King Fahd University of Petroleum & Minerals

College of Computing and Mathematics

Information & Computer Science

ICS 381: Principles of Artificial Intelligence

Assignment #4

Student Name: Abdullah Alzeid

1) Function Breakdown

a) Square Class (Model Class)

1. **Properties**
    - **private int threats;** - Stores the number of threats (queens that can attack this square).
    - **private int placedQueen;** - Indicates whether a queen is placed on this square (**1** for yes, **0** for no).
    - **private int row;** - The row index of this square on the chessboard.
    - **private int col;** - The column index of this square on the chessboard.
2. **Constructor - Square(int row, int col, int t, int pq)**
    - Initializes a **Square** object with specified **row**, **col**, number of **threats** (t), and **placedQueen** status (pq).
3. **Accessors and Mutators**
    - **public int getRow()** - Returns the row index of the square.
    - **public int getCol()** - Returns the column index of the square.

- **public void setThreats(int threats)** - Sets the number of threats for the square.
- **public int getThreats()** - Returns the current number of threats for the square.
- **public void setPlacedQueen(int placedQueen)** - Sets the status of a queen being placed on this square.
- **public int getPlacedQueen()** - Returns the status of a queen placement on this square.

4. **toString() Method**
    - **public String toString()** - Overrides the **Object.toString()** method to return the string representation of the **placedQueen** property. This is used when printing the board to the console, showing **1** if a queen is placed and **0** otherwise.

b) Incremental Versions of the N-Queen

i) Backtracking

1. **Constructor - NQueenProblemBackTracking(int N)**
    - Initializes an instance of the class with a specific board size **N**.
    - Initializes three boolean arrays: **column**, **diagonal**, **antiDiagonal**, which are used to track whether a queen can be placed in a given column or diagonal without being attacked by another queen.
2. **printBoard(int board[][])**
    - Takes a 2-dimensional array **board** representing the chessboard.
    - Prints the chessboard to the console, where each queen is represented by a **1** and empty squares by **0**.
3. **isSafe(int row, int col)**
    - Determines if a queen can be safely placed at the given **row** and **column** without being attacked.

- Returns **true** if the position is safe (no other queens in the same column, diagonal, or anti-diagonal), otherwise **false**.

4. **solveNQ(int board[][], int row)**
    - This is a recursive function that tries to solve the N-Queens problem.
    - The base case is when **row == N**, meaning all queens are placed successfully, and it then prints the board.
    - If not, it iteratively tries to place a queen in every column of the current **row** and recurses to the next row.
    - Utilizes the **isSafe** function to check if the queen can be placed.
    - Backtracks if placing the queen does not lead to a solution, resetting the board and trying the next possibility.
5. **setupNQ(int startPos)**
    - Sets up the chessboard with an initial position for the first queen and attempts to solve the problem from there using **solveNQ**.
    - The **startPos** is where the first queen is placed on the board.
    - If no solution is found, it prints an error message.
6. **getCount()**

- Returns the **count** variable, which records the number of iterations (or recursive calls) made in the process of finding a solution.

7. **main(String args[])**
    - The entry point of the program where the user is prompted to input the size of the board (**N**) and the starting column for the first queen.
    - It creates an instance of **NQueenProblemBackTracking**, records the start time, calls **setupNQ** to find a solution, and then records the end time to calculate the duration.
    - Prints the number of iterations and the time taken to find a solution.

ii) Backtracking & Forward Checking

1. **Constructor - NQueenProblemForwardChecking (int N)**
    - Initializes an instance of the class with a specific board size **N**.
2. **printBoard(Square[][] board)**
    - Takes a 2-dimensional array **board** of **Square** objects representing the chessboard.
    - Prints the chessboard to the console. The **Square** object presumably contains information about the placement of queens and threats on the board.
3. **createThreats(Square[][] board, int row, int col)**
    - Updates the **board** to reflect the increase in threat count for squares that would be attacked by a queen placed at **row** and **col**.
    - Threats are added diagonally, vertically, and horizontally, indicating these squares are under attack and cannot have another queen placed in them.
4. **removeThreats(Square[][] board, int row, int col)**

- Reverses the process of **createThreats**, decreasing the threat count for squares that were under attack by a queen at **row** and **col**.
- Used when backtracking to undo the threats when a queen placement does not lead to a solution.

5. **solveNQ(Square board[][], int row, int startPos)**
    - A recursive function that attempts to place queens on the board in such a way that they do not threaten each other.
    - **row** is the current row on the board where the function tries to place a queen, and **startPos** is the column where the first queen was placed.
    - The function places a queen if there is no threat on the square (**board[row][i].getPlacedQueen() == 0** and **board[row][i].getThreats() == 0**) and then calls itself to solve the next row.
    - If placing a queen leads to a dead end, it backtracks by removing the queen and reducing the threats (**removeThreats**).

6. **setupNQ(int startPos)**
    - Sets up the initial state of the board with all squares initialized as non-threatened, non-queen squares and places the first queen at **startPos**.

- Calls **createThreats** to update the threats from the first queen.
- Calls **solveNQ** to begin solving the N-Queens problem with forward checking.

7. **main(String args[])**
    - The entry point of the program where the user inputs the size of the board (**N**) and the starting column for the first queen.
    - An instance of **NQueenProblemForwardChecking** is created, and the time to solve the problem is measured and printed along with the number of iterations.

iii) Backtracking & Forward Checking & LCV and MRV

1. **Constructor - NQueenProblemMRV_LCV(int N)**
    - Initializes an instance of the class with a specific board size **N**.
2. **printBoard(Square[][] board)**
    - Takes a 2-dimensional array **board** of **Square** objects representing the chessboard.
    - Prints the chessboard to the console. The **Square** object is assumed to contain information about the placement of queens and threats on the board.
3. **selectNextRowMRV(Square[][] board)**
    - Determines the next row to place a queen on using the MRV heuristic, which selects the row with the minimum number of available (threat-free) cells.
4. **getColumnsSortedByLCV(Square[][] board, int row)**
    - Returns a list of column indices sorted by the LCV heuristic, which orders the columns based

on the number of threats placing a queen would create. The goal is to choose a placement that imposes the least constraints on future placements.

5. **createThreats(Square[][] board, int row, int col)**
    - Increases the threat count for squares that would be attacked by a queen placed at **row** and **col**.
    - Threats are added diagonally, vertically, and horizontally, to indicate these squares are under attack.
6. **removeThreats(Square[][] board, int row, int col)**
    - Decreases the threat count for squares that were under attack by a queen at **row** and **col**.
    - Used for backtracking to undo the threat increments when a queen placement does not lead to a solution.
7. **solveNQ(Square board[][], int row)**
    - This is a recursive function tasked with solving the N-Queens problem by placing queens on the board in such a way that they do not threaten each other.
    - The function incorporates the MRV heuristic through the **selectNextRowMRV** method. MRV is applied to select the row

with the fewest legal moves (or the minimum remaining values) for placing the next queen. This approach minimizes the search space by choosing the most constrained row, thereby potentially reducing the overall number of recursive calls.

- The Least Constraining Value (LCV) heuristic is implemented via the **getColumnsSortedByLCV** method, which sorts the columns in a given row based on the number of new threats each potential placement would create. This ensures that the column chosen for placing the next queen minimizes the impact on the remaining empty squares.

- If a queen is successfully placed on the board, the function updates the threat counts on the board using the **createThreats** method.

- The function then recurses to the next row (or column, depending on the implementation) to attempt to place the next queen.

- If placing a queen in a certain position does not lead to a solution (i.e., it fails further down the recursive tree), the function performs backtracking by removing the

queen and the associated threats using the **removeThreats** method. This step undoes the changes made by **createThreats**, thereby resetting the board state for the next attempt.

- The process of selecting rows via MRV, choosing columns through LCV, placing queens, and then recursively continuing or backtracking as necessary is repeated until a solution is found or all possibilities are exhausted.

8. **setupNQ(int startPos)**
    - Initializes the board with **Square** objects and places the first queen at **startPos**.
    - Calls **createThreats** to update the board with threats from the first queen's placement.
    - Begins the solving process with **solveNQ**.
9. **main(String args[])**
    - The entry point of the program, where the user inputs the size of the board (**N**) and the starting column for the first queen.
    - Creates an instance of **NQueenProblemMRV_LCV**, measures the time to solve the problem, and prints the number of iterations and time taken.

c) Hill Climbing Version of the N-Queen (Bonus)

1. **Fields**
    - **private int[] board;** - An array representing the N-Queens board, where the index represents the column and the value at each index represents the row position of the queen in that column.
    - **private int N;** - The size of the board (and the number of queens).
    - **private static final int MAX_RETRIES;** - A constant that limits the number of retries or restarts in case the algorithm gets stuck.
2. **Constructor - NQueenHillClimbing(int N)**
    - Initializes the board size **N** and creates the board array.
3. **initializeBoard()**
    - Initializes the board with a random configuration of queens. It does this by shuffling a list of row numbers and placing each queen in a different row of each column to start.
4. **getConflicts(int[] state)**

- Calculates the number of conflicting pairs of queens on the board. Conflicts arise when two queens are in the same row or on the same diagonal.

5. **solve()**
    - This is the primary function of the class that tries to solve the N-Queens problem.
    - It initializes the board and then enters a loop where it repeatedly tries to reduce the number of conflicts by moving each queen to the position in its column that would result in the least number of conflicts.
    - If a better state is found (fewer conflicts), the board is updated to this state.
    - If the algorithm can't find a better state or reaches the maximum number of retries (**MAX_RETRIES**), it restarts with a new random configuration.
    - The algorithm terminates when a state with zero conflicts is found, which is a solution to the N-Queens problem.
6. **printBoard()**
    - Visualizes the board in the console, printing a **1** for squares with queens and a **0** for empty squares.
7. **main(String[] args)**
    - The entry point of the program, where it prompts the user for the board size **N**, creates an instance of

NQueenHillClimbing, and invokes the **solve()** method to find a solution.

Initial State Justification:-

The initial state of the board is created by generating a list of integers from 0 to N-1, where each integer represents a unique row index. This list is then shuffled, and the shuffled indices are assigned to the board array such that each index of the array corresponds to a column on the chessboard and the value at that index represents the row on which the queen is placed. This method ensures that in the initial state, each queen is placed in a different row and a different column.

This makes the runtime faster for the following reason:

Focused Search Space: Starting with one queen per row (and per column, by the nature of the array representation) eliminates any possibility of row or column conflicts initially. This significantly narrows down the search space from N^N to $N!$ because it rules out a large number of invalid states where queens share rows or columns. The smaller the search space, the fewer states the algorithm has to explore, which can lead to a faster runtime.

Fewer Initial Conflicts: Since each queen begins in a separate row and column, the only type of conflict we need to

resolve is the diagonal conflict. This reduces the complexity of the problem at the outset, allowing the hill-climbing algorithm to focus on minimizing diagonal conflicts right from the start. By not having to deal with row and column conflicts, the algorithm can make more meaningful progress with each step, which can result in reaching the solution state more quickly.

Tweaking Justification:-

The tweaking in the **solve** method involves generating neighbors of the current state where a neighbor is defined as a state that can be reached by moving one queen to another row in its column. The algorithm iterates through each column and moves the queen in that column to each row, one by one, calculating the number of conflicts (attacks) for each move. If a move results in fewer conflicts than the current state, that move is kept; otherwise, it's discarded.

This technique focuses on local optimization. By considering one queen (column) at a time, we reduce the problem's complexity at each step. We only keep changes that improve the state (reduce the number of conflicts). It's an efficient way to navigate the search space because it avoids considering all possible board configurations, which would be computationally infeasible for larger N.

Furthermore, this targeted approach allows the algorithm to quickly hone in on more promising areas of the search space. By moving one queen at a time and immediately checking for improvements, the algorithm can make steady progress towards a solution without the overhead of more complex

moves. This iterative improvement is likely to be faster than a more naive approach that doesn't use the heuristic of minimizing conflicts or that tries to move multiple queens simultaneously.

Heuristic Used:-

The heuristic used for the hill climbing algorithm is the number of pairwise conflicts between queens. A pairwise conflict occurs when two queens are placed such that they can attack each other according to the rules of chess: they are in the same row, column, or diagonal.

How does it Work:

1. **Conflict Calculation:** The heuristic function calculates the total number of pairwise conflicts on the board. This is done by checking every pair of queens and incrementing the conflict count if they are on the same row, same column, or on the same diagonal.
2. **Assessing Moves:** When considering a move (placing a queen in a new row within its column), the heuristic is recalculated for the new state. If the new state has fewer conflicts than the current state, it is considered a better state.
3. **Guiding the Search:** The heuristic guides the search process by evaluating the "goodness" of each state. States with fewer conflicts are preferred. The goal is to reach a state where the heuristic value is zero, meaning no pairwise conflicts and thus a solution to the N-Queens problem.

4. **Decision Making:** At each step, the heuristic is used to make a decision about which move to make. By preferring moves that result in a state with fewer conflicts, the algorithm is effectively climbing the hill of the search landscape where the peak (or in this case, the valley) represents the optimal solution with zero conflicts.

2) Results

Note: Columns of the board are indexed starting at 0

a) Backtracking

N Value	Starting Column	Runtime in Seconds
4	1	0.000010
5	4	0.000013
6	1	0.000007
7	6	0.000023
8	7	0.000055
9	8	0.000100
10	9	0.000074
11	10	0.000114
12	11	0.000054
13	12	0.000246
14	13	0.000186
15	14	0.000172
16	15	0.000176
17	16	0.002469
18	17	0.001025
19	18	0.009768
20	19	0.011081
21	20	0.033355
22	21	0.000585
23	22	0.156923
24	23	0.002291
25	24	0.039302
26	25	0.005937
27	26	0.096970
28	27	0.070813
29	28	0.261411
30	29	0.100859

Average Runtime: 0.02940807777777778 seconds

b) Backtracking & Forward Checking

N Value	Starting Column	Runtime in Seconds
4	1	0.000926
5	4	0.000050
6	1	0.000037
7	6	0.000397
8	7	0.000887
9	8	0.001753
10	9	0.000417
11	10	0.000272
12	11	0.000273
13	12	0.000627
14	13	0.001010
15	14	0.001136
16	15	0.001358
17	16	0.009659
18	17	0.003310
19	18	0.144606
20	19	0.005721
21	20	0.050030
22	21	0.000868
23	22	0.258402
24	23	0.004257
25	24	0.063521
26	25	0.008958
27	26	0.179962
28	27	0.124274
29	28	0.454292
30	29	0.169722

Average Runtime: 0.05506382592592593 seconds

c) Backtracking & Forward Checking & LCV and MRV

N Value	Starting Column	Runtime in Seconds
4	1	0.012403
5	4	0.000958
6	1	0.000913
7	6	0.001843
8	7	0.004715
9	8	0.006520
10	9	0.002643
11	10	0.006486
12	11	0.005893
13	12	0.010685
14	13	0.005565
15	14	0.000489
16	15	0.009723
17	16	0.016778
18	17	0.001556
19	18	0.008345
20	19	0.014790
21	20	0.011696
22	21	0.001020
23	22	0.006549
24	23	0.002522
25	24	0.002031
26	25	0.004523
27	26	0.029937
28	27	0.005043
29	28	0.002732
30	29	0.001933

Average Runtime: 0.006603370370370371 seconds

d) Hill Climbing (Bonus)

N Value	Runtime in Seconds
4	0.003202
5	0.000297
6	0.000624
7	0.000132
8	0.007324
9	0.002024
10	0.003148
11	0.000372
12	0.001231
13	0.005819
14	0.002451
15	0.002052
16	0.039271
17	0.107358
18	0.011100
19	0.106740
20	0.027919
21	0.032511
22	0.488639
23	0.009225
24	0.045547
25	0.235391
26	0.153326
27	0.142145
28	0.098623
29	0.334307
30	0.281101

Average Runtime: 0.0793286962962963 seconds

3) Comments

Backtracking Version

Overview:

- This version employs a straightforward backtracking algorithm. It places queens one by one in different columns, starting from the leftmost column. If a placement doesn't lead to a solution, it backtracks and tries the next possible placement.

Runtime Analysis:

- The runtimes for smaller N values ($N < 15$) are quite low, suggesting that the algorithm handles smaller instances of the problem efficiently.
- As N increases, the runtime increases exponentially, which is expected for backtracking algorithms due to the combinatorial nature of the problem.

- There is an irregular increase at $N=23$, which might indicate a particularly challenging configuration for the backtracking to resolve, or it could be an anomaly.

Performance Considerations:

- The average runtime is relatively low, but this is skewed by the very low runtimes for small N values.
- For larger N values, the performance drops significantly, showing the non-polynomial nature of the algorithm.

Backtracking & Forward Checking Version

Overview:

- This version adds forward checking to the basic backtracking algorithm. After placing a queen, it looks ahead to check if the next queen can be placed without any conflicts.

Runtime Analysis:

- The runtimes are generally lower compared to the plain backtracking version, indicating that forward checking successfully prunes the search space.
- There is a significant spike in runtime at N=23, similar to the backtracking version, suggesting a consistently difficult configuration that both algorithms struggle with.
- The average runtime is slightly higher than the basic backtracking, which may seem counterintuitive. This could be due to the overhead of the forward-checking process, which might not compensate for the reduced search space in every case.

Performance Considerations:

- The algorithm shows improved performance for some values of N but not consistently across the board.
- The overall trend suggests that as N increases, forward checking does not scale as well as one might hope.
- It’s worth noting that a potential reason for the negation of performance improvements of the forward checking version over the naïve backtracking is that the plain backtracking (which forward checking is built upon) versions assign the queens in

different columns by default, so when you choose a starting column the program automatically moves to the neighbor column and tries to assign a new queen there only keeping in mind row and diagonals constraints. So when integrating the forward checking we are gaining the overhead of managing the forward checking process but not gaining much benefit because we are already assigning queens smartly and eliminating conflicts without forward checking even interfering. So in a way we are comparing forward checking with a “superior” naïve backtracking.

Backtracking, Forward Checking, LCV, and MRV Version

Overview:

- This version integrates two heuristics: Least Constraining Value (LCV) and Most Remaining Values (MRV), along with forward checking.
- LCV chooses the next value that rules out the fewest choices for the remaining variables, and MRV selects the variable with the fewest legal values left.

Runtime Analysis:

- The runtimes are again lower than the basic backtracking for smaller values of N, suggesting that the heuristics are effective at reducing the search space.
- The performance seems to degrade less rapidly as N increases compared to the previous versions, especially notable in the N=20 to N=30 range.
- However, there are noticeable runtime spikes at N=17 and N=23, indicating potential inefficiencies or harder problem instances for this algorithm.

Performance Considerations:

- This version shows that the combination of LCV and MRV with forward checking has a more consistent performance benefit over a range of N values.
- The average runtime is lower than both the naïve backtracking and the forward checking versions, which suggests that the added heuristics contribute positively to the algorithm's efficiency.

Hill Climbing Version

Overview:

- Hill climbing is a local search algorithm that starts with an arbitrary solution and iteratively makes small changes to the solution to improve it.
- It doesn't guarantee a global optimum and can get stuck at local optima.

Runtime Analysis:

- The erratic nature of the runtimes, with some N values showing very low times and others extremely high, reflects the stochastic nature of the hill climbing approach.
- The sharp increases in runtime, particularly for N=22 and N=24, suggest that the algorithm encountered configurations that were difficult to optimize locally. This could be due to the initial state leading to a local maximum from which the algorithm couldn't escape.
- The average runtime is the highest among all versions. This is because unlike backtracking approaches, hill climbing does not go back to explore alternative paths when it hits a dead end. In a problem with a large search space like the N-Queen, this means hill climbing might miss the solution entirely and continue searching until a maximum iteration limit is reached, contributing to a higher average time.

Performance Considerations:

- Hill climbing's performance on the N-Queen problem is highly dependent on the initial configuration and the strategy for choosing the next move. It does not backtrack but instead moves only forward, which can lead to suboptimal performance.
- The algorithm can be fast for some instances where the initial state is already close to the global optimum and the path to the solution doesn't have many local optima.
- The lack of a systematic approach to explore the search space, unlike backtracking that methodically explores all possibilities, accounts for hill climbing's higher runtimes and variability.

General Observations

- **Scalability**: As N increases, the complexity of the N-Queen problem grows, and the performance of each algorithm version diverges noticeably. The backtracking-based versions show better scalability compared to the hill climbing version.
- **Algorithm Complexity**: The integration of heuristics in the backtracking algorithms shows varying degrees of effectiveness. While they can improve performance for some N values, they introduce additional computational overhead that can sometimes outweigh their benefits.
- **Heuristics Impact**: The use of LCV and MRV heuristics has demonstrated a more consistent improvement in runtimes across different N values, although this is not uniformly the case.
- **Local vs. Global Search**: The hill climbing algorithm, being a local search method, does not perform as well on the N-Queen problem, which requires a more global approach due to the possibility of numerous local optima.

4) Specification

Processor	Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz 2.59 GHz
Installed RAM	16.0 GB (15.8 GB usable)

5) Appendix

Backtracking:-

```
package nQueens;

import java.util.Scanner;

public class NQueenProblemBackTracking {
    private int N;
    private int count = 0;
    private boolean[] column, diagonal, antiDiagonal;

    public NQueenProblemBackTracking(int N) {
        this.N = N;
        column = new boolean[N];
        diagonal = new boolean[2 * N];
        antiDiagonal = new boolean[2 * N];
    }

    void printBoard(int board[][]) {
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++)
                System.out.print(" " + board[i][j] + " ");
            System.out.println();
        }
        System.out.println("");
        System.out.println("---------------------------------------");
        System.out.println("");
    }

    boolean isSafe(int row, int col) {
        return !column[col] && !diagonal[row + col] && !antiDiagonal[row -
col + N - 1];
    }

    boolean solveNQ(int board[][], int row) {
        if (row == N) {
            System.out.println("Solution:");
            printBoard(board);
            return true;
        }
        for (int i = 0; i < N; i++) {
            count++;
            if (isSafe(row, i)) {
                board[row][i] = 1;
                column[i] = diagonal[row + i] = antiDiagonal[row - i + N - 1]
= true;

//                printBoard(board);
                if (solveNQ(board, row + 1)) {
                    return true;
```

```
                }

                board[row][i] = 0;
                column[i] = diagonal[row + i] = antiDiagonal[row - i + N - 1]
= false;
                count--;
//                printBoard(board);
            }
        }
        return false;
    }

    boolean setupNQ(int startPos) {
        int[][] board = new int[N][N];
        board[0][startPos] = 1;
        column[startPos] = diagonal[startPos] = antiDiagonal[-startPos + N -
1] = true;
        if (solveNQ(board, 1) == false) {
            System.out.print("Error: Can't find solution.");
            System.out.println("");
            return false;
        }
        return true;
    }

    public int getCount() {
        return count;
    }

    public static void main(String args[]) {

        Scanner scan = new Scanner(System.in);

        System.out.println("Enter the board size N: ");
        int N = scan.nextInt();

        System.out.println("Please enter starting column for Q1 (0-" + (N -
1) + "): ");
        int startingColumn = scan.nextInt();

        NQueenProblemBackTracking Queen = new NQueenProblemBackTracking(N);

        long startTime = System.nanoTime();

        System.out.println("\nFinding a Solution . . . \n");
        boolean solved = Queen.setupNQ(startingColumn);

        long endTime = System.nanoTime();

        double duration = (double) (endTime - startTime) / 1_000_000_000.0;


        System.out.println("Iterations: " + Queen.getCount());
        System.out.println("Time taken: " + duration + " seconds");

//        // Array of N values and starting columns as provided in the image
//        int[] nValues = {4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,
```

```
18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30};
//        int[] startingColumns = {1, 4, 1, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29};
//
//        double totalTime = 0; // To sum up the total time for calculating
the average
//        int totalIterations = 0; // To sum up the total iterations for all
runs
//
//        System.out.printf("%-10s %-15s %-15s\n", "N Value", "Starting
Column", "Runtime in Seconds");
//
//        for (int i = 0; i < nValues.length; i++) {
//            int N = nValues[i];
//            int startingColumn = startingColumns[i];
//            NQueenProblemBackTracking Queen = new
NQueenProblemBackTracking(N);
//
//            long startTime = System.nanoTime();
//            Queen.setupNQ(startingColumn);
//            long endTime = System.nanoTime();
//
//            double duration = (double)(endTime - startTime) /
1_000_000_000.0;
//            totalTime += duration;
//            totalIterations += Queen.getCount();
//
//            System.out.printf("%-10d %-15d %-15f\n", N, startingColumn,
duration);
//        }
//
//        double averageTime = totalTime / nValues.length;
//        int averageIterations = totalIterations / nValues.length;
//        System.out.println("\nAverage Runtime: " + averageTime + "
seconds");
//        System.out.println("Average Iterations: " + averageIterations);
    }
}
```

Backtracking & Forward Checking:-

```
package nQueens;

import java.util.Scanner;

public class NQueenProblemForwardChecking {

   private int N;
   private static int count = 0;

   public NQueenProblemForwardChecking(int N) {
      this.N = N;
   }

   void printBoard(Square[][] board) {
      for (int i = 0; i < N; i++) {
         for (int j = 0; j < N; j++)
            System.out.print(" " + board[i][j] + " ");
         System.out.println();
      }
      System.out.println("");
      System.out.println("----------------------------------------");
      System.out.println("");
   }

   void createThreats(Square[][] board, int row, int col) {

      for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) {
         board[i][j].setThreats(board[i][j].getThreats() + 1);
      }

      for (int i = row, j = col; j >= 0 && i < N; i++, j--) {
         board[i][j].setThreats(board[i][j].getThreats() + 1);
      }

      for (int i = row, j = col; i >= 0 && j < N; i--, j++) {
         board[i][j].setThreats(board[i][j].getThreats() + 1);
      }

      for (int i = row, j = col; i < N && j < N; i++, j++) {
         board[i][j].setThreats(board[i][j].getThreats() + 1);
      }

      for (int i = 0; i < N; i++) {
         board[row][i].setThreats(board[row][i].getThreats() + 1);
      }

      for (int i = 0; i < N; i++) {
         board[i][col].setThreats(board[i][col].getThreats() + 1);
      }
   }

   void removeThreats(Square[][] board, int row, int col) {

      for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) {
```

```
         board[i][j].setThreats(board[i][j].getThreats()-1);
      }

      for (int i = row, j = col; j >= 0 && i < N; i++, j--) {
         board[i][j].setThreats(board[i][j].getThreats()-1);
      }

      for (int i = row, j = col; i >= 0 && j < N; i--, j++) {
         board[i][j].setThreats(board[i][j].getThreats()-1);
      }

      for (int i = row, j = col; i < N  && j < N; i++, j++) {
         board[i][j].setThreats(board[i][j].getThreats()-1);
      }

      for (int i = 0; i < N; i++) {
         board[row][i].setThreats(board[row][i].getThreats()-1);
      }

      for (int i = 0; i < N; i++) {
         board[i][col].setThreats(board[i][col].getThreats()-1);
      }

   }

   boolean solveNQ(Square board[][], int row, int startPos) {

      if (row == N) {
         return true;
      }
      if (row == 0) {
         row++;
      }

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

         if (board[row][i].getPlacedQueen() == 0 &&
board[row][i].getThreats() == 0)  {
            count++;

            board[row][i].setPlacedQueen(1);
            createThreats(board,row,i);

            if (solveNQ(board, row + 1, startPos) == true)
               return true;

            // backtrack
            board[row][i].setPlacedQueen(0);
            removeThreats(board,row,i);
         }
      }
      return false;
   }

   boolean setupNQ(int startPos) {
      Square[][] board = new Square[N][N];
      for (int i = 0; i < N; i++) {
```

```
         for (int j = 0; j < N; j++) {
            Square s1 = new Square(i,j,0, 0);
            board[i][j] = s1;
         }
      }
      board[0][startPos].setPlacedQueen(1);
      createThreats(board, 0, startPos);

      if (solveNQ(board, 0, startPos) == false) {
         System.out.print("Error: Can't find solution.");
         System.out.println("");
         return false;
      }

      System.out.println("Solution:");
      printBoard(board);

      return true;
   }

   public static void main(String args[]) {
      Scanner scan = new Scanner(System.in);
      System.out.println("Enter the board size N: ");
      int N = scan.nextInt();
      System.out.println("Please enter starting column for Q1 (0-" + (N-1) +
"): ");
      int startingColumn = scan.nextInt();

      NQueenProblemForwardChecking Queen = new
NQueenProblemForwardChecking(N);

      long startTime = System.nanoTime();

      System.out.println("\nFinding a Solution . . . \n");
      Queen.setupNQ(startingColumn);

      long endTime = System.nanoTime();

      double duration = (double)(endTime - startTime) / 1_000_000_000.0;

      System.out.println("Iterations: " + count);
      System.out.println("Time taken: " + duration + " seconds");

//    // Array of N values and starting columns as provided in the image
//    int[] nValues = {4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30};
//    int[] startingColumns = {1, 4, 1, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29};
//
//    double totalTime = 0; // To sum up the total time for calculating the
average
//    System.out.printf("%-10s %-15s %-15s\n", "N Value", "Starting Column",
"Runtime in Seconds");
//
//    for (int i = 0; i < nValues.length; i++) {
//       int N = nValues[i];
//       int startingColumn = startingColumns[i];
```

```
//       NQueenProblemForwardChecking Queen = new
NQueenProblemForwardChecking(N);
//
//       long startTime = System.nanoTime();
//       Queen.setupNQ(startingColumn);
//       long endTime = System.nanoTime();
//
//       double duration = (double)(endTime - startTime) / 1_000_000_000.0;
//       totalTime += duration;
//
//       System.out.printf("%-10d %-15d %-15f\n", N, startingColumn,
duration);
//    }
//
//    double averageTime = totalTime / nValues.length;
//    System.out.println("\nAverage Runtime: " + averageTime + " seconds");
   }
   }
```

Backtracking & Forward Checking & LCV and MRV:-

```
package nQueens;
import java.util.*;
import java.util.stream.*;

public class NQueenProblemMRV_LCV {

    private int N;
    private static int count = 0;

    public NQueenProblemMRV_LCV(int N) {
        this.N = N;
    }

    void printBoard(Square[][] board) {
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++)
                System.out.print(" " + board[i][j] + " ");
            System.out.println();
        }
        System.out.println("");
        System.out.println("---------------------------------------");
        System.out.println("");
    }

    int selectNextRowMRV(Square[][] board) {
        int minThreats = Integer.MAX_VALUE;
        int rowIndex = -1;
        for (int i = 0; i < N; i++) {
            final int currentRow = i;
            int availableCells = (int) IntStream.range(0, N).filter(j ->
board[currentRow][j].getThreats() == 0).count();
            if (availableCells < minThreats && availableCells > 0) {
                minThreats = availableCells;
                rowIndex = i;
            }
        }
        return rowIndex;
    }


    List<Integer> getColumnsSortedByLCV(Square[][] board, int row) {
        return IntStream.range(0, N)
                .boxed()
                .filter(col -> board[row][col].getThreats() == 0)
                .sorted(Comparator.comparingInt(col -> {
                    board[row][col].setPlacedQueen(1);
                    int threatsCreated = (int) IntStream.range(0, N).filter(j
-> board[row][j].getThreats() > 0).count();
                    board[row][col].setPlacedQueen(0);
                    return threatsCreated;
                }))
                .collect(Collectors.toList());
```

```
    }


    void createThreats(Square[][] board, int row, int col) {

        for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) {
            board[i][j].setThreats(board[i][j].getThreats() + 1);
        }

        for (int i = row, j = col; j >= 0 && i < N; i++, j--) {
            board[i][j].setThreats(board[i][j].getThreats() + 1);
        }

        for (int i = row, j = col; i >= 0 && j < N; i--, j++) {
            board[i][j].setThreats(board[i][j].getThreats() + 1);
        }

        for (int i = row, j = col; i < N && j < N; i++, j++) {
            board[i][j].setThreats(board[i][j].getThreats() + 1);
        }

        for (int i = 0; i < N; i++) {
            board[row][i].setThreats(board[row][i].getThreats() + 1);
        }

        for (int i = 0; i < N; i++) {
            board[i][col].setThreats(board[i][col].getThreats() + 1);
        }
    }

    void removeThreats(Square[][] board, int row, int col) {

        for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) {
            board[i][j].setThreats(board[i][j].getThreats()-1);
        }

        for (int i = row, j = col; j >= 0 && i < N; i++, j--) {
            board[i][j].setThreats(board[i][j].getThreats()-1);
        }

        for (int i = row, j = col; i >= 0 && j < N; i--, j++) {
            board[i][j].setThreats(board[i][j].getThreats()-1);
        }

        for (int i = row, j = col; i < N  && j < N; i++, j++) {
            board[i][j].setThreats(board[i][j].getThreats()-1);
        }

        for (int i = 0; i < N; i++) {
            board[row][i].setThreats(board[row][i].getThreats()-1);
        }

        for (int i = 0; i < N; i++) {
            board[i][col].setThreats(board[i][col].getThreats()-1);
        }

    }
```

```
    boolean solveNQ(Square[][] board, int col) {
        if (col >= N) {
//            System.out.println("Solution:");
//            printBoard(board);
            return true;
        }


        int row = selectNextRowMRV(board);
        if (row == -1) {
            return false;
        }

        List<Integer> lcvColumns = getColumnsSortedByLCV(board, row);
        for (int nextCol : lcvColumns) {
            if (board[row][nextCol].getPlacedQueen() == 0 &&
board[row][nextCol].getThreats() == 0) {
                count++;
                board[row][nextCol].setPlacedQueen(1);
                createThreats(board, row, nextCol);
//                printBoard(board);
                if (solveNQ(board, col + 1)) {
                    return true;
                }
                board[row][nextCol].setPlacedQueen(0);
                removeThreats(board, row, nextCol);
            }
        }
        return false;
    }


    boolean setupNQ(int startPos) {
        Square[][] board = new Square[N][N];
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                Square s1 = new Square(i, j, 0, 0);
                board[i][j] = s1;
            }
        }
        board[0][startPos].setPlacedQueen(1);
        createThreats(board, 0, startPos);

        if (solveNQ(board, 1) == false) {
            System.out.print("Error: Can't find solution.");
            System.out.println("");
            return false;
        }

        return true;
    }

    public static void main(String args[]) {
//        Scanner scan = new Scanner(System.in);
//        System.out.println("Enter the board size N: ");
//        int N = scan.nextInt();
```

```
//         System.out.println("Please enter starting column for Q1 (0-" + (N-
1) + "): ");
//         int startingColumn = scan.nextInt();
//
//         NQueenProblemMRV_LCV Queen = new NQueenProblemMRV_LCV(N);
//
//         long startTime = System.nanoTime();
//
//         System.out.println("\nFinding a Solution . . . \n");
//         Queen.setupNQ(startingColumn);
//
//         long endTime = System.nanoTime();
//
//         double duration = (double)(endTime - startTime) / 1_000_000_000.0;
//
//         System.out.println("Iterations: " + count);
//         System.out.println("Time taken: " + duration + " seconds");

        int[] nValues = {4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,
18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30};
        int[] startingColumns = {1, 4, 1, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29};

        double totalTime = 0; // To sum up the total time for calculating the
average
        System.out.printf("%-10s %-15s %-15s\n", "N Value", "Starting
Column", "Runtime in Seconds");

        for (int i = 0; i < nValues.length; i++) {
            int N = nValues[i];
            int startingColumn = startingColumns[i];
            NQueenProblemMRV_LCV Queen = new NQueenProblemMRV_LCV(N);

            long startTime = System.nanoTime();
            Queen.setupNQ(startingColumn);
            long endTime = System.nanoTime();

            double duration = (double)(endTime - startTime) /
1_000_000_000.0;
            totalTime += duration;

            System.out.printf("%-10d %-15d %-15f\n", N, startingColumn,
duration);
        }

        double averageTime = totalTime / nValues.length;
        System.out.println("\nAverage Runtime: " + averageTime + " seconds");
    }
}
```

Hill Climbing:-

```
package nQueens;

import java.util.Arrays;
import java.util.Collections;
import java.util.List;
import java.util.Scanner;
import java.util.stream.IntStream;
import java.util.stream.Collectors;

public class NQueenHillClimbing {

    private int[] board;
    private int N;
    private static final int MAX_RETRIES = 100;

    public NQueenHillClimbing(int N) {
        this.N = N;
        board = new int[N];
    }

    private void initializeBoard() {
        List<Integer> rows = IntStream.range(0,
N).boxed().collect(Collectors.toList());
        Collections.shuffle(rows);
        for (int i = 0; i < N; i++) {
            board[i] = rows.get(i);
        }
    }

    private int getConflicts(int[] state) {
        int conflicts = 0;
        for (int i = 0; i < state.length; i++) {
            for (int j = i + 1; j < state.length; j++) {
                if (state[i] == state[j] || Math.abs(state[i] - state[j]) ==
j - i) {
                    conflicts++;
                }
            }
        }
        return conflicts;
    }

    public void solve() {
        System.out.println("Initial state:");
        initializeBoard();
        printBoard();
        System.out.println("\nFinding a Solution . . . \n");

        int currentConflicts = getConflicts(board);
        int retries = 0;

        while (currentConflicts != 0) {
            if (retries > MAX_RETRIES) {
                System.out.println("No Solution is Found");
```

```
                return;
            }

            int[] nextState = Arrays.copyOf(board, board.length);
            int minConflicts = Integer.MAX_VALUE;

            for (int col = 0; col < N; col++) {
                int originalRow = nextState[col];
                for (int row = 0; row < N; row++) {
                    if (row == originalRow) continue;

                    nextState[col] = row;
                    int newConflicts = getConflicts(nextState);

                    if (newConflicts < minConflicts) {
                        minConflicts = newConflicts;
                        board[col] = row;
                    }
                }
                nextState[col] = originalRow;
            }

            if (minConflicts >= currentConflicts) {
                retries++;
                initializeBoard();
            } else {
                retries = 0;
            }

            currentConflicts = getConflicts(board);
        }
        System.out.println("Solution:");
        printBoard();
    }

    public void printBoard() {
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                if (board[j] == i) {
                    System.out.print(" 1 ");
                } else {
                    System.out.print(" 0 ");
                }
            }
            System.out.println();
        }
        System.out.println("----------------------------------------");
    }

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the board size N: ");
        int N = scanner.nextInt();
        if (N < 4) {
            System.out.println("There are no solutions for N less than 4.");
```

```
            return;
        }

        NQueenHillClimbing solver = new NQueenHillClimbing(N);

        long startTime = System.nanoTime();
        solver.solve();
        long endTime = System.nanoTime();

        double duration = (double) (endTime - startTime) / 1_000_000_000.0;

        System.out.println("Runtime:" + duration + " seconds");

//        int[] nValues = IntStream.rangeClosed(4, 30).toArray();
//
//        double totalTime = 0;
//        int totalSolutionsFound = 0;
//
//        System.out.printf("%-10s %-20s\n", "N Value", "Runtime in
Seconds");
//
//        for (int N : nValues) {
//            NQueenHillClimbing solver = new NQueenHillClimbing(N);
//
//            long startTime = System.nanoTime();
//            solver.solve();
//            long endTime = System.nanoTime();
//
//            double duration = (double) (endTime - startTime) /
1_000_000_000.0;
//            totalTime += duration;
//
//            System.out.printf("%-10d %-20f\n", N, duration);
//
//
//
//        }
//
//        double averageTime = totalTime / nValues.length;
//        System.out.println("\nAverage Runtime: " + averageTime + "
seconds");

    }
}
```

References:-

https://github.com/RyanBouffard/NQueens