IMPLEMENTING A CSP SOLVER FOR SUDOKU

Benjamin Bittner[1] **and Kris Oosting**[2]

[1] *University of Amsterdam, The Netherlands; bittner@science.uva.nl.*
[2] *Vrije Universiteit Amsterdam, The Netherlands; kris.oosting@lfri.nl.*

ABSTRACT

The most natural formal description of a Sudoku puzzle is to express it as a constraint satisfaction problem. The board presents the user with a number of cells, some already preassigned, and asks to fill in symbols from a finite domain such that each puzzle segment (row, column, or block) does not contain twice the same symbol.

The naïve approach to solve a Sudoku is to try out potential candidate entries for the cells; if a conflict is detected, we backtrack and try out a different value. For Sudoku this is a bad choice: there exists a number of constraint propagation techniques which can solve most Sudokus without performing any search.

In this paper we report on the implementation of a CSP solver for Sudokus, that integrates standard backtracking search with specialized constraint propagation techniques for Sudoku.

Key words: Sudoku, CSP, constraint propagation.

1. INTRODUCTION

Sudoku is a puzzle invented by the American Howard Garns in 1979 at IBM, and popularized in Japan during the 80s, before rising to world fame in the early 2000s. The game consists of a grid of 9x9 cells, divided in 9 boxes of size 3x3, which have to be filled with numbers from 1 to 9. The constraints by which the numbers can be placed (hence the English name for Sudoku, "Number Place") are as follows:

- Each column, row, and box must contain the numbers 1 to 9.
- No column, row, or box may contain the same number multiple times.

Usually a Sudoku puzzle has a unique solution, which is achieved by pre-filling some of the cells (see figure 1 for a possible starting layout); with these givens and the constraints, the other cells have to be filled. What is also generally assumed is that a Sudoku puzzle can be solved by "mere" reasoning, i.e. without trial-and-error or search.

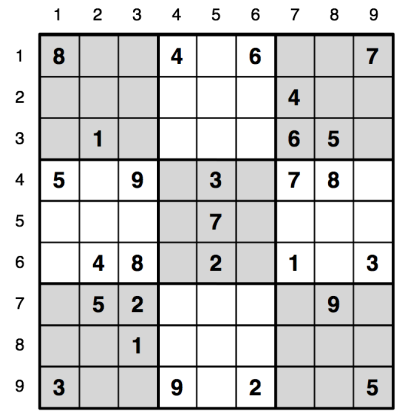

Figure 1. A typical 9x9 Sudoku grid, here with 27 givens. The empty cells have to be filled according to the constraints of the game.

As also shown in [6], Sudoku can be represented as a constraint satisfcation problem (CSP). The author of the paper also notes that people usually employ complex propagation schemes that consider consistency both at a local and a global level, and then goes on to describe generic CSP techniques to solve Sudoku puzzles.

In this work, instead, we want to provide only a basic CSP framework using backtracking, and focusing on the effect of generic and specialized constraint propagation techniques for Sudoku. In section 2 we give a formal description of constraint satisfcation problems, and show how Sudoku can be expressed as one. Section 3 gives a basic backtracking framework for solving (any) CSP, and also our puzzles. In section 4 generic and specialized constraint propagation techniques that speed up the search process are described, and evaluated in terms of efficiency and solving power in section 6. Finally, section 7 gives a summarizing overview on what has been done and achieved, hinting at possible future follow-up works.

2. FORMAL DESCRIPTION

In [3], a constraint satisfaction problem is described as follows:

> We are given a set of variables, a domain of possible values for each variable, and a conjunction of constraints. Each constraint is a relation defined over a subset of the variables, limiting the combination of values that the variables in this subset can take. The goal is to find a *consistent assignment* of values to the variables so that all the constraints are satisfied simultaneously.

More formally, we define a CSP as $\mathcal{CSP} = \langle X, D, C \rangle$, with a finite number of variables $x_1, x_2, \ldots, x_n$, their respective domain $D_1, D_2, \ldots, D_n$, and a set of constraints $C \subseteq D_1 \times D_2 \times \ldots \times D_n$, restricting the possible assignments to the variables (see [1]).

How does this map to the Sudoku problem? The variables in the Sudoku case are the cells of the grid; we adopt the common naming convention x_{rc} for referring to the cell with row r and column c. The domain of each variable, in the here assumed case of 9×9 grids, is $\langle 1 \ldots 9 \rangle$. And C is for Sudoku a set of all-different constraints, that is, each $c \in C$ consists of a subset $X' \subset X$ and the fact that $\forall x_1, x_2 \in X' \,.\, x_1 \neq x_2$. The subsets of variables are all rows, all columns, and all adjacent 3×3 blocks (see figure 2).

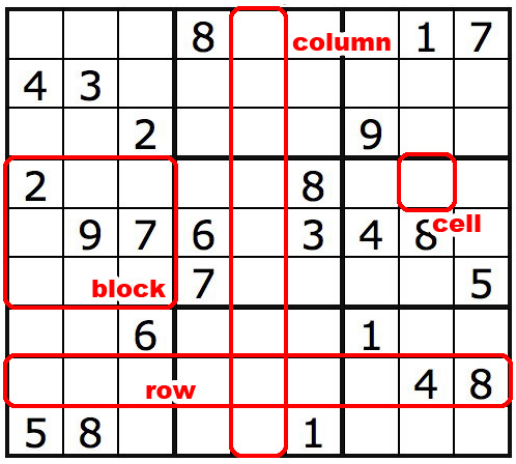

Figure 2. The constraints in Sudoku apply to all rows, columns, and blocks, in which each cell must have a unique value.

3. BACKTRACKING SEARCH

A naïve solution method for constraint satisfaction problems (and also for boolean satisfiability) is

```
function BT (X, V, C)
1    if ∃x, y ∈ V . (x = y) ∈ V ∧ (x ≠ y) ∈ C do
2       return ∅
3    if X = ∅ do
4       return V
5    var := v ∈ X
6    for val ∈ Δ_var do
7       V' := BT(X \ var, V ∪ (var = val), C)
8       if V' ≠ ∅ do
9          return V'
10   return ∅
```

Figure 3. Basic backtracking algorithm. X is the set of unassigned variables, V is the set of assignments, and C the set of constraints on the assignments.

search with backtracking. One can obtain a sound and complete solving algorithm by picking unassigned variables and assigning them a value from their domain, and repeats the procedure recursively, moving through the search tree. If at one point in the search tree a conflict is found (a constraint is violated), the algorithm backtracks unwinds the call stack to the point where some values are still untested for a given variable, undoing any changes to the CSP variable assignments made after that decision point.

Referring to the algorithm in fig 3, an ideal backtracking algorithm is one that would pick variables (line 5) and values (line 6) that will not cause a conflict. It would pick for each variable an assignment that is part of a solution (never reaching line 2 or line 10), with the assumption that there always exists a solution.

So far no such oracle function has been found that can predict which variable should be selected and which value it should be assigned. But one can, as it is done in practice of CSP or SAT search, make an educated guess on what to pick. And furthermore, modern CSP (and SAT) solvers also include methods for fast constraint propagation (computing assignments without search) such that a solution or a conflict can be quickly identified.

One common approach to speeding up the backtracking search is trying to discover conflicts as soon as possible. As described in [1], one approach to go about it is to select, at each search iteration (see line 5), the variable with the smallest domain - this is what is done in our basic framework. Other common heuristics are choosing the most constrained variable, or selecting the variable with the smallest difference between its domain bounds; both, however, are not useful in the case of Sudoku: all variables participate in the same number of constraints, and their domains are symbolic and not numeric (with a numeric representation just for human convenience).

Another point of improvement would be to compute

also a variable ordering for the first-fail principle, for instance by considering which value appears most in the still open domains. But this was out of the scope of the present work, since we wanted to focus on specialized constraint propagation. To minimize the effect of variable ordering, we propose to randomize the values in the domain of the selected variable, and try them out in that order.

The next section will cover in greater detail our proposed methods for constraint propagation for Sudoku puzzles.

4. SPECIALIZED CONSTRAINT PROPAGATION

As opposed to searching, constraint propagation prunes the domains of the variables and assigns values to them by mere reasoning, that is, by using various procedures to enforce local consistency within constraints and global consistency among different constraints. Sudoku puzzles are usually solvable by mere constraint propagation, but one needs many and also complex techniques to do so. Also, especially with the more advanced ones of those techniques, the difference between constraint propagation and search becomes less clear. We argue that a smart backtracking search with a small number of efficient constraint propagation methods can very well achieve satisfying solving times.

We now describe all techniques used in our implementation and experiments. It should be noted that we place the burden of constraint propagation on the algorithm, and never make our constraint network explicit. All propagation techniques have an implicit knowledge of these constraints[1].

Furthermore note the cascading effect some of these techniques have: the more complex techniques usually lead to simplifications that directly allow the application of the simpler rules.

Naked Single The naked single rule states that, if a domain contains only one candidate, then that is the value of the corresponding variable; furthermore, this value can be removed from the domains of all cells in the adjacent constraint groups. This latter step is described in [6] (see also [7]) as shaving, operating "on a list of variables, and which removes any value in the domain of each variable which directly leads to a failure". This is also how the initial pruning of the board works: we eliminate the values of the givens from the variable domains in all relevant constraints.

Full House The full house rule is perhaps the easiest rule that can be employed in solving a Sudoku. It says that if only one variable in a constraint (row/column/block) is unassigned, then it will have the value corresponding to its original domain minus all values of the adjacent cells.

Hidden Single Another very popular and basic technique is the hidden single rule. It states that, if within a constrained group of cells only one cell contains a specific value in its domain, then the domain of that cell can be restricted to that value. In [6] this is solved by channeling, in reference to [2], which essentially adds redundant constraint for improved reasoning. Here we solve it algorithmically by checking if this is the case for any value between 1 and 9 in any row, column, or block.

Naked Pair The naked pair method is another useful technique: if there are two cells in one specific constrained group, which have both an identical domain of size two, then these two values can be removed from any other domain in the group.

Hidden Pair The hidden pair rule says that, if there are two values which both appear in the domain of two cells in a specific constrained group, but in no other domain in that group, then the domains of those two cells can be each restricted to those two values.

Hidden Triples These work in the same way as hidden pairs, only with three cells and three candidates. However, not all three candidates have to be part of the domains of all three candidates. For example, the candidate domains of the hidden triples 4, 8, 9 can be {**4**,5,**8**,**9**}, {1,**4**,6,**8**}, and {2,3,**8**,**9**}.

Locked Candidates The locked candidates rule states that, if in a block all candidates of a certain digit are confined to a row or column, that digit cannot appear outside of that block in that row or column, and the respective domains can be simplified. One can note that this method already covers more than one constraint: it combines information from rows/columns and blocks.

X-Wing The x-wing technique is another propagation technique that considers two different constraints. For rows it is expressed as follows: If for any two rows one can find two columns, such that all candidates of a specific number in both rows are contained in the columns, then that number can be eliminated from all column domains not part of the selected rows. The same approach is valid when substituting rows and columns in the description above. For a graphic representation of the technique see figure 4.

5. IMPLEMENTATION

The CSP Solver was implemented on Apple computers running MAC OS X version 10.6.7. For the

[1]For more information see `http://hodoku.sourceforge.net`. The descriptions here follow in part the wording of that web page about Sudoku techniques.

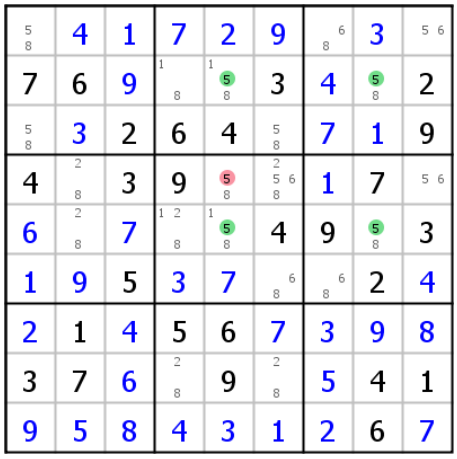

Figure 4. The x-wing propagation techniques. In green the x-wing numbers, in red the numbers to be eliminated.

development of the software LispWorks 6.0.1 Professional was used. The architecture of the program is straightforward. Groups of functions handle specific functionality. These groups of functionality are:

- definitions: definition of constants, structures, and user interface classes.
- general supporting functions: specialized programming functions and strategy statistics functions.
- sudoku support functions: creation of the sudoku puzzle, printing, conversion of givens to internal format, functions that return cells which are part of a specific row, column, or block, update the sudoku puzzle by using the csp variables.
- csp support functions: creation of the csp variables array, printing of csp variable values and domains, determine domain for each csp variable (pencil marking), and constraint related functions like all-different.
- strategy functions: naked single, naked pair, hidden single, hidden pair, hidden triples, full house, locked candidates, and x-wing strategies. These are the constraint propagation functions.
- search functions: propagation of constraints and backtracking search, and the function that integrates backtracking search and constraint propagation.
- the main function: to run the program and to read givens from file or text field for processing one sudoku given.

The main data structure of the program is the CSP variable. Each cell of the Sudoku puzzle was represented by this CSP variable. The storage of the

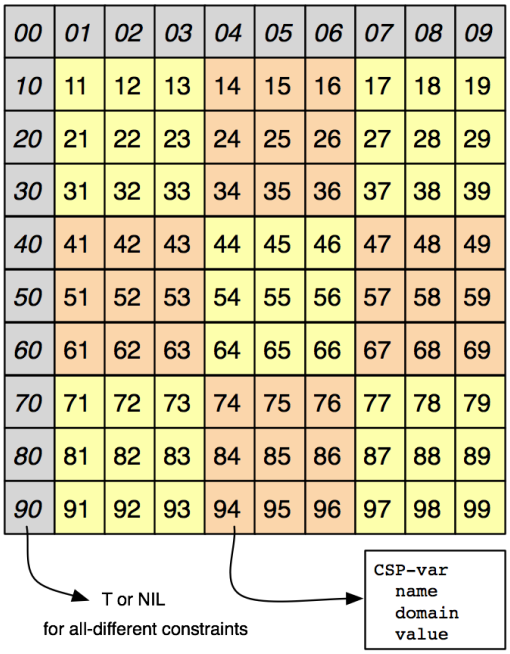

Figure 5. The CSP variable array with CSP variable structures.

CSP variables was realized by using a one dimensional array of 0..99. As explained in [5] a simple one dimensional array could be efficient to represent a board-like game. In our CSP solver we used a similar setup, as depicted in figure 5. This figure shows that the cells with index 1 till 9 represent the columns and the cells with index 10, 20, till 90 represent the rows. These cells contain boolean values *true* or *false* to represent the *all-different* constraint of a row or column. The cell with index 00 was used to indicate if the complete Sudoku was *all-different*, set to *true*, or not yet *all-different*, set to *false*. The cell with indices 11..99 contain the CSP variable structures. These structures consist of the name, the domain and the value of the CSP variable. In this case the name is kept the same as the cell index. This cell index is a two digit number where the first digit represents the row number and the second digit represents the column number of the sudoku grid as shown in figure 1.

The flow of the CSP Solver program is shown in figure 8. The program can process two types of input. The first type is a file with Sudoku givens per line, and the second type is one line of Sudoku givens entered in a text input field. Figure 9 show the graphical user interface (GUI) used during the experiment. The GUI of the program was designed to offer the possibility to enter one line of sudoku givens or to read a file with one or more lines of sodoku givens. While processing of the Sudoku givens, this GUI showed the progress by displaying the strategy statistics and the end result of the CSP Solver; the Sudoku puzzle could be solved or not. After read-

ing the givens, they were converted to an internal format. This internal format was used to create the Sudoku problem and to initialize the CSP variables for this Sudoku problem. The next step was to determine the domains for all the CSP variables; in Sudoku terminology often referred as *pencil marking*. These previous steps were necessary to setup the environment for the CSP Solver. The Solve Sudoku CSP detail in figure 8 shows that the solver performed constrained propagation by using one or more of the Sudoku strategies as described in section 4. Because not all Sudoku strategies were implemented, the program is equipped with backtracking search that would solve the remaining unsolved variables when none the available Sudoku strategies would lead to a solved Sudoku. The final step was displaying the results.

The implementation of the main solution procedure is described in figure 6, and corresponds to the block "Solve Sudoku CSP" in figure 8. The constraint propagation loop is explained in figure 7. Each propagation technique returns `True` if it made some progress, and `False` otherwise. The evaluation of boolean expressions by Lisp guarantees that the loop is quit when either a conflict is found, or no progress could be made by any propagation technique; furthermore, the loop tries technique after technique, and restarts as soon as one technique makes some progress.

```
function solve (csp-vars)
1    propagateConstraints (csp-vars)
2    if existsConflict (csp-vars) do
3      return nil
4    if all are assigned (csp-vars) do
5      return csp-vars
6    var := pickVarByMRV (csp-vars)
7    for val in Δ_var do
8      tmp-vars := copy (csp-vars)
9      tmp-vars[var] := val
10     tmp-vars := solve (tmp-vars)
11     if tmp-vars ≠ nil do
12       return tmp-vars
13   return nil
```

Figure 6. The main solving procedure.

```
loop while
      not hasConflict (csp-vars)
      and
      (technique1 (csp-vars) or
       technique2 (csp-vars) or
       ... or
       techniqueN (csp-vars))
```

Figure 7. The constraint propagation loop.

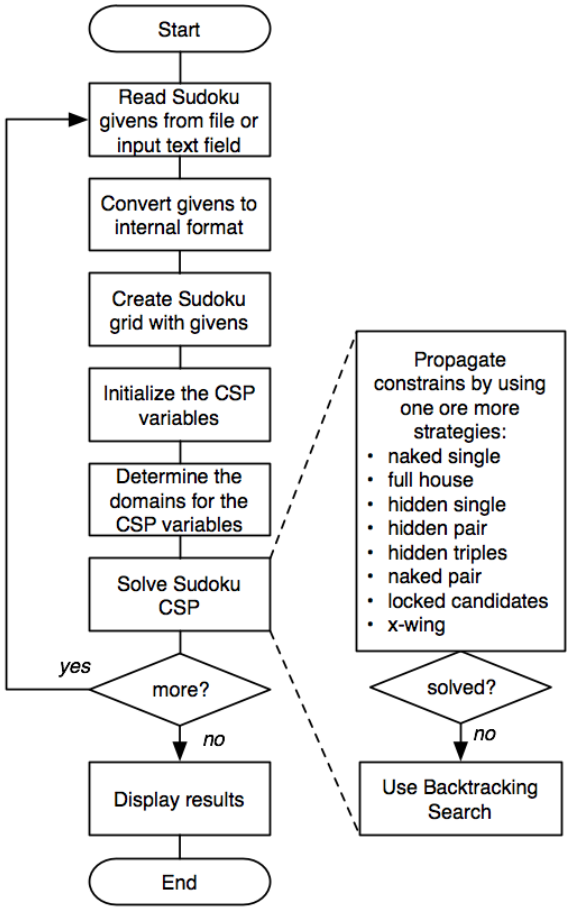

Figure 8. The CSP Solver program flow. Backtracking search calls the Solve-block recursively.

6. EXPERIMENTS

Testing of the program was performed with the help of sudoku games with teaching ability, like Sudoku 401 on the iPhone. Step by step the strategies were tested according the the suggested technique by Sudoku 401. The results of each step were checked with printing functions. For testing the program LispWorks did offer good support. In a running program, functions could be changed, and tested in the next step without exiting the program. Also the strategy functions could be tested in isolation. However, some strategies that were suggested by Sudoku 401, like finned x-wing, were not implemented in the CSP Solver program. This was taken care of by the backtracking search function. When the program could not solve a Sudoku, the backtrack search program solved the remaining open cells.

For simple Sudokus, different strategies led to the same solution. For example, by recurrent use of the naked single strategy only or by the combined use of the naked single, hidden single, full house, hidden pair, and naked pair strategies. Although, the

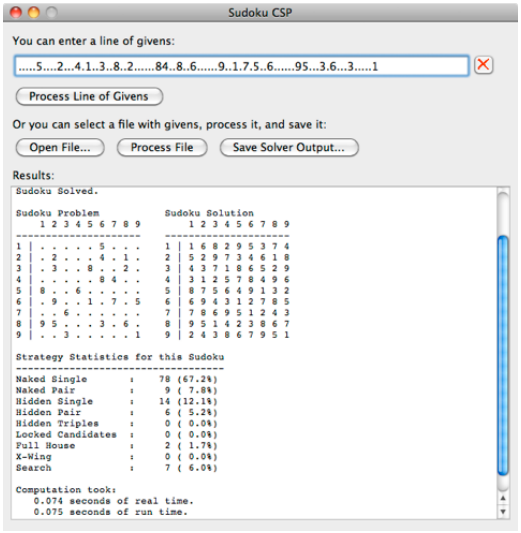

Figure 9. The Graphical User Interface of the CSP Solver program.

naked single strategy could solve a Sudoku, it had to be applied many times. The application of a mix of strategies could reduce the domains of the constraints more rapidly, thereby the number of steps needed to solve a Sudoku were reduced.

	VH Top 10		VU 1011	
Technique	μ	%	μ	%
Naked Single	246.5	57	53.15	67
Naked Pair	30.2	7	1.95	2
Hidden Single	79.2	18	13.68	17
Hidden Pair	12.1	3	1.33	2
Hidden Triples	0.0	0	0.00	0
Locked Candidates	19.3	5	3.08	4
Full House	4.0	1	2.01	3
X-Wing	0.0	0	0.00	0
Search	40.3	9	3.74	5

Table 1. Statistics on algorithms used for solving all given puzzles. The mean refers to the mean number of times a technique was used per puzzle. Average solution time for a puzzle was 0.075*s.*

The CSP Solver program was also tested on the Vengard Hansen[2] top ten super hard Sudokus. These ten Sudokus were all solved by the program by the application of the naked single, hidden single, naked pair, hidden pair, locked candidate, full house, and search strategies. The mean values of the frequency of the applied strategies in solving 1011 Sudokus[3], led to the following numbers as produced by the program. From table 1 the following conclusions can be drawn.

1. Integrating backtracking search with constraint propagation is powerful.
2. The less complex propagation techniques are used more often.
3. Also with difficult Sudokus, search is only used ∼10% of the time.
4. This is coherent with insights from SAT solver benchmarks, where 80-85% of the time is used for constraint propagation, and only the rest for search (see [4]).

7. CONCLUSION

In this work we investigated the use of CSP solving techniques for Sudoku. Human solvers and specialized Sudoku solvers usually only employ propagation techniques, and therefore need a big number of more or less complex ones to be able to solve all puzzles. Here we showed that those complex techniques, which are very rarely used, can very well be substituted with a backtracking search procedure, while still solving the puzzles in a fast manner. It's interesting to note that backtracking involves a substantial amount of bookkeeping, and might thus not be suitable for human solvers.

REFERENCES

1. K.R. Apt. *Principles of constraint programming*. Cambridge Univ Pr, 2003.
2. B. Cheng, J. Lee, and J. Wu. Speeding up constraint propagation by redundant modeling. In *Principles and Practice of Constraint ProgrammingCP96*, pages 91–103. Springer, 1996.
3. A.K. Mackworth. Consistency in networks of relations. *Artificial intelligence*, 8(1):99–118, 1977.
4. M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *Design Automation Conference, 2001. Proceedings*, pages 530–535. IEEE, 2001.
5. P. Norvig. Paradigms of ai programming. *Case Studies in Common Lisp. Redwood City, CA: Morgan Kaufinan Publishers, Inc*, 1992.
6. H. Simonis. Sudoku as a constraint problem. In *CP Workshop on modeling and reformulating Constraint Satisfaction Problems*, pages 13–27. Citeseer, 2005.
7. P. Torres and P. Lopez. Overview and possible extensions of shaving techniques for job-shop problems. In *Proceedings of the Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimization Problems, CPAIOR00*, pages 181–186, 2000.

[2]For more information see `http://www.menneske.no/sudoku/eng/top10.html`.

[3]The set sudoku-training.txt provided by Annette Ten Teije and Frank van Harmelen (Automated Reasoning in AI, VU Amsterdam.)