SudokuSolvers

By Gerrit Vos and Joris Voogt

23 juni 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

First, we made use of a scanner that would scan 81 successive numbers in a long row that the user would enter. These included zeros for all empty spots. It initially worked for testing our method, but it was obviously not very user-friendly.
That's why we decided, when we had some time left, to make a graphical interface where the user can simply enter the numbers of the sudoku he wants to be solved. In this interface, it isn't necessary to enter zeros, which dramatically

reduces the difficulty of entering the sudoku.

There should be noted that the code we wrote for the graphical interface asks for more knowledge than the courses 'Inleiding Programmeren' and 'Algoritmen en Datastructuren' provided. Therefore we used the site 'Stack Overflow[1]' and the book[2] that came with the course 'Inleiding Programmeren'. As a result, our code for the graphical interface and the adjusted code to solve a sudoku may contain less acurate methods and/or variables which might seem unnecessary.

The graphical interface is a seperate class and makes use of a JFrame with a card-layout for easy switching JLabels, JButtons and JTextFields.
The idea is that the user inputs the sudoku that he/she wants to have solved in a 9 by 9 table, thus 81 textfields.
To make sure that the input are only the numbers 1 through 9, we use JFormattedTextField. Thus we only have to check if the numbers the user filled in could possibly form a sudoku.

Now with help of the card-layout, a press of the button 'solve' will change the cards and output 81 new normal JTextFields with the solution or the line 'No Solutions'. Also, the button 'solve' changes to respectively the button 'Another Solution' or disappears.
Again using the card-layout, 'Another Solution' either gives an other solution for the sudoku or it will state that no more solutions exist after which the user can press the button 'Previous Solution' to get the last found solution to the sudoku back on screen.

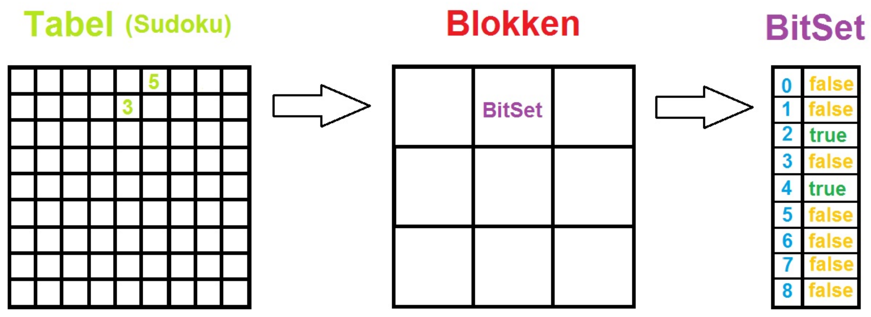
The reset button is obviously used to clear all data from the input sudoku and therefore giving a clean beginning screen.
To get this result, we use threads. There's one thread which will run through our LosOp-method and by using a semaphore, we can make the thread give one solution of a multiple solutions sudoku and give another one when the user presses 'Another Solution'. (We will explain the use of semaphores more thoroughly in the section 'Solutions')

The rest of the threads are used to keep the program together as the main thread runs faster than the LosOp-thread.
As we already mentioned earlier, we use a lot of methods and variables to get our graphical interface. The comments in our code should suffice to make clear what these methods and variables do.

[1]See references (1)
[2]'Java in two semesters', see references (2)

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 3, column number 0). 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 5, column number 2) 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 in 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 change the variable solutions to false which will initialize a 'No Solutions' Label in the graphical interface.
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+1, y) (we work from left to right) unless the x-coordinate was already 8. In this case, it will return (0, y+1). If the y-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 graphical interface, and then 'pause' continuing only when the button 'Another Solution' is clicked. To implement this, we made use of a semaphore. As semaphores are able to grant threads permission one at a time, a semaphore seems to be a good solution to make sure that only one solution to a multiple solutions sudoku is printed each time the user presses 'solve' or 'Another Solution'.

The real challenge here is to know what to do in the different situations. An empty sudoku has so many solutions that the user will never check them all out, so the current thread running will continue as we run another thread for finding solutions to a different sudoku. This means we have to make sure that when not all solutions to a sudoku have been seen by the user, the program first ends the current thread and afterwards starts the new thread which will run the same method. Here we use a lot of variables and thus also methods to get or set these variables to make sure that in every case the reset button actually resets the graphical interface and that no more threads are running in the background.

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

Gerrit

Perhaps the hardest part about this whole project was thinking of a suitable problem that was difficult enough but not too hard. I think we spent at least 2 or 3 weeks just considering things. The result was that we started of relatively slow. What didnt help was that as soon as we had found a problem and got it approved by our tutor, we started trying to work with GitHub. This gave us so much trouble that we were another two weeks further before we actually began with our code, at which point we still didn't get the hang of GitHub. It truly gave us trouble until the very end.

As soon as we started on the actual programming, the work became more fun. We made quick progress, and we had a working code half a week before the previewed hand-in date. So Joris decided to try and make a graphical user interface to make entering a sudoku easier. And, spending a lot of his time, he got it working with little of my help. Fortunately he seemed to enjoy it so I dont have to feel guilty. I am really grateful for all his hard work. In return, even though it didnt take nearly as much time, I typed the major part of the report.

Joris

For me, this was 'the' challenge of programming this year. But more important, it was the most fun. Though it was hard to come up with a good problem which we could actually solve, i'm happy we chose this problem as it made me like programming even more. Personally, i've experienced the collaboration between me and Gerrit as a fun and highly educational time. As we did the practicum of 'Algoritmen en Datastructuren' together, we knew what to expect from each other and that paid off.
The actual code that looks for every possible solution for a sudoku was pretty fast written, though we could really use that, as GitHub took up a lot of our time. To make the input of our solver easier, a graphical interface seemed a good solution. It was, but it took by far the most time, not only to understand all the new methods but also to see and find issues which it brought. Nevertheless, I enjoyed the programming.

Hoofdstuk 5

References

The website 'StackOverflow.com' (1)
Mostly used to find answers on forums on how to use certain methods or find methods, for example, semaphores, JFormattedTextField and threads.

The book 'Java in two semesters' by Quentin Charatan & Aaron Kans, Third Edition. (2)
Mostly used for the card-layout and refreshing how to use graphics in Java.