Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

A Mini Project Report

On

"N-queen problem using backtracking"

[Course Code: COMP 306]

(For partial fulfillment of III Year/ II Semester in Computer Engineering)

Submitted By:

Sansrit Paudel (35)

Submitted To:

Dr. Rajani Chulyadyo

Department of Computer Science and Engineering

Submission Date:

24th November, 2020

Abstract

Backtracking is finding the solution of a problem whereby the solution depends on the previous steps taken. Backtracking algorithm determines the solution by systematically searching the solution space for the given problem. Backtracking is a depth-first search with any bounding function. All solution using backtracking is needed to satisfy a complex set of constraints. Arranging N queens on an NxN chessboard such that no queen can strike down any other queen is the one of the most area where concept of backtracking is implemented. A queen can attack horizontally, vertically, or diagonally. The solution to this problem is also attempted in a similar way. The time of execution with growth of program size has been clearly visualized.

Keywords: *Backtracking, algorithms, chessboard, N-queen, optimization.*

Table of Contents

List of Figures

Chapter 1: Introduction.

1.1. Background

Backtracking:

Backtracking is finding the solution of a problem whereby the solution depends on the previous steps taken. In backtracking, we first take a step and then we see if this step taken is correct or not i.e., whether it will give a correct answer or not. And if it doesn't, then we just come back and change our first step. In general, this is accomplished by recursion. Thus, in backtracking, we first start with a partial sub-solution of the problem (which may or may not lead us to the solution) and then check if we can proceed further with this sub-solution or not. If not, then we just come back and change it.

Thus, the general steps of backtracking are:

- start with a sub-solution
- check if this sub-solution will lead to the solution or not
- If not, then come back and change the sub-solution and continue again

Backtracking Algorithm:

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.

- Start in the leftmost column
- If all queens are placed
  return true
- Try all rows in the current column.

Perform 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 unmark this [row, column] (Backtrack) and go to

step (A) to try other rows.

- If all rows have been tried and nothing worked, return false to trigger backtracking.

1.2. Objectives

N queens on NxN chessboard

One of the most common examples of the backtracking is to arrange N queens on an NxN chessboard such that no queen can strike down any other queen. A queen can attack horizontally, vertically, or diagonally. The solution to this problem is also attempted in a similar way. We first place the first queen anywhere and then place the next queen in any of the safe places. We continue this process until the number of unplaced queens becomes zero (a solution is found) or no safe place is left. If no safe place is left, then we change the position of the previously placed queen.

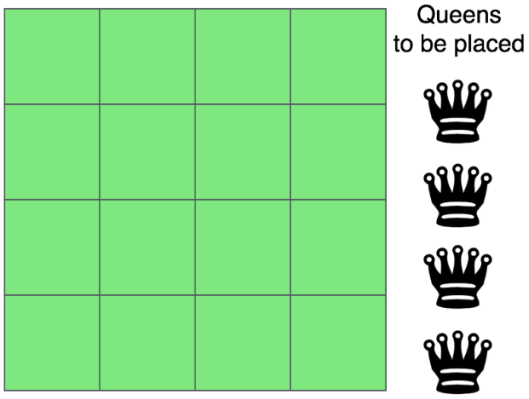
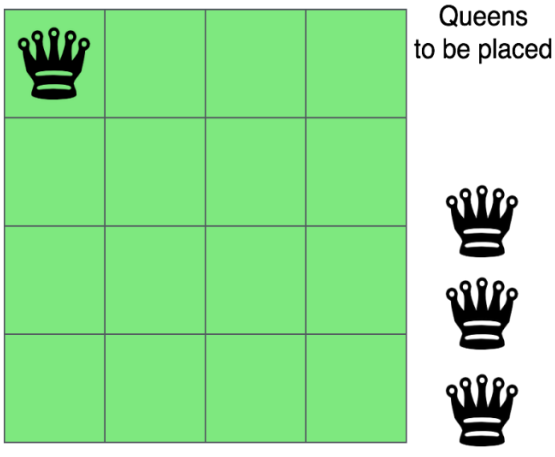
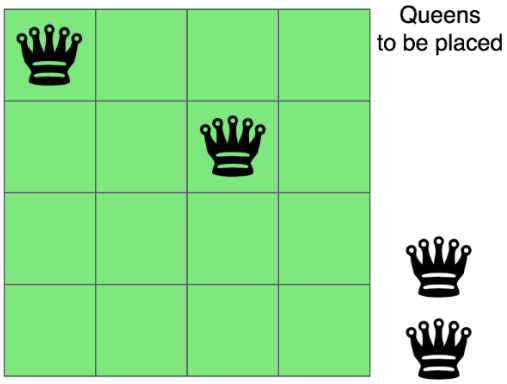


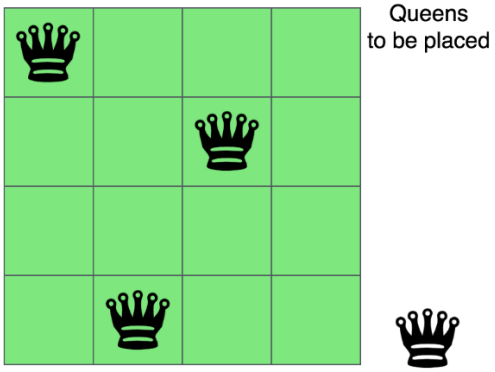
Figure 1: Matrix board for N Queen problem

The above picture shows an NxN chessboard and we have to place N queens on it. So, we will start by placing the first queen.

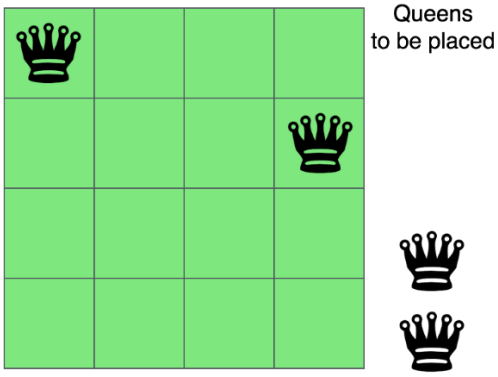


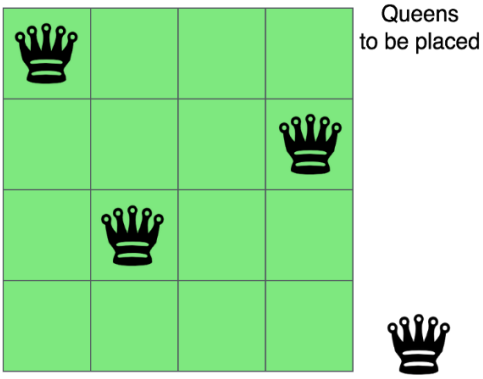
Now, the second step is to place the second queen in a safe position and then the third queen.





Now, we can see that there is no safe place where we can put the last queen. So, we will just change the position of the previous queen. And this is backtracking. Also, there is no other position where we can place the third queen so we will go back one more step and change the position of the second queen.





And now we will place the third queen again in a safe position until we find a solution. We will continue this process and finally, we will get the solution as shown below.

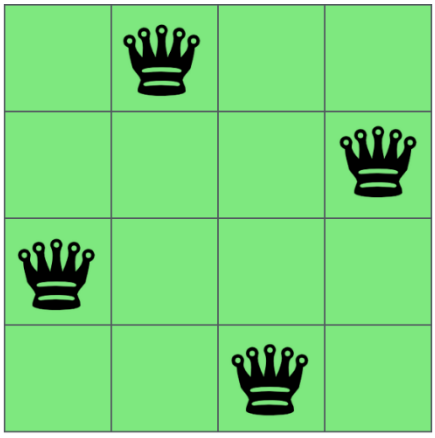


Figure 2:Solved problem on 4x4 board

Chapter 2: Source Code

```
import time

print ("Enter the number of queens")
N = int(input())

start = time.time()

#chessboard
#NxN matrix with all elements 0
board = [[0]*N for _ in range(N)]

def is_attack(i, j):
    #checking if there is a queen in row or column
    for k in range(0,N):
        if board[i][k]==1 or board[k][j]==1:
            return True
    #checking diagonals
    for k in range(0,N):
        for l in range(0,N):
            if (k+l==i+j) or (k-l==i-j):
                if board[k][l]==1:
                    return True
    return False

def N_queen(n):
    #if n is 0, solution found
    if n==0:
        return True
    for i in range(0,N):
        for j in range(0,N):
            '''checking if we can place a queen here or not
            queen will not be placed if the place is being attacked
            or already occupied'''
            if (not(is_attack(i,j))) and (board[i][j]!=1):
                board[i][j] = 1
                #recursion
                #wether we can put the next queen with this arrangment or not
                if N_queen(n-1)==True:
                    return True
                board[i][j] = 0

    return False


N_queen(N)
for i in board:
    print (i)

end = time.time()

print(end - start);
```

Figure 3:Source Code

Explanation of the code

is_attack(int i,int j) → This is a function to check if the cell (i,j) is under attack by any other queen or not. We are just checking if there is any other queen in the row 'i' or column 'j'. Then we are checking if there is any queen on the diagonal cells of the cell (i,j) or not. Any cell (k,l) will be diagonal to the cell (i,j) if k+l is equal to i+j or k-l is equal to i-j.

N_queen → This is the function where we are really implementing the backtracking algorithm.

if(n==0) → If there is no queen left, it means all queens are placed and we have got a solution.

if((!is_attack(i,j)) && (board[i][j]!=1)) → We are just checking if the cell is available to place a queen or not. is_attack function will check if the cell is under attack by any other queen and board[i][j]!=1 is making sure that the cell is vacant. If these conditions are met then we can put a queen in the cell – board[i][j] = 1.

if(N_queen(n-1)==1) → Now, we are calling the function again to place the remaining queens and this is where we are doing backtracking. If this function (for placing the remaining queen) is not true, then we are just changing our current move – board[i][j] = 0 and the loop will place the queen on some another position this time.

Testing on different size board with increments of queen.

```
4
[0, 1, 0, 0]
[0, 0, 0, 1]
[1, 0, 0, 0]
[0, 0, 1, 0]
0.0010042190551757812
```

Figure 5:Solution for 4x4 matrix

```
6
[0, 1, 0, 0, 0, 0]
[0, 0, 0, 1, 0, 0]
[0, 0, 0, 0, 0, 1]
[1, 0, 0, 0, 0, 0]
[0, 0, 1, 0, 0, 0]
[0, 0, 0, 0, 1, 0]
0.094329833984375
```

Figure 6:Solution for 6x6 matrix

```
8
[1, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 1, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 1]
[0, 0, 0, 0, 0, 1, 0, 0]
[0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 0]
[0, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 1, 0, 0, 0, 0]
3.318171977996826
```

Figure 4:Solution for 6x6 matrix

```
10
[1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 1, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 1, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
[0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 1, 0]
[0, 1, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 0, 0, 0]
20.737893104553223
```

Figure 8:Solution for 10x10 matrix

```
12
[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
[0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0]
[0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0]
[0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0]
482.2204704284668
```

Figure 7:Solution for 12x12 matrix

Execution Time Vs Size of board.

Size of board	No Of Queens	Execution Time(ms)
4x4	4	0.001004219
6x6	6	0.094329834
8x8	8	3.318171978
10x10	10	20.7378931
12x12	12	482.2204704

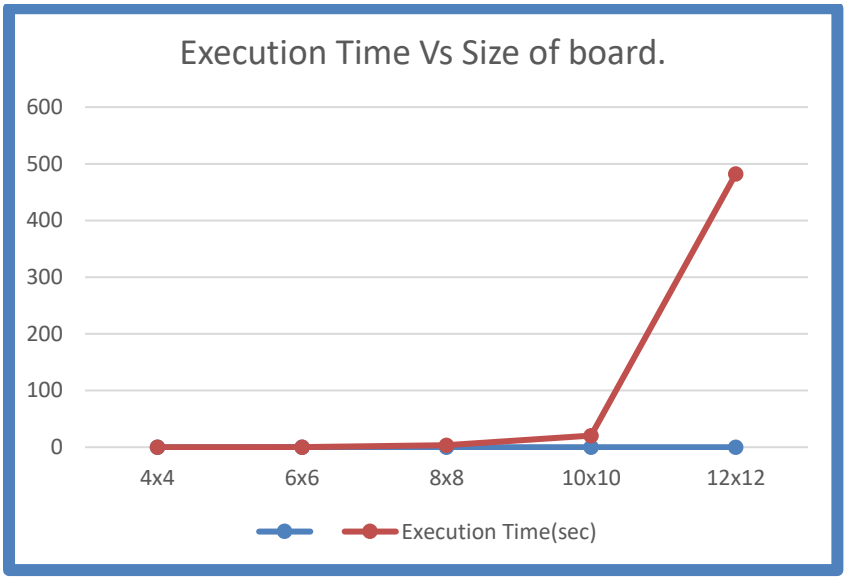


Figure 9:Graph-Execution time(sec) VS size of board

The execution time was calculated with incremental of board size and no of queens. The execution time was less for smaller boards like 4x4, 5x5 and execution time grew on polynomial degree.

Chapter 3: Conclusion and Recommendation

With all of the above test I have successfully verified the execution time grows by exponential or polynomial factor as the size of board and no of queens increases. Use of Backtracking was more robots and better algorithm than brute force. Optimized backtracking algorithm can work even better and faster. Here I have tested on 4x4, 6x6, 8x8, 10x10, 12x12 sizes and got the execution time as 0.001004219, 0.094329834, 3.318171978, 20.7378931 and 482.2204704 seconds respectively.