Project report
on

Solving N-Queens problem using Hill-Climbing Algorithm and its variants

Project Guidance By

Dr. Dewan Ahmed

Team details

Aarti Nimhan
animhan1@uncc.edu
801098198

Uma Sai Madhuri Jetty
ujetty@uncc.edu
801101049

Sahithi Priya Gutta
sgutta@uncc.edu
801098589

AIM

To solve n-queens problem using hill-climbing search and its variants.

PROBLEM STATEMENT

Implement Hill-climbing search, Hill-climbing search with sideway moves and Random-restart hill-climbing with and without sideways move and apply it to n-queens problem. List average number of steps when the algorithm succeeds and fails along with the success and failure rate for multiple iterations.

N-QUEENS PROBLEM

The N-queens puzzle is the problem of placing N queens on a N x N chessboard such that no two queens attack each other. The queen is the most powerful piece in chess and can attack from any distance horizontally, vertically, or diagonally. Thus, a solution requires that no two queens share the same row, column, or diagonal.

PROBLEM FORMULATION

Initial State: A random arrangement on n queens, with one in each column.
Goal State: N queens placed on the board such that no two queens can attack each other.
States: Any arrangement of n queens, one in each column.
Actions: Move any attacked queen to another square in the same column.
Performance: Number of steps and success rate to find a solution.

HILL-CLIMBING ALGORITHM

Hill Climbing is heuristic search used for mathematical optimization problems in the field of Artificial Intelligence. It is an iterative algorithm that starts with an arbitrary solution to a problem, then attempts to find a better solution by making an incremental change to the solution. If the change produces a better solution, another incremental change is made to the new solution, and so on until no further improvements can be found.

Steepest-Ascent Hill-climbing: It first examines all the neighboring nodes and then selects the node closest to the solution state as next node with best heuristic value. If no best successor is found then the search fails.

Average number of steps to succeed: 4
Average number of steps to failure: 3
Average success rate: 14 %
Average failure rate: 85 %

Hill-climbing with Sideway moves: It selects any successor whose heuristic value is equal to the current state node. Because of this another problem of infinite iterations may rise but we can put limit on the number of sideway moves allowed.

Average number of steps to succeed: 21
Average number of steps to failure: 64
Average success rate: 94 %
Average failure rate: 6 %

Random Restart without Sideway moves: Random-restart hill-climbing conducts a series of hill-climbing searches from randomly generated initial states, running each until it halts or makes no progress. This enables comparison of many optimization trials and finding a most optimal solution thus becomes a question of using enough iterations on the data.

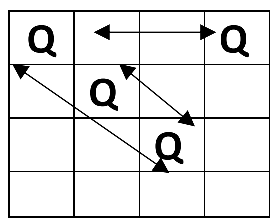
Average number of steps to succeed: 22
Average success rate: 100 %
Number of restarts needed: 6

Random Restart with Sideway moves: This algorithm allows sideway moves for random restart variant of hill climbing search. With random restart hill climbing, the probability of getting goal state tends to be equal to 1. Thus, this gives a much higher success rate than the basic version of hill climbing search algorithm.

Average number of steps to succeed: 25
Average success rate: 100 %
Number of restarts needed: 1

HEURISTIC FUNCTION:

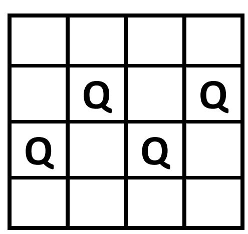
The Heuristic function in the N queen problem is the number of pairs of queens that are attacking each other. The best successor is the state with low heuristic value.



The heuristic value for the above problem is four since there are four pairs of queens that are attacking each other at this moment.

Sample 4-queens problem solved using Hill-Climbing search

Let's generate a random state with each queen placed in a column.



h = 5

The above state results in twelve possible states with different heuristic values. The state with lowest heuristic value will be chosen as the best successor. Let's calculate all the states and find the next best successor.

	Q		Q
		Q	
Q			

h = 3

Q	Q		Q
		Q	

h = 4

Q			
	Q		Q
		Q	

h = 5

	Q		
			Q
Q		Q	

h = 2

			Q
Q	Q	Q	

h = 4

			Q
Q		Q	
	Q		

h = 5

	Q		Q
Q			
		Q	

h = 2

	Q	Q	Q
Q			

h = 4

		Q	
	Q		Q
Q			

h = 5

			Q
	Q		
Q		Q	

h = 3

	Q		
Q		Q	Q

h = 5

	Q		
Q		Q	
			Q

h = 5

From the above twelve states the least heuristic value is 2. So, the best successor is the state with least heuristic value.

	Q		
			Q
Q		Q	

Let's calculate the best successor for the above state by moving queen each time.

	Q		
			Q
		Q	
Q			

h = 1

	Q		
Q			Q
		Q	

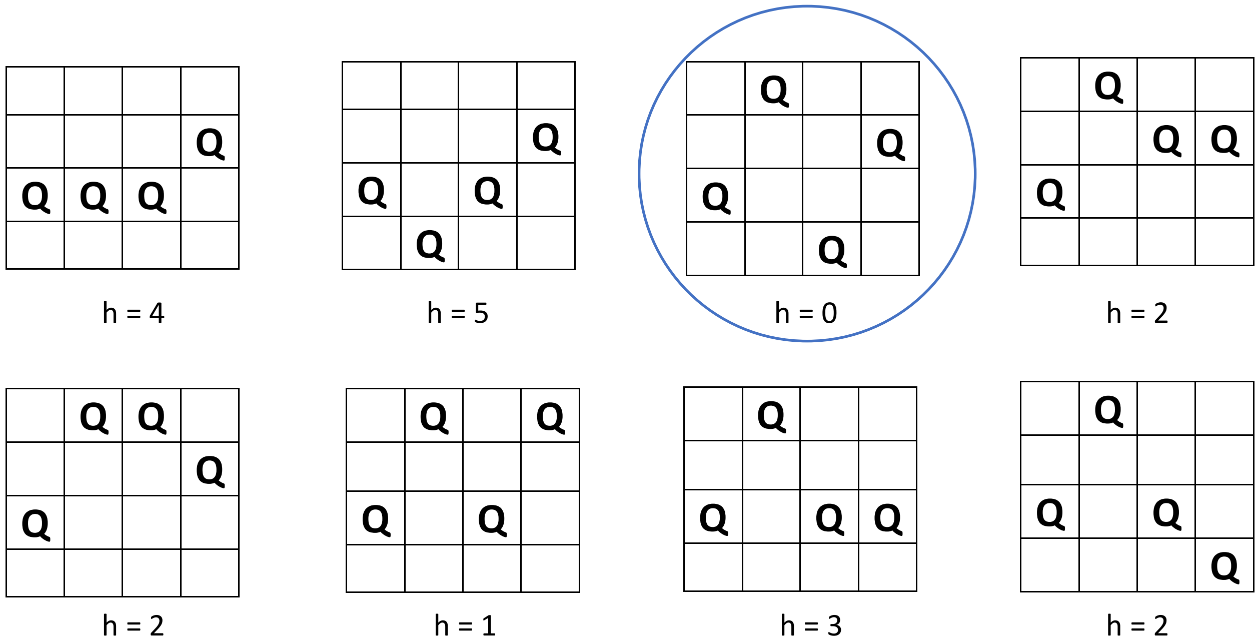
h = 3

Q	Q		
			Q
		Q	

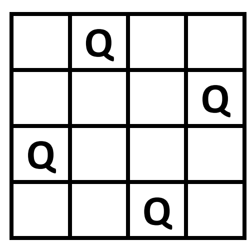
h = 2

	Q		Q
Q		Q	

h = 5



From the above twelve states the least heuristic value is ‘0’ which means that no queens are attacking each other for the state with h value 0. So, that is the solution state



PROGRAM DESIGN AND EXPLANATION

Utility Functions used

generateRandomState()
bestSuccessor()
copyState()
calculateHeuristic()
possibleSuccessors()
printNode()

generateRandomState

This is a utility method used to generate a random state by taking the number of queens input from the user and placing the queens randomly on the board such that each queen is placed in a column and finally returns the new state.

bestSuccessor

This utility method returns the best successor from the list of possible successors generated. The state with the lowest heuristic value is selected as best successor. The heuristic values are compared/sorted using the priority queue.

copyState
This utility method copies one state to another state i.e. parent node to possible child nodes.

calculateHeuristic
Calculates and returns the heuristic value of a state. Checks the indexes of the queen. If two queens are in the same row or two queens are diagonal to each other the heuristic value is incremented. Based on the row and column indexes of the queen, verifies whether two queens are in same row. Verifies if absolute differences of rows and absolute difference of columns matches to know whether two queens are diagonal.

possibleSuccessors
This method finds the possible successors of the current state. Copies the current state and clones to a new state and moves each queen at a time column wise and generates the possible successors. For 4 queens the number of possible successors is 12 and for 8-queens it is 56.

printNode
Prints the state of the current board i.e. prints state of generated board and the best successor selected every time before reaching the solution.

Functions and Variables used in Steepest Ascent
Variables used
numberofQueens
regularMoves
totalSuccessfulMoves
totalFailMoves
totalSuccessfulIterations
totalFailedIterations

process function
This method implements Hill-climbing Steepest ascent. Utility functions generateRandomState, calculateHeuristic, possibleSuccessos, bestSucessor are used to find the ultimate solution state. If the heuristicvalue is not equal to '0' and the possible successors have no better heuristic, results in failure. This function returns the totalSuccessfulMoves, totalSucessfulIterations, totalFailMoves, totalFailIterations.

Functions and Variables used in Hill-Climbing with Sidewaymoves
Variables used
numberofQueens
regularMoves
totalSuccessfulMoves
totalFailMoves
totalSuccessfulIterations
totalFailedIterations
sideWalkStepsLimit
sideStepCount

process function

This method implements hill climbing with Sideway moves. Utility functions generateRandomState, calculateHeuristic, possibleSuccessos, bestSucessor are used to find the ultimate solution state. If the heuristicvalue is not equal to '0' and the possible successors have no better heuristic, then sideway moves are considered. Best successor is picked randomly until the sidewalkStepsLimit. This function returns the totalSuccessfulMoves, totalSucessfulIterations, totalFailMoves, totalFailIterations.

Functions and Variables used in Random Restart without Sideway moves

Variables used

numberofQueens
regularMoves
totalSuccessfulMoves
totalFailMoves
totalSuccessfulIterations
totalFailedIterations
totalNumberOfRestarts
restartUsedCount

process function

This method implements hill climbing with Sideway moves. Utility functions generateRandomState, calculateHeuristic, possibleSuccessos, bestSucessor are used to find the ultimate solution state. If the heuristicvalue is not equal to '0' and the possible successors have no better heuristic, then the board is randomly regenerated again until the solution is found. This function returns the totalSuccessfulMoves, totalSucessfulIterations, totalFailMoves, totalFailIterations, totalNumberOfRestarts, restartUsedCount.

Functions and Variables used in Random Restart without Sideway moves

Variables used

numberofQueens
regularMoves
totalSuccessfulMoves
totalFailMoves
totalSuccessfulIterations
totalFailedIterations
sideWalkStepsLimit
sideStepCount
totalNumberOfRestarts
restartUsedCount

process function

This method implements hill climbing with Sideway moves. Utility functions generateRandomState, calculateHeuristic, possibleSuccessos, bestSucessor are used to find the ultimate solution state. If the heuristicvalue is not equal to '0' and the possible successors have no better heuristic, then sideway moves are considered. Best successor is picked randomly until the sidewalkStepsLimit. If the solution is not found and the sidewalk limit is reached then the board is randomly regenerated again until the solution is found. This function returns the totalSuccessfulMoves, totalSucessfulIterations, totalFailMoves, totalFailIterations, totalNumberOfRestarts, restartUsedCount.

SCREENSHOTS

Input

Enter the number of queens and enter y to print the states along with statistics

```
Enter the value for Number of Queens:
8
Do you wish to print output states alongwith the statistics? (Y/y - Yes And N/n - No)
y

 Enter 1 for n-queens problem using Steepest Ascent
 Enter 2 for n-queens problem using Sideway moves
 Enter 3 for n-queens problem using Random restart without sideways moves
 Enter 4 for n-queens problem using Random restart with sideways moves
2
```

Hill-Climbing Steepest Ascent

```
        _       _       _       _       Q       _       _       _
        _       _       Q       _       _       _       _       _
        Q       _       _       _       _       _       _       _
        _       _       _       Q       _       _       _       _
        _       _       _       _       _       _       Q       Q
        _       _       _       _       _       _       _       _
        _       Q       _       _       _       _       _       _
        _       _       _       _       _       Q       _       _
------------------

----------------------------------------------------------------------------
Steepest-Ascent Hill Climbing Algorithm
# Of Queens: 8
Number of Iterations: 264
Success/Fail Analysis
Success Rate: 14.02%
Failure Rate: 85.98%
Average Number of Steps When It Succeeds: 3.95
Average Number of Steps When It Fails: 3.24
```

Hill-Climbing with Sideway moves

```
----Goal State-----
        _       Q       _       _       _       _       _       _
        _       _       _       _       _       _       Q       _
        _       _       _       _       Q       _       _       _
        _       _       _       _       _       _       _       Q
        Q       _       _       _       _       _       _       _
        _       _       _       Q       _       _       _       _
        _       _       _       _       _       Q       _       _
        _       _       Q       _       _       _       _       _
------------------

----------------------------------------------------------------------------
Sideway moves Hill Climbing Algorithm
# Of Queens: 8
Number of Iterations: 264
Success/Fail Analysis
Success Rate: 94.32%
Failure Rate: 5.68%
Average Number of Steps When It Succeeds: 20.13
Average Number of Steps When It Fails: 63.93
```

Random restart without Sideway moves

```
----Goal State-----
        _       _       _       _       _       _       Q       _
        _       Q       _       _       _       _       _       _
        _       _       _       Q       _       _       _       _
        Q       _       _       _       _       _       _       _
        _       _       _       _       _       _       _       Q
        _       _       _       _       Q       _       _       _
        _       _       Q       _       _       _       _       _
        _       _       _       _       _       Q       _       _
------------------

----------------------------------------------------------------------------
Random Restart without Sideway Moves Hill Climbing Algorithm
# Of Queens: 8
Number of Iterations: 264
Success/Fail Analysis
Success Rate: 100.00%
Failure Rate: 0.00%
Average Number of Steps When It Succeeds: 21.47
Average Number of Steps When It Fails: 0.00
Average Number of Restarts: 6.00
```

Random restart with Sideway moves

```
----Goal State-----
        _       _       _       _       _       _       Q       _
        _       _       _       _       Q       _       _       _
        _       _       Q       _       _       _       _       _
        Q       _       _       _       _       _       _       _
        _       _       _       _       _       Q       _       _
        _       _       _       _       _       _       _       Q
        _       Q       _       _       _       _       _       _
        _       _       _       Q       _       _       _       _
------------------

----------------------------------------------------------------------------
Random Restart with Sideway moves Hill Climbing Algorithm
# Of Queens: 8
Number of Iterations: 264
Success/Fail Analysis
Success Rate: 100.00%
Failure Rate: 0.00%
Average Number of Steps When It Succeeds: 25.47
Average Number of Steps When It Fails: 0.00
Average Number of Restarts: 1.06
```

Input

Enter the number of queens and enter n to print only the statistics

```
Enter the value for Number of Queens:
8
Do you wish to print output states alongwith the statistics? (Y/y - Yes And N/n - No)
n

 Enter 1 for n-queens problem using Steepest Ascent
 Enter 2 for n-queens problem using Sideway moves
 Enter 3 for n-queens problem using Random restart without sideways moves
 Enter 4 for n-queens problem using Random restart with sideways moves
1
```

Steepest Accent

```
 Enter 1 for n-queens problem using Steepest Ascent
 Enter 2 for n-queens problem using Sideway moves
 Enter 3 for n-queens problem using Random restart without sideways moves
 Enter 4 for n-queens problem using Random restart with sideways moves
1

----------------------------------------------------------------------------
Steepest-Ascent Hill Climbing Algorithm
# Of Queens: 8
Number of Iterations: 401
Success/Fail Analysis
Success Rate: 14.21%
Failure Rate: 85.79%
Average Number of Steps When It Succeeds: 4.37
Average Number of Steps When It Fails: 3.21
```

Random Restart without Sideway moves

```
 Enter 1 for n-queens problem using Steepest Ascent
 Enter 2 for n-queens problem using Sideway moves
 Enter 3 for n-queens problem using Random restart without sideways moves
 Enter 4 for n-queens problem using Random restart with sideways moves
3

----------------------------------------------------------------------------
Random Restart without Sideway Moves Hill Climbing Algorithm
# Of Queens: 8
Number of Iterations: 401
Success/Fail Analysis
Success Rate: 100.00%
Failure Rate: 0.00%
Average Number of Steps When It Succeeds: 22.41
Average Number of Steps When It Fails: 0.00
Average Number of Restarts: 6.71
```

SOURCE CODE

CommonUtils.java

This class contains utilities which can be used by all variants of hill climbing.

```java
import java.util.ArrayList;
import java.util.List;
import java.util.PriorityQueue;
import java.util.Random;

public class CommonUtils {

	private static int QUEEN = 1;
	private static int NO_QUEEN = 0;

	/**
	 * This method finds possible successors to a given state. Each successor is a
	 * result of moving one queen from its current position to any position in its
	 * own column.
```

```
         *
         * @param currentNode This is the node for which successors are to be
generated.
         * @return this method returns a list of nodes which are successors to the
input
         *         node.
         */
        public List<Node> findPossibleSuccessors(Node currentNode) {

                List<Node> possibleSuccessors = new ArrayList<Node>();

                int currentQueenColumn, currentQueenRow, newQueenRow, newQueenColumn;
                for (int column = 0; column < currentNode.getboard().length; column++) {
// For each column
                        currentQueenRow = Integer.MAX_VALUE;
                        currentQueenColumn = column;
                        newQueenColumn = column;
                        // find the current queen row position in the current column
                        for (int i = 0; i < currentNode.getboard().length; i++) {
                                if (currentNode.getboard()[i][currentQueenColumn] ==
QUEEN) {
                                        currentQueenRow = i;
                                        break;
                                }
                        }
                        // for each row position swap with current row and create a new
possible
                        // positions
                        for (int row = 0; row < currentNode.getboard().length; row++) {
                                if (row != currentQueenRow && currentQueenRow !=
Integer.MAX_VALUE) {
                                        int[][] currentState = new
int[currentNode.getboard().length][currentNode.getboard().length];
                                        copyState(currentNode.getboard(), currentState);
                                        newQueenRow = row;
                                        // swap
                                        currentState[currentQueenRow][currentQueenColumn] =
NO_QUEEN;

                                        currentState[newQueenRow][newQueenColumn] = QUEEN;
                                        Node newNode = new Node();
                                        newNode.setboard(currentState);
                                        possibleSuccessors.add(newNode);
                                }
                        }
                }
                return possibleSuccessors;
        }

        /**
         * This method calculates the heuristic value for the input node. This value
is
         * calculated by adding all possible attacks of each queen on the board. This
         * heuristic value is updated in the hcost attribute of the input node.
         *
         * @param currentNode the input Node for which heuristic is to be calculated.
```

```
     *
     */
    public void calculateHeuristic(Node currentNode) {

        int[][] currentState = new
int[currentNode.getboard().length][currentNode.getboard().length];
        copyState(currentNode.getboard(), currentState);
        // Find queen positions
        int[] rowIndex = new int[currentNode.getboard().length];
        int[] colIndex = new int[currentNode.getboard().length];
        int queenIndex = 0;
        for (int col = 0; col < currentNode.getboard().length; col++) {
            for (int row = 0; row < currentNode.getboard().length; row++) {
                if (currentState[row][col] == QUEEN) {
                    rowIndex[queenIndex] = row;
                    colIndex[queenIndex] = col;
                    queenIndex++;
                }
            }
        }

        int heuristic = 0;
        boolean heuristicNotCheckedFlag = true;
        // From positions for each queen check attack with other
        for (int i = 0; i < rowIndex.length; i++) {
            // First Queen position is rowIndex[i] colIndex[i]
            for (int j = i + 1; j < rowIndex.length; j++) {
                // Second Queen position is rowIndex[j] colIndex[j]
                heuristicNotCheckedFlag = true;
                // if First and second queen are in same rows then
increment heuristic cost
                if (rowIndex[i] == rowIndex[j]) {
                    heuristic++;
                }

                // if First and second queen are in same rows then
increment heuristic cost
                if (colIndex[i] == colIndex[j]) {
                    heuristic++;
                }

                // if First and second queen are in diagonal then
increment heuristic cost
                if (Math.abs(rowIndex[i] - rowIndex[j]) ==
Math.abs(colIndex[i] - colIndex[j])) {
                    heuristic++;
                }

            }
        }
        if (!heuristicNotCheckedFlag)
            heuristic = Integer.MAX_VALUE;
        currentNode.sethCost(heuristic);
    }
```

```
	/**
	 * This is a utility method to copy one state to another
	 *
	 * @param sourceNodeState      state to be copied.
	 * @param destinationNodeState copied state.
	 */
	public void copyState(int[][] sourceNodeState, int[][] destinationNodeState) {
		for (int i = 0; i < sourceNodeState.length; i++)
			for (int j = 0; j < sourceNodeState.length; j++)
				destinationNodeState[i][j] = sourceNodeState[i][j];

	}

	/**
	 * This method returns the successor with the least hcost value.
	 *
	 * @param possibleSuccessors this is the list of possible successors to a
state.
	 * @return will return a Node with the least hcost value amongst the
successors.
	 */
	public Node bestSuccessor(List<Node> possibleSuccessors) {
		PriorityQueue<Node> pq = new
PriorityQueue<Node>(possibleSuccessors.size(), new HCostComparator());
		for (int i = 0; i < possibleSuccessors.size(); i++) {
			pq.add(possibleSuccessors.get(i));
		}
		List<Node> bestSuccessorsList = new ArrayList<Node>();
		Node bestSuccessor = new Node();
		bestSuccessor = pq.poll();
		int hcost = bestSuccessor.gethCost();
		bestSuccessorsList.add(bestSuccessor);
		while (pq.peek().gethCost() == hcost) {
			bestSuccessorsList.add(pq.poll());
		}
		Random rand = new Random();
		int index = rand.nextInt(bestSuccessorsList.size());
		bestSuccessor = bestSuccessorsList.get(index);
		pq.clear();
		return bestSuccessor;

	}

	/**
	 * This method generates a new Node with a random state. This random state has
	 * one queen per column. All the queens are placed at random rows in their
	 * respective columns.
	 *
	 * @param numberOfQueens the total number of queens on the board.
	 * @return returns a node created with randomly placed queens.
	 */
	public Node generateRandomState(int numberOfQueens) {
		Random rand = new Random();
		int[][] newState = new int[numberOfQueens][numberOfQueens];
		Node newNode = new Node();
```

```
		for (int i = 0; i < numberOfQueens; i++) {
			int randomNumber = rand.nextInt(numberOfQueens - 1);
			for (int j = 0; j < numberOfQueens; j++) {
				if (j == randomNumber) {
					newState[j][i] = QUEEN;
				} else {
					newState[j][i] = NO_QUEEN;
				}
			}
		}
		newNode.setboard(newState);
		return newNode;
	}

	/**
	 * This method prints the state of the input node.
	 *
	 * @param node
	 */
	public void printNode(Node node) {
		for (int i = 0; i < node.getboard().length; i++) {
			for (int j = 0; j < node.getboard().length; j++) {
				String printValue = "_";
				if ((node.getboard())[i][j] == 1) {
					printValue = "Q";
				}
				System.out.print("\t" + printValue);
			}
			System.out.println();
		}
		System.out.println("------------------");
	}

}
```

HCostComparator.java

This is a comparator created which is used in the Priority Queue to decide the ordering based on the hCost. The priority queue is ordered in ascending order of hcost.

```
import java.util.Comparator;

public class HCostComparator implements Comparator<Node> {

	@Override
	public int compare(Node o1, Node o2) {
		if (o1.gethCost() < o2.gethCost())
			return -1;
		if (o1.gethCost() > o2.gethCost())
			return 1;
		return 0;
	}

}
```

HillClimbingFramework.java

This is the main driver class for all implementations of hill climbing.

```
import java.util.Random;

import java.util.Scanner;


public class HillClimbingFramework {


	public static void main(String[] args) {

		float successPercent = 0, avgSuccess = 0, failurePercent = 0, avgFailure = 0;

		float totalSuccessfulMoves = 0.0f, totalSuccessfulIterations = 0.0f, totalFailMoves = 0.0f,

				totalFailedIterations = 0.0f;

		float totalNumberOfRestarts = 0.0f;

		int numberOfIterationsForWhichRestartWasUsed = 0;

		float numberOfRestarts = 0.0f;

		boolean wishToContinue = true;

		int options, noOfQueens;

		Random rand = new Random();

		int iterations = rand.nextInt(400) + 100; // as expected iterations are generated random

		boolean printOutput = false;

		System.out.println("Enter the value for Number of Queens: ");

		Scanner scanner = new Scanner(System.in);

		noOfQueens = scanner.nextInt();

		// validating if the entered number of Queens are greater than 3

		if (noOfQueens < 4) {

			System.out.println("Number of Queens should be greater than 3.");

			System.exit(0);

		}

		System.out.println("Do you wish to print output states alongwith the statistics? (Y/y -
Yes And N/n - No)");
```

```
		if (scanner.next().equalsIgnoreCase("Y"))
			printOutput = true;

		while (wishToContinue) {
			System.out.println("\n Enter 1 for n-queens problem using Steepest Ascent"
					+ "\n Enter 2 for n-queens problem using Sideway moves"
					+ "\n Enter 3 for n-queens problem using Random restart
without sideways moves"
					+ "\n Enter 4 for n-queens problem using Random restart with
sideways moves ");
			options = scanner.nextInt();

			switch (options) {
			case 1:// Iterations of Hill Climbing Steepest Ascent.
				for (int i = 0; i < iterations; i++) {
					SteepestAscent steepestAscentObj = new SteepestAscent();
					int[] tempAnswer = steepestAscentObj.process(noOfQueens,
printOutput);
					totalSuccessfulMoves = totalSuccessfulMoves + tempAnswer[0];
					totalSuccessfullIterations = totalSuccessfullIterations +
tempAnswer[1];
					totalFailMoves = totalFailMoves + tempAnswer[2];
					totalFailedIterations = totalFailedIterations + tempAnswer[3];
				}
				System.out.println("\n-----------------------------------------------------------------
----------");
				System.out.println("Steepest-Ascent Hill Climbing Algorithm");
				System.out.println("# Of Queens: " + noOfQueens);
				System.out.println("Number of Iterations: " + iterations);
				System.out.println("Success/Fail Analysis");
```

```
// Calculating success and failure percent and average moves.
if (totalSuccessfulIterations != 0) {
    successPercent = (totalSuccessfulIterations / iterations) * 100;
    avgSuccess = totalSuccessfulMoves / totalSuccessfulIterations;
}
if (totalFailedIterations != 0) {
    failurePercent = (totalFailedIterations / iterations) * 100;
    avgFailure = totalFailMoves / totalFailedIterations;
}
System.out.println("Success Rate: " + String.format("%.2f",
successPercent) + "%");
System.out.println("Failure Rate: " + String.format("%.2f",
failurePercent) + "%");
System.out.println("Average Number of Steps When It Succeeds: " +
String.format("%.2f", avgSuccess));
System.out.println("Average Number of Steps When It Fails: " +
String.format("%.2f", avgFailure));
break;
case 2:// Iterations of Hill Climbing with Sideway Moves.
for (int i = 0; i < iterations; i++) {
    SidewayMoves sidewayMovesObj = new SidewayMoves();
    int[] tempAnswer = sidewayMovesObj.process(noOfQueens,
printOutput);
    totalSuccessfulMoves = totalSuccessfulMoves + tempAnswer[0];
    totalSuccessfulIterations = totalSuccessfulIterations +
tempAnswer[1];
    totalFailMoves = totalFailMoves + tempAnswer[2];
    totalFailedIterations = totalFailedIterations + tempAnswer[3];
}
System.out.println("\n-----------------------------------------------------------------
----------");
```

```
System.out.println("Sideway moves Hill Climbing Algorithm");
System.out.println("# Of Queens: " + noOfQueens);
System.out.println("Number of Iterations: " + iterations);
System.out.println("Success/Fail Analysis");
// Calculating success and failure percent and average moves.
if (totalSuccessfulIterations != 0) {
	successPercent = (totalSuccessfulIterations / iterations) * 100;
	avgSuccess = totalSuccessfulMoves / totalSuccessfulIterations;
}
if (totalFailedIterations != 0) {
	failurePercent = (totalFailedIterations / iterations) * 100;
	avgFailure = totalFailMoves / totalFailedIterations;
}
System.out.println("Success Rate: " + String.format("%.2f",
successPercent) + "%");
System.out.println("Failure Rate: " + String.format("%.2f",
failurePercent) + "%");
System.out.println("Average Number of Steps When It Succeeds: " +
String.format("%.2f", avgSuccess));
System.out.println("Average Number of Steps When It Fails: " +
String.format("%.2f", avgFailure));

break;
case 3:// Iterations of Hill Climbing Random Restart without Sideway Moves.
numberOfIterationsForWhichRestartWasUsed = 0;
for (int i = 0; i < iterations; i++) {
	RandomRestartForSteepestAscent
randomRestartSteepestAscentObj = new RandomRestartForSteepestAscent();
	int[] tempAnswer =
randomRestartSteepestAscentObj.process(noOfQueens, printOutput);
	totalSuccessfulMoves = totalSuccessfulMoves + tempAnswer[0];
```

```
                    totalSuccessfulIterations = totalSuccessfulIterations +
tempAnswer[1];
                    totalFailMoves = totalFailMoves + tempAnswer[2];
                    totalFailedIterations = totalFailedIterations + tempAnswer[3];
                    totalNumberOfRestarts = totalNumberOfRestarts +
tempAnswer[4];
                    numberOfIterationsForWhichRestartWasUsed =
numberOfIterationsForWhichRestartWasUsed + tempAnswer[5];

                }
                System.out.println("\n-----------------------------------------------------------------
----------");
                System.out.println("Random Restart without Sideway Moves Hill
Climbing Algorithm");
                System.out.println("# Of Queens: " + noOfQueens);
                System.out.println("Number of Iterations: " + iterations);
                System.out.println("Success/Fail Analysis");
                // Calculating success and failure percent and average moves.
                if (totalSuccessfulIterations != 0) {
                    successPercent = (totalSuccessfulIterations / iterations) * 100;
                    avgSuccess = totalSuccessfulMoves / totalSuccessfulIterations;
                }
                if (totalFailedIterations != 0) {
                    failurePercent = (totalFailedIterations / iterations) * 100;
                    avgFailure = totalFailMoves / totalFailedIterations;
                }
                if (numberOfIterationsForWhichRestartWasUsed != 0) {
                    numberOfRestarts = (totalNumberOfRestarts /
numberOfIterationsForWhichRestartWasUsed);
                }
```

```
System.out.println("Success Rate: " + String.format("%.2f",
successPercent) + "%");
System.out.println("Failure Rate: " + String.format("%.2f",
failurePercent) + "%");
System.out.println("Average Number of Steps When It Succeeds: " +
String.format("%.2f", avgSuccess));
System.out.println("Average Number of Steps When It Fails: " +
String.format("%.2f", avgFailure));
System.out.println("Average Number of Restarts: " +
String.format("%.2f", numberOfRestarts));
break;
case 4:// Iterations of Hill Climbing Random Restart with Sideway Moves.
for (int i = 0; i < iterations; i++) {
RandomRestartForSidewaysMoves
randomRestartSidewaysMovesObj = new RandomRestartForSidewaysMoves();
int[] tempAnswer =
randomRestartSidewaysMovesObj.process(noOfQueens, printOutput);
totalSuccessfulMoves = totalSuccessfulMoves + tempAnswer[0];
totalSuccessfulIterations = totalSuccessfulIterations +
tempAnswer[1];
totalFailMoves = totalFailMoves + tempAnswer[2];
totalFailedIterations = totalFailedIterations + tempAnswer[3];
totalNumberOfRestarts = totalNumberOfRestarts +
tempAnswer[4];
numberOfIterationsForWhichRestartWasUsed =
numberOfIterationsForWhichRestartWasUsed + tempAnswer[5];
}
System.out.println("\n-----------------------------------------------------------------
----------");
System.out.println("Random Restart with Sideway moves Hill Climbing
Algorithm");
System.out.println("# Of Queens: " + noOfQueens);
System.out.println("Number of Iterations: " + iterations);
```

```
System.out.println("Success/Fail Analysis");
if (totalSuccessfulIterations != 0) {
    successPercent = (totalSuccessfulIterations / iterations) * 100;
    avgSuccess = totalSuccessfulMoves / totalSuccessfulIterations;
}
if (totalFailedIterations != 0) {
    failurePercent = (totalFailedIterations / iterations) * 100;
    avgFailure = totalFailMoves / totalFailedIterations;
}

if (numberOfIterationsForWhichRestartWasUsed != 0) {
    numberOfRestarts = (totalNumberOfRestarts /
numberOfIterationsForWhichRestartWasUsed);
}
System.out.println("Success Rate: " + String.format("%.2f",
successPercent) + "%");
System.out.println("Failure Rate: " + String.format("%.2f",
failurePercent) + "%");
System.out.println("Average Number of Steps When It Succeeds: " +
String.format("%.2f", avgSuccess));
System.out.println("Average Number of Steps When It Fails: " +
String.format("%.2f", avgFailure));
System.out.println("Average Number of Restarts: " +
String.format("%.2f", numberOfRestarts));

break;

default:
    System.out.println("Entered input is invalid.");
    break;
}
```

```
                System.out.println("Do you wish to continue? (Y/y - Yes And N/n - No)");
                String input = scanner.next();
                if (input.equalsIgnoreCase("Y")) {
                    // clear all variables
                    totalSuccessfulMoves = 0.0f;
                    totalSuccessfulIterations = 0.0f;
                    totalFailMoves = 0.0f;
                    totalFailedIterations = 0.0f;
                    totalNumberOfRestarts = 0.0f;
                    numberOfIterationsForWhichRestartWasUsed = 0;
                    avgSuccess = 0.0f;
                    avgFailure = 0.0f;
                    numberOfRestarts = 0.0f;
                    successPercent = 0.0f;
                    failurePercent = 0.0f;
                    wishToContinue = true;
                } else {
                    wishToContinue = false;
                }

            }

            scanner.close();
        }
}
```

Node.java

This class is a node class which consists of the unique state of the puzzle. Along with the state the Node also contains fields like hcost => the heuristic cost of the current state that is the sum of the number of attacks each queen faces.

```java
public class Node {

	private int[][] board;
	private int hCost;

	public int[][] getboard() {
		return board;
	}

	public void setboard(int[][] board) {
		this.board = board;
	}

	public int gethCost() {
		return hCost;
	}

	public void sethCost(int hCost) {
		this.hCost = hCost;
	}
}
```

SteepestAscent.java

This class implements Hill Climbing with Steepest Ascent.

```java
import java.util.ArrayList;
import java.util.List;

public class SteepestAscent {

	public int numberOfQueens = 0;
	public CommonUtils commonUtils = new CommonUtils();
	public boolean isSuccessful = false;
	public boolean isFail = false;
	List<Node> possibleSuccessors = new ArrayList<Node>();
	Node bestSuccessor = new Node();
	int regularMoves = 0;
	int totalSuccessfulMoves = 0;
	int totalFailMoves = 0;
	int totalSuccessfulIterations = 0;
	int totalFailedIterations = 0;
	int[] result = new int[4];

	/**
	 * This method implements Hill Climbing with Steepest Ascent.
	 *
	 * @param numberOfQueens the total number of queens on the board.
	 * @param printOutput    True if board states are to be printed. False if no
	 *                       board states are printed.
```

```
 * @return returns an integer array with moves required for success and failure
 *        also includes success and failure iterations count.
 */
public int[] process(int numberOfQueens, boolean printOutput) {
	this.numberOfQueens = numberOfQueens;

	// Generate a random state with one queen in every column
	Node currentNode = commonUtils.generateRandomState(numberOfQueens);

	// calculate heuristic
	commonUtils.calculateHeuristic(currentNode);
	if (printOutput) {
		System.out.println("----Initial State-----");
		commonUtils.printNode(currentNode);
	}
	while (!isSuccessful && !isFail) {

		// get possible successors
		possibleSuccessors = commonUtils.findPossibleSuccessors(currentNode);
		// calculate heuristic of possible successors
		for (int i = 0; i < possibleSuccessors.size(); i++) {
			commonUtils.calculateHeuristic(possibleSuccessors.get(i));
		}

		// select best successor if hcost is less than current hcost
		bestSuccessor = commonUtils.bestSuccessor(possibleSuccessors);
		if (currentNode.gethCost() == 0) {
			totalSuccessfulMoves = +regularMoves;
			totalSuccessfulIterations++;
			isSuccessful = true;
			if (printOutput) {
				System.out.println("----Goal State-----");
				commonUtils.printNode(currentNode);
			}
		} else if (bestSuccessor.gethCost() < currentNode.gethCost()) {
			// if yes update current Node with best Successor
			currentNode = bestSuccessor;
			regularMoves++;
		} else {
			totalFailedIterations++;
			totalFailMoves = +regularMoves;
			isFail = true;
			if (printOutput) {
				System.out.println("----Shoulder/Local Minima State-----");
```

```
                                        commonUtils.printNode(currentNode);
                                }
                        }
                }

                result[0] = totalSuccessfulMoves;
                result[1] = totalSuccessfulIterations;
                result[2] = totalFailMoves;
                result[3] = totalFailedIterations;

                return result;

        }
}
```

SidewayMoves.java

This class implements Hill Climbing with Sideway Moves.

```
import java.util.ArrayList;
import java.util.List;

public class SidewayMoves {

	public int numberOfQueens = 0;
	public CommonUtils commonUtils = new CommonUtils();
	public boolean isSuccessful = false;
	public boolean isFail = false;
	List<Node> possibleSuccessors = new ArrayList<Node>();
	Node bestSuccessor = new Node();
	int regularMoves = 0;
	int totalSuccessfulMoves = 0;
	int totalFailMoves = 0;
	int totalSuccessfulIterations = 0;
	int totalFailedIterations = 0;
	int[] result = new int[4];
	int sideWalkStepsLimit = 100;
	int sideStepCount = 0;
	boolean updateCurrentNode = false;

	/**
	 * This method implements Hill Climbing with Sideway Moves.
	 *
	 * @param numberOfQueens the total number of queens on the board.
	 * @param printOutput    True if board states are to be printed. False if no
	 *                       board states are printed.
	 * @return returns an integer array with moves required for success and
failure
	 *         also includes success and failure iterations count.
	 */
	public int[] process(int numberOfQueens, boolean printOutput) {
		this.numberOfQueens = numberOfQueens;
```

```
		// Generate a random state with one queen in every column
		Node currentNode = commonUtils.generateRandomState(numberOfQueens);

		// calculate heuristic
		commonUtils.calculateHeuristic(currentNode);
		if (printOutput) {
			System.out.println("----Initial State-----");
			commonUtils.printNode(currentNode);
		}
		while (!isSuccessful && !isFail) {
			updateCurrentNode = false;
			// get possible successors
			possibleSuccessors =
commonUtils.findPossibleSuccessors(currentNode);
			// calculate heuristic of possible successors
			for (int i = 0; i < possibleSuccessors.size(); i++) {
				commonUtils.calculateHeuristic(possibleSuccessors.get(i));
			}

			// select best successor if hcost is less than current hcost
			bestSuccessor = commonUtils.bestSuccessor(possibleSuccessors);
			if (bestSuccessor.gethCost() == currentNode.gethCost()) {
				bestSuccessor =
commonUtils.bestSuccessor(possibleSuccessors);
			}
			if (currentNode.gethCost() == 0) {
				totalSuccessfulMoves = +regularMoves;
				totalSuccessfulIterations++;
				isSuccessful = true;
				if (printOutput) {
					System.out.println("----Goal State-----");
					commonUtils.printNode(currentNode);
				}
			} else if (bestSuccessor.gethCost() < currentNode.gethCost()) {
				// if yes update current Node with best Successor
				sideStepCount = 0;
				regularMoves++;
				updateCurrentNode = true;
			} else if (bestSuccessor.gethCost() == currentNode.gethCost() &&
sideStepCount < sideWalkStepsLimit) { // move


			// sideways


			// updateCurrentNode


			// =true;
				sideStepCount++;
				regularMoves++;
				updateCurrentNode = true;
			} else {
				totalFailedIterations++;
```

```
                        totalFailMoves = +regularMoves;
                        isFail = true;
                        if (printOutput) {
                              System.out.println("----Flat Minimum/Shoulder State-
----");
                              commonUtils.printNode(currentNode);
                        }
                  }
                  if (updateCurrentNode) {
                        currentNode = bestSuccessor;
                  }
            }

            result[0] = totalSuccessfulMoves;
            result[1] = totalSuccessfulIterations;
            result[2] = totalFailMoves;
            result[3] = totalFailedIterations;

            return result;

      }

}
```

RandomRestartForSteepestAscent.java

This class implements Hill Climbing Random Restart without Sideway Moves.

```
import java.util.ArrayList;
import java.util.List;

public class RandomRestartForSteepestAscent {

      public int numberOfQueens = 0;
      public CommonUtils commonUtils = new CommonUtils();
      public boolean isSuccessful = false;
      public boolean isFail = false;
      List<Node> possibleSuccessors = new ArrayList<Node>();
      Node bestSuccessor = new Node();
      int regularMoves = 0;
      int totalSuccessfulMoves = 0;
      int totalFailMoves = 0;
      int totalSuccessfulIterations = 0;
      int totalFailedIterations = 0;
      int totalNumberOfRestarts = 0;
      int[] result = new int[6];
      boolean wasRestartRequired = false;
      int restartUsedCount = 0;

      /**
```

```
 * This method implements Hill Climbing Random Restart without Sideway Moves.
 *
 * @param numberOfQueens the total number of queens on the board.
 * @param printOutput    True if board states are to be printed. False if no
 *                       board states are printed.
 * @return returns an integer array with moves required for success and failure
 *         also includes success and failure iterations count. The array also
 *         includes number of restarts required.
 */
public int[] process(int numberOfQueens, boolean printOutput) {
	this.numberOfQueens = numberOfQueens;

	// Generate a random state with one queen in every column
	Node currentNode = commonUtils.generateRandomState(numberOfQueens);

	// calculate heuristic
	commonUtils.calculateHeuristic(currentNode);
	if (printOutput) {
		System.out.println("----Initial State-----");
		commonUtils.printNode(currentNode);
	}
	while (!isSuccessful) {

		// get possible successors
		possibleSuccessors = commonUtils.findPossibleSuccessors(currentNode);
		// calculate heuristic of possible successors
		for (int i = 0; i < possibleSuccessors.size(); i++) {
			commonUtils.calculateHeuristic(possibleSuccessors.get(i));
		}

		// select best successor if hcost is less than current hcost
		bestSuccessor = commonUtils.bestSuccessor(possibleSuccessors);
		if (currentNode.gethCost() == 0) {
			totalSuccessfulMoves = +regularMoves;
			totalSuccessfulIterations++;
			isSuccessful = true;
			if (printOutput) {
				System.out.println("----Goal State-----");
				commonUtils.printNode(currentNode);
			}
		} else if (bestSuccessor.gethCost() < currentNode.gethCost()) {
			// if yes update current Node with best Successor
			currentNode = bestSuccessor;
			regularMoves++;
```

```
			} else {
				// restart with a new state
				currentNode = commonUtils.generateRandomState(numberOfQueens);
				commonUtils.calculateHeuristic(currentNode);
				totalNumberOfRestarts++;
				wasRestartRequired = true;
			}
		}
		if (wasRestartRequired) {
			restartUsedCount++;
		}

		result[0] = totalSuccessfulMoves;
		result[1] = totalSuccessfulIterations;
		result[2] = totalFailMoves;
		result[3] = totalFailedIterations;
		result[4] = totalNumberOfRestarts;
		result[5] = restartUsedCount;

		return result;

	}

}
```

RandomRestartForSidewaysMoves.java

This class implements Hill Climbing Random Restart with Sideway Moves.

```
import java.util.ArrayList;
import java.util.List;

public class RandomRestartForSidewaysMoves {

	public int numberOfQueens = 0;
	public CommonUtils commonUtils = new CommonUtils();
	public boolean isSuccessful = false;
	public boolean isFail = false;
	List<Node> possibleSuccessors = new ArrayList<Node>();
	Node bestSuccessor = new Node();
	int regularMoves = 0;
	int totalSuccessfulMoves = 0;
	int totalFailMoves = 0;
	int totalSuccessfulIterations = 0;
	int totalFailedIterations = 0;
```

```java
int totalNumberOfRestarts = 0;
int[] result = new int[6];
int sideWalkStepsLimit = 100;
int sideStepCount = 0;
boolean updateCurrentNode = false;
boolean wasRestartRequired = false;
int restartUsedCount = 0;

/**
 * This method implements Hill Climbing Random Restart with Sideway Moves.
 *
 * @param numberOfQueens the total number of queens on the board.
 * @param printOutput    True if board states are to be printed. False if no
 *                  board states are printed.
 * @return returns an integer array with moves required for success and failure
 *        also includes success and failure iterations count. The array also
 *        includes number of restarts required.
 */
public int[] process(int numberOfQueens, boolean printOutput) {
	this.numberOfQueens = numberOfQueens;

	// Generate a random state with one queen in every column
	Node currentNode = commonUtils.generateRandomState(numberOfQueens);

	// calculate heuristic
	commonUtils.calculateHeuristic(currentNode);
	if (printOutput) {
		System.out.println("----Initial State-----");
		commonUtils.printNode(currentNode);
	}
	while (!isSuccessful && !isFail) {
		updateCurrentNode = false;
		// get possible successors
		possibleSuccessors = commonUtils.findPossibleSuccessors(currentNode);
		// calculate heuristic of possible successors
		for (int i = 0; i < possibleSuccessors.size(); i++) {
			commonUtils.calculateHeuristic(possibleSuccessors.get(i));
		}

		// select best successor if hcost is less than current hcost
		bestSuccessor = commonUtils.bestSuccessor(possibleSuccessors);
		if (bestSuccessor.gethCost() == currentNode.gethCost()) {
			bestSuccessor = commonUtils.bestSuccessor(possibleSuccessors);
		}
```

```
				if (currentNode.gethCost() == 0) {
					totalSuccessfulMoves = +regularMoves;
					totalSuccessfulIterations++;
					isSuccessful = true;
					if (printOutput) {
						System.out.println("----Goal State-----");
						commonUtils.printNode(currentNode);
					}
				} else if (bestSuccessor.gethCost() < currentNode.gethCost()) {
					// if yes update current Node with best Successor
					sideStepCount = 0;
					regularMoves++;
					updateCurrentNode = true;
				} else if (bestSuccessor.gethCost() == currentNode.gethCost() && sideStepCount
< sideWalkStepsLimit) { // move


				// sideways


				// updateCurrentNode


				// =true;
					sideStepCount++;
					regularMoves++;
					updateCurrentNode = true;
				} else {
					// restart with a new state
					currentNode = commonUtils.generateRandomState(numberOfQueens);
					commonUtils.calculateHeuristic(currentNode);
					totalNumberOfRestarts++;
					wasRestartRequired = true;
				}
				if (updateCurrentNode) {
					currentNode = bestSuccessor;
					// sideStepCount=0;
				}
			}

			if (wasRestartRequired) {
				restartUsedCount++;
			}
```

```
                result[0] = totalSuccessfulMoves;
                result[1] = totalSuccessfulIterations;
                result[2] = totalFailMoves;
                result[3] = totalFailedIterations;
                result[4] = totalNumberOfRestarts;
                result[5] = restartUsedCount;

                return result;

        }

}
```

RESULTS

Number of Queens	Search Used	Success rate and number of steps	Failure rate and number of steps	Number of Restarts
8-Queens	Hill-Climbing Steepest Accent	Rate: 14.02 % Steps: 3.95	Rate: 85.98 % Steps: 3.2	No restarts
8-Queens	Hill-Climbing with Sideway moves	Rate: 94.3 % Steps: 20.13	Rate: 5.6 % Steps: 63.93	No restarts
8-Queens	Random-restart without Sideway moves	Rate: 100 % Steps: 21.93	Rate: 0 % Steps: 0	6.3 Restarts
8-Queens	Random-restart with Sideway moves	Rate: 100 % Steps: 25.4	Rate: 0 % Steps: 0	1.06 Restarts

OBESERVATIONS

The failure rate is high when steepest accent method is used and it reduces drastically from 85% to 6% when sideway moves are used and to 0 % when Random restart method is used. As success is expensive each of these processes results in increasing steps to reach solution state.