***N-Queens Problem:***

The N-Queens Problem describes an N x N chess board on which N number of Queens are placed. Each queen is placed at a random row in each column.   
The aim is to place each queen in a position in that column, such that the Queens do no conflict with each other, ie no two queens can be in the same row, column or diagonal from one another. The only applicable moves are moving the queen up the column or down the column.

***Hill-Climbing Algorithm:***

The Hill-Climbing Algorithm does not concern itself with the previous iterations of the board. It only concerns itself with the current node and the neighbour nodes that are generated. In the case of the N-Queens problem, the current node is the current configuration of the board and the neighbour nodes are the various board configurations that can be generated by moving the queens.

The algorithm moves to each neighbour based on the value of the Utility Function of that neighbour or its Heuristic cost. The Hill-Climbing Algorithm chooses the neighbour node with the least Heuristic cost in case of the N-Queens problem. Hence, it is a type of greedy algorithm.

If the Heuristic cost is 0, that is the Board configuration with no conflicts and that is the solution. If the heuristic cost is not 0, then the algorithm compares the heuristic cost of the neighbours with that of the current board and selects the board configuration with the lowest Heuristic cost. In case no such neighbour has a Heuristic cost lower than the Current board’s heuristic cost, we perform a random restart since it is stuck in a local minima.

In a random restart, we generate a new initial configuration and perform the hill climbing algorithm again. This process is repeated until we find a neighbour with heuristic cost=0.

The heuristic we use in this program is:

h(n)=Number of pairs of Attacking queens.

***Program Structure:***

In this program we make use of a single class called Queen. The user is prompted to enter the number of queens for which he wishes to find a solution. In case of this program, we are able to find the solution to 70 queens in around 3 mins. Values greater than that take much longer to find solutions.

After entering the number of queens, we use a random function to generate randomly assigned positions of the queens. This is stored in an array whose size is the number of queens.

We then generate the neighbours for that array by changing the position of each queen from 0 to number of queens and calculating the heuristic cost of that particular configuration and is stored in an array of size Number of Queens x Number of Queens. This represents the heuristic cost that would be incurred by moving the queen in that column to that position. Once the heuristic cost of moving the first queen to all the different positions in the same column has been found we move to the next queen. This goes on until the entire Heuristic Matrix is filled.

We then search the matrix in case a move is present which will give us heuristic cost=0. In case such a node is found we move the queen corresponding to that column to that node and print the solution state.

In case the heuristic cost is not equal to 0 for any of the values in the HeuristicMatrix we find the minimum heuristic cost. If this cost is lesser than the Heuristic of the current board, we make that the current board and repeat the entire process.

In case the current heuristic cost is greater than or equal to the lowest heuristic cost from the heuristic matrix, we perform a random restart by generating a new initial node and repeating the process since the program is stuck in a local minima.

***Global Variables:***

**static** **int** *QueenPosArray*[]; - Stores the position of the queens.

**static** **int** *NumberOfQueens*; - Contains the number of Queens.

**static** **char** *QueenMatrix*[][]; - Used to represent the board.

**static** **int** *HCostMatrix*[][]; - Used to store the different Heuristic costs

**static** **int** *NumberOfRestarts*=0; - Contains the number of random restarts.

**static** **int** *CurrentHCost*; - Stores the Heuristic cost of the current board.

**static** **int** *NumberOfStepsOfClimb*=0; - Represents the number of neighbours that were traversed.

***Functions:***

***generateRandom*()** – Generates random row positions for each queen.  
***chooseNeighbour*()** – Used to store the heuristic cost of each neighbour in the HCostMatrix. Once done it calls **checkForSolution()** – finds least heuristic cost in the HCostMatrix and checks if it is the solution and then prints the solution accordingly. If not, it checks if it is lower than the current heuristic cost and changes the QueenPosArray to reflect the new configuration. If lowest heuristic is greater than or equal to current heuristic, it performs a random restart by calling generateRandom(). **findMinCostLocation()** – Finds the row and column with the minimum value in the HCostMatrix and returns an array of size 3 containing the row, column and value of the minimum cost in HCostMatrix. **findHcost** (**int** col, **int** row, **int**[] queenPosArray2)– Finds the heuristic cost of that board configuration by changing the row position of the queen from 0 to Number of Queens. It calculates the value of the heuristic by searching the QueenPosArray for other queens that are present horizontally, vertically or diagonally. If such a queen is encountered, it increases the paircount value. This function is called until each value in the HCostMatrix is full. **makeQueenMatrix(int[] queenPosArray2)** – This function is used to place the queen values in the QueenMatrix. **findInitialheuristic(int[] queenPosArray1)** – This function is used to generate the initial heuristic cost of the board after a random restart occurs and the values of QueenPosArray are randomized.

***Program Code:***

**import** java.io.\*;

**import** java.util.\*;

**import** java.math.\*;

**public** **class** Queens {

**static** **int** *QueenPosArray*[];

**static** **int** *NumberOfQueens*;

**static** **char** *QueenMatrix*[][];

**static** **int** *HCostMatrix*[][];

**static** **int** *NumberOfRestarts*=0;

**static** **int** *CurrentHCost*;

**static** **int** *NumberOfStepsOfClimb*=0;

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

BufferedReader br=**new** BufferedReader(**new** InputStreamReader(System.*in*));

System.*out*.println("Welcome to N Queens Problem!");

System.*out*.println("This program can find the solution upto 70 queens within 3 mins 30 secs.");

System.*out*.println("For queens greater than 70 it takes much longer.");

System.*out*.println("Please enter the number of queens:");

*NumberOfQueens*=Integer.*parseInt*(br.readLine());

*QueenPosArray*=**new** **int**[*NumberOfQueens*];

*QueenMatrix*=**new** **char**[*NumberOfQueens*][*NumberOfQueens*];

*HCostMatrix*=**new** **int**[*NumberOfQueens*][*NumberOfQueens*];

*generateRandom*();

System.*out*.println("Queen Positions are: ");

**for**(**int** i=0;i<*NumberOfQueens*;i++){

System.*out*.print(*QueenPosArray*[i]+" ");

}

*makeQueenMatrix*(*QueenPosArray*);

System.*out*.println("\nCurrent Board is as follows:");

**for**(**int** i=0;i<*NumberOfQueens*;i++){

**for**(**int** j=0;j<*NumberOfQueens*;j++){

System.*out*.print(*QueenMatrix*[i][j]+" ");

}

System.*out*.println();

}

*chooseNeighbour*();

}

**private** **static** **void** chooseNeighbour() {

//int NextNeighbour[]=new int[NumberOfQueens];

**for**(**int** i=0;i<*NumberOfQueens*;i++){

**for**(**int** j=0;j<*NumberOfQueens*;j++){

*HCostMatrix*[j][i]=*findHcost*(i,j,*QueenPosArray*);//Generates Heuristic values for each position in the HCostMatrix

}

}

//Check for solution or next best node.

*checkForSolution*();

}

**private** **static** **void** checkForSolution() {

**int** minrow=0;

**int** mincol=0;

**int** solfound=0;

**int** MinCostLocation[]=**new** **int**[3];

MinCostLocation=*findMinCostLocation*();//Finds the row, col and value of least value in HCostMatrix.

**for**(**int** i=0;i<*NumberOfQueens*;i++){

**for**(**int** j=0;j<*NumberOfQueens*;j++){

**if**(*HCostMatrix*[i][j]==0){//Checks if goal state is generated

minrow=i;

mincol=j;

solfound=1;

*QueenPosArray*[mincol]=minrow;//Changes the queen position to goal position.

System.*out*.println("Solution is found:");

System.*out*.println("Number of Restarts: "+*NumberOfRestarts*);

System.*out*.println("Number of Steps of Climbing: "+*NumberOfStepsOfClimb*);

*makeQueenMatrix*(*QueenPosArray*);

**for**(**int** k=0;k<*NumberOfQueens*;k++){

**for**(**int** l=0;l<*NumberOfQueens*;l++){

System.*out*.print(*QueenMatrix*[k][l]+" ");

}

System.*out*.println();

}

}

}

}

**if**(solfound==0){//If no solution is found

**if**(MinCostLocation[2]<*CurrentHCost*){//Compares lowest cost in HCostMatrix to CurrentHCost

*QueenPosArray*[MinCostLocation[1]]=MinCostLocation[0];//Updates the QueenPosArray with new location

*CurrentHCost*=MinCostLocation[2];//Updates CurrentHCost to new H cost.

*NumberOfStepsOfClimb*++;//Increases the steps climbed.

*chooseNeighbour*();//Generates the neighbours again.

}

**else** **if**(*CurrentHCost*<=MinCostLocation[2]){//If lowest cost in HCostMatrix is greater than or equal to current cost

*generateRandom*();//Performs random restart

*NumberOfRestarts*++;

*chooseNeighbour*();//Initiates neighbour generation for randomly restarted node.

}

}

**else** **if**(solfound==1){

System.*out*.println("End of program");

}

}

**private** **static** **int**[] findMinCostLocation() {

**int** minhcost=*HCostMatrix*[0][0];

**int** rowpos=0;

**int** colpos=0;

**int** minloc[]=**new** **int**[3];

//Finds the lowest cost in the matrix

**for**(**int** i=0;i<*NumberOfQueens*;i++){

**for**(**int** j=0;j<*NumberOfQueens*;j++){

**if**(*HCostMatrix*[i][j]<minhcost){

minhcost=*HCostMatrix*[i][j];

rowpos=i;

colpos=j;

}

}

}

minloc[0]=rowpos;//Stores position of row where lowest cost is found

minloc[1]=colpos;//Stores position of col where lowest cost if found

minloc[2]=minhcost;//Stores min cost value

**return** minloc;//returns the minloc array contating above details

}

**private** **static** **int** findHcost(**int** col, **int** row, **int**[] queenPosArray2) {

**int** temp\_var1=0;

**int** paircount=0;

temp\_var1=queenPosArray2[col];

queenPosArray2[col]=row;//Changes position of Q to row.

**for**(**int** i=0;i<*NumberOfQueens*;i++){

**for**(**int** j=0;j<*NumberOfQueens*;j++){

//Checks for horizontal and diagonal Q if Q is found it increments number of pairs

**if**(queenPosArray2[i]==queenPosArray2[j] || Math.*abs*(i-j)==Math.*abs*(queenPosArray2[i]-queenPosArray2[j])){

paircount++;

}

}

}

queenPosArray2[col]=temp\_var1;//returns Q to its original position

paircount=(paircount-*NumberOfQueens*)/2;//finds the number of attacking pairs

**return** paircount;

}

**private** **static** **void** makeQueenMatrix(**int**[] queenPosArray2) {

**for**(**int** i=0;i<*NumberOfQueens*;i++){

**for**(**int** j=0;j<*NumberOfQueens*;j++){

*QueenMatrix*[i][j]='\*';//Places '\*' symbol in all locations of array

}

}

**for**(**int** i=0;i<*NumberOfQueens*;i++){

**for**(**int** j=0;j<*NumberOfQueens*;j++){

**if**(j==queenPosArray2[i]){//Finds the row of Q in each column

*QueenMatrix*[j][i]='Q';//Replaces that position with a 'Q' symbol

**break**;

}

}

}

}

**static** **void** generateRandom() {

Random RandomNum=**new** Random();

**for**(**int** i=0;i<*NumberOfQueens*;i++){

*QueenPosArray*[i]=RandomNum.nextInt(*NumberOfQueens*);//Generates random row positions for each queen in every column

}

*findInitialheuristic*(*QueenPosArray*);//Finds the initial heuristic cost for newly randomized QueenPosArray

}

**private** **static** **void** findInitialheuristic(**int**[] queenPosArray1) {

**for**(**int** i=0;i<*NumberOfQueens*;i++){

**for**(**int** j=0;j<*NumberOfQueens*;j++){

//Checks the horizontal and diagonal positions of each Q if Q is found increments CurrentHCost.

**if**(queenPosArray1[i]==queenPosArray1[j] || Math.*abs*(i-j)==Math.*abs*(queenPosArray1[i]-queenPosArray1[j])){

*CurrentHCost*++;

}

}

}

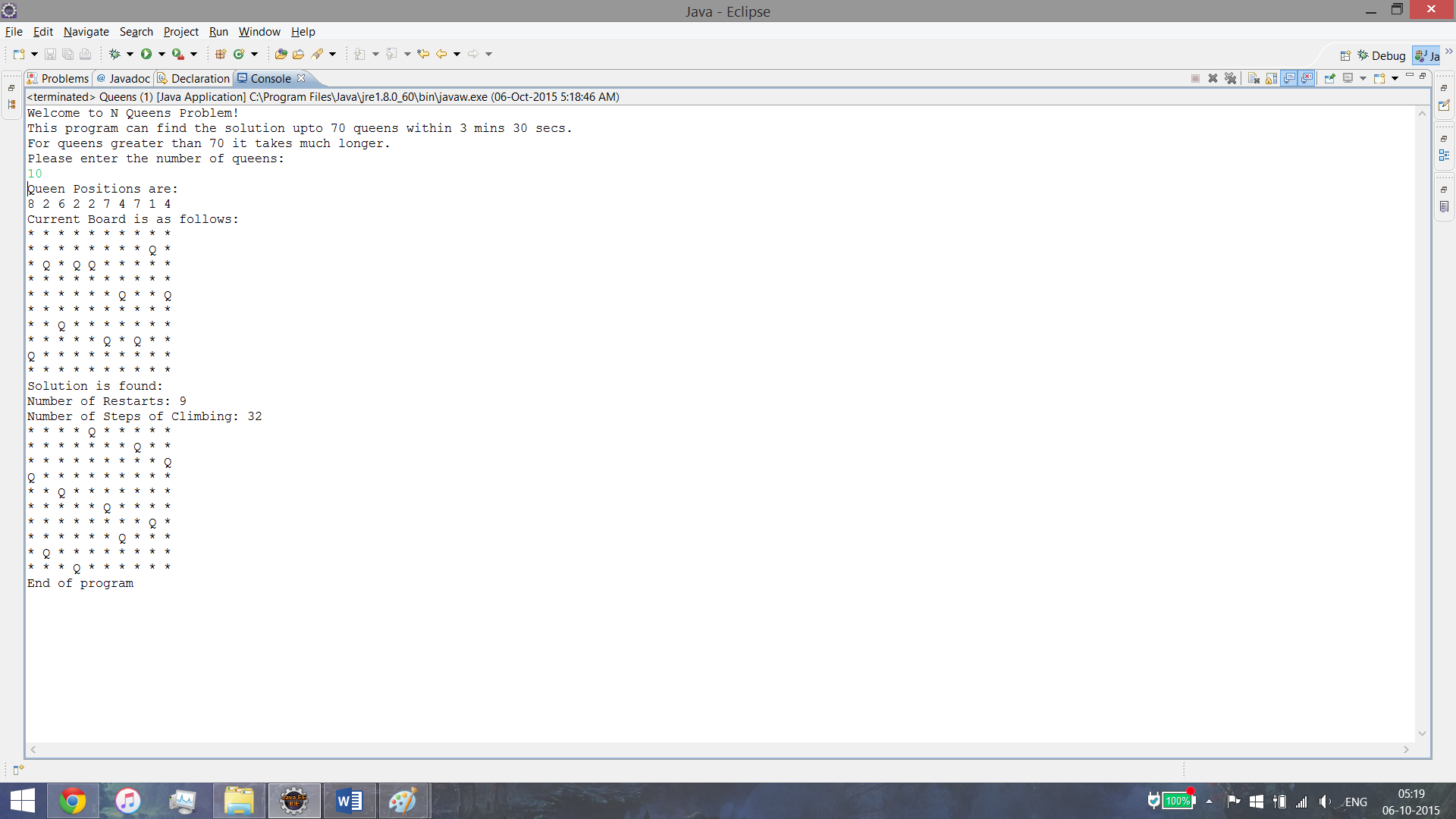
*CurrentHCost*=(*CurrentHCost*-*NumberOfQueens*)/2; //Finds the number of attacking pairs.

}

}

***Output Screens:***

***10 Queens:***



***70 Queens:***

