Implement a solution for a Constraint Satisfaction Problem using Branch and Bound and Backtracking for n-queens problem

Java Code

```
import java.io.*;
import java.util.Arrays;

class NQueens {
    public static void main(String args[]) {
        int N = 8;

        NQBranchAndBond NQBaB = new NQBranchAndBond(N);
        NQBaB.solveNQ();

        NQBacktracking NQBt = new NQBacktracking(N);
        NQBt.solveNQ();
    }
}

class NQBranchAndBond {

    private int N;

    NQBranchAndBond(int N) {
        this.N = N;
    }

    void printSolution(int board[][]) {
        System.out.println("N Queen Branch And Bound Solution:");
        for(int i = 0; i < N; i++) {
            for(int j = 0; j < N; j++)
                System.out.printf("%2d ", board[i][j]);
            System.out.printf("\n");
        }
    }

    static boolean isSafe (
        int row, int col,
        int slashCode[][],
        int backslashCode[][],
        boolean rowLookup[],
        boolean slashCodeLookup[],
        boolean backslashCodeLookup[]
    ) {
        return !(
            slashCodeLookup[slashCode[row][col]] ||
            backslashCodeLookup[backslashCode[row][col]] ||
            rowLookup[row]
        );
    }

    // A recursive utility function to solve N Queen problem
    boolean solveNQUtil(
        int board[][], int col, int slashCode[][],
        int backslashCode[][], boolean rowLookup[],
        boolean slashCodeLookup[], boolean backslashCodeLookup[]
    ) {
        // base case: If all queens are placed then return True
        if (col >= N)
            return true;
```

```
    for(int i = 0; i < N; i++) {
        if (isSafe(
            i, col, slashCode,
            backslashCode, rowLookup,
            slashCodeLookup, backslashCodeLookup
        )) {

            // Place this queen in board[i][col]
            board[i][col] = 1;
            rowLookup[i] = true;
            slashCodeLookup[slashCode[i][col]] = true;
            backslashCodeLookup[backslashCode[i][col]] = true;

            // recur to place rest of the queens
            if (solveNQUtil(
                board, col + 1, slashCode,
                backslashCode, rowLookup,
                slashCodeLookup,
                backslashCodeLookup
            ))
                return true;

            // If placing queen in board[i][col] doesn't
            // lead to a solution, then backtrack

            // Remove queen from board[i][col]
            board[i][col] = 0;
            rowLookup[i] = false;
            slashCodeLookup[slashCode[i][col]] = false;
            backslashCodeLookup[backslashCode[i][col]] = false;
        }
    }

    // If queen can not be place in any row
    // in this column col then return false
    return false;
}

/*
* This function solves the N Queen problem using Branch and Bound.
* It mainly uses solveNQUtil() to solve the problem.
* It returns false if queens cannot be placed, otherwise return
* true and prints placement of queens in the form of 1s.
* This function prints one of the feasible solutions.
*/
boolean solveNQ() {
    int board[][] = new int[N][N];

    // Helper matrices
    int slashCode[][] = new int[N][N];
    int backslashCode[][] = new int[N][N];

    // Arrays to tell us which rows are occupied
    boolean[] rowLookup = new boolean[N];

    // Keep two arrays to tell us which diagonals are occupied
    boolean slashCodeLookup[] = new boolean[2 * N - 1];
    boolean backslashCodeLookup[] = new boolean[2 * N - 1];

    // Initialize helper matrices
```

```
        for(int r = 0; r < N; r++)
            for(int c = 0; c < N; c++) {
                slashCode[r][c] = r + c;
                backslashCode[r][c] = r - c + N - 1;
            }

        if (solveNQUtil(
            board, 0, slashCode,
            backslashCode, rowLookup,
            slashCodeLookup,
            backslashCodeLookup
        ) == false) {
            System.out.printf("Solution does not exist");
            return false;
        }

        // Solution found
        printSolution(board);
        return true;
    }
}


class NQBacktracking {

    private int N;

    NQBacktracking(int N){
        this.N = N;
    }

    /* ld is an array where its indices indicate row-col+N-1 (N-1)
    is for shifting the difference to store negative indices */
    static int []ld = new int[30];

    /* rd is an array where its indices indicate row+col and used to
    check whether a queen can be placed on right diagonal or not */
    static int []rd = new int[30];

    /*column array where its indices indicates column and used
    to check whether a queen can be placed in that row or not*/
    static int []cl = new int[30];

    /* A utility function to print solution */
    void printSolution(int board[][]) {
        System.out.println("\n\nN Queen Backtracking Solution:");
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++)
                System.out.printf("%2d ", board[i][j]);
            System.out.printf("\n");
        }
    }

    /* A recursive utility function to solve N Queen problem */
    boolean solveNQUtil(int board[][], int col) {
        /* base case: If all queens are placed then return true */
        if (col >= N)
            return true;

        /* Consider this column and try placing
        this queen in all rows one by one */
```

```
        for (int i = 0; i < N; i++) {

            /* Check if the queen can be placed on board[i][col]
            A check if a queen can be placed on board[row][col]
           .We just need to check ld[row-col+n-1] and rd[row+coln]
            where ld and rd are for left and right diagonal respectively */
            if ((ld[i - col + N - 1] != 1 &&
                rd[i + col] != 1) && cl[i] != 1) {

                /* Place this queen in board[i][col] */
                board[i][col] = 1;
                ld[i - col + N - 1] =
                rd[i + col] = cl[i] = 1;

                /* recur to place rest of the queens */
                if (solveNQUtil(board, col + 1))
                    return true;

                /* If placing queen in board[i][col] doesn't lead to
                a solution, then remove queen from board[i][col] */

                board[i][col] = 0; // BACKTRACK
                ld[i - col + N - 1] =
                rd[i + col] = cl[i] = 0;
            }
        }

        /* If the queen cannot be placed in any row in
            this column col then return false */
        return false;
    }

    /* This function solves the N Queen problem using Backtracking. It
mainly
    * uses solveNQUtil() to solve the problem. It returns false if queens
    * cannot be placed, otherwise, return true and prints placement of
queens
    * in the form of 1s. This function prints one of the feasible
solutions.
    */
    boolean solveNQ() {
        int board[][] = new int[N][N];

        if (solveNQUtil(board, 0) == false) {
            System.out.printf("Solution does not exist");
            return false;
        }

        printSolution(board);
        return true;
    }
    }
```

Python Code :

```
class NQBranchAndBond:
  def printSolution(self, board):
    print("N Queen Branch And Bound Solution:")
    for line in board:
      print(" ".join(map(str, line)))
```

```
    def isSafe(
        self,
        row,
        col,
        slashCode,
        backslashCode,
        rowLookup,
        slashCodeLookup,
        backslashCodeLookup,
    ):
        return not (
            slashCodeLookup[slashCode[row][col]]
            or backslashCodeLookup[backslashCode[row][col]]
            or rowLookup[row]
        )

    """ A recursive utility function
    to solve N Queen problem """

    def solveNQUtil(
        self,
        board,
        col,
        slashCode,
        backslashCode,
        rowLookup,
        slashCodeLookup,
        backslashCodeLookup,
    ):

        """base case: If all queens are
        placed then return True"""
        if col >= N:
            return True

        for i in range(N):
            if self.isSafe(
                i,
                col,
                slashCode,
                backslashCode,
                rowLookup,
                slashCodeLookup,
                backslashCodeLookup,
            ):

                """Place this queen in board[i][col]"""
```

```
            board[i][col] = 1
            rowLookup[i] = True
            slashCodeLookup[slashCode[i][col]] = True
            backslashCodeLookup[backslashCode[i][col]] = True

            """ recur to place rest of the queens """
            if self.solveNQUtil(
               board,
               col + 1,
               slashCode,
               backslashCode,
               rowLookup,
               slashCodeLookup,
               backslashCodeLookup,
            ):
               return True

            """ If placing queen in board[i][col]
            doesn't lead to a solution,then backtrack """

            """ Remove queen from board[i][col] """
            board[i][col] = 0
            rowLookup[i] = False
            slashCodeLookup[slashCode[i][col]] = False
            backslashCodeLookup[backslashCode[i][col]] = False

      """ If queen can not be place in any row in
      this column col then return False """
      return False

"""" This function solves the N Queen problem using
Branch or Bound. It mainly uses solveNQUtil()to
solve the problem. It returns False if queens
cannot be placed,otherwise return True or
prints placement of queens in the form of 1s.
Please note that there may be more than one
solutions,this function prints one of the
feasible solutions."""

def solveNQ(self):
   board = [[0 for i in range(N)] for j in range(N)]

   # helper matrices
   slashCode = [[0 for i in range(N)] for j in range(N)]
   backslashCode = [[0 for i in range(N)] for j in range(N)]

   # arrays to tell us which rows are occupied
   rowLookup = [False] * N
```

```
        # keep two arrays to tell us
        # which diagonals are occupied
        x = 2 * N - 1
        slashCodeLookup = [False] * x
        backslashCodeLookup = [False] * x

        # initialize helper matrices
        for rr in range(N):
            for cc in range(N):
                slashCode[rr][cc] = rr + cc
                backslashCode[rr][cc] = rr - cc + N - 1

        if (
            self.solveNQUtil(
                board,
                0,
                slashCode,
                backslashCode,
                rowLookup,
                slashCodeLookup,
                backslashCodeLookup,
            )
            == False
        ):
            print("Solution does not exist")
            return False

        # solution found
        self.printSolution(board)
        return True


class NQBacktracking:
    def __init__(self):
        """self.ld is an array where its indices indicate row-col+N-1
        (N-1) is for shifting the difference to store negative indices"""
        self.ld = [0] * 30

        """ self.rd is an array where its indices indicate row+col and used
        to check whether a queen can be placed on right diagonal or not"""
        self.rd = [0] * 30

        """column array where its indices indicates column and
        used to check whether a queen can be placed in that row or not"""
        self.cl = [0] * 30

    """ A utility function to print solution """
```

```
def printSolution(self, board):
   print("\n\nN Queen Backtracking Solution:")
   for line in board:
      print(" ".join(map(str, line)))

""" A recursive utility function to solve N
Queen problem """

def solveNQUtil(self, board, col):

   """base case: If all queens are placed
   then return True"""
   if col >= N:
      return True

   """ Consider this column and try placing
      this queen in all rows one by one """
   for i in range(N):

      """ Check if the queen can be placed on board[i][col]
      A check if a queen can be placed on board[row][col].
      We just need to check self.ld[row-col+n-1] and self.rd[row+coln]
      where self.ld and self.rd are for left and right diagonal respectively"""
      if (self.ld[i - col + N - 1] != 1 and
         self.rd[i + col] != 1) and self.cl[i] != 1:

         """Place this queen in board[i][col]"""
         board[i][col] = 1
         self.ld[i - col + N - 1] = self.rd[i + col] = self.cl[i] = 1

         """ recur to place rest of the queens """
         if self.solveNQUtil(board, col + 1):
            return True

         """ If placing queen in board[i][col]
         doesn't lead to a solution,
         then remove queen from board[i][col] """
         board[i][col] = 0  # BACKTRACK
         self.ld[i - col + N - 1] = self.rd[i + col] = self.cl[i] = 0

         """ If the queen cannot be placed in
         any row in this column col then return False """
   return False

""" This function solves the N Queen problem using
Backtracking. It mainly uses solveNQUtil() to
solve the problem. It returns False if queens
```

```
    cannot be placed, otherwise, return True and
    prints placement of queens in the form of 1s.
    Please note that there may be more than one
    solutions, this function prints one of the
    feasible solutions."""

    def solveNQ(self):
        board = [[0 for _ in range(N)] for __ in range(N)]
        if self.solveNQUtil(board, 0) == False:
            print("Solution does not exist")
            return False
        self.printSolution(board)
        return True


if __name__ == "__main__":
    N = 8

    NQBaB = NQBranchAndBond()
    NQBaB.solveNQ()

    NQBt = NQBacktracking()
    NQBt.solveNQ()
```