PROGRAMMING

PROJECT 2

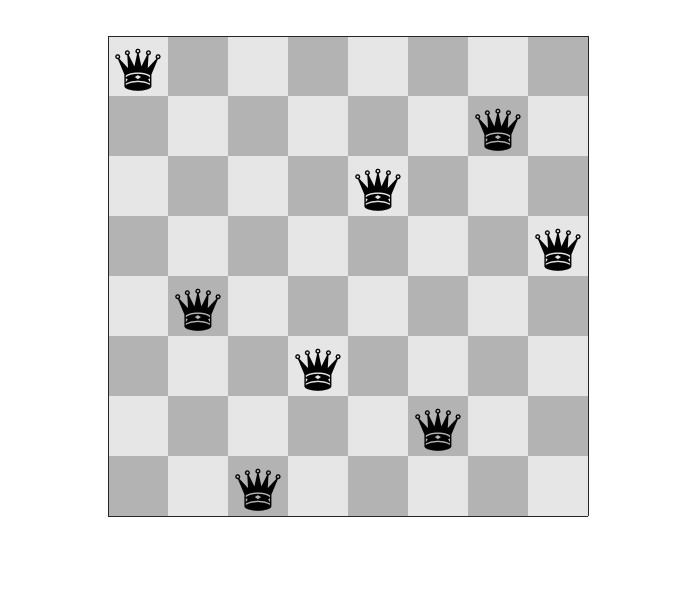
Solving N-queens problem using Hill-climbing and its variants

Avijit Jaiswal

Jinraj Jain

Ankit Pandita

**N-Queens Problem:** The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other. The chess queens can attack in horizontal, vertical and diagonal way. Below is the pictorial representation of placement of 8 queens on chessboard.



**Hill Climbing:** It is a mathematical technique which belongs to the family of local search. It start with initial state and we keep improving the solution until it’s optimal.

The workflow of hill climbing looks as follows.

1. First we define the current state as an initial state
2. Then we loop until the goal state is achieved or no more operators can be applied on the current state:

• First, we apply an operation to current state and get a new state

• Now we compare the new state with the goal

• We quit if the goal state is achieved

• After this we evaluate new state with heuristic function and compare it with the current state

• If the newer state is closer to the goal compared to current state, update the current state

It reaches the goal state with iterative improvements. In Hill-Climbing algorithm, finding goal means reaching the top of the hill.

State space diagram is a graphical representation of the set of states our search algorithm can reach vs the value of our objective function.

**X-axis:** It denotes the state space. States or configuration our algorithm may reach.

**Y-axis:** It denotes the values of function corresponding to to a particular state.

The best solution will be that state space where objective function has minimum value (global minimum).

Different regions in the State Space Diagram

**Local minimum**: This state is not better than its neighboring state however there exists a state which is better than it (global minimum). In local minimum current state is lower than its neighbors.

**Global minimum:** It is the best possible state in the state space diagram. In global minimum, function has lowest value.

**Plateau/flat local minimum:** It is a flat region of state space where neighboring states have the same value.

**Ridge:** It is region which is lower than its neighbors but itself has a slope. It is a special kind of local minimum.

**Current state:** The region of state space diagram where we are currently present during the search.

**Shoulder:** It is a plateau that has an uphill edge.

**With sideways moves:** When stuck on a ridge or plateau (i.e., all successors have the same value), allow it to move anyway hoping it is a shoulder and after some time, there will be a way up.

**Steepest-Ascent Hill-Climbing:** It differs from the basic Hill climbing algorithm by choosing the best successor rather than the first successor that is better. It indicates that it has elements of the breadth first algorithm.

**Random-restart hill-climbing:** If the first hill-climbing attempt doesn’t work, try again and again. That is, generate random initial states and perform hill-climbing again and again. This is random-restart. The number of attempts needs to be limited, this number depends on the problem.

**PROGRAM STRUCTURE**

The problem is formulated as an NxN board out of which N squares are occupied by queens. The goal of the program is to position the N queens such that no two queens are attacking each other.

In the program, the positions occupied by queens are represented by 1’s and empty positions are represented by 0’s.

Each arrangement of N queens on the NxN board is considered as a state. The number of queens attacking each other in any any board state is used as a heuristic function.

**Programming Language: Java**

The source code developed consists of 5 Java source files:

MainNQueensProblem.java : This is the main driving source file consisting of the main driving function.

Board.java : This file consists of all the functions for demonstrating the various board configurations.

SteepestAscent.java : This file consists of the implementation for the basic hill climbing algorithm.

RandomRestart.java : This file consists of the implementation for hill climbing with random restart method.

SidewaysMove.java : This file consists of the implementation for Hill Climbing with sideways moves method.

**Problem Implementation Details:**

The objective of this program is to implement N Queens problem is by using hill climbing search and its variants. The program takes the number of queens as a variable n and allows the user to input the value of n. We are implementing the below mentioned points in this program:

* **Hill Climbing Search**

**Code:**

**1) MainNQueensProblem.java**

package edu.uncc.cci.algods;

import java.text.DecimalFormat;

import java.util.Scanner;

/\*\*

\* Authors: Ankit Pandita, Avijit Jaiswal, Jinraj Jain

\*/

public class MainNQueensProblem {

public static void main(String[] args) {

int numberOfQueens = 0;

Scanner input = new Scanner(System.in);

DecimalFormat format = new DecimalFormat("##.##");

int[] hillClimbingSteepestAscent = {0, 0, 0, 0};

int[] hillClimbingSideways = {0, 0, 0, 0};

int[] RandomRestartWithoutSideways = {0, 0, 0, 0};

int[] RandomRestartWithSideways = {0, 0, 0, 0};

while (numberOfQueens <= 3) {

System.out.println("Please enter the number of queens (must be > 3): ");

numberOfQueens = input.nextInt();

}

System.out.println("Enter the number of runs: ");

int numOfRuns = input.nextInt();

int choice = -1;

while (choice <= 0 || choice >= 5) {

System.out.println("Please select the Hill Climbing Search Method: ");

System.out.println("1. Hill Climbing Search using Steepest Ascent");

System.out.println("2. Hill Climbing Search using Sideways Move");

System.out.println("3. Random Restart Hill climbing Search without Sideways Move");

System.out.println("4. Random Restart Hill climbing Search with Sideways Move");

System.out.println("Enter choice: ");

choice = input.nextInt();

}

input.close();

switch (choice) {

case 1: // Steepest Ascent Hill Climbing Search

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

SteepestAscent game = new SteepestAscent(numberOfQueens); //class object

int[] results = game.runSteepestAscentHillClimbingSearch();

hillClimbingSteepestAscent[0] += results[0];

hillClimbingSteepestAscent[1] += results[1];

hillClimbingSteepestAscent[2] += results[2];

hillClimbingSteepestAscent[3] += results[3];

}

System.out.println("\n -------------------------------------------------------------------------------- ");

System.out.println("Hill Climbing Search using Steepest Ascent");

System.out.println("Number of Queens: " + numberOfQueens);

System.out.println("Number of Iterations: " + numOfRuns);

System.out.println("Success Rate: " + format.format((hillClimbingSteepestAscent[1] \* 100) / (float) numOfRuns) + "%");

System.out.println("Failure Rate: " + format.format((hillClimbingSteepestAscent[3] \* 100) / (float) numOfRuns) + "%");

if (hillClimbingSteepestAscent[1] != 0)

System.out.println("Average Number of Steps when Algorithm Succeeds: " + (hillClimbingSteepestAscent[0] / hillClimbingSteepestAscent[1]));

if (hillClimbingSteepestAscent[3] != 0)

System.out.println("Average Number of Steps when Algorithm Fails: " + (hillClimbingSteepestAscent[2] / hillClimbingSteepestAscent[3]));

break;

case 2: // Sideways Move Hill Climbing Search

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

SidewaysMove game = new SidewaysMove(numberOfQueens); //class object

int[] results = game.runWithSidewaysMoveHCSearchAlgorithm();

hillClimbingSideways[0] += results[0];

hillClimbingSideways[1] += results[1];

hillClimbingSideways[2] += results[2];

hillClimbingSideways[3] += results[3];

}

System.out.println("\n --------------------------------------------------------------------------------");

System.out.println("Hill Climbing Search using Sideways Move");

System.out.println("Number of Queens: " + numberOfQueens);

System.out.println("Number of Iterations: " + numOfRuns);

System.out.println("Success Rate: " + format.format((hillClimbingSideways[1] \* 100) / (float) numOfRuns) + "%");

System.out.println("Failure Rate: " + format.format((hillClimbingSideways[3] \* 100) / (float) numOfRuns) + "%");

if (hillClimbingSideways[1] != 0)

System.out.println("Average Number of Steps when Algorithm Succeeds: " + (hillClimbingSideways[0] / hillClimbingSideways[1]));

if (hillClimbingSideways[3] != 0)

System.out.println("Average Number of Steps when Algorithm Fails: " + (hillClimbingSideways[2] / hillClimbingSideways[3]));

break;

case 3: // Random Restart algorithm

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

RandomRestart game = new RandomRestart(numberOfQueens); //class object

int[] results = game.runRandomRestartHCSearchWithoutSidewayMoves();

RandomRestartWithoutSideways[0] += results[0];

RandomRestartWithoutSideways[1] += results[1];

RandomRestartWithoutSideways[2] += results[2];

}

System.out.println("\n -------------------------------------------------------------------------------- ");

System.out.println("Random Restart Hill Climbing Search without Sideways Move");

System.out.println("Number of Queens: " + numberOfQueens);

System.out.println("Number of Iterations: " + numOfRuns);

System.out.println("Success Rate: " + format.format((RandomRestartWithoutSideways[1] \* 100) / (float) numOfRuns) + "%");

System.out.println("Average Number of Steps when Algorithm Succeeds: " + (RandomRestartWithoutSideways[0] / numOfRuns));

System.out.println("Average Number of Restarts when Algorithm Succeeds: " + (RandomRestartWithoutSideways[2] / numOfRuns));

break;

case 4: // Random Restart with Sideways Move.

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

RandomRestart game = new RandomRestart(numberOfQueens); //class object

int[] results = game.runRandomRestartHCSearchWithSidewayMoves();

RandomRestartWithSideways[0] += results[0];

RandomRestartWithSideways[1] += results[1];

RandomRestartWithSideways[2] += results[2];

}

System.out.println("\n -------------------------------------------------------------------------------- ");

System.out.println("Random Restart Hill Climbing Search with Sideways Move");

System.out.println("Number of Queens: " + numberOfQueens);

System.out.println("Number of Iterations: " + numOfRuns);

System.out.println("Success Rate: " + format.format((RandomRestartWithSideways[1] \* 100) / (float) numOfRuns) + "%");

System.out.println("Average Number of Steps when Algorithm Succeeds: " + (RandomRestartWithSideways[0] / numOfRuns));

if ((RandomRestartWithSideways[2] / (float) numOfRuns) > 0 && (RandomRestartWithSideways[2] / (float) numOfRuns) < 1)

System.out.println("Average Number of Restarts when Algorithm Succeeds: ~ 1");

else

System.out.println("Average Number of Restarts when Algorithm Succeeds: " + (RandomRestartWithSideways[2] / (float) numOfRuns));

break;

}

}

}

**2) Board.java**

package edu.uncc.cci.algods;

import java.util.Random;

/\*\*

\* Authors: Ankit Pandita, Avijit Jaiswal, Jinraj Jain

\*/

public abstract class Board {

private Random randomNumber = new Random();

public final int QUEEN = 1;

public final int NOT\_QUEEN = 0;

public int numberOfQueens;

public int[][] board;

private int[][] board1;

private int[][] board2;

private int[][] board3;

public int regMoves = 0;

public int sideMoves = 0;

public int resetCount = -1;

//Initializing done in the constructor

public Board(int numberOfQueens) {

this.numberOfQueens = numberOfQueens;

this.board1 = new int[numberOfQueens][numberOfQueens];

this.board2 = new int[numberOfQueens][numberOfQueens];

this.board3 = new int[numberOfQueens][numberOfQueens];

this.board = new int[numberOfQueens][numberOfQueens];

this.resetBoard();

this.setBoard();

}

public int getNumberOfQueens() {

return this.numberOfQueens;

}

public int[][] getBoard1() {

return this.board1;

}

public int[][] getBoard2() {

return this.board2;

}

public int[][] getBoard3() {

return this.board3;

}

//Select any one of the three boards

public void setBoard() {

int n = randomNumber.nextInt(2);

if (n == 0)

this.board = this.copyState(board1);

else if (n == 1)

this.board = this.copyState(board2);

else

this.board = this.copyState(board3);

}

public void resetBoard() {

int queenCount = this.numberOfQueens;

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

int queenPosY = randomNumber.nextInt(queenCount - 1);

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

if (j == queenPosY)

this.board1[j][i] = QUEEN;

else

this.board1[j][i] = NOT\_QUEEN;

}

}

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

int queenPosY = randomNumber.nextInt(queenCount - 1);

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

if (j == queenPosY)

this.board2[j][i] = QUEEN;

else

this.board2[j][i] = NOT\_QUEEN;

}

}

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

int queenPosY = randomNumber.nextInt(queenCount - 1);

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

if (j == queenPosY)

this.board3[j][i] = QUEEN;

else

this.board3[j][i] = NOT\_QUEEN;

}

}

this.resetCount += 1; // Increment the number of resets

}

public int[][] getGameBoard() {

return this.board;

}

public int calcHeuristic(int[][] state) {

int[][] board = state;

int heuristicValue = 0;

for (int i = 0; i < this.numberOfQueens; i++) {

int[] row = board[i];

int queenCount = 0;

for (int r : row) {

if (r == QUEEN)

queenCount += 1;

}

if (queenCount > 1) {

for (int q = queenCount; q > 1; --q) {

heuristicValue += q - 1;

}

}

}

for (int i = 0; i < this.numberOfQueens; i++) {

int queenCount = 0;

for (int j = 0; j < this.numberOfQueens; j++) {

if (board[j][i] == QUEEN)

queenCount += 1;

}

if (queenCount > 1) {

for (int q = queenCount; q > 1; --q) {

heuristicValue += q - 1;

}

}

}

int numOfQueens = 0;

int iteration = 0;

while (iteration < (this.numberOfQueens - 1)) {

numOfQueens = 0;

int xValue = 0;

int yValue = iteration;

while (xValue <= iteration) {

if (board[xValue][yValue] == QUEEN)

numOfQueens += 1;

yValue -= 1;

xValue += 1;

}

if (numOfQueens > 1) {

for (int q = numOfQueens; q > 1; --q) {

heuristicValue += q - 1;

}

}

iteration += 1;

}

numOfQueens = 0;

int diagonal = this.numberOfQueens - 1;

for (int i = 0; i < this.numberOfQueens; i++) {

if (board[i][diagonal] == QUEEN)

numOfQueens += 1;

diagonal -= 1;

}

if (numOfQueens > 1) {

for (int q = numOfQueens; q > 1; --q) {

heuristicValue += q - 1;

}

}

numOfQueens = 0;

iteration = 1;

while (iteration < (this.numberOfQueens - 1)) {

numOfQueens = 0;

int xValue = iteration;

int yValue = this.numberOfQueens - 1;

while (xValue < this.numberOfQueens) {

if (board[xValue][yValue] == QUEEN)

numOfQueens += 1;

yValue -= 1;

xValue += 1;

}

if (numOfQueens > 1) {

for (int q = numOfQueens; q > 1; --q) {

heuristicValue += q - 1;

}

}

iteration += 1;

}

numOfQueens = 0;

iteration = this.numberOfQueens - 2;

while (iteration > 0) {

numOfQueens = 0;

int xValue = 0;

int yValue = iteration;

while (yValue < this.numberOfQueens) {

if (board[xValue][yValue] == QUEEN)

numOfQueens += 1;

yValue += 1;

xValue += 1;

}

if (numOfQueens > 1) {

for (int q = numOfQueens; q > 1; --q) {

heuristicValue += q - 1;

}

}

iteration -= 1;

}

numOfQueens = 0;

diagonal = 0;

for (int i = 0; i < this.numberOfQueens; i++) {

if (board[i][diagonal] == QUEEN)

numOfQueens += 1;

diagonal += 1;

}

if (numOfQueens > 1) {

for (int q = numOfQueens; q > 1; --q) {

heuristicValue += q - 1;

}

}

numOfQueens = 0;

iteration = 1;

while (iteration < this.numberOfQueens) {

numOfQueens = 0;

int xValue = iteration;

int yValue = 0;

while (xValue < this.numberOfQueens) {

if (board[xValue][yValue] == QUEEN)

numOfQueens += 1;

yValue += 1;

xValue += 1;

}

if (numOfQueens > 1) {

for (int q = numOfQueens; q > 1; --q) {

heuristicValue += q - 1;

}

}

iteration += 1;

}

return heuristicValue;

}

//Reset regular moves after each iteration

public void resetRegMoves() {

this.regMoves = 0;

}

//Return number of resets done

public int getResets() {

return this.resetCount;

}

//Reset sideway Moves after each iteration

public void resetSideMoves() {

this.sideMoves = 0;

}

//Function to copy a state into a temporary state and return new state.

public int[][] copyState(int[][] oldState) {

int[][] newState = new int[this.getNumberOfQueens()][this.getNumberOfQueens()];

//iteratively copy each element from old to the new state

for (int i = 0; i < oldState.length; i++) {

System.arraycopy(oldState[i], 0, newState[i], 0, oldState[i].length);

}

return newState;

}

//Function to display initial game state heuristic value and board for each iteration.

public void startGameState() {

System.out.println("\n\n ------------------------------------------------------ ");

System.out.println("Initial State");

System.out.println("Current State Heuristic Value: " + this.calcHeuristic(this.board)); //heuristic value

System.out.println("Current State: \n");

for (int[] intArray : this.board) {

for (int j = 0; j < this.board.length; j++) {

System.out.print(intArray[j] + " ");

}

System.out.println("");

}

}

//Function to print current state heuristic value and state board in each step

public void printGameState() {

System.out.println("\nCurrent State Heuristic Value: " + this.calcHeuristic(this.board)); //heuristic value

System.out.println("Current State: \n"); //print each element of the current state board

for (int[] intArray : this.board) {

for (int j = 0; j < this.board.length; j++) {

System.out.print(intArray[j] + " ");

}

System.out.println("");

}

}

//Function to check if two boards passed are equal

public boolean areBoardsEqual(int[][] boardOne, int[][] boardTwo) {

boolean are2BoardsEqual = true;

for (int i = 0; i < this.getNumberOfQueens(); i++) {

for (int j = 0; j < this.getNumberOfQueens(); j++) {

if (boardOne[i][j] != boardTwo[i][j]) {

are2BoardsEqual = false;

break;

}

}

}

return !are2BoardsEqual;

}

}

**3) SteepestAscent.java**

package edu.uncc.cci.algods;

import java.util.ArrayList;

import java.util.List;

import java.util.Random;

/\*\*

\* Authors: Ankit Pandita, Avijit Jaiswal, Jinraj Jain

\*/

public class SteepestAscent extends Board {

private int numberOfStepsWhenSuccess = 0;

private int numberOfIterationsWhenSuccess = 0;

private int numberOfStepsWhenFailure = 0;

private int numberOfIterationsWhenFailure = 0;

private boolean isBoardChanged = true;

private static int printCountSteepestAscent = 0;

public SteepestAscent(int numOfQueens) {

super(numOfQueens);

}

public int[] runSteepestAscentHillClimbingSearch() {

printCountSteepestAscent++;

printStartState();

this.isBoardChanged = true;

int currentStateHeuristic = this.calcHeuristic(this.board);

List<int[][]> possibleStates = new ArrayList<>();

this.resetRegMoves();

while (this.calcHeuristic(this.board) != 0 && (this.isBoardChanged)) {

int columnNumber = 0;

int[][] possibleState = new int[this.getNumberOfQueens()][this.getNumberOfQueens()];

for (int i = 0; i < this.getNumberOfQueens(); i++) {

if (this.getNumberOfQueens() >= 0)

System.arraycopy(this.board[i], 0, possibleState[i], 0, this.getNumberOfQueens());

}

currentStateHeuristic = this.calcHeuristic(this.board);

this.isBoardChanged = false;

while (columnNumber < this.getNumberOfQueens()) {

int queenPositionInCurrentColumn = -1;

for (int i = 0; i < this.getNumberOfQueens(); i++) {

if (possibleState[i][columnNumber] == this.QUEEN)

queenPositionInCurrentColumn = i;

possibleState[i][columnNumber] = this.NOT\_QUEEN;

}

for (int i = 0; i < this.getNumberOfQueens(); i++) {

possibleState[i][columnNumber] = this.QUEEN;

int[][] newState = new int[this.getNumberOfQueens()][this.getNumberOfQueens()];

for (int k = 0; k < this.getNumberOfQueens(); k++) {

if (this.getNumberOfQueens() >= 0)

System.arraycopy(possibleState[k], 0, newState[k], 0, this.getNumberOfQueens());

}

if (this.calcHeuristic(this.board) > this.calcHeuristic(newState))

possibleStates.add(newState); // Store the state in the list of successor states.

possibleState[i][columnNumber] = this.NOT\_QUEEN;

}

possibleState[queenPositionInCurrentColumn][columnNumber] = this.QUEEN;

columnNumber += 1;

}

Random rand = new Random();

if (possibleStates.size() != 0) {

int pick = rand.nextInt(possibleStates.size());

int minHeuristic = currentStateHeuristic;

if (minHeuristic > this.calcHeuristic(possibleStates.get(pick))) {

minHeuristic = this.calcHeuristic(possibleStates.get(pick));

this.board = this.copyState(possibleStates.get(pick));

this.isBoardChanged = true;

this.regMoves += 1;

printCurrentState();

possibleStates.clear();

}

} else {

if (possibleStates.size() == 0) {

this.isBoardChanged = false;

possibleStates.clear();

}

}

}

if (this.calcHeuristic(this.board) == 0) {

this.numberOfStepsWhenSuccess += this.regMoves;

this.numberOfIterationsWhenSuccess += 1;

} else {

printCurrentState();

this.numberOfStepsWhenFailure += this.regMoves;

this.numberOfIterationsWhenFailure += 1;

}

return new int[]{numberOfStepsWhenSuccess, numberOfIterationsWhenSuccess, numberOfStepsWhenFailure, numberOfIterationsWhenFailure};

}

private void printCurrentState() {

if (printCountSteepestAscent <= 4) {

this.printGameState();

}

}

private void printStartState() {

if (printCountSteepestAscent <= 4) {

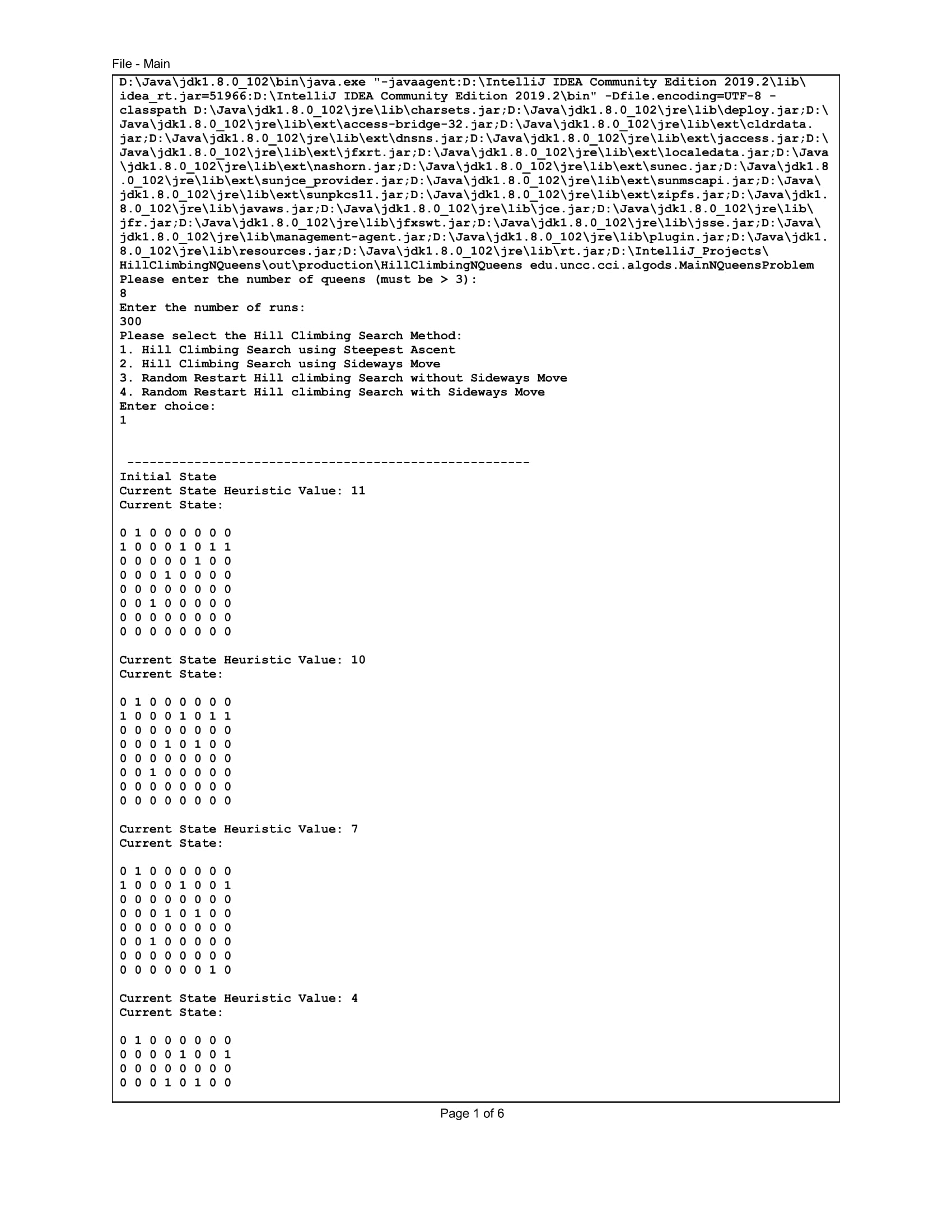
this.startGameState();

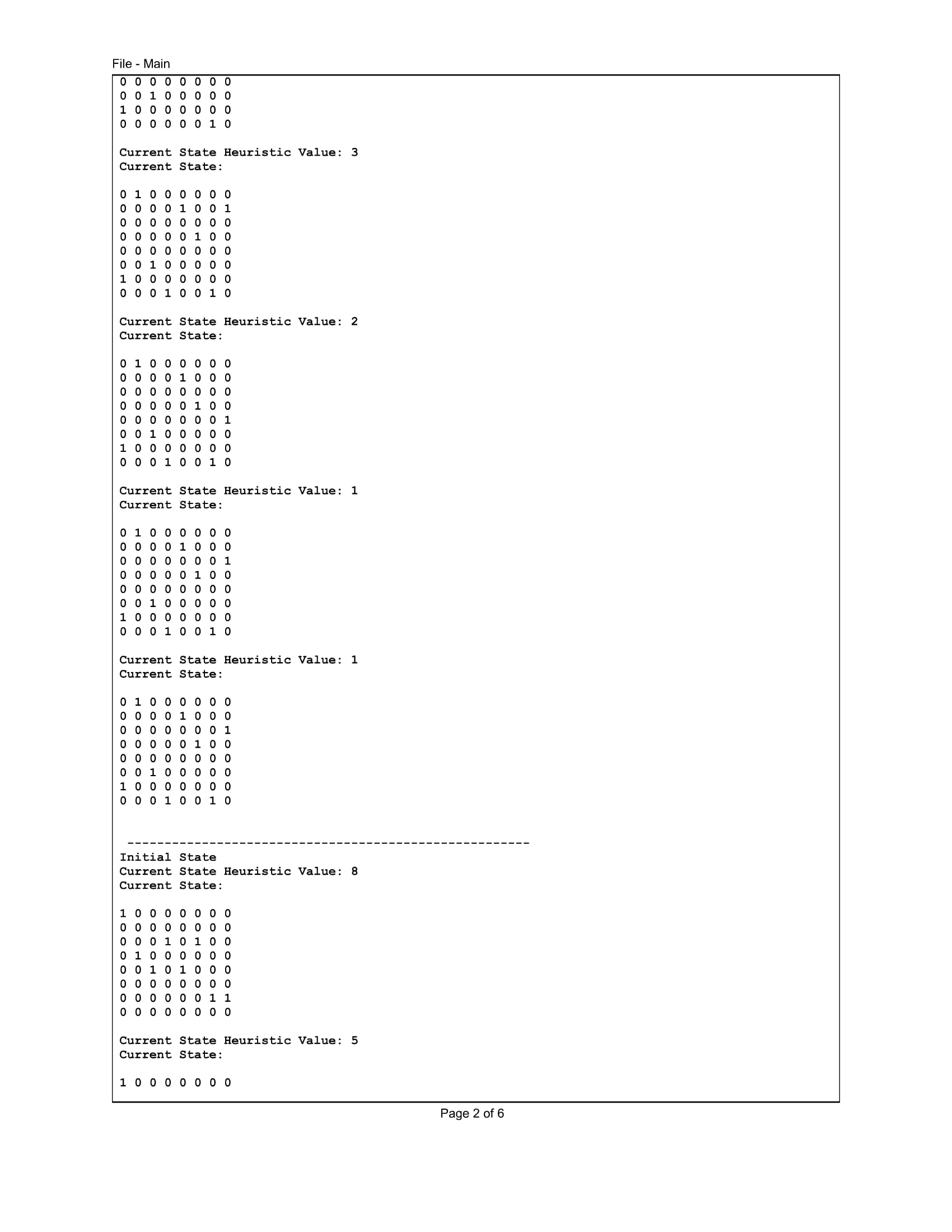
}

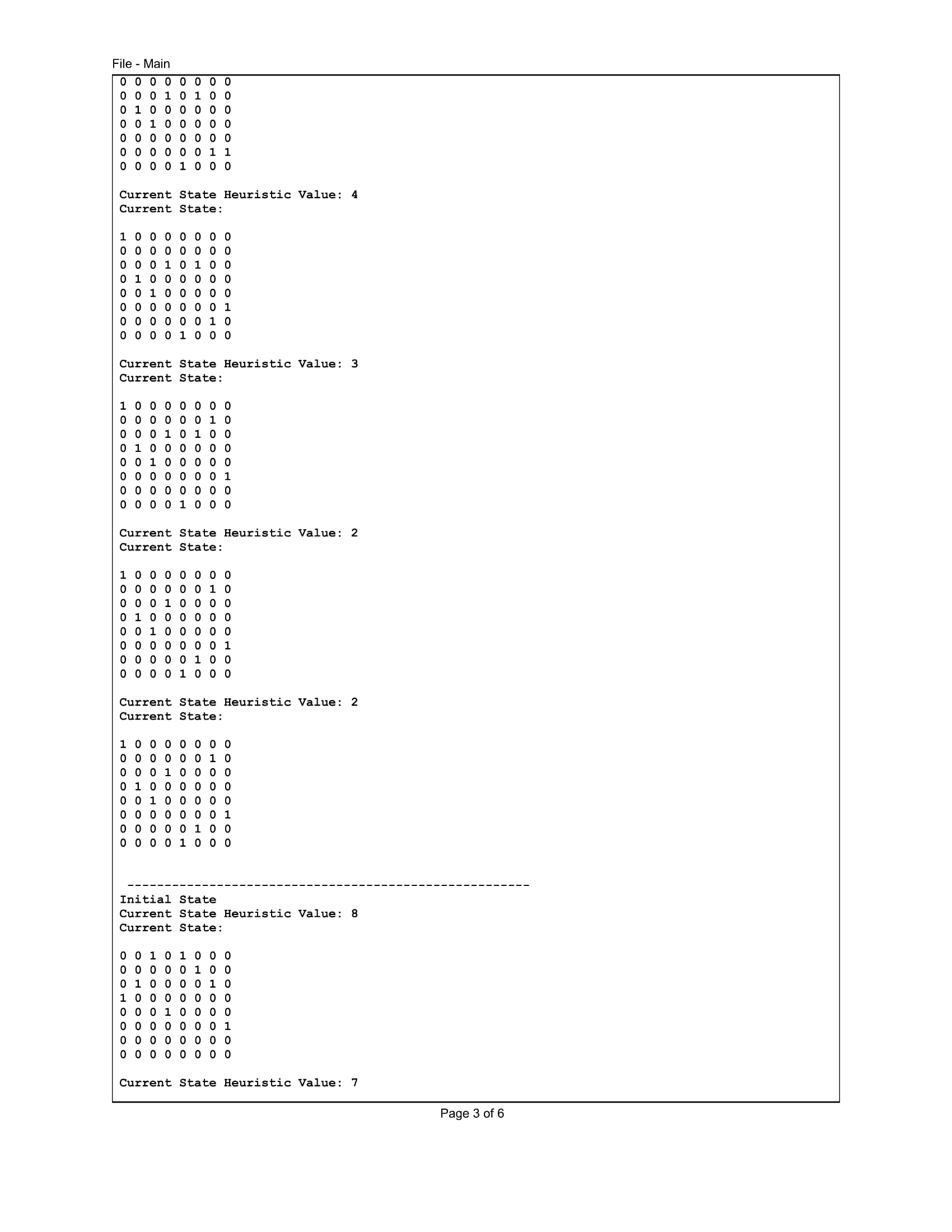
}

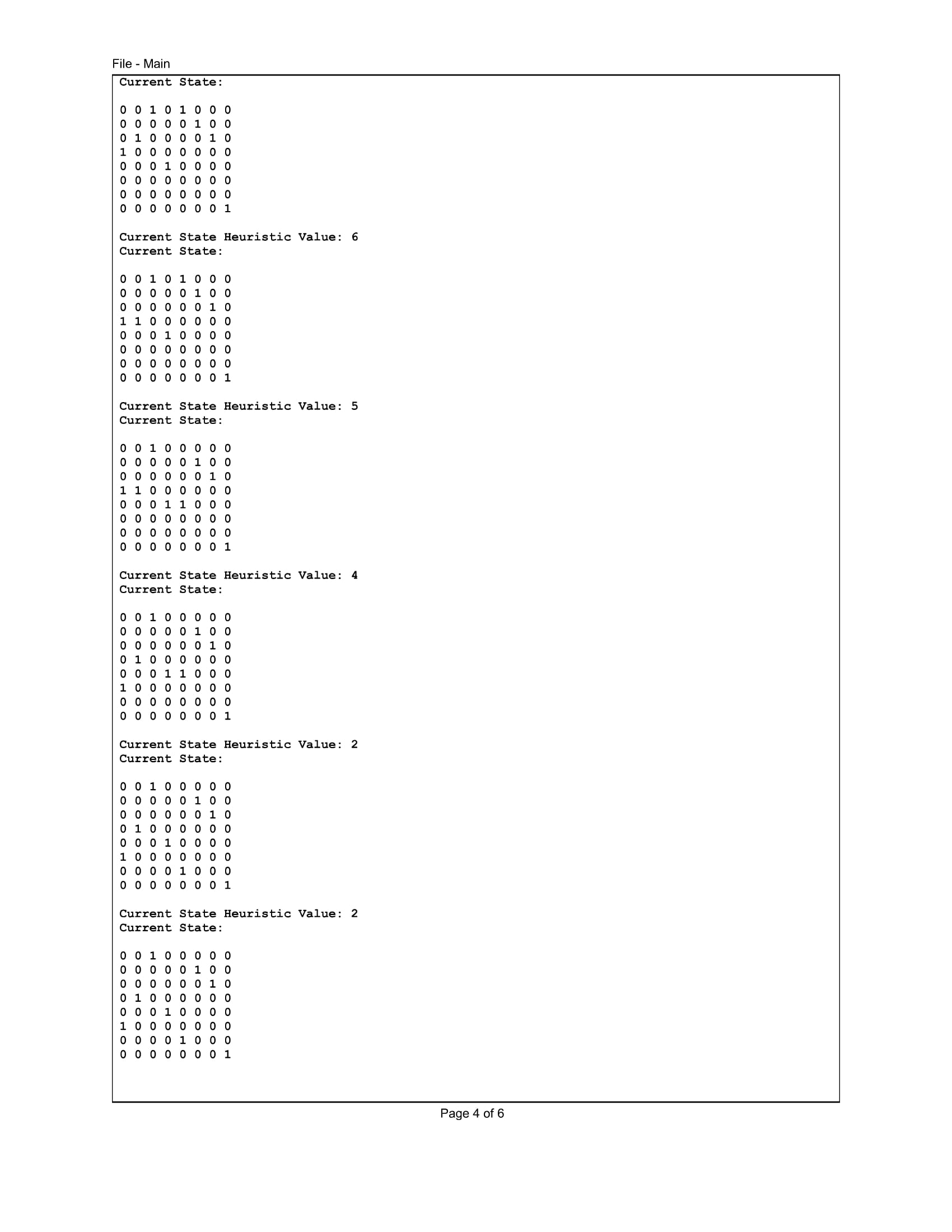
}

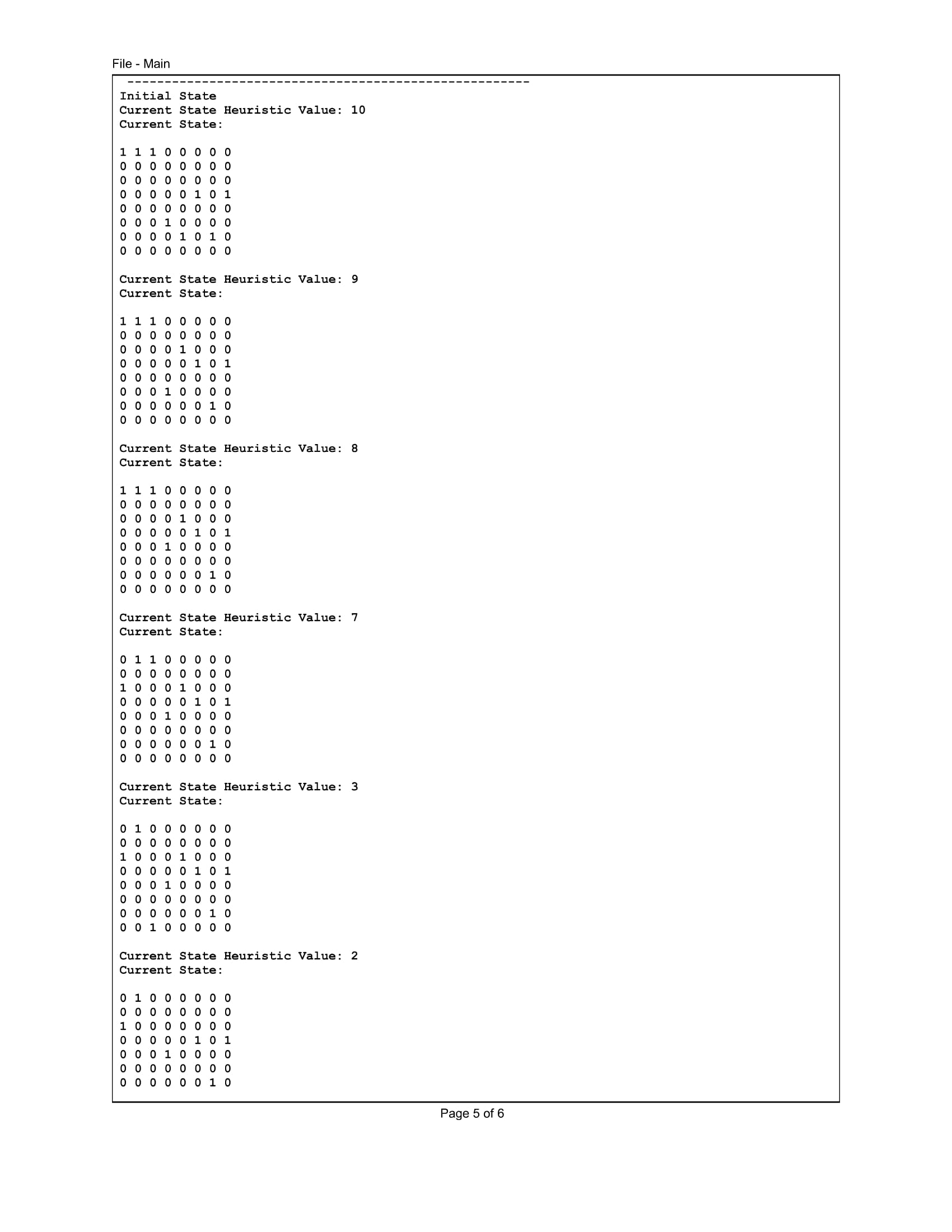
**Output**

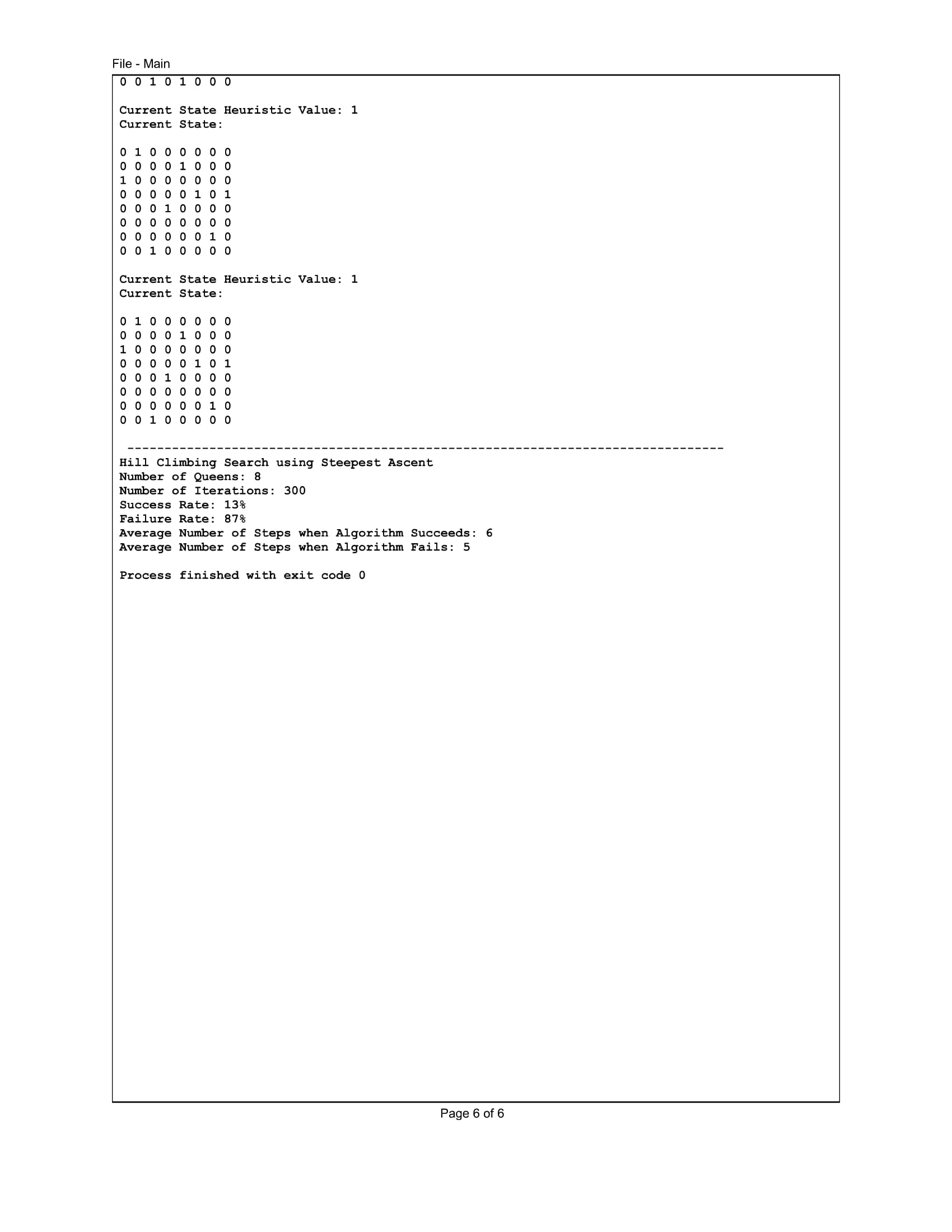












* **Hill Climbing Search with sideways move**

**Code**

package edu.uncc.cci.algods;

import java.util.ArrayList;

import java.util.List;

import java.util.Random;

/\*\*

\* Authors: Ankit Pandita, Avijit Jaiswal, Jinraj Jain

\*/

public class SidewaysMove extends Board {

private int numberOfStepsWhenSuccess = 0;

private int numberOfIterationsWhenSuccess = 0;

private int numberOfStepsWhenFailure = 0;

private int numberOfIterationsWhenFailure = 0;

private boolean boardModified = true;

private int consecutiveMoves = 0;

private final int LIMIT\_SIDEWAYS\_CONSECUTIVE\_MOVES = 100;

private static int printCountSidewaysMove = 0;

public SidewaysMove(int numOfQueens) {

super(numOfQueens);

}

public int[] runWithSidewaysMoveHCSearchAlgorithm() {

printCountSidewaysMove++;

printStartState();

this.boardModified = true;

int currentBoardStateHeuristic = this.calcHeuristic(this.board);

List<int[][]> possibleStates = new ArrayList<>();

this.resetRegMoves();

while (this.calcHeuristic(this.board) != 0 && (this.boardModified)) {

boolean moveMade = false;

int columnNo = 0;

int[][] possibleState = new int[this.getNumberOfQueens()][this.getNumberOfQueens()];

for (int i = 0; i < this.getNumberOfQueens(); i++) {

if (this.getNumberOfQueens() >= 0)

System.arraycopy(this.board[i], 0, possibleState[i], 0, this.getNumberOfQueens());

}

currentBoardStateHeuristic = this.calcHeuristic(this.board);

this.boardModified = false;

while (columnNo < this.getNumberOfQueens()) {

int currentColumnQueenPosition = -1;

for (int i = 0; i < this.getNumberOfQueens(); i++) {

if (possibleState[i][columnNo] == this.QUEEN)

currentColumnQueenPosition = i;

possibleState[i][columnNo] = this.NOT\_QUEEN;

}

for (int i = 0; i < this.getNumberOfQueens(); i++) { //For each row, place a queen in the column and store the state in the list if h <= h(current state)

possibleState[i][columnNo] = this.QUEEN;

int[][] newMove = new int[this.getNumberOfQueens()][this.getNumberOfQueens()];

for (int k = 0; k < this.getNumberOfQueens(); k++) {

if (this.getNumberOfQueens() >= 0)

System.arraycopy(possibleState[k], 0, newMove[k], 0, this.getNumberOfQueens());

}

if (this.calcHeuristic(this.board) >= this.calcHeuristic(newMove) && this.areBoardsEqual(this.board, newMove))

possibleStates.add(newMove);

possibleState[i][columnNo] = this.NOT\_QUEEN;

}

possibleState[currentColumnQueenPosition][columnNo] = this.QUEEN;

columnNo += 1;

}

Random randomNumber = new Random();

int minimumHeuristic = currentBoardStateHeuristic;

if (possibleStates.size() != 0) {

int pick = randomNumber.nextInt(possibleStates.size());

if (minimumHeuristic > this.calcHeuristic(possibleStates.get(pick))) {

minimumHeuristic = this.calcHeuristic(possibleStates.get(pick));

this.board = this.copyState(possibleStates.get(pick));

this.boardModified = true;

this.regMoves += 1;

this.consecutiveMoves = 0;

printCurrentState();

possibleStates.clear();

} else if (minimumHeuristic == this.calcHeuristic(possibleStates.get(pick)) &&

this.consecutiveMoves < this.LIMIT\_SIDEWAYS\_CONSECUTIVE\_MOVES) {

minimumHeuristic = this.calcHeuristic(possibleStates.get(pick));

this.board = this.copyState(possibleStates.get(pick));

this.boardModified = true;

this.consecutiveMoves += 1;

this.regMoves += 1;

printCurrentState();

possibleStates.clear();

}

} else {

this.boardModified = false;

}

}

if (this.calcHeuristic(this.board) == 0) {

this.numberOfStepsWhenSuccess += this.regMoves;

this.numberOfIterationsWhenSuccess += 1;

} else {

printCurrentState();

this.numberOfStepsWhenFailure += this.regMoves;

this.numberOfIterationsWhenFailure += 1;

}

return new int[]{numberOfStepsWhenSuccess, numberOfIterationsWhenSuccess, numberOfStepsWhenFailure, numberOfIterationsWhenFailure};

}

private void printCurrentState() {

if (printCountSidewaysMove <= 4) {

this.printGameState();

}

}

private void printStartState() {

if (printCountSidewaysMove <= 4) {

this.startGameState();

}

}

}

**Output**

Please refer to the output file Result\_Sideways\_Move.pdf

* **Random restart Hill Climbing Search**

**Code**

package edu.uncc.cci.algods;

import java.util.ArrayList;

import java.util.List;

import java.util.Random;

/\*\*

\* Authors: Ankit Pandita, Avijit Jaiswal, Jinraj Jain

\*/

public class RandomRestart extends Board {

private int numStepsSuccess = 0;

private int numIterationsSuccess = 0;

private boolean boardModified = true;

private int consecutiveSidewaysMoves = 0;

private final int LIMIT\_CONSECUTIVE\_SIDEWAYS\_MOVES = 100;

public RandomRestart(int numOfQueens) {

super(numOfQueens);

}

public int[] runRandomRestartHCSearchWithoutSidewayMoves() {

int currentStateHeuristic = this.calcHeuristic(this.board);

List<int[][]> possibleStates = new ArrayList<>();

this.resetRegMoves();

while (this.calcHeuristic(this.board) != 0 && boardModified) {

int columnNo = 0;

int[][] possibleState = new int[this.getNumberOfQueens()][this.getNumberOfQueens()];

for (int i = 0; i < this.getNumberOfQueens(); i++) {

if (this.getNumberOfQueens() >= 0)

System.arraycopy(this.board[i], 0, possibleState[i], 0, this.getNumberOfQueens());

}

currentStateHeuristic = this.calcHeuristic(this.board);

this.boardModified = false;

while (columnNo < this.getNumberOfQueens()) {

int queenPositionInCurrentColumn = -1;

for (int i = 0; i < this.getNumberOfQueens(); i++) {

if (possibleState[i][columnNo] == this.QUEEN)

queenPositionInCurrentColumn = i;

possibleState[i][columnNo] = this.NOT\_QUEEN;

}

for (int i = 0; i < this.getNumberOfQueens(); i++) {

possibleState[i][columnNo] = this.QUEEN;

int[][] newState = new int[this.getNumberOfQueens()][this.getNumberOfQueens()];

for (int k = 0; k < this.getNumberOfQueens(); k++) {

if (this.getNumberOfQueens() >= 0)

System.arraycopy(possibleState[k], 0, newState[k], 0, this.getNumberOfQueens());

}

if (this.calcHeuristic(this.board) > this.calcHeuristic(newState))

possibleStates.add(newState);

possibleState[i][columnNo] = this.NOT\_QUEEN;

}

possibleState[queenPositionInCurrentColumn][columnNo] = this.QUEEN;

columnNo += 1;

}

Random randomNumber = new Random();

if (possibleStates.size() != 0) {

int pick = randomNumber.nextInt(possibleStates.size());

int minimumHeuristic = currentStateHeuristic;

if (minimumHeuristic > this.calcHeuristic(possibleStates.get(pick))) {

minimumHeuristic = this.calcHeuristic(possibleStates.get(pick));

this.board = this.copyState(possibleStates.get(pick));

this.boardModified = true;

this.regMoves += 1;

possibleStates.clear();

}

} else {

this.resetBoard();

this.setBoard();

this.boardModified = true;

possibleStates.clear();

}

}

if (this.calcHeuristic(this.board) == 0) {

this.numStepsSuccess += this.regMoves;

this.numIterationsSuccess += 1;

}

return new int[]{numStepsSuccess, numIterationsSuccess, this.getResets()};

}

public int[] runRandomRestartHCSearchWithSidewayMoves() {

int currentStateHeuristic = this.calcHeuristic(this.board);

List<int[][]> possibleStates = new ArrayList<>();

this.resetRegMoves();

while (this.calcHeuristic(this.board) != 0 && boardModified) {

int columnNo = 0;

int[][] possibleState = new int[this.getNumberOfQueens()][this.getNumberOfQueens()];

for (int i = 0; i < this.getNumberOfQueens(); i++) {

if (this.getNumberOfQueens() >= 0)

System.arraycopy(this.board[i], 0, possibleState[i], 0, this.getNumberOfQueens());

}

currentStateHeuristic = this.calcHeuristic(this.board);

this.boardModified = false;

while (columnNo < this.getNumberOfQueens()) {

int queenPositionInCurrentColumn = -1;

for (int i = 0; i < this.getNumberOfQueens(); i++) {

if (possibleState[i][columnNo] == this.QUEEN)

queenPositionInCurrentColumn = i;

possibleState[i][columnNo] = this.NOT\_QUEEN;

}

for (int i = 0; i < this.getNumberOfQueens(); i++) {

possibleState[i][columnNo] = this.QUEEN;

int[][] newState = new int[this.getNumberOfQueens()][this.getNumberOfQueens()];

for (int k = 0; k < this.getNumberOfQueens(); k++) {

if (this.getNumberOfQueens() >= 0)

System.arraycopy(possibleState[k], 0, newState[k], 0, this.getNumberOfQueens());

}

if (this.calcHeuristic(this.board) >= this.calcHeuristic(newState) && this.areBoardsEqual(this.board, newState))

possibleStates.add(newState);

possibleState[i][columnNo] = this.NOT\_QUEEN;

}

possibleState[queenPositionInCurrentColumn][columnNo] = this.QUEEN;

columnNo += 1;

}

Random randomNumber = new Random();

int minimumHeuristic = currentStateHeuristic;

if (possibleStates.size() != 0) {

int pick = randomNumber.nextInt(possibleStates.size());

if (minimumHeuristic > this.calcHeuristic(possibleStates.get(pick))) {

minimumHeuristic = this.calcHeuristic(possibleStates.get(pick));

this.board = this.copyState(possibleStates.get(pick));

this.boardModified = true;

this.consecutiveSidewaysMoves = 0;

this.regMoves += 1;

possibleStates.clear();

} else if (minimumHeuristic == this.calcHeuristic(possibleStates.get(pick)) &&

this.consecutiveSidewaysMoves < this.LIMIT\_CONSECUTIVE\_SIDEWAYS\_MOVES) {

minimumHeuristic = this.calcHeuristic(possibleStates.get(pick));

this.board = this.copyState(possibleStates.get(pick));

this.boardModified = true;

this.consecutiveSidewaysMoves += 1;

this.regMoves += 1;

possibleStates.clear();

} else {

this.resetBoard();

this.setBoard();

this.boardModified = true;

possibleStates.clear();

}

} else {

this.resetBoard();

this.setBoard();

this.boardModified = true;

possibleStates.clear();

}

}

if (this.calcHeuristic(this.board) == 0) {

this.numStepsSuccess += this.regMoves;

this.numIterationsSuccess += 1;

}

return new int[]{numStepsSuccess, numIterationsSuccess, this.getResets()};

}

}

**Output**

Result\_Random\_Restart\_With\_Sideways\_Moves



Result\_Random\_Restart\_Without\_Sideways\_Moves

