30.08.2017

Oskar Leirvåg

Student@Uib

The 8-queens validation algorithm:

A Complete Analysis of two possible solutions for the validation algorithm

//Check all  
for (int x = 0; x < 7; x++) {  
 clone1 = (mask2 & clone1) >>> 7;  
 clone2 = (mask1 & clone2) >>> 9;  
 clone3 = (mask1 & clone3) >>> 1;  
 clone4 >>>= 8;  
 if (

(clone1 & grid) != 0L

|| (clone2 & grid) != 0L

|| (clone3 & grid) != 0L

|| (clone4 & grid) != 0L

)

return false;  
}

Innholdsfortegnelse

[The problem: 2](#_Toc491880354)

[The Object-oriented solution: 3](#_Toc491880355)

[The Bitwise solution: 3](#_Toc491880356)

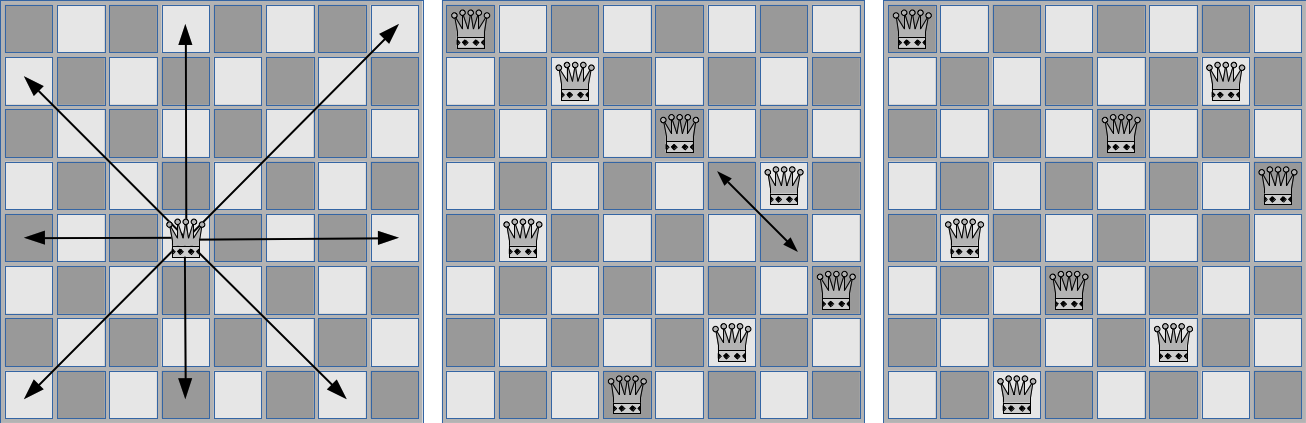
[Performance: 4](#_Toc491880357)

[Memory usage: 5](#_Toc491880358)

[Conclusion: 5](#_Toc491880359)

Analysis of the 8queens validation algorithm:

# The problem:

In the eight queens puzzle, eight queens must be placed on a standard 8×8 chess board so that no queen can attack another. The center figure below shows an invalid solution; two queens can attack each other diagonally. The figure on the right shows a valid solution. Given a description of a chess board, your job is to determine whether or not it represents a valid solution to the eight queens puzzle.

Taken from [Open.kattis.com](https://open.kattis.com/problems/8queens)

Given all these rules we are given a single string of 64 (8\*8) that represents the grid with ‘.’ For empty tile and ‘\*’ for queen:

\*..............\*.....\*..........\*.\*........\*.........\*...\*....\*.....

/ 1 2 3 4 5 6 7 8

1 \* . . . . . . .  
2 . . . . . . \* .  
3 . . . . \* . . .  
4 . . . . . . . \*  
5 . \* . . . . . .  
6 . . . \* . . . .  
7 . . . . . \* . .  
8 . . \* . . . . .

# The Object-oriented solution:

The object oriented solution is to create a grid object which is 2 dimensional but saves the values in a one dimensional char[] array. This solution is also easy to abstract is such a way that it can check all N-queens problems.

In this way, the main method body only needs to read in the string and add everything to a grid, then return/print the result of the ***queenGrid.checkGrid()*** function.

Although this solution is easy to understand and in use, it is not the quickest way to iterate through all the combinations there are.

private boolean OOGridCheck(String g) {  
 queenGrid grid = new queenGrid(8, 8, '\*');  
  
 for (int i = 7; i >= 0; i--) {  
 char[] line = g.toCharArray();  
 for (int x = 0; x < 8; x++)  
 if (line[x] == '\*') {  
 grid.setPos(x, i, 'q');  
 break;  
 }  
 }  
 return grid.checkGrid();  
}

# The Bitwise solution:

The bitwise solution is to represent the grid of 8\*8=64 tiles in a long which is 64 bits, where a 0-bit = empty tile and a 1-bit = a queen. In this way, we can check the whole board in 7 iterations, using only bit shift ‘>>>’

This results in smaller memory usage, less iterations and comparisons, also the bitwise operands in java is way faster for a CPU to calculate. This is possibly the most optimized we can make this.

private boolean BitGridCheck(String g) {  
 long grid = 0x00;  
 int count = 0;  
  
// Reads inn all values and adds them to a long for bitwise array  
 for (byte row = 7; row >= 0; row--) {  
 char[] line = g.toCharArray();  
 for (byte x = 7, y = 0; x >= 0; x--, y++)  
 if (line[y] == '\*') {  
 grid |= (1L << (row \* 8) + x);  
 count++;  
 break;  
 }  
 }  
  
 return count == 8 && *checkBitwiseGrid*(grid);  
 }

# Performance:

As both methods are T(o) = x \* N, the only factor is iterations and comparisons in the functions (x). Although this is of little matter in smaller runs, we can see a huge increase in difference in huge tests. If we generate 1,2,4,8 and 16E6 random grids and feed them into both methods we get this graph:

Doing a log (T(o)) = log (x \* N) gives us a completely different perspective:

We can only assume form testing that the sudden jump from 4E6 iterations is that the grid-list becomes too large for our processor cache, and therefore we get RAM-delay. Although this should not affect the result since it happened in both cases.

# Memory usage:

While performing a single validation these algorithms will create an interpretation of the grid as an object. That is either a long or a queenGrid object. During a worst-case scenario (a valid grid requires the entire algorithm to finish) these datatypes are created in bytes:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **queenGrid - Obj size:** | **Long size:** |  |  | Datatypes size |
| 200 | 56 |  | int | 32 |
|  |  |  | long | 64 |
|  |  |  | char | 16 |
|  |  |  |  |  |
|  |  |  |  |  |

A **queenGrid** will consist of a char[64], 8 position objects = (int x, int y), 2 int for width and height. This will result in 200Bytes. Meanwhile the **Bit shift** solution will only use 1Long + 4 copies + 2 masks which results in only 56Bytes.

# Conclusion:

Both methods have their usage, although the Bit Shift solution is faster the object oriented can still solve N-queens. If we are to brute-force all possible iterations for the placements to find all valid solutions the Bit-Shift solution will be the better option due to smaller memory usage, and higher performance. Although taking a look at possible permutations:

|  |  |  |  |
| --- | --- | --- | --- |
| **Total possible** | **Number if linecheck** | | **line and column** |
| 178 462 987 637 760,00 | 16777216 | 40320 | |

We would save immense time if we had to iterate trough all possible grids, but if we only generate with one queen on each line we go from (64 \* 63 \* 62 . . \* 57) to only (8 ^ 8) solutions. And by excluding columns too we only get 40320 possible combinations. And this will save only 30ms runtime.