Second Year Project - Operations Research
Solving Sudoku (Sports Scheduling)

Vincent Kreuzen, Maastricht University

1 Sudoku

You probably have heard of, or even solved, a puzzle commonly known as *Sudoku*. If not, have a look at the game on Wikipedia or `www.sudoku.org.uk`. A Sudoku is simply a game, but it shares quite a lot of characteristics with scheduling problems as they occur in practice. That the problem is not trivial can be seen by going to scholar.google.com. Submitting 'Sudoku' as the search term you can find quite a few relatively recent publications.

So what would be some practical applications of solving a Sudoku? Consider a sports competition where 9 teams play each other over the course of 9 weekends. Each team plays one home game and one away game against each opposing team. Additionally, each team plays two matches each weekend, except for one weekend where the team has a day off.

Now take a solved Sudoku and compare it to the above sports scheduling problem: Let the rows correspond to the home games for each team, and the columns to the away games. Each entry in a cell now tells us the weekend of the match, the row number tells us which team is playing the home game, and the column number which other team is visiting the match as an away game. The diagonal has equal row and column numbers; this gives us the weekend on which the particular team is not playing. The 3×3 blocks can be interpreted as follows: Assume we can categorize the teams in groups of three teams: high ranking teams, average ranking teams and low ranking teams. Then the requirement that a number between 1 and 9 occurs exactly once tells that each weekend, we want to have exactly one game between teams of any combination of strengths and home/away games.

Your solution procedures will combine two principles that can be found in many optimization algorithms. The first is pre-processing: you will implement a collection of logic rules that iteratively reduce the set of possible solutions. This is indispensable e.g. when solving large integer linear programming problems. The second is branching: when logic rules allow no further reduction, one can try all possible values for a specific variable (in our case, the content of one cell in the matrix). For each value, one gets a so-called *branch* in the search for a

solution. The idea is that new opportunities for pre-processing arise after fixing one variable to a particular value. However, it may happen that the chosen value was not feasible in the sense that one cannot find any feasible solution in that branch. In this case, one has to try another branch. Branching will be discussed some more later on.

The goal of the project is to implement a Java program with classes with appropriate methods to solve Sudoku puzzles of as high a difficulty as possible. For simplicity, we will restrict ourselves to the classical Sudokus of order 3, that is consisting of a 9×9 square. As a computer is fast, there is a straight forward approach: try all combinations. But there might be too many combinations to do so. For every cell without a hint, there are 9 possibilities, and given e.g. an instance with 60 free cells there will be 9^{60} combinations to be tested!

Therefore we need to apply some logic reasoning, as we do when solving a Sudoku by hand. A collection of ideas is presented in

http://www.sudoku.org.uk/PDF/Solving_Sudoku.pdf .

In this project, the focus will be on two basic rules explained in the following two sections, but you are free to choose to implement any additional rules. You are also required to implement a complete enumeration method which finds a solution, should the rules that you have implemented fail to solve an instance. These approaches, their implementations and the advisory data structures will be discussed in the following sections. Implementing the two basic logic rules will already allow you to solve simple Sudokus.

2 Data structures and a very simple logic rule

Given an instance, you will have some *hints*. A hint is a cell which already contains a number. Create a class `cell` which captures the state of a cell. In cells without hints, this data structure will initially contain all numbers from 1 to 9 as candidates for a solution. In cells with a hint, this data structure will contain the number. Our logic rules will reduce the set of candidates step by step, until exactly one is left. We refer to a cell as *solved* if we have reached this status.

The next class that you need is one that combines 81 cells to a Sudoku instance. Obviously, this class consists of some kind of 9×9 matrix of cells, plus member functions to operate on these cells in order to solve the Sudoku, plus some private data members that support these members functions.

The first logic rule is based on the following simple observation: Every cell belongs to exactly one row, one column and one block. Consider an unsolved cell. The only candidates for an unsolved cell are those numbers which are neither contained in a solved cell in this row, nor in a solved cell in this column, nor in a solved cell in this block. Therefore, all candidates in a cell which are solutions to cells in the same row, column or block can be removed from the set

of candidates. If one is lucky, one ends up with exactly one remaining candidate, which means that this cell is also solved.

The first logic rule works as follows. Start with all hints and use them to reduce the set of candidates of all other cells in the same row, column and block. While doing this, memorize whenever a cell is solved and use it as an additional hint. A good way to implement this is to organize a queue of hints (using for instance java.util class `PriorityQueue` or the class from Programming, `IntegerQueue`), initially fill it with all hints, process the queue (until it is empty), during which you add any solved cell as a new hint to the end of the queue. Do this until the queue is empty.

If implemented correctly, this logic rule solves the instance number 1 found on Eleum.

3 A generalization

This first rule can be generalized by taking approaching the problem from the following perspective: Consider a solution of a Sudoku. For each cell it tells you which number to write in this cell. A cell is thereby characterized by its row index and its column index. Now, instead of providing the number to be written for every combination of a row index and column index, you may also think of a solution in the following way: for every number (between 1 and 9) and every row, the solution tells you in which column of this row to write this number. A Sudoku is solved if for each number and each row, the number of column candidates is exactly one. Similarly, for every number and each block, the solution tells you in which of the nine positions of this block to write the number. Or, for every number and column, the solutions specifies in which row to put this number.

The key observation here is that the data structure as described before does not make transparent that for a particular combination of a row and number (or a column and a number, or a block and a number), there is only a single column (or row, or position) left where this number can go! In other words, we might reach a state where several cells in the same row contain more than one candidate, but some of these candidates appear only once in those cells, and therefore have to be written to this cell in order to find a solution. In this case, they will no longer be candidates for other cells in the same column, or for other cells in the same block.

This generalization of the first reduction method organizes a systematic search for such situations. Whenever it detects these, it can solve an additional cell. Newly solved cells can then be used to perform further reductions using the simple logic rules and this generalization.

4 Enumeration

Once the above logic rules are not sufficient to solve a problem instance, one needs to use a *branching algorithm* to enumerate possible solutions. Such a branching algorithm works as follows:

Take any cell for which the list of candidates has a size larger than 1. Take any number n in this list, delete all elements except n, and try to solve this simplified instance (using all the solved cells that have already been found at this point). Usually this guess will allow you to again apply the logic rules as described above to further reduce the problem. If this still does not help you further your search for a solution, you need to make a new guess in another cell. If you always guess correctly, this will solve the problem. But it can also happen that you guess wrongly, and the problem in a branch has no solution. Your algorithm will detect this when the logic rules suddenly create an empty cell. In this case, the search in this branch should be stopped, the algorithm should backtrack, and a new branch guessing a different number should be started. As soon as your enumeration has found a complete solution, it should stop.

The difficulty in implementing such an enumeration using branching lies in backtracking. That is, one might have to reverse some guesses - and all implications of these guesses - when a branch turns out to lead to infeasibility. The best way to deal with this difficulty is to enter every branch with a complete copy of the current situation. If the branch does not solve the problem, your algorithm will still have the original situation as it was before branching. From this you can make a new copy and enter another branch. If your implementation is sound, Java will use the Garbage Collection in order to free up memory slots which were used for branches which have been completely traversed and will no longer be used, thus making your implementation effective.

5 Deliverables

The following is a list of tasks for this project. For each task, there is a maximum number of points which can be obtained. You will need at least 55 points in order to get a passing grade for this part of the skills.

1. (20 points) Construct a collection of classes which stores the necessary data in appropriate data structures in order to capture all aspects of the Sudoku. Use the data structures as described in the sections above.

2. (15 points) Implement the simple logic rules. After this, your program should be able to solve Sudoku number 1.

3. (25 points) Implement the generalization as described in Section 4.

4. (25 points) Implement the enumeration algorithm from Section 5.

5. (15 points) Describe your implementation in a report of maximum 2000 words. You should shortly introduce the problem (max 500 words) and explain how you implemented the problem: describe the structure of your data structures and discuss your solution methods and their usage of the data structures. Close with a short concluding section. Do not add appendices or your code!

The quality of the program will be evaluated in terms of

1. readability and documentation,
2. organization of data structures and methods, and their interaction, clever usage of the java.util class library (do not reinvent the wheel!), and a clear distinction between the high-level logic of your solution procedure and the methods which are necessary to implement this logic.
3. efficiency in terms of omitting unnecessary computational work.

Your program has to make good use of Java classes and its libraries. In particular the logic rules should be implemented as methods of appropriate classes. Your algorithms should also keep track of some performance information, for example how often branching is used, or how often a particular logic rule helped to reduce the size of a list in a cell. If you implement several rules, count how often each of the rules helped to reduce the problem. This provides more information about the speed of your implementation than keeping track of the time; different computers yields different running times, but deterministic logic rules will always yield the same amount of rule-usage and branches, making a fair comparison possible. Include this performance information in your report.

Additional training instances can be found at

http://www.sudoku.org.uk/PDF/Solving_Sudoku.pdf

Submit your source files (.java files) and the report as a PDF file on studentportal (Eleum). The deadline for your submission is Friday, January 27, 2017 at 23:00. Points will be deducted for late submissions! Tutorials will be organized on Monday the 23rd, at the start of the second week of the skills period. Details will follow on Studentportal.

5.1 Presentations

Presentations will last 15 minutes each. For the programming presentations, please heed the following guidelines:

Use 1 minute to shortly introduce the problem (everyone should know this by now ;-))

Use 5 minutes to explain your general implementation: which datastructures and which methods for solving the sudoku did you implement?

Use 5 minutes to elaborate on one of your implementations. DO NOT PRESENT JUST THE CODE! Instead, describe and/or visualise your implementations (e.g. draw your datastructures as lists of lists, as I did during the lecture)

Use 1 minute for concluding remarks (e.g. did you solve everything? do you have ideas for improvement?)

Then there will be about 3 minutes left for questions and potential swapping between sessions.

6 File format

A file with an instance of a Sudoku starts with the number of hints, followed by as many rows as there are cells with hints. Each row in the file first gives you the row number of the Sudoku (a number between 0 and 8), then the column number of the Sudoku (a number between 0 and 8), and finally the hint (a number between 1 and 9). You can find 6 example files on Eleum.