N-Queens problem

Objective:

How backtracking is used to solve N-queens problem

Standard chess board has 8X8 cells. We will use a 4X4 board to reduce the problem and understand its implementation using various approaches.

After going through the lecture notes, research work, and textbooks on the Principles of Artificial Intelligence, I have come up with two different methodologies for solving the N queens problem. Both solutions use a 4X4 board to realize the implementation.

Methodology 1: Using state space search tree and backtracking to get the optimal path

Description: We have 4 queens (Q1, Q2, Q3 and Q4) and a 4X4 board shown below, in which we need to place 4 queens in a way that they do not capture each other, i.e, they do not attack one another by being in the same row, same column, and diagonal. The moves of the queens will be horizontal in rows and columns and diagonal. There can be more than one solution to this problem and we will investigate a few of them. There are more than one arrangements for the queens on the board, we are interested in knowing about those arrangements that satisfy the conditions.

	1	2	3	4
1				
2				
3				
4				

Condition: No two queens can be in the same row, same column and same diagonal.

Approach: We will use backtracking to solve this problem, i.e we want all possible solutions/arrangements that satisfies the condition.

1. Place the queens in a 4X4 board in total 16 cells. There will be ${}^{16}C_4$ ways to place them. This is a big number denoting a lot of possibilities.
2. To reduce the problem, one can start with the condition that one queen can be in one row and one column, followed by other queens in the next row and column. For example, if the 4 queens are Q1, Q2, Q3, and Q4, then Q1 will be placed in row 1 (any column) followed by Q2 in row 2 and so on.
3. The final list of the row column combinations of the queen's placement will be the solution.
4. Refer to Fig.1 and Fig.2 for the implementation of backtracking:
    a. Generate the state space tree for queens placement in cells without attack. This can be done by placing Q1 in cell (1,1), Q2 in (2,2), Q3 in (3,3), and Q4 in (4,4). Q4 can't move beyond cell (4,4) so will stop here.
    b. Backtrack and move Q3 in cell (3,4) and Q4 in cell (4,3).
    c. Move Q2 in cell (2,3), and Q3 in cell (3,2)and Q4 in cell (4,4)
    d. Move Q3 in cell (3,4) and Q4 in cell (4,2)
    e. Move Q2 to cell (2,4), this leads to either Q3 to cell (3, 2) and Q4 to cell (4, 3) or Q3 to cell (3,3) and Q4 to cell (4,2). Now Q2 can't move further so we backtrack to Q1.
    f. Move Q1 to cell (1,2), the tree will expand similar to steps 4 through 8.
    g. Similarly, move Q1 to cells (1,3) and (1,4) and the tree will be generated accordingly.
    h. Total nodes visited in this approach = 1 + 4 + 4 * 3 + 4 * 3 * 2+ 4 * 3 * 2 *1 = 65
    i. A generic formula for this problem is $Total\,nodes\ = 1\ +\sum_{i=0}^{3}[\prod_{j=0}^{i}(4\ -\ j)]$, considering 4 queens.

j. For N queens the formula will be $Total\, nodes \;=\; 1 \;+\; \sum_{i=0}^{N-1} [\; \prod_{j=0}^{i} (N \;-\; j)]$

k. Application of bounding function (condition) on the problem - no same row, no same column, and no same diagonal.

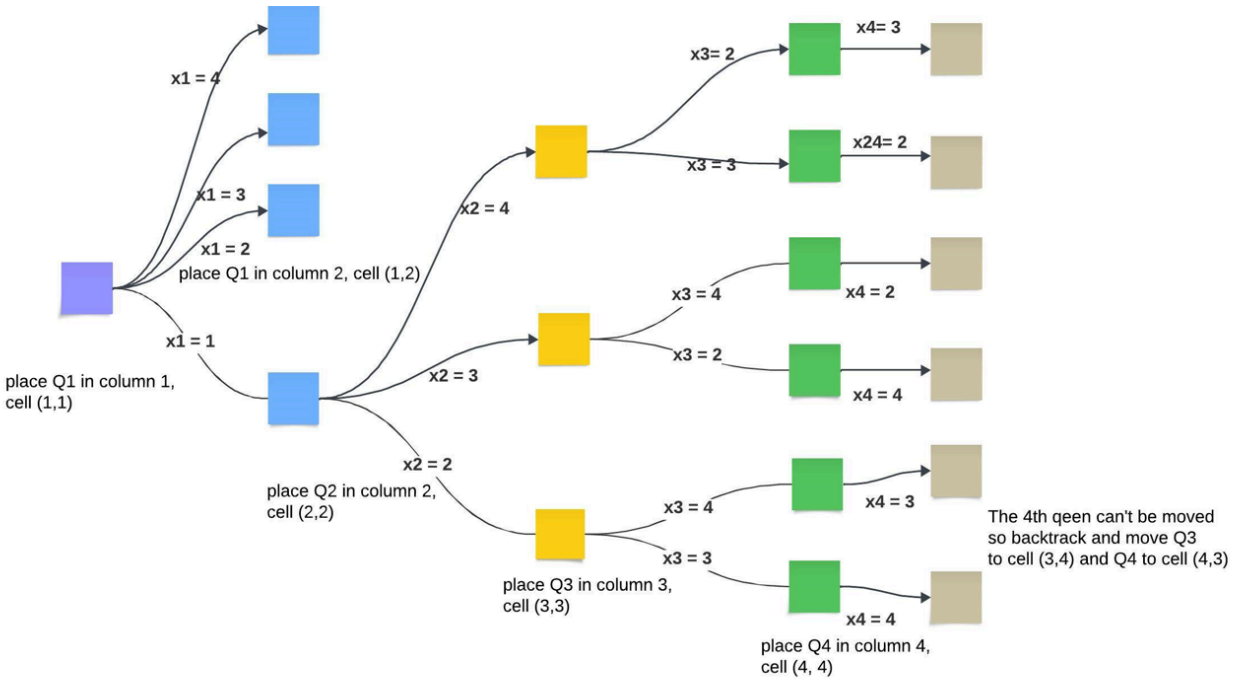
x1 = 4
x1 = 3
x1 = 2
place Q1 in column 2, cell (1,2)
x1 = 1
place Q1 in column 1, cell (1,1)
x2 = 4
x2 = 3
x2 = 2
place Q2 in column 2, cell (2,2)
x3= 2
x3 = 3
x3 = 4
x3 = 2
x3 = 4
x3 = 3
place Q3 in column 3, cell (3,3)
x4= 3
x24= 2
x4 = 2
x4 = 4
x4 = 3
x4 = 4
place Q4 in column 4, cell (4, 4)
The 4th qeen can't be moved so backtrack and move Q3 to cell (3,4) and Q4 to cell (4,3)

Fig.1 State space search tree

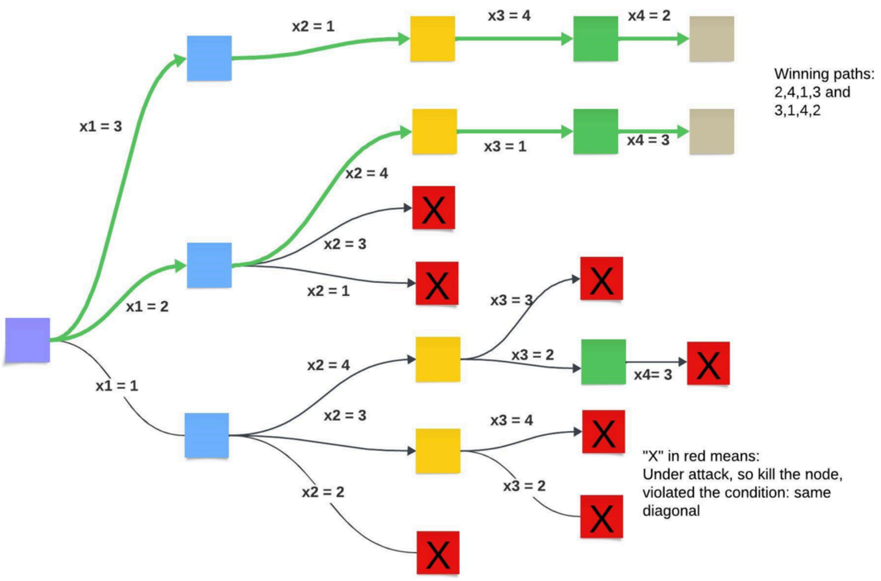
x1 = 3
x1 = 2
x1 = 1
x2 = 1
x2 = 4
x2 = 3
x2 = 1
x2 = 4
x2 = 3
x2 = 2
x3 = 4
x3 = 1
x3 = 3
x3 = 2
x3 = 4
x3 = 2
x4 = 2
x4 = 3
x4= 3
X
Winning paths: 2,4,1,3 and 3,1,4,2
"X" in red means: Under attack, so kill the node, violated the condition: same diagonal

Fig.2 Apply bounding function

Methodology 2 : Using an algorithm proposed by Nilsson (1982) and creating the Artificial Intelligence production system (AIPS) for this problem.

An AIPS consists of a database (DB), rules (operators) and a control strategy to solve a given problem, such as N-queens.

For this problem,

Database (DB) consists of a 4X4 board, and the initial state consists of no queen marks (X) on this board.

The goal state will be the placement of queens on the board such that no two X marks cancel (capture/attack) each other, i.e, no two queens in the same row, or same column or same diagonal.

Rules (Operators)

$R_{i,j}\ for\ 1 \leq i \leq 4,\ 1 \leq j \leq 4$

Precondition:

For i = 1, $1 \leq j \leq 4$, there is no X mark on the board, then

Action: Place X in row = 1, column j

For $2 \leq i \leq 4, 1 \leq j \leq 4$, there is an X mark in row i-1 then

Action: Place X in row i

Rules $R_{i,j}\ for\ 2 \leq i \leq 4,\ 1 \leq j \leq 4$ if there is an X mark in row i - 1 then place X in cell(i,j)

For example, rules here would be denoted as $R_{1,1}$, $R_{1,2}$, $R_{1,3}$, $R_{1,4}$ for the first row.

Control Strategy:

We can introduce heuristics to resolve the placement of queens and the preference given for a particular position.

Below are the two ***heuristics***:

1. ***Index based***: if two rules $R_{i,j}$ and $R_{i,k}$ can be applied then select $R_{i,j}$ if j < k, i.e, $R_{i,j}$ precedes $R_{i,k}$

2. ***A diagonal (i,j) function based***: Let diagonal i, j be the length of the longest diagonal crossing cell (i,j), $R_{i,j}$ precedes $R_{i,k}$ if diagonal(i,j) < diagonal (i,k)

   If diagonal (i,j) = diagonal (i,k), then use index based ordering.

For example, length of diagonal for cell(1,1) is 4 (longest diagonal)

Length of diagonal for cell(1,2) is 3 (longest diagonal) and 2 (shortest diagonal)

Now we apply the below backtracking algorithm to generate the solution.

Recursive procedure BACKTRACK(DATA)

```
1.  if TERM(DATA),
            return NIL;
            TERM is a predicate true for arguments that satisfy the termination condition of
    the production system. Upon successful termination, NIL, the empty list, is returned.
2.  if DEADEND(DATA),
            return FAIL;
            DEADEND is a predicate true for arguments that are known not to be on a path
    to a solution. In this case, the procedure returns the symbol FAIL.
3.  RULES ← APPRULES(DATA);
            APPRULES is a function that computes the rules applicable to its argument and
    orders them (either arbitrarily or according to heuristic merit).
4.  LOOP:
            if NULL(RULES)
            return FAIL; if there are no (more) rules to apply, the procedure fails.
5.  R ← FIRST(RULES);
            the best of the applicable rules is selected.
6.  RULES ← TAIL(RULES);
            the list of applicable rules is diminished by removing the one just selected.
```

7. RDATA ← R(DATA);

    rule R is applied to produce a new database.

8. PATH ← BACKTRACK(RDATA);

    BACKTRACK is called recursively on the new database.

9. if PATH = FAIL, go LOOP;

    if the recursive call fails, try another rule.

10. return CONS(R, PATH);

    otherwise, pass the successful list of rules up, by adding R to the front of the list.

Implementation of the algorithm:

Procedure backtracking

First Iteration:

1. Term (data) → No
2. Deadend → No
3. Rule → apply rule ($R_{1,1}$, $R_{1,2}$, $R_{1,3}$, $R_{1,4}$)
4. Loop
5. R ← First (Rules) returns $R_{1,2}$ (arrange according to the diagonal $R_{1,2}$, $R_{1,3}$, $R_{1,1}$, $R_{1,4}$)
6. Rules – { $R_{1,3}$, $R_{1,1}$, $R_{1,4}$ } (tail of the rules after reordering)
7. RDATA: Perform the action → Place X on cell(1,2)
8. Path ← backtracking (data)

Second iteration:

1. Term (data) → No
2. Deadend → No
3. Rule → apply rule ($R_{2,1}$, $R_{2,2}$, $R_{2,3}$, $R_{2,4}$)

4. Loop
5. R ← First (Rules) returns $R_{2,1}$ (arrange according to the diagonal $R_{2,1}$, $R_{2,4}$, $R_{2,2}$, $R_{2,3}$)
6. Rules – { $R_{2,4}$, $R_{2,2}$, $R_{2,3}$ } (tail of the rules after reordering)
7. RDATA: Perform the action → Place X on cell(2,1) (violates the diagonal criteria)
8. Path ← backtracking (data)
9. GOTO LOOP

Rewrite:

1. Term (data) → No
2. Deadend → Yes → return fail
3. Rule → apply rule ($R_{2,1}$, $R_{2,4}$, $R_{2,2}$, $R_{2,3}$)
4. Loop
5. R ← First (Rules) returns $R_{2,4}$ (arrange according to the diagonal $R_{2,1}$, $R_{2,4}$, $R_{2,2}$, $R_{2,3}$)
6. Rules – { $R_{2,2}$, $R_{2,3}$ } (tail of the rules after reordering)
7. RDATA: Perform the action → Place X on cell(2,4)
8. Path ← backtracking (data)

Make a recursive call to top of algorithm:

1. Term (data) → No
2. Deadend → NO
3. Rule → apply rule ($R_{3,1}$, $R_{3,2}$, $R_{3,3}$, $R_{3,4}$)
4. Loop

   If NULL(Rules) → No

5. R ← first (Rules) ← $R_{3,1}$

6. Rules = {$R_{3,4}$, $R_{3,2}$, $R_{3,3}$ }
7. RDATA ← Place X on $R_{3,1}$
8. Path ← backtracking (data)

Make a recursive call to top of algorithm:

1. Term (data) → No
2. Deadend → No
3. Rule → apply rule ($R_{4,1}$, $R_{4,2}$, $R_{4,3}$, $R_{4,4}$)
4. Loop

   If NULL(Rules) → No
5. R ← first (Rules) ← $R_{4,2}$
6. Rules = {$R_{4,3}$, $R_{4,1}$, $R_{4,4}$ }
7. RDATA ← Place X on $R_{4,2}$
8. Path ← backtracking (data)
9. GOTO LOOP

1. Term (data) → No
2. Deadend (data) → yes → return fail → Path ← fail

   9. GOTO LOOP → Step 4

5. R ← $R_{4,3}$

6. Rules ← {$R_{4,1}$, $R_{4,4}$}

Realize the construction of path (using recursive calls) in procedure backtracking:

CONS (R, Path)

→ CONS ($R_{4,2}$, path)

→ $R_{4,2}$, CONS ($R_{3,1}$, path)

→ $R_{4,3,}$, $R_{3,1}$, $R_{2,4}$, $CONS\ (R_{1,2},\ path)$

→ ($R_{4,2}$, $R_{3,1}$, $R_{2,4}$, $R_{1,2}$)

References:

Nilsson, N. J. (1982). *Principles of artificial intelligence*. Springer Science & Business Media.