Sudoku Solvers
A Different Approach

DD143X Degree Project in Computer Science, First Level
CSC KTH

Supervisor: Michael Minock

Christoffer Nilsson
Professorsslingan 10
114 17 Stockholm
Tel: 073-097 87 24
christn@kth.se

Mikaela Nöteberg
Nyborgsvagen 27
195 71 Rosersberg
Tel: 073-759 92 79
mnot@kth.se

16 maj 2012

Abstract

The purpose of this essay is to implement and study different techniques for solving sudoku puzzles, a problem similar to graph coloring with fixed size and dependencies. Three approaches are presented and compared regarding efficiency (time needed, space required and success rate). These approaches are rule-based solving, simulated annealing, and searching for solutions. Finally the parts found to be most important for an efficient solver are combined, creating an even better solver.

Sammanfattning

Syftet med denna uppsats är att implementera och studera olika tekniker för att lösa sudoku, ett problem som liknar graffärgning men med fast storlek och fasta beroenden. Tre tillvägagångssätt presenteras och jämförs angående effektivitet (tid, minne och korrekthet). Dessa tillvägagångssätt är regelbaserad problemlösning, simulated annealing och sökning. Slutligen kombineras de delar som visat sig vara viktigast för en effektiv lösare, för att skapa en ännu bättre lösare.

Contents

1 Introduction

Sudoku is a logic based puzzle with the goal to complete a 9×9 grid so that each row, each column and each of the nine 3×3 boxes contain all the numbers 1 through 9, given a partial filling to a unique solution. See example in figure 1.

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

(a) Problem

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

(b) Solution

Figure 1: A sudoku problem (a) and its solution (b).

The problem itself is a popular brainteaser but can easily be used as an algorithmic problem, with similarities to the graph coloring problem only with fixed size and dependencies. The modern version of Sudoku was introduced in Japan in the eighties and has since then attracted a lot of programmers and therefore there are a great deal of different sudoku solving algorithms available on the Internet, both implementations and construction methods.

Sudoku solvers are interesting because they are practical applications of very theoretical problems, and most people are familiar with them. Sudoku puzzles are commonly included in the crossword section of newspapers.

The purpose of this project is to investigate different techniques for solving sudoku. Sudoku itself can be solved using brute-force in a reasonable amount of time in most cases, but there are special cases where it takes a long time to brute-force. Therefore our task is to try to find efficient algorithms for all instances of the problem and evaluate them to make an efficient solver.

The different strategies we will look at are:

- Rule-based solving
- Simulated annealing
- Searching for solutions

These three have been chosen because their approaches are completely different. This will give an understanding about what is important in a sudoku solver and be useful to create an optimal solver.

1.1 Overview of the article

The article consists of 4 sections. The first one is Background were the theory needed for the remaining parts of the article is explained. It contains all general information about the three strategies. The background is followed by Approach where it is explained how these general strategies are applied to solving sudokus. In Implementation the data structures and other implementation details of the algorithms are described in detail. Finally Results show all graphs and test results received from the test runs of the programs.

1.2 Literature

Artificial Intelligence: A modern approach[1] by Stuart Russel and Peter Norvig is used to find information about searching for solutions in a more intelligent way than exhaustive search.

Complexity and Approximation - Combinatorial Optimization Problems and Their Approximability Properties[2] by Georgio Ausiello et al. describes the heuristic simulated annealing.

A set of logic rules for solving sudokus ranging from easy to hard is presented and explained in *The Logic of Sudoku*[3] by Andrew Stuart, these are supposed to be used by humans but many of them can be interpreted as pseudo code for writing algorithms.

1.3 Statement of collaboration

Although most parts of this project were done by us together as a group, some work were divided between us. We did most of the programming together, but for the rule-based solver, because of its extent, we divided up the implementations of the rules. Mikaela did the testing.

As to the Report, Christoffer wrote the majority of the sections concerning simulated annealing (2.2, 3.2, 4.2) and similarly Mikaela wrote the majority of the sections concerning searching for solutions (2.3, 3.3, 4.3). The sections on rule-based solving (2.1, 3.1, 4.1) were written by us together as a group, as well as the general parts of the report, including *Introduction*, *Results*, *Discussion*, and *Conclusion*.

2 Background

There are many different approaches to problem solving. We have investigated the following three techniques to base our sudoku solvers on.

2.1 Rule-based solving

Many problems can be solved using sets of different rules derived from logical reasoning. A rule-based solver uses a set of applicable rules by iterating over them to find matches to a problem and act according to the matching rule. As long as no solution is found, and there are matches, the iteration continues. A rule-based solver tends to act like a human solving a problem by hand.

If the rules are implemented in order of complexity, or probability of occurrence, the iteration can start over for every match found, since it means that the problem has changed. When the problem changes, it is likely that the simpler rules can be applied again and therefore you do not have to try the more complex rules until it is really necessary.

2.2 Simulated annealing

The simulated annealing algorithm is a heuristic algorithm for solving complex problems. This means that the algorithm does not always find the optimal solution to a problem. But on the other hand it generates a solution close to the optimal one in a reasonable amount of time considering the complexity of the problem.

There are three essential elements needed to be able to construct a functioning simulated annealing algorithm; a cost operator, a neighbor function, and a control parameter[2]. The cost operator takes a proposed solution and rates it according to some set of rules. For an optimal solution the cost is the lowest possible, in many cases zero. The neighbor function takes a proposed solution and randomly alters it generating a new solution. The control parameter is often referred to as the temperature and is used to control whether or not a change of solution with a higher cost is accepted.

Initially a proposed solution is generated, often randomly, and the cost of the the solution is calculated. After this the algorithm is iterative and does the same steps in each iteration[2]. These steps are:

1. Get a neighbor solution
2. Calculate the cost for the neighbor solution
3. Decide to keep the neighbor solution or to discard it
4. Decrease the temperature

For the first two steps the operations described earlier are used. The third step on the other hand is a bit more complicated. A neighbor solution is accepted if it is better than the current solution; having a lower cost or by a probability $e^{-\delta/t}$, where δ is the difference between the new cost and the current cost and t is the control parameter, i.e. temperature. The fourth step can easily be done for example with percentage; $t_{i+1} = t_i \times \alpha$ where α is between 0 and 1.

2.3   Searching for solutions

Generally a solution to any problem is a sequence of actions.[1] With a restricted problem instance it is always possible to find a solution by considering the different possible sequences. A search tree is built with the initial state at the root, where every branch is an action expanding a node, i.e. the current state, to another[1], which without further optimization results in an exhaustive search, in other words a brute-force algorithm.

The search algorithm works by following one branch at a time expanding the states until either a solution is found or there are no more actions to consider in the current sequence. To make the algorithm more intelligent it can check whether every state it passes is actually legal, eliminating "dead branches" as soon as possible.

Further optimization is possible by choosing which order the algorithm follows the branches, trying to eliminate as large parts of the search tree as possible. This is done by choosing the nodes in order of the least amount of further expanding, i.e. number of branches.

3   Approach

We intend to implement three different sudoku solving techniques. These implementations will be based on regular human rule-based solving, simulated annealing and searching for solutions (see Section 2, Background).

3.1   Rule-based solving

As mentioned in Section 2.1 the rule-based solver is going to solve sudokus as a human being would do; a human being good at solving sudokus. Since sudoku puzzles are meant to be solved by hand, naturally there are a number of rules one can apply to solve them. The solver will consist of several of these small algorithms and will apply them over and over until a solution is

found or until none of the rules no longer has any effect, meaning that the sudoku could not be solved using only these rules.

3.2 Simulated Annealing

As mentioned in Section 2.2 there are three parts needed in order to make the simulated annealing algorithm work. Two of these parts have to be applied to manipulate a sudoku instance. The initial solution is generated by assigning random numbers in each empty cell, making sure each box contains the numbers 1 through 9 exactly once.

The first part needed for the algorithm is the count operator that will answer the question, how good the proposed solution is for the sudoku. One way to do this is to simply add the number of missing digits in each row and in each column, see example in figure 2. The count operator can oversee the boxes if we always make sure that all boxes contain the numbers 1 through 9 in all other steps in the algorithm.

Row count

4	6	1	2	5	3	6	9	4	2
2	7	3	6	8	4	3	7	1	2
9	5	8	9	7	1	8	5	2	3
7	8	9	8	5	3	4	5	6	2
5	6	1	4	2	9	8	7	3	0
3	2	4	1	6	7	2	1	9	2
5	9	6	3	5	7	1	9	3	3
4	7	3	1	4	6	8	2	4	2
1	8	2	9	2	8	7	6	5	2
2	3	2	2	3	2	2	3	2	Σ 39

Col count

Figure 2: Example of count

Secondly a neighbor operator for a sudoku is required. The neighbor operator is going to swap a number of cell values around but keeping all boxes intact, containing the numbers 1 through 9 exactly once. It also has to keep all initial clues intact to guarantee that the final solution is in fact a solution to the given problem instance.

All other parts are identical to the explanation of the basic algorithm in Section 2.2 and do not require a special implementation for solving sudokus.

Another thing needed to be able to solve sudokus using simulated annealing is reheats. The algorithm can end up in a local minimum with a low

count and if the temperature is too low it would not be able to get to the global minimum. If this happens the algorithm has to be able to increase the temperature again and continue towards the actual solution.

3.3 Searching for solutions

Searching for a solution is a more intelligent way to try all possible digits in all free cells. To simply do this without any optimizations is what would be called an exhaustive search. This approach will solve most 9 by 9 sudokus in a reasonable amount of time, but will perform really bad for some cases (execution time about one hour on a modern computer). For the Search algorithm used for this project, the algorithm is altered to eliminate branches in the search tree as early as possible and shrinking the search space for the algorithm. This is done by looking at the box with the smallest amount of possible digits in each step of the algorithm.

3.4 Evaluation method

To measure an algorithm's performance there are four different criteria to have in mind; completeness, optimality, time complexity and space complexity.[1] Optimality is not applicable on evaluation of sudoku solvers since the problem instances by definition only have one possible solution, so the algorithms can not find solutions which are not optimal.

The working solutions will be benchmarked against each other regarding time and space needed and success rate, i.e., completeness, on sudoku puzzles of varying difficulties.[1] The testing will be conducted on a Macbook Pro with a 2.3 GHz Intel Core i5 processor and 8 GB RAM.

Based on the results of the first evaluation we will combine the best parts of each implementation to make a more efficient solver. The goal of this combined solver will be to have it solve as many puzzles as possible in a reasonable amount of time.

4 Implementation

All three projects were implemented using Java version 1.6, in order to be able to time the solvers and compare them against each other without taking

---

[1]With varying difficulties we mean puzzles with different numbers of already given digits. Our test data consists of 10 000 puzzles of each 45, 40, 35, 30, 25, and 17 digits given, collected from `http://www.printable-sudoku-puzzles.com/wfiles/` and `http://mapleta.maths.uwa.edu.au/~gordon/sudokumin.php`

into account the difference between programming languages. All of the implementations have the same way of storing a sudoku internally. This is done by keeping a 2-dimensional array with 81 integers (9×9 matrix) containing the current numbers for each cell.

The input, output, and timing for the solvers are also identical between the implementations and can easily be handled by the same class as long as the solver interfaces work the same way. This main class reads digits from the standard input and fills the 9×9 integer matrix row by row. When all 81 digits have been read it tries to solve the problem instance using the chosen solver algorithm. The solver's execution time, excluding the time needed to read the input, is stored as well as whether or not the solver was successful.

Then the main class continues to read from the input source and reads the following 81 digits as a new problem instance or a -1 indicating that this was the last problem. After trying to solve all sudoku puzzles of that input source, the main class also handles the output and prints the total time used to solve the puzzles as well as how many of the instances that were actually solved. Since there are a number of unknown digits in a problem instance 0 is used to indicate an empty cell. Below is an example of input consisting of two sudoku puzzles.

```
9 0 2 0 4 0 5 6 0 0 0 0 0 0 9 0 0 0 0 6 1 2 5 0 4 7 0 0 4 0 0 3
0 1 0 2 6 0 0 4 8 0 0 9 0 0 0 3 0 7 0 0 8 0 5 0 0 0 0 8 0 0 0 3
0 6 5 0 0 9 4 7 1 0 0 3 6 0 0 0 5
0 2 0 0 0 1 6 3 0 0 9 0 5 0 0 4 0 0 8 0 6 0 4 9 0 0 2 9 0 0 0 0
5 7 0 1 0 0 0 9 0 0 3 0 0 3 5 2 0 7 6 8 0 0 0 0 9 0 0 4 5 0 6 0
8 0 0 5 0 0 0 0 0 4 5 6 0 0 0 1 8
-1
```

4.1   Rule-based solving

The rule-based solver has to keep track of all possible digits for each cell. This is achieved by a $9 \times 9 \times 10$ boolean matrix b such as if b[i][j][k] is true, then k is a possible digit on row i, column j (k = 0 is unused since a sudoku only contains the numbers 1 through 9).

This solver was the most time consuming to implement, since there is not much reusable code between the rules and a lot of rules are needed to make a functional solver.

To make the algorithm easy to improve, i.e. add more rules for it to use, an interface is implemented describing how the main program should communicate with all the rules. The only method needed for all the rules is a method called invoke that takes a reference to the solver as a parameter

and returns a boolean. The reference to the solver is for the rule to be able to access public methods in the solver such as eliminating possibilities and setting digits in the solution. The boolean returned indicates if the rule made any change to the problem instance, which is used to know whether or not to retry the easier rules.

Since all rules implement this interface they can all be stored in a single array with the interface as its type. This makes it easy to iterate through all the rules and starting over when one of them returns true, or knowing that when we get to the end of the array this problem could not be solved using only the implemented set of rules.

Box/line reduction is an example of an implemented rule. Like many of the other rules it is used for eliminating possibilities in the cells. Looking at one row, if all possible cells for a certain digit are found to be in only one box, then that digit has to be in one of those cells. The digit is therefore not possible in, and can be removed from, all other cells in that same box. The rule works exactly the same way with columns instead of rows. It is implemented by going through one row at the time counting the number of boxes each digit is possible in. This is done for all digits at the same time, storing the result in an array. Then the array is checked for a digit that is only possible in one box and if one is found it is determined which box it is. The box is checked to see if the digit is still listed as a possibility in any other cell of that box. If so they are removed and the rule returns true, otherwise it continues trying to find another applicable digit. Finally if the rule can not be applied to any row or column the rule returns false making the solver try a more complicated rule.

There are 11 more rules, of varying difficulties, like box/line reduction implemented for this solver and they can be found by name in appendix B and all of them are explained in detail in Andrew Stuart's book *The Logic of Sudoku*[3].

4.2   Simulated annealing

For the simulated annealing solver a data structure is needed to keep track of what digits were given as clues, since they cannot be swapped around without destroying the initial problem. In that case it would cause the final solution to not correspond to the problem given as input, i.e. it would come up with a solution but not to the correct problem instance. The solver also has to hold the current count for the solution so it can easily be compared with the count of a new neighbor solution.

The initial solution for the solver is created by iterating through all empty cells and assigning a random number between 1 and 9 not yet contained in

the surrounding box to that cell. This initial solution is probably not the correct one giving the solution a count > 0.

Upon creating this initial solution the main loop starts. It contains all steps listed in Section 2.2 as well as an upper bound on how many iterations are allowed. To handle reheats a guard is implemented increasing the temperature if the loop iterates 1000 times without altering the solution. The reheats are also limited to 10 since the execution time has to be limited in some way for a probabilistic algorithm.

4.3   Searching for solutions

Like the rule-based solver, searching for solutions requires a possibility matrix to keep track of all possible numbers for each cell. The solver itself is implemented using a search tree that is built dynamically while traversing it. As said in Section 3.3 we choose the cell with the smallest amount of possible digits in each step, to attempt to eliminate the biggest possible branch from the search tree in each step. This is achieved by iterating through all cells storing the one with the lowest possibility count, and then with the stored cell branch the current level of the tree.

The algorithm then tries the lowest possible digit in this cell followed by the second lowest and so on, calling itself recursively in each step. This guarantees that the correct solution will be found eventually.

The algorithm itself is recursive and works in the same way as a depth first search, explained in *Artificial Intelligence: A modern approach*[1], but as previously mentioned it can choose which cell to operate on in each step. When the algorithm finds that all cells has been filled when searching for the cell with the lowest possibility count the solution has been found.

5   Results

The implementations were tested by twice per difficulty-level solving 10 000 sudoku puzzles. The complete results table, with mean values from the two runs, can be found in appendix A.

Figure 3 shows an example run of the simulated annealing algorithm, with two reheats. First, at about 750 iterations, the algorithm stays for a long time at the same count value. When our iteration limit is exceeded a reheat occurs. Another local minimum is found at approximately 2100 iterations, and after the following reheat the algorithm finally gets the count variable down to 0, which means that the solution is found.
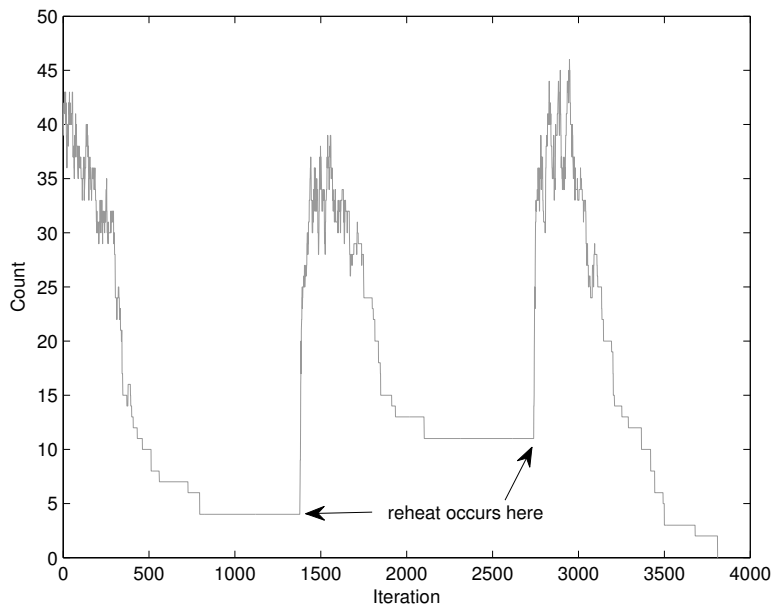
Figure 3: Example run of the simulated annealing algorithm

5.1 Success rate

Success rate is how many puzzles, out of 10 000, that a certain algorithm solves for a certain difficulty-level. Searching for solutions solved every single puzzle it received. With many given clues simulated annealing and the rule-based solver solved every puzzle, but as the number of clues decreased - so did the success rate. Figure 4 displays the results in a bar chart.
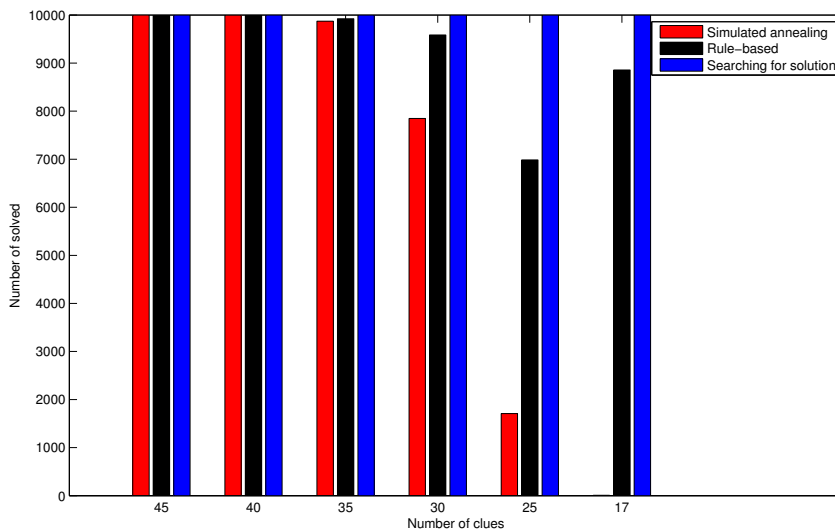

Figure 4: Number of puzzles correctly solved by each implementation for each difficulty

11

5.2 Execution time

Time was measured for each implementation, on each difficulty-level, during testing. The rule-based solver was the fastest implementation and never took more than 3.5 seconds to try to solve 10 000 puzzles. Searching for solutions stayed below 12 seconds until the hardest test, with only 17 clues given to start with, where the total time drastically increased to over 74 minutes.

Figure 5 displays the time graph for the rule-based solver and searching for solutions. Notice that the range of the y-axis is limited to 20 seconds, otherwise the extreme value of searching for solutions with 17 clues would make the graph unreadable.
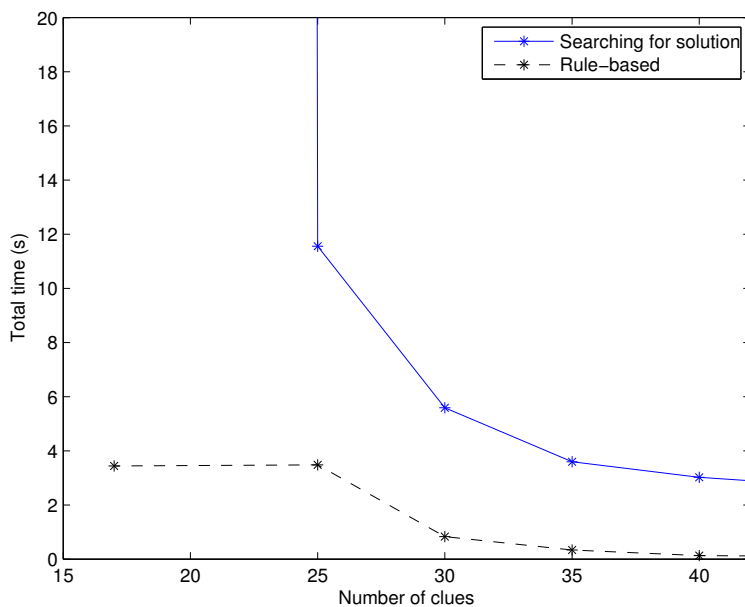
Figure 5: The total time elapsed when trying to solve 10 000 puzzles of each difficulty with searching for solutions and rule-based solver

Simulated annealing had a smoother decrease of run time over number of clues given, ranging between 20 seconds and 20 minutes, shown in figure 6.
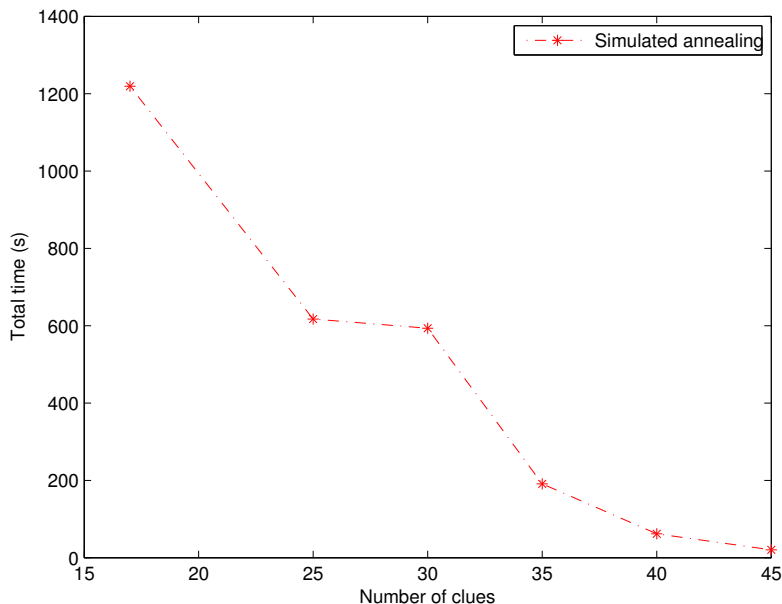
Figure 6: The total time elapsed when trying to solve 10 000 puzzles of each difficulty with simulated annealing

5.3 Memory

The solver using the most memory of our implementations is the one searching for solutions, which copies the sudoku with all of its data for each level of the search tree. In the case with only 17 clues this leads to 64, $81 - 17$, copies when the solution is finally found. Each copy requires 426 bytes of memory[2], which - times 64 - adds up to a total of 27 kB.

The simulated annealing implementation only stores two sudoku puzzles at the same time, the current solution and a neighbor solution. This solver only requires a 9×9 integer matrix and a boolean matrix with the same size which makes a total of 335 bytes per sudoku. Together, this comes up to a total less than 1 kB.

The rule-based solver only has one copy of the sudoku, the same way as the searching for solutions solver, 426 bytes. Even though it uses large temporary variables inside some of the rules, it is still safe to say that it ends up well below 1 kB.

---

[2]As seen in Section 4 a sudoku is stored in 81 integers and 810 booleans, making it a total of 426 bytes

13

5.4 Combined solver

Based solely on execution time and success rate we combined the rule-based solver with searching for solutions. When the rules no longer apply, the algorithm simply continues with a search to complete the puzzle. This new combined solver completed every sudoku puzzle on every difficulty-level. The time results, ranging between approximately 0.1 and 10.1 seconds, is shown in figure 7.
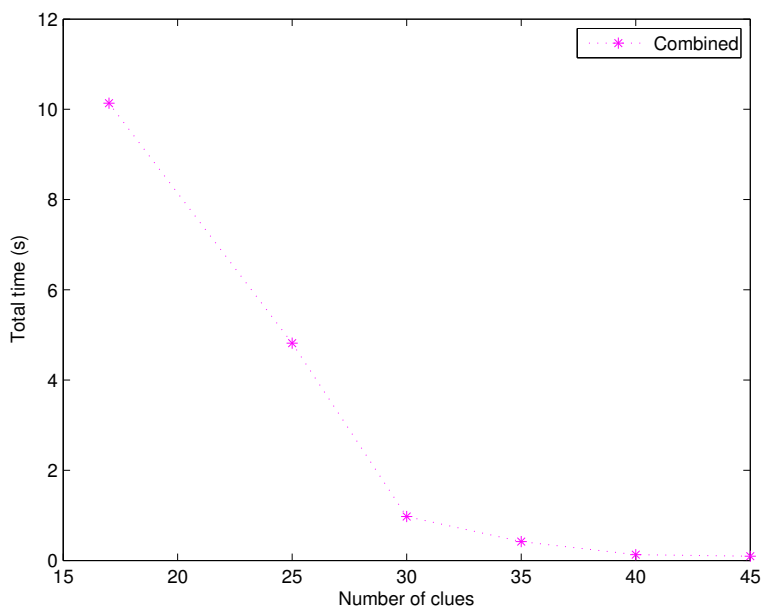

Figure 7: The total time elapsed when trying to solve 10 000 puzzles of each difficulty with the combined solver

The combined solver's time results can be compared to the results of its implemented parts in figure 8. It is considerably faster than the search solver, while it is at most three times slower than the rule-based. In the case of it being three times slower, it solves 13 percent[3] more puzzles.

---

[3]The combined solver solved all 10 000 puzzles with 17 clues given, while the rule-based solver only solved 8 857.
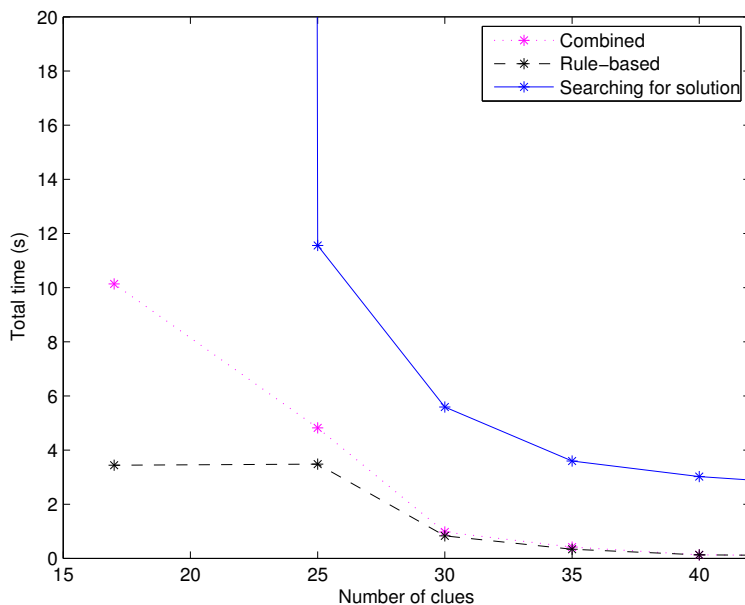
Figure 8: Total time elapsed for the combined solver, the rule-based and searching for solution side by side

6 Discussion

We chose to use Java for all three solvers to be able to make a good and fair comparison on the solvers' execution times. It would not have been fair to compare an algorithm in Java with one in for example C, e.g. since we can not do the same low level optimizations in Java as in C. We also feel the most comfortable in working with Java.

When it comes to measuring the efficiency of the solvers we decided to focus mainly on their execution time and their success rate since all our solvers only use a rather small amount of memory, 27 kB is a small amount of memory with todays standards.

You could also take into account that the simulated annealing solver and the one searching for solutions are both only a few hundred lines of code. Most of the code for these solvers' main parts can be found in various books and on the internet and they do not require a lot of time to implement. The rule-based solver on the other hand requires a lot of code and you can only find fuzzy explanations of how the rules work and you have to come up with the implementations all by yourself. The same main class is used for all solvers, which handles all input and output as well as measuring the solvers' execution times. All of this to make a fair comparison of the solvers.

We noticed that the rule-based solver is more efficient on 17 clues given

than on 25 but we are not sure about why that is. One possible explanation is that the rules we implemented works well for a sudoku with a lot of blank cells, or that due to that 17 is the lowest number of clues currently known for a sudoku there could be limitations on how hard they get.

We are disappointed that the simulated annealing solver worked as poorly as it did. It would have been fun to find a new way to solve sudoku puzzles in an efficient way using a heuristic algorithm. Our first guess was that the simulated annealing algorithm would perform equally well however hard the problem, but this was in fact not the case. The explanation for this is that the fewer the clues in the sudoku the more local minimums it contains, which causes the solver to stall. If we would have had more time to adjust the parameters for the algorithm we could probably have got it to work better than it did, but since it was as bad as it was we chose to focus more on the two that actually worked as anticipated.

The combined solver performed really well, it solved every sudoku and it was almost as quickly as the rule-based solver. The rule-based solver can really fast determine that a problem instance is not solvable, if no rules match, and therefore it is reasonable that it takes longer to solve harder problem instances for the combined solver. After all it only takes about 10 seconds to solve all 10 000 of the hardest sudokus, which we think is a reasonable amount of time.

7 Conclusion

We succeeded in finding different ways to solve sudokus and they all had different strengths and weaknesses. The rule-based solver and the one searching for a solution could easily be combined into an efficient solver. The rule-based solver makes the combined solver fast on many instances of the problem and the searching for solutions gives it the security to always find a solution. We ended up with a solver more efficient than anyone we came across during the project.

The simulated annealing solver was not as good as we initially hoped but there is not much to read about it so there was no way to know it from the beginning.

During the course of the project we learned a lot, mainly how to plan and execute tests for the different solvers in order to be able to present the test data in a good way; using graphs of different types to visualize the results. We also learned how to think when constructing algorithms we did not know from before. Simulated annealing for example we did not know anything about when the project started but we were still able to apply it to sudoku.

8 References

[1] S. Russel and P. Norvig, *Artificial Intelligence: A Modern Approach.* Prentice Hall, 3 ed., 2010.

[2] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi, *Complexity and Approximation - Combinatorial Optimization Problems and Their Approximability Properties.* Springer-Verlag Berlin Heidelberg, 1999.

[3] A. Stuart, *The Logic of Sudoku.* Michael Mepham Publishing, 1 ed., 2007.

Appendices

A   Results table

	45		40		35	
	nr. solved (of 10 000)	total time (in seconds)	nr. solved (of 10 000)	total time (in seconds)	nr. solved (of 10 000)	total time (in seconds)
Rule-based solver	9 997	0.103	9 995	0.127	9 922	0.336
Simulated annealing	10 000	20.265	9 997	61.7905	9 872	190.783
Searching for solutions	10 000	2.699	10 000	3.023	10 000	3.593
Combination solver	10 000	0.0955	10 000	0.1285	10 000	0.41918

	30		25		17	
	nr. Solved (of 10 000)	total time (in seconds)	nr. Solved (of 10 000)	total time (in seconds)	nr. Solved (of 10 000)	total time (in seconds)
Rule-based solver	9 585	0.833	6 986	3.4805	8 857	3.4365
Simulated annealing	7 848	593.1735	1 709	1121.1655	7	1219.2085
Searching for solutions	10 000	5.5935	10 000	11.5585	10 000	4448.718
Combination solver	10 000	0.9795	10 000	4.8185	10 000	10.1375

B Rules implemented in rule-based solver

The following rules are implemented in our rule-based sudoku solver. How each rule works can be read about in Andrew Stuart's *The Logic of Sudoku*[3].

1. Check for solved squares

2. Hidden singles

3. Naked pairs

4. Naked triples

5. Hidden pairs

6. Hidden triples

7. Naked quads

8. Pointing pairs

9. Box/line reduction

10. X-Wing

11. Simple colouring

12. Y-Wing