Vectorized Method for Solving the n-queens Problem using Bohrium
Larsen, Mads Ohm

Publication date:
2016

Document Version
Publisher's PDF, also known as Version of record

Citation for published version (APA):
Vectorized Method for Solving the \( n \)-queens Problem using Bohrium

Mads Ohm Larsen <ohm@nbi.ku.dk>

**Introduction**

On a normal \( 8 \times 8 \)-chessboard we find that 8 queens can be positioned in

\[
\binom{64}{8} - \binom{64}{n} = 4,426,165,308
\]

different legal ways. This is too many to brute force, especially as we go for larger \( n \).

For the \( 8 \times 8 \)-queens problem we only have 92 distinct solutions.

**Backtracking Algorithm**

Since only one queen are allowed in each row and column, we can skip placing another queen in the same column or row in the backtracking algorithm.

![Image 1: The 8 \times 8 "Identity" chessboard.](image)

If we do the same calculations for \( n = 4 \) as we did for \( n = 8 \), we get that we have

\[
\binom{16}{4} = 1,820
\]

legal configurations of the board, which can be boiled down to

\[
4^4 = 256
\]

placements where we only check the diagonals, but only \( 4^2 = 16 \) permutations of rows.

The \( 4 \times 4 \) "identity" chessboard, which is shown in figure 2, can be built similar to figure 1.

![Image 2: The 4 \times 4 "Identity" chessboard.](image)

For the four-queens problem there are only two solutions. These are shown in figure 3.

![Image 3: The 2 solutions for the four-queens problem.](image)

These two are reflections (Figure 4) or rotations (Figure 5) of each other.

**Using Permutations**

Instead of looking for the right combinations, we can instead compute permutations of rows with a queen in each column. There are

\[
8! = 40,320
\]

permutations of a \( 8 \times 8 \)-chessboard’s rows.

Because we construct the rows with a queen in each column, we now know we only have to check the diagonals, to see if the current permutation is a solution.

![Image 4: Reflection](image)

![Image 5: Rotation](image)

**Doing so yields**

\[
4 \times 4 \# = 256
\]

boards, which all needs to be checked. Generalizing for the \( n \)-queens problem, this would be

\[
u^2 - n!
\]

For the four-queens problem, this yields \( 4^2 = 16 \) traces of which we only need to count the number of 1. As we have seen already, there is only two solutions, and we only have two max-traces which are 1.

Note that since there is always queens in the diagonals or offset-diagonals, we can never have a max-trace of zero value.

**Using NumPy and Bohrium**

With our knowledge, we can construct a simple Python program using NumPy or Bohrium to calculate how many distinct solutions are to the \( n \)-queens problem for a given \( n \). In this program we simply list all permutations of the \( n \times n \)-chessboard, flip it, then trace all diagonals from \(-n \) to \( n \) and max that matrix into a 10 vector. The result is then found by counting is in this vector.

**Future Work**

Unfortunately we do not currently gain any performance boost using Bohrium, however this is largely due to the creation of the permutation boards.

We are planning to implement a permutation generator into the Bohrium runtime as a streaming generator for the GPU, and will hopefully get a performance speed-up doing so.

```python
import numpy as np
from itertools import permutations

import numpy as np
from numba import jit

@jit(nopython=True)
def nqueens(n):
    # Generate all permutations of n identity boards
    boards = list(permutations(np.ones((n, n))
    # Attach the flipped boards as another dimension
    rotation_boards = np.array((boards, np.flip(boards)))
    # Calculate all the traces, from -n to n for all the boards
    n = np.max([
        rotation_boards
        # for each board in the board
        np.trace(
            rotation_boards, n)
            # for each axis in range(-n, n)
            np.max(axis=(i, i))
    # Count the number of 1s in the maximum of the traces
    print(np.sum(n == 1)), "solutions for", n, "by", n, "board"

nqueens(8) # => 92 solutions for 8 by 8 board.
```