Solving N-Queen Problem with Hill Climbing Search

Name: Soham Dhodapkar
ID: 801075881
Name: Jitesh Golatkar
ID: 801073392

What is N-Queen problem?
The N-Queens puzzle or N-Queens problem is the problem of placing n queens on a *nxn* board (assume chessboard) such that no two queens attack each other, or share the same spot. Two queens attack each other if they can directly see a queen horizontally, vertically or diagonally. Goal state is achieved when all the queens (n) are placed on the board without any queen attacking any other queen.

Hill-Climbing Algorithm:
Hill-Climbing algorithm is a local search algorithm which continuously moves in the direction to find the peak or best possible solution to the problem. It reaches a peak value when none of its neighbor has a better value. It keeps on doing so until it finds the global maximum. That being said, it can get stuck in various situations like local maximum (may think this is the best solution), flat local minimum (all neighboring values are same) and a shoulder (All neighbors have same value but there is a chance of having a better value).

Problem Formulation:
States: A configuration representing n queens on *nxn* board. One column, one queen (imagine a matrix)
Actions: Add queens, move one queen at a time to check collisions and next possible states
Goal: n-queens on the board, none of which can attack each other.

Variants of Hill-Climbing Search Algorithm:

1. Steepest Ascent Hill-Climbing: Here heuristic is calculated by taking the state which lowest number of attacks. If there are multiple states with same number of attacks then these are selected randomly. It checks if the current configuration is viable (no attacks) else it proceeds to calculating such a configuration by generating lower heuristics. A state denoting 0 heuristic closes the solution. We can get 14% success rate and 86% failure rate with this approach.

2. Hill-Climbing with sideways moves: When the system gets stuck, sideways moves make sure that the position is a shoulder and not a local maximum. Sideways loop is implemented to find the best neighbor. If it is indeed a local maximum then the program ends. the success of finding the goal state increases from 14% to 94%.

3. Random restart without sideways moves: Random restart can overcome the problem of local maximum. Algorithm loops till it finds a state better than current, where number of attacks is lesser than the current. If we get a state equal to the current then it restarts from a random position. For 8-Queen we can get solution in 1 restart theoretically.

4. Random restart with sideways moves: When the limit of sideways moves is reached this algorithm will randomly restart. For 8-Queen we can get solution in 1 restart theoretically.

Program Structure:

Board class: This class implements supporting functions to all the main implementations like generating a board, calculations of heuristics and generating a matrix of heuristic values for all possible cases on the board.

SteepestAscentHillClimbing class: Implements the steepest ascent variant of hill climbing search. Input is taken as the number of queens and number of iterations to run. Function climb() recursively calls itself if the current heuristic is not the solution. If no successor has a heuristic less than current then it fails. It will report number of steps in success and failure and success and failure rates.

HillClimbingSideways class: Implements hill climbing with sideways moves. Input is taken as the number of queens and number of iterations to run. It chooses any random queen configuration and checks if that configuration is solution or not. If it is not solution, then function recursively calls itself to generate next possible successor. If flat or shoulder local minimum is reached, then algorithm allows sideways moves up to 100 in order to reach solution. It will report number of steps in success and failure and success and failure rates.

RandomRestartWithoutSideways class: Implements hill climbing with random restart-without sideways moves. Input is taken as the number of queens and number of iterations to run. It chooses any random queen configuration and checks if that configuration is solution or not. If it is not solution, then function recursively calls itself to get next possible successors. If no successor has heuristic less than current heuristic, then algorithm will restart and proceed till it finds the solution. It will report number of random restarts and number of steps.

RandomRestartWithSideways class: Implements hill climbing with random restart-with sideways moves. Input is taken as the number of queens and number of iterations to run. It chooses any random queen configuration and checks if that configuration is the solution or not. If it is not the solution, then function recursively calls itself to get next possible successors. If a flat or shoulder or a local minimum is reached, then algorithm allows sideways moves up to 100 in order to reach the solution. If sideways count is greater than 100, then the algorithm restarts till it finds solution. It reports number of random restarts and the number of steps.

Solver class: Solver contains the main function to run the program. On execution, it prompts the user to input the number of queens and iterations. It also prompts for the choice of algorithm to run. Based on the choice it executes corresponding algorithm.

Source Code:

Board.java

```
import java.util.Random;

public class Board {

	/**
	 * Create a random board
	 *
	 * @param N
	 * @return
	 */
	public static int[] createBoard(int N) {
		int[] newPosition = new int[N];
		Random random = new Random();

		for (int i = 0; i < N; i++) {
			newPosition[i] = random.nextInt(N);
		}
		return newPosition;
	}

	/**
	 * Get the heuristic value of the current state
	 *
	 * @param board
	 * @return
	 */
	public static int getHeuristic(int[] board) {
		int heuristic = 0;
		for (int i = 0; i < board.length; i++) {
			int temp = board[i];
			for (int j = i + 1; j < board.length; j++) {
				if (temp == board[j] || Math.abs(i - j) == Math.abs(temp - board[j])) {
					heuristic++;
				}
			}
		}
		return heuristic;
	}

	/**
	 * Generate matrix of heuristic values
	 *
	 * @param queen
	 * @return
```

```
	 */
	public static int[][] getHMatrix(int[] queen) {
		int n = queen.length;
		int[] temp = new int[n];
		int[][] mat = new int[n][n];
		for (int i = 0; i < n; i++) {
			for (int j = 0; j < n; j++) {
				temp[j] = queen[j];
			}
			for (int p = 0; p < n; p++) {
				for (int k = 0; k < n; k++) {
					if (i == k) {
						temp[k] = p;
					}
				}
				mat[p][i] = getHeuristic(temp);
			}
		}
		return mat;
	}

	public static void printBoard(int[] queens, int N) {
		int[][] board = new int[N][N];
		for (int column = 0; column < N; column++) {
			board[queens[column]][column] = 1;
		}
		for (int row = 0; row < N; row++) {
			for (int column = 0; column < N; column++) {
				if (board[row][column] == 1) {
					System.out.print("Q\t");
				} else {
					System.out.print(board[row][column] + "\t");
				}
			}
			System.out.println();
		}

	}

}
```

Solver.java

```
import java.util.Scanner;

public class Solver {

	public static void main(String[] args) {

		int N, iterations = 1;
		int[] initial;
		int ch;
		String cont;
		Scanner sc = new Scanner(System.in);
		do {
			System.out.println("Enter number of N (Must be greater than 3):");
			N = sc.nextInt();
			System.out.println("Enter number of iteration: ");
			iterations = sc.nextInt();
			System.out.println("Enter Choice: \n1: Steepest Ascent Hill Climbing\n2:
Steepest Hill Climbing With Sideways"+ "\n3: Random Restart Without Sideways\n4: Random Restart
With Sideways");
			ch = sc.nextInt();

			switch (ch) {
			case 1:
				for (int i = 0; i < iterations; i++) {
					initial = Board.createBoard(N);
					SteepestAscentHillClimbing.climb(initial);
				}
				SteepestAscentHillClimbing.printStatistics(iterations);
				break;
			case 2:
				for (int i = 0; i < iterations; i++) {
					initial = Board.createBoard(N);
					HillClimbingSideways.climb(initial);
				}
				HillClimbingSideways.printStatistics(iterations);
				break;
			case 3:
				for (int i = 0; i < iterations; i++) {
					initial = Board.createBoard(N);
					RandomRestartWithoutSideways.climb(initial);
				}
				RandomRestartWithoutSideways.printStatistics(iterations);
				break;
			case 4:
				for (int i = 0; i < iterations; i++) {
					initial = Board.createBoard(N);
```

```
                                        RandomRestartWithSideways.climb(initial);
                                }
                                RandomRestartWithSideways.printStatistics(iterations);
                                break;
                        }
                        System.out.println("Run another variant? y/n");
                        cont = sc.next();
                } while (cont.equalsIgnoreCase("y"));
                sc.close();
        }
}
```

SteepestAscentHillClimbing.java

```
import java.util.HashMap;
import java.util.Random;
public class SteepestAscentHillClimbing {
	static int success = 0;
	static int steps = 0;
	static int successSteps = 0;
	static int failSteps = 0;
	public static void printStatistics(int iterations) {
		float successRate;
		successRate = (float) success * 100 / (float) iterations;
		System.out.println("Success rate: " + successRate + "%");
		System.out.println("Failure rate: " + (100 - successRate) + "%");
		float successStepCount = ((float) successSteps / (float) success);
		float failureStepCount = ((float) failSteps / (float) (iterations - success));
		System.out.println("Average steps in success: " + successStepCount);
		System.out.println("Average steps in failure: " + failureStepCount);
	}
	public static void climb(int[] queens) {
		int c, j, N = queens.length;
		int[] temp = new int[N];
		int[][] mat = new int[N][N];
		HashMap<Integer, int[]> map = new HashMap<Integer, int[]>();
		int bestSoFar = Board.getHeuristic(queens);
```

```
mat = Board.getHMatrix(queens);
int min = mat[0][0];
for (c = 0; c < N; c++) {
        for (int r = 0; r < N; r++) {
                if (min > mat[c][r]) {
                        min = mat[c][r];
                }
        }
}
int count = 1;
for (c = 0; c < N; c++) {
        for (j = 0; j < N; j++) {
                for (int k = 0; k < N; k++) {
                        temp[k] = queens[k];
                }
                if (mat[c][j] == min) {
                        int[] nextBest = new int[N];
                        for (int k = 0; k < N; k++) {
                                nextBest[k] = temp[k];
                        }
                        nextBest[j] = c;
                        map.put(count, nextBest);
                        count++;
                }
        }
```

```
                }
                Random rand = new Random();
                int index = rand.nextInt(count);
                if (index == 0) {
                        index = index + 1;
                }
                temp = map.get(index);
                if (min == 0) {
                        System.out.println("Solution Found");
                        Board.printBoard(temp, temp.length);
                        System.out.println("Conflicts: " + Board.getHeuristic(temp));
                        steps++;
                        success++;
                        successSteps = successSteps + steps;
                        steps = 0;
                } else if (min < bestSoFar) {
                        steps++;
                        climb(temp);
                } else if (min == bestSoFar) {
                        failSteps = failSteps + steps;
                        steps = 0;
                }
        }
}
```

HillClimbingSideways.java

```
import java.util.*;

public class HillClimbingSideways {
	static int success = 0;
	static int steps = 0;
	static int successSteps = 0;
	static int failSteps = 0;
	public static int sCount = 0;

	public static void printStatistics(int iterations) {
		float rateOfSuccess;
		rateOfSuccess = (float) success * 100 / (float) iterations;
		System.out.println("Success rate: " + rateOfSuccess + "%");
		System.out.println("Failure rate: " + (100 - rateOfSuccess) + "%");
		float successStepCount = ((float) successSteps / (float) success);
		float failureStepCount = ((float) failSteps / (float) (iterations - success));

		System.out.println("Average steps in success: " + successStepCount);
		System.out.println("Average steps in failure: " + failureStepCount);
	}

	public static void climb(int[] queens) {
		int c, j, N = queens.length;
		int[] temp = new int[N];
		int[][] mat = new int[N][N];
		HashMap<Integer, int[]> map = new HashMap<Integer, int[]>();
		int bestSoFar = Board.getHeuristic(queens);
		mat = Board.getHMatrix(queens);
		int min = mat[0][0];
		for (c = 0; c < N; c++) {
			for (int r = 0; r < N; r++) {
				if (min > mat[c][r]) {
					min = mat[c][r];
				}
			}
		}
		int count = 0;
		for (c = 0; c < N; c++) {
			for (j = 0; j < N; j++) {
				for (int k = 0; k < N; k++) {
					temp[k] = queens[k];
				}
				if (mat[c][j] == min) {
					int[] nextBest = new int[N];
					for (int k = 0; k < N; k++) {
						nextBest[k] = temp[k];
```

```
                                        }
                                        nextBest[j] = c;
                                        if (!(Arrays.equals(nextBest, queens))) {
                                                map.put(count, nextBest);
                                                count++;
                                        }
                                }

                        }
                }
                if (count > 0) {
                        Random rand = new Random();
                        int index = rand.nextInt(count);

                        temp = map.get(index);

                        if (min == 0) {
                                System.out.println("Solution Found");
                                Board.printBoard(temp, temp.length);
                                System.out.println("Conflicts : " + Board.getHeuristic(temp));
                                steps++;
                                success++;
                                successSteps = successSteps + steps;
                                steps = 0;
                        } else if (min < bestSoFar) {

                                steps++;
                                sCount = 0;
                                climb(temp);
                        } else if (min == bestSoFar) {

                                if (sCount > 99) {
                                        System.out.println("Solution not found. ");
                                        failSteps = failSteps + steps;
                                        steps = 0;
                                } else {
                                System.out.println("Allowing the sideways move number " + sCount);
                                        sCount++;
                                        steps++;
                                        climb(temp);
                                }
                        }
                } else {
                        failSteps = failSteps + steps;
                        steps = 0;
                }
        }
}
```

RandomRestartWithSideways.java

```
import java.util.Arrays;
import java.util.HashMap;
import java.util.Random;
public class RandomRestartWithSideways {
	public static int count = 1;
	public static int steps = 0;
	public static int sCount = 0;
	public static void printStatistics(int iterations) {
		float avgRestart = ((float) count / (float) iterations);
		float avgsteps = ((float) steps / (float) iterations);
		System.out.println("The average number of restarts:  " + (avgRestart));
		System.out.println("The average number of steps:  " + avgsteps);
	}
	public static void climb(int[] queens) {
		int c, j, N = queens.length;
		int[] temp = new int[N];
		int[][] mat = new int[N][N];
		Board.printBoard(queens, N);
		int bestSoFar = Board.getHeuristic(queens);
		HashMap<Integer, int[]> map = new HashMap<Integer, int[]>();
		mat = Board.getHMatrix(queens);
		int min = mat[0][0];
		for (c = 0; c < N; c++) {
			for (int r = 0; r < N; r++) {
```

```
                    if (min > mat[c][r]) {
                        min = mat[c][r];
                    }
                }
            }
            int count = 0;
            for (c = 0; c < N; c++) {
                for (j = 0; j < N; j++) {
                    for (int k = 0; k < N; k++) {
                        temp[k] = queens[k];
                    }
                    if (mat[c][j] == min) {
                        int[] nextBest = new int[N];

                        for (int k = 0; k < N; k++) {
                            nextBest[k] = temp[k];
                        }
                        nextBest[j] = c;
                        {
                            if (!(Arrays.equals(nextBest, queens))) {
                                map.put(count, nextBest);
                                count++;
                            }
                        }
                    }
```

```
            }
}
if (count > 0) {
            Random rand = new Random();
            int index = rand.nextInt(count);
            temp = map.get(index);
            Board.printBoard(temp, temp.length);

            if (min == 0) {
                        System.out.println("Solution Found");
                        Board.printBoard(temp, temp.length);
                        System.out.println("Conflicts: " + Board.getHeuristic(temp));
                        steps++;
            } else if (min < bestSoFar) {
                        steps++;
                        sCount = 0;
                        climb(temp);
                        bestSoFar = min;
            } else if (min == bestSoFar) {

                        if (sCount > 99) {
                                    System.out.println("Doing random restart.");
                                    count++;
                                    climb(Board.createBoard(N));
```

```
                        } else {
                        System.out.println("Allowing the sideways move number " + sCount);
                                steps++;
                                sCount++;
                                climb(temp);
                        }
                }
        } else {
                System.out.println("Doing random restart.");
                count++;
                climb(Board.createBoard(N));
        }
    }
}
```

RandomRestartWithoutSideways.java

```
import java.util.HashMap;
import java.util.Random;

public class RandomRestartWithoutSideways {

	static int count = 1;
	static int steps = 0;

	public static void printStatistics(int iterations) {
		float avgRestart = ((float) count / (float) iterations);
		float avgsteps = ((float) steps / (float) iterations);
		System.out.println("The average number of restarts:  " + (avgRestart));
		System.out.println("The average number of steps:  " + avgsteps);
	}

	public static void climb(int[] queens) {

		int c, j, N = queens.length;
		int[] temp = new int[N];
		int[][] mat = new int[N][N];
		int[] next = new int[N];
		Board.printBoard(queens, N);
		int bestSoFar = Board.getHeuristic(queens);
		HashMap<Integer, int[]> map = new HashMap<Integer, int[]>();
		mat = Board.getHMatrix(queens);
		int min = mat[0][0];
		for (c = 0; c < N; c++) {
			for (int r = 0; r < N; r++) {
				if (min > mat[c][r]) {
					min = mat[c][r];
				}
			}
		}
		int count = 1;
		for (c = 0; c < N; c++) {
			for (j = 0; j < N; j++) {
				for (int k = 0; k < N; k++) {
					temp[k] = queens[k];
				}
				if (mat[c][j] == min) {
					for (int k = 0; k < N; k++) {
						next[k] = temp[k];
					}
					next[j] = c;
					map.put(count, next);
```

```
                                        count++;
                                }
                        }
                }
                Random rand = new Random();
                int index = rand.nextInt(count);
                if (index == 0) {
                        index = index + 1;
                }
                temp = map.get(index);

                if (min == 0) {
                        System.out.println("Solution Found");
                        Board.printBoard(temp, temp.length);
                        System.out.println("Conflicts: " + Board.getHeuristic(temp));
                        steps++;
                } else if (min < bestSoFar) {
                        steps++;
                        climb(temp);
                } else if (min == bestSoFar) {
                        System.out.println("Doing random restart.");
                        count++;
                        climb(Board.createBoard(N));

                }

        }

}
```

Results Report:

Hill Climbing:

Success rate: 15.0%
Failure rate: 85.0%
Average steps in success: 4.3333335
Average steps in failure: 2.964706

Search sequences for four random initial configurations:

```
Initial Configuration:
0     0     0     0     0     Q     0     Q
0     0     0     0     0     0     0     0
0     0     Q     0     0     0     0     0
0     0     0     0     0     0     0     0
0     0     0     0     Q     0     0     0
0     Q     0     0     0     0     0     0
0     0     0     Q     0     0     Q     0
Q     0     0     0     0     0     0     0
Solution Found
0     0     0     0     0     Q     0     0
0     0     Q     0     0     0     0     0
Q     0     0     0     0     0     0     0
0     0     0     0     0     0     0     Q
0     0     0     0     Q     0     0     0
0     Q     0     0     0     0     0     0
0     0     0     Q     0     0     0     0
0     0     0     0     0     0     Q     0
Conflicts: 0

Initial Configuration:
0     0     0     0     0     0     0     Q
0     0     0     0     0     0     0     0
0     0     0     Q     0     0     0     0
0     0     Q     0     0     0     0     0
0     0     0     0     0     0     0     0
Q     0     0     0     0     0     0     0
0     Q     0     0     0     0     Q     0
0     0     0     0     Q     Q     0     0
Solution Found
0     Q     0     0     0     0     0     0
0     0     0     0     0     Q     0     0
0     0     0     0     0     0     0     Q
0     0     Q     0     0     0     0     0
Q     0     0     0     0     0     0     0
0     0     0     Q     0     0     0     0
0     0     0     0     0     0     Q     0
0     0     0     0     Q     0     0     0
Conflicts: 0
```

```
Initial Configuration:
0     0     Q     0     0     0     0     0
0     0     0     0     0     0     0     0
0     0     0     Q     0     Q     0     0
Q     0     0     0     0     0     0     0
0     0     0     0     0     0     0     0
0     0     0     0     Q     0     0     Q
0     Q     0     0     0     0     0     0
0     0     0     0     0     0     Q     0
Solution Found
0     0     0     0     Q     0     0     0
0     0     0     0     0     0     0     Q
0     0     0     Q     0     0     0     0
Q     0     0     0     0     0     0     0
0     0     Q     0     0     0     0     0
0     0     0     0     0     Q     0     0
0     Q     0     0     0     0     0     0
0     0     0     0     0     0     Q     0
Conflicts: 0

Initial Configuration:
0     Q     Q     0     0     0     0     0
0     0     0     Q     0     0     0     0
0     0     0     0     0     0     0     0
Q     0     0     0     0     0     0     0
0     0     0     0     Q     0     Q     0
0     0     0     0     0     0     0     0
0     0     0     0     0     0     0     0
0     0     0     0     0     Q     0     Q
Solution Found
0     0     0     0     0     Q     0     0
0     0     0     Q     0     0     0     0
0     0     0     0     0     0     Q     0
Q     0     0     0     0     0     0     0
0     0     Q     0     0     0     0     0
0     0     0     0     Q     0     0     0
0     Q     0     0     0     0     0     0
0     0     0     0     0     0     0     Q
Conflicts: 0
```

Hill Climbing with Sideways:

Success rate: 96.0%
Failure rate: 4.0%
Average steps in success: 18.572916
Average steps in failure: 55.75

Search sequences for four random initial configurations:

```
Initial Configuration:
0     0     Q     0     0     0     0     0
0     0     0     0     0     0     0     0
0     0     0     Q     0     0     0     0
0     0     0     0     0     0     0     0
Q     Q     0     0     0     Q     Q     0
0     0     0     0     0     0     0     Q
0     0     0     0     Q     0     0     0
0     0     0     0     0     0     0     0
Allowing the sideways move number 0
Allowing the sideways move number 1
Allowing the sideways move number 0
Allowing the sideways move number 1
Allowing the sideways move number 2
Allowing the sideways move number 3
Allowing the sideways move number 4
Allowing the sideways move number 5
Allowing the sideways move number 6
Allowing the sideways move number 7
Allowing the sideways move number 8
Allowing the sideways move number 9
Allowing the sideways move number 10
Allowing the sideways move number 11
Allowing the sideways move number 12
Allowing the sideways move number 13
Allowing the sideways move number 14
Allowing the sideways move number 15
Allowing the sideways move number 16
Allowing the sideways move number 17
Allowing the sideways move number 18
Allowing the sideways move number 19
Allowing the sideways move number 20
Solution Found
0     0     Q     0     0     0     0     0
0     0     0     0     0     Q     0     0
0     0     0     Q     0     0     0     0
Q     0     0     0     0     0     0     0
0     0     0     0     0     0     0     Q
0     0     0     0     Q     0     0     0
0     0     0     0     0     0     Q     0
0     Q     0     0     0     0     0     0
```

```
Conflicts : 0

Initial Configuration:
0     Q     0     Q     Q     0     0     0
0     0     Q     0     0     0     0     Q
0     0     0     0     0     0     0     0
0     0     0     0     0     0     Q     0
0     0     0     0     0     0     0     0
0     0     0     0     0     0     0     0
Q     0     0     0     0     0     0     0
0     0     0     0     0     Q     0     0
Allowing the sideways move number 0
Allowing the sideways move number 1
Allowing the sideways move number 2
Allowing the sideways move number 3
Allowing the sideways move number 4
Allowing the sideways move number 5
Allowing the sideways move number 6
Allowing the sideways move number 7
Allowing the sideways move number 8
Allowing the sideways move number 9
Allowing the sideways move number 0
Allowing the sideways move number 1
Allowing the sideways move number 2
Allowing the sideways move number 3
Solution Found
0     0     0     0     0     Q     0     0
0     0     Q     0     0     0     0     0
Q     0     0     0     0     0     0     0
0     0     0     0     0     0     0     Q
0     0     0     Q     0     0     0     0
0     Q     0     0     0     0     0     0
0     0     0     0     0     0     Q     0
0     0     0     0     Q     0     0     0
Conflicts : 0

Initial Configuration:
0     0     0     0     0     Q     0     0
0     0     0     0     Q     0     0     0
0     0     0     0     0     0     0     0
0     0     0     Q     0     0     0     Q
0     0     0     0     0     0     Q     0
0     0     0     0     0     0     0     0
Q     Q     0     0     0     0     0     0
0     0     Q     0     0     0     0     0
Allowing the sideways move number 0
Allowing the sideways move number 0
Allowing the sideways move number 1
Allowing the sideways move number 2
Allowing the sideways move number 3
```

```
Allowing the sideways move number 4
Allowing the sideways move number 5
Solution Found
0     0     Q     0     0     0     0     0
0     0     0     0     Q     0     0     0
0     Q     0     0     0     0     0     0
0     0     0     0     0     0     0     Q
0     0     0     0     0     Q     0     0
0     0     0     Q     0     0     0     0
0     0     0     0     0     0     Q     0
Q     0     0     0     0     0     0     0
Conflicts : 0

Initial Configuration:
0     0     0     0     Q     Q     0     0
0     Q     0     Q     0     0     0     0
Q     0     Q     0     0     0     0     0
0     0     0     0     0     0     0     0
0     0     0     0     0     0     0     0
0     0     0     0     0     0     Q     0
0     0     0     0     0     0     0     0
0     0     0     0     0     0     0     Q
Allowing the sideways move number 0
Allowing the sideways move number 1
Allowing the sideways move number 2
Allowing the sideways move number 0
Allowing the sideways move number 1
Allowing the sideways move number 2
Allowing the sideways move number 3
Allowing the sideways move number 4
Allowing the sideways move number 5
Allowing the sideways move number 6
Allowing the sideways move number 7
Solution Found
0     0     0     0     Q     0     0     0
0     0     0     0     0     0     Q     0
Q     0     0     0     0     0     0     0
0     0     Q     0     0     0     0     0
0     0     0     0     0     0     0     Q
0     0     0     0     0     Q     0     0
0     0     0     Q     0     0     0     0
0     Q     0     0     0     0     0     0
Conflicts : 0
```

Random Restart Without Sideways

The average number of restarts: 0.01
The average number of steps: 23.95

Random Restart with Sideways

The average number of restarts: 0.01
The average number of steps: 17.83