Solving the N-Queen's Problem Using Heuristics

CISC 352

Professor Sidney Givigi
Group 10

Christopher Pop, 20071937
Chengyuan Sha, 20013204
Shengan Li, 20048177
Noah Jacobs, 20076018

Abstract

In this report, we solve the renowned “N-Queen’s” problem, which was proposed by Max Bezzel in 1948. (Engelhardt, 2017) Our algorithm solves this problem in an efficient manner through the use of heuristics.

Purpose

The purpose of this program is to solve the n-Queens problem using the QS4 algorithm and to minimize conflict heuristics. The solution for this problem requires that there are no queens with the ability to attack each other on an n x n chess board. The goal is to achieve a constant or linear time complexity so that the solution for a 1,000,001-queen puzzle can be found in under 3 minutes.

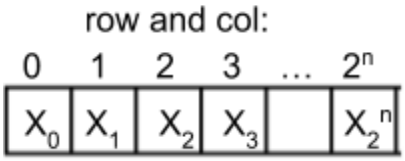
Introduction

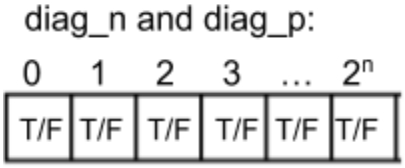
The N-Queen’s problem consists of a checkerboard with n number of queens placed on arbitrary spots. The goal of the problem is to place all of the queens in a position in which no queen poses a threat to another, a threat being visibility to another queen vertically, horizontally, or diagonally. The algorithm we created to solve this problem works for any number of queens, taking a text file with the dedicated number of queens as an input. A list with the indices representing columns and each value representing rows is then printed to an additional text file. The problem that our code solves is an example of the *Constraint Satisfaction Problem*, which entails search problems limited by constraints. The best way to solve the N-Queen’s problem is through a backtracking search with the use of min-conflict heuristics. However, a regular backtracking problem cannot solve the N-Queen’s problem for a large input. (Sosic and Gu, 1991) To do this, a polynomial time local search algorithm that uses a gradient-based conflict minimization heuristic must be used. To distinguish this algorithm from the backtracking algorithm, we will call it the *Queen Search 1 (QS1)* algorithm. (Sosic and Gu, 1991)

From there, we derive a linear time algorithm QS4 algorithm that runs much faster than QS1. (Sosic and Gu, 1991) Not only is it faster than QS1, but it also uses the same conflict minimization heuristic. (Sosic and Gu, 1991) In the QS4 algorithm, an initial random permutation is generated such that the number of collisions among queens is minimized. (Sosic and Gu, 1991) From there, this algorithm begins by placing queens on successive rows. The position for the next queen is then randomly generated from columns that are not occupied up until a conflict free place is found for this queen. After a certain number of queens have been placed in a conflict free manner, the remaining queens are placed randomly on free columns regardless of any conflicts existing on diagonal lines. The process of generating the initial permutation does not require backtracking.

Data Structure

Whilst it is easy to imagine a chess board as being like a 2D array, this is actually a very impractical way of structuring the storage for this algorithm. As n increases, the storage required to support an n x n array becomes unrealistic and inefficient. The algorithm instead generates two 1 dimensional arrays of unsigned integers (row and col) to track the current state and conflicts. In this system, the index of the array represents the row (0, 1, ..., 2^n) while the value represents the column (X_0, ...,X_2^n) as seen in the diagram on the right. This is possible because we are just trying to eliminate conflicts, and can therefore use this method to ensure there will not be two queens in the same row. The algorithm also uses two more 1D arrays of type boolean (diag_p and diag_n). These are to track the diagonal lines for the positive and negative slopes.





Technique

Function Descriptions

main()

- Driving force behind the algorithm. Handles input/output as well as conflicts to sort the queens. In addition, keeps track of runtime.

write_output(solution, fname)

- Print the sorted queens to a text file as an array, with each index representing a column and each value representing a row.

Parameters

- solution
    - The array of sorted queens with their corresponding positions.
- fname
    - The text file in which the output will be printed (nqueens_out.txt).

read_txt(fname)

- Opens the input text file (nqueens.txt) which holds the number of queens that the user passes into the algorithm.

Parameters

- fname
    - The text file from which the input will be derived (nqueens.txt).

visualize(sz, queen_loc)

- Prints a matrix representation of the checkerboard to the console, with all queens solved for no conflicts. 0s represent vacant positions, 1s represent queens.

Parameters

- sz
    - Represents each individual queen, used to display the dimensions of the checkerboard.
- queen_loc
    - Represents the location on the checkerboard for each queen.

min_conflict(n, m)

- Rearranges the queens so that no conflicts with respect to the N-Queen's problem are present on the checkerboard.

Parameters

- n
    - Represents the total number of queens on the checkerboard.
- m
    - Represents the queens on the checkerboard without conflicts.

update_values(row, col, queen, i, j, n)

- Works in conjunction with *min_conflict(n,m)* in order to update the values of each queen as they are being rearranged to satisfy the constraints of the N-Queen's problem.

Parameters

- row
    - Represents the row of the queen being manipulated.
- col
    - Represents the column of the queen being manipulated.
- queen
    - Represents the queen being manipulated.
- i
    - First index position used to rearrange the queens.
- j
    - Second index position used to rearrange the queens.
- n
    - Represents the total number of queens on the checkerboard.

delta(i, j, row, col, queen, n)

- Helper function used by *min_conflict(n,m)* in order to calculate the current number of conflicts on the checkerboard.

Parameters

- row
    - Represents the row of the queen being manipulated.
- col
    - Represents the column of the queen being manipulated.
- queen
    - Represents the queen being manipulated.
- i
    - First index position used in calculating the current number of conflicts.
- j
    - Second index position used in calculating the current number of conflicts.
- n
    - Represents the total number of queens on the checkerboard.

with_conflict(num, const_list)

- Once the first queen has been placed and is shown to be conflict free, the rest of the queens are then randomly laid out on the checkerboard. The number of the rest of the queens that could have a conflict is returned.

Parameters

- num
    - Represents the current number of queens.
- const_list
    - Passes in a list of the number of queens that could have a conflict for a given total number of queens. Please refer to the figure below for a comparison of the two variables.

Number of Queens n	10	10^2	10^3	10^4	10^5	10^6	2×10^6	3×10^6
Queens with Conflict c	8	30	50	50	80	100	100	100

Fig 1. Number of Queens vs. Queens with Conflict (Sosic and Gu, 1991)

conflict_sum(n, row, col)

- Sums the number of conflicts in the checkerboard at a given time.

Parameters

- n
    - Represents the total number of queens.
- row
    - Row of the current position being analyzed for conflicts.
- col
    - Column of the current position being analyzed for conflicts.

generate_conflict_free(queen, row, col, diag_p, diag_n, n, m)

- Once the queens are laid out on the checkerboard, this function attempts to move the queens out of any diagonal conflicts that exist.

Parameters

- queen
    - Represents the queen being manipulated.
- row
    - Represents the row of the queen being manipulated.
- col
    - Represents the column of the queen being manipulated.
- diag_p
    - Represents the positive slope diagonal.
- diag_n
    - Represents the negative slope diagonal.
- n
    - Represents the total number of queens on the checkerboard.
- m
    - Represents the queens without conflicts.

generate_regardless_conflicts(queen, row, col, n, m)

- Generates the checkerboard with the queens laid out. No attempts to resolve any conflicts are made in this function.

Parameters

- queen
    - Represents the queen being manipulated.
- row
    - Represents the row of the queen being manipulated.
- col
    - Represents the column of the queen being manipulated.
- n
    - Represents the total number of queens on the checkerboard.
- m
    - Represents the queens without conflicts.

initialization(n, m)

- Creates the queens, checkerboard rows/columns, and positive/negative diagonals to represent diagonal conflict between queens. These values are then passed to its helper functions (*generate_regardless_conflicts* and *generate_conflict_free*) in order to generate the checkerboard without diagonal conflicts.

Parameters

- n
    - Represents the total number of queens on the checkerboard.
- m
    - Represents the queens on the checkerboard without conflicts.

Pseudocode

def initialization(n, m):
  Create the queens
  Create the checkerboard
  Create positive and negative slope diagonals to be used when checking for diagonal conflicts

  If a diagonal conflicts exists, position the queens to avoid them
  Place the queens back on the checkerboard

  return queen, row, column

def generate_regardless_conflicts(queen, row, col, n, m):
  While (the number of conflicts) < (the number of queens):
    Move the queen into a different vacant position
    Decrement the count of the number of queens without conflicts
    Loop until the count of the number of queens without conflicts is 0

  return queen, row, column

def generate_conflict_free(queen, row, col, diag_p, diag_n, n, m):
  While (the number of conflicts) < (the number of queens without conflicts)::
    Loop:
      If no queens have a diagonal intersection:
        Stop

    Move the queen into a different vacant position
    Decrement the count of the number of queens without conflicts

  return queen, row, column, positive diagonal, negative diagonal

def conflict_sum(n, row, col):

For i in range(the whole board):

If there is more than 1 queen in a row:

Add to conflict sum

If there is more than 1 queen in a column:

Add to conflict sum

return conflict sum

def with_conflict(num, const_list):

If there are <= 10 queens:

If there are > 8 queens:

return const_list[0]

else:

return the total number of queens

Else if there are < 100 queens:

return return the total number of queens

Else if there are < 1000 queens:

return const_list[1]

Else if there are < 10000 queens:

return const_list[2]

Else if there are < 100000 queens:

return const_list[3]

Else:

return const_list[4]

def delta(i, j, row, col, queen, n):

Traverse through the checkerboard and count how many queens have conflicts

return the number of queens with conflicts

```
def update_values(row, col, queen, i, j, n):
    Move one queen
    Move another queen
    return row, column, queen

def min_conflict(n, m):
    Create the queens and the checkerboard
    Add up all of the existing conflicts
    Start the timer to keep track of runtime

    While the conflict sum is > 0
        For i in range of (queens without conflicts) up until (the total number of queens):
            If a queen exists at the position:
                If more than 4 seconds is spent on the previous step:
                    Add to the existing conflict count
                    Restart
                Continue

            Else:
                For j in range(total number of queens):
                    if i does not equal j:
                        Find the number of queens with conflicts

                        If the number of queens with conflicts is returned as < 0:
                            Break

                    If the number of queens with conflicts is returned as < 0:
                        Break

            If the number of queens with conflicts is returned as < 0:
                Move the queen to another vacant position
                Add to the existing conflict count

            Else:
                Restart
                Add to the existing conflict count

    print("Number of reset: " number of times the algorithm had to reset)
    return queen, step
```

def read_txt(fname):
  Read the contents of the input text file

  If the value(s) in the input are > 3 and <=10 000 000:
    return the value

  Else:
    raise ValueError("Each line of input should consist of a single integer value, n, where n > 3 and n <= 10,000,000")

def write_output(solution, fname):
  print("Writing to txt...")
  Write the array of sorted queens to the output text file
  print("Writing complete.")

def main():
  Delete the output text file if one exists prior to completion
  Number of queens = value in input text file

  For the number of queens:
    Start the timer
    print(“Number of queens:” Number of queens)
    Remove any conflicts
    End the timer
    print("Time elapsed (in seconds):” Total runtime)
    print("Steps in calculation:” Total number of steps)
    Write the output to the output text file

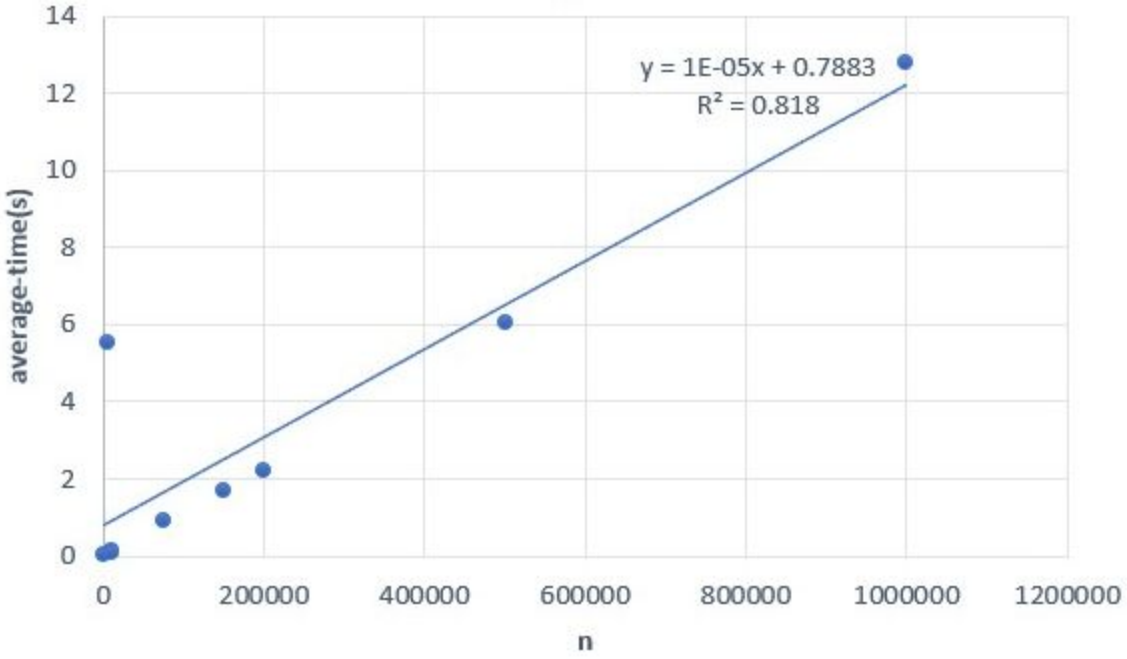
Call *main()*

Results

The program is tested on small, medium and large n's. Three sample n's are defined in each of the small, medium and large samples test. The table below was generated by running the program on each sample n ten times, followed by computing the average time it took to solve the n-Queens at each specified n.

small					**medium**					**large**				
n	**time/s**				**n**	**time/s**				**n**	**time/s**			
	t1	**t2**	**t3**	**t_{avg}**		**t1**	**t2**	**t3**	**t_{avg}**		**t1**	**t2**	**t3**	**t_{avg}**
10	0.0399	0.03991	0.00998	0.0269	**10000**	0.09574	0.1107	0.09478	0.100	**200000**	2.305	2.098	2.257	2.22
5000	4.222	4.1330	8.1731	5.51	**75000**	0.89966	0.924	0.9664	0.929	**500000**	6.773	5.916	5.467	6.052
9999	0.0957	0.1287	0.0937	0.1060	**150000**	1.61667	1.7800	1.5917	1.663	**1000001**	14.08	12.30	12.056	12.81

The graph below was produced by comparing the time data from the table above with their respective input size n. As can be seen from the graph, our program achieved a linear time complexity. The linear time complexity that we have achieved with this algorithm is superior to the exponential time complexity that would have been achieved by most algorithms. This is due to the implementation of min-conflict heuristics.



Conclusion

In conclusion, we were able to achieve the goal mentioned in the purpose section of this paper. Looking at the data recorded in the results section, it can be seen that the line of best-fit plotted for the comparison between the sample size n and their average times represents a linear time complexity.

Works Cited

1. Matthias, Engelhardt. “The n Queens Problem.” *NQueens*, 10 Nov. 2017, www.nqueens.de/.
2. Sosic Rok, and Jun Gu. “3,000,000 Queens in Less than One Minute.” ACM Digital Library, Feb. 1991, dl.acm.org/.