initializes a visited array, calls the findword function to search for the word starting from that position, and returns True if the word is found. If the word is not found after iterating over all positions, it returns False.

Implementation of Word Search in Python

```
from typing import List

def isvalid(row, col, board, visited, word, index, n, m):
    if row < 0 or row >= n or col < 0 or col >= m or
visited[row][col] or board[row][col] != word[index]:
        return False
    return True

def findword(row, col, board, visited, word, k, n, m):
    if k == len(word) - 1:
```

```
        return True

    drow = [1, -1, 0, 0]
    dcol = [0, 0, 1, -1]

    for i in range(4):
        next_row = row + drow[i]
        next_col = col + dcol[i]

        if isvalid(next_row, next_col, board, visited, word,
k + 1, n, m):
            visited[next_row][next_col] = True
            ans = findword(next_row, next_col, board,
visited, word, k + 1, n, m)
            if ans:
                return True
            visited[next_row][next_col] = False

    return False

def present(board: List[List[str]], word: str, n: int, m:
int) -> bool:
    for i in range(n):
        for j in range(m):
            if board[i][j] == word[0]:
                visited = [[False for _ in range(m)] for _ in
range(n)]
                ans = findword(i, j, board, visited, word, 0,
n, m)
                if ans:
                    return True
    return False

# Test case
board = [
    ['c', 'z', 'k', 'l'],
    ['o', 'd', 'i', 'a'],
    ['r', 'g', 'n', 'm'],
    ['m', 'r', 's', 'd']
```

```
]
word = "coding"
n = 4
m = 4

print(present(board, word, n, m))  # Output: True
```