# Term Project: N-Queens

CP 468 Artificial Intelligence

Dr. Ilias Kotsireas

Group 18


Arshdeep Grewal - 170264950

Raman Kala - 120869110

Jiachun Xiang - 160365750

Danni Shang - 150675520


2020.12.06

**Github Link:** https://github.com/mpa-mxiang/n-queens.git **(minconflict.py)**

**Discussion - Data Structures and Functions**

**Creating the Board:**

We had two functions that could be used to create the chess board: createInitialdomain() and createRandomdomain(). createInitialdomain() is what the program generally uses, since it can create a board that is initially already "closer" to the solution. In createInitialdomain(), we first create an ordered set that stores the indices of all rows (rowsRemaining). We then iterate through each column in the board and pop the first value out of the rowsRemaining set to get a square to place the queen on. Using this column and row value, we find the number of conflicts (number of queens that are in the same row, left diagonal, or right diagonal as the square). If there are no conflicts, then we append this row index to the board and call updateConflicts(). If there are conflicts, then we place the row index back into rowsRemaining and repeat the process again with the next row. We decided to repeat this process twice because then the program wouldn't have to spend too much time generating a board, but could still generate a fairly good initial board. If both column and row pairs have conflicts, then we append the column to a list called colsRemaining, which we'll fill in at the end with whatever rows are left in rowRemaining, without checking for number of conflicts.

createRandomdomain() is only called when the global variable infiniteLoop is set to true. This happens when the program detects that on the last attempt to reach a solution, a queen would repeatedly be moved back and forth between the same 2 rows. To break out of this infiniteLoop, on the next attempt, a random board is generated by randomly shuffling rowsRemaining. While createInitialdomain() always gives us the same initial board, createRandomdomain() will give us different initial boards, so it won't always get stuck in a loop between 2 rows.

**Min-Conflicts Algorithm and Updating Conflicting Queens:**

We designed the minimum-conflict algorithm to handle the actual meat of the program, this case being solving the n-queen problem. It's a search algorithm that is used to solve constraint-satisfaction problems, where the constraints in the problem of n-queens would be the number of possible attacking queens. The minimum conflicts algorithm we designed searches each column on the board, and locates the queen with the highest number of conflicts using an auxiliary function called maxCol(). Once the queen with the highest number of conflicts is returned, a search for each row in that column is done to look for the least amount of conflicts. This is accomplished with the function minConflict() which returns the row with the fewest number of attacking queens, and moves the queen to that row. Afterwards, the arrays storing conflicts in each row (numRow), right diagonally (numRightDiag), and left diagonally (numLeftDiag) are updated using the function updateConflicts(), and this process is repeated until there is a solution. In the minConflict() function itself, first the minimum number of conflicts(minimum_conflict) is set equal to the total number of queens(numQueens), which is n. An empty list(minRows) is created to store rows that have a minimum number of attacking queens, and each row in every column is iterated through. A calculation is done to determine the total number of attacking queens, and if it's less than minimum_conflict, then minimum_conflict is updated. Afterwards, minRows is set to the index of the current row, and if the number of conflicting queens is equal to minimum_conflict, then minRows is updated. This entire process builds minRow to have a list containing the rows with least number of conflicts from attacking queens, and returns minRow when called for.

**Maximum Conflicts Column Heuristic:**

The maxCol() function is a heuristic used to decide which queen on the board to reposition next. The function returns the index of the column that contains the most conflicting queens. The function iterates through each column on the board and calculates the number of conflicting queens by adding the number of queens in the same row, left diagonal, and right diagonal. This is done using the row list, left diagonal list, and right diagonal list (numRow, numRightDiag, and numLeftDiag). The column with the largest number of conflicting queens is then returned, with ties being stored in the conflictColsList list and broken at random using random.choice().

**Input and Output Format:**

For Input, we ask for N in the console. Then, for output, it would be stored in both the console and another text file called “output.txt”. In the console, the output would display the diagram for the solution. However, in the output text file, it displays a list which indexes the location for each queen. The values in the list represent the row number in that column where the queen would be placed. Additionally, the output can be visualized with matplotlib to have a more clear and simple view where each queen locates.

**Solve():**

The solve() function is used to run the program for a number of iterations that doesn’t exceed the maximum number of iterations. For the maximum number of iterations, we decided to keep the number at 60% of the total number of queens (0.6 * numQueens), because this number allowed the program to generate a solution without running for an excessively long time. For smaller boards where n < 100, we increased the number to equal to the number of queens, since this proved to be enough iterations for the program to find a solution. During each iteration, a column is selected using the maxCol() heuristic. If the number of conflicting queens in the column is 3, then we have arrived at a solution, since there would only be a maximum of 1 queen in each row, left diagonal, and right diagonal. Otherwise, the program calls minConflict() to decide which row to put the queen in, and then calls updateConflicts() to remove and add conflicting queens (-1 and +1) in each of the 3 conflict arrays for the given column and row. solve() returns true if a solution was found; otherwise, it returns false and the program will continue to call solve() until a solution is found.

**How to Install, Compile, and Execute the Code: - Arshdeep write this one.**

To install, compile and Execute the N-queens problem we can follow the steps outlined below:

**To install:**

To download the code source files you can head over to

https://github.com/mpa-mxiang/n-queens

To download you can:

1. Use the Github Desktop app if you have.
2. Download the zip file and extract files.
3. Use Git Bash and clone the repository.

**To run the code:**

**System requirements:**

- Have the latest version of python installed.
- Install the matplotlib library:

```
> pip install matplotlib
```

- Install the seaborn library.

```
> pip install seaborn
```

**Note:** For the input, enter the size of board configuration in the input.txt file. Input any number 10, 100, 1000, 10000, 100000, 1000000 and then add the input.txt file as parameter when executing the minconflicts.py file.

1. Make sure you have the most up to date version of the code and everything is saved/compiled.
2. You can either run the program traditionally by hitting the run button depending on the IDE and changing the parameters in the input.txt file that you would like to use for input, OR you can use the terminal command to execute:

```
> python3 minconflict.py input.txt
```

3. To see the results/output you can check the output.txt file which is automatically generated when the program is executed.

**Source Code(minconflict.py):**

```
from math import trunc
import random
import time
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import seaborn
from typing import Generator, List

# creates an initial domain


def createInitialdomain():
    # use global keyword so that we can modify each variable inside the
function
    global domain
    global numRow
    global numRightDiag
    global numLeftDiag

    # create a list to store the chess domain
    domain = []

    # initialize the lists for # of queens in each row, left diagonal, and
right diagonal
    # the size of the lists are the # of rows and # of diagonals in the
domain
    # initially, there are 0 queens in each list
    numRow = [0] * numQueens
    numRightDiag = [0] * ((2 * numQueens) - 1)
    numLeftDiag = [0] * ((2 * numQueens) - 1)

    # create an ordered set of all row index values
    rowsRemaining = set(range(0, numQueens))
    # create a list that keeps track of which columns have not been solved
for
    colsRemaining = []
```

```
    # iterate through each column
    for col in range(0, numQueens):
        # test a row in the column by popping the first row index out of
the set of all rows
        tryRow = rowsRemaining.pop()

        # add the total # of queens that can attack the square using col
and tryRow
        conflicts = numRow[tryRow] + numRightDiag[col +
                                                  tryRow] +
numLeftDiag[col + (numQueens - tryRow - 1)]

        # if there are 0 attacking queens, then immediately place the
queen at that position
        # this is to make as "good" an initial domain as possible to make
it easier to find a solution
        if conflicts == 0:
            # append the row index to the domain list to show the queen's
positioning
            domain.append(tryRow)
            # update row and diagonal lists, passing add = 1 to show that
conflicts need to be added
            updateConflicts(col, tryRow, 1)

        else:
            # if there are attacking queens, then add the row to the back
of the set to test it again later
            rowsRemaining.add(tryRow)
            # pop the next row from the set to test
            tryRow = rowsRemaining.pop()
            conflicts = numRow[tryRow] + numRightDiag[col +
                                                      tryRow] +
numLeftDiag[col + (numQueens - tryRow - 1)]

            # if there are no attacking queens, place the queen at that
location and append the row index to the domain, update conflicts
            if conflicts == 0:
```

```
                domain.append(tryRow)
                updateConflicts(col, tryRow, 1)

            else:
                # Otherwise, add this row at the back of the set as well
to test it later
                rowsRemaining.add(tryRow)
                # add "none" to the domain to show that the column has not
yet been filled
                domain.append(None)
                # append the column index to the colsRemaining list to
come back to that column later
                colsRemaining.append(col)

    # after iterating through all columns in the domain, iterate through
the list of all columns that have not yet been placed
    for col in colsRemaining:
        # place the queen at the first row in rowsRemaining, regardless of
if there are conflicts or not to finish creating the initial domain
        domain[col] = rowsRemaining.pop()

        # update row and diagonal lists, passing add= 1 to show that
conflicts need to be added
        updateConflicts(col, domain[col], 1)


# creates a random domain (called if there's an infinite loop)
def createRandomdomain():
    # use global keyword so that we can modify each variable inside the
function
    global domain
    global numRow
    global numRightDiag
    global numLeftDiag

    # create a list to store the chess domain
    domain = []
```

```
    # initialize the lists for # of queens in each row, left diagonal, and
right diagonal
    # the size of the lists are the # of rows and # of diagonals in the
domain
    # initially, there are 0 queens in each list
    numRow = [0] * numQueens
    numRightDiag = [0] * ((2 * numQueens) - 1)
    numLeftDiag = [0] * ((2 * numQueens) - 1)

    # create an ordered set of all row index values
    rowsRemaining = list(range(0, numQueens))
    # randomly shuffle the list
    random.shuffle(rowsRemaining)

    # iterate through each column
    for col in range(0, numQueens):
        # pop the first row from the list
        tryRow = rowsRemaining.pop()
        # append the row index to the domain list to show the queen's
positioning
        domain.append(tryRow)
        # update row and diagonal lists, passing add = 1 to show that
conflicts need to be added
        updateConflicts(col, tryRow, 1)


# whenever a queen is moved to/from a position, there will be changes made
to the row, left diagonal, and
# right diagonal lists that store the # of queens. this function updates
the lists.
# parameters: col - column that queen is moved to/from
# row - row that the queen is moved to/from
# add - if 1 (queen is moved to a position), will add a queen to the list;
# if -1 (queen is moved from a position), will subtract a queen from the
list
def updateConflicts(col, row, add):
    # update the row, right diagonal, and left diagonal lists that the
queen is moved from/to by either adding or subtracting 1
```

```
    numRow[row] += add
    numRightDiag[col + row] += add
    numLeftDiag[col + (numQueens - row - 1)] += add


# finds the row in the column that has the fewest # of attacking queens
(min-conflicts algorithm).
# this is where the queen will be moved to
# parameters: col - the index of the column that the queen is in
def minConflict(c):
    # initially set minimum # of conflicts equal to total # of queens
    minimum_conflict = numQueens
    # list to store the rows with minimum # of attacking queens
    minRows = []

    # iterate through each row in the column
    for r in range(numQueens):
        # calculate the total number of attacking queens by adding the
number of queens in that row, left diagonal, and right diagonal
        conflicts = numRow[r] + numRightDiag[c + r] + \
            numLeftDiag[c + (numQueens - r - 1)]

        # if the number of queens is less than minimum_conflict, update
minimum_conflict and set the minRows list as containing only the index of
the current row
        if conflicts < minimum_conflict:
            minRows = [r]
            minimum_conflict = conflicts

        # if the number of queens is equal to minimum_conflict, append the
row index to the list
        elif conflicts == minimum_conflict:
            minRows.append(r)

    # select randomly a row index from list of rows with smallest # of
queens
    minRow = random.choice(minRows)
    return minRow
```

```
# heuristic that finds the column with the most # of attacking queens;
will move queen in this column next
def maxCol():
    con = 0
    maximum_conflicts = 0
    # create a list to store the index of the max column
    conflictColsList = []

    # iterate through all the columns
    for c in range(0, numQueens):
        # get the row value for the current column (where the queen is
currently placed)
        r = domain[c]
        # find the # of attacking queens
        con = numRow[r] + numRightDiag[c + r] + \
            numLeftDiag[c + (numQueens - r - 1)]
        # if the # of attacking queens is greater than the current max,
then set the column as maximum_conflictsCol and update maximum_conflicts
        if (con > maximum_conflicts):
            conflictColsList = [c]
            maximum_conflicts = con
        # if the column ties with the current max column, then append the
index value to the conflictColsList list
        elif con == maximum_conflicts:
            conflictColsList.append(c)
    # Randomly choose from the list of tied columns
    maxCol = random.choice(conflictColsList)
    return maxCol, maximum_conflicts


# sets up an initial domain by calling createInitialdomain() and then
returns true if a solution is found
# returns false if a solution is not found under the given number of
iterations
def solve():
    global infiniteLoop
```

```
    # if there was an infinite loop detected, create a random domain
    if (infiniteLoop == True):
        createRandomdomain()
    else:
        # else, call createInitialdomain() to create a domain that's
"close" to the solution
        createInitialdomain()
    # counts the number of iterations
    iteration = 0
    # keeps track of the movement of the queen between iterations
    positions = []

    # # of iterations that the program will run for during each new
attempt to find a solution
    # keep it under numQueens to allow the program to run faster; stop an
attempt if a solution is not found within the max # of iterations and
start a new attempt
    # for smaller domain sizes, totalIterations needs to be increased so
that there are enough iterations to successfully reach a solution
    if numQueens < 100:
        totalIterations = numQueens
    else:
        totalIterations = 0.6 * numQueens

    # continue while current # of iterations doesn't exceed max iterations
    while (iteration < totalIterations):
        # find the column with the most attacking queens
        # return the column index and the # of attacking queens
        col, numConflicts = maxCol()
        # print("conflicts:", numConflicts)
        # print("col:", col)
        # if the total # of conflicts is greater than 3 (1 in row, 1 in
left diagonal, and 1 in right diagonal), then there are more than 1
attacking queen
        if (numConflicts > 3):
            # call minConflict to find the row the move the queen to
(returns the row with the smallest # of attacking queens)
```

```
            position = minConflict(col)
            # append the position that the queen is moving to to the
positions list
            positions.append(position)
            # if position is different from the queen's current position:
            if (position != domain[col]):
                # call updateConflicts with add= -1 to remove 1 from the
rows, left diagonal, and right diagonal lists
                updateConflicts(col, domain[col], -1)
                # replace the row index in domain[col] with the new row
index to show the queen's new position
                domain[col] = position
                # call updateConflicts with add = 1 to add 1 to the rows,
left diagonal, and right diagonal lists of the queen's new position
                updateConflicts(col, domain[col], 1)
        # if total # of conflicts is 3 (1 in row, 1 in left diagonal, and
1 in right diagonal), then there is only 1 attacking queen so the queen
doesn't need to be moved
        elif numConflicts == 3:
            # solution is found
            return True
        iteration += 1

    # if there are only 2 distinct values in the positions list, then
there is an infinite loop (queen keeps moving back and forth between 2
spots, never reaching a solution)
    if (len(set(positions)) == 2):
        infiniteLoop = True

    # no solution is found after max # of iterations is reached
    return False


# read from file containing # of queens (n)
def readInput():
    numQueensList = []
    with open('input.txt', 'r') as file:
```

```
        # create a list to store all of the values of n that we need to
find a solution for
        for line in file:
            # store each value of n into the list
            numQueensList.append(int(line.rstrip('\n')))
    file.close()
    # clear the output file
    open('output.txt', 'w').close()
    # return the list
    return numQueensList


# create an output file to insert the solutions into
def writeOutput():
    # add 1 to each index value in the solution list to improve
readability
    # ex. turns index 0 into index 1 to represent the 1st square from the
top of the chess domain in a given column
    for i in range(len(domain)):
        domain[i] += 1

    solution = str(domain)
    with open('output.txt', 'a', 64) as file:
        # write the list containing the solution to output file
        file.write(solution + "\n\n")
    file.close()


def printDomain(domain, numQueens):
    # print(domain)
    row = [['-' for x in range(0, numQueens)] for y in range(0,
numQueens)]
    for i in range(numQueens):
        num = domain[i] - 1
        row[i][num] = 'Q'

    for i in row:
        print(*i)
```

```
    return(row)


def plot_solution(num_queens):
    """
        Given a solution, plot it and save the result to disk.
    """
    fig = plt.figure()
    ax = fig.add_subplot(111, aspect='equal')
    ax.set_xlim((0, num_queens))
    ax.set_ylim((0, num_queens))

    count = 0
    for queen in solution:
        ax.add_patch(patches.Rectangle((queen, count), 1, 1))
        count += 1
    fig.savefig(''.join(num_queens) +
                '.png', dpi=150, bbox_inches='tight')
    plt.close(fig)
# this function is called to initiate the program


def main():
    # use global keyword so that we can modify each variable inside the
function
    global numQueens
    global domain
    global numRow
    global numRightDiag
    global numLeftDiag
    global infiniteLoop

    # call the function that reads the input file and returns the list of
#s of queens (n)
    numQueensList = readInput()
```

```
    # iterate through the list of #s of queens and return a solution for
each
    for num in numQueensList:
        # set numQueens as equal to the # of queens read from input file
        numQueens = num
        # initially set infiniteLoop as false
        infiniteLoop = False
        # start the timer to count how long it takes to find a solution
        startTime = time.time()
        # whether or not a solution has been found
        solved = False
        print("# of Queens (n): " + str(num))

        # while not solved, will continue to call solve to try and find a
solution
        while (solved == False):
            # returns solved if a solution is found, false otherwise
            solved = solve()

        # writes the solution to the output file
        writeOutput()

        # figures out the time it took to find a solution and outputs it
in the console
        endTime = time.time()
        totalTime = endTime - startTime
        timeStr = str(trunc(totalTime * 100) / 100)
        print("Solution found in " + timeStr + " seconds\n")
        printDomain(domain, numQueens)
#          for i in range(len(domain)):
#              domain[i] += 1
#          print(domain)


if __name__ == '__main__':
    main()
```

**<u>Output</u>**

**<u>n = 10:</u>**

# of Queens (n): 10

Solution found in 0.01 seconds

[6, 4, 2, 7, 9, 3, 1, 8, 10, 5]

- - - - - Q - - - -

- - - Q - - - - - -

- Q - - - - - - - -

- - - - - - Q - - -

- - - - - - - - Q -

- - Q - - - - - - -

Q - - - - - - - - -

- - - - - - - Q - -

- - - - - - - - - Q

- - - - Q - - - - -

**<u>n = 100:</u>**

# of Queens (n): 100

Solution found in 0.01 seconds

[98, 100, 4, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 82, 29, 31, 33, 35, 37, 39, 41, 43, 45, 47, 49, 97, 53, 55, 57, 59, 61, 63, 65, 67, 69, 71, 73, 75, 77, 79, 81, 83, 85, 87, 89, 91, 93, 95, 2, 99, 3, 6, 8, 1, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 5, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80, 27, 84, 86, 88, 90, 51, 94, 96, 10, 42, 92]

```
-------------------------------------------------------------------------------------------------Q--
---------------------------------------------------------------------------------------------------Q
---Q------------------------------------------------------------------------------------------------
------Q---------------------------------------------------------------------------------------------
--------Q-------------------------------------------------------------------------------------------
----------Q-----------------------------------------------------------------------------------------
------------Q---------------------------------------------------------------------------------------
--------------Q-------------------------------------------------------------------------------------
----------------Q-----------------------------------------------------------------------------------
------------------Q---------------------------------------------------------------------------------
--------------------Q-------------------------------------------------------------------------------
----------------------Q-----------------------------------------------------------------------------
------------------------Q---------------------------------------------------------------------------
---------------------------------------------------------------------------------Q------------------
----------------------------Q-----------------------------------------------------------------------
------------------------------Q---------------------------------------------------------------------
--------------------------------Q-------------------------------------------------------------------
----------------------------------Q-----------------------------------------------------------------
------------------------------------Q---------------------------------------------------------------
--------------------------------------Q-------------------------------------------------------------
----------------------------------------Q-----------------------------------------------------------
------------------------------------------Q---------------------------------------------------------
--------------------------------------------Q-------------------------------------------------------
----------------------------------------------Q-----------------------------------------------------
------------------------------------------------Q---------------------------------------------------
------------------------------------------------------------------------------------------------Q---
----------------------------------------------------Q-----------------------------------------------
------------------------------------------------------Q---------------------------------------------
--------------------------------------------------------Q-------------------------------------------
----------------------------------------------------------Q-----------------------------------------
------------------------------------------------------------Q---------------------------------------
--------------------------------------------------------------Q-------------------------------------
----------------------------------------------------------------Q-----------------------------------
------------------------------------------------------------------Q---------------------------------
--------------------------------------------------------------------Q-------------------------------
```

```
----------------------------------------------------------------------Q-----------------------------
------------------------------------------------------------------------Q---------------------------
--------------------------------------------------------------------------Q-------------------------
----------------------------------------------------------------------------Q-----------------------
------------------------------------------------------------------------------Q---------------------
--------------------------------------------------------------------------------Q-------------------
----------------------------------------------------------------------------------Q-----------------
------------------------------------------------------------------------------------Q---------------
--------------------------------------------------------------------------------------Q-------------
----------------------------------------------------------------------------------------Q-----------
------------------------------------------------------------------------------------------Q---------
--------------------------------------------------------------------------------------------Q-------
----------------------------------------------------------------------------------------------Q-----
-Q--------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------Q-
--Q-------------------------------------------------------------------------------------------------
-----Q----------------------------------------------------------------------------------------------
-------Q--------------------------------------------------------------------------------------------
Q---------------------------------------------------------------------------------------------------
-----------Q----------------------------------------------------------------------------------------
-------------Q--------------------------------------------------------------------------------------
---------------Q------------------------------------------------------------------------------------
-----------------Q----------------------------------------------------------------------------------
-------------------Q--------------------------------------------------------------------------------
---------------------Q------------------------------------------------------------------------------
-----------------------Q----------------------------------------------------------------------------
-------------------------Q--------------------------------------------------------------------------
---------------------------Q------------------------------------------------------------------------
-----------------------------Q----------------------------------------------------------------------
-------------------------------Q--------------------------------------------------------------------
---------------------------------Q------------------------------------------------------------------
-----------------------------------Q----------------------------------------------------------------
-------------------------------------Q--------------------------------------------------------------
---------------------------------------Q------------------------------------------------------------
----Q-----------------------------------------------------------------------------------------------
-------------------------------------------Q--------------------------------------------------------
---------------------------------------------Q------------------------------------------------------
-----------------------------------------------Q----------------------------------------------------
-------------------------------------------------Q--------------------------------------------------
---------------------------------------------------Q------------------------------------------------
-----------------------------------------------------Q----------------------------------------------
-------------------------------------------------------Q--------------------------------------------
---------------------------------------------------------Q------------------------------------------
-----------------------------------------------------------Q----------------------------------------
-------------------------------------------------------------Q--------------------------------------
```

```
---------------------------------------------------------------Q------------------------------------
-----------------------------------------------------------------Q----------------------------------
-------------------------------------------------------------------Q--------------------------------
---------------------------------------------------------------------Q------------------------------
-----------------------------------------------------------------------Q----------------------------
-------------------------------------------------------------------------Q--------------------------
---------------------------------------------------------------------------Q------------------------
-----------------------------------------------------------------------------Q----------------------
-------------------------------------------------------------------------------Q--------------------
--------------------------Q-------------------------------------------------------------------------
-----------------------------------------------------------------------------------Q----------------
-------------------------------------------------------------------------------------Q--------------
---------------------------------------------------------------------------------------Q------------
-----------------------------------------------------------------------------------------Q----------
--------------------------------------------------Q-------------------------------------------------
---------------------------------------------------------------------------------------------Q------
-----------------------------------------------------------------------------------------------Q----
---------Q------------------------------------------------------------------------------------------
-----------------------------------------Q----------------------------------------------------------
-------------------------------------------------------------------------------------------Q--------
```

**n = 1000:**

# of Queens (n): 1000

Solution found in 0.52 seconds

[1, 776, 5, 7, 742, 11, 13, 15, 17, 19, 21, 984, 25, 27, 926, 31, 317, 35, 143, 39, 586, 879, 45, 47, 49, 51, 53, 55, 57, 59, 61, 63, 693, 67, 69, 952, 978, 525, 77, 79, 81, 83, 85, 87, 564, 91, 93, 95, 97, 99, 883, 788, 382, 972, 494, 868, 113, 115, 117, 114, 121, 785, 125, 127, 129, 131, 133, 23, 137, 139, 141, 850, 576, 968, 149, 151, 153, 855, 157, 159, 161, 163, 165, 167, 169, 171, 173, 175, 177, 828, 181, 183, 185, 187, 189, 191, 193, 195, 197, 199, 201, 203, 205, 207, 209, 211, 213, 215, 217, 219, 976, 223, 225, 227, 962, 231, 233, 235, 237, 775, 241, 243, 245, 247, 244, 736, 119, 255, 257, 259, 261, 263, 265, 267, 269, 271, 892, 275, 277, 802, 281, 283, 179, 287, 975, 291, 293, 295, 297, 299, 301, 303, 305, 307, 309, 311, 313, 315, 772, 674, 321, 786, 325, 946, 329, 331, 333, 335, 337, 339, 341, 930, 345, 347, 349, 351, 353, 355, 858, 359, 361, 363, 852, 367, 369, 371, 373, 664, 994, 379, 381, 383, 385, 387, 389, 391, 393, 395, 397, 399, 756, 403, 988, 407, 722, 734, 413, 415, 417, 419, 872, 423, 425, 427, 429, 431, 433, 435, 437, 439, 441, 922, 445, 447, 449, 451, 453, 455, 457, 844, 461, 463, 465, 467, 469, 471, 959, 475, 98, 479, 481, 483, 485, 487, 992, 491, 493, 495, 497, 499, 501, 503, 505, 507, 75, 511, 801, 515, 517, 519, 521, 523, 52, 527, 529, 531, 533, 535, 537, 33, 541, 543, 545, 547, 549, 551, 553, 555, 557, 559, 561, 563, 565, 567, 569, 571, 573, 575, 577, 579, 581, 583, 860, 587, 589, 591, 593, 595, 597, 599, 411, 914, 605, 607, 609, 611, 613, 615, 617, 619, 621, 623, 625, 832, 629, 631, 633, 635, 637, 639, 641, 643, 645, 647, 998, 651, 36, 655, 71, 210, 661, 663, 665, 667, 669, 109, 673, 854, 677, 679, 681, 683, 685, 687, 689, 990, 37, 695, 697, 699, 701, 703, 896, 707, 709, 711, 713, 715, 717, 719, 721, 723, 725, 727, 729, 731, 733, 735, 737, 739, 954, 743, 745, 747, 749, 751, 753, 755, 757, 759, 761, 763, 765, 767, 769, 771, 773, 111, 777, 779, 781, 783, 979, 787, 789, 791, 793, 795, 958, 799, 89, 904, 805, 807, 809, 811, 813, 815, 817, 819, 821, 823, 825, 827, 829, 831, 833, 835, 837, 839, 841, 843, 845, 847, 849, 851, 853, 953, 857, 859, 861, 863, 865, 867, 869, 871, 873, 875, 877, 428, 881, 96, 885, 887, 889, 891, 893, 895, 897, 899, 901, 903, 905, 907, 909, 911, 913, 915, 917, 919, 921, 923, 925, 927, 929, 931, 933, 935, 937, 939, 941, 943, 945, 947, 949, 951, 155, 955, 957, 420, 961, 963, 965, 967, 969, 971, 973, 116, 977, 248, 981, 983, 985, 987, 989, 991, 993, 995, 997, 999, 164, 323, 352, 10, 123, 14, 16, 18, 20, 274, 192, 73, 28, 30, 32, 34, 206, 38, 40, 42, 172, 46, 285, 50, 136, 84, 56, 58, 60, 456, 64, 66, 68, 166, 132, 196, 76, 78, 319, 82, 86, 22, 504, 180, 284, 342, 404, 100, 102, 104, 106, 108, 249, 142, 496, 308, 578, 124, 126, 128, 130, 740, 134, 956, 444, 434, 432, 306, 146, 148, 150, 436, 154, 156, 377, 438, 414, 902, 252, 174, 176, 178, 182, 184, 186, 188, 190, 366, 194, 442, 198, 200, 446, 204, 513, 208, 448, 26, 212, 214, 216, 218, 220, 450, 266, 8, 498, 960, 228, 797, 103, 362, 65, 454, 426, 330, 364, 354, 964, 365, 273, 375, 110, 768, 357, 410, 343, 221, 460, 54, 704, 386, 280, 462, 970, 846, 492, 838, 798, 516, 473, 48, 378, 888, 489, 105, 239, 974, 468, 440, 812, 572, 229, 409, 314, 374, 502, 320, 472, 324, 300, 236, 112, 474, 458, 289, 392, 653, 94, 279, 70, 1000, 986, 478, 12, 422, 898, 916, 506, 406, 224, 459, 370, 222, 482, 230, 336, 396, 92, 484, 862, 490, 606, 242, 118, 508, 246, 466, 604, 90, 488, 4, 470, 906, 510, 512, 514, 418, 518, 526, 528, 530, 254, 538, 540, 542, 592, 705, 88, 550, 554, 556, 558, 560, 430, 568, 570, 598, 600, 602, 608, 610, 614, 616, 618, 620, 622, 624, 626, 628, 630, 632, 476, 636, 638, 640, 642, 534, 671, 646, 648, 650, 120, 327, 258, 268, 270, 262, 675, 627, 966, 562, 662, 41, 666, 251, 524, 672, 135, 676, 322, 272, 107, 544, 686, 980, 402, 692, 286, 276, 140, 710, 702, 920, 290, 716, 718, 720, 712, 714, 724, 726, 728, 730, 732, 452, 464, 388, 738, 982, 294, 744, 746, 748, 750, 752, 754, 582, 758, 760, 762, 764, 3, 152, 398, 770, 44, 774, 160, 778, 780, 782, 408, 298, 480, 2, 29, 790, 792, 794, 796, 158, 43, 800, 566, 908, 804, 806, 808, 810, 170, 122, 304, 848, 318, 580, 6, 596, 144, 836, 421, 226, 62, 74, 202, 477, 416, 145, 312, 659, 24, 316, 500, 878, 594, 509, 741, 585, 880, 384, 334, 356, 234, 784, 238, 240, 288, 520, 346, 168, 256, 250, 884, 882, 360, 264, 870, 866, 260, 601, 886, 652, 80, 282, 552, 834, 826, 368, 691, 522, 900, 278, 766, 548, 292, 405, 296, 649, 890, 302, 680, 310, 830, 822, 147, 380, 326, 328, 332, 657, 9, 338, 340, 486, 344, 584, 708, 348, 350, 912, 910, 928, 412, 72, 932, 232, 894, 400, 162, 574, 856, 424, 996, 376, 372, 936, 940, 942, 590, 401, 390, 818, 874, 394, 539, 803, 358, 934, 588, 944, 864, 816, 603, 876, 443, 612, 938, 532, 634, 918, 840, 820, 644, 948, 814, 536, 654, 950, 658, 660, 138, 656, 824, 668, 670, 678, 253, 682, 684, 688, 690, 842, 694, 696, 698, 546, 924, 101, 706, 700]

**n = 10,000:**

https://github.com/mpa-mxiang/n-queens/blob/main/output2.txt

**n = 100,000:**

https://raw.githubusercontent.com/mpa-mxiang/n-queens/main/output.txt