Practical 4 (N Queens Problem)

Name:Srushti Tatte Roll no:13353 Div:C Batch :C3

In [ ]:

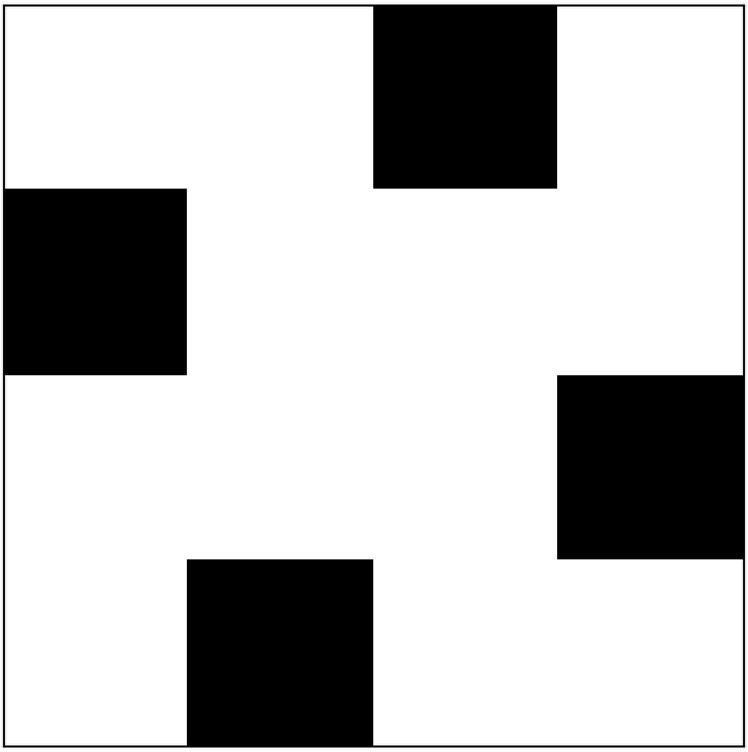
```
import matplotlib.pyplot as plt
import numpy as np

class NQueens:
    def __init__(self, n): #single argument n, which represents the size of the che
        self.n = n #assigns the size of the chessboard to the instance variable n.
        self.board = [[0] * n for _ in range(n)] #initializes the chessboard as a 2
        self.solutions = [] #initializes an empty list to store the solutions found

#for checking whether it is safe to place a queen at the given position (row, col)
    def is_safe(self, row, col):
        for i in range(col):#iterates through the columns preceding the current col
            if self.board[row][i] == 1: #checks if there is already a queen placed
                #but in a previous column (i)
                #f a queen is found in the same row in any of the preceding columns
                #(row, col) would result in a horizontal conflict, so it returns Fa
                return False
            #checks if there is a queen in the upper-left diagonal from position (r
            #ensures that no queen is present in the upper-left diagonal direction
            if row - i - 1 >= 0 and self.board[row - i - 1][col - i - 1] == 1:
                return False
            #checks if there is a queen in the lower-left diagonal from position (r
            if row + i + 1 < self.n and self.board[row + i + 1][col - i - 1] == 1:
                #If a queen is found in the lower-left diagonal, it means placing a
                #(row, col) would result in a conflict, so it returns False.
                return False
            #If no conflicts are found (i.e., no queens are present in the same row
            #and lower-left), it means it's safe to place a queen at position (row,
        return True

#solve_backtracking method serves as the termination condition for the recursion. I
#have been successfully placed on the board and, if so, adds the current board conf
    def solve_backtracking(self, col):
        if col >= self.n:
            self.solutions.append([row[:] for row in self.board])
            return True

# iterates through each row of the current column.#For each row (i), it checks if i
#the current column (col) using the is_safe method.If it's safe, it proceeds to pla
#current step of the backtracking process, indicating which column is being process
        for i in range(self.n):
            if self.is_safe(i, col):
                self.board[i][col] = 1
                print(f"Step {col+1}:")
                self.print_board() #prints the current state of the board after pla
                #for visualization purposes.
                if self.solve_backtracking(col + 1):
                    return True
#If placing a queen in row i of the current column leads to a dead end (i.e., no so
#it backtracks by resetting the cell to 0
                self.board[i][col] = 0
        return False

    def solve_branch_bound(self):
        self.solve_backtracking(0) #Solves the problem using backtracking
```

```
        min_solutions = [] #List to store solutions with minimum conflicts
        min_conflicts = float('inf') #Initialize minimum conflicts to positive infi
        for solution in self.solutions:# Iterate through each solution found
            conflicts = self.calculate_conflicts(solution)#Calculate conflicts in t
            if conflicts < min_conflicts: #If conflicts are less than the current m
                min_conflicts = conflicts #Update minimum conflicts
                min_solutions = [solution]#Replace existing min_solutions with the
            elif conflicts == min_conflicts:# If conflicts are equal to the current
                min_solutions.append(solution) # Add the current solution to min_sc
        return min_solutions ## Return the list of solutions with minimum conflicts

    def calculate_conflicts(self, solution):
        conflicts = 0 # Initialize conflicts counter
        for col in range(self.n):# Iterate through each column
            for i in range(self.n):# Iterate through each row (queen)
                for j in range(i + 1, self.n): # Iterate through each subsequent ro
                # Check for conflicts in the same column
                    if solution[i][col] == solution[j][col] == 1:
                        conflicts += 1# Increment conflicts counter Check for confl
                    if solution[col][i] == solution[col][j] == 1:
                        conflicts += 1 #Increment conflicts counter Check for confl
                    if (i + col - j >= 0 and i + col - j < self.n and
                            solution[i][col] == solution[i + col - j][j] == 1):
                        conflicts += 1 # Increment conflicts counter Check for conf
                    if (i - col + j >= 0 and i - col + j < self.n and
                            solution[i][col] == solution[i - col + j][j] == 1):
                        conflicts += 1 # Increment conflicts counter
        return conflicts # Return the total number of conflicts

    def visualize_solution(self, solution):
        board = np.array(solution)
        plt.imshow(board, cmap='binary')
        plt.xticks([])
        plt.yticks([])
        plt.title('N Queens Solution')
        plt.show()

    def print_board(self):
        for row in self.board:
            print(" ".join(map(str, row)))
        print()

# Menu-driven code
def menu():
    print("N Queens Problem Solver")
    print("1. Solve using Backtracking")
    print("2. Solve using Branch and Bound")
    print("3. Exit")
    choice = input("Enter your choice: ")
    return int(choice)

def main():
    n = int(input("Enter the number of queens: "))
    solver = NQueens(n)
    while True:
        choice = menu()
        if choice == 1:
            solver.solutions = []
            solver.solve_backtracking(0)
            print("Number of solutions found:", len(solver.solutions))
            if len(solver.solutions) > 0:
                for idx, solution in enumerate(solver.solutions):
                    print(f"Solution {idx + 1}:")
                    solver.visualize_solution(solution)
```

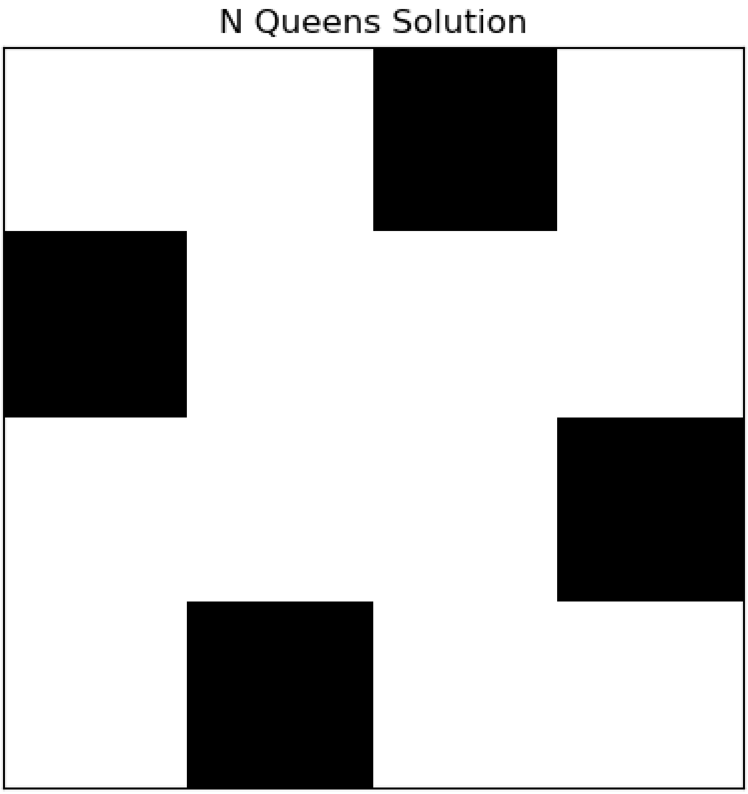
```
        elif choice == 2:
            solutions = solver.solve_branch_bound()
            print("Number of solutions found:", len(solutions))
            if len(solutions) > 0:
                for idx, solution in enumerate(solutions):
                    print(f"Solution {idx + 1}:")
                    solver.visualize_solution(solution)
        elif choice == 3:
            print("Exiting...")
            break
        else:
            print("Invalid choice. Please enter a valid option.")

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

```
Enter the number of queens: 4
N Queens Problem Solver
1. Solve using Backtracking
2. Solve using Branch and Bound
3. Exit
Enter your choice: 1
Step 1:
1 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0

Step 2:
1 0 0 0
0 0 0 0
0 1 0 0
0 0 0 0

Step 2:
1 0 0 0
0 0 0 0
0 0 0 0
0 1 0 0

Step 3:
1 0 0 0
0 0 1 0
0 0 0 0
0 1 0 0

Step 1:
0 0 0 0
1 0 0 0
0 0 0 0
0 0 0 0

Step 2:
0 0 0 0
1 0 0 0
0 0 0 0
0 1 0 0

Step 3:
0 0 1 0
1 0 0 0
0 0 0 0
0 1 0 0

Step 4:
0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0

Number of solutions found: 1
Solution 1:
```

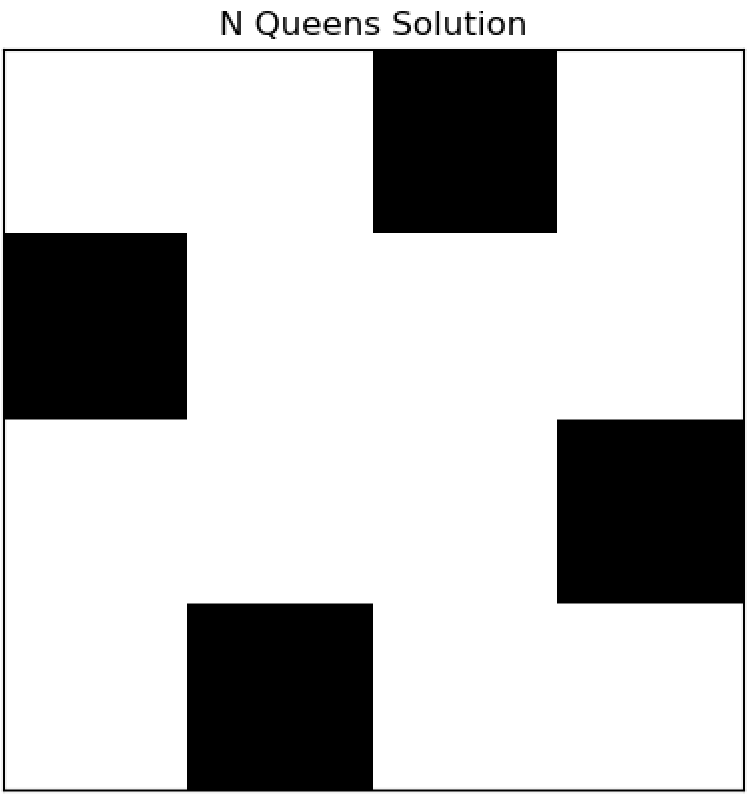
N Queens Solution



```
N Queens Problem Solver
1. Solve using Backtracking
2. Solve using Branch and Bound
3. Exit
Enter your choice: 2
Step 1:
1 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0

Step 2:
1 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0

Step 1:
0 0 1 0
1 0 0 0
0 0 0 1
0 0 0 0

Step 2:
0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0

Step 3:
0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0

Step 4:
0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0

Number of solutions found: 2
Solution 1:
```




```
Solution 2:
```




```
N Queens Problem Solver
1. Solve using Backtracking
2. Solve using Branch and Bound
3. Exit
```

In [ ]:

In [ ]: