Csp

class NQueens:

def \_\_init\_\_(self, n):

self.n = n

self.solutions = []

self.board = [['.' for \_ in range(n)] for \_ in range(n)]

# Branch and Bound helpers:

self.columns = [False] \* n

self.left\_diagonal = [False] \* (2 \* n - 1)

self.right\_diagonal = [False] \* (2 \* n - 1)

def solve(self, row):

if row == self.n:

# Store solution

solution = [''.join(self.board[i]) for i in range(self.n)]

self.solutions.append(solution)

return

for col in range(self.n):

if (not self.columns[col] and

not self.left\_diagonal[row - col + self.n - 1] and

not self.right\_diagonal[row + col]):

# Place Queen

self.board[row][col] = 'Q'

self.columns[col] = True

self.left\_diagonal[row - col + self.n - 1] = True

self.right\_diagonal[row + col] = True

# Recurse to next row

self.solve(row + 1)

# Backtrack

self.board[row][col] = '.'

self.columns[col] = False

self.left\_diagonal[row - col + self.n - 1] = False

self.right\_diagonal[row + col] = False

def print\_solutions(self):

if not self.solutions:

print("No solutions found.")

else:

print(f"\nTotal solutions for {self.n}-Queens: {len(self.solutions)}\n")

for idx, solution in enumerate(self.solutions, 1):

print(f"Solution {idx}:")

for row in solution:

print(row)

print()

# Main Execution

if \_\_name\_\_ == '\_\_main\_\_':

n = int(input("Enter the number of Queens (N): "))

queens = NQueens(n)

queens.solve(0)

queens.print\_solutions()

# explanation :for backtracking : • Recursive Search: The solve method is called recursively to #place queens row by row.

#Backtrack: If a safe spot is found for the queen in the current row (row), the queen is placed, #and the algorithm moves to the next row. If no valid position is found in a row, the algorithm backtracks by removing the queen and trying other positions.

# Branch and bound :The solution uses a bounding technique by maintaining three arrays: columns, left\_diagonal, and right\_diagonal.

# These arrays track the positions where queens are already placed in the respective columns and diagonals.

# Before placing a queen, the algorithm checks whether the current column or diagonal is already occupied (not self.columns[col], not self.left\_diagonal[...], not self.right\_diagonal[...]). This step is essentially a bound that prevents the algorithm from trying configurations that are obviously invalid.

Code explaination

**ode Explanation:**

This code implements the **N-Queens problem** using a **backtracking algorithm** with a **branch and bound** optimization to solve it efficiently.

**Class: NQueens**

1. **\_\_init\_\_(self, n)**:
   * **n**: The number of queens and the size of the board (i.e., an n x n chessboard).
   * **self.board**: A 2D list representing the chessboard. Initially, all positions are empty (denoted by .).
   * **self.columns**: A list to track which columns have a queen placed. The index represents the column, and True means that column is occupied.
   * **self.left\_diagonal** and **self.right\_diagonal**: These lists track which diagonals are already blocked. The diagonals are represented using their respective indices (calculated using row - col for left diagonal and row + col for right diagonal).
   * **self.solutions**: A list that stores all the solutions found. Each solution is represented as a list of strings (each string representing a row on the board).
2. **solve(self, row)**:
   * **Base case**: If row == self.n, that means all queens have been placed on the board successfully, and the solution is stored.
   * **Recursive case**: For each column in the current row (row), the algorithm checks whether it's safe to place a queen. If placing a queen is valid (the column and both diagonals are not occupied), it places the queen and moves on to the next row by calling solve(row + 1).
   * **Backtracking**: If placing a queen in the current position doesn't lead to a valid solution later, the algorithm removes the queen (backtracks) and tries the next possible position in the row.
   * **Bounding**: The algorithm uses the columns, left\_diagonal, and right\_diagonal arrays to ensure that a queen isn't placed in a column or diagonal where another queen already exists. This makes the algorithm more efficient by avoiding obviously invalid configurations.
3. **print\_solutions(self)**:
   * After solving, this method prints all the solutions. If no solution is found, it prints "No solutions found."
   * If solutions exist, it prints each solution, one by one.

**Backtracking:**

* **Recursive Search**: The solve() method is called recursively, attempting to place a queen row by row. For each row, it tries placing a queen in every column. If a safe position is found, the queen is placed, and the algorithm proceeds to the next row. If no valid position is found for a queen in a row, the algorithm backtracks by removing the queen and trying the next column.
* **Backtrack Process**: If a queen is placed in a position and the algorithm finds that this leads to a conflict later (e.g., a queen is placed in a position where another queen can attack it), the algorithm removes the queen from the board (backtracks) and tries the next position in the current row.

**Branch and Bound:**

* **Bounding** is achieved by maintaining the arrays:
  + **columns**: A boolean array where columns[i] is True if a queen is already placed in column i. This helps prevent placing multiple queens in the same column.
  + **left\_diagonal**: This array keeps track of the occupied left diagonals. A queen placed at position (i, j) will occupy a left diagonal indexed by i - j + n - 1.
  + **right\_diagonal**: This array keeps track of the occupied right diagonals. A queen placed at position (i, j) will occupy a right diagonal indexed by i + j.
* Before placing a queen, the algorithm checks if the current column or diagonals are already blocked using the columns, left\_diagonal, and right\_diagonal arrays. This bounding step ensures that the algorithm doesn’t waste time trying invalid configurations.

**Main Execution:**

1. The program asks for the value of n, the number of queens and the size of the board.
2. An instance of the NQueens class is created with n queens.
3. The solve() method is called starting from row 0 to begin the recursive search.
4. After solving, the print\_solutions() method is called to print all the solutions.

**Example:**

**Input:**

mathematica

CopyEdit

Enter the number of Queens (N): 4

**Execution:**

1. **Board Configuration**: The program tries placing queens on a 4x4 chessboard. It explores all valid configurations where queens do not threaten each other (no two queens in the same row, column, or diagonal).
2. **Finding Solutions**: After going through possible configurations, the program finds all valid solutions. For n=4, there are two valid solutions.

**Output:**

less

CopyEdit

Total solutions for 4-Queens: 2

Solution 1:

. Q . .

. . . Q

Q . . .

. . Q .

Solution 2:

. . Q .

Q . . .

. . . Q

. Q . .

**Explanation of Output:**

* **Solution 1**: One possible configuration is:
  + Queen in row 0, column 1.
  + Queen in row 1, column 3.
  + Queen in row 2, column 0.
  + Queen in row 3, column 2.
* **Solution 2**: Another valid configuration is:
  + Queen in row 0, column 2.
  + Queen in row 1, column 0.
  + Queen in row 2, column 3.
  + Queen in row 3, column 1.

Both configurations are valid because no two queens threaten each other.