ASSIGNMENT 8

Chessboard Eight Queens Problem

Mohd Mohtashim Nawaz* , Harsh Aryan†, Mayank Taksande‡ and Sneha Mishra§
Indian Institute of Information Technology, Allahabad
Allahabad-211012
Email: *itm2017005@iiita.ac.in, †itm2017003@iiita.ac.in, ‡ itm2017008@iiita.ac.in, § itm2017004@iiita.ac.in

Abstract*—This Paper introduces an algorithm to find the solution of the classic eight queen problem. The idea is to find out an efficient algorithm such that the time complexity, as well as space complexity, is minimized. The idea is to use the branch and bound technique to look for the optimal solution.

I. Introduction

Chessboard and eight queens problem states that given a chessboard of 8x8, how 8 queens can be placed on the chessboard such that no two queens attack each other. In other words, all the queens must be safe.

This is a classic problem also called as N-queens problem. Here, in the problem statement, the number of queens is fixed to 8. To solve the problem two methods have been developed. One uses backtracking to try all possibilities and find the optimal solution while other one uses the branch and bound technique.

The solution proposed here makes use of branch and bound technique. In this approach, the unwanted possibilities are bounded at each step and thus the optimal solution is achieved.

In Branch and Bound solution, after building a partial solution, we figure out that there is no point going any deeper as we are going to hit a dead end.

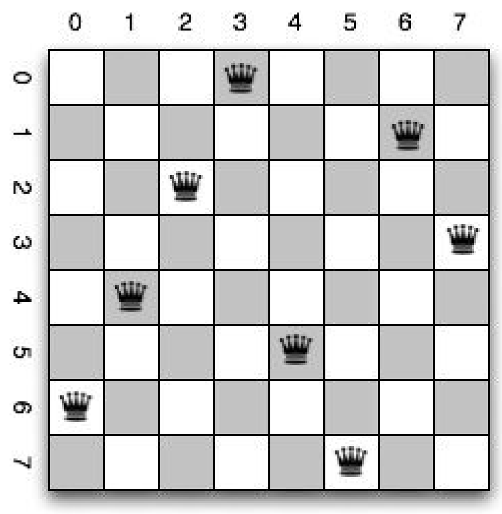

Fig 1.1: A solution of n-Queens problem

Here we will try to develop an efficient algorithm to solve the problem using the branch and bound.

This algorithm is tested multiple times for time complexity, space complexity and accuracy. In our case, we tend to focus on establishing a rule or a relationship between the time and input data.

This report further contains -

II. Algorithm Design And Analysis

An efficient algorithm can only be created by considering every possibility. So we have to consider every possibility while designing the algorithm.

Before designing the algorithm, the following points should be kept in mind.

1. No two queens share a column.

2. No two queens share a row.

3. No two queens share a top-right to left-bottom diagonal.

4. No two queens share a top-left to bottom-right diagonal.

Steps for designing the algorithm include:

1). Before placing the queen on a row, first, check whether the row is used.

2). Then check whether the left and right diagonals are used.

3). If yes, then try a different location for the queen.

4). If not, mark the row and the two diagonals as used and recurse on to next queen.

5). After the recursive call returns and before trying another position for the queen, reset the row, left diagonal and right diagonal as unused again, like in the code from the previous notes.

Algorithm to Print the Chessboard

```
def ShowBoard(board[N][N] : int)
        i: int                          [1]
        while(i<N)                      [1]
                while(j<N)              [1]
                        print board[i][j]   [1]
                print "\n"              [1]
```

Time Analysis:
Worst Case: O(n*n)
Best Case: Ω(n*n)
Average Case: Θ(n*n)

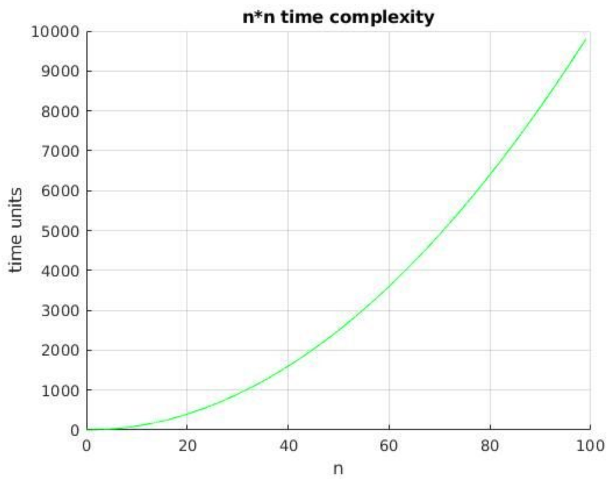

*Fig 2.1: Graph depicting n*n complexity*

Algorithm to check safety of queen

```
bool CheckSafety (row:int, int:col, diag1[N][N]: int ,
diag2[N][N]: int, row_check[]: bool, diag1_check[]:bool,
diag2_check[]:bool )

        if (diag1_check [diag1[row][col]] || diag2_check
        [diag2 [row][col]] || row_check[row])
                                        [3+1+3+1+1]
                return false            [1]
        return true                     [1]
```

Time Analysis:
= 9+1=10 units
Worst Case: O(1)
Best Case: Ω(1)
Average Case: Θ(1)

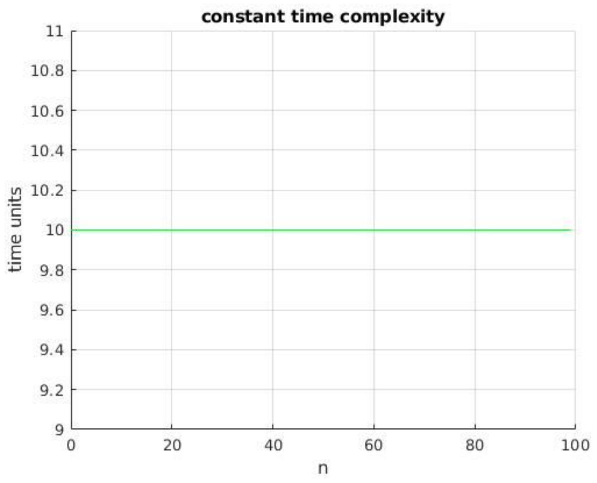

Fig 2.2: Graph depicting constant complexity

Algorithm to put Queens at chessboard safely

```
bool main_solve(int board[N][N], int col,
   int diag1[N][N], int diag2[N][N], bool row_check[N],
   bool diag1_check[], bool diag2_check[] )

    if (col >= N)                               [1]
        return true                             [1]
    for (int i = 0; i < N; i++)                 [4]
        if ( CheckSafety(i, col, diag1, diag2,  [11]
        row_check, diag1_check, diag2_check) )
            board[i][col] = 1                   [1]
            row_check[i] = true                 [1]
            diag1_check[diag1[i][col]] = true   [1]
            diag2_check[diag2[i][col]] = true   [1]
            if ( main_solve(board, col + 1, diag1,
            diag2,row_check, diag1_check, diag2_check)
                return true                     [1]

            board[i][col] = 0                   [1]
            row_check[i] = false                [1]
                diag1_check[diag1[i][col]] = false
                                                [4]
                diag2_check[diag2[i][col]] = false
                                                [4]
    return false                                [1]
```

Time Analysis for Best Case:
= 21n+34
Worst Case: O(n!)
Best Case: Ω(n)
Average Case: Θ(n)

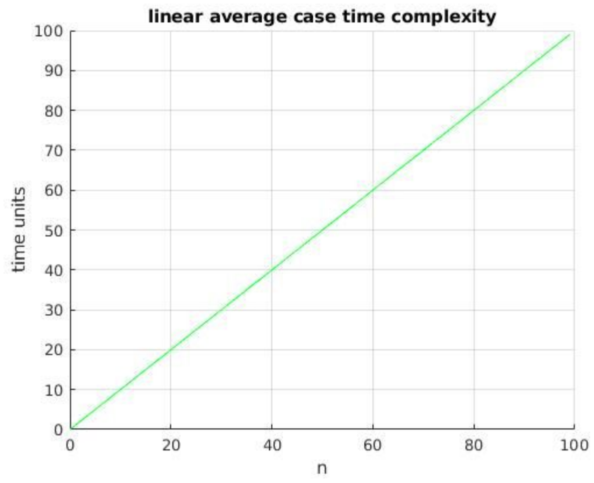

Fig 2.3:Graph depicting linear complexity

Initiator Function (code body)

```
def initiator_function()
    board[N][N] : int                                [1]
    memset(board, 0, sizeof board)                   [1]
    diag1[N][N] : int,  diag2[N][N] : int            [2]

    bool row_check[N] ←false                         [2]
    bool diag1_check[2*N - 1]←false                  [4]
    bool diag2_check[2*N - 1]←false                  [4]

    while(r<n)                                       [1]
        while(c<n)                                   [1]
                diag1[r][c]←r + c                    [4]
                diag2[r][c]←r - c + 7                [5]

    if (main_solve(board, 0, diag1, diag2,
    row_check, diag1_check, diag2_check) = false )   [n!]
        print "No Solution\n"                        [1]
        return false                                 [1]
    ShowBoard(board)                                 [1]
    return true                                      [1]
```

Time Complexity:
Worst Case: O(n!)
Best Case: Ω(n)

Main function (code body)

```
def main()
    initiator_function()
    //No input needed as problem has
    //fixed size of N=8
  return 0
```

Time Complexity:
Worst Case: O(n!+n*n)
Best Case: Ω(n+n*n)=Ω(n*n)

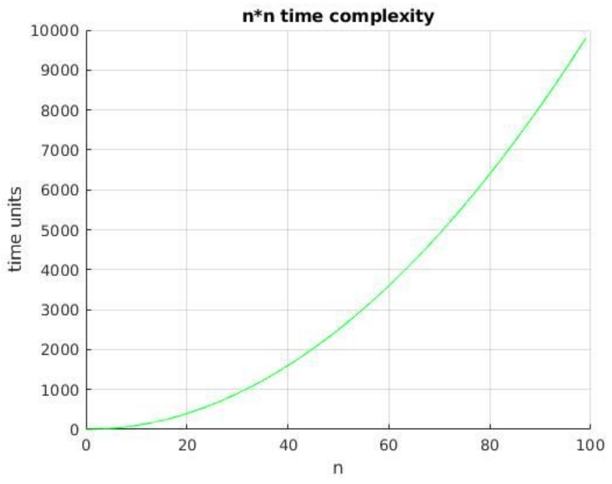

Fig 2.4:Graph depicting quadratic best case complexity

III. Illustration

We first place a queen at a particular row and column. For example we first place a queen at point 4,2 (shown in black). Corresponding to that we find blocked cells, that is the cells located horizontally, vertically or diagonally and mark them false.

Next we find a position which is still marked true and then place the queen there. For example cell 3,4 (shown in white) We now find blocked cell and repeat this process for n-queens (8 in our case). Finally, we get a solution as shown in Fig 1.1 in Page 1.

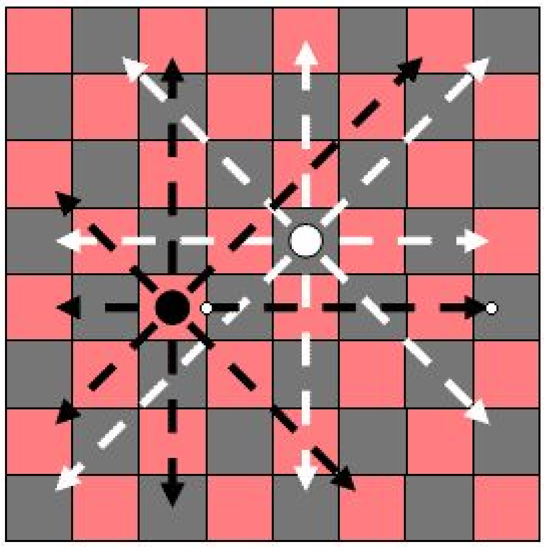

Fig 3.1:Figure illustrating positioning of queens

IV. Parallel Approach

An alternate method to solve the problem of the chessboard and eight queens is to make use of backtracking.

The idea is to place queens one by one in different columns, starting from the leftmost column. When we place a queen in a column, we check for clashes with already placed queens. In the current column, if we find a row for which there is no clash, we mark this row and column as part of the solution. If we do not find such a row due to clashes then we backtrack and return false.

Algorithm for the backtracking solution is:

1) Start in the leftmost column.
2) If all queens are placed return true.
3) Try all rows in the current column. Do the following for every tried row.
   a) If the queen can be placed safely in this row then mark this [row, column] as part of the solution and recursively check if placing queen here leads to a solution.
   b) If placing the queen in [row, column] leads to a solution then return true.
   c) If placing queen doesn't lead to a solution then umark this [row, column] (backtrack) and go to step (a) to try other rows.
3) If all rows have been tried and nothing worked, return false to trigger backtracking.

V. Result

Overall Time Complexity

Time Complexity of the algorithm is not equal to the actual time required to execute particular code but the number of times a statement executes.
We have here used the Branch and Bound technique to solve this 8-Queen Problem.
The Utility function which is called recursively is called n times in the first iteration and then n-1 times, n-2 times so on. Thus, the Overall Time Complexity becomes n!.

The normal Backtracking involves function calls n times in each iteration thus the time complexity grows to be n^n.

When executed on local machines for N = 32, the backtracking solution takes 659.68 seconds while above branch and bound solution takes a fairly less 119.63 seconds. The difference will be even huge for larger values of N.

In Conclusion, The Branch and Bound Method is at least 20 times faster than normal Backtracking.

Space Complexity

The Branch and Bound Approach involves managing two matrices of size n * n which only stores integer values in diagonals. Thus the total space required by Branch and Bound Algorithm is 3(n^2).
The Backtracking Algorithm on the other hand stores only variables and the main matrix, all the operations are performed on the same matrix no extra matrix is required

Thus Space Complexity of Branch and Bound Algorithm is O($3n^2$) whereas the Space Complexity of Backtracking Algorithm is O(n^2).

VI. Conclusion

So here the algorithms to solve the eight queens problem have been developed by using the method of the branch and bound. It can be thought of as an efficient algorithm to solve the problem as the other approach i.e. use of backtracking takes much more time to solve the same problem. The situation worsens as the input size increases.

At first sight, backtracking may seem convenient and easy but on studying deeper it is found that backtracking causes troubles and takes a lot of time as compared to the branch and bound. Branch and bound chooses the best possibility at each step and hence reduces the number of calculations and gives an optimal solution in much lesser time.

Thus, the use of the branch and bound approach to solve the problem is the best possible solution to the given problem.

VII. References

[1] ”Eight queens puzzle”, Wikipedia [Online] Available:- https://en.wikipedia.org/wiki/Eight_queens_puzzle
[Accessed: April 10, 2019]

[2] ”Eight Queens Problem”, GeeksforGeeks [Online] Available:-https://www.geeksforgeeks.org/8-queen-problem/
[Accessed: April 10 2019]

[5] “N Queen Problem Backtracking”, GeeksforGeeks [Online]
Available:-
https://www.geeksforgeeks.org/8-queen-problem
[Accessed: Arpil 12 2019]

[3] Thomas H. Corman, Introduction to Algorithms, 3rd Edition 2003

[4] Ellis Horowitz,Sartaj Sahni,Sanguthevar Rajasekaran,Fundamentals of Computer Algorithms, 2nd Edition