DEPARTMENT OF COMPUTER SCIENCE

ITCS 6150 – INTELLIGENT SYSTEMS

Programming Project -2

Solving N-queens problem using hill-climbing and its variants

SUBMITTED TO

Dewan T. Ahmed, Ph.D.

SUBMITTED BY:

Akhil Chundarathil	**ID# 801137901**
Ravi Goalla	**ID# 801134121**
Nikhil Kumar Mutyala	**ID# 801136096**
Venkata Sai Koushik Koritala	**ID# 801135486**

Table Of Contents

1. Introduction……………………………………………………………..3

2. Program and Design…………………………………………………10

3. Outputs……………………………………………………………….26

4. Results…………………………………………………………….....30

5. Conclusion……………………………………………………….....31

1. INTRODUCTION

1.1.N-Queens Problem

The N-Queens problem is a chess board problem in which 'N' represents the size of the chessboard and the number of queens that we are considering in the problem. The N-Queen problem essentially is the arrangement of 'N' number of queens on a NxN size chessboard, We know that the Queen in chess has row, column and diagonal moves in all possible directions, The solution is achieved when the arrangement of Queens is done in such a way that they do not attack each other. For this there is a possible solution for all natural number values of 'N' except for 2 and 3.

1.2.Solution For N-Queens Problem

The N-Queens problem is solved by searching for the solution using the Hill Climbing search algorithm. The searching involves a sequence of actions that lead us from the initial state to the final state or the goal state in which the queens do not attack each other.

1.3.Hill Climbing Technique

Hill Climbing is a mathematical optimization technique. Hill climbing search belongs to a family of local search. This technique tries to find the best solution possible by making changes to the position of the N-Queens by an iterative process. Here for solving the N-Queens problem by using the Hill Climbing technique, we use a heuristic which is the number of queens which can be attacked by other queens. So, we take this heuristic under consideration while searching for a new solution in a new iteration of the hill climbing algorithm. So in the new iteration if the changes produce a better solution, it is considered as the new solution. This process continues until no further improvement can be found. In this way the solution is found for the N-Queens problem using the Hill Climbing algorithm.

There are different ways in which the Hill Climbing algorithm can be implemented. The following are the different types of hill climb search we used.

1.3.1.Simple Hill Climbing

This method of the hill climb search is the orthodox way of searching which just involves choosing a better solution one after the other in the increasing iterations.

Average success rate:14%

Average failure rate: 86%

The average number of steps to succeed: 4

The average number of steps to fail:3

1.3.2.Hill Climbing with sideways moves

This version makes sideways movements in the expectation of becoming a shoulder on the plateau. The amount of side-moves is reduced to stop infinite side-moves.

Average success rate:94%

Average failure rate: 6%

The average number of steps to succeed: 21

The average number of steps to fail: 64

1.3.3.Random restart without sideways moves

Random hill-climbing restart looks iteratively for the goal state from randomly created initial states. Random initial state restarts occur until approaching the global maximum. This is the optimal solution.

Average success rate: 100 %

The average number of steps to succeed: 22

Number of restarts needed: 6 (7-iterations)

1.3.4.Random restart with sideways moves

This version allows sideways moves with random restart hill-climbing variants. The probability of finding the goal is almost 1 here as it will eventually find the goal at some point.

Average success rate: 100 %

The average number of steps to succeed: 25

Number of restarts needed: 1

1.4.Heuristic of the N-Queens problem

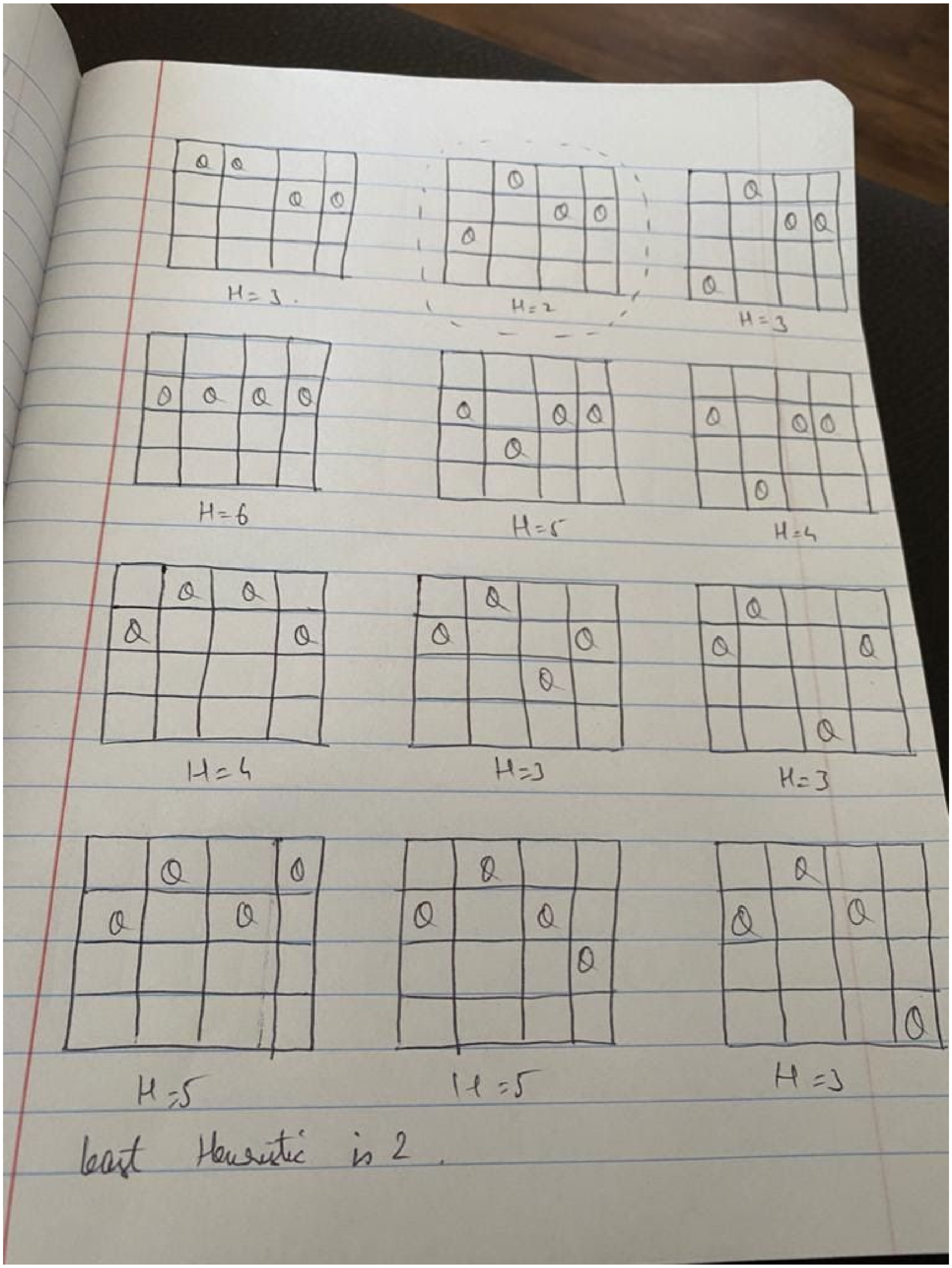
The heuristic function for the N-queens problem is the number of pairs of queens that attack each other directly or indirectly. whenever a successor is to be chosen, the one with the lower heuristic is chosen as the best successor.

Sample 4-queens problem solved using Hill-Climbing search Let's generate a random state with each queen placed in a column. Actions: We are solving for the 4-queen problem here.

	Q		
Q		**Q**	**Q**

H=5

The heuristic value for the above problem is five since there are five pairs of queens that are attacking each other at this moment. The above state results in twelve possible states with different heuristic values. The state with the lowest heuristic value will be chosen as the best successor. Let's calculate all the states and find the next best successor.



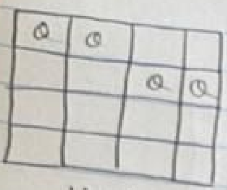
least Heuristic is 2.

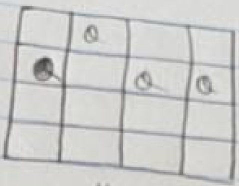
From the above twelve states, the least heuristic value is 2. So, the best successor is the state with the least heuristic value is:

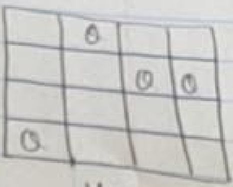
	Q		
		Q	**Q**
Q			

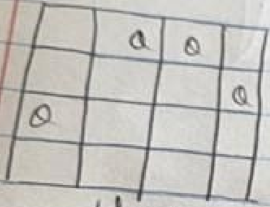
H=2

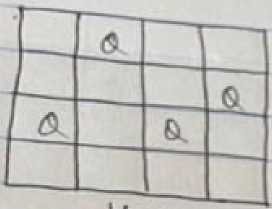
Let’s calculate the best successor for the above state by moving queen each time.

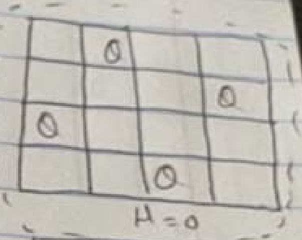

H = 3

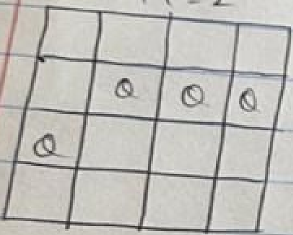

H = 4

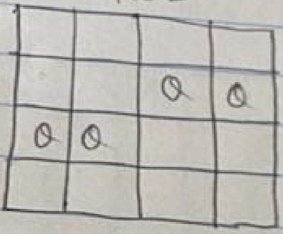

H = 3

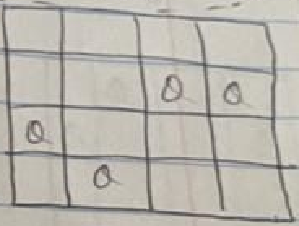

H = 2

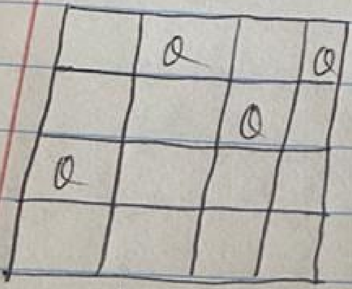

H = 2

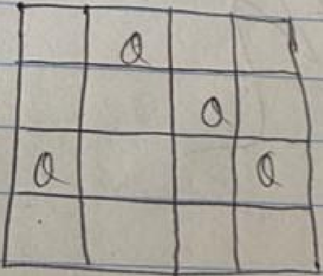

H = 0

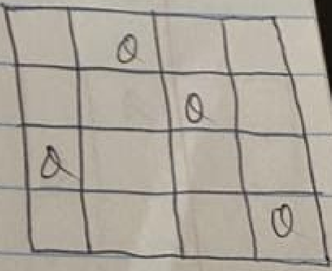

H = 4


H = 3


H = 3


H = 3


H = 4


H = 1

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 Goal state Obtained:

	Q		
			Q
Q			
		Q	

H=0

2. PROGRAM AND DESIGN

Queen, Board, HillClimbing are the classes created

2.1.Class description

Queen.java:

Instance Variables:

row: To indicate in which row the queen is present.

column: To indicate in which column the queen is present.

Methods:

getRow( ): To get the row in which the queen is present.

setRow( ): To set the row in which the queen is present.

getColumn( ): To get the column in which the queen is present.

setColumn( ): To set the column in which the queen is present.

Queen: This is a constructor used to set the row and column for the queen object.

Walk_up( ): To make the queen walk up i.e. decrease the row number.

Walk_down( ): To make the queen walk down i.e. increase the row number.

toString( ): This function prints the row and column of the queen as a string.

Program:

```
public class Queen implements Cloneable{
  int row;
  int column;

  public int getRow() {
     return row;
  }

  public void setRow(int row) {
     this.row = row;
  }

  public int getColumn() {
     return column;
```

```
    }

    public void setColumn(int column) {
        this.column = column;
    }

    public Queen(int row, int column) {
        this.row = row;
        this.column = column;
    }

    boolean walk_up(int n) {
        if(row > 0) {
            --row;
            return true;
        }
        return false;
    }

    boolean walk_down(int n) {
        if(row < n-1) {
            ++row;
            return true;
        }
        return false;
    }

    @Override
    public String toString() {
        return "Queen: Row = "+row+" Column = "+column;
    }
}
```

<u>Board.java:</u>

Instance Variables:

dimensions: This is the dimension of the chess board.

queens[]: This is an array list of queens present on the chess board.

successor_index: Index of the successor node from the array list of successors.

successors[]: Array list of successor nodes generated.

Methods:

setQueens( ): Sets the queen into the instance variable queens.

Board: This is a constructor of the class which creates the board and sets queens.

addQueen( ): This adds the queen if it fits in the board created.

getSuccessors( ): Returns a list of successor nodes to a current node.

gethCost( ): Returns the heuristic cost of the current board.

getNextSuccessor( ): It returns the next successor in the successors list.

getBestSuccessor( ): It finds the best out of all the generated successors.

printQueens( ): It prints the current State in board format.

Program:

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

public class Board {
  int dimensions;
  ArrayList<Queen> queens = new ArrayList<>();
  //int[] array = new int[]{4,5,6,3,4,5,6,5};
  int successor_index = -1;
  ArrayList<Board> successors = new ArrayList<>();

  /**
   * It sets the list of Queens
   * @param queens list of queens
   */
  public void setQueens(ArrayList<Queen> queens) {
     this.queens = queens;
  }

  /**
   * It generates the list of queens with its random positions on the board
   * @param dimensions
   * @param isRandomPlacement indicates Whether to place the queens randomly or
not
   */
  public Board(int dimensions, boolean isRandomPlacement) {
     this.dimensions = dimensions;

     if(isRandomPlacement) {
         Random r = new Random();
         for (int i = 0; i < dimensions; i++) {
             Queen q = new Queen(r.nextInt(dimensions), i);
             this.addQueen(q);
         }
     }
  }

  /**
   * It adds the queen into the current list of queens
   * @param q Queen to be added to the list
   */
  public void addQueen(Queen q){
     if(queens.size() < dimensions)
         queens.add(q);
  }

  /**
   * It generates the successors of the current board
   * @return returns the successors generated
   */
  public ArrayList<Board> getSuccessors(){
     ArrayList<Board> successors = new ArrayList<>();
     int prev_row;
     int prev_column;

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

         prev_row = queens.get(i).getRow();
         prev_column = queens.get(i).getColumn();

         while(queens.get(i).walk_up(dimensions)){
             ArrayList<Queen> newqueenlist = new ArrayList<>();
             Board b = new Board(dimensions, false);
             for(Queen q : queens){
                 newqueenlist.add(new Queen(q.getRow(), q.getColumn()));
             }
             b.setQueens(newqueenlist);
             successors.add(b);
         }
```

```
            queens.get(i).setRow(prev_row);
            queens.get(i).setColumn(prev_column);

            while(queens.get(i).walk_down(dimensions)){
                ArrayList<Queen> newqueenlist = new ArrayList<>();
                Board b = new Board(dimensions, false);
                for(Queen q : queens){
                    newqueenlist.add(new Queen(q.getRow(), q.getColumn()));
                }
                b.setQueens(newqueenlist);
                successors.add(b);
            }
            queens.get(i).setRow(prev_row);
            queens.get(i).setColumn(prev_column);
        }
        return successors;
    }

    /**
     * It computes the heuristic cost (cost of collisions) of the current board
     * @return returns the heuristic cost
     */
    public int gethCost(){
        int rowattacks = 0;
        int[] row_wise = new int[dimensions];
        for(int i = 0; i < dimensions; i++)
            row_wise[i] = 0;
        // variable for counting number of attacks in diagonals
        int diagonalattacks = 0;

        int[] diagonal_wise = new int[2*dimensions-1];
        int[] diagonal_wise_2 = new int[2*dimensions-1];

        for(int i = 0; i < 2*dimensions-1; i++) {
            diagonal_wise[i] = 0;
            diagonal_wise_2[i] = 0;
        }
        // counting number of queens
        int x, y;
        for(int i = 0; i < dimensions; i++) {
            x = queens.get(i).getRow();
            y = queens.get(i).getColumn();
            ++row_wise[x]; // in each row
            ++diagonal_wise[x+y]; // in each antidiagonal direction
            ++diagonal_wise_2[dimensions-1+y-x]; // in each main diagonal direction
        }
        // number of attacks in rows
        for(int i = 0; i < dimensions; i++)
            if(row_wise[i] > 1)
                rowattacks += row_wise[i]*(row_wise[i]-1)/2;
        // number of attacks in diagonals
        for(int i = 0; i < 2*dimensions-1; i++) {
            if(diagonal_wise[i] > 1)
                diagonalattacks += diagonal_wise[i]*(diagonal_wise[i]-1)/2;
            if(diagonal_wise_2[i] > 1)
                diagonalattacks += diagonal_wise_2[i]*(diagonal_wise_2[i]-1)/2;
        }
        return diagonalattacks + rowattacks;
    }

    /**
     * It returns the next successor in the successors list
     * @return
     */
    public Board getNextSuccessor(){
        if(successor_index == -1){
            successors = getSuccessors();
            return successors.get(++successor_index);
        }
        if(++successor_index < successors.size())
            return successors.get(successor_index);

        successor_index = -1;
        return null;
    }

    /**
     * It finds the best successor out of all the generated successors
     * @return returns the best successor
     */
    public Board getBestSuccessor(){
        ArrayList<Board> successors = getSuccessors();
        ArrayList<Board> bestsuccessors = new ArrayList<>();
        Random r = new Random();

        int minattacks = dimensions*dimensions;
        int currentboardattack;

        for (int i = 0; i < successors.size(); i++){
            currentboardattack = successors.get(i).gethCost();
            if(minattacks > currentboardattack){
                minattacks = currentboardattack;
                bestsuccessors.clear();
                bestsuccessors.add(successors.get(i));
            }
            else if(minattacks == currentboardattack){
                bestsuccessors.add(successors.get(i));
            }
        }
        return bestsuccessors.get(r.nextInt(bestsuccessors.size()));
    }

    /**
     * It prints the current State in board format
     */
    public void printQueens(){
        char[][] NbyNBoard = new char[dimensions][dimensions];

        //System.out.println(this.queens);
        for(int i = 0; i < dimensions; i++)
            for(int j = 0; j < dimensions; j++)
                NbyNBoard[i][j] = '_';
        // filling board with its queens
        for(int i = 0; i < dimensions; i++)
            NbyNBoard[queens.get(i).getRow()][queens.get(i).getColumn()] = 'Q';
        // actually printing
        for(int i = 0; i < dimensions; i++) {
            System.out.println("");
            for (int j = 0; j < dimensions; j++)
                System.out.print(NbyNBoard[i][j]+" ");
        }
        System.out.print("\n\n"+"collisions = " + gethCost()+"\n\n");

    }

}
```

HillClimbing.java:

Instance Variables:

isSuccess: indicates if the Hill climbing search was success.

isFailure: indicates if the Hill climbing search was failure..

successMoves: Number of moves made for the success case.

failureMoves: Number of moves made for the failure case.

Moves: Number of moves made from start to end of the search.

sideMoveCount: Number of sideway moves made.

randomRestartCount: indicates if the current search was a restarted search

noOfRestarts: Number of restarts made in the restarted search.

isRestarted: indicates if any restart made at any point in the search.

sideMovesLimit: Limit for the number of sideway moves.

Methods:

SimpleHillClimbing( ): It performs Hill Climbing the traditional one.

HillClimbingWithSidewaysMove( ): Performs hill climbing with sideways steps.

RandomRestartWithoutSidewaysMove( )

RandomRestartWithSidewaysMove( )

main( ): This is the entry point of the program.

Program:

```
import java.util.Random;
import java.util.Scanner;

public class HillClimbing {
  int isSuccess;
  int isFailure;
```

```
    int successMoves;
    int failureMoves;
    int moves;
    int sideMoveCount;
    int randomRestartCount;
    int noOfRestarts;
    boolean isRestarted = false;

    int sideMovesLimit = 100;

    /**
     * It performs Hill Climbing search by creating board with given dimensions.
     * @param dimensions dimensions of the board
     * @param taskToBeDone indicates the type of task (Report Generation or Search
  Sequence Generation)
     * @return returns the metrics of the search performed
     */
    public int[] SimpleHillClimbing(int dimensions, int taskToBeDone){
        Board board = new Board(dimensions, true);
        System.out.println("-------------------------------------------------------------------------");
        System.out.println("**********Initial State**********");
        board.printQueens();

        while(true){
            Board bestsuccessor = board.getBestSuccessor();
            int hcost = bestsuccessor.gethCost();
            int boardhcost = board.gethCost();

            if(hcost == 0){
                if(taskToBeDone == 2) {
                    System.out.println("\t|");
                    System.out.println("\t|");
                    System.out.println("\tV");
                }
                System.out.println("*****Goal State******");
                bestsuccessor.printQueens();

System.out.println("----------------------------------------------------------------------");
                ++isSuccess;
                successMoves += moves;
                break;
            }
            else if(hcost < boardhcost){
                board = bestsuccessor;
                if(taskToBeDone == 2) {
                    System.out.println("\t|");
                    System.out.println("\t|");
                    System.out.println("\tV");
                    board.printQueens();
                }
                ++moves;
            }
            else {
                if(taskToBeDone == 2) {
                    System.out.println("\t|");
                    System.out.println("\t|");
                    System.out.println("\tV");
                }
                System.out.println("******Reached plateau/Local Minima State*****");
                bestsuccessor.printQueens();

System.out.println("----------------------------------------------------------------------");
                ++isFailure;
                failureMoves += moves;
                return new int[]{isSuccess, successMoves, isFailure, failureMoves};
            }
        }
        return new int[] {isSuccess, successMoves, isFailure, failureMoves};

    }
    /**
     * It performs Hill Climbing with Sideways Move search by creating board with given
dimensions.
     * @param dimensions dimensions of the board
     * @param taskToBeDone indicates the type of task (Report Generation or Search
Sequence Generation)
     * @return returns the metrics of the search performed
     */
    public int[] HillClimbingWithSidewaysMove(int dimensions, int taskToBeDone){

        Board board = new Board(dimensions, true);
        System.out.println("---------------------------------------------------------------------");
        System.out.println("**********Initial State**********");
        board.printQueens();

        while (true){
            Board bestsuccessor = board.getBestSuccessor();

            int hcost = bestsuccessor.gethCost();
            int boardhcost = board.gethCost();

            if(hcost == 0){
                if(taskToBeDone == 2) {
                    System.out.println("\t|");
                    System.out.println("\t|");
                    System.out.println("\tV");
                }
                System.out.println("*****Goal State******");
                bestsuccessor.printQueens();
```

```
System.out.println("------------------------------------------------------------------------");
            ++isSuccess;
            successMoves += moves;
            break;
        }
        else if(hcost < boardhcost || (hcost == boardhcost && sideMoveCount <
sideMovesLimit)){
            if(hcost < boardhcost)
                sideMoveCount = 0;
            else
                ++sideMoveCount;

            board = bestsuccessor;

            if(taskToBeDone == 2) {
                System.out.println("\t|");
                System.out.println("\t|");
                System.out.println("\tV");
                board.printQueens();
            }
            ++moves;
        }
        else{
            if(taskToBeDone == 2) {
                System.out.println("\t|");
                System.out.println("\t|");
                System.out.println("\tV");
            }
            System.out.println("******Reached plateau/Local Minima State*****");
            bestsuccessor.printQueens();

System.out.println("------------------------------------------------------------------------");
            ++isFailure;
            failureMoves += moves;
            return new int[]{isSuccess, successMoves, isFailure, failureMoves};
         }
      }

      return new int[] {isSuccess, successMoves, isFailure, failureMoves};
   }

   /**
    * It performs Random Restart Hill Climbing Without Sideways Move search by
creating board with given dimensions.
    * @param dimensions dimensions of the board
    * @param taskToBeDone indicates the type of task (Report Generation or Search
Sequence Generation)
    * @return returns the metrics of the search performed
    */
   public int[] RandomRestartWithoutSidewaysMove(int dimensions, int
taskToBeDone){
      Board board = new Board(dimensions, true);
      System.out.println("----------------------------------------------------------------------");
      System.out.println("**********Initial State**********");
      board.printQueens();

      while (true){
         Board bestsuccessor = board.getBestSuccessor();

         int hcost = bestsuccessor.gethCost();
         int boardhcost = board.gethCost();

         if(hcost == 0){
            if(taskToBeDone == 2) {
               System.out.println("\t|");
               System.out.println("\t|");
               System.out.println("\tV");
```

```
        }
        System.out.println("********Goal State*********");
        bestsuccessor.printQueens();
        ++isSuccess;
        successMoves += moves;
        break;
    }
    else if(hcost < boardhcost){
        board = bestsuccessor;
        if(taskToBeDone == 2) {
            System.out.println("\t|");
            System.out.println("\t|");
            System.out.println("\tV");
            board.printQueens();
        }
        ++moves;
    }
    else{
        board = new Board(dimensions, true);
        if(taskToBeDone == 2) {
            System.out.println("\t|");
            System.out.println("\t|");
            System.out.println("\tV");
            System.out.println("*********Restarted************");
            board.printQueens();
        }
        ++noOfRestarts;
        isRestarted = true;
    }
}

if(isRestarted)
    ++randomRestartCount;
```

```
        return new int[] {isSuccess, successMoves, 0, 0, randomRestartCount,
noOfRestarts};
    }


    /**
     * It performs Random Restart Hill Climbing With Sideways Move search by creating
board with given dimensions.
     * @param dimensions dimensions of the board
     * @param taskToBeDone indicates the type of task (Report Generation or Search
Sequence Generation)
     * @return returns the metrics of the search performed
     */
    public int[] RandomRestartWithSidewaysMove(int dimensions, int taskToBeDone){
        Board board = new Board(dimensions, true);
        System.out.println("-----------------------------------------------------------------------");
        System.out.println("**********Initial State**********");
        board.printQueens();

        while (true){
            Board bestsuccessor = board.getBestSuccessor();

            int hcost = bestsuccessor.gethCost();
            int boardhcost = board.gethCost();

            if(hcost == 0){
                if(taskToBeDone == 2) {
                    System.out.println("\t|");
                    System.out.println("\t|");
                    System.out.println("\tV");
                }
                System.out.println("*****Goal State******");
                bestsuccessor.printQueens();
                ++isSuccess;
                successMoves += moves;
                break;
            }
else if(hcost < boardhcost || (hcost == boardhcost && sideMoveCount < sideMovesLimit)){
                if(hcost < boardhcost)
                    sideMoveCount = 0;
                else
                    ++sideMoveCount;

                board = bestsuccessor;

                if(taskToBeDone == 2) {
                    System.out.println("\t|");
                    System.out.println("\t|");
                    System.out.println("\tV");
                    board.printQueens();
                }
                ++moves;
            }
            else{
                board = new Board(dimensions, true);
                if(taskToBeDone == 2) {
                    System.out.println("\t|");
                    System.out.println("\t|");
                    System.out.println("\tV");
                    System.out.println("*********Restarted************");
                    board.printQueens();
                }
                ++noOfRestarts;
                isRestarted = true;
            }
        }
        if(isRestarted)
            ++randomRestartCount;

        return new int[] {isSuccess, successMoves, 0, 0, randomRestartCount, noOfRestarts};
    }

    public static void main(String[] args){
        Scanner scanner = new Scanner(System.in);
        Random rand = new Random();
        int iterations = rand.nextInt(300) + 200; // as expected iterations are generated
random
        int taskToBeDone;
        int noOfQueens;
        int algorithm;
        int[] metrics = new int[4];
        double totalSuccess = 0.0f, totalFailure = 0, totalSuccessSteps = 0, totalFailureSteps
= 0, successRate=0, failureRate=0, totalRestarts=0, totalNoOfRestarts=0, avgRestarts=0,
avgSucessSteps=0, avgFailureSteps=0;
        String[] algorithms = new String[] {"Hill Climbing Search", "Hill Climbing with
Sideways Move", "Random Restart without SideWays Move", "Random Restart
with Sideways Move"};

        System.out.println("Enter the value for number of queens");
        noOfQueens = scanner.nextInt();
        System.out.println("Enter 1 for hill climbing search");
        System.out.println("Enter 2 for hill climbing search with sideways move");
        System.out.println("Enter 3 for random-restart hill-climbing without sideways
move");
        System.out.println("Enter 4 for random-restart hill-climbing with sideways
move");
        algorithm =scanner.nextInt();

        System.out.println("Enter 1 for Report Generation");
        System.out.println("Enter 2 for Search Sequences of 4 Random
Configurations");
        taskToBeDone = scanner.nextInt();
        if(taskToBeDone == 2)
            iterations = 4;

        switch (algorithm){
            case 1:
                for(int i=0; i < iterations; i++){
                    HillClimbing hillClimbing = new HillClimbing();
                    metrics = hillClimbing.SimpleHillClimbing(noOfQueens, taskToBeDone);
                    totalSuccess += metrics[0];
                    totalSuccessSteps += metrics[1];
                    totalFailure += metrics[2];
                    totalFailureSteps += metrics[3];
                }
                break;
            case 2:
                for(int i=0; i < iterations; i++){
                    HillClimbing hillClimbing = new HillClimbing();
                    metrics = hillClimbing.HillClimbingWithSidewaysMove(noOfQueens,
taskToBeDone);
                    totalSuccess += metrics[0];
                    totalSuccessSteps += metrics[1];
                    totalFailure += metrics[2];
                    totalFailureSteps += metrics[3];
                }
                break;
            case 3:
                for(int i=0; i < iterations; i++){
                    HillClimbing hillClimbing = new HillClimbing();
                    metrics = hillClimbing.RandomRestartWithoutSidewaysMove(noOfQueens,
taskToBeDone);
                    totalSuccess += metrics[0];
                    totalSuccessSteps += metrics[1];
                    totalFailure += metrics[2];
                    totalFailureSteps += metrics[3];
                    totalRestarts += metrics[4];
                    totalNoOfRestarts += metrics[5];
                }
                break;
            case 4:
                for(int i=0; i < iterations; i++){
                    HillClimbing hillClimbing = new HillClimbing();
                    metrics = hillClimbing.RandomRestartWithSidewaysMove(noOfQueens,
taskToBeDone);
                    totalSuccess += metrics[0];
                    totalSuccessSteps += metrics[1];
                    totalFailure += metrics[2];
                    totalFailureSteps += metrics[3];
                    totalRestarts += metrics[4];
                    totalNoOfRestarts += metrics[5];
                }
                break;
        }

        if(taskToBeDone != 2) {

            if (totalSuccess != 0) {
                successRate = (totalSuccess / iterations) * 100;
                avgSucessSteps = totalSuccessSteps / totalSuccess;
            }

            if (totalFailure != 0) {
                failureRate = (totalFailure / iterations) * 100;
                avgFailureSteps = totalFailureSteps / totalFailure;
            }

            if (totalRestarts != 0)
```

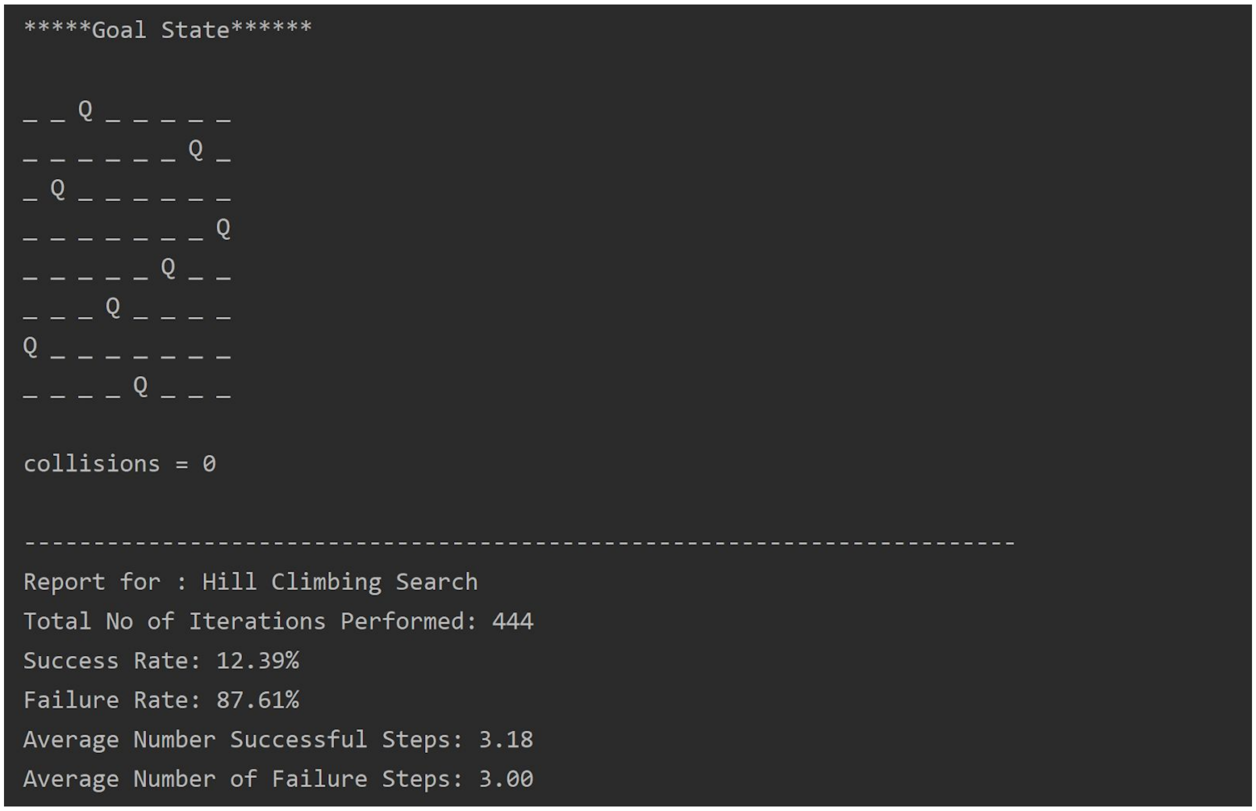
```
            avgRestarts = totalNoOfRestarts / totalRestarts;


        System.out.println("Report for : " + algorithms[algorithm-1]);
        System.out.println("Total No of Iterations Performed: " + iterations);
        System.out.println("Success Rate: " + String.format("%.2f", successRate) +
"%");
        System.out.println("Failure Rate: " + String.format("%.2f", failureRate) +
"%");
        System.out.println("Average Number Successful Steps: " +
String.format("%.2f", avgSucessSteps));
        System.out.println("Average Number of Failure Steps: " +
String.format("%.2f", avgFailureSteps));
        if(algorithm != 2 && algorithm != 1) {
            System.out.println("Average Number of Restarts Done: " +
String.format("%.2f", avgRestarts));
        }
    }


  }
}
```

3. OUTPUTS

3.1.Simple Hill Climbing



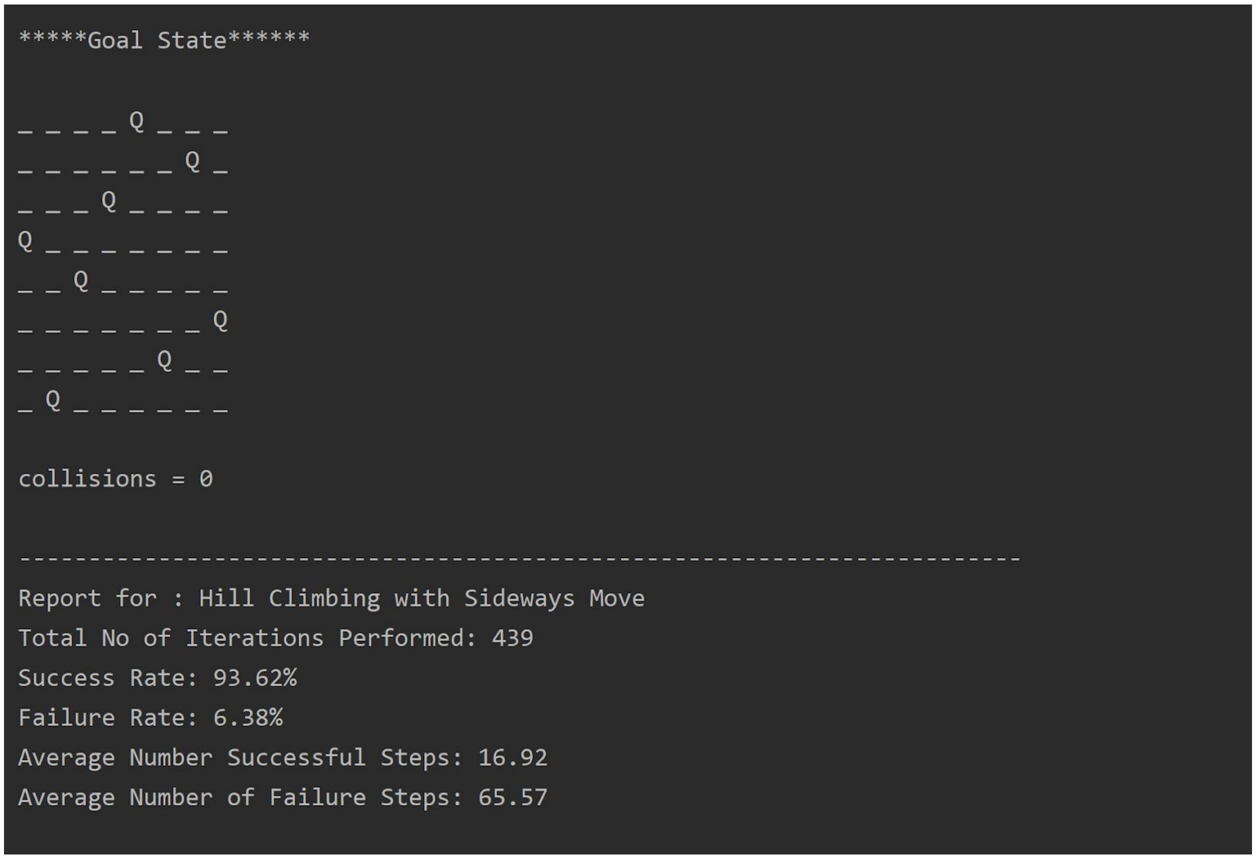
```
*****Goal State******


_ _ Q _ _ _ _ _
_ _ _ _ _ _ Q _
_ Q _ _ _ _ _ _
_ _ _ _ _ _ _ Q
_ _ _ _ _ Q _ _
_ _ _ Q _ _ _ _
Q _ _ _ _ _ _ _
_ _ _ _ Q _ _ _

collisions = 0


------------------------------------------------------------------------
Report for : Hill Climbing Search
Total No of Iterations Performed: 444
Success Rate: 12.39%
Failure Rate: 87.61%
Average Number Successful Steps: 3.18
Average Number of Failure Steps: 3.00
```

3.2.Hill Climbing with sideways moves



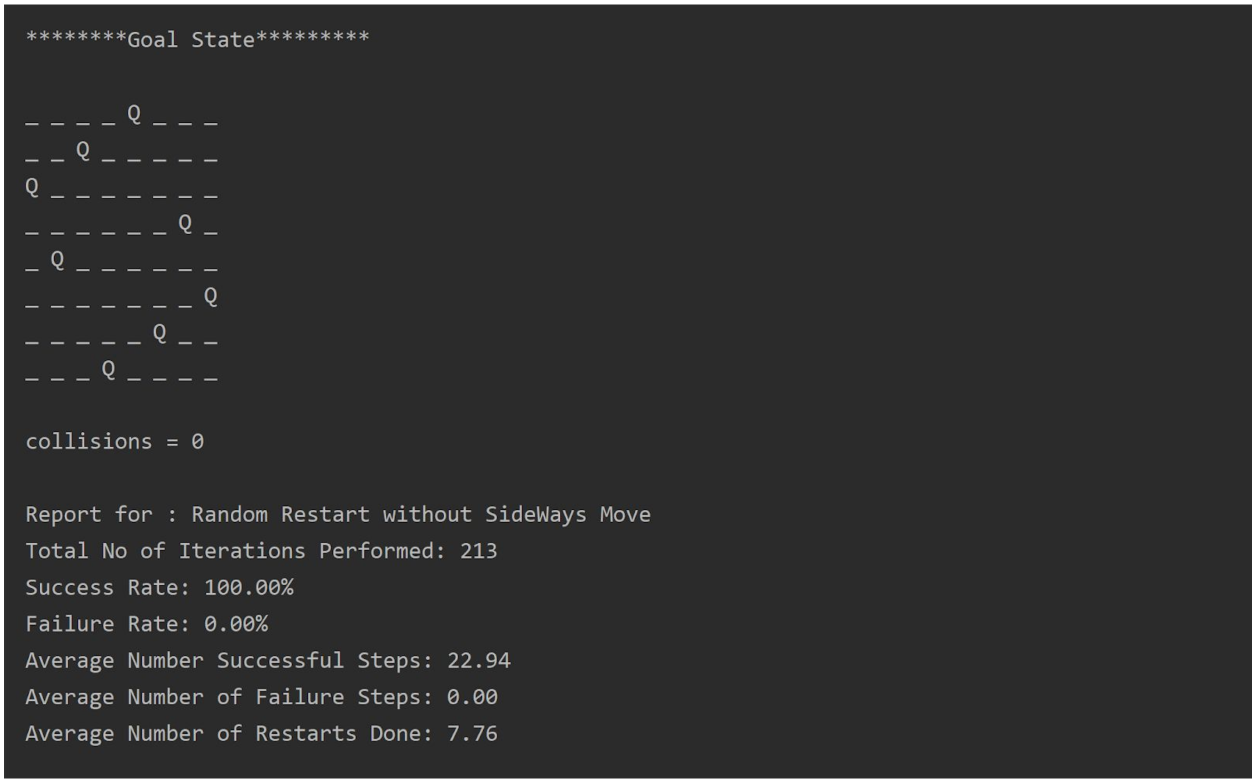
```
*****Goal State******

_ _ _ _ Q _ _ _
_ _ _ _ _ _ Q _
_ _ _ Q _ _ _ _
Q _ _ _ _ _ _ _
_ _ Q _ _ _ _ _
_ _ _ _ _ _ _ Q
_ _ _ _ _ Q _ _
_ Q _ _ _ _ _ _

collisions = 0

-------------------------------------------------------------------------
Report for : Hill Climbing with Sideways Move
Total No of Iterations Performed: 439
Success Rate: 93.62%
Failure Rate: 6.38%
Average Number Successful Steps: 16.92
Average Number of Failure Steps: 65.57
```

3.3.Random Restart Hill Climbing without sideways moves



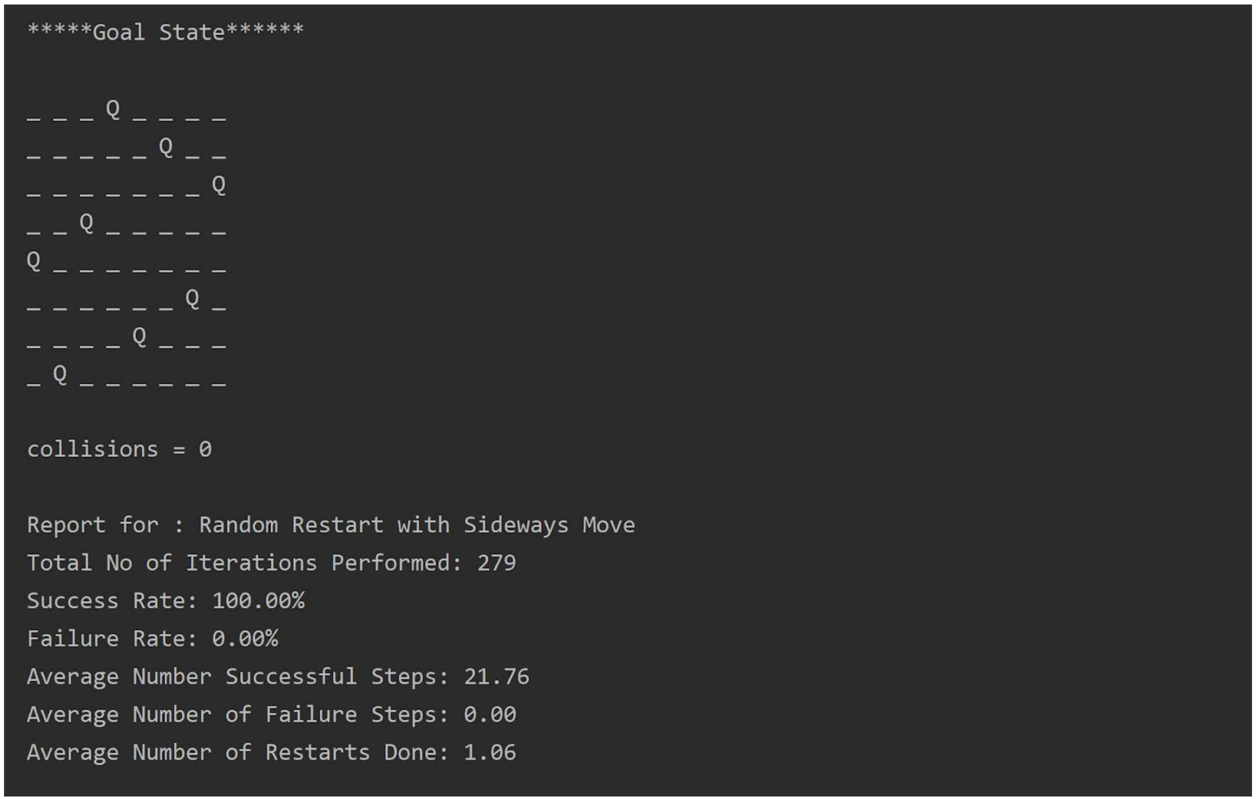
```
********Goal State*********


_ _ _ _ Q _ _ _
_ _ Q _ _ _ _ _
Q _ _ _ _ _ _ _
_ _ _ _ _ _ Q _
_ Q _ _ _ _ _ _
_ _ _ _ _ _ _ Q
_ _ _ _ _ Q _ _
_ _ _ Q _ _ _ _

collisions = 0

Report for : Random Restart without SideWays Move
Total No of Iterations Performed: 213
Success Rate: 100.00%
Failure Rate: 0.00%
Average Number Successful Steps: 22.94
Average Number of Failure Steps: 0.00
Average Number of Restarts Done: 7.76
```

3.4.Random Restart Hill Climbing with sideways moves



```
*****Goal State******


_ _ _ Q _ _ _ _
_ _ _ _ _ Q _ _
_ _ _ _ _ _ _ Q
_ _ Q _ _ _ _ _
Q _ _ _ _ _ _ _
_ _ _ _ _ _ Q _
_ _ _ _ Q _ _ _
_ Q _ _ _ _ _ _

collisions = 0

Report for : Random Restart with Sideways Move
Total No of Iterations Performed: 279
Success Rate: 100.00%
Failure Rate: 0.00%
Average Number Successful Steps: 21.76
Average Number of Failure Steps: 0.00
Average Number of Restarts Done: 1.06
```

4. RESULTS

NUMBER OF QUEENS	VARIANT	NUMBER OF RESTARTS	SUCCESS RATE, NUMBER OF STEPS	FAILURE RATE, NUMBER OF STEPS
8- Queens	Hill Climbing without Sideways	N/A	12.39%, 3.18	87.61%, 3.00
8- Queens	Hill climbing with sideways moves	N/A	93.62%, 16.92	6.38%, 65.57
8- Queens	Random restart with sideways moves	1.06	100.00%, 21.76	0.00%, 0.00
8- Queens	Random restart without sideways moves	7.76	100.00%, 22.94	0.00%, 0.00

5. CONCLUSION

The N-Queens problem was successfully implemented using a Hill Climbing algorithm using its variants Simple Hill Climbing, Hill Climbing With sideways moves, Random Restart Hill Climbing without sideways moves, Random Restart Hill Climbing with sideways moves. The success rate, failure rate and the corresponding number of steps for each version of the Hill Climbing algorithm were also listed in a tabular form.