n-Queens in Haskell

Luke Thomas

January 4, 2016

Abstract

This is a small test of Literate Programming. It solves the n-queens problem for $n = 8$. The n-queens problem is as follows: Find an arrangement of n queens on an $n \times n$ chessboard such that no two queens threaten one another.

1 Strategy

We first note that each queen must solely inhabit one file, therefore there must be one queen in each file given that there are the same number of queens as files. Any solution can thus be represented as an offset into each file.

We interpret the board configuration as a base-$n + 1$ integer where the absence of a queen is a zero, and the presence of a queen on rank n is an n digit where the a file is the most significant digit. Hereafter "valid" means that a configuration contains no threats. Hereafter all references to "counting up" or "adding one" refer to this interpretation of a configuration as an integer.

By counting up through all base-$n + 1$ numbers of n digits we will thus consider all correct solutions.

However if we find that a given value of the number, which contains tailing zeros, is invalid then we know that all other numbers with that prefix are also invalid, and so we can skip them. Thus we accelerate the process of counting up.

2 Types

We'll need to convert Chars to and from their numerical representations for the purpose of incrementing and decrementing files. We use Maybe values because its not always possible to find the next valid configuration, or the next square in a given direction and doing so moves the complexity out of the function that has to deal with these problems into higher-level functions better equipped to solve the problem.

Ranks are in $['a'..'h']$ (lower-case) and files in [1..8], thus the form of this tuple is $(rank, file)$. $a1$ is the lower left square, and a8 the upper left square.

```
type ChessPosition = (Char, Int)
```

The head element will always be the rank of the queen on the rightmost file, this is true recursively of the list's tail. A configuration may have ≤ 8 and ≥ 0 queens.

```
type Configuration = [ChessPosition]
```

3 The Solutions

numSols is the length of the list containing all solutions to the 8-queens problem.

```
numSols :: Int
numSols = length solutions
```

solutions is a list of all solutions which can be arrived at by adding queens to an empty board.

```
solutions :: [Configuration]
solutions = solutionsFollowing []
```

4 solutionsFollowing

solutionsFollowing c is a list of all solutions which can be arrived at by adding queens to the configuration described by c, without removing any specified by c. It is arrived at by finding all configurations that can be thus arrived at from c which contain no threats and then filtering this list to only those elements of it with length eight.

Since all solutions are of length eight this discards everything but correct solutions from the list of states that can be arrived at from c and don't contain a threat.

```
solutionsFollowing :: Configuration → [Configuration]
solutionsFollowing c = filter ((8==).length) $ allValidConfsFollowing c
```

5 allValidConfsFollowing

Finds all valid configurations that can be arrived at solely by adding queens to the configuration described by c. nv is the next valid configuration that can be arrived at by counting upwards from the given configuration c.

We enumerate all valid configurations following from c by checking if $next$ is a valid configuration, in which case we add it to the list and recurse, and in the absence of a next valid configuration we return the empty list because valid configurations are found in ascending order (because nextValid counts up) and thus there are no unfound valid configurations.

```
allValidConfsFollowing :: Configuration → [Configuration]
allValidConfsFollowing c
    | nv == Nothing = []
    | otherwise     = fromJust nv : (allValidConfsFollowing $ fromJust nv)
    where nv        = nextValid c
```

6 nextValid

$nextValid$ finds the next valid configuration following c, as previously defined in the explanation of the $allValidConfsFollowing$ function.

First we add either a queen (whenever possible) or one to the present configuration (since it need not be considered itself as we're looking for the first valid configuration following it), and name this next (potentially invalid) configuration $next$. If $next$ is valid then we simply return it.

The reason we favour adding a queen is because no digit can be a zero in a correct solution so there's no point considering Configurations derived from one which is missing a queen.

If the next configuration is invalid, then it isn't worth considering configurations derived by adding more queens to it since they can only maintain or increase the number of threats but never reduce them.

Therefore naively adding one to $next$ is a waste of time and we should instead skip to the first Configuration without $next$ as a prefix. Specifically: $skippedNext$ is the first Configuration worth checking if next is invalid, either the rightmost file has changed (relative to $next$), or one left of it (relative to $next$).

If $skippedNext$ is valid then we simply return it, otherwise we return the first valid configuration following from it. If there is no such valid configuration arising from it then next takes on the value $Nothing$ which is considered valid and thus we return through the first case where $next$ is valid avoiding the call to fromJust.

This function in combination with the concept of viewing the board as a base-$n+1$ integer is the core of this algorithm.

```
nextValid :: Configuration → Maybe Configuration
nextValid c
    | valid next        = next
    | valid skippedNext = skippedNext
    | otherwise         = nextValid ∘ fromJust $ skippedNext
    where next          = advance c
          skippedNext   = advanceWithoutAdding ∘ fromJust $ next
```

7 valid

A configuration is said to be valid iif no two queens on the board threaten one another. This code relies on threatening being symmetric, which is trivial to see.

It first checks if the rightmost queen threatens any other queen, and if so returns false; if the rightmost queen poses no threat then no other queen threatens her and so she can be removed from the board. The remaining queens are then considered alone. If there are zero or one queens then there cannot exist a threat and so we return true.

This is achieved by first checking for zero (*Nothing* pattern) queens or a configuration consisting of just one queen and returning true if either of these are the case.

Otherwise we partially apply *threat* to the rightmost queen to get a function (*threat* x) which determines whether the rightmost queen threatens another. This function is mapped over the list of all other queens to produce a list of Bools indicating whether the respective queen is threatened by the rightmost queen. If any element of this list is true then the rightmost queen threatens at least one other queen. We determine this by folding the list with ||. We then return "does the rightmost queen threaten any other?" && "is the configuration sans the rightmost queen valid?"

We have to wrap the call to ourself in a *Maybe* because other functions will be passing us potentially *Nothing* values.

```
valid :: Maybe Configuration → Bool
valid Nothing       = True
valid (Just [x])    = True
valid (Just (x:xs)) = not (foldl1 (||) (map (threat x) xs))
                      && valid (Just xs)
```

8 advance

This adds 1 to the given Configuration until we get to the next Configuration which is worth considering without taking validity into account.

First we check if we're attempting to increment the empty Configuration, if so then we return $(a, 1)$ since it's required that each digit be non-zero and 1 is the least digit. We add it to the most significant digit because there's no point considering configurations which lack a queen in the first file as they'll either be invalid or have less than eight queens. The Configurations we consider increase monotonically and so we know that by jumping to $(a, 1)$ we skip no potential solutions.

If the configuration is less than eight in length then the rightmost digits of it are zeros. In this case to advance we just return the same configuration with a queen added to the right in rank one. This doesn't actually add one to the Configuration because there's no point even considering Configurations derived from those containing a zero as they cannot contain eight queens each on different files.

If all queens are in rank eight, then there is no next position and so we return Nothing. We determine whether all the queens are at the top by mapping a function which returns true iif its argument is in the eighth rank over all queens and then folding it with &&. Note that the x that is the argument to the allAtTop function shadows the x from the pattern match.

If there are eight queens on the board and the rightmost is not at rank eight ($x < 8$) then we advance the rightmost queen (least significant digit) up a rank. The rationale for not advancing a queen on file eight being the same as before.

If there are eight queens and the rightmost queen in in rank eight (*otherwise*) then we call *advanceWithoutAdding* on the configuration sans the rightmost queen. The effect of this is the same as this function except that it will never add a new queen; that is to say it increments the least amount without reducing the number of trailing zeros, though it may increase them.

```
advance :: Configuration → Maybe Configuration
advance [] = Just [('a',1)]
advance conf@((file,x):xs)
    | length conf < 8 = Just $ (nextFile file, 1) : conf
    | allAtTop conf   = Nothing
    | x < 8           = Just $ (file, x+1) : xs
    | otherwise       = advanceWithoutAdding xs
```

```
    where allAtTop x  = foldl1 (&&) $ map (λ(_,r) → r == 8) x
```

9 advanceWithoutAdding

This increments the Configuration the least amount possible without decreasing the number of trailing zeros. This function is used when we've already ruled out all Configurations with the given Configuration as a prefix and so need to change the given configuration rather than just deriving a new configuration from it.

It works much like the advance function, but it never adds a new queen. If the rightmost queen is in a rank less than the eighth then we just move her up. If she is in rank eight then the remove her and recurse.

If we bottom out at a configuration with one queen in the eighth rank then we just return *Nothing* as there is no Configuration meeting our requirements.

If we are called with the empty list then since we can't add a queen we just have to return the empty list.

```
advanceWithoutAdding :: Configuration → Maybe Configuration
advanceWithoutAdding ((file,rank):xs)
    | rank < 8      = Just $ (file,rank+1):xs
    | length xs > 0 = advanceWithoutAdding xs
    | otherwise     = Nothing
advanceWithoutAdding [] = Just []
```

10 threat

"Does a queen at a threaten a queen at b?"

We first get the list of all possible locations b could move to in one turn, and we check whether any of the positions are equal to the position of a. If so then there is a threat and we return True, otherwise we return False.

```
threat :: ChessPosition → ChessPosition → Bool
threat a b = any (== a) (possibleLocations b)
```

11 possibleLocations

This generates a list of all locations a queen at the given position could move to in one turn, assuming an otherwise empty board (this includes the space she currently occupies). This is done by generating a list of each in-file move she could make, each in rank move she could make, and each diagonal move she could make and then folding these lists with (++).

The in-file, and in-rank moves are just list comprehensions that vary one part of her coordinate. The diagonal moves are delegated to dedicated functions. Keeping these list comprehensions simple and easy to read is the motivation for including the square she presently occupies in the list of possible moves.

```
possibleLocations :: ChessPosition → [ChessPosition]
possibleLocations (file, rank) = foldl1 (++) [
    [(file, r) | r ← [1..8]],
    [(f, rank) | f ← ['a'..'h']],
    allUpLeftOf (file, rank),
    allUpRightOf (file, rank),
    allDownLeftOf (file, rank),
    allDownRightOf (file, rank)]
```

12 allXYOf

These functions generate all squares in a particular diagonal direction from the square given as the argument.

Since they are all simply minor variations of the same function (kept separate for readability) only $allUpLeftOf$ will be explained.

If we're in the leftmost file then there are no squares left of us and so we return the empty list. Similarly if we are in the eighth rank then there are no squares above us and we return the empty list. If, otherwise, we are elsewhere then we return the next square in the up-left direction and recurse to find all squares up-left of this next square in the up-left direction.

We find the next square in the up-left direction by decrementing the file and incrementing the rank.

```
allUpLeftOf :: ChessPosition → [ChessPosition]
allUpLeftOf (f, r)
    | f == 'a'  = []
```

```
    | r == 8    = []
    | otherwise = next : allUpLeftOf next
    where next  = (prevFile f, r+1)

allUpRightOf :: ChessPosition → [ChessPosition]
allUpRightOf (f, r)
    | f == 'h'  = []
    | r == 8    = []
    | otherwise = next : allUpRightOf next
    where next  = (nextFile f, r+1)

allDownLeftOf :: ChessPosition → [ChessPosition]
allDownLeftOf (f, r)
    | f == 'a'  = []
    | r == 1    = []
    | otherwise = next : allDownLeftOf next
    where next  = (prevFile f, r-1)

allDownRightOf :: ChessPosition → [ChessPosition]
allDownRightOf (f, r)
    | f == 'h'  = []
    | r == 1    = []
    | otherwise = next : allDownRightOf next
    where next  = (nextFile f, r-1)
```

13 nextFile & prevFile

nextFile gets the file to the right of the given file. We convert the Char representing the file to its numerical representation and then increment this before converting it back to a Char. This works because the letters are represented as contiguous integers, and we never deal with file z and define the predecessor of a as undefined. The file following h is undefined.

prevFile works similarly to *nextFile*.

```
nextFile :: Char → Char
nextFile 'h' = undefined
nextFile f   = chr $ ord f + 1

prevFile :: Char → Char
```

```
prevFile 'a' = undefined
prevFile f   = chr $ ord f - 1
```