Sudoku Solving and Generation

Jan van Eijck
CWI & ILLC, Amsterdam

Specification and Testing, Week 5, 2014

Abstract

These slides document the implementation of a Sudoku puzzle solver, starting from a more or less formal specification, using constraint resolution and depth first tree search.

We make an excursion to the important topic of search trees, and depth first versus breadth first search algorithms.

Next, a random generator for Sudoku problems is constructed, by randomly deleting values from a randomly generated Sudoku grid.

The Sudoku setting will give us ample opportunity for specification and testing, and for discussing the choice of appropriate datatypes.

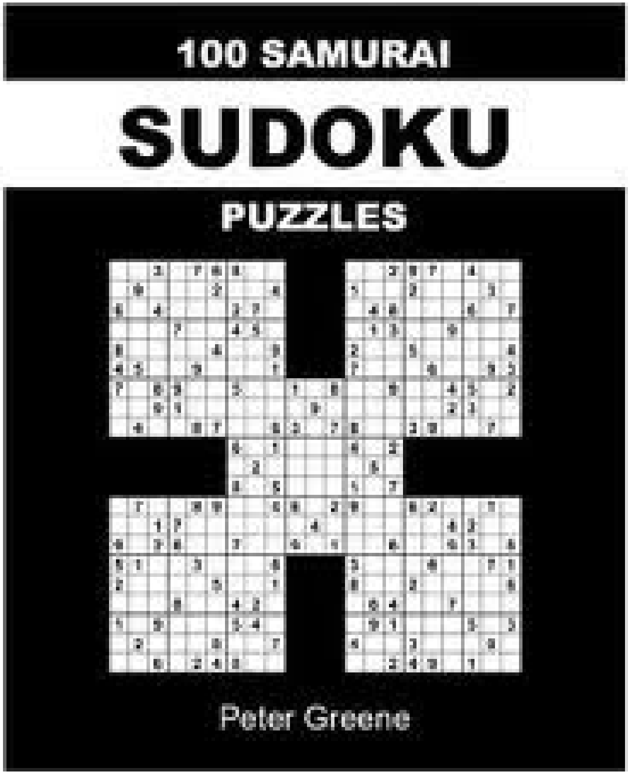
Key words:

Magic square, Sudoku grid, Sudoku puzzle, constraint solving, search trees, search algorithms, depth first search, breadth first search, problem generation, random search.

```
module SS where

import Data.List
import System.Random
```

Specifying Sudoku Solving



A Sudoku is a 9×9 matrix of numbers in $\{1, \ldots, 9\}$, possibly including blanks, satisfying certain constraints. A Sudoku problem is a Sudoku containing blanks, but otherwise satisfying the Sudoku constraints. A Sudoku solver transforms the problem into a solution.

You have already carried out the following exercise during a workshop session:

Exercise 1 Give a Hoare triple for a Sudoku solver. If the solver is called P, the Hoare triple consists of

$$\{\text{precondition}\}$$
$$P$$
$$\{\text{postcondition}\}$$

The precondition of the Sudoku solver is that the input is a correct Sudoku problem.

The postcondition of the Sudoku solver is that the transformed input is a solution to the initial problem.

State the pre- and postconditions as clearly and formally as possible.

If declarative specification is to be taken seriously, all there is to solving Sudokus is specifying what a Sudoku problem is. A Sudoku is a 9×9 matrix of numbers in $\{1, \ldots, 9\}$ satisfying the following constraints:

- Every row should contain each number in $\{1, \ldots, 9\}$
- Every column should contain each number in $\{1, \ldots, 9\}$
- Every subgrid $[i, j]$ with i, j ranging over $1..3$, $4..6$ and $7..9$ should contain each number in $\{1, \ldots, 9\}$.

A Sudoku problem is a partial Sudoku matrix (a list of values in the matrix). A solution to a Sudoku problem is a complete extension of the problem, satisfying the Sudoku constraints.

A partial Sudoku should satisfy the following constraints:

- The values in every row should be in $\{1, \ldots, 9\}$, and should all be different.
- The values in every column should be in $\{1, \ldots, 9\}$, and should all be different.
- The values in every subgrid $[i, j]$ with i, j ranging over $1..3$, $4..6$ and $7..9$ should be in $\{1, \ldots, 9\}$, and should all be different.

```
+-------+-------+-------+     +-------+-------+-------+
| 5 3   |   7   |       |     | 5 3 4 | 6 7 8 | 9 1 2 |
| 6     | 1 9 5 |       |     | 6 7 2 | 1 9 5 | 3 4 8 |
|   9 8 |       |   6   |     | 1 9 8 | 3 4 2 | 5 6 7 |
+-------+-------+-------+     +-------+-------+-------+
| 8     |   6   |     3 |     | 8 5 9 | 7 6 1 | 4 2 3 |
| 4     | 8   3 |     1 |     | 4 2 6 | 8 5 3 | 7 9 1 |
| 7     |   2   |     6 |     | 7 1 3 | 9 2 4 | 8 5 6 |
+-------+-------+-------+     +-------+-------+-------+
|   6   |       | 2 8   |     | 9 6 1 | 5 3 7 | 2 8 4 |
|       | 4 1 9 |     5 |     | 2 8 7 | 4 1 9 | 6 3 5 |
|       |   8   |   7 9 |     | 3 4 5 | 2 8 6 | 1 7 9 |
+-------+-------+-------+     +-------+-------+-------+
```

Figure 1: Sudoku problem with solution

Figure 1 gives an example problem, with a solution. This is the format we are going to use for display; see below for details.

Sudoku constraints as injectivity requirements

To express the Sudoku constraints, we have to be able to express the property that a function is injective (or: one-to-one, or: an injection).

A function $f : X \to Y$ is an injection if it preserves distinctions: if $x_1 \neq x_2$ then $f(x_1) \neq f(x_2)$.

Equivalently: a function $f : X \to Y$ is injective if $f(x_1) = f(x_2)$ implies that $x_1 = x_2$.

Thus, we can represent a Sudoku as a matrix $f[i, j]$, satisfying:

- The members of each row should be all different. I.e., for every i, the function $j \mapsto f[i, j]$ should be injective (one to one). I.e., the list of values

  ```
  [f [i,j] | j <- [1..9] ]
  ```

  should not have duplicates.

- The members of every column should be all different. I.e., for every j, the function $i \mapsto f[i,j]$ should be injective (one to one). I.e., the list of values

  ```
  [f [i,j] | i <- [1..9] ]
  ```

  should not have duplicates.

- The members of every subgrid $[i,j]$ with i,j ranging over $1..3$, $4..6$ and $7..9$ should be all different. I.e., the list of values

  ```
  [f [i,j] | i <- [1..3], j <- [1..3] ]
  ```

  should not have duplicates, and similarly for the other subgrids.

Implementing a Sudoku Solver

The specification in the previous section suggests the following declarations:

```
type Row    = Int
type Column = Int
type Value  = Int
type Grid   = [[Value]]

positions, values :: [Int]
positions = [1..9]
values    = [1..9]

blocks :: [[Int]]
blocks = [[1..3],[4..6],[7..9]]
```

Showing Sudoku stuff

Use 0 for a blank slot, so show 0 as a blank. Showing a value:

```
showVal :: Value -> String
showVal 0 = " "
showVal d = show d
```

Showing a row by sending it to the screen; not the type `IO()` for the result:

```
showRow :: [Value] -> IO()
showRow [a1,a2,a3,a4,a5,a6,a7,a8,a9] =
 do  putChar '|'          ; putChar ' '
     putStr (showVal a1) ; putChar ' '
     putStr (showVal a2) ; putChar ' '
     putStr (showVal a3) ; putChar ' '
     putChar '|'          ; putChar ' '
     putStr (showVal a4) ; putChar ' '
     putStr (showVal a5) ; putChar ' '
     putStr (showVal a6) ; putChar ' '
     putChar '|'          ; putChar ' '
     putStr (showVal a7) ; putChar ' '
     putStr (showVal a8) ; putChar ' '
     putStr (showVal a9) ; putChar ' '
     putChar '|'          ; putChar '\n'
```

Showing a grid, i.e., a sequence of rows.

```
showGrid :: Grid -> IO()
showGrid [as,bs,cs,ds,es,fs,gs,hs,is] =
 do putStrLn ("+-------+-------+-------+")
    showRow as; showRow bs; showRow cs
    putStrLn ("+-------+-------+-------+")
    showRow ds; showRow es; showRow fs
    putStrLn ("+-------+-------+-------+")
    showRow gs; showRow hs; showRow is
    putStrLn ("+-------+-------+-------+")
```

Sudoku Type

Define a Sudoku as a function from positions to values

```
type Sudoku = (Row,Column) -> Value
```

Useful conversions:

```
sud2grid :: Sudoku -> Grid
sud2grid s =
  [ [ s (r,c) | c <- [1..9] ] | r <- [1..9] ]

grid2sud :: Grid -> Sudoku
grid2sud gr = \ (r,c) -> pos gr (r,c)
  where
  pos :: [[a]] -> (Row,Column) -> a
  pos gr (r,c) = (gr !! (r-1)) !! (c-1)
```

Showing a Sudoku

Show a Sudoku by displaying its grid:

```
showSudoku :: Sudoku -> IO()
showSudoku = showGrid . sud2grid
```

Picking the block of a position

```
bl :: Int -> [Int]
bl x = concat $ filter (elem x) blocks
```

Picking the subgrid of a position in a Sudoku.

```
subGrid :: Sudoku -> (Row,Column) -> [Value]
subGrid s (r,c) =
  [ s (r',c') | r' <- bl r, c' <- bl c ]
```

Free Values

Free values are available values at open slot positions. Free in a sequence are all values that have not yet been used.

```
freeInSeq :: [Value] -> [Value]
freeInSeq seq = values \\ seq
```

Free in a row are all values not yet used in that row.

```
freeInRow :: Sudoku -> Row -> [Value]
freeInRow s r =
  freeInSeq [ s (r,i) | i <- positions  ]
```

Similarly for free in a column.

```
freeInColumn :: Sudoku -> Column -> [Value]
freeInColumn s c =
  freeInSeq [ s (i,c) | i <- positions ]
```

And for free in a subgrid.

```
freeInSubgrid :: Sudoku -> (Row,Column) -> [Value]
freeInSubgrid s (r,c) = freeInSeq (subGrid s (r,c))
```

The key notion

The available values at a position are the values that are free in the row of that position, free in the column of that position, and free in the subgrid of that position.

```
freeAtPos :: Sudoku -> (Row,Column) -> [Value]
freeAtPos s (r,c) =
  (freeInRow s r)
   `intersect` (freeInColumn s c)
   `intersect` (freeInSubgrid s (r,c))
```

Injectivity

A list of values is injective if each value occurs only once in the list:

```
injective :: Eq a => [a] -> Bool
injective xs = nub xs == xs
```

Injectivity Checks

Check (the non-zero values on) the rows for injectivity.

```
rowInjective :: Sudoku -> Row -> Bool
rowInjective s r = injective vs where
   vs = filter (/= 0) [ s (r,i) | i <- positions ]
```

Check (the non-zero values on) the columns for injectivity.

```
colInjective :: Sudoku -> Column -> Bool
colInjective s c = injective vs where
   vs = filter (/= 0) [ s (i,c) | i <- positions ]
```

Check (the non-zero values on) the subgrids for injectivity.

```
subgridInjective :: Sudoku -> (Row,Column) -> Bool
subgridInjective s (r,c) = injective vs where
   vs = filter (/= 0) (subGrid s (r,c))
```

Consistency Check

Combine the injectivity checks defined above.

```
consistent :: Sudoku -> Bool
consistent s = and $
               [ rowInjective s r |  r <- positions ]
                ++
               [ colInjective s c |  c <- positions ]
                ++
               [ subgridInjective s (r,c) |
                    r <- [1,4,7], c <- [1,4,7]]
```

Sudoku Extension

Extend a Sudoku by filling in a value in a new position.

```
extend :: Sudoku -> ((Row,Column),Value) -> Sudoku
extend = update
```

```
update :: Eq a => (a -> b) -> (a,b) -> a -> b
update f (y,z) x = if x == y then z else f x
```

Search for a Sudoku Solution

A Sudoku constraint is a list of possible values for a particular position.

```
type Constraint = (Row,Column,[Value])
```

Nodes in the search tree are pairs consisting of a Sudoku and the list of all empty positions in it, together with possible values for those positions, according to the constraints imposed by the Sudoku.

```
type Node = (Sudoku,[Constraint])

showNode :: Node -> IO()
showNode = showSudoku . fst
```

Solution

A Sudoku is solved if there are no more empty slots.

```
solved  :: Node -> Bool
solved = null . snd
```

Successors in the Search Tree

The successors of a node are the nodes where the Sudoku gets extended at the next empty slot position on the list, using the values listed in the constraint for that position.

```
extendNode :: Node -> Constraint -> [Node]
extendNode (s,constraints) (r,c,vs) =
   [(extend s ((r,c),v),
     sortBy length3rd $
         prune (r,c,v) constraints) | v <- vs ]
```

`prune` removes the new value v from the relevant constraints, given that v now occupies position (r, c). The definition of `prune` is given below.

Put constraints that are easiest to solve first

```
length3rd :: (a,b,[c]) -> (a,b,[c]) -> Ordering
length3rd (_,_,zs) (_,_,zs') =
  compare (length zs) (length zs')
```

Pruning

Prune values that are no longer possible from constraint list, given a new guess (r, c, v) for the value of (r, c).

```
prune :: (Row,Column,Value)
      -> [Constraint] -> [Constraint]
prune _ [] = []
prune (r,c,v) ((x,y,zs):rest)
  | r == x = (x,y,zs\\[v]) : prune (r,c,v) rest
  | c == y = (x,y,zs\\[v]) : prune (r,c,v) rest
  | sameblock (r,c) (x,y) =
        (x,y,zs\\[v]) : prune (r,c,v) rest
  | otherwise = (x,y,zs) : prune (r,c,v) rest

sameblock :: (Row,Column) -> (Row,Column) -> Bool
sameblock (r,c) (x,y) = bl r == bl x && bl c == bl y
```

Initialisation

Success is indicated by return of a unit node `[n]`.

```
initNode :: Grid -> [Node]
initNode gr = let s = grid2sud gr in
              if (not . consistent) s then []
              else [(s, constraints s)]
```

The open positions of a Sudoku are the positions with value 0.

```
openPositions :: Sudoku -> [(Row,Column)]
openPositions s = [ (r,c) | r <- positions,
                            c <- positions,
                            s (r,c) == 0 ]
```

Sudoku constraints, in a useful order

Put the constraints with the shortest lists of possible values first.

```
constraints :: Sudoku -> [Constraint]
constraints s = sortBy length3rd
    [(r,c, freeAtPos s (r,c)) |
                              (r,c) <- openPositions s ]
```

Depth First Search

The depth first search algorithm is completely standard. The goal property is used to end the search.

Aside: Search Trees

Here is a datatype for trees with lists of branches:

```
data Tree a = T a [Tree a] deriving (Eq,Ord,Show)
```

Here are some example trees:

```
exmple1 = T 1 [T 2 [], T 3 []]
exmple2 = T 0 [exmple1,exmple1,exmple1]
```

This gives:

```
*SS> exmple1
T 1 [T 2 [],T 3 []]
*SS> exmple2
T 0 [T 1 [T 2 [],T 3 []],
     T 1 [T 2 [],T 3 []],T 1 [T 2 [],T 3 []]]
```

Growing Trees

If you have a step function of type `node -> [node]` and a seed, you can grow a tree, as follows.

```
grow :: (node -> [node]) -> node -> Tree node
grow step seed = T seed (map (grow step) (step seed))
```

Growing Trees

If you have a step function of type `node -> [node]` and a seed, you can grow a tree, as follows.

```
grow :: (node -> [node]) -> node -> Tree node
grow step seed = T seed (map (grow step) (step seed))
```

```
SS> grow  (\x -> if x < 2 then [x+1, x+1] else []) 0
T 0 [T 1 [T 2 [],T 2 []],T 1 [T 2 [],T 2 []]]
```

Tree Puzzle

```
count :: Tree a -> Int
count (T _ ts) = 1 + sum (map count ts)
```

Consider the following output:

```
*SS> count (grow  (\x -> if x < 2 then [x+1, x+1] else []) 0)
7
```

Can you predict the value of the following:

```
count (grow  (\x -> if x < 6 then [x+1, x+1] else []) 0)
```

Tree Puzzle

```
count :: Tree a -> Int
count (T _ ts) = 1 + sum (map count ts)
```

Consider the following output:

```
*SS> count (grow  (\x -> if x < 2 then [x+1, x+1] else []) 0)
7
```

Can you predict the value of the following:

```
count (grow  (\x -> if x < 6 then [x+1, x+1] else []) 0)
```

Explanation

The tree grown by `(\x -> if x < 6 then [x+1, x+1] else []) 0` is a balanced binary branching tree of depth 6.

A balanced binary tree of depth n has $2^n + 1$ internal nodes and 2^n leaf nodes, which makes $2^{n+1} - 1$ nodes in total. For a balanced binary tree of depth 6 this gives $2^7 - 1 = 127$ nodes.

Another Tree

```
*SS> :set +m
*SS> grow (\ (x,y) -> if x+y < 10 then [(x,x+y),(x+y,y)]
*SS|      else []) (1,1)

T (1,1) [T (1,2) [T (1,3) [T (1,4) [T (1,5) [T (1,6)
 [T (1,7) [T (1,8) [T (1,9) [],T (9,8) []],T (8,7) []],
T (7,6) []],T (6,5) []],T (5,4) [T (5,9) [],T (9,4) []]],
T (4,3) [T (4,7) [],T (7,3) []]],T (3,2) [T (3,5)
 [T (3,8) [],T (8,5) []],T (5,2) [T (5,7) [],T (7,2)
 [T (7,9) [],T (9,2) []]]]],T (2,1) [T (2,3) [T (2,5)
 [T (2,7) [T (2,9) [],T (9,7) []],T (7,5) []],T (5,3)
 [T (5,8) [],T (8,3) []]],T (3,1) [T (3,4) [T (3,7) [],
T (7,4) []],T (4,1) [T (4,5) [T (4,9) [],T (9,5) []],
T (5,1) [T (5,6) [],T (6,1) [T (6,7) [],T (7,1)
 [T (7,8) [],T (8,1) [T (8,9) [],T (9,1) []]]]]]]]]
```

Depth First Search

```
search :: (node -> [node])
       -> (node -> Bool) -> [node] -> [node]
search children goal [] = []
search children goal (x:xs)
  | goal x     = x : search children goal xs
  | otherwise = search children goal
                                ((children x) ++ xs)
```

This is a generic algorithm for depth-first search.

The third argument, of type `[node]`, gives the list of nodes that still have to be searched.

Understanding this . . .

Explain as clearly as you can:

- What does the first argument `children :: node -> [node]` represent?
- What does the second argument `goal :: node -> Bool` represent?
- Can you explain the `types` of these arguments?

Understanding this . . .

Explain as clearly as you can:

- What does the first argument `children :: node -> [node]` represent?
- What does the second argument `goal :: node -> Bool` represent?
- Can you explain the `types` of these arguments?

Answers:

- The first argument is the step function that can be used to generate the search tree.
- The second argument represents the nodes that are success nodes. These are the nodes we are looking for while searching through the tree.

```
*SS>  search (\x -> if x < 6 then [x+1, x+2] else []) odd [0]
[1,3,5]
```

Depth First Versus Breadth First

Q: What makes this a definition of depth-first search?

Depth First Versus Breadth First

Q: What makes this a definition of depth-first search?

A: The successors of a node are visited before the siblings of a node.

Depth First Versus Breadth First

Q: What makes this a definition of depth-first search?

A: The successors of a node are visited before the siblings of a node.

Q: How can you change this into a definition of breadth-first search (where the tree is searched layer-by-layer)?

Depth First Versus Breadth First

Q: What makes this a definition of depth-first search?

A: The successors of a node are visited before the siblings of a node.

Q: How can you change this into a definition of breadth-first search (where the tree is searched layer-by-layer)?

A: Just make sure that the siblings are visited before the daughters, by changing

```
search successors goal ((successors x) ++ xs)
```

into

```
search successors goal (xs ++ (successors x))
```

Pursuing the Sudoku Search

```
solveNs :: [Node] -> [Node]
solveNs = search succNode solved

succNode :: Node -> [Node]
succNode (s,[]) = []
succNode (s,p:ps) = extendNode (s,ps) p
```

Solving and showing the results

This uses some monad operators: `fmap` and `sequence`.

```
solveAndShow :: Grid -> IO[()]
solveAndShow gr = solveShowNs (initNode gr)

solveShowNs :: [Node] -> IO[()]
solveShowNs = sequence . fmap showNode . solveNs
```

Examples

```
example1 :: Grid
example1 = [[5,3,0,0,7,0,0,0,0],
            [6,0,0,1,9,5,0,0,0],
            [0,9,8,0,0,0,0,6,0],
            [8,0,0,0,6,0,0,0,3],
            [4,0,0,8,0,3,0,0,1],
            [7,0,0,0,2,0,0,0,6],
            [0,6,0,0,0,0,2,8,0],
            [0,0,0,4,1,9,0,0,5],
            [0,0,0,0,8,0,0,7,9]]
```

```
example2 :: Grid
example2 = [[0,3,0,0,7,0,0,0,0],
            [6,0,0,1,9,5,0,0,0],
            [0,9,8,0,0,0,0,6,0],
            [8,0,0,0,6,0,0,0,3],
            [4,0,0,8,0,3,0,0,1],
            [7,0,0,0,2,0,0,0,6],
            [0,6,0,0,0,0,2,8,0],
            [0,0,0,4,1,9,0,0,5],
            [0,0,0,0,8,0,0,7,9]]
```

```
example3 :: Grid
example3 = [[1,0,0,0,3,0,5,0,4],
            [0,0,0,0,0,0,0,0,3],
            [0,0,2,0,0,5,0,9,8],
            [0,0,9,0,0,0,0,3,0],
            [2,0,0,0,0,0,0,0,7],
            [8,0,3,0,9,1,0,6,0],
            [0,5,1,4,7,0,0,0,0],
            [0,0,0,3,0,0,0,0,0],
            [0,4,0,0,0,9,7,0,0]]
```

```
example4 :: Grid
example4 = [[1,2,3,4,5,6,7,8,9],
            [2,0,0,0,0,0,0,0,0],
            [3,0,0,0,0,0,0,0,0],
            [4,0,0,0,0,0,0,0,0],
            [5,0,0,0,0,0,0,0,0],
            [6,0,0,0,0,0,0,0,0],
            [7,0,0,0,0,0,0,0,0],
            [8,0,0,0,0,0,0,0,0],
            [9,0,0,0,0,0,0,0,0]]
```

```
example5 :: Grid
example5 = [[1,0,0,0,0,0,0,0,0],
            [0,2,0,0,0,0,0,0,0],
            [0,0,3,0,0,0,0,0,0],
            [0,0,0,4,0,0,0,0,0],
            [0,0,0,0,5,0,0,0,0],
            [0,0,0,0,0,6,0,0,0],
            [0,0,0,0,0,0,7,0,0],
            [0,0,0,0,0,0,0,8,0],
            [0,0,0,0,0,0,0,0,9]]
```

```
+---------+---------+---------+
|         | 3       |         |
|   +-----|--+   +--|-----+   |
|   |     | 7|   |  | 3   |   |
| 2 |     |  |   |  |     | 8 |
+---------+---------+---------+
|   |   6 |  |   |5 |     |   |
|   +-----|--+   +--|-----+   |
|    9  1 | 6       |         |
|   +-----|--+   +--|-----+   |
| 3 |     |  | 7 |1 | 2   |   |
+---------+---------+---------+
|   |     |  |   |  |    3| 1 |
|   |8    |  | 4 |  |     |   |
|   +-----|--+   +--|-----+   |
|       2 |         |         |
+---------+---------+---------+
```

Figure 2: Sudoku exercise, NRC, Saturday Nov 26, 2005.

Sudoku Generation

An empty node is a Sudoku function that assigns 0 everywhere, together with the trivial constraints that forbid nothing.

```
emptyN :: Node
emptyN = (\ _ -> 0,constraints (\ _ -> 0))
```

Get a random integer from the random generator:

```
getRandomInt :: Int -> IO Int
getRandomInt n = getStdRandom (randomR (0,n))
```

Pick a random member from a list; the empty list indicates failure.

```
getRandomItem :: [a] -> IO [a]
getRandomItem [] = return []
getRandomItem xs =
  do n <- getRandomInt maxi
     return [xs !! n]
     where maxi = length xs - 1
```

Randomize a list.

```
randomize :: Eq a => [a] -> IO [a]
randomize xs = do y <- getRandomItem xs
                  if null y
                    then return []
                    else do ys <- randomize (xs\\y)
                            return (head y:ys)
```

```
sameLen :: Constraint -> Constraint -> Bool
sameLen (_,_,xs) (_,_,ys) = length xs == length ys
```

```
getRandomCnstr :: [Constraint] -> IO [Constraint]
getRandomCnstr cs = getRandomItem (f cs)
  where f [] = []
        f (x:xs) = takeWhile (sameLen x) (x:xs)
```

```
rsuccNode :: Node -> IO [Node]
rsuccNode (s,cs) =
  do xs <- getRandomCnstr cs
     if null xs
        then return []
        else return (extendNode (s,cs\\xs) (head xs))
```

Find a random solution

```
rsolveNs :: [Node] -> IO [Node]
rsolveNs ns = rsearch rsuccNode solved (return ns)
```

This uses random search . . .

```
rsearch :: (node -> IO [node])
        -> (node -> Bool) -> IO [node] -> IO [node]
rsearch succ goal ionodes =
  do xs <- ionodes
     if null xs then return []
     else
       if goal (head xs) then return [head xs]
       else
        do ys <- rsearch succ goal (succ (head xs))
           if (not . null) ys then return [head ys]
           else
             if null (tail xs) then return []
             else rsearch succ goal (return $ tail xs)
```

```
genRandomSudoku :: IO Node
genRandomSudoku = do [r] <- rsolveNs [emptyN]
                     return r
```

```
randomS = genRandomSudoku >>= showNode
```

```
uniqueSol :: Node -> Bool
uniqueSol node = singleton (solveNs [node]) where
  singleton [] = False
  singleton [x] = True
  singleton (x:y:zs) = False
```

Erase a position from a Sudoku.

```
eraseS :: Sudoku -> (Row,Column) -> Sudoku
eraseS s (r,c) = update s ((r,c),0)
```

Erase a position from a Node.

```
eraseN :: Node -> (Row,Column) -> Node
eraseN n (r,c) = (s, constraints s)
  where s = eraseS (fst n) (r,c)
```

Return a ‘minimal’ node with a unique solution by erasing positions until the result becomes ambiguous.

```
minimalize :: Node -> [(Row,Column)] -> Node
minimalize n [] = n
minimalize n ((r,c):rcs)
   | uniqueSol n' = minimalize n' rcs
   | otherwise    = minimalize n  rcs
  where n' = eraseN n (r,c)
```

```
filledPositions :: Sudoku -> [(Row,Column)]
filledPositions s =
  [ (r,c) | r <- positions,
            c <- positions, s (r,c) /= 0 ]
```

```
genProblem :: Node -> IO Node
genProblem n = do ys <- randomize xs
                  return (minimalize n ys)
   where xs = filledPositions (fst n)
```

```
main :: IO ()
main = do [r] <- rsolveNs [emptyN]
          showNode r
          s  <- genProblem r
          showNode s
```

Example output from this:

```
*SS> main
+-------+-------+-------+
| 8 9 5 | 6 7 1 | 4 2 3 |
| 7 4 2 | 9 5 3 | 8 6 1 |
| 6 1 3 | 2 4 8 | 9 5 7 |
+-------+-------+-------+
| 9 2 4 | 1 3 7 | 5 8 6 |
| 5 8 7 | 4 9 6 | 1 3 2 |
| 1 3 6 | 5 8 2 | 7 9 4 |
+-------+-------+-------+
| 2 7 1 | 8 6 9 | 3 4 5 |
| 4 6 9 | 3 1 5 | 2 7 8 |
| 3 5 8 | 7 2 4 | 6 1 9 |
+-------+-------+-------+
```

```
+-------+-------+-------+
|     5 |       |       |
|   4   | 9     |     1 |
|       | 2   8 |   5 7 |
+-------+-------+-------+
|       |   3   | 5 8 6 |
|     7 |   9   |   3   |
|   3   |       |   9   |
+-------+-------+-------+
|       | 8     | 3     |
| 4 6   |       |       |
|   5 8 | 7     |       |
+-------+-------+-------+
```

Exercises

See Lab work for this week.

Some further exercises:

Exercise 2 Let's say that a crossed Sudoku is a Sudoku satisfying the additional constraints that the values on the two diagonals are all different. Write a program that generates minimal problems for crossed Sudokus. How many hints do these problems contain, on average?

Exercise 3 How about crossed NRC Sudokus or NRCX Sudokus: Sudokus satisfying both the NRC constraints and the diagonal constraints? Write a program that generates minimal problems for crossed NRC Sudokus. How many hints do these problems contain, on average? Can you generate examples with 9 hints? With less than 9 hints?

Exercise 4 Andries Brouwer mentions on his NRC Sudoku webpage that there are

$$6337174388428800$$

different NRC Sudokus. Confirm this number with a program. See http://homepages.cwi.nl/~aeb/games/sudoku/nrc.html.

Exercise 5 How many NRCX Sudokus are there?

Further Reading

For further background, you might wish to read Rosenhouse and Taalman [2011]. Information about counting methods for Sudokus can be found in Felgenhauer and Jarvis [2006] and Russell and Jarvis [2006].

References

B. Felgenhauer and F. Jarvis. Enumerating possible sudoku grids, 2005.

B. Felgenhauer and F. Jarvis. Mathematics of Sudoku I, 2006.

R. Pelánek. Difficulty rating of sudoku puzzles: An overview and evaluation, 2014.

J. Rosenhouse and L. Taalman. Taking Sudoku Seriously: The Math behind the World's Most Popular Pencil Puzzle. Oxford University Press, 2011.

E. Russell and F. Jarvis. Mathematics of Sudoku II, 2006.