KULeuven

Department of Computer Science

ADVANCED PROGRAMMING LANGUAGES FOR A.I. (H02A8A)

Project: Sudoku & Shikaku

Jeroen Craps (r0292642)
Jorik De Waen (r0303087)

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 $n^2 \times n^2$ grid, and has n^2 blocks. These blocks are of size $n \times n$. Sudoku puzzles are form of latin squares these are proven to be **NP-Complete**. This implies that the general case of sudoku puzzles are also **NP-Complete**. The collection of these rows, columns and blocks are all **units** in the puzzle. So we come to the following definition:

Definition: *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:	$x_{1,1}, x_{1,2}, \cdots, x_{i,i}, x_{i,i+1}, \cdots, x_{n^2,n^2} \in \{1, 2, \cdots, n^2\}$	
Constraints:	$\forall i, j, k \in \{1, 2, \cdots, n^2\} : x_{i,k} \neq x_{j,k}$	**Rows**
	$\forall i, j, k \in \{1, 2, \cdots, n^2\} : x_{i,j} \neq x_{i,k}$	**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 for the classical viewpoint of a *n-sudoku* puzzle. Now we will define the puzzle as a problem of n^2 rows instead of n^4 separate variables, so that we can 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 goal of this viewpoint is to reduce the amount of constraints needed to express the problem. The amount of variables that are used 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 $x_{i,j}$ in the classical sudoku viewpoint. The variable r_i is the collection of $x_{i,1}, x_{i,2}, \cdots, x_{i,n^2}$ in this case. A visualisation of this can be seen in Figure 2

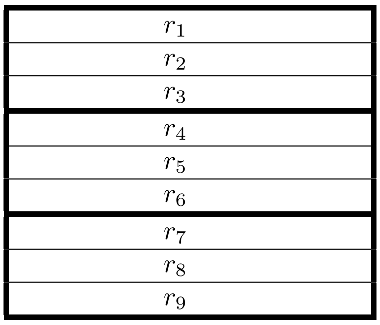


Figure 2: Example of the alternative viewpoint on an 3-sudoku puzzle.

The row constraints from the original viewpoint are no longer necessary because they are already included in the fact that each row has to be a permutation of the sequence $[1, 2, \cdots, n^2]$. We will still need to define the column and block constraints. To do so we will say that $r_{i,j}$ is the element with index j in the ith row of the puzzle. So that we can still properly define the columns and blocks.

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 for a good viewpoint according to us are the following: an easy understanding of the representation, a low computational complexity to get to a solution, a low amount of backtracks or logical inferences that are required to get a solution from the implementation of the proposed viewpoint. In our opinion the alternative viewpoint that we suggested is a good viewpoint, because the amount of required backtracks should be greatly reduced. This is due to the decrease in variables in the problem and the further restriction of their domain.

2.4 Channeling Constraints

The channeling constraints between the classical viewpoint and the alternative viewpoint we proposed can only be defined in one direction. It is only possible to define the values of the individual variables in the classical viewpoint when we have the rows in the alternative viewpoint. This should always be valid in any case $r_i \to x_{i,1}, x_{i,2}, \cdots, x_{i,n^2}$, but in the other direction this is not the case. 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 isn't possible unless all of the variables in a row have been assigned a value and they are all different from each other. So that these values form a permutation of the original list. The alternative viewpoint has as main advantage that the row-constraints are automatically implied when defining the domain of the variables. The influence of this would be mainly noticed in the search process, instead of instantiating every variable separately 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

In this section we will be discussing the implementation of our sudoku solvers. The reasoning behind why we chose `ECLiPSe` and `CHR` is later explained in section 2.5.3. As mentioned previously we will not be implementing any channeling constraints due to them being too straightforward and it is only possible to define them in one direction.

2.5.1 ECLiPSe

Original viewpoint In the original viewpoint we view the sudoku as a matrix with n^4 variables. We extract the required information from this matrix by defining rows, columns and blocks. The constraints that are required for this viewpoint of the problem are the following. 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 indexes 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 indexes 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 are 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 utilize the built-in *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 utilize 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 searching by the constraints so that less backtracking is required. This is called Forward Checking.

Further optimizations 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. We have tried out multiple of these methods to decide which one performs best with our given implementation.

- **input first**: The order of the instantiation of the variables is done in the same order as they have been put into the system by our implementation. This method utilizes no extra information about the domain size of the variables.
- **first fail**: The variable with the smallest domain size will be instantiated first, meaning that the deeper we go into the search tree the larger the domain size should be. This would normally imply that failure should occur earlier rather than later.
- **anti first fail**: In this method the variables with the largest domain size will be instantiated first. This implies that harder decisions are made later in the search tree.

The result of these experiments can be found in the section 2.6. Where we will further evaluate these different methods, so that we can decide which one produces the best results.

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.

```
(for(J,1,N2) do
        (foreach(Row,Sudoku), param(J), 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 utilize 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 2.6

2.5.2 CHR

First of all we want to clearly mention how the implementation of our sudoku solver works. To start everything off we retrieve the `Puzzle` by its name and calculate n^2, which we add to the constraint store. Now the *import(Puzzle)* will be added to the constraint store to further add the elements of the puzzle. Afterwards the constraint store is printed out, the search is started and the solution is printed out once more. We will also empty the constraint store afterwards for convenience.

```
solve(Name) <=>
        puzzles(Puzzle,Name),
        length(Puzzle,N2),
        dom(N2),
        import(Puzzle),
        show,
        search,
        !,
        show,
        clean.
```

The way the elements are added to the constraint store and processed beforehand both depend on the viewpoint that is used.

Original viewpoint In the original viewpoint the *import(puzzle)* will be used to change the value of the element at the specified position in the existing lists of length n^2. But before this can be done we need to specify a few constraints. For every column, row and block we will add a constraint to the constraint store. Each of these constraints will contain an index to specify about which column/row/block we are talking about and a list of the length n^2.

Now we can start adding the known elements into the lists. So for example if there is an element at position (1,2) with value 3, then in the *col(2,ColList)* constraint the list will be changed so that at the position with index 1 the value 3 is placed. Of course for the other lists this is also done, so at the position with index 2 in the RowList the value 3 is placed. For the blocks, the exact position in the list is calculated from the row and column of the given value. Something to explain is that the column, row and block constraints aren't removed from and added again to the constraint store with their updated list because we don't want these rules to trigger again.

```
dom(N), row(Y,RowList), col(X,ColList), block(BlockN,BlockList)
        \ import_row([CVal|Vals],Y,X) <=>
        in_block(Y,X,BlockN,N), position_in_block(Y,X,PosInBlock,N) |
        nth1(X,RowList,CVal), nth1(Y,ColList,CVal),
        nth1(PosInBlock, BlockList, CVal),
        X1 is X + 1, import_row(Vals,Y,X1).
```

After adding all of the known values into the domain, the only thing that remains is to specify that all of the elements in the list need to be different from each other. We will force this constraint by using a built-in function namely the *alldifferent* function. This function will enforce that all of the elements in the list are different from each other, if not the function will fail.

```
imported, col(_,L) ==> all_different(L); fail.
imported, block(_,L) ==> all_different(L); fail.
imported, row(_,L) ==> all_different(L); fail.
```

The only thing that for remains for us is to iterate over the search space. We have done this by using the built-in labeling functionality. We show the puzzle before and after solving it. Finally we end by cleaning our entire constraint store.

Alternative viewpoint In the alternative viewpoint we will exchange the *import*$([CRow|Rows], Y)$ for two new constraints namely *import_row*$(CRow, Y, 1)$ and *import*$(Rows, Y1)$ which is Y incremented by one.

This method of inserting the domain knowledge into the constraint store will be continued recursively until all the given information is handled. So depending on the fact that an element is known or not will determine how it is added to the constraint store. If the element is a variable *option*(X, Y, L) is used where X stands for column, Y stands for row and the L contains all possible values in the domain. On the other hand if a element is a value, we add *known*(X, Y, V) to the constraint store where this is the position of the specified value in the puzzle. When there is only one remaining value in the domain of an *option*(X, Y, L) we will simplify this to *known*(X, Y, V) with V being the only remaining value in the domain.

```
import([],_) <=> true.
import([CRow|Rows],Y) <=>
     import_row(CRow,Y,1), Y1 is Y + 1 ,import(Rows,Y1).

import_row([],_,_) <=> true.
import_row([CVal|Vals],Y,X) <=>
     nonvar(CVal) | known(Y,X,CVal), X1 is X + 1, import_row(Vals,Y,X1).

import_row([CVal|Vals],Y,X) <=>
     var(CVal) | upto(9,L), option(Y,X,L), X1 is X + 1, import_row(Vals,Y,X1).
```

Now that all of the information is added to the system we will use constraints to change the domains of the *option* elements in the constraint store. If an known element is on the same row as a variable that is not instantiated yet, we can remove the value of the known element from the possibilities from the variable. We also do this for the columns.

```
%% ROW Constraints
known(X,_,V) \ option(X,Y,L) <=> member(V,L) | delete(L,V,NL), option(X,Y,NL).
%% COL Constraints
known(_,Y,V) \ option(X,Y,L) <=> member(V,L) | delete(L,V,NL), option(X,Y,NL).
```

The block constraints have the same purpose but were less obvious to implement. This was because of the way we needed to decide when a certain variable was occurring in the same block as the known value. For this we used the integer division to give an index to the blocks. As a result we have blocks (0,0) to (n-1,n-1) which we can use to filter the domain of the remaining variables even further. We always remove the existing option from the constraint store before adding an updated one. This way we ensure that all of the rules will be checked for the new constraint.

```
%% BLOCK Constraints
known(X,Y,V), dom(N2) \ option(X1,Y1,L) <=> member(V,L),
          N is integer(sqrt(N2)),
          XB is (X-1)//N, YB is (Y-1)//N,
          XB is (X1-1)//N, YB is (Y1-1)//N
            | delete(L,V,NL), option(X1,Y1,NL).
```

The only thing that remains now is to enumerate the search space. This is done by adding the *search* constraint into the constraint store by the first method we mentioned. When an *option* constraint is present in the constraint store it will be retrieved and stored together with a *permute* constraint. This permute constraint will remain in the constraint store until all of the options at the row specified by the permute constraint are removed from the constraint store.

Due to the enumeration being implemented in Prolog we can backtrack if necessary. Once a single variable is instantiated and a known constraint is added to the store, the remaining variables on that row will remove the selected value from their domain so that there can never be a conflict in a row. Here we instantiated the entire row as a permutation from the sequence one to n^2. We say that failure occurs when an option has no possible values left. If this happens, we will backtrack to the last choice point. The only time a choice is made in our implementation is when we enumerate the possible options in the domain of a variable.

2.5.3 Decision

Before deciding which languages we would use for our implementation, we made a comparison between the programming languages. We will explain why they are usually used, what the idea behind them is and how we would reason about them when applying them to sudoku puzzles. We will start to discussion with `ECLiPSe`. Afterwards we will continue with `CHR` and `Jess`.

ECLiPSe is an constraint logic programming language that is backward-compatible with `Prolog`. The constraint logic programming is used for constraint satisfaction problems. More specific, optimization, planning and other similar kind of problems. These problems are defined by some variables which can take a possible value from a certain domain. A solution for these kind of problems is found when each variable has exactly one value and that it fulfills all of the constraints. `ECLiPSe` provides several libraries which can be used in application programs. In case of the sudoku problem the variables would be the unknown values in the puzzle with their domain being the interval between 1 and n^2. The constraints would be that there can only be one occurrence of a value in a certain row, column or block.

CHR is short for Constraint Handling Rules. In `CHR` we will work with a constraint store which we will use to represent the state of the problem that we are trying to solve. This is used often used to implement Rule Based Systems, an example of this is the domain-specific expert system. This uses rules to make deductions or choices, which is very applicable to the provided problems. By using simplification and propagation rules we will be making committed choices. This means once we change the state of the constraint store this can't be undone. Whenever a new constraint is added to the constraint store every rule is checked to see if it is necessary to execute it. Propagation rules are only executed once for the same constraint. Again for the sudoku problem we would see the constraint store as a way to represent our knowledge of the given information. The row, column and block constraints will change the constraint store to reduce the search space. While we said that `CHR` makes committed choices, it is still possible to backtrack due to the usage of `Prolog`.

Jess stands for the Java expert system shell. It uses backward chaining and works by deriving conclusions from premises. So due to its integration in with Java we can use objects and other Java functionality to fulfill our needs. Just like with `CHR` this can easily be used to implement Rule Based Systems. In this system there are three parts: the Rule Base, the Working Memory (Fact Base) and the Inference Engine. The Rule Base contains functions calls that manipulate the Fact Base, which contains all of the prior information for the problem. While the Inference Engine will match the elements from the Fact Base to the Rule Base. This is done by comparing all the rules to the Fact Base to see which rules need to be activated. Further it will handle conflicts and execute the commands on the Fact Base. The Fact Base will represent the state of our system during the processing of the information. The order in which the rules are activated depends on the patterns. The most specific pattern will be triggered first, but it is possible to prioritize certain rules above others by defining a weight. If we look at Sudoku, we could say that the known values are the Fact Base and that all of the row, column and block constraints are part of the Rule Base. These rules will deduce the options on the other variables.

Decision We eventually chose to use `ECLiPSe` and `CHR` for our implementations of our Sudoku solver. The reasoning behind this was that `CHR` and `Jess` are in a way very similar and we found the syntax of `CHR` much more clearer compared to `Jess`. Since we already had a basic understanding of `Prolog`, it was preferable that the chosen languages had a certain integration with it. `ECLiPSe` was chosen because of its difference from the two other languages and the similarities to `Prolog`. On the other hand `CHR` was chosen due to it being a more understandable high level language that was still closely related to `Prolog`. `Jess` looked less straightforward and suitable for the specified problems in our eyes. It seemed like the least desirable option out of the three.

2.6 Experiments

In these experiments we want to see the difference between the two viewpoints and the different programming languages. But it is also our intention to measure the influence of the different search methods, so that we can find the optimal setup for our implementations. The most important aspects that we will consider are the time it takes to solve the problem, and the amount of inferences that are required, these are good measuring points. During all of our experiments we used the same computer[1]. It is in our interest to time-out the process after a certain time. Due to using the same computer for all the experiments we should get a good indication of the performance of our different sudoku solvers.

2.6.1 Results

Viewpoint 1

	ECLiPSe Input Order		ECLiPSe First Fail		ECLiPSe Anti First Fail		CHR	
	Time (s)	nbOfBt	Time (s)	nbOfBt	Time (s)	nbOfBt	Time (s)	inferences
verydifficult	0.00	0	0.00	0	0.00	0	0.01	157,196
expert	0.00	0	0.01	1	0.00	0	0.03	537,655
lambda	0.00	3	0.00	3	0.00	0	0.93	18,295,296
hard17	0.00	1	0.01	1	1.10	273	0.46	8,960,808
symme	0.14	42	0.07	19	4.09	912	0.49	9,590,025
eastermonster	0.16	51	0.08	33	1.79	351	0.18	3,291,075
tarek052	0.25	59	0.12	35	1.84	398	0.23	4,581,841
goldennugget	0.49	104	0.22	76	0.23	30	0.52	9,906,969
coloin	0.17	88	0.02	8	12.84	3004	0.13	2,240,845
extra1	0.01	0	0.00	1	0.00	0	0.18	3,321,574
extra2	0.00	0	0.00	0	0.00	0	6.79	132,119,829
extra3	0.00	3	0.01	3	0.00	0	0.93	18,295,296
extra4	0.01	4	0.00	3	0.00	0	1.88	37,006,351
inkara2012	0.02	3	0.06	17	1.13	244	0.42	8,091,754
clue18	0.19	69	0.03	8	0.55	129	0.47	9,212,851
clue17	0.00	0	0.00	0	0.00	0	0.24	4,378,228
sudowiki_nb28	0.75	413	0.40	297	0.08	11	2.32	46,412,038
sudowiki_nb49	0.15	48	0.14	58	1.14	242	0.81	15,918,183
peter	0.00	0	0.00	0	0.00	0	0.06	741,743
Average	0.12	47	0.06	30	1.34	294	0.90	22,075,239

Table 1: The results from the experiments on the first viewpoint

- The first thing we can notice is that all of our implementations of the first viewpoint successfully solve all of the sudoku puzzles. We added an extra 16×16 sudoku puzzle to show that our implementations can handle larger puzzles than the original 9×9 sudoku.
- When comparing the `input order` against the `fail first` we see that there is significant difference in performance. The `first fail` performs equally good or even better in almost all cases, expect for a few puzzles like for example **inkara2012**. The `first fail` will make the hard decisions early on with the intention to require less backtracking, which can also be seen in our results.
- On every puzzle, except for a few, the `anti first fail` will perform worse compared to the other implementations in `ECLiPSe`. The puzzles in question are **sudowiki_nb28** and **goldennugget**. Our reasoning is that the variables with the smallest domains don't have a lot of influence on the rest of the variables in these few cases. So almost no progress is made by prioritizing them, while the variables with the larger domain have more influence.
- Our `CHR` implementation works slower than the `ECLiPSe` implementations, except for the `anti first fail` one. We believe this is due to the usage of the built-in libraries in our `ECLiPSe` implementations, these should perform much better than our own written methods in `CHR`.

[1]Intel Core i5-4570S CPU @ 2.90GHz 4

- Which puzzle is the most difficult depends on the search heuristic that is used. In case of the `anti first fail` heuristic the **coloin** puzzle is the hardest, but when we look at the `first fail` heuristic we see that the `sudowiki_nb28` is the most difficult puzzle. Our reasoning is that it mainly depends on the importance of the variables with a small domain, if these variables have a large influence on the domain of the other variables then the `first fail` will see an increase in performance (due to the increased importance of the choice) and a decrease in the amount of backtracks that are made.

Viewpoint 2

	ECLiPSe Input Order		ECLiPSe First Fail		ECLiPSe Anti First Fail		CHR	
	Time (s)	nbOfBt	Time (s)	nbOfBt	Time (s)	nbOfBt	Time (s)	inferences
verydifficult	0.00	0	0.00	0	0.00	0	0.35	6,462,861
expert	0.01	0	0.00	1	0.00	0	1.87	34,471,978
lambda	0.00	3	0.08	3	0.00	0	TO	
hard17	0.00	1	0.00	1	1.18	273		
symme	0.15	42	0.00	19	4.39	909		
eastermonster	0.19	52	0.11	33	1.97	352		
tarek052	0.30	59	0.18	35	1.97	398		
goldennugget	0.53	104	0.30	76	0.00	30		
coloin	0.25	88	0.00	8	13.71	3006		
extra1	0.00	0	0.00	1	0.00	0		
extra2	0.00	0	0.00	0	0.00	0		
extra3	0.00	3	0.00	3	0.00	0		
extra4	0.00	4	0.00	3	0.00	0		
inkara2012	0.00	3	0.00	17	1.52	252		
clue18	0.22	69	0.00	8	0.00	129		
clue17	0.00	0	0.00	0	0.00	0		
sudowiki_nb28	0.95	413	0.59	297	0.00	11		
sudowiki_nb49	0.22	47	0.21	58	2.08	265		
peter	0.00	0	0.00	0	0.00	0		
Average	0.12	47	0.06	30	1.34	296		

Table 2: The results from the experiments on the second viewpoint.

- Again we notice that all of the `ECLiPSe` implementations are able to solve all of the puzzles. In case of the `CHR` implementation we notice that almost none of the puzzles can be solved under the time limit that we specified.
- The moment we start using the `first fail` heuristic, we see a drastic increase in performance for almost all puzzles. The only puzzle that performs worse is the **lambda** puzzle, here we see a decrease in performance. Our reasoning here is that the optimal ordering of the variables is close to the `input order`, while currently we are looking at the variables with the smallest domain. This suspicion is confirmed when we take a look at the performance of the `anti first fail` heuristic, which also performs better than the `first fail` heuristic.
- When we compare the `anti first fail` against the other implementations we see a decrease in performance except for a couple of instances where it performs drastically better than the other heuristics. The puzzles in question are **lambda**, **goldennugget**, **sudowiki_nb28** and **clue18**. We saw a similar pattern in the original viewpoint. The worst puzzle for this viewpoint was **coloin**, where it takes 13.71 seconds compared to 0.25 in the `input order` heuristic. The reasoning again is that in this puzzle `anti first fail` is the worst possible search heuristic.
- We also observe an increase in the amount of backtracks when using the `anti first fail` heuristic, which we expected.
- In general the alternative viewpoint performs worse than the original viewpoint. We think that the reason for this is that due to using the `ic library` we already use the forward checking that we were trying to achieve by using this viewpoint. So we end up creating extra overhead by imposing these constraints on the variables ourselves.

- The implementation in `CHR` works pretty poorly considering the other results. Our reasoning behind why it is not able to finish the majority of the puzzles is that we go too deep into our search tree before we observe a failure. With improvements to this method we should be able to increase the further performance of our implementation in `CHR`.

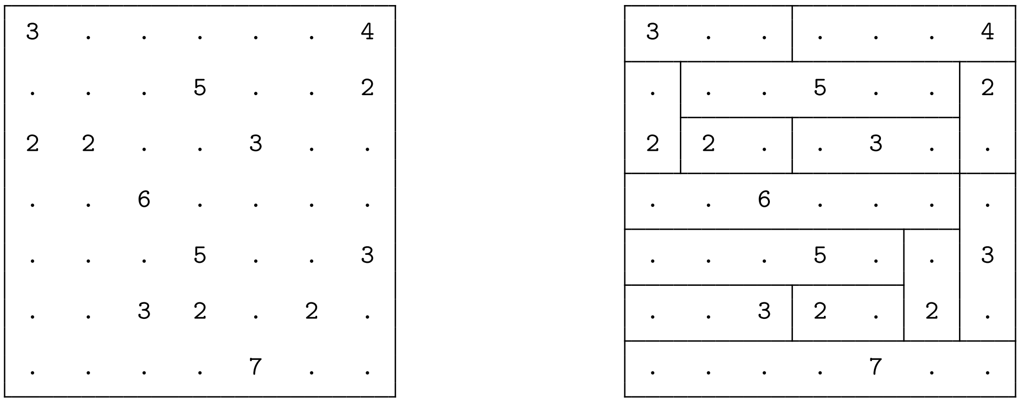
2.6.2 Conclusions

We can see that our alternative viewpoint in general is performing worse on almost all puzzles, we believe this is due to the fact that the `ic library` will use forward checking which we tried to simulate with our implementation. The conclusion is that in almost all cases the `first fail` search heuristic performs best in the original viewpoint. While the `anti first fail` performs the worst overal, except for our implementation in `CHR`. We believe that `CHR` is less suited for problems that can be explained in a very straightforward fashion compared to `ECLiPSe`. Adding all of the constraints to the constraint store and processing them by using the simplification or propagation rules takes too much time compared to the quick built-in libraries and methods that can be used from `ECLiPSe`.

3 Task 2: Shikaku

3.1 Introduction

A shikaku is a puzzle consisting of a recangular grid where some positions on the grid are filled by a number. The goal of the puzzle is to paritition the grid into rectangles where each rectangle contains exactly one number and its area is equal to the value of that number. This a a fairly easy task to do manually, but it quickly gets more complicated as the puzzle gets larger. We have attempted to find a source which calculated the computational complexity of Shikaku puzzles. Sadly, we could only find references to a single article [1]. This article claims that Shikaku puzzles are NP-Complete, but since we could not find a way to get a hold of the actual article itself we cannot verify its validity.



(a) The Shikaku puzzle... (b) ... and its solution

Figure 3: An example of a 7 by 7 Shikaku puzzle

3.2 Terminology

Because a Shikaku cannot easily be described as a matrix of values, we will introduce some terminology. To denote the value of number on the grid write $v(x, y)$, where x denotes the column of the grid and y denotes the row. Note that both coordinates start at 1. For instance: $v(1, 1) = 3$ and $v(4, 2) = 5$ in the Shikaku shown in figure 3.

The field itself has certain dimensions. All dimensions are denoted as a pair $s(w, h)$, where w is the width (the amount of columns) and h is the height (the amount of rows). In the running example, the dimensions of the field are $s(7, 7)$.

To uniquely define a rectangle we need both a position on the grid (the top left corner) and a size. Positions are denoted in a similar way as sizes: $c(x, y)$ represents the grid position shared by column x and row y. With this, we can define a rectangle like this: $rect(c(i, j), c(x, y), s(w, h))$. Here is $c(i, j)$ the number contained within the rectangle, $c(x, y)$ the top-left position of the rectangle and $s(w, h)$ the dimensions of the rectangle. We have decided to also put the position of the value contained by the rectangle in its definition to make the notation more analogous to our implemented code. As an example: $rect(c(7, 1), c(4, 1), s(4, 1))$ is the rectangle in the top right corner of the solved puzzle in Figure 3b.

While the following concept seems obvious from the drawings of the game, we will still include their formal definitions for the sake of completeness. The first important concept is $contains(c(x, y), s(w, h), c(i, j))$.

This implies that the grid position $c(i,j)$ lies inside the boundary of the rectangle $rect(p, c(x,y), s(w,h))$.

$$contains(c(x,y), s(w,h), c(i,j)) \iff i \in [x, x+w-1] \land j \in [y, y+h-1] \tag{1}$$

Another important concept is when two rectangles overlap. Two rectangles overlap when they share at least 1 grid position. It is expressed as $overlap(c(x_a,y_a), s(w_a,h_a), c(x_b,y_b), s(w_b,h_b))$. For $rect(p_a, c(x_a,y_a), s(w_a,h_a))$ and $rect(p_b, c(x_b,y_b), s(w_b,h_b))$ this means:

$$\begin{aligned} overlap(c(x_a,y_a), s(w_a,h_a), c(x_b,y_b), s(w_b,h_b)) \iff\ & x_a \le x_b + w_b - 1 \land x_b \le x_a + w_a - 1 \\ & \land y_a \le y_b + h_b - 1 \land y_b \le y_a + h_a - 1 \end{aligned} \tag{2}$$

3.3 Shikaku in ECLiPSe

3.3.1 Problem representation

For our `ECLiPSe` Shikaku representation we make heavy use of libraries provided by the platform. We use the rect structure provided by the `gfd library` to represent the rectangles. This rectangle structure has fields for an x and y coordinate, as well as a *width* and a *height*. The grid itself is not explicitely represented, it only comes forward in the constraints placed on the positions and sizes of the rectangles.

3.3.2 Solver outline

As a first step, the solver generates all of the variables needed to define the rectangles. *create_rectangles* reads all the input values one by one and accumulates the constructed rectangles. *rectangle* actually takes the dimensions of the grid as well as the coordinates and value of the number on the grid and builds the variables of the rectangle with their constraints. The constraints make sure that the rectangle is the right size, fits on the grid and contains the given value. Note that this rectangle representation is not the structure from the `gfd library`.

```
create_rectangles(_,_,[],_).
create_rectangles(Width,Height,[(X,Y,Val)|Tail],[R|Rects]) :-
          rectangle((X,Y,Val),Tail,Width,Height,R),
          create_rectangles(Width,Height,Tail,Rects).

rectangle((I,J,N),Others,Width,Height,rect(c(I,J),c(X,Y),s(W,H))):-
         X :: 1..Width,
         Y :: 1..Height,
         W :: 1..N,
         H :: 1..N,
         W*H #= N,
         X+W-1 #=< Width,
         Y+H-1 #=< Height,
          inside(c(X,Y),s(W,H),c(I,J)).
```

Once the list of rectangles in our notation has been generated, they are converted to the rect structure so we can use the *disjoint2* contstraint provided by the `gfd library`. The defining variables of all of the rectangles are then combined in a single flat list which is used to search for the solution.

```
rect_to_struct(Rects,Structs),
disjoint2(Structs),
(foreach(rect(X,Y,W,H,_),Structs), foreach([X,Y,W,H],List) do
          true
),
   flatten(List,FlatList)
```

All of these constraints are active constraints by the very nature of the `gfd library`. Whenever a new constraint is applied, it immediately impacts the domains of the variables this constraint impacts.

3.3.3 Search strategies

We have also experimented with search strategies for the Shikaku puzzle. Just like with the Sudoku puzzle, we have measured the time and amount of backtracks needed to solve each puzzle with several different search strategies. These strategies include `input order`, `first fail` and `anti first fail`. We have also implemented a search strategy we came up with to the speed up our CHR Shikaku solver. This strategy will select the largest rectangles first. A full explanation of the reasoning behind this search strategy can be found in Section 3.4.3 as the second additional optimization.

	ECLiPSe Input Order		ECLiPSe First Fail		ECLiPSe Anti First Fail		ECLiPSe Largest First	
	Time (s)	nbOfBt	Time (s)	nbOfBt	Time (s)	nbOfBt	Time (s)	nbOfBt
tiny	0.00	0	0.00	0	0.00	0	0.01	0
helmut	0.01	639	0.00	829	0.00	262	0.00	202
p(0,1)	0.00	5	0.01	8	0.01	46	0.00	17
p(0,2)	0.00	3	0.00	2	0.00	3	0.01	3
p(0,3)	0.00	8	0.00	6	0.00	4	0.00	4
p(0,4)	0.01	9	0.01	6	0.00	24	0.00	18
p(0,5)	0.00	4	0.00	5	0.00	48	0.00	6
p(1,1)	0.00	96	0.02	2910	0.00	71	0.00	133
p(1,2)	0.01	244	0.01	320	0.00	112	0.00	94
p(1,3)	0.00	273	0.03	5233	0.00	49	0.01	19
p(1,4)	0.00	23	0.01	308	0.01	70	0.00	57
p(1,5)	0.01	244	0.01	2721	0.00	406	0.01	760
p(2,1)	0.41	64 058	0.49	70 838	0.00	155	0.00	558
p(2,2)	0.04	4086	0.20	29 005	0.00	2368	0.03	1827
p(2,3)	0.01	257	1.77	241 490	0.03	306	0.00	58
p(2,4)	0.02	2251	3.34	628 514	0.01	168	0.00	82
p(2,5)	0.01	286	0.02	643	0.00	43	0.00	124
p(3,1)	0.16	14 322	TO	TO	0.04	1944	0.01	625
p(3,2)	0.01	94	TO	TO	0.01	73	0.00	158
p(3,3)	TO	TO	TO	TO	0.07	5526	0.14	9944
p(3,4)	0.34	42452	TO	TO	0.03	2910	0.15	14675
p(3,5)	0.15	15445	TO	TO	0.02	820	0.00	241
p(4,1)	TO	TO	TO	TO	5.58	282 333	1.22	54745
p(4,2)	0.91	91482	TO	TO	0.16	11 686	3.97	253 208
p(4,3)	TO	TO	TO	TO	0.01	773	0.49	55 545
p(4,4)	TO	TO	TO	TO	0.11	7098	0.37	30 352
p(4,5)	TO	TO	TO	TO	0.21	18 537	0.88	62 447
p(5,1)	TO	TO	TO	TO	7.90	307 116	217.67	8 457 170
p(5,2)	TO	TO	TO	TO	TO	TO	TO	TO
p(5,3)	TO	TO	TO	TO	TO	TO	TO	TO
p(5,4)	TO	TO	TO	TO	TO	TO	TO	TO
p(5,5)	TO	TO	TO	TO	TO	TO	TO	TO
p(6,1)	TO	TO	TO	TO	TO	TO	TO	TO

Table 3: The execution time and number of backtracks for all the puzzles with different search methods.

The first observation from these results is that `first fail` is by far the worst search strategy and `anti first fail` is the best search strategy. Our own largest first strategy ends up being slightly slower than `anti first fail`. This is a very strange result. First fail selects the variables with the smallest domain first to minimize the amount of backtracks but in this case it has the exact opposite effect.

We believe that the fact that largest first and anti first fail are so closely matched is an important clue as to why this happens. Variables belonging to larger rectangles have a larger domain than those belonging to small rectangles. The width and height can each generally be any of the divisors of the area of the rectangle. Larger numbers have generally more divisors so the domains of these variables are generally larger. The possible coordinates for the top left corner are directly coupled to the size of the rectangle. A larger rectangle can be positioned in more ways around the value than a smaller rectangle, making the domain for the coordinates of the top left corner larger.

This explains why largest first and anti first fail behave similarly, but not why they perform better. It is here that the insight given by figuring out why our CHR implementation was initially performing poorly helped us put these results into context. Our suspicion is that selecting larger rectangles limits the domains of other rectangles much more strongly because they are generally surrounded by more neighboring rectangles. Even though the large rectangles have a much larger domain themselves, the benefits of greatly reducing the domains of many other rectangles outweighs the downside of deeper backtracks as a result of selecting the larger domains first.

3.3.4 Adding a redundant constraint

We have also added a redundant constraint to our solver. This constraint explicitly prohibits rectangles from containing other values than the one for which they are defined. This can rule out many position/size combinations before the search begins, preventing unnecessary backtracks. Our implementation of this constraint was inspired by the experiments of H. Simonis [2]

	ECLiPSe Input Order		ECLiPSe First Fail		ECLiPSe Anti First Fail		ECLiPSe Largest First	
	Time (s)	nbOfBt	Time (s)	nbOfBt	Time (s)	nbOfBt	Time (s)	nbOfBt
tiny	0.00	0	0.00	0	0.00	0	0.01	0
helmut	0.01	364	0.00	281	0.00	28	0.00	171
p(0,1)	0.01	5	0.01	8	0.01	39	0.01	15
p(0,2)	0.01	3	0.00	2	0.00	3	0.00	3
p(0,3)	0.01	5	0.00	5	0.00	9	0.00	4
p(0,4)	0.00	5	0.00	6	0.00	15	0.00	14
p(0,5)	0.00	4	0.01	5	0.00	42	0.00	6
p(1,1)	0.01	44	0.01	1245	0.00	59	0.00	98
p(1,2)	0.01	53	0.01	110	0.00	65	0.00	44
p(1,3)	0.00	188	0.03	2308	0.00	40	0.01	18
p(1,4)	0.00	12	0.01	104	0.01	57	0.00	41
p(1,5)	0.01	62	0.02	1237	0.00	279	0.01	383
p(2,1)	0.09	6664	0.35	31 785	0.01	49	0.01	241
p(2,2)	0.02	1110	0.22	17 867	0.05	2051	0.03	1725
p(2,3)	0.02	156	1.39	100 663	0.01	119	0.01	32
p(2,4)	0.02	1056	7.01	570 537	0.01	128	0.00	57
p(2,5)	0.02	258	0.03	643	0.01	35	0.01	118
p(3,1)	0.04	1708	TO	TO	0.04	659	0.02	331
p(3,2)	0.01	51	TO	TO	0.01	64	0.01	92
p(3,3)	TO	TO	TO	TO	0.03	579	0.12	3753
p(3,4)	0.19	7143	TO	TO	0.04	1698	0.15	6343
p(3,5)	0.02	702	TO	TO	0.02	339	0.01	224
p(4,1)	TO	TO	TO	TO	7.34	118 200	1.06	20 003
p(4,2)	TO	TO	TO	TO	0.36	7418	3.32	91 431
p(4,3)	TO	TO	TO	TO	0.01	317	0.16	6991
p(4,4)	TO	TO	TO	TO	0.18	4893	0.54	20 606
p(4,5)	TO	TO	TO	TO	0.09	1988	0.28	7936
p(5,1)	TO	TO	TO	TO	0.96	15 655	54.4	957 812
p(5,2)	TO	TO	TO	TO	42.19	585 179	141.1	1 824 228
p(5,3)	TO	TO	TO	TO	TO	TO	4.44	116 811
p(5,4)	TO	TO	TO	TO	TO	TO	44.46	978 549
p(5,5)	TO	TO	TO	TO	TO	TO	TO	TO
p(6,1)	TO	TO	TO	TO	TO	TO	TO	TO

Table 4: The execution time and number of backtracks for all the puzzles with different search methods when the additional constraint is applied.

We see an almost unilateral improvement for the amount of backtracks among all search strategies, as expected. What we did not expect is that our largest first search method managed to solve two puzzles that the anti first fail was not able to solve. Although anti first fail is stil faster for most other puzzles. We believe that due to the extra constraint, the assumption that variables belonging to a larger rectangle usually have a larger domain no longer holds. It is hard to draw conclusions from this limited set of data. However, we suspect that because this assumption is likely no longer true, the benefit of reducing the domains of other rectangles may no longer outweigh increased amount of backtracks when solving large puzzles. More domain-reducing optimizations would be needed to fully investigate this hypothesis.

3.4 Shikaku in CHR

3.4.1 Problem representation

The CHR implementation of our Shikaku solver makes less use of libraries than our ECLiPSe implementation. The first step is generating what we call temporary rectangles. These are rectangles which have not yet been defined fully defined. The goal of the solver is to fully define all these temporary rectangles while respecting the given constraints. This is the solution for the puzzle. The temporary rectangles consist of a grid locations corresponding to the location of the value it contains, and a list of tuples containing a grid location and dimensions each. Each tuple represents a possible top left corner of the rectangle and its size.

Many of the constraints can be checked without having to know about other rectangles. This makes it possible to completely apply these constraints without any backtracking at all. We do this by checking against these constraints when generating the list of tuples for each temporary rectangle. The constraints taken into account when generating the domain of possible rectangles in the first place are: Each rectangle needs to have an area equal to the value it contains, each rectangle needs to actually contain the value it claims to contain in its definition, and each rectangle needs to fully fit within the field. Because these constraints are applied proactively, they are active constraints.

3.4.2 Solver outline

In the code below, `make_domains` iteratively calls `generate_options` for each of the values on the grid. With the resulting list of options, a `rect_temp` CHR constraint is generated. `generate_options` collects all the possible options generated by `generate_single_option` and returns them as a list. `generate_single_option` contains the constraints described above and will return all possible solutions satisfying those constraints one by one.

```
make_domains([],_) <=> true.
make_domains([(X,Y,Value)|Points],FieldSize) <=>
        generate_options(c(X,Y),Value,FieldSize,OptionsList),
        rect_temp(c(X,Y),OptionsList),
        make_domains(Points,FieldSize).

generate_options(Pos,Value,FieldSize,OptionsList):-
    findall(Option,
              generate_single_option(Pos,Value,FieldSize,Option),
              OptionsList)

generate_single_option(Point,Value,FieldSize,(Pos,Size)):-
        correct_size(Size,Value),
        contains(Pos,Size,Point),
        in_field(Pos,Size,FieldSize).
```

Once all these temporary rectangles are generated, only one more constraint needs to be checked: Rectangles may not overlap eachother.

```
rect(_,Pos,Size) \ rect_temp(Point,OptionsList) <=>
        remove_rect_overlap(Pos,Size,OptionsList,NewList),
        OptionsList \== NewList
        | rect_temp(Point,NewList).
```

```
failure @ rect_temp(_,[]) <=> fail.
```

This constraint removes all overlapping options from the list of options of every temporary rectangle. If at some point there is a temporary rectangle with no options left, the search will backtrack. This constraint is also an active constraint because it proactively limits the domains of all other temporary rectangles as soon as a permanent rectangle is added.

```
search, rect_temp(Point,OptionsList) <=>
        try_option(Point,OptionsList),
        search.
search <=> true.

try_option(_,[]) :- fail.
try_option(Point,[(Pos,Size)|Options]):-
    rect(Point,Pos,Size)
    ;
    try_option(Point,Options).
```

The search is very simple. It picks a temporary rectangle and tries every option in its list of options. This is done by constructing a permanent rectangle from the option and checking whether or not that leads to a failure. If a failure is found, backtracking happens and the next option is tried. In case that not a single option succeeds, another failure is caused which will backtrack the search to its last decision. Once a permanent rectangle is found that does not lead to a failure, the search starts over again with the next temporary rectangle.

3.4.3 Additional optimizations

First optimization Just like with out ECLiPSe implementation, one additional constraint is that a rectangle may not contain any values other than the one for which it is combined. This constraint is already provided by the combination of having each rectangle contain the value in its definition and not allowing rectangles to overlap. The critical part here is that the solver can decide whether ot not a rectangle contains other values without the need to know about other rectangles. This observation allows the solver to limit the domains of each temporary rectangle even more before and prevents a large amount of backtracks. The way we implemented this is very similar to how the overlap between rectangles is handled. Since this is constraint proactively limits the search domain, it is also an active constraint.

```
rect_temp(RefPoint,_) \ rect_temp(Point,OptionsList) <=>
        remove_point_overlap(RefPoint,OptionsList,NewList),
        OptionsList \== NewList
        | rect_temp(Point,NewList).
```

	CHR Basic solver		CHR Additional Constraint	
	Time (s)	inferences	Time (s)	inferences
tiny	0.00	3084	0.00	4431
helmut	0.07	517 641	0.04	182 333
p(0,1)	0.01	28 257	0.01	29 899
p(0,2)	0.01	33 808	0.01	31 088
p(0,3)	0.01	45 009	0.00	34 562
p(0,4)	0.00	46 032	0.00	35 782
p(0,5)	0.00	24 721	0.00	36 614
p(1,1)	0.02	228 824	0.02	87 952
p(1,2)	0.02	59 097	0.02	122 824
p(1,3)	0.01	65 448	0.02	83 102
p(1,4)	0.02	74 441	0.02	61 179
p(1,5)	0.02	190 380	0.01	62 018
p(2,1)	0.71	6 507 516	0.05	357 974
p(2,2)	2.75	25 407 954	0.10	758 057
p(2,3)	0.37	3 295 683	0.03	214 269
p(2,4)	0.62	5 881 840	0.05	346 635
p(2,5)	0.94	8 888 968	0.04	224 674
p(3,1)	TO	TO	0.38	3 458 752
p(3,2)	0.25	2 234 784	0.12	1 067 363
p(3,3)	TO	TO	0.25	2 140 242
p(3,4)	19.56	185 782 112	0.08	650 691
p(3,5)	TO	TO	0.59	5 452 485
p(4,1)	TO	TO	TO	TO
p(4,2)	TO	TO	TO	TO
p(4,3)	TO	TO	TO	TO
p(4,4)	TO	TO	4.99	46 192 343
p(4,5)	TO	TO	1.29	11 824 321
p(5,1)	TO	TO	TO	TO
p(5,2)	TO	TO	33.47	310 379 885
p(5,3)	TO	TO	TO	TO
p(5,4)	TO	TO	1.70	14 858 545
p(5,5)	TO	TO	TO	TO
p(6,1)	TO	TO	TO	TO

Table 5: The execution time and number of inferences of the basic solver and the solver with an additional constraint

Just like in our other implementation, the additional redundant constraint as a positive effect on performance. Where the basic solver struggles to solve the 15 by 15 puzzles, the improved solver solves all of the 15 by 15 puzzles and even managed to solve some of the 20 by 20 and 25 by 25 puzzles. The constraint clearly has significantly reduced the search space.

Second optimization Our second optimization is not a constraint, but instead it is a smarter search heuristic. Our first approach to searching did not define an order in which the temporary rectangles should be searched. During testing we observed that a small rectangle (for instance size 2) placed the wrong way early into the search process could make the solver take a really long time trying to fit all the other rectangles. The solver would often fail really deep into the search and have to backtrack all the way to the start.
Another observation was the fact that the really large rectangles in the larger puzzles only had relatively few options compared to their size thanks to our previous optimization. Most options would get filtered out right away because they would contain other values. These large rectangles also had a very large effect on the domains of other temporary rectangles. A rectangle of size 2 can only affect at least one temporary rectangle, but a rectangle of size 30 or higher will have many neighbors. Our reasoning was that by trying to place the large rectangles first, the solver could significantly reduce the search domain going forward and limit the depth at which backtracking usually occurs.

```
 search(N), rect_temp(Point, OptionsList) <=>
        member((_, s(Width, Height)), OptionsList),
        Width*Height >= N
        | try_option(Point, OptionsList),
        search(N).
search(0) <=> true.
search(N) <=> N1 is N - 1, search(N1).
```

The search constraint should be initialized with a value at least as large as the largest possible rectangle. For simiplicity we simply initialize it with the total area of the grid, but looking for the highest value on the grid will also work. In each search step, it will try to place rectangles of at least the current value. If no temporary rectangles of sufficient size are matched, the value for the search is decremented repeatedly until the next match.

We can immediately see in Table 6 that for the first time one of our Shikaku solvers manages to solve the largest puzzle. At 30 by 40 this puzzle is now just barely solvable using the new search heuristic. Knowing this, it is no surprise that we can see significant improvements across the board when compared to the basic solver. We believe that selecting the large rectangles first significantly reduces the search domain of many other rectangles, just like in our ECLiPSe implementation.
When combining those two optimizations, the solver becomes even faster. We expected some gains in performance but we were surprised by just how much faster it became. Even the hardest puzzles in the data set take no longer than a few second to solve. We also do not see any of the large peaks in the number of inferences required to solve certain puzzles. All puzzles of the same size now take roughly the same amount of inferences to solve.

	CHR Basic Solver		CHR Largest First		CHR Combined	
	Time (s)	inferences	Time (s)	inferences	Time (s)	inferences
tiny	0.00	3084	0.00	3328	0.00	3660
helmut	0.07	517 641	0.02	133 198	0.03	159 904
p(0,1)	0.01	28 257	0.01	29 134	0.01	25 657
p(0,2)	0.01	33 808	0.01	40 705	0.01	26 494
p(0,3)	0.01	45 009	0.01	34 550	0.01	30 800
p(0,4)	0.00	46 032	0.01	29 512	0.01	31 178
p(0,5)	0.00	24 721	0.01	36 639	0.00	32 022
p(1,1)	0.02	228 824	0.02	18 331	0.02	76 854
p(1,2)	0.02	59 097	0.02	119 194	0.02	112 199
p(1,3)	0.01	65 448	0.02	108 673	0.01	69 607
p(1,4)	0.02	74 441	0.02	76 805	0.01	67 014
p(1,5)	0.02	190 380	0.01	73 670	0.01	63 522
p(2,1)	0.71	6 507 516	0.15	1 392 238	0.08	612 939
p(2,2)	2.75	25 407 954	0.08	699 178	0.04	262 105
p(2,3)	0.37	3 295 683	0.04	234 719	0.02	200 600
p(2,4)	0.62	5 881 840	0.04	300 085	0.04	233 985
p(2,5)	0.94	8 888 968	0.04	263 053	0.03	225 389
p(3,1)	TO	TO	0.28	2 504 518	0.09	677 965
p(3,2)	0.25	2 234 784	0.08	565 555	0.06	519 563
p(3,3)	TO	TO	0.13	1 085 122	0.11	893 334
p(3,4)	19.56	185 782 112	0.28	2 519 750	0.08	653 212
p(3,5)	TO	TO	0.24	2 306 833	0.10	735 528
p(4,1)	TO	TO	0.78	7 359 453	0.35	3 055 868
p(4,2)	TO	TO	0.29	2 587 162	0.21	1 784 959
p(4,3)	TO	TO	0.34	2 972 902	0.15	1 351 991
p(4,4)	TO	TO	1.03	10 035 411	0.28	2 593 443
p(4,5)	TO	TO	0.34	3 143 284	0.20	1 696 816
p(5,1)	TO	TO	12.79	123 529 917	0.59	5 084 746
p(5,2)	TO	TO	95.54	898 562 921	1.09	7 776 059
p(5,3)	TO	TO	2.93	26 276 870	0.41	3 511 457
p(5,4)	TO	TO	7.61	75 856 587	1.33	12 134 475
p(5,5)	TO	TO	44.24	437 154 752	3.65	35 034 535
p(6,1)	TO	TO	163.78	1 556 158 403	4.05	37 371 197

Table 6: The execution time and number of inferences of the basic solver, the solver with an improved search selection order and the final solver with both optimizations combined.

4 Conclusions

Our experiments have shown that the viewpoint, search method and redundant constraints are critical factors which determine how quickly a solution can be found. We have also found that it was almost trivial to solve Sudokus using ECLiPSe while it took a lot of effort to make sure our CHR implementation could solve all puzzles in a reasonable amount of time. Even after a lot of experimentation, our fastest CHR Sudoku is still an order of magnitude slower than our ECLiPSe Sudoku solver.

When solving Shikakus we saw the opposite: We never managed to solve all puzzles in our ECLiPSe Shikaku solver, but with just a few easy optimizations the CHR solver can solve the entire data set in less than 20 seconds. Obviously we wondered why these two seemingly similar types of puzzles end up being so different to implement and optimize in ECLiPSe and CHR.

A first important influence is the quality of our code. Given more time, we are confident we can improve the performance of all of our solvers and probably close the gap between them slightly. However, given that the implementations between the languages use similar viewpoints, search methods and constraints this is probably not the only cause.
What we believe is more likely is that the different languages are better suited for different types of problems. ECLiPSe has libraries specialized in propagating constraints and searching over integer domains. Sudoku puzzles are a perfect match to solve using those specialized libraries. In our CHR implementation we ended up attempting to immitate the way constraints are propagated in the ic library. It is just extremely hard to capture all the advanced techniques and optimizations found in those libraries.

The opposite is true when solving Shikaku puzzles. There is no ECLiPSe library that can handle constraints on domains of arbitrary predicates. While it is certainly possible to make the Shikaku solver work in ECLiPSe, the libraries are simply not suited for problems where constraints apply to more complex structures. The freedom that made it hard for our Sudoku CHR solver to compete with its ECLiPSe counterpart actually made it very easy to manually construct and update the domains of the rectangles. It is possible to also do this work manually in the ECLiPSe solver, but added value of working within an ECLiPSe environment is greatly diminished.

References

[1] Y. Takenaga, "Shikaku and ripple effect are np-complete," *Congressus Numerantium*, no. 216, pp. 119–128, 2013.

[2] H. Simonis, "Shikaku as a constraint problem," *CSCLP*, 2009.