Solving Sudoku and the Efficiency of a Haskell Program

Report for COMP1130 Assignment 3:Sudoku

Zichen Zhang

UID: U6161816

U6161816@anu.edu.au

Date: 26/05/2017

Collaborators: None

Report Structure

Background

Algorithm

Implementation and Optimization

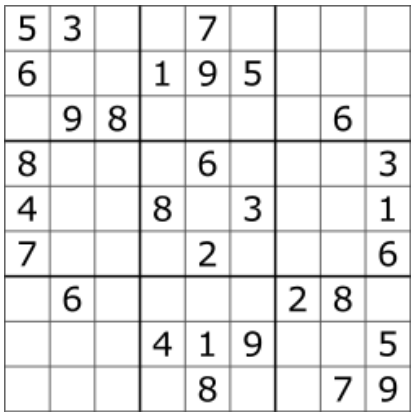
Test Result

Resources

Background

Sudoku is a logic-based, combinatorial number-placement puzzle. The objective is to fill a 9×9 grid with digits so that each column, each row, and each of the nine 3×3 subgrids that compose the grid (also called "boxes", "blocks", or "regions") contains all of the digits from 1 to 9. The puzzle setter provides a partially completed grid, which for a well-posed puzzle has a single solution.

(source: https://en.wikipedia.org/wiki/Sudoku)



The objective of this report is to explain the algorithm I used to approach the puzzle and how different algorithms affect the efficiency of our programs.

Algorithm

One of the most common algorithm to solve a Sudoku is by trying every possible value (e.g. 1..9) on every blank square, followed by testing whether the result matrix is a solution to our Sudoku. We call this algorithm “the brutal way” as it is direct and easy to implement. However, the cost of using the brutal way is a huge amount of time and space, and in some circumstance, it may take hours or days to solve a single Sudoku.

Why is the brutal way so inefficient? The main reason is that it involves too many steps. Let’s model the brutal way as a tree with any one of the blank cell of Sudoku to be the top node. As we know, we have 9 choices of values to fill in that blank cell, which we can model as the nodes under the top node. Then at depth 2, we have 9 subtrees and at depth 3, we have 9^2 and so on. So we need to try 9^n times to solve a Sudoku with 9 blank cells. It doesn’t make a big difference when we are dealing with easy Sudoku but apparently, as the number of blank cells increases, the workload of the computer grows exponentially.

As mentioned, the brutal way is very inefficient when it comes to a harder Sudoku. But can we just do it in another way which doesn’t need to try values at all? The answer seems to be No. As I researched about NP-complete problems, there is no efficient algorithm without tries of values. But we can improve this algorithm by cutting down the choices of values for nearly each blank cell by applying propagations.

I applied the following methods to limit the choice of values in my implement:

1. When we can confirm that a value must be in one cell (e.g. the given values), we eliminate that value from the possible value range of its peers (entries on the same row, the same column and the same 3*3 block).
2. When a cell must contain a value (e.g. the cell is the only one on its row which is possible to contain 1), we also eliminate the value from its peers.
3. If there is no other way to limit the choice of values of the cells on the Sudoku, we try from a cell with the least possible values. (e.g. if we try from a cell which is possible to have 6 values, the possibility of trying the wrong value is 5/6. But if we pick a cell with 2 choices at first, the possibility becomes 1/2.)

By applying the above rules, the number of possibilities is greatly reduced and for most of the Sudokus, my program can solve them in less than 0.5 second instead of few hours.

Implementation and Optimization

First of all, I want to clarify that I implemented all the functions (except the Solver) given on the assignment page. However, I didn't use them as I wrote a set of new functions. The main difference of my own functions is the model of a Sudoku. After some research, I decided to model the cells of a Sudoku as a tuple, containing the possible values on that entry and a character 'y' or 'n' to denote whether I eliminate its value (if it is confirmed) from its peers. This model is not only clear to see our progress to solve the Sudoku, but also ease our workload when we need to try values one by one.

Here are some things I want to mention for individual functions:

boxes : a 3*3 box contains the first three/ fourth to sixth / seventh to ninth elements of three adjacent rows. So each time I took 3 from each row, concat them together and then move forward. When the rows have no element to take (e.g. when I already took three times), we move to the next 3 rows.

blank: I implemented this function twice to optimize it. In my first implement I arrange the entries as a string so I can get the coordinates of one entry by using the numeric property of a list (div and mod). Then I improved it as shown on the script.

Then from the function "update", I implemented them in my own way.

matrixIsValid: the function is designed this way because the assign function does two things: assign a value and call eliminate function to eliminate the value from the entries of its block. So if a value appear twice in the same block, one must appear later than the other, so the later one will delete this value from the list of possible values of the earlier one, then an empty list is produced. To check the matrix is valid or not, we just have to check whether there is a empty list.

Test Results

All of the tested I defined return true.

The time taken to solve a easy Sudoku is 0.01s.
The time taken to solve all the easy Sudokus is 0.703s

The time taken to solve a hard Sudoku is 0.8s to 3s
The time taken to solve all the hard Sudokus is 2min.

Resources

https://www.haskell.org/hoogle/ (for checking the uses of preclude function)
http://norvig.com/sudoku.html (research about algorithm)