Katholieke
Universiteit
Leuven

**Department of
Computer Science**

APLAI (H02A8A)

Project: Sudoku & Shikaku

Jeroen Craps (r0292642)
Jorik De Waen (TODO)

Academic year 2015–2016

Contents

1 Introduction

We will discuss several declaritive systems for solving sudoku and shikaku puzzles. An attempt at a proper discussion about the different systems is made. An alternative viewpoint to the sudoku problem is proposed and tested. Together with the experiments we will further explain the results and our findings of this project.

2 Task 1: Sudoku

2.1 Introduction

An *n-sudoku* puzzle contains n^4 squares, in an n^2 by n^2 grid, and has n^2 blocks. These blocks are of size N by N. Sudoku puzzles are proven to be **NP-Complete**. The collection of these rows, columns and zones are all units in the puzzle. So we come to the following statement:

A puzzle is solved if every unit contains every element
from the interval 1 *to* n^2 *exactly once.*

This means that every unit is filled with a permutation of $[1, 2, \cdots, n^2$. The classical viewpoint for Sudoku states that all numbers in a row must be different, that all numbers in a column must be different, and that all numbers in a block must be different.

$x_{1,1}$	$x_{1,2}$	$x_{1,3}$	$x_{1,4}$	$x_{1,5}$	$x_{1,6}$	$x_{1,7}$	$x_{1,8}$	$x_{1,9}$
$x_{2,1}$	$x_{2,2}$	$x_{2,3}$	$x_{2,4}$	$x_{2,5}$	$x_{2,6}$	$x_{2,7}$	$x_{2,8}$	$x_{2,9}$
$x_{3,1}$	$x_{3,2}$	$x_{3,3}$	$x_{3,4}$	$x_{3,5}$	$x_{3,6}$	$x_{3,7}$	$x_{3,8}$	$x_{3,9}$
$x_{4,1}$	$x_{4,2}$	$x_{4,3}$	$x_{4,4}$	$x_{4,5}$	$x_{4,6}$	$x_{4,7}$	$x_{4,8}$	$x_{4,9}$
$x_{5,1}$	$x_{5,2}$	$x_{5,3}$	$x_{5,4}$	$x_{5,5}$	$x_{5,6}$	$x_{5,7}$	$x_{5,8}$	$x_{5,9}$
$x_{6,1}$	$x_{6,2}$	$x_{6,3}$	$x_{6,4}$	$x_{6,5}$	$x_{6,6}$	$x_{6,7}$	$x_{6,8}$	$x_{6,9}$
$x_{7,1}$	$x_{7,2}$	$x_{7,3}$	$x_{7,4}$	$x_{7,5}$	$x_{7,6}$	$x_{7,7}$	$x_{7,8}$	$x_{7,9}$
$x_{8,1}$	$x_{8,2}$	$x_{8,3}$	$x_{8,4}$	$x_{8,5}$	$x_{8,6}$	$x_{8,7}$	$x_{8,8}$	$x_{8,9}$
$x_{9,1}$	$x_{9,2}$	$x_{9,3}$	$x_{9,4}$	$x_{9,5}$	$x_{9,6}$	$x_{9,7}$	$x_{9,8}$	$x_{9,9}$

Figure 1: Example of the classical viewpoint on a 3-sudoku puzzle.

We can now define the Sudoku problem formally. A value at position row i and column j in the grid is represented as $x_{i,j}$. We define the following constraints for an *n-sudoku* puzzle:

Variables:	$x_{1,1}, x_{1,2}, \cdots, x_{i,i}, x_{i,i+1}, \cdots, x_{n^2,n^2} \in \left\{1, 2, \cdots, n^2\right\}$	
Constraints:	$\forall i, j, k \in \left\{1, 2, \cdots, n^2\right\} : x_{i,k} \neq x_{j,k}$	**Rows**
	$\forall i, j, k \in \left\{1, 2, \cdots, n^2\right\} : x_{i,j} \neq x_{i,k}$	**Columns**
	$\forall i, j \in \{0, 1, \cdots, n-1\}, \forall a, b, c, d \in \{0, 1, \cdots, n\} : x_{i*n+a,j*n+b} \neq x_{i*n+c,j*n+d}$	**Blocks**

2.2 Alternative viewpoint

In this section we will discuss an alternative viewpoint for an classical *n-sudoku* puzzle. Now we will define the puzzle as a problem of n^2 rows, so we group all the elements in a row into a single variable. These rows will be permutations of the list $[1, 2, \cdots, n^2]$. The reasoning behind this viewpoint is to reduce the amount of constraints needed to express the problem. The amount of variables need has been reduced from n^4 to n^2.

Again we can define this more formally. We define n^2 variables r_i instead of the n^4 variables in the classical sudoku viewpoint. **TODO!!!** So to visualize this:

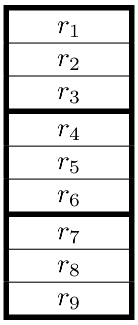

Figure 2: Example of the alternative viewpoint on an 3-sudoku puzzle.

Variables:	$r_1, r_2, \cdots, r_{n^2} \in \{[1,2,\cdots,n^2],[2,1,\cdots,n^2],\cdots,[n^2,n^2-1,\cdots,1]\}$	
Constraints:	$\forall i,j,k \in \left\{1,2,\cdots,n^2\right\} : r_{i,k} \neq r_{j,k}$	**Columns**
	$\forall i,j \in \{0,1,\cdots,n-1\}, \forall a,b,c,d \in \{0,1,\cdots,n\} : r_{i*n+a,j*n+b} \neq r_{i*n+c,j*n+d}$	**Blocks**

2.3 Criteria

The criteria according to us for a good viewpoint are the following: an easy understanding of the representation, the computational complexity to get to a solution, the amount of backtracks or logical inferences that are necessary to get a solution from the implementation of the proposed viewingpoint. In our opinion the alternative viewpoint that we suggested is a good viewpoint, because the amount of backtracking that is necessary is greatly reduced by reducing the amount of variables in the problem and restricting their domain.

2.4 Channeling Constraints

The channeling constraints between the classical viewpoint and the alternative viewpoint can only be defined in one direction. It is only possible to define the values of the individual cells in the classical viewpoint when coming from the rows in the alternative viewpoint. If a row is assigned a certain permutation as its value, then the elements of the corresponding row in the original viewpoint can be assigned the values from this permutation. In the other direction this is not possible unless all of the variables in a row have been assigned a value and that they are all different from each other. So that these values from a permutation of the original list. The alternative viewpoint has as main advantage that the row-constraints are implied when declaring its domain. The influence of this is mainly noticed in the search process, instead of instantiating every variable separatly the entire row will be instantiated at once. Due to these channeling constraints being so trivial we think that there is no point in implementing them, since the influence would be minimal.

2.5 Implementation

2.5.1 ECLiPSe

Original viewpoint In the original viewpoint we view the sudoku as a matrix with n^4 variables. The constraints on this are the following. We extract the required information from this matrix by defining rows, columns and blocks. For each row, column and block we say that every value in these lists must be different. So all of the constraints are exactly the same, but the way of extracting this information from the matrix is different.

A row with index I is defined in a matrix by *Row is Sudoku*$[I, 1..N^2]$. For extracting a column with index I we use *Col is Sudoku*$[1..N^2, I]$. While extracting the information for the rows and columns is straight forward, it becomes clear when extracting the variables required to fulfill the block constraints. We define the block with index (I, J) as follows. The indices I, J are elements from the interval $[0, 1, \cdots, (n-1)]$. The block at the upperleft corner of the puzzle will be named the $(0, 0)$-block and the block at the bottomright corner will be the $(n-1, n-1)$-block. For each pair (I, J) we now need to define the list containing its n^2 elements. We need two new variables (K, L) to define the rows and columns needed to form these blocks. For example, the X coordinate in our matrix will be equal to $(I*n)+K$. So for the $(0, 0)$ the required rows are $1, 2, \cdots, n-1$. A similar approach can be made for the columns where $Y = (J * n) + L$. Combining all of the possible X and Y values based on the given (I, J) pair leads to the all of the indices required to create the specified (I, J)-block.

```
( multifor([I,J],0,(N-1)), param(Sudoku, N) do
        ( multifor([K,L],1,N), param(Sudoku, I, J, N), foreach(B,Block) do
                R is (I*N)+K,
                C is (J*N)+L,
                B is Sudoku[R,C]
    ),
        alldifferent(Block)
).
```

Important to note is that after all of these constraints the most important aspect of the problem remains how to decide if every value in a list is unique. There a couple of different ways to implement this. In this first viewpoint every variable is instantiated separately with an element from the domain $[1, 2, \cdots, n^2]$.

As can be seen in the code snippet above we utilise the buildin *alldiferent/1* function from the ic library, which we have imported into our program with the *:- lib(ic)* command. We could have used passive constraints where the entire search space would be used instead of the pruned version that is created by the ic library. This would have caused a lot of backtracking to occur while looking for a possible solution since every variable would be instantiated separately from the others. For every variable in the list the rest of the list is checked to see if it doesn't contains the same value.

Another option was to use the suspend library which puts constraints on the variables before searching. But afterwards it still iterates over the entire search space, because it still doesn't utilise the additional constraints to reduce the search space of the other variables in a list/column/block.

We ended up choosing the ic library for two reasons. Firstly this allows us to define the constraint before starting the search process. Secondly, the search domain is reduced while search by the constraints so that less backtracking is required. This is called Forward Checking.

Further optimalisations can be achieved by value and variable ordering. Which we can change by using different methods in the *search(+L, ++Arg, ++Select, +Choice, ++Method, +Option)* command of the ECLiPSe library.

-

Alternative viewpoint In this alternative viewpoint we view the puzzle as a list of lists. There are n^2 list with each n^2 variables. In this representation we see this list of variables as a single variable of which the possible values are all of the possible permutations of the sequence $[1, 2, \cdots, n^2]$.

```
( foreach(Row,Sudoku), param(N2)  do
      permutation([1..N2],Row)
)

permutation(Sequence,Permutation):-
   Permutation :: Sequence,
   alldifferent(Permutation).
```

By defining the entire row as a variable we no longer need the row constraints since these are implied by the domain of the variables. The column and block constraints are similar to the original viewpoint, except for the fact that instead of working with the matrix we now need to iterate over the lists to extract the necessary variables to form the columns and blocks.

```
(foreach(Row,Sudoku), param(N2) do
     (for(J,1,N2), foreach(E,Col) do
         selectElement(E,J,Row)
     ),
     alldifferent(Col)
)
```

As a consequence of this viewpoint is that we use active propagation by only having permutation of the sequence 1 to n^2. The reasoning behind this is that we will no longer attempt a certain value for a variable in a row when the value already exists in the row. This kind of active propagation is already done when we utilise the ic library, so we are not expecting to see a big difference in performance between these two implementations.

We will further discuss the results in the section **??**

2.5.2 CHR

2.5.3 Decision

2.6 Experiments

2.6.1 Results

2.6.2 Heuristics

2.6.3 Difficulties

2.7 Conclusions

3 Introduction

The goal of this project is to design an architecture for a smart home system. Smart home systems provide home automation, which is the use and control of home appliances remotely or automatically. This system improves convenience, comfort, energy efficiency and security for the user. The system consists of a smart home gateway and an online system. The smart home gateway can communicate with plug-and-play sensors and actuators, which are spread around the home of the user. The online system will communicate with the gateway to make sure that any information is synchronized and properly processed.

The points of differentiation from existing technology providers will be twofold. Firstly, as the HomeSys gateway will feature a 3G communication module, it will will be 100% plug-and-play, i.e. not requiring any pre-existing infrastructure. Secondly, the gateway will be designed with explicit focus on flexibility, hereby allowing the development and integration of new smart-home end-to-end applications, perhaps even allowing third party applications.