

Report

for

Solution of N-Queen Problem Using Backtracking Algorithm

Prepared by:

Amr Muhammad Gaber

Muhammad Ehab Muhammad

Muhammad Ahmed Abdel-Aziz

Group: 3

Assuit University

May24, 2022

Contents

1. Introduction.....	1
1.1 Constraint Satisfaction Problems (CSPs):.....	1
1.2 Formal Definition:.....	1
1.3 Solution:	2
1.4 About Backtracking:	2
2. Methodology.....	3
2.1 Backtracking:	3
2.2 Backtracking Methodology:	3
2.3 Pseudocode:.....	3
2.4 Discussion how the algorithm can solve the 8-Queen problem:	4
2.5 Time Complexity of N-Queen Algorithm:	5
3. Experimental Simulation	5
3.1 Programming Language: We Developed This Project Using <i>Python</i>	5
3.2 Environment: VS Code.	5
3.3 Primary Function:.....	5
4. Results and Technical Discussion.....	12
5. Conclusion & Recommendations	17
5.1 Conclusion:	17
5.2 Recommendations:	17
.6 References.....	18
.7 Appendix.....	18
7.1 What is backtracking algorithm!?.	Error! Bookmark not defined.

1. Introduction

1.1 Constraint Satisfaction Problems (CSPs):

Constraint satisfaction problems (*CSPs*) are mathematical questions defined as a set of objects whose state must satisfy several constraints or limitations. *CSPs* represent the entities in a problem as a homogeneous collection of finite constraints over variables, solved by constraint satisfaction methods. *CSPs* are the subject of research in both artificial intelligence and operations research since the regularity in their formulation provides an everyday basis to analyze and solve problems of many seemingly unrelated families. *CSPs* often exhibit high complexity, requiring a combination of heuristics and combinatorial search methods to be solved in a reasonable time. Constraint programming (*CP*) is the field of research that specifically focuses on tackling these kinds of problems. Additionally, Boolean satisfiability problem (*SAT*), the satisfiability modulo theories (*SMT*), mixed integer programming (*MIP*), and answer set programming (*ASP*) are all fields of research focusing on the resolution of particular forms of the constraint satisfaction problem.

Examples of problems that can be modeled as constraint satisfaction problems include:

- Type inference.
- 8-Queen puzzle.
- Map coloring problem.
- Maximum cut problem.
- Sudoku.

1.2 Formal Definition:

Formally, a constraint satisfaction problem is defined as a triple (X, D, C) , where

- $X = \{X_1, \dots, X_n\}$
- $D = \{D_1, \dots, D_n\}$
- $C = \{C_1, \dots, C_n\}$

Each variable X_i can take on the values in the nonempty domain D_i . Every Constraint $C_i \in C$ is in turn a pair (t_i, R_i) , where $t_i \subseteq X$ is a subset of K variables and R_i is a K -ary relation on the

corresponding subset of domains D_i . An evaluation of the variables is a function from a subset of variables to a particular set of values in the corresponding subset of domains. An evaluation v satisfies a constraint (t_i, R_i) if the values assigned to the variable's t_i satisfies the relation R_i .

An evaluation is consistent if it does not violate any of the constraints. An evaluation is complete if it includes all variables. An evaluation is a solution if it is consistent and complete; such an evaluation is said to solve the constraint satisfaction problem.

1.3 Solution:

Constraint satisfaction problems on finite domains are typically solved using a form of search. The most used techniques are variants of backtracking, constraint propagation, and local search. These techniques are also often combined, as in the VLNS method, and current research involves other technologies such as linear programming.

One of these solutions is "**Backtracking**", and that is our main algorithm to solve our N-Queen Problem.

1.4 About Backtracking:

Backtracking can be applied only for problems which admit the concept of a "partial candidate solution" and a relatively quick test of whether it can possibly be completed to a valid solution. It is useless, for example, for locating a given value in an unordered table. When it is applicable, however, backtracking is often much faster than brute-force enumeration of all complete candidates since it can eliminate many candidates with a single test.

Backtracking is an important tool for solving constraint satisfaction problems, such as crosswords, verbal arithmetic, Sudoku, and many other puzzles. It is often the most convenient technique for parsing, for the knapsack problem and other combinatorial optimization problems. It is also the basis of the so-called logic programming languages such as Icon, Planner and Prolog.

2. Methodology

2.1 Backtracking:

The classic textbook example of the use of backtracking is the 8-Queen puzzle, that asks for all arrangements of eight chess queens on a standard chessboard so that no queen attacks any other. In the common backtracking approach, the partial candidates are arrangements of k queens in the first k rows of the board, all in different rows and columns. Any partial solution that contains two mutually attacking queens can be abandoned.

2.2 Backtracking Methodology:

The idea is to place queens one by one in different columns, backtracking mechanism is to start from the left column to the right or from the upper row to down in order to check the prior situations. When a queen is placed in a board, we check for clashes with already placed queens, as each stationed board produces a red zone expands to row, column, and diagonal. In the current column, if a row is found for which there is no clash, we mark this row and column as part of the solution, if only clashes then backtrack and return false.

2.3 Pseudocode:

1. Start in the leftmost column.
2. If all queens are placed
 return true.
3. Try all rows in the current column.
 Do 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 returns 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.
4. if all rows have been tried and nothing worked,
 return false to trigger backtracking.

2.4 Discussion how the algorithm can solve the 8-Queen problem:

To explain the role of Backtracking in solving our problem let's do it through some steps as follows:

First 1: we will put the first queen in the first correct place knowing that there are other valid places, but we will do it in succession.

Step 2: We want to put another queen on our board so we'll go to the next column and check every cell inside until we find a valid place, then put the queen in and see if we can find another valid place in the next column for the next queen.

But what should we do if we do not find any valid place?

This is when Backtracking is used.

Step 3: When there is no valid place for the queen we must go back and change the place of the previous of the current queen and this is called Backtracking, so why do that? Because this means that there were other valid places for the previous queen, we should try, maybe this new arrangement meets our need and will provide a valid place for the next queen.

Step 4: What do we do if this also doesn't work and there is no other valid place for the previous queen? Then we will go back to the previous queen of the current and look for another valid place for her.

Step 5: What if this also doesn't work? Then we will do it again and go back to the previous from the previous. What if this also doesn't work?

Step 6: We will do this again until we get to the first queen where all the cells of the column were valid, and we choose the next cell of the current in the same column.

Step 7: This can finally solve the problem but let's see if this is true or not. we will do these steps from the first one again until we make sure that all the queens on the board are in the right places according to the rules or there is no solution to this problem.

All the backing steps that led us to the solution were through backtracking and that would explain the role of Backtracking in the N Queens problem.

2.5 Time Complexity of N-Queen Algorithm:

Time complexity of N-Queen algorithm: For finding a single solution where the first queen 'Q' has been assigned the first column and can be put on N positions, the second queen has been assigned the second column and would choose from N-1 possible positions and so on; the time complexity is $O(N) * (N - 1) * (N - 2) * \dots 1$. i.e., The worst-case time complexity is $O(N!)$. Thus, for finding all the solutions to the N Queens problem the time complexity runs in polynomial time.

3. Experimental Simulation

3.1 Programming Language: We Developed This Project Using *Python*

3.2 Environment: VS Code.

3.3 Primary Function:

First, our code consists of 4 methods, each of which we will explain in detail as follows:

- **Print Solution Function:**

```
def printSolution(board):  
    for i in range(N):  
        for j in range(N):  
            print(board[i][j], end=" ")  
        print()
```

- *Arguments:*

⇒ Board: the matrix that represents the chessboard.

- *Function:*

This function is responsible for printing the chessboard. The board is taken as an input, and the board represents a square matrix 4x4, 8x8 or 16x16 matrix based on N number of your Queens.

- *Structure:*

To print a matrix, we need 2 nested loops 1 for the rows and 1 for each cell in those rows so we have written two loops the I loop, and j loop will stop when we reach N Queen the printing operation is done as follow, one print function to print each cell in the row separated by white spaces and there Another print function that separates each row with one line.

- **IsSafe Function:** This is one of our basic functions and it takes 3 arguments.

```
def isSafe(board, row, col):
    for i in range(col):
        if board[row][i] == 1:
            return False

    for i, j in zip(range(row, -1, -1),
                    range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    for i, j in zip(range(row, N, 1),
                    range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    return True
```

- *Arguments:*

⇒ Board: the matrix that represents the chessboard.

⇒ Row: The current row we are dealing with.

⇒ Column: also, current column.

- *Function:*

IsSafe is responsible for checking the current cell validity to put the queen inside or not in other words to check if the queen can be placed on the board[row] [column] that is passed to the job.

- *Structure:*

The function consists of 3 parts in the form of 3 loops, each part to check a constraint of the problem, we will explain each of them as follows:

First thing, keep in mind that for the place on board to be valid 3 constraints must be fulfilled to ensure that there is no queen threatening another.

- 1) Only 1 Queen could be placed at each row.
- 2) No Queens should be placed at the same upper diagonal on left side.
- 3) No Queens should be placed at the same lower diagonal on left side.

(Note that this function is called when "col" queens are already placed in columns from 0 to col -1. So, we need to check only left side for attacking queens)

- Now first part of IsSafe function is:

Simple for Loop responsible for checking the validity of the first constraint there are no Queens placed in the current row ,the function will start iterating on every cell preceding the cell whose validity we want to determine in the same row (we stop at column -1, we just want to check preceding cells in the same row) If there is 1 at any preceding cell, it means there is already a Queen in that row, then IsSafe function will finish returning false as a result of validating this place (cell) if it is not false, We continue to check for the next constraint after the loop ends.

- Second part of IsSafe:

Is another for loop this time is responsible for checking the validity of the second constrain there are no Queens placed at the same upper diagonal on left side .it starts looping from the current place at board[row][col] then decreasing the row and column by 1 at each cycle to access the upper diagonal of the current place or cell we wants to check it 's validity if there is 1 in any of these diagonal cells this means there is already a Queen in this diagonal then the Whole IsSafe function will be ended returning false as a result of the validity check of this place (cell) if not false then we continue to check the next constraint after the loop finishes.

(The loop will be terminated if row or column reached 0 or returned false as well).

- Third part of IsSafe:

Another for loop but this time is responsible for checking the validity of the third constrain there are no Queens placed at the same lower diagonal on left side .it starts looping from the current place at board[row][col] then increasing the row by one and decreasing column by 1 at each cycle to access the lower diagonal of the current place or cell we wants to check it's validity if there is 1 at any of these diagonal cells ,this means there is already a Queen in this diagonal then the Whole IsSafe function will be ended returning false as a result of the validity check of this place (cell) if not false then we continue to check the next constraint after the loop finishes

(The loop will be terminated if row reached N the number of Queens or column reached 0 or returned false as well).

If all of the 3 Loops ended without returning false, then the IsSafe function returns true means this place board[row][col] is valid for putting Queen.

- **solveNQUtil:** This can be considered our main function because it contains the Backtracking algorithm, which is our main algorithm for solving the N Queen problem.

```
def solveNQUtil(board, col):  
    if col >= N:  
        return True  
  
    for i in range(N):  
        if isSafe(board, i, col):  
            board[i][col] = 1  
            if solveNQUtil(board, col + 1) == True:  
                return True  
            board[i][col] = 0  
  
    return False
```

- *Arguments:*
 - ⇒ Board: the matrix that represents the chessboard.
 - ⇒ Column: current column value.
- *Function:*

solveNQUtil() is responsible for placing Queens at only valid places at the board to solve our N Queen problem by trying all possible arrangements until reach the solution when all N Queens are placed at the NxN board and no Queen is threatening another.
- *Structure:*

Basically solveNQUtil() represent a good example of recursion it contains two parts:
- first part of solveNQUtil function is:

If statement to set the base case: If all queens are placed, then the function should return true and terminate the recursion .

- Second part of solveNQUtil:

For loop iterates until reaches N (To be accurate it's N-1). This loop tries to place the Queen at any cell(row) of the current column by checking the validity of the cell using IsSafe function.

If it wasn't valid we iterates the loop and try another cell in the same current column (increment row) but if it was valid then we put 1 in this cell as a mark of putting a Queen here then we start the recursion and call solveNQUtil() function ,passing the current state of board and increasing the current column by 1.

To check if we can put another Queen in the next column or not (knowing that we have not finished the loop of the previous call of solveNQUtil() function yet) after calling the function again we do the same previous steps and see if any cell is valid in this column we call the function again and go to the next column until we put all Queens in place on board.

But if we haven't found any valid places in this column we will go back to the previous column (Backtracking) and remove the Queen that we have put already by changing the 1 to 0 and try to put the Queen at another cell of this column.

Then if this did not happen and we reached the last cell of the current column (finished the loop) then we go back again and remove the previous Queen of the previous column and do the whole thing all over again until we find valid new place for the Queen we would call the solveNQUtil() to check if we can put another Queen in the next column and then we will do it again and again with 2 cases to happen if valid we proceed if not we continue until loop finishes and if still not we go back (backtracks) to choose different arrangements for the previous Queen or Queens ,but if all arrangements was unable to solve our problem by putting all the N Queen in the NxN board considering the previous constraints then the solveNQUtil() function returns false.

(That's the whole concept of Backtracking in our N Queen Problem.)

- **solveNQ:**

```
def solveNQ():
    board = [[0]*N for _ in range(N)]

    if solveNQUtil(board, 0) == False:
        print("Solution does not exist")
        return False

    printSolution(board)
    return True
```

-

Arguments:

⇒ This function takes no arguments.

- *Function:*

This function solves the N-Queen problem using *Backtracking*. It mainly uses solveNQUtil() to solve the problem. It returns false if queens cannot be placed, otherwise return true, and placement of queens in the form of 1s.

(Note that there may be more than one solution, this function prints one of the feasible solutions.)

- *Structure:*

⇒ First:

we used List Comprehension to make the N*N matrix that represent the chessboard (*N is a global attribute set to 4, 8 or 16 as example at the beginning of the code to specify this wanted N in the N Queen problem*)

⇒ Second:

if statement to check the return result of the solveNQUtil function if it was false then there is no solution found. And solveNQ function will be ended returning false but if solveNQUtil returned true that function print Solution will be called to print the solution (the current

arrangement of Queens on the chessboard in a form of matrix with Queens represented by ones and other empty spots by zeros) and the solveNQ returns true.

4. Results and Technical Discussion

We are going to explain 2 main problems solved by our algorithm as example of our N problem solution:

1) 4-Queen problem.

2) 8-Queen problem.

we are going explain the output of our code after solving each and compare it with the expected result to ensure that our code is valid to be considered as a solution of the N Queen problem:

- *First* the 4-Queen problem:

let's trace the code manually to see what the expected result is from applying our algorithm and compare it with the actual output of the code.

We start with empty 4x4 board that has 4 columns and 4 rows each from index 0 to 3 as follow:

Step 0 : $c=0$ $r=0$ now we check the cell of board $[0][0]$ if it safe or not board $[0][0]$ =safe we put the first Queen then increment column by 1.


Step 1.0 : $c=0$ $r=0$ check ,board $[0][1]$!= safe so we increment row by 1.

Step 1.1 : $c=1$ $r=1$ check ,board $[1][1]$!= safe so we increment row by 1.



Step 1.2 : $c=1$ $r=2$ check ,board $[2][1]$ = safe it's safe

so, we will place Queen.

Now because we have already placed a Queen in this column, we increment column by 1

	0	1	2	3
0				
1				
2				
3				

Board[r][c]

	0	1	2	3
0		Step1.0 X		
1		Step1.1 X		
2				
3				

Board[r][c]

Step 2.0 : $c=2$ $r=0$ check ,board $[0][2] \neq$ safe so we increment row by 1.

Step 2.1: $c=2$ $r=1$ check ,board $[1][2] \neq$ safe so we increment row by 1.



Step 2.2: $c=2$ $r=1$ check ,board $[2][2] \neq$ safe so we increment row by 1.

Step 2.3: $c=2$ $r=1$ check ,board $[3][2] \neq$ safe

We reached last row of column 2 and there is no safe position.

We have to backtrack

Note: we have to remove last placed Queen the one at position board $[2][1]$ new row = $2+1 = 3$ column = 1

	0	1	2	3
0		Step1.0 X	Step2.0 X	
1		Step1.1 X	Step2.1 X	
2			Step2.2 X	
3			Step2.3 X	

Board[r][c]

Step 1.3 : $c=1$ $r=3$ check ,board $[3][1] =$ safe

so, we will place Queen,





we increment column by 1.

Step 2.0 : $c=2$ $r=0$ check ,board $[0][2] \neq$ safe so we increment row by 1.

Step 2.1 : $c=2$ $r=1$ check ,board $[1][2] =$ safe

so, we will place Queen,

we increment column by 1.

	0	1	2	3
0		Step1.0 X	Step2.0 X	
1		Step1.1 X		
2				
3				

Board[r][c]

Step 3.0 : $c=3$ $r=0$ check ,board $[0][3] \neq$ safe so we increment row by 1.

Step 3.1: $c=3$ $r=1$ check ,board $[1][3] \neq$ safe so we increment row by 1.

Step 3.2: $c=3$ $r=2$ check ,board $[2][3] \neq$ safe so we increment row by 1.





Step 3.3: $c=3$ $r=3$ check ,board $[3][3] \neq$ safe

We reached last row of last column and there is no safe position.

We have to backtrack Note: we have to remove last placed Queen the one at position board $[1][2]$

new row = $1+1 = 2$

column = 2

	0	1	2	3
0		Step1.0 X	Step2.0 X	Step3.0 X
1		Step1.1 X		Step3.1 X
2				Step3.2 X
3				Step3.3 X

Board[r][c]

Step 2.2: $c=2$ $r=2$ check ,board $[2][2] \neq$ safe so we increment row by 1.

Step 2.3: $c=2$ $r=3$ check ,board $[3][2] \neq$ safe

We reached last row of last column and there is no safe position.

We have to backtrack.

Note: we have to remove last placed Queen the one at position board $[3][1]$

Again, no more valid positions in this column we have to backtrack

Note: we have to remove last placed Queen the one at position board $[0][0]$

new row = $1+0 = 1$ column = 0

Step 0.1 : $c=0$ $r=1$ check ,board $[1][0] =$ safe,

so, we will place Queen,

So, we increment column by 1.

Step 1.0 : $c=1$ $r=0$ check ,board $[0][1] \neq$ safe so we increment row by 1.

Step 1.1 : $c=1$ $r=1$ check ,board $[1][1] \neq$ safe so we increment row by 1.

Step 1.2: $c=1$ $r=2$ check ,board $[2][1] \neq$ safe so we increment row by 1.

Step 1.3: $c=1$ $r=3$ check ,board $[3][1] =$ safe

so, we will place Queen,

so, we increment column by 1.

Step 2.0 : $c=2$ $r=0$ check ,board $[0][2] =$ safe

so, we will place Queen,

So, we increment column by 1

	0	1	2	3
0				
1	♙			
2				
3				

Board[r][c]

	0	1	2	3
0		Step1.0 X		
1	♙	Step1.1 X		
2		Step1.1 X		
3		♙		

Board[r][c]

	0	1	2	3
0		Step1.0 X	♙	
1	♙	Step1.1 X		
2		Step1.1 X		
3		♙		

Board[r][c]

Step 3.0 : $c=3$ $r=0$ check ,board $[0][3]$!= safe so we increment row by 1.

Step 3.1 : $c=3$ $r=1$ check ,board $[1][3]$!= safe so we increment row by 1.

Step 3.2 : $c=3$ $r=2$ check ,board $[2][3]$ = safe Place Queen.

Finally, all 4 Queens are placed at the board, so we reached or solution.

	0	1	2	3
0		Step1.0 X	♔	
1	♔	Step1.1 X		
2		Step1.1 X		♔
3		♔		

Board[r][c]

Now when we compare this result with our output there are exactly the same so that proves that our algorithm worked successfully

0	0	1	0
1	0	0	0
0	0	0	1
0	1	0	0

Our output

	0	1	2	3
0			♔	
1	♔			
2				♔
3		♔		

Board[r][c]

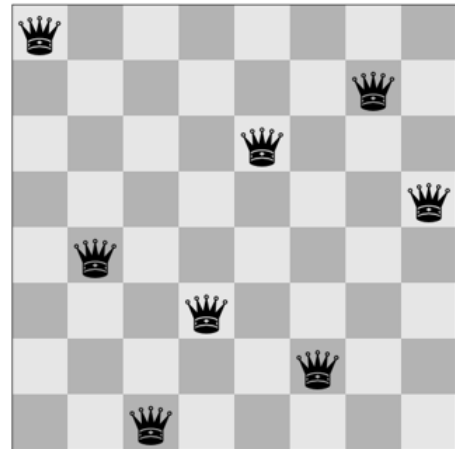
One of the expected solutions

- Now let's talk about the 8 Queen problem: 8-Queen problem has so many arrangements to check so we will only compare our output with one of the known solutions of the problem to see if our algorithm works or not.

Now when we compare this result with our output there are exactly the same so that proves that our algorithm worked successfully

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	1
0	1	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0

Our output



One of the expected solutions

5. Conclusion & Recommendations

5.1 Conclusion:

This problem is to find an arrangement of N queens on a chess board, such that no queen can attack any other queens on the board.

The chess queens can attack in any direction as horizontal, vertical, horizontal and diagonal way.

N -queen problem can be easily solved using backtracking as shown in cases of $n = 4$ & $n = 8$ even if $n = 16$ it will fit for a solution put there's no possible way to fit 3 queens in a 3×3 chess board without creating any conflict.

Generally, it is N . As ($N \times N$ is the size of a normal chess board.)

Output: The matrix that represents in which row and column the N Queens can be placed. If the solution does not exist, it will return false.

5.2 Recommendations:

- 1) Working on the computations of the time complexity formula to reduce time used to solve the problem
- 2) Find more optimal solutions with reduced cost which is represented in the solution steps as results show that with a linear increase in the number of queens, the number of recursive calls made by the algorithm increases exponentially
- 3) Conducting more experiments for reaching better results
- 4) Working on new algorithms in order to achieve more results
- 5) Comparing the results of low N value and high N value may contribute in some progresses

6. References

- ☐ [Constraint satisfaction problem - Wikipedia](#)
- ☐ [Backtracking - Wikipedia](#)
- ☐ [Eight queens puzzle - Wikipedia](#)
- ☐ [N Queen Problem | Backtracking-3 - GeeksforGeeks](#)
- ☐ <https://developers.google.com/optimization/cp/queens>

7. Appendix

Source Code:

[AI Project - Google Drive](#)