Sudoku

Ari An, Xinyao Peng

2022 Fall

Introduction

Sudoku is a game to fill a 9 × 9 grid with digits so that each column, each row, and each of the nine 3 × 3 subgrids(cells) that compose the grid contain all of the digits from 1 to 9. Our project will parallel the backtracking algorithm and use it to solve the sudoku.

Goal

We will implement the backtracking algorithm using the minimum remaining value (MRV) heuristic to solve the sudoku problem. The algorithm will pick one of the possible values for an unfilled value in sudoku and do forward checking when a value is chosen in order to further reduce possible value domains.

Set-up Functions

In order to solve the sudoku, we firstly need to define several set up functions.
To illustrate the set-up functions, let's raise a sudoku example.

- The input is a string with 81 characters, where each character represents a square in the 9x9 grid. Note that the string is defined as a character list.

```
example :: String
example = "000000021430000000600000000201500000000006370000000000068000400000230000000070000"
```

- lineToList function converts the example into a list of integers. Denote the result as "list".

```
ghci> list = lineToList example
ghci> list
[0,0,0,0,0,0,0,2,1,4,3,0,0,0,0,0,0,0,6,0,0,0,0,0,0,0,0,2,0,1,5,0,0,0,0,0,0,0,0,0,0,6,3,7,0,0
,0,0,0,0,0,0,0,0,0,6,8,0,0,0,4,0,0,0,0,0,2,3,0,0,0,0,0,0,0,0,7,0,0,0,0]
```

- showGrid function displays the list in the form of 9x9 grid and is used for testing purpose.

```
ghci> showGrid list
[0,0,0,0,0,0,0,2,1]
[4,3,0,0,0,0,0,0,0]
[6,0,0,0,0,0,0,0,0]
[2,0,1,5,0,0,0,0,0]
[0,0,0,0,0,6,3,7,0]
[0,0,0,0,0,0,0,0,0]
[0,6,8,0,0,0,4,0,0]
[0,0,0,2,3,0,0,0,0]
[0,0,0,0,7,0,0,0,0]
```

- getRowGrid, getColGrid, and getCellGrid convert the original list into a new list of 9 inner lists where each of them represents a row, a column, or a cell.

```
ghci> getRowGrid list
[[0,0,0,0,0,0,0,2,1],[4,3,0,0,0,0,0,0,0],[6,0,0,0,0,0,0,0,0],[2,0,1,5,0,0,0,0,0],[0,0,0,0,0,
6,3,7,0],[0,0,0,0,0,0,0,0,0],[0,6,8,0,0,0,4,0,0],[0,0,0,2,3,0,0,0,0],[0,0,0,0,7,0,0,0,0]]
ghci> getColGrid list
[[0,4,6,2,0,0,0,0,0],[0,3,0,0,0,0,6,0,0],[0,0,0,1,0,0,8,0,0],[0,0,0,5,0,0,0,2,0],[0,0,0,0,0,
0,0,3,7],[0,0,0,0,6,0,0,0,0],[0,0,0,0,3,0,4,0,0],[2,0,0,0,7,0,0,0,0],[1,0,0,0,0,0,0,0,0]]
ghci> getCellGrid list
[[0,0,0,4,3,0,6,0,0],[0,0,0,0,0,0,0,0,0],[0,2,1,0,0,0,0,0,0],[2,0,1,0,0,0,0,0,0],[5,0,0,0,0,
6,0,0,0],[0,0,0,3,7,0,0,0,0],[0,6,8,0,0,0,0,0,0],[0,0,0,2,3,0,0,7,0],[4,0,0,0,0,0,0,0,0]]
```

Algorithm

1. **Backtracking**

   We search every possible combination in an attempt to solve the sudoku. Also, we utilise a "possibility grid" to store the potentially legal values for each square tile. The "possibility grid" is generated by the possibleGrid function. A possibility grid is a list of 81 sets where each set indicates all possible values for each square. We denote the result as possGrid.

```
ghci> possGrid = possibleGrid list
ghci> possGrid
[fromList [1,2,3,4,5,6,7,8,9],fromList [1,2,3,4,5,6,7,8,9],fromList [1,2,3,4,5,6,7,8,9],fromList [1,2,3,4,5,6,7,8,9]
,fromList [1,2,3,4,5,6,7,8,9],fromList [1,2,3,4,5,6,7,8,9],fromList [1,2,3,4,5,6,7,8,9],fromList [2],fromList [1],fr
omList [4],fromList [3],fromList [1,2,3,4,5,6,7,8,9],fromList [1,2,3,4,5,6,7,8,9],fromList [1,2,3,4,5,6,7,8,9],fromL
ist [1,2,3,4,5,6,7,8,9],fromList [1,2,3,4,5,6,7,8,9],fromList [1,2,3,4,5,6,7,8,9],fromList [1,2,3,4,5,6,7,8,9],fromL
ist [6],fromList [1,2,3,4,5,6,7,8,9],fromList [1,2,3,4,5,6,7,8,9],fromList [1,2,3,4,5,6,7,8,9],fromList [1,2,3,4,5,6
,7,8,9],fromList [1,2,3,4,5,6,7,8,9],fromList [1,2,3,4,5,6,7,8,9],fromList [1,2,3,4,5,6,7,8,9],fromList [1,2,3,4,5,6
,7,8,9],fromList [2],fromList [1,2,3,4,5,6,7,8,9],fromList [1],fromList [5],fromList [1,2,3,4,5,6,7,8,9],fromList [1
,2,3,4,5,6,7,8,9],fromList [1,2,3,4,5,6,7,8,9],fromList [1,2,3,4,5,6,7,8,9],fromList [1,2,3,4,5,6,7,8,9],fromList [1
,2,3,4,5,6,7,8,9],fromList [1,2,3,4,5,6,7,8,9],fromList [1,2,3,4,5,6,7,8,9],fromList [1,2,3,4,5,6,7,8,9],fromList [1
,2,3,4,5,6,7,8,9],fromList [6],fromList [3],fromList [7],fromList [1,2,3,4,5,6,7,8,9],fromList [1,2,3,4,5,6,7,8,9],f
romList [1,2,3,4,5,6,7,8,9],fromList [1,2,3,4,5,6,7,8,9],fromList [1,2,3,4,5,6,7,8,9],fromList [1,2,3,4,5,6,7,8,9],f
romList [1,2,3,4,5,6,7,8,9],fromList [1,2,3,4,5,6,7,8,9],fromList [1,2,3,4,5,6,7,8,9],fromList [1,2,3,4,5,6,7,8,9],f
romList [1,2,3,4,5,6,7,8,9],fromList [6],fromList [8],fromList [1,2,3,4,5,6,7,8,9],fromList [1,2,3,4,5,6,7,8,9],from
List [1,2,3,4,5,6,7,8,9],fromList [4],fromList [1,2,3,4,5,6,7,8,9],fromList [1,2,3,4,5,6,7,8,9],fromList [1,2,3,4,5,
6,7,8,9],fromList [1,2,3,4,5,6,7,8,9],fromList [1,2,3,4,5,6,7,8,9],fromList [2],fromList [3],fromList [1,2,3,4,5,6,7
,8,9],fromList [1,2,3,4,5,6,7,8,9],fromList [1,2,3,4,5,6,7,8,9],fromList [1,2,3,4,5,6,7,8,9],fromList [1,2,3,4,5,6,7
,8,9],fromList [1,2,3,4,5,6,7,8,9],fromList [1,2,3,4,5,6,7,8,9],fromList [1,2,3,4,5,6,7,8,9],fromList [7],fromList [
1,2,3,4,5,6,7,8,9],fromList [1,2,3,4,5,6,7,8,9],fromList [1,2,3,4,5,6,7,8,9],fromList [1,2,3,4,5,6,7,8,9]]
```

   Note that we are not supposed to traverse all possibility tiles, since it is time consuming. Pruning methods will be applied later to eliminate possibilities.

2. **Forward Checking**

   For each variable in the possibility grid, we apply forward checking to reduce variable domains. To be more specific, we find the most constrained square and return a list of the remaining potential values for each square. This procedure is implemented in hardPrune functions.

```
ghci> hardPrune possGrid
[fromList [5,7,8,9],fromList [5,7,8,9],fromList [5,7,9],fromList [3,4,6,7,8,9],fromList [4,5,6,8,9],fromList [3,4,5,
7,8,9],fromList [5,6,7,8,9],fromList [2],fromList [1],fromList [4],fromList [3],fromList [2,5,7,9],fromList [1,6,7,8
,9],fromList [1,2,5,6,8,9],fromList [1,2,5,7,8,9],fromList [5,6,7,8,9],fromList [5,6,8,9],fromList [5,6,7,8,9],fromL
ist [6],fromList [1,2,5,7,8,9],fromList [2,5,7,9],fromList [1,3,4,7,8,9],fromList [1,2,4,5,8,9],fromList [1,2,3,4,5,
7,8,9],fromList [5,7,8,9],fromList [3,4,5,8,9],fromList [3,4,5,7,8,9],fromList [2],fromList [4,7,8,9],fromList [1],f
romList [5],fromList [4,8,9],fromList [3,4,7,8,9],fromList [6,8,9],fromList [4,6,8,9],fromList [4,6,8,9],fromList [5
,8,9],fromList [4,5,8,9],fromList [4,5,9],fromList [1,4,8,9],fromList [1,2,4,8,9],fromList [6],fromList [3],fromList
 [7],fromList [2,4,5,8,9],fromList [3,5,7,8,9],fromList [4,5,7,8,9],fromList [3,4,5,6,7,9],fromList [1,3,4,7,8,9],fr
omList [1,2,4,8,9],fromList [1,2,3,4,7,8,9],fromList [1,2,5,6,8,9],fromList [1,4,5,6,8,9],fromList [2,4,5,6,8,9],fro
mList [1,3,5,7,9],fromList [6],fromList [8],fromList [1,9],fromList [1,5,9],fromList [1,5,9],fromList [4],fromList [
1,3,5,9],fromList [2,3,5,7,9],fromList [1,5,7,9],fromList [1,4,5,7,9],fromList [4,5,7,9],fromList [2],fromList [3],f
romList [1,4,5,8,9],fromList [1,5,6,7,8,9],fromList [1,5,6,8,9],fromList [5,6,7,8,9],fromList [1,3,5,9],fromList [1,
2,4,5,9],fromList [2,3,4,5,9],fromList [1,4,6,8,9],fromList [7],fromList [1,4,5,8,9],fromList [1,2,5,6,8,9],fromList
 [1,3,5,6,8,9],fromList [2,3,5,6,8,9]]
```

   Given a square with fixed value, the hardPrune function eliminates this value from all squares that are located in the same row, column, and cell of the selected square tile. It repeats the process until there is no way to further eliminate the possibilities in adjacent tiles.

3. **Minimum remaining value heuristic**

We apply this heuristic to choose the variable with the fewest legal remaining values in its domain. Given a possibility grid, we use the softPrune method to find the square with least number of possibilities. Then, we choose a possible value from the set and return a tuple of chosen grid and unchosen grid. Chosen grid is the grid constructed by the selected possible values, and unchosen grid eliminates the selected value from the current set.

Method

To begin with, we set up two condition checkers: ifSolved and ifValid.

- **ifSolved** function checks whether or not a sudoku is solved. That is, it returns true if all rows, columns, and cells contain exactly nine increasing numbers (1,2,3,4,5,6,7,8,9); returns false if any of the conditions does not meet.
- In contrast, the **ifValid** function checks whether or not values in a newly-generated grid are consistent. That is, after the softPrune function is generated to produce a chosen grid, we apply ifValid to check whether the chosen grid contains repeated values that are out of bound.

Finally, we combine all these functions to create the solveSudoku function. If a solution is found, return the list of values in such a grid; if the value is not found, report the error. In our case, the result is shown below.

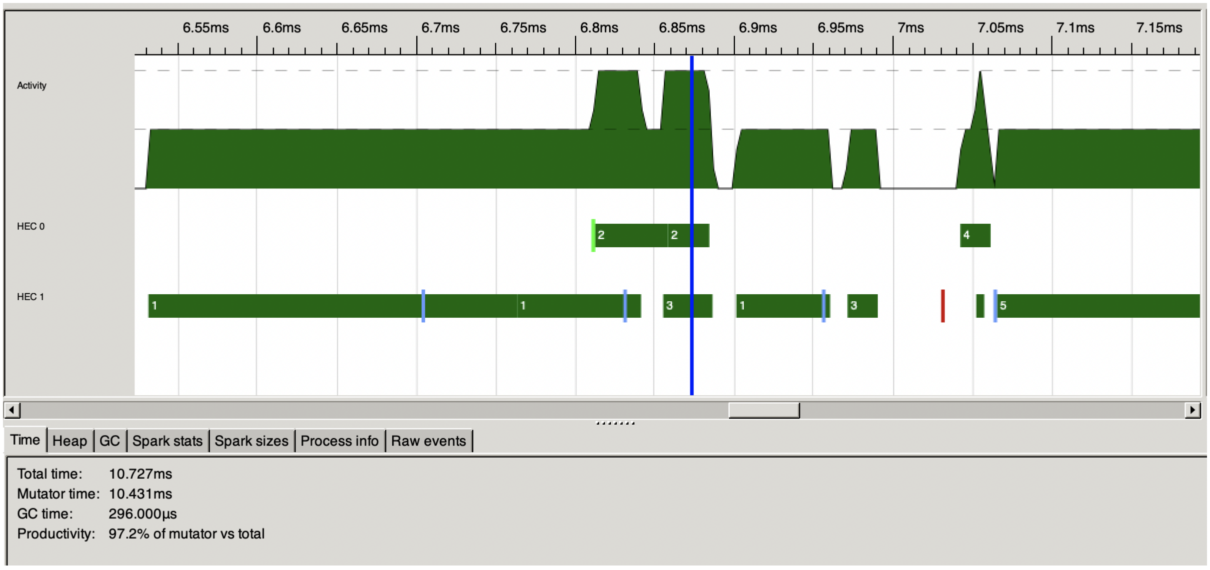
```
ghci> solveSudoku possGrid
Just [8,5,7,3,4,9,6,2,1,4,3,2,8,6,1,5,9,7,6,1,9,7,5,2,8,4,3,2,7,1,5,8,3,9,6,4,9,4,5,1,2,6,3,7,8,3,8,6,4
,9,7,2,1,5,7,6,8,9,1,5,4,3,2,1,9,4,2,3,8,7,5,6,5,2,3,6,7,4,1,8,9]
ghci> showGrid $ fromJust result
[8,5,7,3,4,9,6,2,1]
[4,3,2,8,6,1,5,9,7]
[6,1,9,7,5,2,8,4,3]
[2,7,1,5,8,3,9,6,4]
[9,4,5,1,2,6,3,7,8]
[3,8,6,4,9,7,2,1,5]
[7,6,8,9,1,5,4,3,2]
[1,9,4,2,3,8,7,5,6]
[5,2,3,6,7,4,1,8,9]
```

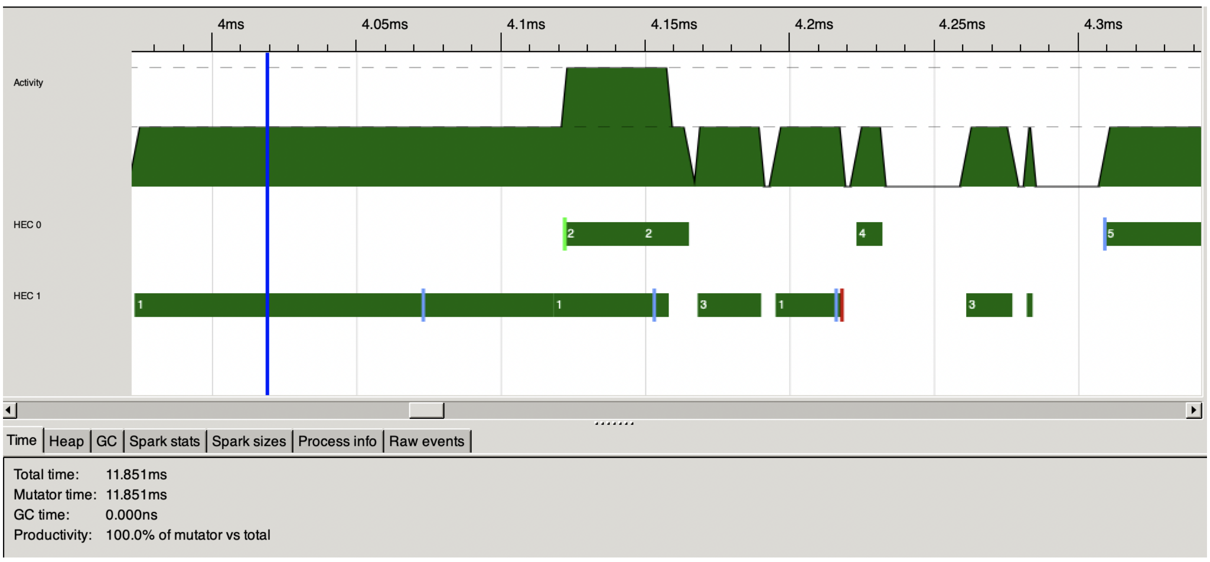
The next step is to parallel the sudoku algorithms.

Parallel

By using the Static Partitioning, we speed up our model a lot. Before it took about 2s for each sudokus in the test.txt, and now it only takes 11.8ms for all 1000 sudoku problem.

```
        335,976 bytes allocated in the heap
         26,712 bytes copied during GC
        115,936 bytes maximum residency (1 sample(s))
         39,712 bytes maximum slop
              3 MiB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0         0 colls,     0 par    0.000s   0.000s     0.0000s    0.0000s
  Gen  1         1 colls,     0 par    0.000s   0.000s     0.0003s    0.0003s

  TASKS: 6 (1 bound, 5 peak workers (5 total), using -N2)

  SPARKS: 3 (2 converted, 0 overflowed, 0 dud, 0 GC'd, 1 fizzled)

  INIT    time    0.001s  (  0.011s elapsed)
  MUT     time    0.000s  (  0.001s elapsed)
  GC      time    0.000s  (  0.000s elapsed)
  EXIT    time    0.000s  (  0.002s elapsed)
  Total   time    0.002s  (  0.015s elapsed)

  Alloc rate    735,177,242 bytes per MUT second

  Productivity  21.7% of total user, 8.2% of total elapsed
```





Comparison

After parallelling, we compare our final version of the algorithm with the sudoku1.hs shown in class, which is taken from https://github.com/simonmar/parconc-examples/archive/master.tar.gz. The performance of our algorithm took an advantage over the sample solution. Below are the running time statistics for the sample solution with about 6 sudoku puzzles..

```
 123,503,549,360 bytes allocated in the heap
   1,901,670,360 bytes copied during GC
         192,472 bytes maximum residency (247 sample(s))
          45,824 bytes maximum slop
               4 MiB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0     118792 colls, 118792 par    8.063s   4.391s     0.0000s    0.0107s
  Gen  1       247 colls,   246 par    0.068s   0.036s     0.0001s    0.0006s

  Parallel GC work balance: 1.05% (serial 0%, perfect 100%)

  TASKS: 6 (1 bound, 5 peak workers (5 total), using -N2)

  SPARKS: 0 (0 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)

  INIT    time    0.001s  (  0.008s elapsed)
  MUT     time   27.584s  ( 27.719s elapsed)
  GC      time    8.130s  (  4.426s elapsed)
  EXIT    time    0.000s  (  0.005s elapsed)
  Total   time   35.716s  ( 32.158s elapsed)

  Alloc rate    4,477,306,503 bytes per MUT second

  Productivity  77.2% of total user, 86.2% of total elapsed
```

Coding

sudoku.hs

```
{-# OPTIONS_GHC -Wno-unrecognised-pragmas #-}
{-# HLINT ignore "Avoid lambda" #-}
{-# HLINT ignore "Eta reduce" #-}
module Sudoku where
import Data.Char (digitToInt)
import Data.List (transpose, elemIndex)
import Data.Set (Set, fromList, toList, member, size, difference, unions, lookupGT,
deleteAt)
import Data.Maybe (fromJust,isJust)
import Control.Applicative ((<|>))


splitList :: Int -> [a] -> [[a]]
splitList _ [] = []
```

```
splitList n oriList = prev : splitList n next
 where
   (prev, next) = splitAt n oriList

example :: String
example =
"000000021430000000600000000201500000000006370000000000006800040000023000000000070000"

lineToList :: [Char] -> [Int]
lineToList oriLine = map digitToInt oriLine

getCell :: Int -> [a] -> [a]
getCell n oriList = newList !! cellIndex ++ newList !! (cellIndex + 3) ++ newList !!
(cellIndex + 6)
 where
   cellIndex = div n 3 * 9 + mod n 3
   newList = splitList 3 oriList

getRow :: Int -> [a] -> [a]
getRow n cellGrid = newList !! rowIndex ++ newList !! (rowIndex + 3) ++ newList !!
(rowIndex + 6)
 where
   rowIndex = mod n 3 + (div n 3) * 9
   newList = splitList 3 cellGrid

getRowGrid :: [a] -> [[a]]
getRowGrid oriList = splitList 9 oriList

getColGrid :: [a] -> [[a]]
getColGrid oriList = transpose $ getRowGrid oriList

getCellGrid :: [a] -> [[a]]
getCellGrid oriList = [ getCell i oriList | i <- [0..8] ]

showGrid :: [Int] -> IO ()
showGrid oriList = mapM_ print (getRowGrid oriList)

possibleGrid :: (Ord a, Num a, Enum a) => [a] -> [Set a]
possibleGrid oriList = [ if member val def then fromList [val] else def | val <- oriList]
 where
   def = fromList [1..9]

getFixedByRow :: Ord a => [Set a] -> [Set a]
getFixedByRow possGrid = [ unions $ filter (\x -> size x == 1) row | row <- getRowGrid
possGrid ]

getFixedByCell :: Ord a => [Set a] -> [Set a]
getFixedByCell possGrid = [ unions $ filter (\x -> size x == 1) row | row <- getCellGrid
possGrid ]

getFixedByCol :: Ord a => [Set a] -> [Set a]
```

```haskell
getFixedByCol possGrid = [ unions $ filter (\x -> size x == 1) row | row <- getColGrid
possGrid ]

hardPruneHelper :: Ord a => [[Set a]] -> [Set a] -> [[Set a]]
hardPruneHelper allSet fixedRowSet = [ map (\x -> if size x/=1 then x `difference` f else
x) r | (r,f) <- match ]
 where
   match = zip allSet fixedRowSet

hardPruneEach :: Ord a => [Set a] -> [Set a]
hardPruneEach possGrid = concat [ getRow i (concat thiPrune) | i <- [0..8] ]
 where
   fstPrune = hardPruneHelper (getRowGrid possGrid) (getFixedByRow possGrid)
   sndPrune = hardPruneHelper (getColGrid (concat fstPrune)) (getFixedByCol possGrid)
   thiPrune = hardPruneHelper (getCellGrid (concat $ transpose sndPrune)) (getFixedByCell
possGrid)

hardPrune :: Ord a => [Set a] -> [Set a]
hardPrune possGrid | possGrid ==  hardPruneEach possGrid = possGrid
                   | otherwise = hardPruneEach possGrid

softPrune :: Ord a => [Set a] -> ([Set a], [Set a])
softPrune poss | minSize == Nothing = (poss, poss)
               | otherwise = (chosenGrid, unchosenGrid)
 where
   (prev, mid : next) = splitAt index poss
   sizeGrid = map size poss
   minSize = lookupGT 1 (fromList sizeGrid)
   index = fromJust $ elemIndex (fromJust minSize) sizeGrid
   chosenGrid = prev ++ [fromList [head $ toList mid]] ++ next
   unchosenGrid = prev ++ [deleteAt 0 mid] ++ next

ifSolved :: (Ord a, Num a, Enum a) => [Set a] -> Bool
ifSolved poss = and [unions row == fromList [1..9]| row <- getRowGrid poss]
     && and [unions col == fromList [1..9]| col <- getColGrid poss]
     && and [unions cell == fromList [1..9]| cell <- getCellGrid poss]
     && map (\x -> size x) poss == take 81 [1,1..]

ifValid :: (Ord a, Num a) => [Set a] -> Bool
ifValid poss = and [s /= 0 | s <- map (\x -> size x) poss] && and boolList
 where possList = map (\x -> if size x > 1 then -1 else head $ toList x) poss
       allList = getColGrid possList ++ getRowGrid possList ++ getCellGrid possList
       boolList = [ length l == size (fromList l) | list <- allList, let l = filter (/=
(-1)) list]

possToGrid :: [Set b] -> [b]
possToGrid poss = map (\x -> head $ toList x) poss

solveSudoku :: (Num a, Enum a, Ord a) => [Set a] -> Maybe [a]
solveSudoku poss | ifSolved poss = Just (possToGrid poss)
                 | not $ ifValid poss = Nothing
```

```
                 | otherwise = solveSudoku (hardPrune chosen) <|> solveSudoku (hardPrune
unchosen)
 where
   (chosen, unchosen) = softPrune poss
solve :: [Char] -> Maybe [Int]
solve text = solveSudoku grid
 where
   replace = map (\c -> if c=='.' then '0'; else c)
   editedText = replace text
   grid = possibleGrid $ lineToList editedText
```

Main.hs

```
import Sudoku
import Control.Parallel.Strategies ( rpar, rseq, runEval )
import Control.DeepSeq
import Data.Maybe(isJust)

main :: IO ()
main =do sudos <- lines <$> readFile "test.txt"
        let [as,bs,cs] = splitList 3 sudos
            solutions = runEval $ do
               as' <- rpar (force (map solve as))
               bs' <- rpar (force (map solve bs))
               cs' <- rpar (force (map solve cs))
               _ <- rseq as'
               _ <- rseq bs'
               _ <- rseq cs'
               return (as' ++ bs' ++ cs')
        print (length (filter isJust solutions))
```

References

1. https://hackage.haskell.org/package/containers-0.6.6/docs/Data-Set.html
2. https://hackage.haskell.org/package/base-4.17.0.0/docs/Data-List.html
3. https://haskell-containers.readthedocs.io/en/latest/set.html
4. https://www.simplilearn.com/tutorials/data-structure-tutorial/backtracking-algorithm
5. https://ktiml.mff.cuni.cz/~bartak/constraints/propagation.html
6. https://www.7sudoku.com/very-difficult
7. https://github.com/simonmar/parconc-examples