Fixing N-Queen Problem

Author: Ji Yang

Problem:

This paper focusses mainly on the development and optimization of a solution to the Fixing N-Queen problem, an extension of the N-Queen problem. N-Queen problem describes a problem where given an N x N chess board and N number of queens with given board coordinates. Find the minimum number of a moves required such that the resulting board does not have a queen attacking another queen.

This paper will explore possible optimizations behind the solution for the fixing N-Queen problem. Finding, if any a fast algorithm to solve large boards within reasonable time. This paper also explores numerous optimizations that are possible to speed up the solver.

Optimization Methods Include:

1. Adding SIMD before major inner loops
2. Scaling 64-bit integer to 8-bit integer
3. Reducing branching factor
4. Reducing depth

Language:

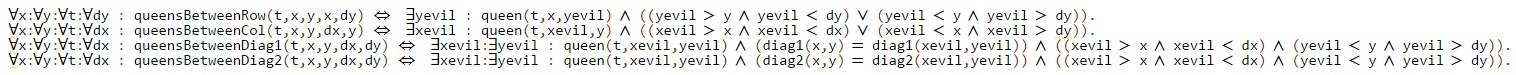
Due to unfathomable reasons IDP proved to be very unreliable encountering error after error upon compilation, so Julia has been selected as a backup language in cases of system compilation failures. In Julia, all constraints has been written as Boolean functions, and a function named “solver” will be called to solve the constraints and return an integer which will be the least number of moves required to solve the problem. The main reason Julia was picked to solve the problem is due to its fast Boolean computation nature, and its flexibility to generate required solutions. Due to the nature of the problem, computations of large N’s will be very slow and computationally heavy, thus a fast program like Julia is a reasonable choice.

Initial Solution (IDP):

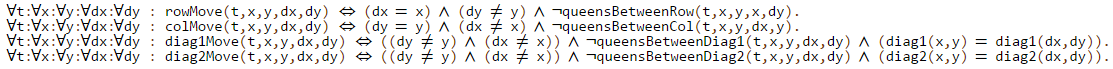
The initial solution is built upon the example of an N-Queen problem solution

Constraints for a legal move:

1. The set of constraints below as name suggests are satisfied if there exist a queen between two points of a queen move, a source and a destination. Diag1 represents y = -x diagonal, and Diag2 represents y = x diagonal.



1. The row move and column move constrain are satisfied if it is moved along a row and a column respectively, AND there does not exist a queen between source and destination. The two diagonal constraints are satisfied if both row source and column source are different from its respective destinations AND along the sources’ diagonal, AND there does not exist a queen between source and destination.



1. All the above constraints can be linked together to create a valid move constraint.



1. Checking for a safe board was copied directly from IDP’s N-Queen example.



1. A queen present at x, y at time t means that it didn’t move and was there before or it has been moved there.



1. Only 1 queen can move at a time.



Initial board must be >=4 or there are no solutions, input value N is incremented to test the speed of the solution in big-O. Since not all solutions can be obtained within N moves, and to cap the maximum move count so that it does not run indefinitely. Round (N\*1.5) has been chosen because It caps the limit such that it does not search excessively for solutions, but is large enough that the output will be <= Round (N\*1.5).

Results (Julia):

A problem specific program has been coded in Julia for benchmark testing.

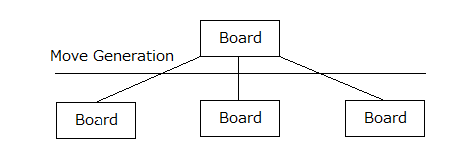
|  |  |
| --- | --- |
| Board Type | Average Runtime Over 5 Trials (Seconds) |
| 4 x 4 Solved | 0.063 |
| 5 x 5 Solved | 0.064 |
| 6 x 6 Solved | 0.064 |
| 7 x 7 Solved | 0.065 |

By the benchmark progression of solving an already solved board, increasing N does not increase time takes to solve the board. The slight increase in time is likely due to larger boards, where it has to check every piece to ensure that the board is safe. Therefore a very large solved board will still be extremely fast to solve.

Unsolved boards have very large difference in run times. For example, a 4 x 4 unsolved board can have run time between 0.110 and 0.175 seconds based on the number of moves required to solve the board. The difference is even larger on larger boards due to larger possible number of moves required to solve the board. Due to the nature of extremely wide branching factor, it will take extremely long to solve large boards. For the sake of experimentation, all unsolved boards require only one column move to solve.

|  |  |
| --- | --- |
| Board Type | Average Runtime Over 5 Trials (Seconds) |
| 4 x 4 Unsolved | 0.133 |
| 5 x 5 Unsolved | 0.623 |
| 6 x 6 Unsolved | 7.028 |
| 7 x 7 Unsolved | 31.869 |

The written program will differ from computer to computer based on the hardware of the testing computer. Since the program does not require memory allocation, due to the recursion nature of the solver, RAM usage is minimal, which means that deeper searches will not cause memory overflow, and only time will be affected. The program does not use a tree structure, but rather a pseudo N-ary tree, as it generates possible moves for a particular board state, and its children are the next board state based off the generated moves, like such:



To limit the number of searches, depth is capped at Round (N\*1.5) for subtrees with no solutions and depth will be slowly updated and reduced with the current smallest number of possible moves that can solve the problem. Move generation uses the list of constraints implemented from the initial solution using Boolean functions in Julia (Julia file will be attached). Checking for a safe board state requires checking individual pieces for safety. A piece is safe if and only if it is the only piece on that row, column, Diag1, and Diag2, therefore the pseudo code for safeBoard is as follows:

function safeBoard(board):

counter = 0

for row = 1:n

for col = 1:n

if queen(row, col)

if safePiece(row, col)

counter++

if counter == N; return true

return false

With such, the pseudo code for the solver:

function solver(board, depth, limit):

if safeBoard(board); return depth // returns least number of moves to solve the board

if depth >= limit; return limit // caps depth at a limit = Round (N\*1.5)

for row = 1:n

for col = 1:n

if queen(row, col)

location = moveGeneration(row, col) // generates all possible moves for that piece

for i = 1:length(location)

nextboard = simulate location[i] // simulates that move on a virtual board

limit = solver(nextboard, depth+1, limit) // recursive call

return limit

Every time solver manages to find a solution, it returns the current depth of the solved board and updates it as the new limit, so that newer solutions must be within the limit indicating lesser move required to solve the board. This allows the program to self-consciously update its own depth without modification from programmer. This will run much faster than a full depth tree, because it cuts down the number of recursion after every successful solution.

Optimization Methods:

1. Using SIMD, proved to be ineffective as it actually increased run time. A possible explanation is that the program does not actually utilize vectors, as SIMD is mainly used for vector operations. With SIMD, result follows:

|  |  |
| --- | --- |
| Board Type | Average Runtime Over 5 Trials (Seconds) |
| 4 x 4 Solved | 0.071 |
| 5 x 5 Solved | 0.071 |
| 6 x 6 Solved | 0.073 |
| 7 x 7 Solved | 0.075 |

Same sort of occurrence happened with unsolved boards.

Ex 1) Given: n = 4, row = [1, 1, 2, 4], col = [2, 3, 1, 3]

Without SIMD: 0.167s, with SIMD: 0.175s

Ex 2) Given: n = 5, row = [1, 2, 3, 5, 5], col = [1, 4, 3, 2, 5]

Without SIMD: 1.280s, with SIMD: 1.300s

1. Scaling data from 64-bit integer to 8-bit integer also increased calculation time. A possible explanation is that the compiler needed to time to determine individual types of each integer.

|  |  |
| --- | --- |
| Board Type | Average Runtime Over 5 Trials (Seconds) |
| 4 x 4 Solved | 0.204 |
| 5 x 5 Solved | 0.205 |
| 6 x 6 Solved | 0.207 |
| 7 x 7 Solved | 0.208 |

Ex 1) Given: n = 4, row = [1, 1, 2, 4], col = [2, 3, 1, 3]

Without Scale: 0.167s, with Scale: 0.823s

Ex 2) Given: n = 5, row = [1, 2, 3, 5, 5], col = [1, 4, 3, 2, 5]

Without Scale: 1.280s, with Scale: 3.242s

1. Reducing width: Run time is mainly determined by the solver and its recursion. The branching factor is the number of possible moves of that board state, and its individual children is the board state after one move. The children needs to recalculate the whole board to determine all possible moves, but since a lot of moves don’t really have meaning because it is either a repeated move or a move that will not guarantee a safe board. If these moves can be ignored during the recursion, the branching factor would be significantly reduced.
2. Reducing depth: Another run time determinant is the depth the pseudo tree travels. The initial subtrees without solutions would exhibit a depth of Round (N\*1.5), if this number can be reduced further, run time will also decrease for unsolved boards.

Ex 1) Given: n = 4, row = [1, 1, 2, 4], col = [2, 3, 1, 3]

depth = Round (N\*1.5): 0.167s, depth = N: 0.145

Ex 2) Given: n = 5, row = [1, 2, 3, 5, 5], col = [1, 4, 3, 2, 5]

depth = Round (N\*1.5): 1.280s, depth = N: 0.842s

Conclusion:

Even with good algorithms to optimize branching factor and depth, it is still impossible to solve large boards within reasonable time frame on a regular quad-core computer. Solver is required to travel full width and full depth before it finds a solution reducing the depth, which largely impacts search time. A subtree without a solution requires (Number of moves) ^depth amount of searches.

Reference:

Problem: <http://picat-lang.org/lp_cp_pc/FixQueens.html>

System: <https://dtai.cs.kuleuven.be/software/idp>

Partial Solution: <https://dtai.cs.kuleuven.be/krr/idp-ide/?example=basic/5-Nqueens>

Julia: <http://julialang.org/>

Benchmarking: <https://github.com/JuliaCI/BenchmarkTools.jl>

Received huge help from Karol Swietlicki