**Programming Project 2:**

**Solving the n-queens problem by random-restart hill climbing**

*Pedram Bashiri*

*800874993*

## Problem Description

The goal of the N-queens problem is to place N queens on a chessboard such that no queen attacks any other. (A queen attacks any piece in the same row, column or diagonal). Only boards of size four or higher have solutions, and there is usually more than one solution per board, of course, some of these solutions are mirrors of others. The solution to the four-queen problem is an example of one such board that has only one solution that can be mirrored only. All subsequent boards have more than one arrangement. As queens are placed on the board, the lines of force that they exhibit down the ranks, files, and diagonals drastically decreases the search space as they are added, such that when n – 1 queens have been placed, only one position is left open for the remaining queen in the case of a solution. In the case of a board not having a solution, the entire board has been placed in danger when the n – 1th queen has been placed, if not before.

## Hill Climbing Algorithm

The hill-climbing search algorithm (steepest-ascent version) is simply a loop that continually moves in the direction of increasing value—that is, uphill. It terminates when it reaches a “peak” where no neighbor has a higher value.

This algorithm randomly generates a parent array, as well as (n \* 2) children arrays. A heuristic function is run on the parent array and the array is then copied to all the children arrays. One bit is flipped in per child to make it different from the parent. A new heuristic is computed for the child, and the best child is taken to be the next parent.

Unfortunately, hill climbing often gets stuck for the following reason:

Local maxima: a local maximum is a peak that is higher than each of its neighboring states but lower than the global maximum. Hill-climbing algorithms that reach the vicinity of a local maximum will be drawn upward toward the peak but will then be stuck with nowhere else to go.

Thus, the algorithm is incomplete - it often fails to find a goal when one exists because it can get stuck on local maxima. Random-restart hill climbing adopts the well-known adage, “If at first you don’t succeed, try, and try again.” It conducts a series of hill-climbing searches from randomly generated initial states, until a goal is found. It is trivially complete with probability approaching 1, because it will eventually generate a goal state as the initial state.

## Problem Formulation

**Search Algorithm**: Random-Restart Hill Climbing

**Problem Formulation:**

**Initial State:** No queen is placed on the board.

**Goal State:** Goal is a state in which all N queens are placed on the chess board and no 2 queens attack each other.

**Possible Actions:** Put a queen on a board piece (tile), remove a queen from a piece

**State Space:** N\*N array of chars, Q for a queen, 1 for under attack tiles, and 0 for tiles with no attack on

**Path:** Does not matter, in this problem the only thing matters is the goal state.

**Heuristic Function:** Number of conflicting queens in each state.

## Implementation

**Main Class:**

The class that contains the main function. The program starts with this file. The program first asks the number of queens (or the chess board dimension) from user and then call Board class to design and solve the problem.

**Queen Class:**

This Class has three variables, *row* and *column* for storing the position of the queen and a Boolean variable *inPosition* that tells whether or not the queen has been placed on the board.

**Tile Class:**

This class represents every piece of the chess board and has only two variables, *occupied* to specify if the tile has a queen in it, and *attacked* to tell if the tile is under attack by any queen.

**Board Class:**

This is the class that defines how the problem looks like, as well as contains the Random-Restart Algorithm to solve the problem. For this class, there is a single integer variable, *N*, which defines how large the board is for a particular problem. A two-dimensional array of *tiles* and an array of *queens* are also defined in the Board class.

*int heuristic(int[] board)* function in Board class counts and returns the number of conflicting queens.

*long[] randomRestartHillClimbing()* function applies the algorithm, solves the problem and returns two values, number of hill climbing steps (state changes) and number of restarts.

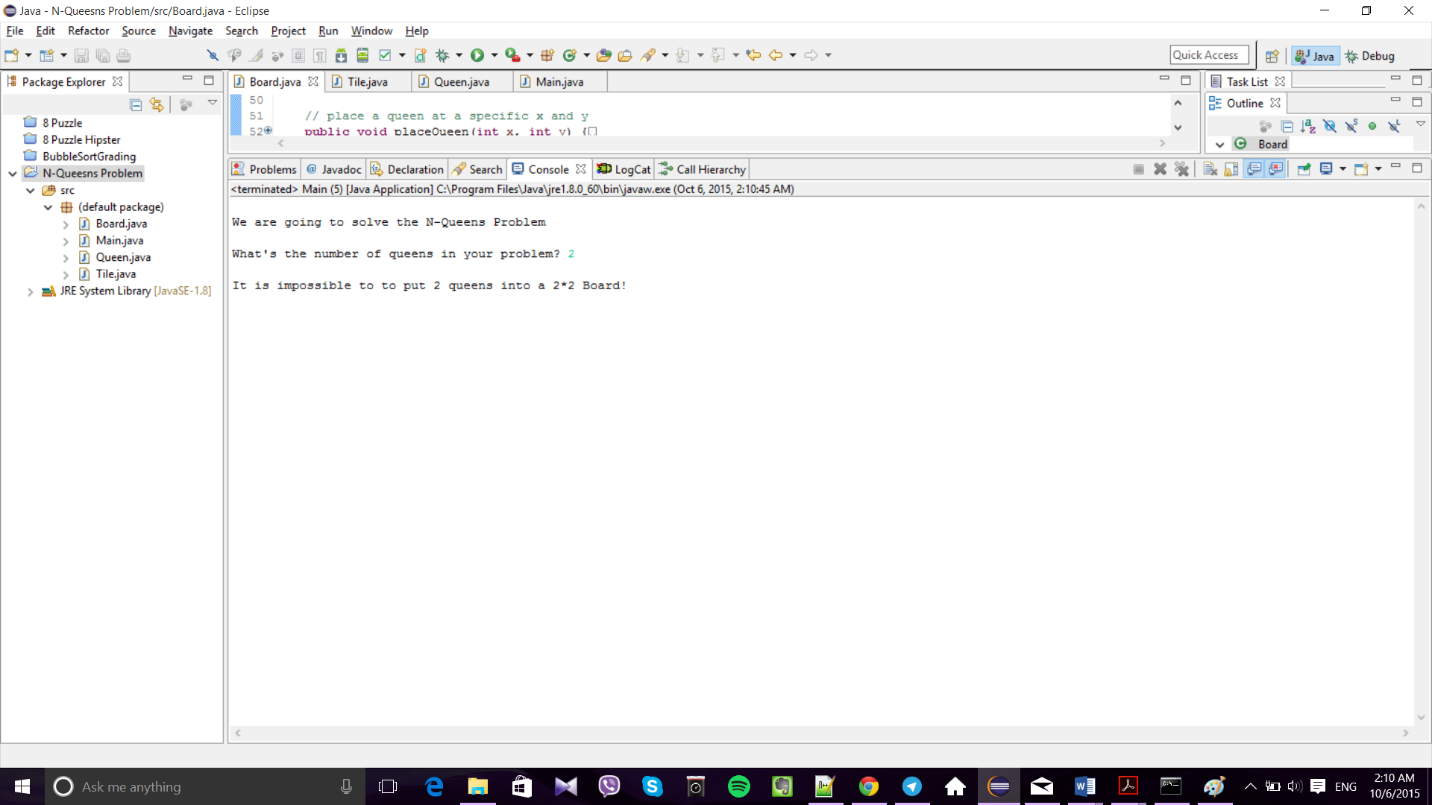
More detailed information about methods of each class could be found as in-line comments in source code.

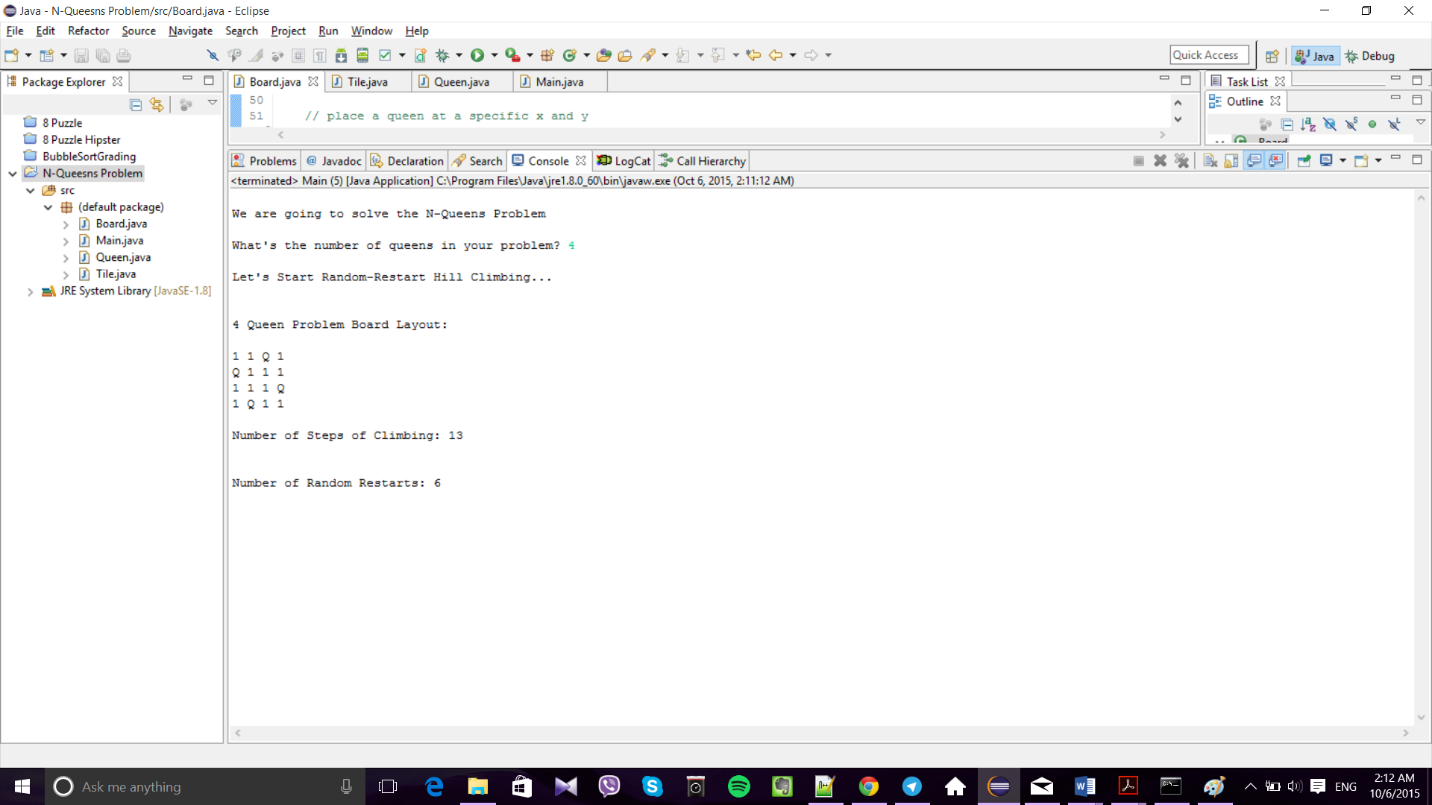
## Execution Instruction

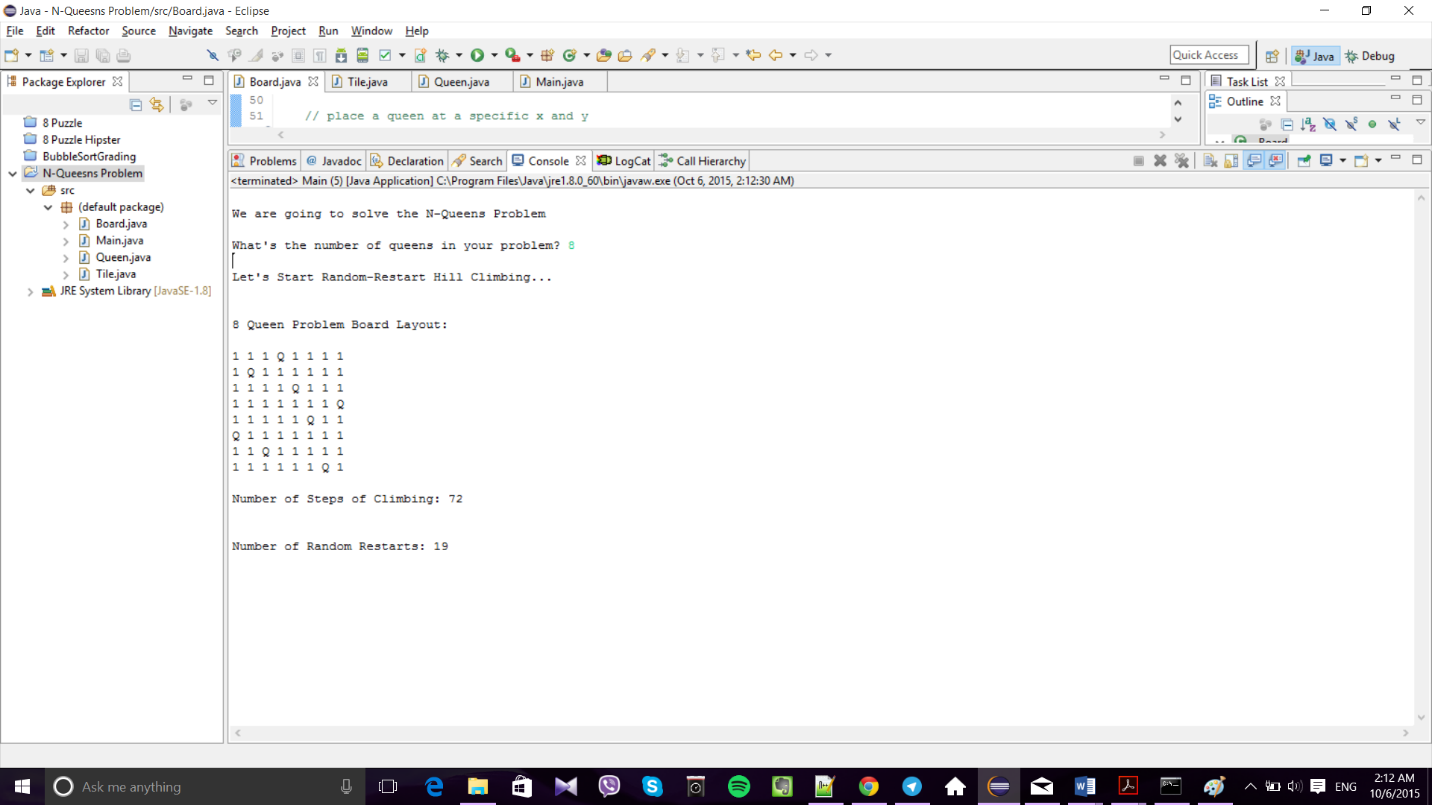
The program could be executed both from the command line and from IDE. Either way the program will prompt and ask for the number of queens. Just keep in mind, to run from command line you should first compile all the four .java files to create .class files.

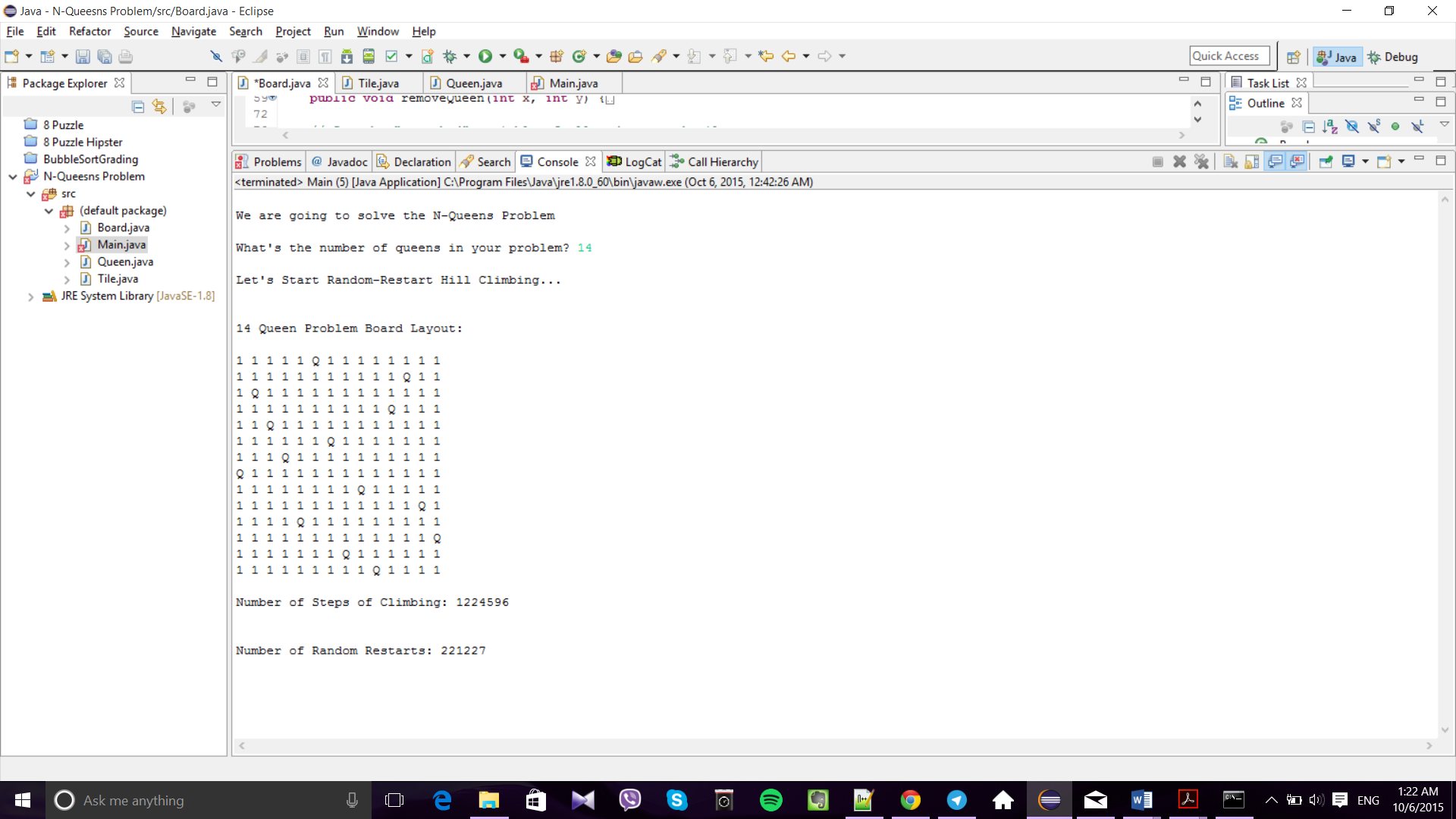
## Maximum N

My program works quickly for n<13 but for N greater than that it starts slowing down. For n=13 it takes a couple of minutes to find a solution. For n=14 the time increased to a little more than 15 minutes, and for n=15 after 40 minutes of running the program, I terminated it. Below are screen shots of some of the outputs.









## Source Code

**Main.java:**

public class Main {

public static void main(String[] args) throws IOException {

Board board;

long[] result = new long[2];

int queensNumber; //Number of queens entered by the user which is also the dimension of the board

BufferedReader reader;

reader = new BufferedReader(new InputStreamReader(System.in));

System.out.print("\nWe are going to solve the N-Queens Problem");

System.out.print("\n\nWhat's the number of queens in your problem? ");

try {

queensNumber = Integer.valueOf(reader.readLine()).intValue();

} catch (NumberFormatException e) {

System.err.println("Caught NumberFormatException: "

+ e.getMessage());

return;

}

if (queensNumber < 4) {

System.out.println("\nIt is impossible to to put " + queensNumber

+ " queens into a " + queensNumber + "\*" + queensNumber

+ " Board!");

return;

}

System.out.println("\nLet's Start Random-Restart Hill Climbing...\n");

board = new Board(queensNumber);

result = board.randomRestartHillClimbing();

System.out.println(board + "\nNumber of Steps of Climbing: " + result[0] + "\n");

System.out.println("\nNumber of Random Restarts: " + result[1] + "\n");

}

}

**Queen.java:**

public class Queen {

private int row;

private int column;

private boolean inPosition;

public Queen() {

row = 0;

column = 0;

inPosition = false;

}

public Queen(int n) {

row = 0;

column = 0;

inPosition = false;

}

public int getRow() {

return row;

}

public int getColumn() {

return column;

}

public boolean getInPosition() {

return inPosition;

}

// place a queen at a specific x and y

public void placeQueen(int x, int y) {

row = x;

column = y;

inPosition = true;

}

// function to remove a queen from a specific x, y

public void removeQueen(int x, int y) {

row = 0;

column = 0;

inPosition = false;

}

@Override

public String toString() {

String print = new String("\nQueen\n");

print = print + "Row: " + row + "\n";

print = print + "Coloumn: " + column + "\n";

if (inPosition)

print = print + "In Position: True\n";

else

print = print + "In Position: False\n";

print = print + "\n";

return print;

}

}

**Tile.java:**

// A class for each piece of the chess board

public class Tile {

private boolean occupied;

private boolean attacked; //whether or not the tile is under attack by any queens

public Tile() {

occupied = false;

attacked = false;

}

public void setAttacked() {

attacked = true;

}

public boolean getAttacked() {

return attacked;

}

public void liftAttack() {

attacked = false;

}

// Place a queen on this tile

public void placeQueen() {

occupied = true;

attacked = true;

}

public void removeQueen() {

occupied = false;

attacked = false;

}

public boolean getOccupied() {

return occupied;

}

@Override

public String toString() {

String toPrint = new String("Square\n\n");

if (attacked)

toPrint = toPrint + "Under Attack: True\n";

else

toPrint = toPrint + "Under Attack: False\n";

if (occupied)

toPrint = toPrint + "Occupied: True\n";

else

toPrint = toPrint + "Occupied: False\n";

toPrint = toPrint + "\n";

return toPrint;

}

}

**Board.java:**

import java.util.Random;

public class Board {

private int N;

private Tile[][] tiles;

private Queen[] queens;

// If number of queens(dimension) is not specified, the default value will

// be 8

public Board() {

N = 8;

tiles = new Tile[N][N];

queens = new Queen[N];

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

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

tiles[j][i] = new Tile();

}

queens[i] = new Queen(N);

}

}

public Board(int squares) {

N = squares;

tiles = new Tile[N][N];

queens = new Queen[N];

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

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

tiles[j][i] = new Tile();

}

queens[i] = new Queen(N);

}

}

public boolean isSolution() {

boolean flag = false;

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

if (queens[i].getInPosition()) {

flag = true;

} else {

flag = false;

break;

}

}

return flag;

}

// place a queen at a specific x and y

public void placeQueen(int x, int y) {

tiles[x][y].placeQueen();

queens[x].placeQueen(x, y);

setAttacks(x, y);

}

// Remove a queen at a specific x and y

public void removeQueen(int x, int y) {

tiles[x][y].removeQueen();

queens[x].removeQueen(x, y);

liftAttacks(x, y);

// Removing a queen may have undesired effects on some tiles

// We put all the queens to set the attacks once again

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

if (queens[i].getInPosition()) {

placeQueen(queens[i].getRow(), queens[i].getColumn());

}

}

}

// Set the "attacked" variable of all under-attack tiles

public void setAttacks(int x, int y) {

int tempx;

int tempy;

// Tiles on the Same Row

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

tiles[i][y].setAttacked();

}

// Tiles on the same column

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

tiles[x][j].setAttacked();

}

// Tiles diagonally under attack

tempx = x - 1;

tempy = y - 1;

while (tempx >= 0 && tempy >= 0) {

tiles[tempx][tempy].setAttacked();

tempx--;

tempy--;

}

tempx = x + 1;

tempy = y - 1;

while (tempx < N && tempy >= 0) {

tiles[tempx][tempy].setAttacked();

tempx++;

tempy--;

}

tempx = x - 1;

tempy = y + 1;

while (tempx >= 0 && tempy < N) {

tiles[tempx][tempy].setAttacked();

tempx--;

tempy++;

}

tempx = x + 1;

tempy = y + 1;

while (tempx < N && tempy < N) {

tiles[tempx][tempy].setAttacked();

tempx++;

tempy++;

}

}

// Relieve attack from tiles

public void liftAttacks(int x, int y) {

int tempx;

int tempy;

// Tiles on the Same Row

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

tiles[i][y].liftAttack();

}

// Tiles on the same column

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

tiles[x][j].liftAttack();

}

// Tiles diagonally under attack

tempx = x - 1;

tempy = y - 1;

while (tempx >= 0 && tempy >= 0) {

tiles[tempx][tempy].liftAttack();

tempx--;

tempy--;

}

tempx = x + 1;

tempy = y - 1;

while (tempx < N && tempy >= 0) {

tiles[tempx][tempy].liftAttack();

tempx++;

tempy--;

}

tempx = x - 1;

tempy = y + 1;

while (tempx >= 0 && tempy < N) {

tiles[tempx][tempy].liftAttack();

tempx--;

tempy++;

}

tempx = x + 1;

tempy = y + 1;

while (tempx < N && tempy < N) {

tiles[tempx][tempy].liftAttack();

tempx++;

tempy++;

}

}

// Heuristic function: the number of conflicting queens

public int heuristic(int[] board) {

int conflicts = 0;

int index = 0;

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

index = board[i];

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

if (i != j) {

placeQueen(j, board[j]);

if (tiles[i][index].getAttacked()) {

conflicts++;

}

removeQueen(j, board[j]);

}

}

}

return conflicts;

}

// Creates a board from a genetically created array

public void createBoard(int[] array) {

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

placeQueen(i, array[i]);

}

}

// Hill Climbing algorithm (Steepest Ascent version) to solve the problem

public long[] randomRestartHillClimbing() {

Random rand = new Random();

boolean solved = false;

long iterations = 0;

long restartNumbers = 0;

int row = 0;

int index = 0;

int[] parent = new int[(N + 1)];

int[][] children = new int[(N \* 2)][(N + 1)];

int bestChildIndex = -1;

int bestChildHeuristic = 0;

// Initialize the parent

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

parent[i] = rand.nextInt(N);

}

parent[N] = heuristic(parent);

while (!(solved)) {

// Set the heuristic of the best Child to the parent initially

bestChildHeuristic = parent[N];

bestChildIndex = -1;

// Initialize all the children to be the same as the parent

for (int j = 0; j < (N \* 2); j++) {

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

children[j][k] = parent[k];

}

children[j][N] = heuristic(children[j]);

}

for (int l = 0; l < (N \* 2); l++) {

if (parent[N] == 0) {

solved = true;

break;

}

// Find all children that are one more than their parent

if (l < N) {

children[l][row] += 1;

if (children[l][row] >= N) {

children[l][row] = 0;

}

}

// Find all children that are one less than their parent

else {

children[l][row] -= 1;

if (children[l][row] <= 0) {

children[l][row] = N - 1;

}

}

// Set the child's heuristic

children[l][N] = heuristic(children[l]);

// Pick the best child

if (children[l][N] < bestChildHeuristic) {

bestChildHeuristic = children[l][N];

bestChildIndex = l;

}

// Reset the row if it has reached the end of the board

// increment otherwise

if (row >= N) {

row = 0;

} else {

row++;

}

}

if (bestChildIndex != -1) {

// Now we have a better hill to climb to

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

parent[m] = children[bestChildIndex][m];

}

} else {

// We could'nt find a better hill to climb

// So let's start over

// Initialize the parent

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

parent[i] = rand.nextInt(N);

}

parent[N] = heuristic(parent);

// increment the number of times algorithm restarts

restartNumbers++;

}

// Is the new hill a solution?

if (parent[N] == 0) {

solved = true;

index = bestChildIndex;

}

// reset the row to start from scratch

row = 0;

// One iteration passed

iterations++;

}

// When solution is found

if (children[index][N] == 0) {

createBoard(children[index]);

} else {

createBoard(parent);

}

long[] result = { iterations, restartNumbers };

return result;

}

@Override

public String toString() {

String print = new String("\n" + N + " Queen Problem Board Layout:\n\n");

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

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

if (tiles[j][i].getOccupied())

print = print + "Q ";

else if (tiles[j][i].getAttacked())

print = print + "1 ";

else

print = print + "0 ";

}

print = print + "\n";

}

return print;

}

}