Sudoku Solver

Full name: Shiqin Huo

University ID: u5949730

Date: 22 May

Introduction

Enjoying a worldwide popularity, Sudoku never fails to appeal to programmers. This report mainly introduces the implementation of some human-like algorithms to solve Sudoku in Haskell.

Algorithm

Solving Algorithm 1: Depth-first Search (Brute Force/Backtracking) (Tony's Skeleton Solution)

The solver searches the puzzle from left to right, top to bottom, attempting to find a digit from 1-9 in each blank cell until it satisfies the three constraints of the game rules. If the constraints cannot be satisfied whatever the value of the position is, the solver backtracks to the previous blank position and replaced the filled digit there into another new potential digit. In this way the solver continuously performs the search until all the potential solutions are searched out and recorded.

Solving Algorithm 2: Prioritize the Guessing Order (Extension 1)

In the solver level 0, I implement a function called expand which is to guess the value in a cell without considering the guessing order. Based on human logic, the simplest optimization figured out is to prioritize the guessing order, and that is what the Extension 1 requires. It provides a solution that we can fill the block with maximum candidates in it. Using the same logic mentioned in the assignment description, I defined a new data type Choices to describe the list of values/possibilities to choose for a single position. If the Choices for a unit cell contains only one value, this value must be the solution filled here. Therefore, we can always give preference to the position with the fewest potential possibilities (larger than 1), thereby reducing redundancy dramatically, especially when the Sudoku puzzle is complicated by naïve guessing.

Solving Algorithm 3: Constraint Propagation (Extension 2)

- **Hidden Singles (Solver level 1 in code)**

Optimize the Backtracking process by trimming the known value from the potential Choices of other cells which share block(row/column/box) with the known position. This constraint is consistent with the game rule that, in a block, one digit can merely occur once. (Note: column, row & box are all called as "block".)

1	8 6	9
8 6	4	5
7	2 ~~5~~ ~~3~~	3

To illustrate the operation, we consider an example above: due to the uniqueness principle, we have to delete the potential Choices of digit 3 and 5 from the second cell in the last row. The given circumstance is a box, the same with all other blocks (row/column) as well. So, each known unit cell can constrain 3 blocks related to it, thereby removing this known digit from the candidates of other cells in this 3 blocks and resulting in a dramatic decrease in redundancy.

- **Prune Naked Pairs (Solver level 2 in code)**

A 3*3 box is given below to explain this optimization:

1	8 6	9
8 6	4	5
7	2 ~~8~~ ~~6~~	3

As the given box illustrates, the digit(s) in each cell is the potential Choices. We consider a certain situation: if two cells in a block contain 2 same choice values 8 and 6, then no other cell contain any of them due to the simple reason that if there exist some other cells contained this 2 values, and a digit were selected, it is inevitable that one of the original cell would have no choice to fill and would the program would backtrack which led to redundancy. Therefore, we can remove this 2 values from all other related cells' Choice list to avoid meaningless backtracking and help to reduce redundancy. (Note: define "related cells" as those cells sharing blocks with the given cell.)

- **Prune Naked Triples (Solver level 3 in code)**

1 ~~2~~	2 8 6	~~8~~ 9
2 8 6	4	5 ~~6~~
7	2 8 6	3

This level just resembles pruning naked pairs: if there exist 3 cells containing 3 same values, remove these 3 digits from all other cells' choice list.

Generating Algorithm: Las Vegas Algorithm (Extension 3)

Jiang (2009) suggests that using digging holes method to generate a Sudoku puzzle by erasing several digits in confirmed cells of a terminal pattern. In order to boost the generation process, Jiang's team (2009) applies the greedy strategy to digging holes. Once a filled cell is dug out, the following operations are forbidden from filling another digit into the cell again.

- Check the whether the given Sudoku is valid
- Set constraints to the Sudoku when finished
- Check whether a Sudoku has a unique solution by a Depth-First-Search solver
- Prune the searching: Set Non-duplicates constraints in terms of each block)
- Perform propagating at a dug-out puzzle to raise the diversity of the output Sudoku

Implementation

1. **Basic Skeleton for Brute Force: (Solver - LEVEL 0)**
   This part consists of some basic functions, which completely follow the given assignment instructions.

   Three functions were constructed to constrain the validity of a Sudoku: *isSudoku* constrains the whether this puzzle is 9*9 representation; *okSudoku* constrains the duplicates; *noBlanks* constrains the Sudoku is finished without *"Nothing"* cell respectively.

   The given skeleton uses the Pos type to represent the cell position. In order to find all the blanks (*"Nothing"* cells), I indexed the Sudoku puzzle and mapped the *getBlanks* function to the whole puzzle after the previous map *getBlanksRow* and finally constructed the *allBlankPos.*
   Update and *(!!=)* are helper functions to change the position coordinate. I also add the QuickCheck property here to check whether the length remains the same after replacing an element.
   Solve_level_0 is a naïve searching without any optimization, Brute Force solution. The main purpose here is to discuss different situations and how to deal with the input. Apparently, Guards are quite useful. First, we have to discuss whether the input can be converted to a valid Sudoku. Since the input type is string but our main type during operating process is Sudoku, we have to construct *fromString* & *toString* to convert types and complete the IO function. In order to convert 81-character canonical string to a 9*9 Sudoku, firstly we have to group the string into a matrix (a nested list consisting 9 lists and each list has 9 members). Then we have to split the string as 9 sub-strings and covert the sub-string to a row (list). As helper functions, group, chunk and convert were constructed

   For the *boxs* function, it converts a matrix to a list of blocks(rows). Each row contains 3 elements and it has 3 rows for a box. The *chop* helps to split sub-lists into 3 elements. We zipped the 3-element groups using transpose. The last step is to flatten it using concat and map.

   This kind of Top-down logic is frequently applied in Haskell programming. The same with the *toString* part. Just analyse what we have and what we want, so what kind of function we should construct to help me to approach the expected output. That is why *selectRow* is constructed in *toString* part.

2. **Prioritize the Guessing Order: (replace expand by expandFirst Function)**
   Bird (2006) points out in his paper: A program to solve Sudoku. The best choice of cell to perform expansion is the smallest-choices one (greater than 1). *min_choice* can also be improved by adding a breaking situation that when a cell only has 2 choices, it will be selected and the filter can be ended.

3. Constraint Propagation

1) Hidden Singles (Solver - LEVEL 1)

First, we have to filter the single value choice list, and remove it using *minus*, and then the pruning part: according to the calculation given by Bird (2006) based on 3 laws:

filter (p . f) = map f . filter p . map f

filter (all p) . cp = cp . map (filter p)

filter nodups . cp = filter nodups . cp . reduce

Let f is one of the rows/cols/boxes,

filter (all nodups . f) . mcp = filter (all nodups . f) . mcp . pruneBy f

since,

filter p . filter q = filter q . filter p (in this case)

we can finally rewrite:

prune = pruneBy boxs . pruneBy cols . pruneBy rows

Calculation process is shown below (Bird, 2006):

Here is the calculation. Let f be one of *rows*, *cols* or *boxs*:

$$
\begin{aligned}
& \mathit{filter}\ (\mathit{all\ nodups} \cdot f) \cdot \mathit{mcp} \\
= \quad & \{\text{since } \mathit{filter}\ (p \cdot f) = \mathit{map}\, f \cdot \mathit{filter}\ p \cdot \mathit{map}\, f \text{ if } f \cdot f = \mathit{id}\} \\
& \mathit{map}\, f \cdot \mathit{filter}\ (\mathit{all\ nodups}) \cdot \mathit{map}\, f \cdot \mathit{mcp} \\
= \quad & \{\text{since } \mathit{map}\, f \cdot \mathit{mcp} = \mathit{mcp} \cdot f \text{ if } f \in \{\mathit{boxs}, \mathit{cols}, \mathit{rows}\}\} \\
& \mathit{map}\, f \cdot \mathit{filter}\ (\mathit{all\ nodups}) \cdot \mathit{mcp} \cdot f \\
= \quad & \{\text{definition of } \mathit{mcp}\} \\
& \mathit{map}\, f \cdot \mathit{filter}\ (\mathit{all\ nodups}) \cdot \mathit{cp} \cdot \mathit{map\ cp} \cdot f \\
= \quad & \{\text{since } \mathit{filter}\ (\mathit{all}\ p) \cdot \mathit{cp} = \mathit{cp} \cdot \mathit{map}\ (\mathit{filter}\ p)\} \\
& \mathit{map}\, f \cdot \mathit{cp} \cdot \mathit{map}\ (\mathit{filter\ nodups} \cdot \mathit{cp}) \cdot f \\
= \quad & \{\text{property of } \mathit{reduce}\} \\
& \mathit{map}\, f \cdot \mathit{cp} \cdot \mathit{map}\ (\mathit{filter\ nodups} \cdot \mathit{cp} \cdot \mathit{reduce}) \cdot f \\
= \quad & \{\text{since } \mathit{filter}\ (\mathit{all}\ p) \cdot \mathit{cp} = \mathit{cp} \cdot \mathit{map}\ (\mathit{filter}\ p)\ \} \\
& \mathit{map}\, f \cdot \mathit{filter}\ (\mathit{all\ nodups}) \cdot \mathit{cp} \cdot \mathit{map}\ (\mathit{cp} \cdot \mathit{reduce}) \cdot f \\
= \quad & \{\text{since } \mathit{map}\, f \cdot \mathit{filter}\ p = \mathit{filter}\ (p \cdot f) \cdot \mathit{map}\, f \text{ if } f \cdot f = \mathit{id}\} \\
& \mathit{filter}\ (\mathit{all\ nodups} \cdot f) \cdot \mathit{map}\, f \cdot \mathit{mcp} \cdot \mathit{map\ reduce} \cdot f \\
= \quad & \{\text{since } \mathit{map}\, f \cdot \mathit{mcp} = \mathit{mcp} \cdot f \text{ if } f \in \{\mathit{boxs}, \mathit{cols}, \mathit{rows}\}\} \\
& \mathit{filter}\ (\mathit{all\ nodups} \cdot f) \cdot \mathit{mcp} \cdot f \cdot \mathit{map\ reduce} \cdot f \\
= \quad & \{\text{definition of } \mathit{pruneBy}\, f\text{; see below}\} \\
& \mathit{filter}\ (\mathit{all\ nodups} \cdot f) \cdot \mathit{mcp} \cdot \mathit{pruneBy}\, f
\end{aligned}
$$

The definition of *pruneBy* is

$$\begin{aligned}
\mathit{pruneBy} \quad &:: \quad (\mathit{MatrixChoices} \to \mathit{MatrixChoices}) \to \\
&\qquad (\mathit{MatrixChoices} \to \mathit{MatrixChoices}) \\
\mathit{pruneBy}\, f \quad &= \quad f \cdot \mathit{map}\ \mathit{reduce} \cdot f
\end{aligned}$$

We have shown that, provided f is one of *rows*, *cols* or *boxs*,

$$\mathit{filter}\ (\mathit{all}\ \mathit{nodups} \cdot f) \cdot \mathit{mcp} \quad = \quad \mathit{filter}\ (\mathit{all}\ \mathit{nodups} \cdot f) \cdot \mathit{mcp} \cdot \mathit{pruneBy}\, f$$

For the final step we need one more law, the fact that we can interchange the order of two *filter* operations:

$$\mathit{filter}\ p \cdot \mathit{filter}\ q \quad = \quad \mathit{filter}\ q \cdot \mathit{filter}\ p$$

This law is not generally valid in Haskell without qualification on the boolean functions p and q, but provided p and q are total functions, as is the case here, the law is OK. Indeed we implicitly made use of it when claiming that the order of the component filters in the expansion of *filter correct* was unimportant.

Now we can calculate, abbreviating *nodups* to *nd* to keep the expressions short:

$$\begin{aligned}
& \mathit{filter}\ \mathit{correct} \cdot \mathit{mcp} \\
= \quad & \{\text{rewriting } \mathit{filter\ correct} \text{ as three filters}\} \\
& \mathit{filter}\ (\mathit{all}\ \mathit{nd} \cdot \mathit{boxs}) \cdot \mathit{filter}\ (\mathit{all}\ \mathit{nd} \cdot \mathit{cols}) \cdot \mathit{filter}\ (\mathit{all}\ \mathit{nd} \cdot \mathit{rows}) \cdot \mathit{mcp} \\
= \quad & \{\text{calculation above}\} \\
& \mathit{filter}\ (\mathit{all}\ \mathit{nd} \cdot \mathit{boxs}) \cdot \mathit{filter}\ (\mathit{all}\ \mathit{nd} \cdot \mathit{cols}) \cdot \mathit{filter}\ (\mathit{all}\ \mathit{nd} \cdot \mathit{rows}) \cdot \mathit{mcp}\ \cdot \\
& \mathit{pruneBy}\ \mathit{rows} \\
= \quad & \{\text{interchanging the order of the filters}\} \\
& \mathit{filter}\ (\mathit{all}\ \mathit{nd} \cdot \mathit{rows}) \cdot \mathit{filter}\ (\mathit{all}\ \mathit{nd} \cdot \mathit{boxs}) \cdot \mathit{filter}\ (\mathit{all}\ \mathit{nd} \cdot \mathit{cols}) \cdot \mathit{mcp}\ \cdot \\
& \mathit{pruneBy}\ \mathit{rows} \\
= \quad & \{\text{using the calculation above again}\} \\
& \mathit{filter}\ (\mathit{all}\ \mathit{nd} \cdot \mathit{rows}) \cdot \mathit{filter}\ (\mathit{all}\ \mathit{nd} \cdot \mathit{boxs}) \cdot \mathit{filter}\ (\mathit{all}\ \mathit{nd} \cdot \mathit{cols}) \cdot \mathit{mcp}\ \cdot \\
& \mathit{pruneBy}\ \mathit{cols} \cdot \mathit{pruneBy}\ \mathit{rows} \\
= \quad & \{\text{repeating the last two steps one more time}\} \\
& \mathit{filter}\ (\mathit{all}\ \mathit{nd} \cdot \mathit{rows}) \cdot \mathit{filter}\ (\mathit{all}\ \mathit{nd} \cdot \mathit{boxs}) \cdot \mathit{filter}\ (\mathit{all}\ \mathit{nd} \cdot \mathit{cols}) \cdot \mathit{mcp}\ \cdot \\
& \mathit{pruneBy}\ \mathit{boxs} \cdot \mathit{pruneBy}\ \mathit{cols} \cdot \mathit{pruneBy}\ \mathit{rows} \\
= \quad & \{\text{definition of } \mathit{filter\ correct}\} \\
& \mathit{filter}\ \mathit{correct} \cdot \mathit{mcp} \cdot \mathit{pruneBy}\ \mathit{boxs} \cdot \mathit{pruneBy}\ \mathit{cols} \cdot \mathit{pruneBy}\ \mathit{rows}
\end{aligned}$$

Hence, we can define *prune* by

$$\begin{aligned}
\mathit{prune} \quad &:: \quad \mathit{MatrixChoices} \to \mathit{MatrixChoices} \\
\mathit{prune} \quad &= \quad \mathit{pruneBy}\ \mathit{boxs} \cdot \mathit{pruneBy}\ \mathit{cols} \cdot \mathit{pruneBy}\ \mathit{rows}
\end{aligned}$$

Readers who gave this solution (or a similar one in which the three components appear in any other order) can award themselves full marks.

I marked the prune as prune_level_1 in my code since I also implemented prune_level_2 and prune_level_3 as well. These three functions are similar to some extent.

2) Naked Pairs (Solver - LEVEL 2)

The first is how to get the same pairs, I defined a function *to_samepair* with QuickCheck property. It can find the repeating pairs in the nested list and return a list with 2 elements, that is, the duplicate pair. *Reduce_samepair* takes a list of choice list and prune the "naked pairs".

I imported isInfixOf to check the sub-list, but there still exists redundancy since I did not remove the single value from other related cells. In other words, I just deal with this situation:

Assume cell 1,2,3 are in the same block.
Choices for cell 1: [Just 2, Just 3]
Choices for cell 2: [Just 2, Just 3]
Choices for cell 3: [Just 2, Just 3, Just 4]
…
Then the choices for cell 3 will be pruned to [Just 4]
However, when it comes to another situation:
Assume cell 1,2,3,4 are in the same block.
Choices for cell 1: [Just 2, Just 3]
Choices for cell 2: [Just 2, Just 3]
Choices for cell 3: [Just 2, Just 4]
Choices for cell 4: [Just 3, Just 5, Just 6]
…
If this algorithm is implemented well, it will find *dup_pair* = [Just 2, Just 3], and the choice for cell 3 will be updated by [Just 4] (removing Just2); similarly, the choice for cell 4 will be updated by [Just 5, Just 6] (removing Just 2).
I once wrote code like:

```
reduce_samepair x = [if (dup_pair == xs) then xs else xs\\dup_pair | xs <- x]
```

it works well for easy.txt with 0.6s by expandFirst & prune_level_2 but when it comes to hard.txt, nothing is printed. Probably it did not check the validity then lead to bugs.
Therefore, I changed the code to:

```
[if (isInfixOf dup_pair xs)&&(dup_pair/=xs) then xs\\dup_pair else xs | xs <- x]
```

isInfixOf is a built-in function to check the sub_list and returns a Boolean type.
This condition statement is much more restricted and it can be run successful with a 3m58.542s using expandFirst and level 2.

3) Naked Triples (Solver - LEVEL 3)

The first is how to get the same triples, I defined a function *to_sametriple* with QuickCheck property. It can find the triples occurring 3 times in the nested list and return a list with 3 elements, that is, the 3-times triples. I constructed the *count* function to find the frequency of an element in a list. And the anonymous function and *filter* help to write concise code.
Also I changed my previous code:

reduce_sametriple x = [if (dup_triple/=xs) then xs\\dup_triple else xs | xs <- x]

to:

[if (isInfixOf dup_triple xs)&&(dup_triple/=xs) then xs\\dup_triple else xs | xs <- x]

except the *count* function and filter 3-times triples which are slightly different from level 2, other functions are quite similar, *prune_dup_triple*, *reduce_same_triple* as well as the concise version of *pruneBy* function.

Besides, helper functions were constructed: is_single, is_pairs and is_triple as well as getDups.

4) Enforce Consistency Property

After implementing 4 levels (including the level 0: brute force), we have to set the winner condition: *complete*. The puzzle is completed when all cells' choice list only contains one valid value. If any position in the puzzle has no choice, this Sudoku is unsolvable. *Consistent* constrains the "uniqueness" game rule and satisfy apply consistent to each row, column and box.

4. Generating Sudoku (Extension 3)

- based on a paper Named "Sudoku Puzzles Generating: From Easy To Evil"

Test & Analysis (Extension 4)

Easy.txt

	Solver - Level 0 **Brute Force**	**Solver - Level 1** **Hidden Singles**	**Solver - Level 2** **Naked Pairs**	**Solver - Level 3** **Naked Triples**
expand **Note:** **(Level 0 doesn't use)**	**real 5m45.574s** **user 5m42.720s** **sys 0m4.686s**	**real 5m53.522s** **user 5m49.372s** **sys 0m4.010s**	**NULL**	**NULL**
expandFirst	**NULL**	**real 0m1.175s** **user 0m1.184s** **sys 0m0.284s**	**real 0m0.695s** **user 0m0.720s** **sys 0m0.255s**	**real 0m0.789s** **user 0m0.814s** **sys 0m0.266s**

Hard.txt

	Solver – Level 0 Brute Force	Solver – Level 1 Hidden Singles	Solver – Level 2 Naked Pairs	Solver – Level 3 Naked Triples
expand	**Killed**	**24min27s**	**NULL**	**NULL**
expandFirst	**NULL**	**real 4m13.680s user 4m11.827s sys 0m3.315s**	**real 3m58.542s user 3m56.831s sys 0m3.264s**	**real 5m13.327s user 5m10.379s sys 0m2.624s**

Since "Naked triple" is not very general so Solver-level 3 performs a lot of invalid checks that leads to the relatively slower than Solver_level 2.

But we can obviously find the dramatic decline in time if we replace the naïve expand by expandFirst. The guessing order makes a huge difference indeed!

Sudoku17.txt

To deal with this huge file, I found a very fast solver online (https://wiki.haskell.org/Sudoku#Very_fast_Solver) and I tested it for the Sudoku17.txt, only taking 43.898s to print all the output, approximately 50000 Sudoku!). Data.vector is imported in this solution and this library has a wide range of built-in functions and the complexity of the majority is merely O(1)! Indexing, update, imap, ifilter are all built-in. It is so awesome that this Data.vector library seems to be born to solve Sudoku. Besides, the solution provided applies the Monad in an extremely skilled manner, which also enables to boost the solver.

Acknowledgement

I really appreciate those people who shared their thoughts online and the open source platform online. Sudoku is so attractive for Haskell programmers that they even created a wiki Haskell page (https://wiki.haskell.org/Sudoku) especially for Sudoku, which I really benefited from.

Reference:

BIRD, RICHARD. "FUNCTIONAL PEARL: A Program To Solve Sudoku". Journal of Functional Programming 16.06 (2006): 671. Web. 7 July 2006. http://www.cs.tufts.edu/~nr/cs257/archive/richard-bird/sudoku.pdf

Functions for solving a Sudoku https://hackage.haskell.org/package/hsudoku-0.1.1.0/src/src/Sudoku/Solver.hs

[Haskell-cafe] Norvig's Sudoku Solver in Haskell https://mail.haskell.org/pipermail/haskell-cafe/2007-August/031049.html

Human-like algorithms for solving Sudoku http://www.programming-algorithms.net/article/42521/Sudoku

Jiang, Biaobin. "Sudoku Puzzles Generating: From Easy To Evil". Chinese Journal of Mathematics in Practice and Theory 39.21 (2009): 1-7. Web. 26 May 2017. http://zhangroup.aporc.org/images/files/Paper_3485.pdf

Open source project on github: https://github.com/Freezard/haskell-sudoku/blob/master/Sudoku.hs

Problem 97 in 99 Haskell Problems http://mfukar.github.io/2015/12/11/haskell-xvi.html

Wiki Haskell: A very fast solver https://wiki.haskell.org/Sudoku#Very_fast_Solver