So, you want to generate your own sudoku? *

Raphael Finkel Victor Marek Mirek Truszczyński

Abstract

This paper is attempting to introduce the reader to the use of declarative tools such as *aspps* or *smodels* to generate tabular numerical puzzles such as sudoku.

1 Introduction

Cellular puzzles take the form of **cells** whose **values** are constrained by **rules** involving groups of cells. The puzzle starts with **clues**, which are cells that already possess their value. It is the pleasant task of the puzzle **enthusiast** to solve the puzzle by entering values in all the cells in a way that satisfies the constraints. A **well-formed** puzzle has exactly one solution. To assist the enthusiast, the puzzle may contain a **hint sequence**, which is a sequence of cells that suggest the order in which the enthusiast should be able to complete the puzzle. While the puzzles published in newspaper do not provide assistance to the person solving the puzzle beyond providing (either at the same time, or later) a solution, it is possible to assist the person solving the puzzle in a subtler way, and we will discuss as we will discuss it in the body of this paper.

*Computer Science Department, University of Kentucky. email: raphael | marek | mirek@cs.uky.edy

This paper is a (curious) mixture of science and a cookbook. On one hand, we will provide a rigorous description of the puzzles of sudoku class. To make it even worse, we will present the unsuspecting reader with the introduction to *constraint satisfaction problems* and then cast the problem of constructiong sudoku puzzles as a two-phase application of constraint satisfaction; first, to find a solution, and second, to use the solution that was found and a certain algorithm for building a sudoku problem - out of that solution. Of course, nothing like this is done by hand. The computer and its software (we will use a class of publicly (and freely) available software called *solvers*) produce the problem. The output of the solver requires an additional processing; we will comment on this issue as well.

Here is the outline of our paper. First we will (gently) introduce the reader to the concept of cellular puzzles that we will discuss in the paper. Then, a heavy artillery starts, we will introduce the *constraint satisfaction problems* and we will cast the tabular problems we focus on in this paper as constraint satisfaction problems. In this process we will introduce *basic constraints* that define so-called Euler's *Latin Squares* [DK74] that are not yet a sudoku, but almost are. Then we introduce the additional constraint that selects out of latin squares *sudoku solutions*, that is latin squares that satisfy the additional *sudoku constraint* (namely that every one of nine *quadrants* (also called *sections*) contains every number in the range 1..9 exactly once.) To scare the reader more we formulate (but not prove) the proposition asserting correctness of our construction. At that stage of our proceedings we know that the sudoku solutions can be obtained by running software capable of solving constraint satisfaction problems. But our goal

is *not* to find sudoku solutions, but sudoku *problems*. We then formally define what a sudoku problem is and state (but not prove) formally, the criterion for a constraint satisfaction problem to be a sudoku problem. Two (entirely immaterial for a layperson) propositions give two different representations and assert the correctness of that representation. But we will *not* create sudoku problems on commonly available constraint satisfaction problem software such as ECLiPSe [WNS97], but use the representation of (finite-domain) constraint satisfaction problems using something quite exotic for general public (but known in some Computer Science and also Combinatorial Optimization circles) called *cardinality constraints*. Instead we will first formulate sudoku *solutions* as solutions to some system of cardinality constraints, and then we give an algorithm that allows for creating reasonably hard sudoku *problems* via repeated application of solving a system of cardinality constraints. The cardinality constraints solvers are freely available (one such solver is due to the third author and his students and is called *aspps* `http://www.cs.uky.edu/ai/aspps/`. This is the software we use for building sudoku solutions and problems. We save the reader the scare of revealing of what *aspps* is an acronim of. At that stage of our paper the reader can (with some small amount of programming, say in Perl [WS90] or VB) produce a description of a nice sudoku puzzle. One script (it is not provided, we need to keep something in our sleeve) transforms the output of the solver creating the sudoku problem into a familiar, partly filled grid that can be enjoyed by a person solving the puzzles. But now the question occurs if all problems we create will be equally difficult. It turns out that the solvers (the software we use) have at least three different modes for producing their

solutions. Those modes can be used for differentiating among the difficulty of sudoku problems. But this is not the end of our paper. We discuss several classes of tabular problems related to sudoku. For instance, we can require that one or both diagonals in our puzzle has all entries different. We may require that the centers of quadrants house different numbers. We may even require both! We may want to change the shape of pieces in which we partition the basic grid - in the *classical sudoku* we partition the grid into 3-by-3 quares, but we do not have to. This leads to the generalization of sudoku to situations where the grid is no longer 9-by-9. For instance we may have 8-by-8 grid, partition it into additional 8 subgrids (each containing 8 cells) and require that besides of being a latin square, every subgrid also contains each number (in the range 1..8) exactly once. There is a wealth of generalizations and the reader who musters patience to read and understand this paper will have an opportunity to produce her very own special version of sudoku, or sudoku generalization. Finally, we comment on weakening the condition that every row and column contains ever number from prespecified range once.

2 Classical sudoku puzzles

We will consider Sudoku puzzles [TFE05] as a typical cellular puzzle. Figure 1 shows a sample Sudoku puzzle. The cells are arranged in a 9×9 grid subdivided into nine 3×3 **sections** or *quadrants*. We say a set of 9 cells is **complete** if the numbers $1 \ldots 9$ appear exactly once in that set. The constraints that link the cells are that each row, column, and section is complete.

Puzzle

	A	B	C	D	E	F	G	H	I
a				8				2	3
b									4
c	6				7	5	9		
d						2			
e				9			8	3	
f		4				8		1	6
g		3	8		5				
h				7		1			
i	1	5		2				6	

Solution

	A	B	C	D	E	F	G	H	I
a	5	7	4	8	1	9	6	2	3
b	8	9	1	3	2	6	5	7	4
c	6	2	3	4	7	5	9	8	1
d	3	8	5	1	6	2	7	4	9
e	2	1	6	9	4	7	8	3	5
f	9	4	7	5	3	8	2	1	6
g	7	3	8	6	5	4	1	9	2
h	4	6	2	7	9	1	3	5	8
i	1	5	9	2	8	3	4	6	7

Figure 1: A 9×9 Sudoku puzzle, and its solution

3 Constraint satisfaction problems

We start with a formal definition. Later on we will move from constraint satisfaction to cardinality constraints and some readers may want to move to that place (Section 6). A *constraint satisfaction problem* is a tuple $\mathcal{P} = \langle X, \{D_x\}_{x \in X}, R_1, \ldots, R_k, S \rangle$ where

1. $X = \{x_1, \ldots, , x_m\}$ is a finite set of *variables*. The set X is called as *scheme* of $\mathcal{P}$

2. S assigns to each realtion R_j, $1 \leq j \leq k$ its *scheme*, $S_j \subseteq S$.

3. For each variable $x \in X$, there is a finite *domain* of variable x, D_x

4. For each j, $1 \leq j \leq k$, with $S_j = \{i_1, \ldots, i_j\}$, the relation $R_j \subseteq D_{i_1} \times \ldots \times D_{i_j}$

The *solution* to the constraint satisfaction problem $\mathcal{P}$ is any sequence $\langle p_1, \ldots, p_m \rangle$ (remember that X has m elements) such that

(a) For all i, $1 \leq i \leq m$, $p_i \in D_i$

(b) For all j, $1 \leq j \leq m$, $\langle p_{i_i}, \ldots, p_{i_j} \rangle \in R_j$

Let us see how the complete sudoku solutions (i.e. entirely and correctly filled grids) can be cast in the language of constraint satisfaction. What we introduce are 81 variables, one for each cell of the grid. It will be convenient to use double indices for variables: the variable $x_{r,c}$ (where the row index r ranges over letters 1..9 and the column index c ranges over 1..9 [1] denotes the content of the cell (r, c). See the diagram of our sudoku for the self-explanation of our convention. Domains of all variables are the same; they are 1..9. So now we know the scheme of the problem and the domains of variables. Now we need to define relations. There will be 27 relations. Nine relations for the rows of our puzzle, nine relations for columns of our puzzle, and nine relations for quadrants (sections). The tables will be very similar, they will differ by schemes, but not anything else. Let us see the scheme for the constraint for the first row: it consists of variables: $x_{a,A}, x_{a,B}, \ldots, x_{a,I}$. Likewise, the constraint corresponding to the last row has the scheme consisting of variables: $x_{i,A}, x_{i,B}, \ldots, x_{i,I}$. We hope the reader have seen the scheme for the other rows. The scheme for the first column will be: $x_{a,A}, x_{b,A}, \ldots, x_{i,A}$. We hope the the reader see the schemes for the constraints of other columns. The schemes for quadrant constraint have more complicating form. Here is the one for the middle quadrant: $\{x_{d,D}, x_{d,E}, x_{d,F}, x_{e,D}, x_{e,E}, x_{e,F}, x_{f,D}, x_{f,E}, x_{f,F}\}$. We leave the problem of founding the other schemes for quadrants to the reader. Surprisingly, besides of having different

[1] To distinguish between row indices and column indices we enumerated rows with latter a..i and columns with letters A..I, see Figure 1

schemes, the actual tables for the constraints are all the same. It is a table with nine columns (remember they will be relabeled as we move through the constraints). Each table has exactly 9! rows (roughly 38 thousand lines - but do not worry, we will never write it explicitely!). The researchers in constraint satisfaction have a language for this very constraint. They call it **all_different** (and they add the names of variables which are required to be all different). So, our first constraint is

$$\textbf{all\_different}(x_{a,A}, x_{a,B}, x_{a,C}, x_{a,D}, x_{a,E}, x_{a,F}, x_{a,G}, x_{a,H}, x_{a,I})$$

Now a bit of combinatorics. A *latin square* over $(1..n)$ is an $n \times n$ table filled with numbers $(1..n)$ such that every row and every column contains each number from $(1..n)$ *exactly once*. Thus sudoku solution (like in our example) is a latin square, but not every latin square is a sudoku solution. To be totally formal we define a *complete sudoku solution* (*sudoku solution* for short) a latin square over m^2 for some m where every quadrant (there will be m^2 quadrants) has also every number from 1 to m^2 exactly once. The classical sudoku solution is one for $m = 3$ (and so $n = 9$). But there are sudokus in $n = 16, 25, 36$ etc.

Now, to be on the safe side we formulate a proposition that tells us that, in principle, constraint satisfaction problem solvers generate sudoku solutions.

Proposition 1 *There is a bijective correspondence between solution to the constraint satisfaction problem $\mathcal{P}$ and complete sudoku solutions.*

4 Sudoku *problems* as constraint satisfaction problems

So far we have seen that we can cast complete sudoku *solutions* as constraint satisfaction problems. But, of course, our goal is to construct sudoku problems, and in fact interesting sudoku problems. So now we will describe sudoku problems (complete or otherwise) as constraint satisfaction problems. We will start with the general definition, specially for theoreticians, especially those who know the concept of a *partial fiunction*. Let $\mathcal{C}$ be some constraint satisfaction problem with its set of variables X. A *partial assignment* for $\mathcal{C}$ is a function f satisfying the following conditions:

1. The domain of the function f, $dom(f)$, is included in X
2. for all $x \in dom(f)$, $f(x) \in D_x$

This looks intimidating, but partial assignment is just assignment of values to *some* variables, not necessarily all. It may be correct or not. It is quite clear when such assignment is intuitively correct – when we can complete f to a solution. So now the following definition makes sense. A *partial solution* to a constraint satisfaction problem $\mathcal{C}$ is a function f which is included in some complete solution in of $\mathcal{C}$. This should immediately ring the bell: we assign values to some cells in the grid. We did a reasonable job if what we did can be completed to a solution. So we are almost at the correct definition. A *partial complete solution* is a partial solution which extends to an *exactly one* complete solution. There are all sort of characterizations of partial solutions and of partial complete solutions. To give a suitable characterization we need to modify constraint satisfaction problem associated with sudoku. Let $\mathcal{P}$ be

$x_{e,D}$
9

Table 1: The relation associated with clue $x_{e,D} = 9$

a constraint satisfaction problem, and f a partial assignment of variables from X. We extend $\mathcal{P}$ to a new constraint satisfaction problem by adding a number of very trivial relations. Namely, we add, for each $x \in dom(f)$ a relation R_x. This relation is very simple. Its scheme consists of just one element, x (so it has just one column). The relation R_x has just one row consisting of $f(x)$. Let us look at our example. The clues (depicted in the left side of our Figure 1) are the partial assignment. For instance we assigned the value to $x_{a,D}$, namely 8. We assigned the value to $x_{i,A}$, namely 1, etc. Altogether, our partial assignment had 25 variables in its domain. therefore we will extend our constraint satisfaction problem for classical sudoku (81 variables, 27 relations) by 25 very simple tables. Here is one such table (scheme in the top row): Given partial assignment f, we define $\mathcal{P}_f$ as the constraint satisfaction problem:

$$\langle X, \{D_x\}_{x \in X}, R_1, \ldots, R_k, \langle R_x \rangle_{x \in dom(f)}, S \rangle$$

We then have

Proposition 2 *1. A partial assignment f is a partial solution for $\mathcal{P}$ if and only if the constraint satisfaction problem $\mathcal{P}_f$ has a solution*

2. A partial assignment f is a partial complete solution for $\mathcal{P}$ if and only if the constraint satisfaction problem $\mathcal{P}_f$ has a unique solution

We can also express our Proposition 2 in a database langage SQL (yes!). We need a bit of terminology. Given a partial valuation f, with the domain of f being $h_1, \ldots, h_p$ let φ_f be the following propositional formula:

$$h_1 = f(h_1) \text{ AND} \ldots \text{AND} h_p = f(h_p).$$

Now we write an SQL query Q_f:

SELECT $*$
FROM $R_1, \ldots, R_k$
WHERE φ_f

We then have

Proposition 3 *1. A partial assignment f is a partial solution to $\mathcal{P}$ if and only if the answer to the query Q_f is nonempty*

2. A partial assignment f is a partial complete solution to $\mathcal{P}$ if and only if the answer to the query Q_f consists of just one tuple.

Even though Propositions 2 and 3 looks mathematical, they actually tells us what do we need to do. We have to find the partial assignments f so that $\mathcal{P}_f$ possesses a unique solution. But Proposition 2 does *not* tell us how to do this, and, in particular, what kind of software means can we use in this venture.

5 Possible strategies for a random complete solution generation

The strategy to construct a meaningful sudoku problem, i.e. finding some reasonable correct set of clues consists of two steps. First we will find a

sudoku solution. Then once we know that solution, we will build out of that solution a set of clues (complete partial assignment). This partial assignment will be what will be presented to the human solver.

In this section we need to see what kind of strategy can be used to find a solution. Actually there are many possible strategies. We list four. First, the programmer (constructor of the puzzle) may design and implement a specialized program in some imperative computer language such as C++ or JAVA. In order to produce different solutions some random number generation may be needed, to direct the program in its search for solutions. The second approach, for enthusiasts of SQL could be to directly translate our constraint satisfaction problem $\mathcal{P}$ to SQL. True, there will be 27 tables, each approximately 38,000 records long, and 81 variables (thus attribute names), but, in principle, a query that produces *all* sudoku solutions can be written. But there is a cosmic number of sudoku solutions even for classical sudoku. There will not be enough of space to write them all to the disk. The third possibility is to translate our constraint satisfaction problem to some program which can be further processed by a constraint programming language such as EcLiPSe or CHIP. This can certainly be done. There is a fourth one (we guess that there may be others) namely converting the constraint satisfaction problem $\mathcal{P}$ to a problem in logic. This is what we will do. This will *drastically* decrease the number of constraints we need to do. Then, by pure miracle, we will find a computer language (known as PS+ [ET01]) where these constraints can be easily written. The effect will be a short (couple of lines) description of classical sudoku. Even more miraculous will be the fact that there is a software (publicly available, not less!) that will process such

programs and return a random solution. Such software is called a *solver*. The type of solver we use is based on two techniques, one taken from logic, another from combinatorial optimization. From logic we will take the way to write *formulas*. From combinatorial optimization we will use a technique generalizing atomic expressions to so-called *pseudo-boolean constraints*. Both are pretty intuitive and we will devote next section to the syntax of logic and and what do we need to write.

6 Logic and its extension with cardinality constraint atoms, computing sudoku solutions

We will assume that the reader had sometimes in the past some kind of logic course, and that she is familiar with logical notation, in particular quantifiers. But we will go a step further; we will extend the language of logic to include *cardinality atoms*. The reason is that tools like *aspps* are aware of cardinality atoms and are able to process some expressions containing cardinality atoms. The idea is that we require that certain number of atoms (at least as much as the lower bound, but not more than upper bound) is true. Here is a simple example. An expression $1\{p, q, r, s\}2$ means that at least one but not more than two among p, q, r, and s are true. So if p and r are true but q and s are false we satisfied our cardinality constraint. But if, say, p, q, r are true, but s false too many of atoms are true, and our constraint fails. Now, for $1 \leq i \leq 9$, $1 \leq j \leq 9$, $1 \leq k \leq 9$ the proposition $p_{i,j,k}$ denote this fact: *the cell with coordinates* (i, j) *holds* k. Then look how our constraint that cell (i, j) contains exactly one value could be written:

$$1\{p_{i,j,1}, p_{i,j,2}, \ldots, p_{i,j,9}\}1.$$

Likewise, the fact that given i and k the i^{th} row contains number k exactly once can be written as:

$$1\{p_{i,1,k}, p_{i,2,k}, \ldots, p_{i,9,k}\}1.$$

And with column j? – also very easy:

$$1\{p_{1,j,k}, p_{2,j,k}, \ldots, p_{9,j,k}\}1.$$

For quadrants, we need to be a bit careful. Here is the cardinality constraint expressing the fact that number 1 occurs in the first quadrant exactly once:

$$1\{p_{1,1,1}, p_{1,2,1}, p_{1,3,1}, p_{2,1,1}, p_{2,2,1}, p_{2,3,1}, p_{3,1,1}, p_{3,2,1}, p_{3,3,1}\}1.$$

The issue now is if we have to do all this by hand or if additional means can be used. Even though it is certainly easy to write a script to produce all these constraints involving propositional variables, there is a better way. Namely we use the predicate calculus. We add a new predicate letter p with 3 places and we write $p(i, j, k)$ to denote that *the cell* (i, j) *contains* k. So now our constraint on contents of cell is:

$$\forall_{x,y}\exists_z 1\{p(x, y, z)\}1$$

that is for all row indices x and column indices y that exactly one of of atoms $p(x, y, 1), p(x, y, 2), \ldots, p(x, y, 9)$ (here the range of z is $(1..9)$) is true. and we certainly can do this for row and column constraints getting:

$$\forall_{x,z}\exists_y 1\{p(x, y, z)\}1$$

and

$$\forall_{y,z}\exists_x 1\{p(x, y, z)\}1$$

```
 1  numsmall(1..n).
 2  numlarge(1..m).

 3  pred place(numlarge, numlarge, numlarge).
 4  var numlarge I,J,N.
 5  var numsmall K,M.

 6  1 { place(I,J,N)[N] } 1.
 7  1 { place(I,J,N)[I] } 1.
 8  1 { place(I,J,N)[J] } 1.
 9  1 { place(I,J,N)[I,J]: I<=n*K: J<=n*M: n*(K-1) < I:
10      n*(M-1) < J} 1.
```

Figure 2: Aspps rules for the Sudoku puzzle

It is a bit more difficult for contraints on quadrants. We need to look at some arithmetic conditions. We will see the constraint on quadrants soon.

We have to be able to tell the processing engine that we want to deal with numbers (1..9) - but it could be (1..16) for more complex sudoku, or even 25, or 36. Next we have to be able to distinguis between the universally quantified and universally quantified variables.

When the dust settles we have the following expression, written in the language understood by *aspps*:

Lines 1 and 2 introduce `numsmall` and `numlarge` as predicates true over the range $1 \ldots 3$ and $1 \ldots m$, respectively. The parameters n and m can be instantiated from command line. In classical sudoku, n is instantiated to 3 and m to 9 Lines 3, 4, and 5 give the signature of the ternary predicate `place` and the variables `I`, `J`, `N`, `K`, and `M`. We use `place(I,J,N)` to represent the fact that the value `N` is placed in cell `(I,J)`. Line 6 can be read "given a cell `(I,J)`, there is at least one and at most one (that is, exactly one) value

`N` that is placed in that cell." Line 7 represents the constraint that there is exactly one row `I` for which any given value `N` appears in column `J`. Line 8 constrains columns in a similar way. Lines 9 and 10 introduce the section constraints. Given a section numbered `(K,M)` and a value `N`, there is exactly one cell `(I,J)` in that section that has that value.

The software such as *aspps* converts the program such as the one in Figure 2 to a *propositional program*, in effect eliminating variables, and putting the constant values (from the range of variables) instead. This process is known as *grounding*. The grounded program has only propositional variables. The casual user does not have to see the grounded version of the program at all. The grounded version of the program is *solved*. This process finds its *propositional models* also known as *satisfying valuations*. There is a huge number of solutions. To avoid repetition, the processing of the propositional program is randomized, for instance by randomly assigning values of some propositional variables. The solver is based on *backtracking search*, a form of search closely related to so-called DPLL algorithm for finding propositional models of CNF propositional theories ([DLL62]). The DPLL algorithm was extended by the designers of *aspps* top handle clauses admitting cardinality constraints.

The following fact is crucial for the purpose of constructing sudoku puzzles.

Proposition 4 *There is a bijective correspondence between the models of program listed in Figure 2 with parameters $n = 3$ and $m = 9$, and classical sudoku solutions.*

We will call the program consisting of lines (1), (3), (4), and (5) only the

basic program. The models of this program initialized with $m = k$ correspond to latin squares of length k. Therefore, when we build later in the paper other puzzles based on latin squares, we will reuse this basic program, heaping on the top of it additional constraints.

The question arises if *aspps* is crucial for the kind of applications we have in mind. In fact we could use (with pretty small changes in the program, but still the changes would be required) solver such as *smodels*[NS97]. or PBS[ARMS03]. We could (with still more work on our part) use SAT solvers such as zchaff [MMZZM01], or other solvers publicly available. Since these solvers usually do not support cardinality constraints, and generally appear to be weak in the area of knowledge representation, an additional work on coding would be needed before we could use the results returned by the solver.

Let us add that the number of sudoku *solutions* is known, see [FJ05]. The authors of [FJ05] give a method of calculation of the number of sudoku solutions, and compute that number. They also quote a recomputation by an independent researcher (with his own program) of their result. For the sake of completeness, we state that result of Felgenhauer and Jarvis:

$$6670903752021072936960.$$

The number of 16×16 sudoku solutions is not known.

7 From the sudoku solution to sudoku *problem*

So far we have seen how declarative tools, in particular *aspps* can be used to compute sudoku solutions. But, of course, the puzzle enthusiast wants the puzzle, moreover an interesting puzzle, not the solution. we will outline an algorithm for building an interesting puzzle. This concept can be formally defined. Before we give a formal definition recall that a sudoku *problem* is a partial assignment f of values to variables so that there is a unique solution extending f. Here is how we define the concept of interesting problem. A sudoku problem f is *interesting* (*complete*) if

1. f is a sudoku problem
2. for every variable x in the domain of f, the problem g arising from f by eliminating the pair $(x, f(x))$ from f is not a sudoku problem.

This definition of interesting problem requires that an interesting problem be minimal; eliminating of any clue would result in multiple solutions extending the revised set of clues. Here is an algorithm that does build interesting sudoku problems.

The algorithm starts with a computation of one solution. We presented above means that make such computation feasible. Let f be such solution. We proceed recursively. The initial assignment of variable *removableClues* is the entire set of clues. Then, in the main loop, once we have the current value of the set *removableClues* and that value is not empty, we check which values can be eliminated from *removableClues* so that the uniqueness of extension is preserved. One such value is selected at random, and eliminated (this value

```
 1  removableClues := solution
 2  preservedClues := ∅
 3  while removableClues ≠ ∅ do
 4    victimClue := randomChoice(removableClues)
 5    removableClues := removableClues - {victimClue}
 6    if puzzleBad(removableClues + preservedClues) then
 7      preservedClues := preservedClues + {victimClue}
 8    end if
 9  end while
10  clues := preservedClues
```

Figure 3: Algorithm to reduce the clue set

is called *victimClue*. This loop is iterated until removal of any clue results in having more than one extension of *removableClues* to a solution. Here is the algorithm.

Each iteration randomly picks a single clue and tries to solve the puzzle without it. If the puzzle is now bad (we'll explain that shortly), the clue is added to the set of fixed clues, which must be preserved. Eventually, all the original clues are either removed or preserved; the preserved clues become the final set of clues.

The only complex part of this algorithm is the `puzzleBad` function, which determines whether a given set of clues makes an good puzzle, that is admits a unique solution. To this end we again use *aspps*. If the set of clues(the clue of the form "*the cell* (i, j) *contains value* k" is coded as a fact of the form $place(i, j, k)$ - thus the set of clues corresponding to our running examples contains the clues such $place(3, 5, 7)$) can be no further reduced without forfeiting uniqueness of solutions, it is returned. The function `puzzleBad` tests exactly this.

But *aspps* allows not only for finding a solution, but allows to trace the difficulty of finding of a solution. Specifically, *aspps* may work in an incomplete mode; it may use only some means to solve the solution, or use its complete power. So far, *aspps* allows for three modes of solving (in the increased degree of completeness):

1. Solving in the grounder
2. Solving with so-called *1-lookahead*
3. Solve by complete search of the search tree of partial solutions

Here is what it means. As mentioned above, the grounder transforms the very short description of the problem (this description is written in predicate calculus, just that it is streamlined so it is understandable to the machine) into a propositional theory, without individual variables (but with propositional variables - grounded predicate letters). But the grounder execute so-called *Boolean Constraint Propagation* (BCP), essentially finding more facts that *stare in your face as you solve it* [2]. It may happen that the problem is so easy that this is enought to find a solution. The *1-lookahead* is a technique in which the solver, besides of Boolean Constraint Propagation can make a single guess and then check if the resulting theory admits a solution. The complete search uses both the previously mentioned techniques and if stuck, makes additional guesses. If a guess turns out wrong (leads to contradiction), the opposite value for the last guess is tried. If both eventually fail, backtrack (or even stronger form of backtrack, *backjump* [BS97]) is applied.

[2]By staring in your face we mean: "There isbjust one possibility for a value and we see it now (for instance 8 values are already assigned in a row, column, or section).

The following results can arise from applying Aspps.

- The puzzle has no solutions. This situation should never happen, because each iteration tests a less-constrained puzzle than the previous one.

- The grounder finds a solution without lookahead. The puzzle is "good" and the clue may be safely removed.

- The grounder finds the solution, but it needs lookahead. Depending on how difficult we wish to make the puzzle, we might call the puzzle either "good" or "bad". In practice, unless we are trying to generate a hard puzzle, we tell the grounder not to use lookahead. So when this situation arises, we call the puzzle "good".

- Aspps requires the full solver and backtrack to find a sole solution. Although the puzzle is well-formed, it is too hard for general enthusiasts. We call the puzzle "bad".

- The puzzle has multiple solutions. The missing clue is required to keep the puzzle well-formed. The puzzle is "bad".

These last two situations require the solver to distinguish. But we consider both to be "bad". Therefore, we don't ever call the solver. Either the grounder can solve the puzzle ("good") or it can't ("bad").

If we are trying to generate a hard puzzle, we let the grounder use one-step lookahead. We also apply one more test after reaching the final set of clues: Can the grounder solve the final set of clues without using lookahead? If so, then the puzzle isn't hard enough, but removing any clues would make

it ill-formed or too hard. In this case, we reject the puzzle completely and start afresh.

Let us also observe, that the Algorithm 3 can be used to make puzzles *easier*. The reason for this is that we do not have to return the minimal set of clues. We can break the loop earlier, and the resulting puzzle will be only easier. Our own puzzles do not break earlier, but if one wants to produce less interesting puzzles, a numerical switch such as "no more than 30 clues left" can be used.

8 Generating a hint sequence

In a cellular puzzle, most atoms turn out to be false. For example, in the puzzle of Figure 1 (page 5), it turns out that `place(1,1,5)` is true. Therefore the grounder at some point also derives that `place(1,1,1)` is false, as is `place(1,1,2)` and all the other related values. The positive (true) instances of `place` are the interesting ones. By examining the grounder's log, we can determine the order in which it discovers positive atoms by unit propagation. That list begins with the clues themselves. The rest of the positive instances of `place` in the log form the hint sequence.

We show now the hint sequence for our running example:

9 Performance

The logic-program approach to generating cellular puzzles is remarkably efficient in programmer time. Each puzzle type requires only a few lines of Aspps code. The algorithm of Figure 3 is encoded in about 500 lines of

```
 1:bE  2:eF  3:eE  4:fE  5:hG
 6:iF  7:fD  8:cH  9:cI 10:gG
11:fG 12:cB 13:eI 14:hH
15:bH 16:eA 17:hC 18:gI
19:gA 20:fC 21:fA 22:hA
23:aA 24:dC 25:bG 26:aG
27:aB 28:dA 29:dB 30:bA
31:iG 32:iC 33:bB 34:bF
35:gD 36:gF 37:cD 38:bD
39:cC 40:bC 41:eB 42:eC
43:hB 44:aE 45:dD 46:dE
47:aF 48:hE 49:iE 50:hI 51:iI
52:dG 53:dI 54:gH 55:dH
56:aC
```

Figure 4: The sequence of hints generated by the solver

Perl [WS90], much of which is devoted to generating formatted puzzle and hint output.

Given the Aspps rules, generating a puzzle has two phases: Finding the solution and reducing the clues. The time needed for the first phase depends, of course, on the complexity of the constraints and the size of the puzzle. On a 3GHz Pentium 4 running Linux, we accomplish the first phase for Sudoku puzzles of various sizes in time and memory shown in Figure 5. The time for the first phase has a large variance, especially for larger puzzles. Figure 5 also shows the time for each iteration in the second phase. This value has much smaller variance.

10 Modifying classical sudoku

The aspps program that computes the the classical sudoku solutions has a very compact form. It expresses four basic constraints (we get four formulas, shown above in Figure 2). It turns out that various additional constraints can be further imposed. The first type of modification of sudoku (existing for all types of sudoku considered here) is one where we require that each diagonal also contains the numbers 1..n exactly once. The sudoku in Figure 1 is not a diagonal sudoku, because the number 1 occurs on the NE diagonal twice (whereas 2 does not occur there at all). It is very easy to generate diagonal sud kus. First, let us observe that our constraints can be written as:

1. $\forall_y \exists_x place(x, x, y)$
2. $\forall_y \exists_x place(x, 9 - x, y)$

(For sudokus of dimentions 16, 25, 36 etc, the formula (2) needs to be modified in an obvious waya) It is very easy to write the diagonal constraints in aspps:

(1') $1\{place(I, I, N)[I]\}1$

(2') $1\{place(I, 9 - I, N)[I]\}1$

We observe that while the fist part of the process is modified, the second phase does not change at all, except that the badness is tested with respect to a different program.

The second modification we implemented is related to the fact that we deal with the odd length of the side of the grid. In such circumstations each

quadrant (segment) has a center cell. We can require that these n cells hold different numbers. We call such puzzles CenterDot sudokus. There are such sudokus and they are regularly published on our site.

There are sudoku solutions that are both diagonal sudokus and CenterDot. The sizes of the clue set are much smaller for both diagonal sudokus and CenterDot sudokus, thus putting a bigger cognitive strain on the puzzle enthusiast.

There are other generalizations; for instance when we consider (in 9×9 grid) corners, centers of sides and the center cell. All these can be easily implemented.

What to do when the side of the grid is not a square? Here a generalization is still possible. The idea is to partition n^2 cells into contiguous (and esthetically pleasing) n regions, each of size n. Those are of special interest to younger puzzle enthusiasts for whom 4×4 sudoku is to easy, but 9×9 sudoku of type we routinely publish – too difficult. We call these sudoku variations MultiSpot sudokus and our experience is that while for $n = 5, 6, 7$ those are generally easy, for $n = 8$ they may be pretty tough. Again, please visit our site for these generalizations.

The reader must have noticed that up to now we have used cardinality constraints with both upper and lower bounds equal to 1. Those are usually called *choice* rules. But it turns out that there are variations of sudoku where we do not stipulate that the solution is a latin square. We implemented two such puzzles: Multifour and DoubleNine. In the first of these puzzles, we have a grid of side 12, and we require that each number in 1..4 occurs in each row and in each column exactly three times (of course tough enthusiasts can

size	initial seconds	iteration seconds	MB
16×16	0.2	0.10	4
25×25	0.6 – 2.7	0.14	6
36×36	4.8 – 54	0.24	14
49×49	22 – 300	0.49	30

Figure 5: Time and memory requirements for generating Sudoku puzzles

generalize it immediately). Here, we do not use the basic program generating latin square, because the resulting grid is not a latin square. Instead we modify the *aspps* rules (7) and (8) to:

$$3\{place(I, J, N)[I]\}3$$

and

$$3\{place(I, J, N)[J]\}3$$

Yet another variation (this time on classical sudoku) is a square grid of side 18, where we put the following constrains. First, every row and column contains each number from 1..9 twice. Second, there are 36 3×3 sections (yes, just count!). We require that each section contain every number in 1..9 exactly once.

(FIGURE NEEDED)

11 Conclusions

REUSE MIREK TEXT AS MUCH AS POSSIBLE

Puzzle

	A	B	C	D	E	F	G	H	I
a							1		4
b						2			3
c				9	5		2		
d									1
e									
f		9						8	
g	1			2	9	3			
h						8	6	7	
i				7	4				

Solution

	A	B	C	D	E	F	G	H	I
a	2	8	5	6	3	7	1	9	4
b	9	1	6	4	8	2	7	5	3
c	3	7	4	9	5	1	2	6	8
d	6	4	8	3	7	9	5	2	1
e	5	2	1	8	6	4	9	3	7
f	7	9	3	1	2	5	4	8	6
g	1	6	7	2	9	3	8	4	5
h	4	3	9	5	1	8	6	7	2
i	8	5	2	7	4	6	3	1	9

Figure 6: A 9×9 diagonal Sudoku puzzle and its solution

Puzzle

	A	B	C	D	E	F	G	H	I	J	K	L
a	3					1	3		1	4	4	
b		1		2			4	3	4	2	3	
c	1	1	4	4					3	1		
d		1	1	2				2				2
e	2		1	1		1		2	4	4		4
f	2		2	1		4		1		4		
g		3			4		3					2
h	4		4		2			2		1	2	1
i		4			3	2		3				1
j	3	3		2	4		2			2		3
k							2	1	3	2		2
l	4	3		3	2	2					4	1

Solution

	A	B	C	D	E	F	G	H	I	J	K	L
a	3	2	2	3	2	1	3	1	1	4	4	4
b	2	1	3	2	1	1	4	3	4	2	3	4
c	1	1	4	4	4	2	2	3	3	1	2	3
d	3	1	1	2	1	3	4	2	4	3	4	2
e	2	2	1	1	3	1	3	2	4	4	3	4
f	2	2	2	1	3	4	4	1	1	4	3	3
g	1	3	2	1	4	4	3	4	2	3	1	2
h	4	4	4	3	2	3	1	2	3	1	2	1
i	1	4	4	4	3	2	1	3	2	3	2	1
j	3	3	1	2	4	4	2	4	1	2	1	3
k	4	4	3	4	1	3	2	1	3	2	1	2
l	4	3	3	3	2	2	1	4	2	1	4	1

Figure 7: An $n = 3$ MultiFour puzzle and its solution

Puzzle

	A	B	C	D	E	F	G
a			4	3			
b		7					
c	3	6	1		4		
d			5				3
e							
f	7					1	
g							

Solution

	A	B	C	D	E	F	G
a	2	5	4	3	7	6	1
b	1	7	6	5	2	3	4
c	3	6	1	7	4	2	5
d	4	2	5	1	6	7	3
e	6	3	7	4	1	5	2
f	7	4	3	2	5	1	6
g	5	1	2	6	3	4	7

Figure 8: An $n = 7$ MultiSpot puzzle and its solution

References

[ARMS03] F. Aloul, A. Ramani, I. Markov and K. Sakallah. PBS v0.2, incremental pseudo-boolean backtrack search SAT solver and optimizer, `http://www.eecs.umich.edu/~faloul/Tools/pbs/`, 2003.

[BS97] R.J. Bayardo, Jr and R.C. Schrag. Using CSP Look-back techniques to solve real-world SAT instances. "Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-1997), pp. 203–208. 1997.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving, *Communications of the Association for Computing Machinery*, 7:394–397, 1962.

[DK74] J. Denes and A.D. Keedwell. *Latin Squares and their applications*. Academic Press, 1974.

[EFLP00] Thomas Eiter, Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Declarative problem-solving in DLV. In Jack Minker, editor,

Logic-Based Artificial Intelligence, pages 79–103. Kluwer Academic Publishers, Dordrecht, 2000.

[ET01] D. East and M. Truszczyński. *aspps* — an implementation of answer-set programming with propositional schemata. In *Proceedings of Logic Programming and Nonmonotonic Reasoning Conference, LPNMR 2001*, volume 2173, pages 402–405. Lecture Notes in Artificial Intelligence, Springer Verlag, 2001.

[FJ05] B. Felgenhauer and F. Jarvis. Enumerating possible Sudoku grids. `web.info.tu-dresden.de/~bf3/sudoku/sudoku.pdf`. 2005.

[MMZZM01] M. Moskewicz, C. Madigan, Y. Zhao. L. Zhang and S. Malik. SAT solver *chaff*. `http://www.ee.princeton.edu/~chaff/`. 2001.

[NS97] I. Niemelä and P. Simons. Smodels — an implementation of the stable model and well-foundd semantics for normal logic programs. In *Logic Programming and Nonmonotonic Reasoning (the 4th International Conference, Dagstuhl, Germany, 1997)*, volume 1265 of *Lecture Notes in Computer Science*, pages 420–429. Springer-Verlag, 1997.

[TFE05] Wikipedia: The Free Encyclopedia. Sudoku, 2005. `http://en.wikipedia.org/wiki/Sudoku`.

[WS90] L. Wall and R. L. Schwartz. *Programming Perl*. O'Reilly and Associates, 1990.

[WNS97] M. Wallace, S. Novello and J. Schimpf. ECL^{i}PS^{e}: A Platform for Constraint Logic Programming. `http://www.icparc.ic.ac.uk/eclipse/reports/eclipse.ps.gz`