Nonogram

From HaskellWiki

(Redirected from 99 questions/Solutions/98)

Here are some solvers for Nonogram puzzles. A description of what a nonogram is can be found in Ninety-Nine Haskell Problems.

1 Backtracking solver

The first solution is a simple backtracking algorithm, but is quite slow for larger problems.

```
data Square = Blank | Cross deriving (Eq)
instance Show Square where
    show Blank = " "
    show Cross = "X"

-- create all possibilities of arranging the given blocks in a line of "n" elements
rows n []             = [replicate n Blank]
rows n (k:ks) | n < k = []
rows n (k:ks)         =
    [Blank : row | row <- rows (n-1) (k:ks)] ++
    if null ks then [replicate k Cross ++ replicate (n-k) Blank]
               else [replicate k Cross ++ Blank : row | row <- rows (n-k-1) ks]

-- contract a given line into the block format
-- i.e. contract [Cross,Blank,Cross] == [1,1]
contract = map length . filter (\(x:_) -> x==Cross) . group

-- create all solutions by combining all possible rows in all possible ways
-- then pick a solution and check whether its block signature fits
solver horz vert = filter fitsVert possSolution
    where possSolution = mapM (rows (length vert)) horz
          fitsVert rs  = map contract (transpose rs) == vert

-- output the (first) solution
nonogram horz vert = printSolution $ head $ solver horz vert
    where printSolution = putStr . unlines . map (concatMap show) . transpose
```

This is a solution done for simplicity rather than performance. It's SLOOOOW. If I were to check intermediate solutions against the blocks that should come out instead of sequencing everything, the List monad would fail much earlier...

It builds all combinations of blocks in a row (stolen from solution 2 :) and then builds all combinations of rows. The resulting columns are then contracted into the short block block form and the signature compared to the target.

2 Deducing solver

We can make the search much faster (but more obscure) by deducing the values of as many squares as possible before guessing, as in this solution:

```
module Nonogram where

import Control.Monad
import Data.List
import Data.Maybe

type Row s = [s]
type Grid s = [Row s]

-- partial information about a square
type Square = Maybe Bool

-- Print the first solution (if any) to the nonogram
nonogram :: [[Int]] -> [[Int]] -> String
nonogram rs cs = case solve rs cs of
        [] -> "Inconsistent\n"
        (grid:_) -> showGrid rs cs grid

-- All solutions to the nonogram
solve :: [[Int]] -> [[Int]] -> [Grid Bool]
solve rs cs = [grid' |
                        -- deduce as many squares as we can
                grid <- maybeToList (deduction rs cs),
                        -- guess the rest, governed by rs
                grid' <- zipWithM (rowsMatching nc) rs grid,
                        -- check each guess against cs
                map contract (transpose grid') == cs]
  where nc = length cs
        contract = map length . filter head . group

-- A nonogram with all the values we can deduce
deduction :: [[Int]] -> [[Int]] -> Maybe (Grid Square)
deduction rs cs = converge step init
  where nr = length rs
        nc = length cs
        init = replicate nr (replicate nc Nothing)
        step = (improve nc rs . transpose) <.> (improve nr cs . transpose)
        improve n = zipWithM (common n)
        (g <.> f) x = f x >>= g

-- repeatedly apply f until a fixed point is reached
converge :: (Monad m, Eq a) => (a -> m a) -> a -> m a
converge f s = do
        s' <- f s
        if s' == s then return s else converge f s'

-- common n ks partial = commonality between all possible ways of
-- placing blocks of length ks in a row of length n that match partial.
common :: Int -> [Int] -> Row Square -> Maybe (Row Square)
common n ks partial = case rowsMatching n ks partial of
        [] -> Nothing
        rs -> Just (foldr1 (zipWith unify) (map (map Just) rs))
  where unify :: Square -> Square -> Square
        unify x y
```

```
          | x == y = x
          | otherwise = Nothing

-- rowsMatching n ks partial = all possible ways of placing blocks of
-- length ks in a row of length n that match partial.
rowsMatching :: Int -> [Int] -> Row Square -> [Row Bool]
rowsMatching n [] [] = [[]]
rowsMatching n ks [] = []
rowsMatching n ks (Nothing:partial) =
        rowsMatchingAux n ks True partial ++
        rowsMatchingAux n ks False partial
rowsMatching n ks (Just s:partial) =
        rowsMatchingAux n ks s partial

rowsMatchingAux :: Int -> [Int] -> Bool -> Row Square -> [Row Bool]
rowsMatchingAux n ks False partial =
        [False : row | row <- rowsMatching (n-1) ks partial]
rowsMatchingAux n [k] True partial =
        [replicate k True ++ replicate (n-k) False |
                n >= k && all (/= Just False) front && all (/= Just True) back]
  where (front, back) = splitAt (k-1) partial
rowsMatchingAux n (k:ks) True partial =
        [replicate k True ++ False : row |
                n > k+1 && all (/= Just False) front && blank /= Just True,
                row <- rowsMatching (n-k-1) ks partial']
  where (front, blank:partial') = splitAt (k-1) partial

showGrid :: [[Int]] -> [[Int]] -> Grid Bool -> String
showGrid rs cs ss = unlines (zipWith showRow rs ss ++ showCols cs)
  where showRow rs ss = concat [['|', cellChar s] | s <- ss] ++ "| " ++
                unwords (map show rs)
        showCols cs
          | all null cs = []
          | otherwise = concatMap showCol cs : showCols (map advance cs)
        showCol (k:_)
          | k < 10 = ' ':show k
          | otherwise = show k
        showCol [] = "  "
        cellChar True = 'X'
        cellChar False = '_'
        advance [] = []
        advance (x:xs) = xs
```

We build up knowledge of which squares must be filled and which must be blank, until we can't make any more deductions. Some puzzles cannot be completely solved in this way, so then we guess values by the same method as the first solution for any remaining squares.

3 Set based solver

By: Twan van Laarhoven

The idea behind this solver is similair to that of most Sudoku solvers, in each cell a set of its possible values are stored, and these sets are iterativly reduced until a single value remains. Instead of only using the possible values black and white this solver uses positions. If for some row the lengths [4,3] are given, then there

are 10 possible positions:

- white, left of both sections of black
- 4 positions inside the first black section.
- between the two black sections
- 3 positions inside the second black section.
- after both sections.

Each cell has a both a horizontal and a vertical set of possible positions/values.

There are two kinds of passes that are made:

- hStep: for each cell, it can only have values that can follow that of its left neighbour.
- efStep: If a cell is guaranteed to be white/black according to its horizontal value its vertical value must also be white/black, and vice-versa.

The hStep is applied in all four directions by reversing and transposing the board.

If no more progress can be made using this algorithm, the solver makes a guess. In the first cell that still has multiple choices all these choices are inspected individually by 'splitting' the puzzle into a list of puzzles. These are then solved using the deterministic algorithm. Puzzles that lead to a contradiction (no possible values in a cell) are removed from the list.

```haskell
module Nonogram where

import qualified Data.Set as Set
import Data.Set (Set)
import qualified Data.Map as Map
import Data.Map (Map)
import Data.List

--------------------------------------------------------------------------
-- Cells

-- | The value of a single cell
newtype Value = Value Int
        deriving (Eq, Ord, Show)

-- | Negative values encode empty cells, positive values filled cells
empty (Value n) = n <= 0
full = not . empty

type Choice = Set Value

--------------------------------------------------------------------------
-- Puzzle

type Grid = [[Choice]]

-- | Datatype for solved and unsolved puzzles
data Puzzle = Puzzle
        -- | List of rows, containing horizontal choices for each cell
```

```
        { gridH :: Grid
        -- | List of columns, containing vertical choices for each cell
        , gridV :: Grid
        -- | What is allowed before/after a specific value?
        --   (after (Value 0)) are the values allowed on the first position
        , afterH, beforeH :: [Value -> Choice]
        , afterV, beforeV :: [Value -> Choice]
        }

instance Eq Puzzle where
        p == q = gridH p == gridH q

instance Show Puzzle where
        show = dispGrid . gridH

-- | Transpose a puzzle (swap horizontal and vertical components)
transposeP :: Puzzle -> Puzzle
transposeP p = Puzzle
        { gridH      = gridV p
        , gridV      = gridH p
        , afterH     = afterV p
        , beforeH    = beforeV p
        , afterV     = afterH p
        , beforeV    = beforeH p
        }

-- | Display a puzzle
dispGrid = concatMap (\r -> "[" ++ map disp'' r ++ "]\n")
 where  disp'' x
         | Set.null     x = 'E'
         | setAll full  x = '#'
         | setAll empty x = '.'
         | otherwise      = '/'

--------------------------------------------------------------------------
-- Making puzzles

-- | Make a puzzle, when given the numbers at the edges
puzzle :: [[Int]] -> [[Int]] -> Puzzle
puzzle h v = Puzzle
        { gridH   = map (replicate cols . Set.fromList) ordersH
        , gridV   = map (replicate rows . Set.fromList) ordersV
        , afterH  = map mkAfter ordersH
        , beforeH = map mkAfter (map reverse ordersH)
        , afterV  = map mkAfter ordersV
        , beforeV = map mkAfter (map reverse ordersV)
        }
 where  rows = length h
        cols = length v
        ordersH = map order h
        ordersV = map order v

-- | Order of allowed values in a single row/column
--   Input = list of lengths of filled cells, which are separated by empty cells
--   Repeats empty values, because those values may be repeated
--   example:
--     order [1,2,3] = map Value [-1,-1, 1, -2,-2, 2,3, -4,-4, 4,5,6, -7,-7]
order :: [Int] -> [Value]
order = order' 1
 where  order' n []     = [Value (-n), Value (-n)] -- repeated empty cells allowed at the end
        order' n (x:xs) = [Value (-n), Value (-n)] ++ map Value [n..n+x-1] ++ order' (n+x) xs
```

```haskell
-- | What values are allowed after a given value in the given order?
mkAfter :: [Value] -> Value -> Choice
mkAfter order = (mkAfterM order Map.!)

mkAfterM order  = Map.fromListWith (Set.union) aftersL
 where  aftersL = -- after the start (0) the first non empty value, at position 2 is allowed
                  -- this is a bit of a hack
                  (if length order > 2
                    then [(Value 0, Set.singleton (order !! 2))]
                    else []) ++
                  -- after each value comes the next one in the list
                  zip (Value 0:order) (map Set.singleton order)

-------------------------------------------------------------------------
-- Classifying puzzles

-- | Is a puzzle completely solved?
done :: Puzzle -> Bool
done = all (all ((==1) . Set.size)) . gridH

-- | Is a puzzle invalid?
invalid :: Puzzle -> Bool
invalid = any (any Set.null) . gridH

-------------------------------------------------------------------------
-- Solving

-- | Solve a puzzle deterministicly, i.e. don't make any guesses
-- make sure
solveD :: Puzzle -> Puzzle
solveD = takeSame . iterate step

-- | All solving steps combined, the orientation after a step is the same as before
step = efStep . transposeP . hStep . transposeP . hStep

-- | A step in the solving process.
--   Propagate allowed values after from left to right
hStep p = p { gridH = gridH'' }
 where  gridH'  = zipWith hStepLTR (afterH  p) (gridH p) -- left to right
        gridH'' = zipWith hStepRTL (beforeH p) (gridH')  -- right to left

-- | hStep on a single row, from left to right, after is a function that gives the allowed af
hStepLTR after row = hStepLTR' (after (Value 0)) row
 where  hStepLTR' _          []     = []
        hStepLTR' afterPrev (x:xs) = x' : hStepLTR' afterX' xs
         where  x'      = Set.intersection x afterPrev
                afterX' = Set.unions $ map after $ Set.toList x'
-- | Same as hStepRTL, but from right to left, should be given allowed before values
hStepRTL before = reverse . hStepLTR before . reverse

-- | A step in the solving process
--   Combine horizontal and verticall grids, empty/full in one <-> empty/full in the oter
--   Note: we transpose gridV, to make it compatible with gridH (row-of-cells)
efStep puzzle = puzzle { gridH = gridH', gridV = transpose gridV't }
 where  (gridH', gridV't) = zzMap ef (gridH puzzle) (transpose (gridV puzzle))
        -- Step on a single cell
        ef h v = filterCell empty . filterCell full $ (h,v)
        -- Step on a single cell, for a single predicate, if either h or v satisfies the pred
        -- then the other is filtered so it will satisfy as well
        filterCell pred (h,v)
```

```
        | setAll pred h = (h, Set.filter pred v)
        | setAll pred v = (Set.filter pred h, v)
        | otherwise     = (h, v)

-------------------------------------------------------------------------
-- Guessing

-- | Solve a puzzle, gives all solutions
solve :: Puzzle -> [Puzzle]
solve puzzle
 | done    puzzle' = [puzzle'] -- single solution
 | invalid puzzle' = []        -- no solutions
 | otherwise       = concatMap solve (guess puzzle') -- we have to guess
 where puzzle' = solveD puzzle

-- | Split a puzzle into multiple puzzles, by making a guess at the first position with multi
--   we return all possible puzzles for making a guess at that position
guess :: Puzzle -> [Puzzle]
guess puzzle = map (\gh -> puzzle {gridH = gh} ) gridHs
 where gridHs = trySplit (trySplit splitCell) (gridH puzzle)

-- | Try to split a cell into multiple alternatives
splitCell :: Choice -> [Choice]
splitCell = map Set.singleton . Set.toList

-- | Try to split a single item in a list using the function f
--   Stops at the first position where f has more than 1 result.
--   TODO: A more soffisticated guessing strategy might be faster.
trySplit :: (a -> [a]) -> [a] -> [[a]]
trySplit f []     = []
trySplit f (x:xs)
 | length fx  > 1 = map (:xs) fx -- This element is split, don't look further
 | length fxs > 1 = map (x:) fxs -- The list is split furter on
 | otherwise      = []
 where  fx  = f x
        fxs = trySplit f xs

-------------------------------------------------------------------------
-- Utilities

-- | Set.all, similair to Data.List.all
setAll f = all f . Set.toList

-- | Map a function simultaniously over two lists, like zip
zMap :: (a -> b -> (c, d)) -> [a] -> [b] -> ([c], [d])
zMap f a b = unzip $ zipWith f a b

-- | Map a function simultaniously over two lists of lists, like zip
zzMap :: (a -> b -> (c, d)) -> [[a]] -> [[b]] -> ([[c]], [[d]])
zzMap f a b = unzip $ zipWith (zMap f) a b

-- | Find the first item in a list that is repeated
takeSame :: Eq a => [a] -> a
takeSame (a:b:xs)
 | a == b    = a
 | otherwise = takeSame (b:xs)

-------------------------------------------------------------------------
-- Test
```

Here is a test puzzle that can be used in the solver:

```
-- | A test puzzle
test = puzzle [[6],[3,1,3],[1,3,1,3],[3,14],[1,1,1],
               [1,1,2,2],[5,2,2],[5,1,1],[5,3,3,3],[8,3,3,3]]
              [[4],[4],[1,5],[3,4],[1,5],[1],[4,1],[2,2,2],
               [3,3],[1,1,2],[2,1,1],[1,1,2],[4,1],[1,1,2],
               [1,1,1],[2,1,2],[1,1,1],[3,4],[2,2,1],[4,1]]
```


Category:

- Code
