SudokuSolvers

By Gerrit Vos and Joris Voogt

31 Maart 2014

Inhoudsopgave

Hoofdstuk 1

Introduction

Japanese sudoku puzzles are immensely popular throughout the entire world; hundreds of thousands of sudokus of greatly varying difficulty exist. Of course, many programs to solve a sudoku with the computer also exist. For the course Algorithms and Datastructures, we decided to make our own version of a sudokusolver program. In this report, we will explain the ideas behind our algorithm thorougly and briefly discuss the tests we designed to test our methods. Finally, we will give a bit of evalution on the whole process.

Hoofdstuk 2

Method

2.1 Problem Analysis

We need to find a method that effectively finds a solution to a given partially filled Sudoku. Firstly, we need to make an easy way to manually enter a sudoku in the program. It then needs to be saved in a data structure, most likely a (9 by 9) table. We'll also need methods to check whether filling in a certain number breaks the rules of a sudoku. Finally, we need to find a recursive pattern that can try all possible options and find the correct solution for us. The last problem is obviously the main problem, so we'll treat that in a seperate section. The other problems will be tackled below.

2.1.1 Data structure

Of course, we chose the data type of a table to implement our sudoku. It works most practically when assigning numbers to a certain cell and checking the existence of a number in a row or column. We used the implementation of the Table class that we made for this course during practicum. One disadvantage of a table is that it is basically a 2-dimensional array-object, which means the cells are 'numbered' from 0 to 8 instead of 1 to 9. This was rather annoying to work with at some times and might cause some confusion later in the report.

We also made the choice to use the object 'Punt' to represent the coordinates of a cell. It is definitely possible to work without this class, but we chose to use it because it makes it easer to perform checks and systematically go along cells in the sudoku.

2.1.2 Manually entering a sudoku

2.1.3 Checks

For every number a we fill in, we'll need three different checks: We need to check the row, the column, and the 3 by 3 square for the presence of that number a before we fill it in.
Checking the rows and the columns is not hard; we can simply check every cell in the respective column or row with a for-loop. Of course, we do this before

we actually fill in the number.
Checking the 3 by 3 square is much more of a challenge. Firstly, the program would need to find out in what square the cell is. Then it needs to check every cell in that square. Of course, it is possible to do this within the table object, but we chose a different approach. We decided to represent the 9 'blocks' of the table in a new 3 by 3 table that we call 'Blokken'. This is illustrated in the figure below.

Every cell of this table contains a BitSet that contains numbers from 1 to 9 (in practice 0 to 8). For example, if the number 3 is present in the top middle block, the variable Blokken[0][1][2] would have the value true. Blokken is therefore a 3-dimensional boolean array with measurements 3 by 3 by 9.

To implement this, we wrote a method called 'nummerVierkant' to find the appropriate cell in the 3 by 3 square given a cell in the sudoku. Say, for example, we want to find the block which includes the cell (0,3) (row number 1, column number 4). This would be the middle block on the left, which would have coordinates (0,1) in 'Blokken'. We need a formula that correctly assigns the row-number to 0 and the column number to 1. However, the cell (2,5) (row number 3, column number 6) should point to the same block. In practice, the formula would need to assign the number 0-2 to 0, 3-5 to 1 and 6-8 to 2. The formula that finds the correct number a between 0 and 2 for a given number x between 0 and 8 is therefore:

$$a = \frac{x - x\%3}{3}$$

The actual method for checking the block for a certain number x now becomes really easy. We use nummerVierkant to find the correct block, and then return the value belonging to x we find in the corresponding BitSet.

2.2 Solutions

Now all that's left is the solving method 'LosOp' itself. This method is recursive; every call of LosOp will work on a single cell in the sudoku. It will fill in the numbers 1 through 9, checking for every number whether it breaks the sudoku rules or not. If it does, it tries the next number, and if it doesn't, it fills in the number in the table and ik Blokken and calls 'LosOp' in the next cell. If LosOp has tried to fill in all numbers in a cell unsuccesfully, it will terminate, leaving the LosOp working on the previous cell to continue. The previous LosOp method will then put the respective value in Blokken back to false and attempt the next number.

To make this backtracking idea work, we fill in zeros in every empty cell of the sudoku. LosOp will only run through the numbers on a cell if the number that is in there at the beginning is a zero; of course, we don't want it to change the number in a cell that was given at the start. If the number at the beginning is not a zero, it will simply call LosOp on the next cell immediately. If a LosOp call had checked all the numbers in its cell unsuccesfully, it will change the number in its cell back to zero so that the cell can be researched again later

on.

There are two special cases within this method. The first special case happens when the LosOp-method working on the first cell has tried all 9 numbers. In this case, there are no (more) solutions, and the method will simply print 'Geen oplossingen'.
The other special case happens when the LosOp-method working on the last cell fills in a number that's correct (in other words, when we fill in a correct number in the last cell). It will then need to print the sudoku it found, and go back to find other solutions. To check whether there is a next cell (and if yes, what it is) we created a method called 'next '. Calling next with a 'Punt' object (x,y) will return the Punt object (x, y+1) (we work from left to right) unless the y-coordinate whas already 8. In this case, it will return (x+1, 0). If the x-coordinate is also already 8, it will return (-1,-1), and the LosOp method will know that it was the last cell.
However, we do not want the method to continue finding other solutions immediately when a solution is found. Rather, it needs to print the solution in the graphic interface, and then 'pause' continuing only when the button 'find next solution' is clicked. To implement this, we made use of a

Hoofdstuk 3

Tests

To test our methods we created serveral testing methods. For testing our check methods we simply filled in the 9x9 table manually in a test method and proceeded to run checks on several rows, columns and blocks and checked if the outcome was correct. When LosOp was almost finished (which was before the graphical inteface) we just typed a sequence of 81 numbers (zeros for the empty spaces) and ran it through LosOp to see if it printed the correct solution. As an ultimate test we ran a sudoku from the telegraph that was advertised as the 'hardest sudoku ever'. By running it we actually found some errors, but after fixing them we got the correct solution.

Hoofdstuk 4

Evaluation