Queen's University

Solving N-Queen Problems Using Iterative Repair

CISC 352

Professor Sidney Givigi

Friday February 15, 2019

Group 13

Echo Gao [10149157]

Tianyu Hua [20142378]

Shirley Xiao [10145624]

Feiyi Yang [10151502]

Wanyu Zhang [10141510]

Purpose

The purpose of this program is to solve the n-Queens problem using iterative repair algorithm and the minimum conflicts heuristics. A solution for this problem requires that for *n*, where *n* is the number of queens on a *n* x *n* board, there should be no queens sharing the same row, column, or diagonal. The goal is to achieve a constant or linear time complexity so that the solution for a 100,000-queen puzzle can be found in under 5 minutes.

Introduction

N-Queens problem is an example of constraint satisfaction problems. It was discussed in the lectures that the best way to solve n-queens is through iterative repairs using the min-conflicts heuristics. We started out using the min-conflicts heuristic described in *The Min-Conflicts Heuristic: Experimental and Theoretical Results (Minton et al., 1991)*. A randomly generated initial state was repaired by moving an attacking queen to the position on the row or column with the least conflicts with the other present queens. However, we were not able to achieve the result proposed in the assignment description (solving one million queens in 55 steps on average). In our initial attempt, each position on the row for the concerning queen is looped through to count for the total conflicts by checking through all columns of the game board. The queen is then moved to the position with the least conflicts counted, and the next concerning queen is chosen randomly from the queens conflicting with the queen we just moved. The resulting steps needed to find solution grows linearly as *n* increases, and the time grows exponentially.

A more efficient algorithm is found in *Fast Search Algorithms for the N-Queens Problem (Sosic and Gu, 1991)*. In the article, the authors described two probabilistic local search algorithms based on a gradient-based heuristic – QS2 and QS3. The same repair method was used in the two algorithms, however, in QS3 the initial states were generated using the greedy method. After experimentations, we found QS3 to be superior to QS2 – a 40% increase in speed was observed on our computer. Therefore, QS3 is the algorithm we decided to use for this assignment. The choice of data structures used in this algorithm will be discussed in the Data Structure section, and the algorithm will be explained in detail in the Technique section. The performance of our implementation of QS3 is summarized and analyzed in the Results section.

Data Structure

1D arrays of unsigned int are generated to contain state and the conflicts. While the chess board is better represented as a 2D array, it is impractical to store the board when the *n* is large. For an n-Queens problem with *n* = 1e+6, a 2D array will need to store 1e+12 items, which is 125 gigabytes of data if each item is represented by a bit. Therefore, 1D array is used to describe the 2D matrix of the game board, where the index represents the row and the value represents the column. Since our goal is to eliminate conflicts, we can be sure there will not be two queens on the same row — so the 1D array can sufficiently represent the game board. Unsigned int32 is used to describe the column index of the queen in the state of the game board (db) and the number of conflicts counted in the positive and negative diagonal (dp and dn) arrays (the arrays will be explained in the next section). As for the data type, we chose to use unsigned int because we don't need to store negative numbers in the array. By using unsigned int, we can use this program to solve for n-Queens problems with double the size of *n* in comparison to if we use signed int. The diagram below (left) shows an illustration of the db, dp and dn arrays for the game state [1, 0, 2, 3]. The dp array contains the total number of queens in its positive diagonal (as indicated by the same shade in the diagram) and the dn array contains the total in its negative diagonal (as indicated by the same text colour, diagonals are italicized and bolded alternately for better illustration). The right side shows a change to game state [1, 2, 0, 3]. In the case that the two queens are not in the same diagonal, it can be seen that 8 values in the dp and dn arrays are affected by the swap of position in row 1 and 2. If the two queens that are supposed to be swapped are in the same diagonal, the swap will not affect all 8 possible diagonals. The method we are using to check conflicts will then produce duplicates, therefore the conflicts are stored in a set instead of array.

dp	0	2	0	0	1	0	1
0	1	0	0				
1	0	0	0				
0	0	1	0				
0	0	0	1				
dn	0	0	1	2	1	0	0

dp	0	1	1	1	0	0	1
0	1	0	0				
0	0	1	0				
1	0	0	0				
0	0	0	1				
dn	0	1	0	1	2	0	0

Technique

The initial state is generated via greedy method and stored in the 1D array is manipulated by looping through a set of different operations to determine whether or not to switch positions. In this algorithm, when the queen in column i moves to column j, the queen in column j will be swapped to column i. This is to make sure that there will never be two queens in the same column, so when checking conflicts, only diagonals need to be checked. The decision of whether or not to switch is made by checking the number of conflicts of the concerning queen has with the other queens on the board before and after the switch. If more conflicts are generated after the switch than before the switch, the switch will not be performed. What's more, two constants: C1 and C2 are defined to increase efficiency. It is expensive to make changes to big arrays. In order to decrease the cost of changing arrays every iteration, we only update the *attack* array when the decrease in the total number of collisions reach a certain limit, which is defined by *C1 * collisions*. It should be noted that the iterative repair method cannot guarantee to find a result. To resolve that issue, we set a threshold for when to give up and start with a new board, which is defined by *C2 * n*.

In the constructor of class board, __init__

bd - array that will contain the state of the queens

dp - set array that contains the number of collisions in the positive diagonals

dn - set array that contains the number of collisions in the negative diagonals

attack - array that contains the row index of the queens that conflict with other queens

collisions - the number of possible diagonal conflicts

limit - value defined to increase efficiency and reduce cost.

- Set *C1*: It is used to adjust *limit* value, *limit = C1 * collisions*
- The *attack* array is only updated when *collisions* decrease to smaller than *limit.*

loopcount - the while loop counter for the repair function

- Set *C2*: this is used to regulate the loop count.
- When *loopcount* goes over the value of *C2 * n*, a new permutation starts[1]

threshold - is used to get the position on the board that has the least # of conflicts

greedyinit() - algorithm to minimize steps of the iterative repair

iter_num - the number of step it takes to solve the problem

_old - set of 8 old collisions

_new - set of 8 new collisions

[1] R. Sosic and J. Gu. "Fast Search Algorithms for the N-queens Problem." *IEEE Transactions on Systems, Man, and Cybernetics*21, no. 6 (1991): 8, doi:10.1109/21.135698.

def compute_attack_matrix()

attack = []

For the size of the board, *N*

    If there is a collision in diagonal negative or diagonal positive

        Add to *attack*

Return *attack*

def compute_collisions(self)

Initialize *dp* and *dn* list with 0

For size of the board

    Add up queens in the positive diagonal

    Add up queens in the negative diagonal

total = 0

For the values in *dn* and *dp*

    If the element in *dn* is greater than 1 then subtract the value by 1

        add to *total*

Set *total* to *collisions*

Return *total*

def get8collisions(self, i, j)

Check for collisions in the diagonals of queen at position i and queen at position j

old_collisions = 0

For each value in diagonal positive array and diagonal negative array, *dpset* and *dnset*

    Add up the *old_collisions*,

        if the value is greater than 1,

            Subtract -1 otherwise add 0

Return *old_collisions*

def swap_ok(self, i, j)

Get collisions for old queen states i, j

_preform_swap()

Get collisions for new queen states i, j

_perform_swap()

Return true if the new queen positions collisions are less than the old collisions otherwise false

def _perform_swap(self, i, j)

Swap positions of queeni and queenj in diagonal positive array, *dp*

Swap positions of queeni and queenj in diagonal negative array, *dn*

Swap positions of queeni and queenj in the states of queens, *bd*

def perform_swap(self, i, j)

Call *perform_swap*()

Update the number of *collisions*

def repair()

This function is used to solve the problem by calling the functions above.

While the number of collisions is greater than 0

    Increase iterating number

    Call *randinit*()

    Set *collisions* to value returned by *compute_collisions*()

    Set *limit* calculation

    Initialize *loopcount*

    While *loopcount* is smaller than or equal to *C2 * N*

        For elements in *attack*

            Get random index from size of board

            If it is okay to swap, *swap_ok*(i,j), then

                perform_swap(i,j)

                If the number of collisions is 0

                    Stop

                If the number of *collisions* is smaller than *limit*

                    Update *limit*

                    Compute_attack_matrix()

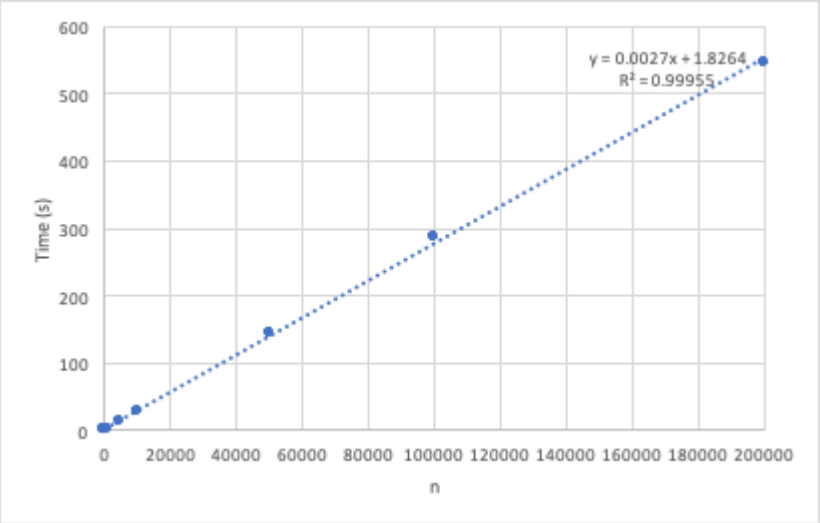
        Update *loopcount*

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 and compute the average time it took to solve for the n-Queens at the specified *n*. The detailed data on the test runs are summarized in the Appendix section. The program is run on an Acer SWIFT3 computer about 2 months old, the processor is quad-core 8th generation Intel i7.

small		medium		large	
n	time	n	time	n	time
10	0.601 s	1000	2.729 s	50000	143.417 s
50	0.841 s	5000	13.597 s	100000	285.345 s
100	1.106 s	10000	29.105 s	200000	546.560 s

The graph below is produced by plotting the time data summarized in the table above against their size n. As can be seen from the graph, our program has achieved a linear time operation. The linear time that we have achieved with this algorithm is superior to the exponential time complexity by implementing the min-conflicts heuristics the way we have discussed in class.



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, we can see the plotted points displaying results of linear time. The time on average to solve for a 100000-queens problem is approximately 4.76 minutes.

References

Minton, S., Philips, A., Johnston, M., & Laird, P. (1991). The min-conflicts heuristic: Experimental and theoretical results.

Sosic, R. and Gu, J. (1991). Fast search algorithms for the n-queens problem. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(6), pp.1572-1576.

Appendix

Table 1. A summary of the time (in seconds) required to solve n-Queens with small *n*

Test #	n = 10	n = 50	n = 100
1	0.92	1.08	0.89
2	0.01	0.42	0.78
3	1.69	0.71	0.23
4	0.28	0.66	0.32
5	0.15	0.92	3
6	0.42	1.91	4.01
7	0.61	1.1	0.88
8	0.14	0.73	0.17
9	1.76	0.74	0.26
10	0.03	0.14	0.52
AVG	0.601	0.841	1.106

Table 2. A summary of the time (in seconds) required to solve n-Queens with medium *n*

Test #	n = 1000	n = 5000	n = 10000
1	3	13.58	27.69
2	2.44	13.19	25.35
3	2.33	14.34	28.95
4	2.57	14.35	29.72
5	2.68	13.71	33.95
6	3.05	14.61	31.28
7	2.43	12.79	31.05
8	2.7	13.18	26.9
9	2.91	12.91	27.94
10	3.18	13.31	28.22
AVG	2.729	13.597	29.105

Table 3. A summary of the time (in seconds) required to solve n-Queens with large *n*

Test #	n = 50000	n = 100000	n = 200000
1	141	281.63	556.58
2	144.76	280.56	549.35
3	141.66	299.8	550.94
4	141.58	290.62	543.02
5	146.93	282.98	550.34
6	146.98	277.47	529.19
7	145.94	279.54	546.05
8	142.7	290.93	566.35
9	144.59	284.53	522.57
10	138.03	285.39	551.21
AVG	143.417	285.345	546.56