**Overview:**

This project deals with the N-Queens problem using Hill Climbing and random restart. This program also explores the use of Min Conflicts algorithm to solve the problem of N-Queens. The goal of the N-Queens problem is to solve for a user defined number of queens in a NxN board without any queen attacking another. Hill Climbing is used. This is an algorithm where given an initial state, we check to see if any successors of that state can give a solution closer to the goal. If one of the successors is found to be closer to the goal, we make that our current state, however if none of the successors are any closer to the goal than the initial state, we then randomly restart, meaning we generate a new board and pursue hill climbing with that board.

This is called hill climbing because we incrementally go to a better state each time, which is like climbing a hill to find the “peak”. The top of the hill is defined as the point where there is minimum heuristic value, that is, there is no queens attacking each other.

Problems with hill climbing are that the algorithm may get stuck in some local maximum, where the goal state isn’t present. To combat this, we use **random restart**. Random restart places us on another initial state on the hill climbing surface, and we incrementally find better and better successors from that new random state.

**Note:** In my program, I am able to solve for 100 queens using regular hill climbing. However, it took a long time for it to get there, approximately 45 minutes. I believe it can solve for more however, one would have to wait a good amount of time.

**How to run my program:** please execute the executable through command prompt. When that’s done the console application will ask you if you want to use the regular hill climbing/random restart method or use the min-conflicts method. Once you choose the application then asks you to enter the number of queens you want to solve for, after entering that, my program will solve for that number of queens.

**Problem Formulation:**

Initial Environment:

The N-Queens problem takes place on an NxN chess board, where initially a user picks the number of queens (N) that they want to solve for. The number of queens that the user picks is then placed on the NxN chessboard randomly with the limitation that only one queen can be in one column. However, there are no restrictions on the initial state with regards to number of queens on each row or diagonals. This means that the rows and diagonals can have multiple queens on them. For example, if the user picks 3 queens, then 3 queens will be placed on a 3x3 board, with one queen in each column at a random row.

Valid moves:

When generating a new state, we move one queen only one space, either up or down the column. We are not allowed to move any queen along the diagonal or the row.

Heuristic Function:

My program uses the total number of queens attacking each other either directly or indirectly as the heuristic. The Queens that attack directly are directly in the path of each other. The queens attacking indirectly are those queens that are behind the queen that is directly attacking on the same path. This is counted because we know that moving the first queen will not give us a favorable state because there is another queen directly behind that can attack.

For a particular queen, the number of queens attacking the queen directly or indirectly is the sum of the queens on the same row and the sum of the queens on the diagonals of the queen being considered.

*Example 1: Example of board attacking directly and indirectly*

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | |  | 0 | 1 | 2 | 3 | | 0 | Q | Q | Q | Q | | 1 |  |  |  |  | | 2 |  |  |  |  | | 3 |  |  |  |  | |  |  |  |  |

In this state, we will have a heuristic of 6. The queen at (0,0) is attacked directly by queen at (0,1) and indirectly by queen (0,2) and (0,3). The queen at (0,1) is attacked directly by (0,2) and indirectly by (0,3). The queen at (0,2) is attacked directly by (0,3).

*Example 2: Example of attacking only directly*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 |
| 0 | Q |  |  | Q |
| 1 |  | Q | Q |  |
| 2 |  |  |  |  |
| 3 |  |  |  |  |

In this example, there are only directly attacking pairs: the queen at (0,0) is attacked directly by queen at (1,1). Queen (0,0) is also attack directly by queen at (0,3). Queen at (1,1) is attacked directly by queen at (1,2). Queen at (1,2) is attacked directly by queen (3,0). So the heuristic will be 4.

Note: although its true that queen (0,0) is attacking queen at (1,1) directly and queen(1,1) is attacking queen at (0,0) directly, remember we are only considering attacking pairs so we do not double count.

*Example 3:*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 |
| 0 | Q |  |  |  |
| 1 |  | Q | Q |  |
| 2 |  |  |  |  |
| 3 |  |  |  | Q |

In this example we have a heuristic of 5. The queen at (0,0) is attacked by queens (1,1) and (3,3). Queen at (1,1) Is attacked by queens (1,2) and (3,3) and finally queen at (1,2) is attacked by queen at (3,3).

*Example of Goal State:*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 |
| 0 |  | Q |  |  |
| 1 |  |  |  | Q |
| 2 | Q |  |  |  |
| 3 |  |  | Q |  |

Notice how no queens attack each other in any way. This heuristic would be 0.

Goal State:

the goal is reached when each of the queens are placed on the chess board such that no queens can attack each other. This will give a heuristic value of 0 when achieved.

**Program Board Description**:

In my program, I represent the entire board by a 1 D array. The indices of the array are the column numbers and the values inside the indices are the rows. Thus if we want to find a queen on the same row, we only need to traverse the array and find two column values that are equal. If we want to find a queen on the diagonal, we only need to increase and decrease the current row value by the quantity row(i)-row(j), where row(i) is the row of the current queen and the row(j) is the row of the adjacent blocks. We vary i and j to find the queens on the diagonals.

**Functions Descriptions**

**Program Class:**

**Heuristic**:

This inputs the N-Queens board and the number of queens

this returns an int, which is the heuristic value of the board.

1. For each queen we traverse the column that it is on as described by the state description above.
2. For each queen we traverse the diagonals both forward and backward as described by the board description above.
3. We count the number of directly and indirectly attacking queens for the queen of interest.
4. The sum of the indirectly and directly attacking queens is the heuristic.

**Main function**:

This function is where everything begins. The steps are as follows:

1. Ask the user to input whether they want to try the min-conflicts function to find the solution or the regular hill climbing,.
2. If they choose min conflict function, then we feed the randomly generated board into the min conflicts function.
3. If they chose regular hill climbing:
   1. We generate a random board
   2. Find its heuristic value by feeding board into the heuristic function
   3. Check to see if the heuristic value is 0, if it is, then we are done, print the board
   4. If it is not, then we find the best successor for the board by feeding the board into the successor function
   5. we check to see if the bestSuccessors h value is less than the current one, if so, then we replace the intial board witht his current board and the initial h value with the best successors value
   6. if the successors h value is worse, then we randomly restart by generating a new board (we increment the number of random restarts variable as well).
   7. We repeat steps b-f until we find a board with a heuristic value of 0.

**Generate Random Board**:

returns a random board ( int[])

The input to this function is the number of queens the user picks (N)

* This function basically generates a 1D array where the columns are represented by the array indices and the rows are the values (between 0 and N-1).
* This function then returns the integer array.

**Successor Function:**

This function inputs the N-Queens board and the number of queens

Returns the least heuristic value successor object that has the N-Queens board and its associated heuristic value.

1. Save the original row value for the column we are interested in
2. Decrease the row value at the column until it is 0.
   1. Example: if the row in the column of interest is 3, that means that the queen is in row 3. The row value is decreased to 2,1 and 0, this is the way to place the queen on each row.
3. After decreasing the row, find the boards heuristic
4. Using the board and the heuristic, create an object and add it to the nodeList
5. Set the row back to the original value when done
6. Now increase the row values until we hit the number of queens
7. Every time you increase the row value, you get the board heuristic
8. Once you get the heuristic, we create the object using the current board and the heuristic
9. We add that object to the nodeList
10. The nodeList now has all the successor of the original board
11. We find the successor object in the nodeList with the minimum heuristic
12. We return the successor.

**PrintBoard:**

Inputs the board and the number of queens, along with the number of random restarts and number of states traversed.

Outputs the board and the number of states along with the number of random restarts.

**NQueensNode Class:**

**Private variables:**

nQueensBoard: this is an integer array that represents the nQueensBoard with a particular configuration

heuristic: this is an integer value which is the heuristic of the nQueensBoard.

**NQueendsNode constructor:**

Inputs the nQueensboard, the heuristic and the number of queens. Sets the field values to these input values.

**Properties:**

Nqueens: this property is of type int[], this is basically a get and set method for the Nqueens board. So this returns a int[] in its get method.

Hproperty: this property is of type int, this are get and set methods for the heuristic value. So this returns an int in its get method.

Extra Credit:

**Min-Conflicts Function:**

The min conflict’s algorithm first finds two conflicting queens on the board, and then picks one randomly. The randomly chosen queen is then moved to a spot in its own column that gives the least heuristic. This is repeated over and over again until the goal is reached. In this function, there is a set maximum number of states allowed to be traversed. Once this maximum limit is reached, we randomly generate a new board, and start all over again. The flow of the algorithm is the following:

1. Feed in the initial board
2. Find a pair a queens with a conflict, this is done by using the “ColumnWithConflicts” function.
3. Choose one of the conflicting queens randomly (also done by the “ColumnWithConflicts” Function.
4. Check each position in the column that the queen is in, find the position in the column with the least number of conflicts for the entire board.
5. Move the queen to that position
6. Repeat steps 2 to 5 until we find a goal.
7. If at anytime the number of states traversed with one board reaches the maximum limit of 10,000, generate a new board and start all over again.

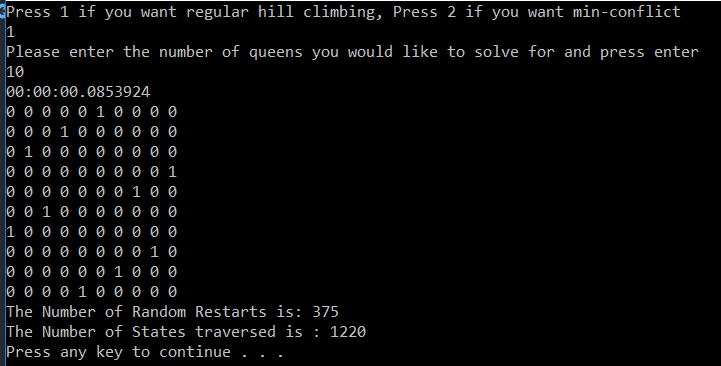
**ColumnWithConflicts**:

The inputs the current board and finds an instance where two queens are threatening each other. Once it finds that, it then picks on randomly and returns the column number. If no conflicts are found it returns -1.

1. Input the board and the number of queens
2. Search the board to see if any queens are conflicting in the row
3. If we find a queen with a conflict, then pick, at random, one of the queens columns and return it
4. If no conflicting is found on the row, then search the diagonals.
5. If there is a conflicting queen on the diagonals then return, at random, one of the queens columns.

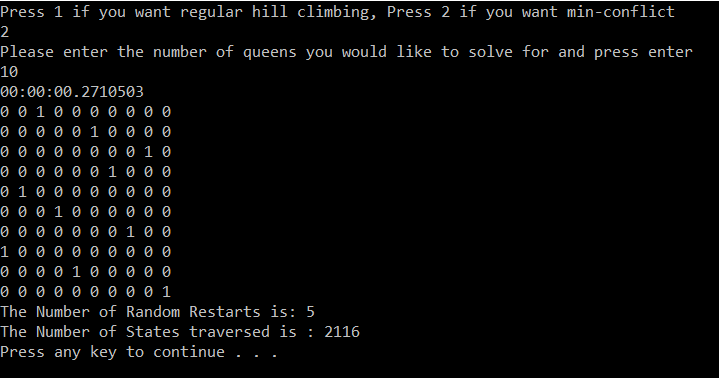
**Performance:**

**Regular Hill Climbing:**



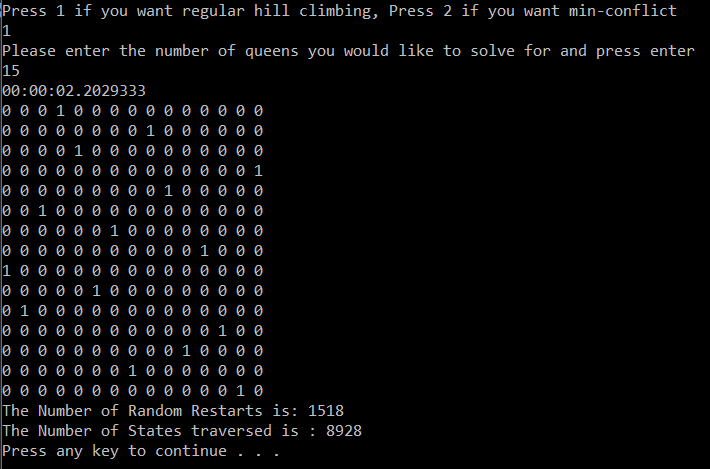
We can see that in regular hill climbing, for 10 queens, we traverse 1220 states and have to do 375 random restarts. The time taken for 10 queens is .0853 seconds for the regular algorithm.

**Min Conflicts Algorithm Performance:**

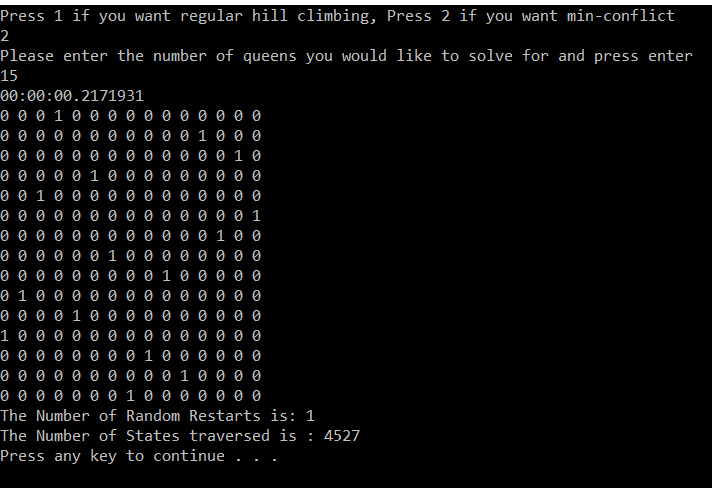


As we can see with the min conflicts algorithm, we only needed 5 random restarts and traversed only 2116 states in the last restart. The time required to do this was .27105 seconds.

**Regular Hill Climbing for 15:**



**Min Conflicts for 15:**



If we compare the regular hill climbing algorithm and min conflicts algorithm for 15, we can see that the regular hill climbing took longer, (around 2 seconds), however the min conflict took only .2 seconds. Furthermore, we only need 1 restart for min conflict where as we needed 1518 random restarts for the regular hill climbing algorithm.

Comparison of Regular Hill Climbing and Min Conflicts for various amounts of queens.

\*Note: All quantities (Time, Random Restarts and States Traversed), are an average of 3.

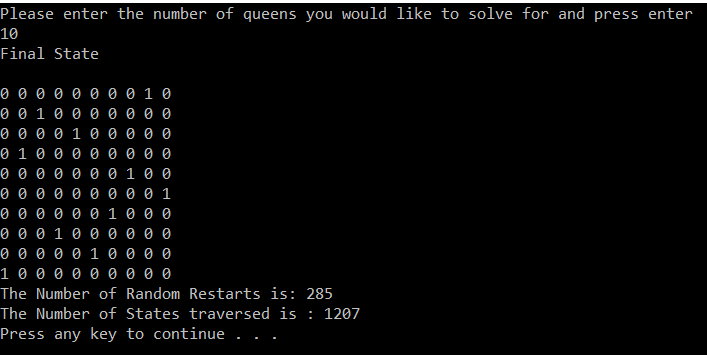
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | **MinConflict** |  |  |
| Number of Queens |  | Time to Goal (seconds) | Random Restarts | States Traversed |
| 10 |  | 0.27 | 5 | 2116 |
| 15 |  | 0.217 | 1 | 4527 |
| 20 |  | 21.99 | 72 | 1925 |
|  |  |  |  |  |
| r |  | **Regular Hill Climbing** |  |  |
| Number of Queens |  | Time to Goal (seconds) | Random Restarts | States Traversed |
| 10 |  | 0.0853 | 375 | 1220 |
| 15 |  | 2.2 | 1518 | 8928 |
| 20 |  | 1.23 | 207 | 1744 |

We can see that for larger values, we have less number of random restarts for Min Conflict. Overall Min Conflicts gives better performance. However, we cannot really judge based on these trials because it really depends on the initial state that each run starts at.

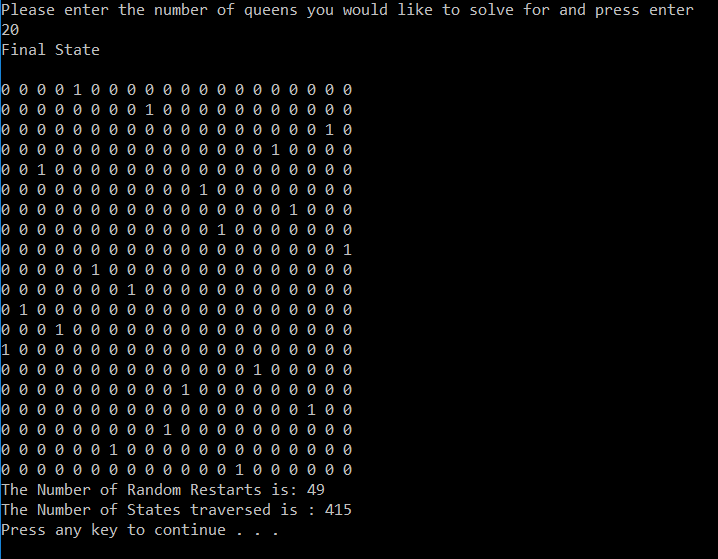
**Other Examples of Program using regular Hill Climbing: the 100 queens example is at the bottom**

Example 1:

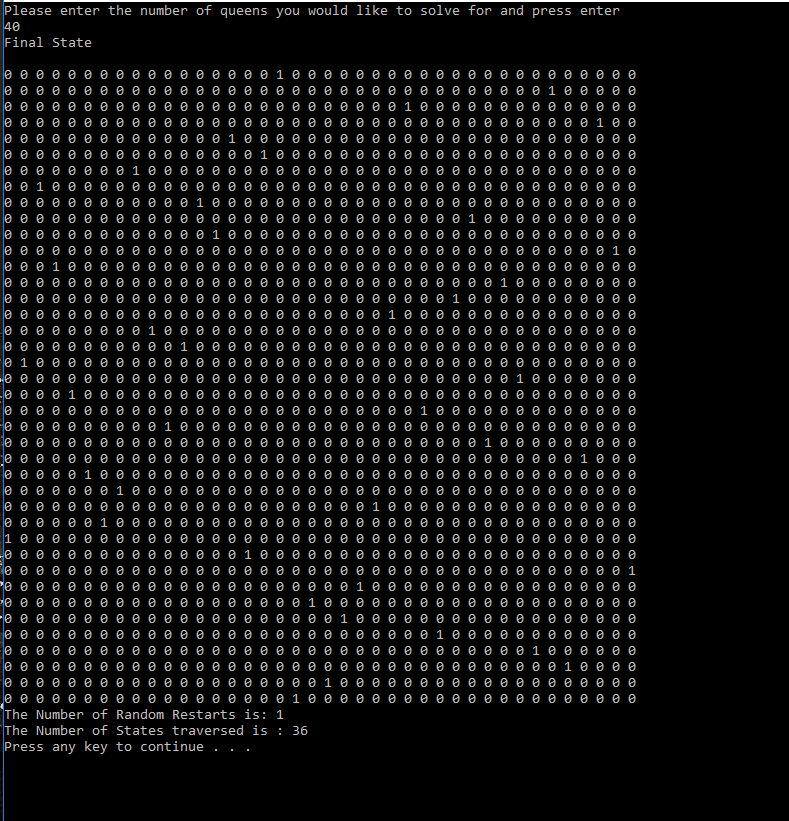
For 10 queens:



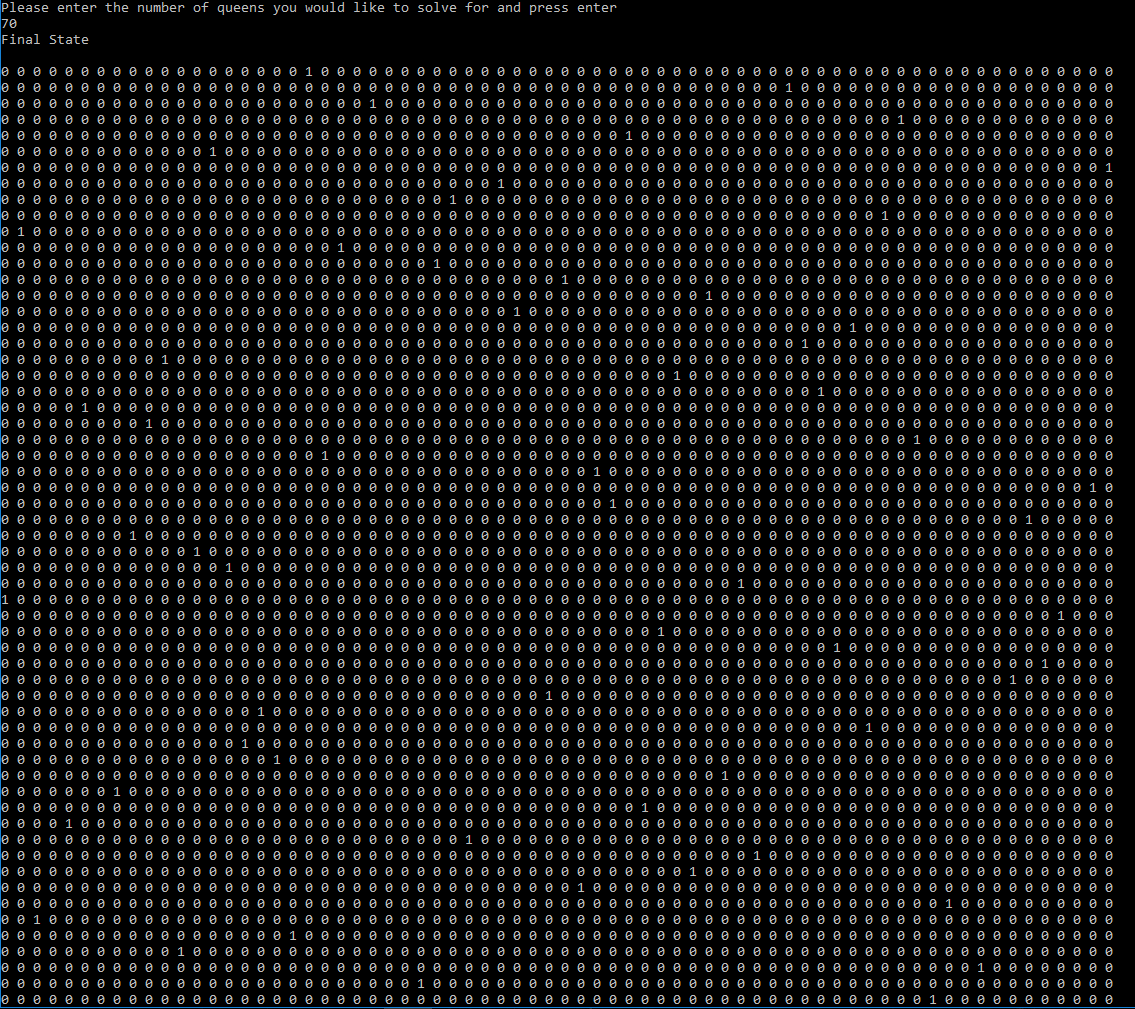
For 20 queens:



For 40 Queens:

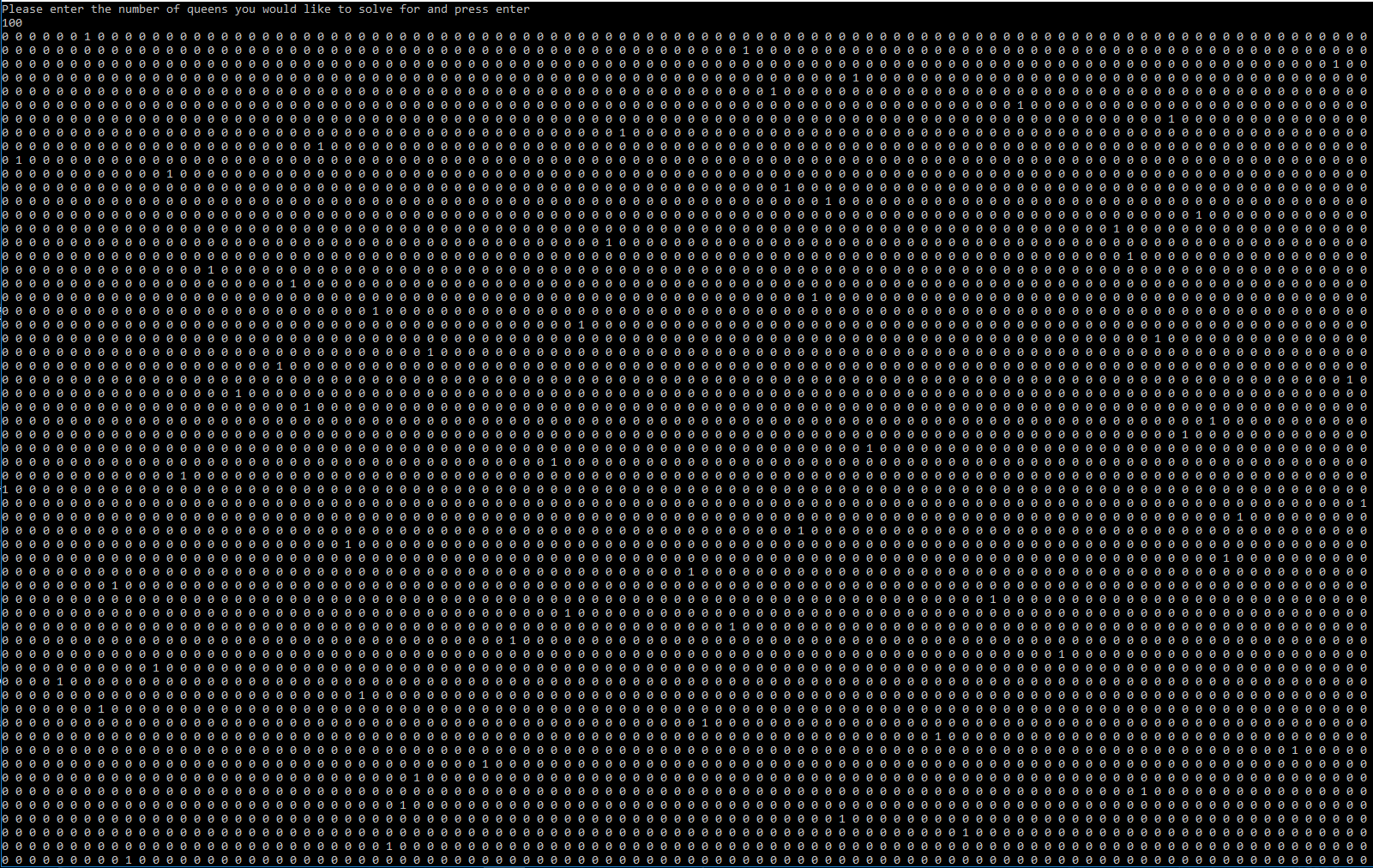


For 70 Queens:





For 100 Queens:





**PROGRAM CODE:**

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace nQueens

{

class Program

{

static void Main(string[] args)

{

// Define variables for number of random restarts and the number of states traversed:

int numRandRestarts = 0;

int numStates = 0;

//Ask User to enter the number of queens to solve for

Console.WriteLine("Press 1 if you want regular hill climbing, Press 2 if you want min-conflict");

int choice = Convert.ToInt32(Console.ReadLine());

Console.WriteLine("Please enter the number of queens you would like to solve for and press enter");

int numOfQueens= Convert.ToInt32(Console.ReadLine());

int[] nQueensBoard = new int[numOfQueens];

//Generate the raondom board and find the heuristic

nQueensBoard = generateRandomBoard(numOfQueens);

int h=Heuristic(nQueensBoard, numOfQueens);

// start the stop watch to see how long the algorithm runs

Stopwatch watch = new Stopwatch();

if(choice == 2)

{ // this starts the min conflict algorithm for the user if the user wants

MinConflictsSolution(nQueensBoard, numOfQueens);

}

else

{

watch.Start();

do

{ // check to see if the board's hueristic is 0, if it is, print the board

if (h == 0)

{

watch.Stop();

Console.WriteLine(watch.Elapsed);

printBoard(nQueensBoard, numOfQueens,numStates,numRandRestarts);

break;

}

// if heuristic is not 0, then find the boards lowest heuristic valued successor

nQueensNode bestSuccessor = SuccessorFunction(nQueensBoard, numOfQueens);

// see if the best successors value is less than the current states heurisitc

if (bestSuccessor.Hproperty < h)

{

// if it is, then replace the current board and hueristic value with that of the successors

nQueensBoard = bestSuccessor.Nqueens;

h = bestSuccessor.Hproperty;

numStates++;

}

else

{

// if the best successors heuristic is greater than that of the current state, generate a new board

nQueensBoard = generateRandomBoard(numOfQueens);

h = Heuristic(nQueensBoard, numOfQueens);

numRandRestarts++;

}

} while (true);

}

}

public static int[] generateRandomBoard(int numOfQueens)

{ // this method generates a random board, we intialize the board to all 0's first

int[] nQueensBoard = new int[numOfQueens];

Random rand = new Random();

// we then put a queen randomly in a row in each column, this loop ensures that there is only 1 queen per column

for (int i = 0; i < numOfQueens; i++)

{

nQueensBoard[i] = Convert.ToInt32(rand.Next()) % numOfQueens;

}

return nQueensBoard;

}

public static void printBoard(int[] nQueensBoard,int numOfQueens,int numStates, int numRestarts)

{

int[,] finalBoard = new int[numOfQueens, numOfQueens];

for(int i = 0; i < numOfQueens; i++)

{

finalBoard[nQueensBoard[i], i ] = 1;

}

//print the board

for(int i =0; i < numOfQueens; i++)

{

for(int j = 0; j < numOfQueens; j++)

{

Console.Write(finalBoard[i, j] + " ");

}

Console.Write(Environment.NewLine);

}

// prints the number of random restarts and the number of states

Console.WriteLine("The Number of Random Restarts is: " + numRestarts);

Console.WriteLine("The Number of States traversed is : " + numStates);

}

public static int Heuristic(int[] Board, int numQueens)

{

int heuristic = 0;

for(int i = 0; i < numQueens; i++)

{

for(int j = i + 1; j < numQueens; j++)

{ // loop through the array, if any of the columns are equal, then increment the heuristic

// this is beacuse there are 2 qeeuns on the same row

if (Board[i] == Board[j])

{

heuristic++;

}

// loop through the 1d arrayand ehck the forward and backward diagonal of every queen, if

// there is a queen, then increment the heuristic value

if(Board[i]== Board[j]+ (j-i) || Board[i] == Board[j] - (j - i))

{

heuristic++;

}

}

}

// return the heuristic

return heuristic;

}

public static nQueensNode SuccessorFunction(int[] Board,int numQueens)

{ // initialize a list of nodes

List<nQueensNode> nodeList = new List<nQueensNode>();

for(int i =0; i < numQueens; i++)

{

int originalValue = Board[i];

while (Board[i] != 0)

{ // go through each column, decrement the row value, find the heuristic and create the object

// store the object in a list for further processing

Board[i]--;

int h = Heuristic(Board, numQueens);

nodeList.Add(new nQueensNode(Board, h, numQueens));

}

Board[i] = originalValue;

while (Board[i]+1 < numQueens)

{ // go through each column, increment the row value, findthe heuristic, and create the object

// store object in nodeList for further processing

Board[i]++;

int h = Heuristic(Board, numQueens);

nodeList.Add(new nQueensNode(Board, h, numQueens));

}

Board[i] = originalValue;

}

// find the successor in the nodeList that has the least heuristic value

nQueensNode Successor = nodeList.Aggregate((n1, n2) => n1.Hproperty < n2.Hproperty ? n1 : n2);

// return the successor

return Successor;

}

public static void MinConflictsSolution(int[] board, int numQueens)

{

Stopwatch watch = new Stopwatch();

// initialize new board

int[] newBoard = new int[numQueens];

// set the max number of steps that min conflicts is allowed to take before random restart

int maxSteps = 10000;

// set numStates that we traversed and the number of random restarts to 0

int numStates = 0;

int numRestarts = 0;

int[] boardWithLeastH = new int[numQueens];

// copy the board to a new board

for (int i = 0; i < numQueens; i++)

{

newBoard[i] = board[i];

}

watch.Start();

while (true) {

// find a column where there is a conflicting queen

int columnConflict = TheColumnWithConflicts(newBoard, numQueens);

int leastHeuristic = 10000;

// iterate the board at that confliciting column

for (int i = 0; i < numQueens; i++)

{

newBoard[columnConflict] = i;

int h = Heuristic(newBoard, numQueens);

if (h < leastHeuristic)

{ // this loop finds the board with the least hueristic and stores the board and the

// hueristic

leastHeuristic = h;

for (int j = 0; j < numQueens; j++)

{

boardWithLeastH[j] = newBoard[j];

}

}

} //end of for loop

// check to see if the leastHeuristic found is 0, if so we have our goal

if (leastHeuristic == 0)

{

watch.Stop();

Console.WriteLine(watch.Elapsed);

printBoard(boardWithLeastH, numQueens, numStates, numRestarts);

break;

}

else

{

numStates++;

for (int i = 0; i < numQueens; i++)

{ // if the boardWith least huristic is not 0, then we reset newBoard with this board with least heuristic

newBoard[i] = boardWithLeastH[i];

}

}

// if we have reached the max number of steps allowed, we generate a new board

if (numStates == maxSteps) {

numStates = 0;

numRestarts++;

newBoard = generateRandomBoard(numQueens);

}

//repeat in a while loop until goal is found

}// end of while

}// end of min Conflicts

public static int TheColumnWithConflicts(int[] Board,int numQueens)

{

// find two rows with the same conflict and jsut pick one randomly:

Random rand = new Random();

int num;

for (int i = 0; i < numQueens; i++)

{

for (int j = i + 1; j < numQueens; j++)

{ // if two rows are queal that means that 2 queens are conflicting

if (Board[i] == Board[j])

{ // rnadomly select either queen and return it

num = Convert.ToInt32(rand.Next()) % 2;

if (num == 1)

{

return i;

}

else

{

return j;

}

}

else if (Board[i] == Board[j] + (j - i) || Board[i] == Board[j] - (j - i))

{ // search the diagonla to see if any queens are conflicting, randomnly choose a

//queen and return the column

num = Convert.ToInt32(rand.Next()) % 2;

if (num == 1)

{

return i;

}

else

{

return j;

}

}

}

}

// if no conflicts, return -1

return -1;

}

} //end of class

} //end of namespace

**NQUEENS NODE CLASS:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace nQueens

{

class nQueensNode

{

// private variables corresponding to the nQueens board and the hueristic value

private int[] nQueensBoard;

private int heuristic;

// Basic constructor for the Queen Node object, only stores the nQueen Board and the corresponding heuristic

public nQueensNode(int[] nQueens, int heuristic,int numQueens)

{

nQueensBoard = new int[numQueens];

nQueensBoard = nQueens.Clone() as int[];

this.heuristic = heuristic;

}

//Public Properties (get and set methods) corresponding to the nQueens board and the heuristic value

public int[] Nqueens { get { return nQueensBoard; } set { nQueensBoard=value; } }

public int Hproperty { get { return heuristic; } set { heuristic=value; } }

}

}