**Problem Statement-**Implement a solution for a Constraint Satisfaction Problem using Branch and Bound and Backtracking for n-queens problem or a graph coloring problem

class NQueens:

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

self.n = n

self.board = [[0] \* n for \_ in range(n)]

self.solutions = []

def is\_safe(self, row, col):

# Check if it's safe to place a queen at (row, col)

# Check the same column

for i in range(row):

if self.board[i][col] == 1:

return False

# Check upper left diagonal

for i, j in zip(range(row, -1, -1), range(col, -1, -1)):

if self.board[i][j] == 1:

return False

# Check upper right diagonal

for i, j in zip(range(row, -1, -1), range(col, self.n)):

if self.board[i][j] == 1:

return False

return True

def solve(self, row):

if row == self.n:

# All queens are placed successfully, add the solution

self.solutions.append([list(row) for row in self.board])

return

for col in range(self.n):

if self.is\_safe(row, col):

self.board[row][col] = 1

self.solve(row + 1)

self.board[row][col] = 0 o

def nqueens(self):

self.solve(0)

return self.solutions

def print\_solution(self, solution):

for row in solution:

print(' '.join('Q' if cell == 1 else '.' for cell in row))

print()

if \_\_name\_\_ == "\_\_main\_\_":

n = 4 # Change the value of n for different board sizes

nqueens\_solver = NQueens(n)

solutions = nqueens\_solver.nqueens()

if solutions:

print(f"Solutions for {n}-Queens:")

for i, solution in enumerate(solutions, start=1):

print(f"Solution {i}:")

nqueens\_solver.print\_solution(solution)

else:

print(f"No solutions found for {n}-Queens.")

OUTPUT

