CISC 352 Assignment 1: N-Queens

Yuhang Cao 10201134
Hongkai Chen 10179264
Bote Jiang 101296930
Yishan Li 10182827
Ruoran Liu 10191244
Yuqi Wang 10193405

Premise

The N-Queens Problem is a classical question in AI computing, requiring the placement of N queens on a chess board of NxN size such that none of the queens will attack each other.

A common way of solving this problem is to initialize the board with the greedy hill-climbing approach followed by the iterative repair method to eliminate the conflicts. This way, the number of conflicts can be minimal before entering the repair process. The key aspect of this algorithm is to achieve O(N) in both the initialization and the repair phase. However, this proves to be rather difficult as traditionally it requires the use of nested for-loops for column and row traversal.

To bring the complexity down from $O(N^2)$ to O(N), we can make several modifications. The first of which is to look for a particular minimal conflict value per column instead of iterating through all N rows. The second of which is the method that we used. Details are discussed below.

Method and Discussion

<u>Representation</u>

N: the size of the board
R[]: the solution. R[x] = y represents the Queen in the x-th row is at y-th row
p_diag[]: A list of the number of Queens in each positive diagonal (index starts from bottom-right[0] to top-left[2N-1])
n_diag[]: A list of the number of Queens in each negative diagonal ((index from bottom-left[0] to top-right[2N-1])
p_check[]:A list of boolean values for each positive diagonal. p_check[x] == True -> a Queen can be inserted in x-th positive diagonal
n_check=[]: A list of boolean values for each negative diagonal. p_check[x] == True -> a Queen can be inserted in x-th negative diagonal

The above representations will ensure it will only take O(1) instead of O(N) when we are searching if there are some queens in a particular diagonal.

Initial Placement

The method below uses an initialization function that was inspired by a paper published by Jun Gu from the University of Alberta. The number of Queens with possible conflicts, c, can be categorized by the board size: For N>100 c=30, N>1,000 c=50, N>10,000 c=50, N>100,000 c=80, N>1000,000 c=100. With this mind, we can ensure N – c queens without conflict. To do so, we can first go through and assign a queen to each successive row. For the first N-c rows, we can afford to randomly generate a column value such that there are no conflicts. If the generated position for that queen results in a conflict, we generate another column value for that row. We do so until that queen is placed in a conflict free column. This method will progressively take more generations as there is lowering possibility of getting a conflict free position on the first couple of attempts. At first, the generation of the column value per row would be linear time and will gradually lean towards O(N). However, based on the Gu's paper, we can afford this up until the N-c^{th} row.

After the N–c^{th} row, we can randomly assign column positions to the remaining c rows, disregarding any potential conflicts.

Below is how we have implemented the idea above:

1. For i in (0, N-1), we initialize all the queens by giving i-th queen with its coordinate (i,i)

   This ensures no conflict would exist in each row and column.

2. For j in (0, N-c-1), we randomly generate a value p between 0 and N-1. Assign p-th queen from Step 1 to j-th row if it will not lead to conflicts. Otherwise, re-generate the p. We do so until that we find a conflict free square for this queen.

3. For k in (N-c-1, N-1), we can randomly assign queens to the remaining c rows without checking if it will lead to a diagonal conflict.

Repair

The repair process is a modified version of iterative repair. Conventionally, we randomly choose a queen moving it to the lowest conflict position. However, knowing that only the last c queens has conflicts and also that the conflicts are only diagonal we can limit the number of queens we look at and how the queens can be repaired. We choose two queens for comparison, one queen in c and one completely at random. If swapping of the two queens reduces the number of diagonal conflicts, we perform the swap. If not, we choose another pair to try and swap. If we reach a local maximum where no swap can result in a lower conflict and the number of

conflicts > 0, we restart the initialization process. We reach a solution when there are no more conflicts.

In the traditional method, moving within a column might decrease the conflict for one queen but might result additional conflicts elsewhere. Compared with the traditional iterative repair method, the swap method allows the algorithm to address the overall conflict number rather than for one particular column.

For our implementation:

1. Keep track of total conflict

   Algorithm will stop if total conflict is 0.

2. Go through each of the queens in rows that were in c and checks if each diagonal direction has no conflicts.

3. If no diagonal conflicts, we skip that queen.

   No need to repair queen.

4. Else, we will try swapping the column position between the two queens to see if the change is of diagonal contacts is positive or negative (using the function diff_conflict()).

   a. If positive, we swap
   b. If negative we try another queen.

5. Swap R[i] , R[j]= R[j] , R[i]

   Swaps the column positions of the two queens.

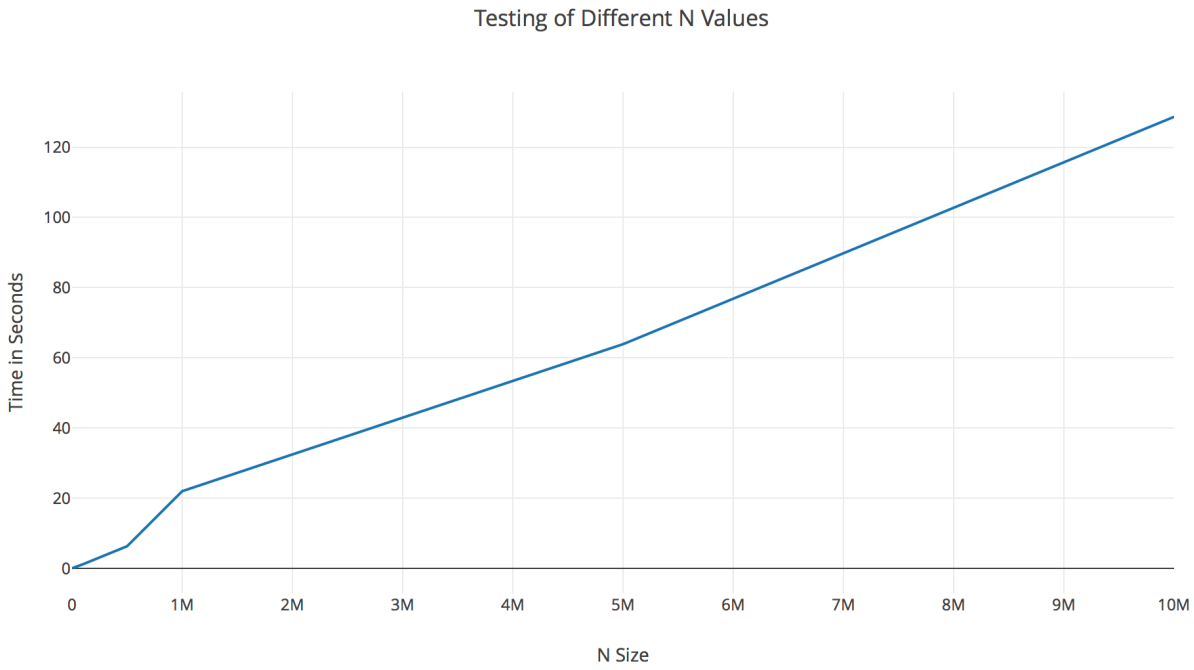
6. Update and new diagonals and the new total conflict number.

7. If total number of diagonal conflicts >=0, we reinitialize the board.

Results

For initial testing, we used 11 different n values ranging from the basic n=8 to a large n=10,000,000. The times were recorded below.

N	Time (Sec)
8	0.0030
100	0.0128
1000	0.0196
5000	0.0614
10000	0.2131
50000	0.5797
100000	1.1505
500000	6.2874
1000000	21.9713
5000000	63.8679
10000000	128.6718



As for the provided small, medium and large N data sets, the total times are below:

Small Sample Size, 20*(Values from 4-900): 7.817 sec
Medium Sample Size, 20*(Values from 1000-100,000): 70.353 sec
Large Sample Size, 3*(Values from 100,000-10,000,000): 184.232

Discussion

In our own testing, we can see that before n=1,000,000, the growth in time resembles an exponential function. After that, the trend becomes relatively linear. This is likely the case because the c values we used were capped at 100 after n=1,000,000 **SOURCE**.

The step that contributes to this phenomenon is likely when we randomly generate a column position for each row. The larger the board, the faster this step will be as there is a less likely possibility of landing on a position with conflict. This suggest that using our initializing function, the larger the board size, the more efficient it relatively is.

As for the required testing, our algorithm satisfies the time constraints. When N=10,000,000, it only 184 seconds, well under the 5-minute limit.

The speed at which large N values are solved can be improved if we used a lower level language such as C. However, as the initialization phase of our algorithm is still not strictly O(N), it is still not as fast as the NASA algorithm.