APLAI Assignment 2010-2011

Karl Dox Li Quan Nathan Vandecauter

June 1, 2011

Contents

1 Sudoku

1.1 Discussion

1.1.1 ECLiPSe

Sudoku is a perfect example for which constraint programming is very easy and efficient to use. This arises automatically from the formulation of the Sudoku problem: each unit (a row, a column and a block) contains a permutation of 1–9. It is very easy to code this using the alldifferent constraint from the ic library (or ic_global, see experiments) in ECLiPSe.

1.1.2 CHR

CHR uses only one rule per time: it will try to fire the first rules first. We can use this property to make sure that the system first propagates as much as possible values prior to making guesses about possible values of certain locations. Using guards will help test if a rule is valid or not. We expect that the performance of these custom-made propagations to be less efficient than the constraint propagation of the libraries included with ECLiPSe.

1.1.3 JESS

JESS is a rule-based system, so naturally, Sudoku should be easily solvable using this system. At first view, Jess does not seem to have disadvantages: normal propagation rules have to be written. If nothing can be propagated, make a guess and assign a possible value at a certain location.

It is however not clear when which rule will be used. Depth-search would activate the most recently used rules. What rules will be used to begin? A random allocation or a propagation? Assume we start with propagation until we can't propagate anything anymore, we have to choose a random value to fill in at a random allocation, of course without breaking the Sudoku-rules. How do we know that after having made a random choice, we will continue to propagate?

Breadth-search would invoke rule by rule. Logically, this is not a good choice, since it will sometimes make a random choice when it's not needed. The breadth-first search also consumes a lot of additional memory.

Of course, we could use these strategies, since both would come up with a solution eventually, but this would require a lot of memory and time. Possibly, we could even run out of working memory.

As a result, we would have to write our own strategy. However, because our lack of experience with Jess, this does not seem an easy task.

Another feature in JESS is the ability to use Java. This means we could let Java handle the input. Changing the Sudoku in Java could fire events, making sure certain rules fire. This would be a work-around for the previous problem: only propagation rules

are written in Jess. If Jess can't find any more useful rules, Java would make a random choice.

This poses a new problem: if Java makes a wrong guess, we would have to backtrack. As fas as we know, we would have to implement that ourselves in Java. Even though it would not be difficult, since we are working in a very narrow domain (whenever we have to make a choice, we make a 'snapshot' of the Sudoku, this would amount to extra amount of code. Thus, this approach might be more complex than writing our own strategy.

In conclusion, we decided not to solve Sudoku in JESS because we have the least experience with it and there are many decisions to be made that can have a great impact on performace.

1.1.4 Alternatives

Of course, very naive approaches also exist, e.g. brute force, or a simple backtracking search (in for instance Matlab, Python, Java, ...), but are of course very inefficient. More interestingly, local search and meta-heuristics such as simulated annealing and tabu search have been fairly successfully used to solve Sudoku's [6]; also SAT solvers have been used to solve Sudoku's [7]. Integer programming models for Sudoku have also been studied, with relatively good performance [1]. The Wikipedia article on Sudoku [12] gives a good overview of different techniques.

1.2 Viewpoints and programs

1.2.1 Standard viewpoint

Of course, the standard viewpoint (which follow directly from the Sudoku definition and already given in the introduction) is the most natural and easy way to define the constraints.

Some alternative viewpoints and a discussion of constraint programming for Sudoku's in general are shown in [9]. The criteria to judge whether a viewpoint is a good one or not, is based on efficiency and readability (complexity in terms of program structure). Other qualities would be scalability and modifiability in general. A simple metric for performance could be time or number of backtracks. Ideally, one would like a search-free way of solving Sudoku's i.e., without guessing [9].

1.2.2 Alternative viewpoint

First, we can of course consider the trivial viewpoints such as rows versus columns, but these would not make a difference because of the symmetry of the Sudoku puzzle.

We consider the dual viewpoint of Sudoku (which is also described in [8]). We use 9 variables $N_1, \ldots, N_9$ which respectively represent the numbers $1, \ldots, 9$, and the domain of every number $N_i \in \mathcal{P}(\{1, \ldots 81\})$, i.e. the cells a number occurs; an assignment $N_i = \{c_1, \ldots, c_n\}$ means that number i occurs in cell $c_1, \ldots, c_n$, where $\{c_1 \ldots, c_n\} \subset \{1, \ldots, 81\}$.

In this viewpoint, the constraints of a Sudoku are more "clumsy" to express.

- the cardinality of every set representing a number is 9;
- all those sets are disjoint;
- every number has exactly one value in every row and every column and exactly one value in every 3×3 block.

The channeling constraints are of the following form: $\forall i, j : i \in N_j \Leftrightarrow Sudoku[i] = j$.

1.3 Experiments

1.3.1 ECLiPSe

These experiments were done on a Ubuntu 64-bit laptop with an Intel Core i3 processor and 4 GB RAM memory, with ECLiPSe Version 6.0 #169 (x86_64_linux).

Normal viewpoint Table 1 shows the given Sudoku puzzles using the normal viewpoint in ECLiPSe. (All given puzzles were well-formed, i.e. there existed only one solution per puzzle.) We also added 2 extra Sudoku's (Figure 2) so we could compare with the given puzzles. These 2 puzzles (*diabolical* and *platinum*) were considered difficult, as rated by some program to predict the difficulty of a puzzle.

It is clear that this is a very good approach: all puzzles took less than one second to solve. Moreover, we see that the first-fail ordering of variables is always better (the number of backtracks is in almost all cases—sometimes significantly—reduced), except for the *goldennugget* puzzle.

The implementation was done using the alldifferent constraint from the ic_global library. We also used the version of the standard ic library, but noticed no difference. This might be because the problem size is too small for the more global reasoning of the constraints[1] to have effect, but we have not been able to test this to confirm.

Clearly, the *lambda* Sudoku is the most difficult puzzle for our solver. (Extra3 is apparantly the exact same puzzle.) If we inspect the puzzle (Figure 1) more carefully, we suspect that this is a puzzle where very few hints are given, which makes the propagation of constraints more difficult. This can be confirmed using results of the *extra* puzzles, which are basically the same puzzles with more hints.

Interestingly, our own two examples were not at all considered difficult by our solver (we also tried to solve them manually, and we experienced them to be difficult).

[1] From the ECLiPSe tutorial:

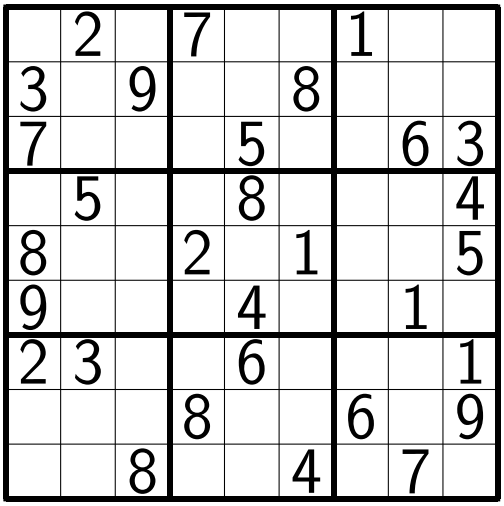
> The constraints have the same declarative semantics (what they do, conceptually), but different operational semantics (how they do it, in practice). Those in ic_global perform more computation, but achieve more propagation. In simple terms, the propagation in the ic_global takes a more "global" view of the constraint by reasoning over several variables at the same time, rather than just pairs of variables. For example, all_different in ic is decomposed into #\= constraints between all possible pairs of variables, whereas the ic considers the variables in the constraint together.

	input order		first-fail	
puzzle	run time (s)	backtracks	run time (s)	backtracks
medium	0.00	0	0.00	0
difficult	0.00	0	0.00	0
verydifficult	0.00	0	0.00	0
expert	0.01	5	0.00	4
lambda	0.92	10 786	0.13	773
hard17	0.14	1383	0.07	476
symme	0.09	548	0.06	377
eastermonster	0.12	648	0.09	424
tarek_052	0.01	53	0.01	5
goldennugget	0.05	295	0.08	354
coloin	0.12	763	0.13	750
hardest	0.12	648	0.08	424
extra1	0.29	4229	0.11	871
extra2	0.42	4236	0.27	1454
extra3	0.91	10 786	0.13	773
extra4	0.25	2856	0.14	1074
diabolical	0.01	1	0.00	1
platinum	0.04	260	0.01	36

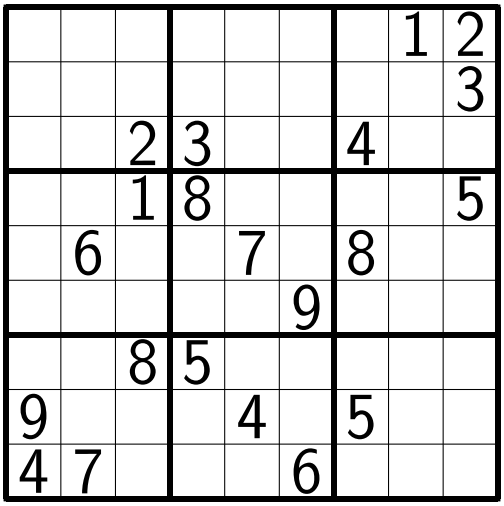
Table 1: Experiments standard viewpoint Sudoku ECLiPSe.

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

Figure 1: *lambda* Sudoku.



(a) *diabolical* [10]



(b) *platinum* [11]

Figure 2: Extra Sudoku's.

Alternative viewpoint The results of the alternative viewpoint in ECLiPSe is given in Table 2. Using the ic_sets library the first two constraints are fairly easy to write, but the last one is more troublesome. It can be done by constructing for-loops and making calculations on the cell id's, but this can be fairly difficult to read. Therefore, we decided to "hardcode" these last constraints for a 9×9 Sudoku. This makes the program less portable, but greatly improves the readability.

Surprisingly, this viewpoint does not perform significantly worse than the normal viewpoint (for instance the times for the first 3 puzzles are only slightly larger). For some puzzles, it even performs significantly better (for instance, the *extra* puzzles). Apparently, this form of reasoning is better for those puzzles.

name	time (s)
medium	0.04
difficult	0.02
verydifficult	0.03
expert	0.04
lambda	0.04
hard17	0.04
symme	0.38
eastermonster	0.14
tarek_052	0.05
goldennugget	0.89
coloin	2.08
hardest	0.14
extra1	0.04
extra2	0.05
extra3	0.03
extra4	0.03
diabolical	0.04
platinum	0.86

Table 2: Experiments alternative viewpoint Sudoku ECLiPSe.

1.3.2 CHR

The following experiments were done on the same laptop running SWI-Prolog (Multi-threaded, 64 bits, Version 5.8.2).

Normal viewpoint The classic Sudoku in CHR (based on `http://dtai.cs.kuleuven.be/CHR/summerschool/contest/sudoku.chr`, but fixed a few bugs). It should be clear that the performance is comparable to the standard viewpoint using the first-fail heuristic in ECLiPSe.

puzzle	run time (s)
medium	0.07
difficult	0.06
verydifficult	0.07
expert	0.07
lambda	0.11
hard17	0.10
symme	0.09
eastermonster	0.09
tarek_052	0.09
goldennugget	0.08
coloin	0.09
hardest	0.08
extra1	0.11
extra2	0.10
extra3	0.11
extra4	0.11
diabolical	0.06
platinum	0.10

Table 3: Experiments classic Sudoku CHR.

Contrary to the solvers in ECLiPSe, the *platinum* puzzle was considered difficult by this solver.

Alternative viewpoint In CHR, this gives rise to making a cell per value: the cell would have 2 parameters: the value $1 \ldots 9$ and the list of possible values `V` (of length 9). To specify which locations could have a certain value, but we are not yet certain of, we use the 'maybe' predicate with arity 3. We define `maybe(X,Vals,LL)`, where `X` is the value $1 \ldots 9$, `Vals` the possible locations and `length(Vals,LL)`.

First, we need to convert the input, which still uses the classical viewpoint to our alternative viewpoint.

```
sudoku(Input,Output) :-
```

```
    numlist(1,81,D),
    forCell(1,9,Output),
    make_vals(Input,1,D),!.
```

In this predicate, Input is, of course, the Sudoku to solve. It contains the Sudoku according to the classical viewpoint. Output is the alternative viewpoint. Even though we don't need the output as a return, it's interesting to be able to take a look at it during program execution.

The `forCell` predicate creates the 9 cells and add the location-list to Output. It also creates the initial maybe's.

```
forCell(Max,Max,[Cell]) :- createCell(Max,Cell).
forCell(Cur,Max,[H|T]) :-
  Curn is Cur+1,
  forCell(Curn,Max,T),
  createCell(Cur,H).

createCell(Cell,D) :-
  length(D,9),
  cell(Cell,D),
  numlist(1,81,Maybe),
  maybe(Cell,Maybe,81).
```

After the creation of the cells, the input is finally converted to the classical viewpoint using the `make_vals/3` predicate:

```
make_vals([],_,_).

make_vals([[]|L],Count,D) :-
  make_vals(L,Count,D).
make_vals([[E|Es]|L],Count,D) :-
  (   var(E) ->
    NewD = D
  ;
    assignLocation(E,Count),
    select(Count,D,NewD)
  ),
  NewCount is Count+1,
  make_vals([Es|L],NewCount,NewD).
```

Note the 'assignLocation'. By assigning an index in the location array, we can improve performance. If a location/value coupling does not seem to be correct, the program might backtrack and assign the location to the same value again, but on a different index. The program would be forced to make unnecessary operations again.

```
assignLocation(Cell,Value) :-
  cell(Cell,Vals),
  X is (Value+8)//9,
  nth1(X,Vals,Value).
```

This predicate makes sure that locations on the same row always have the same index. Afterwards, we can start solving the sudoku:

```
solve :- (solved ->  true ; choice,solve).
```

This is also where the output in `sudoku/2` comes in handy: we can print it out between different steps:

```
solve(P) :- get_single_char(_),writeOutput(P),(solved -> true; choice,
    solve(P)).
```

First, we should discuss the normal propagation rules.

```
rememberCell @ cell(N,Vals) \ cell(N,Vals2) <=> Vals=Vals2.
```

This rule is self-explanatory: we want to remember our cells.

```
cell(X,V) \ maybe(X,[L],_) <=> getGrounded(V,O),length(O,8)|
    assignLocation(X,L).
cell(X,V) \ maybe(X,_,_) <=> getGrounded(V,O),length(O,9) | true.
```

This two rules belong together. If a maybe has only one possibility left, we will check the assigned locations of the corresponding cell. (getGrounded(V,O): O contains grounded items in V). If the length of O is 8, we will assign that last value to X and remove that maybe.

On the other hand, if all locations for a certain cell are 'known', the corresponding maybe should be removed. is slightly more complicated. If a 'maybe' has only one possibility left, we will check the assigned locations of the corresponding cell. (getGrounded(V,O): O contains grounded items in V). If the length is of O is 8, we will assign that final

```
maybe(_,_,0) <=> fail.
```

If any maybe is found with zero possible locations, we assume we made an error earlier. (Normally, by the time we arrive at this rule, any cells that are completely filled in, should have their maybe's removed).

```
cell(X,V) \ maybe(X,L,LL) <=>
        member(Elem,V),nonvar(Elem),
  member(Diff,L),
  sees(Elem,Diff), select(Diff,L,R) |
  RL is LL-1, maybe(X,R,RL).
```

If Elem is a set location for X, then every location L that is seen by Elem, should be removed from that maybe. The old maybe is removed and an updated maybe is set.

sees/2 and relevant predicates are shown below:

```
sees(X,Y) :-
  toXY(X,X1,Y1),
  toXY(Y,X2,Y2),
  sees(X1,Y1,X2,Y2).

sees(X,_,X,_) :- !,true.          % same row
sees(_,Y,_,Y) :- !,true.          % same column
sees(X,Y,A,B) :-        % same box
  (X-1)//3 =:= (A-1)//3,
```

```
        (Y-1)//3 =:= (B-1)//3.

toXY(Val,X,Y) :-
  X is (Val+8)//9,
  Y is Val-((Val-1)//9)*9.
```

The second rule:

```
cell(_,V) \ maybe(A,L,LL) <=> %If a location 'j'already has a value, it
    can't be set anywhere else.
  member(Elem,V),nonvar(Elem),
  select(Elem,L,R) |
  RL is LL-1, maybe(A,R,RL).
```

If Elem is a set location for any cell, then that location cannot be a 'possible' location for every other value. Therefore, all maybe's are updated accordingly.

These are the set propagation rules. If these are exhausted, we have to guess. This is done by the 'solve' predicate shown above. We will repeat it here:

```
solve :- (solved ->  true ; choice,solve).
choice :- ff(1).
```

If the sudoku is not yet solved (it is solved when there are no more maybe's), it will make a 'choice', and then solve it again. 'choice' will set ff/1 to 1. This makes that 2 possible rules are fired:

```
cell(X,_) \ ff(LL), maybe(X,L,LL) <=>
   select(D,L,R),assignLocation(X,D)|
   RL is LL-1,maybe(X,R,RL).

ff(LL) <=> LL1 is LL+1, ff(LL1).
```

The first, will, if there exists a maybe with a LL number of possible locations, select a possible location from that list and see if it can be assigned to the corresponding value. If so, update the maybe.

If that rule cannot be fired, ff(LL) will be increased to ff(LL+1).

Finally, we have to know that the Sudoku is solved.

```
maybe(_,_,_) \ solved <=> fail.
solved <=> true.
```

If there exists a maybe, it can not be solved. We still are not of all locations. Otherwise, if there is no doubt, the puzzle is solved.

This rule cleans up afterwards:

```
cleanup \ cell(_,_) <=> true.
```

This is the code for getGrounded/2:

```
getGrounded([],[]).
getGrounded([H|T],Tmp) :-
  (   nonvar(H) ->
      [H1|Out] = Tmp,
```

```
      H1=H,
      NO = Out
  ;
  NO = Tmp
  ),
  getGrounded(T,NO).
```

In order to convert the solution back to the classical viewpoint, we use the consider/2 predicate, which is invoked by consider(Input,1).

```
consider(_,10).

consider(Input,Val) :-
  ValN is Val+1,
  consider(Input,ValN),
  cell(Val,Locs),
  getGrounded(Locs,Vals),
  consider(Input,Vals,1,Val).

consider([],_,_,_).
consider(_,[],_,_).
consider([[]|Rest],T,Count,Value) :-
  consider(Rest,T,Count,Value).
consider([[Es|E]|Rest],[H|T],H,Value) :-
  Es=Value,
  NewC is H+1,
  consider([E|Rest],T,NewC,Value).

consider([[_|Tail]|Rest],[H|T],Count,Value) :-
  Count \= H,!,
  NewC is Count+1,
  consider([Tail|Rest],[H|T],NewC,Value).
```

To test this program, we also wrote a very easy Sudoku, where only a few locations need to be filled in:

```
tooeasy(P) :- P =
  [[1,2,3,4,5,6,7,8,9],
   [4,5,6,7,8,_,1,2,3],
   [7,8,9,1,2,3,4,5,6],
   [2,3,4,5,_,7,8,9,1],
   [5,6,7,8,9,_,2,3,4],
   [8,9,1,2,3,4,5,6,7],
   [3,4,5,_,7,8,9,1,2],
   [6,7,8,9,1,2,3,4,5],
   [9,1,2,3,4,5,6,7,_]].
```

To solve this puzzle, we use the following code:

```
?- tooeasy(P), sudoku(P,O), solve, consider(P,1) ,writeln('Input: '),
    writeOutput(P), writeln('Viewpoint view: '), writeOutput(O).
```

This gives the following output:

```
Input:
[1,2,3,4,5,6,7,8,9]
[4,5,6,7,8,9,1,2,3]
[7,8,9,1,2,3,4,5,6]
[2,3,4,5,6,7,8,9,1]
[5,6,7,8,9,1,2,3,4]
[8,9,1,2,3,4,5,6,7]
[3,4,5,6,7,8,9,1,2]
[6,7,8,9,1,2,3,4,5]
[9,1,2,3,4,5,6,7,8]
Viewpoint view:
[1,16,22,36,42,48,62,68,74]
[2,17,23,28,43,49,63,69,75]
[3,18,24,29,44,50,55,70,76]
[4,10,25,30,45,51,56,71,77]
[5,11,26,31,37,52,57,72,78]
[6,12,27,32,38,53,58,64,79]
[7,13,19,33,39,54,59,65,80]
[8,14,20,34,40,46,60,66,81]
[9,15,21,35,41,47,61,67,73]
cell(1,[1,16,22,36,42,48,62,68,74])
cell(2,[2,17,23,28,43,49,63,69,75])
cell(3,[3,18,24,29,44,50,55,70,76])
cell(4,[4,10,25,30,45,51,56,71,77])
cell(5,[5,11,26,31,37,52,57,72,78])
cell(6,[6,12,27,32,38,53,58,64,79])
cell(7,[7,13,19,33,39,54,59,65,80])
cell(8,[8,14,20,34,40,46,60,66,81])
cell(9,[9,15,21,35,41,47,61,67,73])
P = [[1, 2, 3, 4, 5, 6, 7, 8|...], [4, 5, 6, 7, 8, 9, 1|...], [7, 8, 9, 1, 2, 3|...],
[2, 3, 4, 5, 6|...], [5, 6, 7, 8|...], [8, 9, 1|...], [3, 4|...], [6|...], [...|...]],

O = [[1, 16, 22, 36, 42, 48, 62, 68|...], [2, 17, 23, 28, 43, 49, 63|...],
[3, 18, 24, 29, 44, 50|...], [4, 10, 25, 30, 45|...], [5, 11, 26, 31|...],
[6, 12, 27|...], [7, 13|...], [8|...], [...|...]]
```

The viewpoint view gives per row (as a value) the corresponding values, e.g.: row 1 shows the locations of value 1, row 2 shows the locations of value 2, ...

Unfortunately, the given Sudoku's appear not to be solvable by this program. At the time of writing, the reason why has not yet been discovered. (We also provide a different way of implementing this viewpoint, see the readme file.)

1.3.3 Conclusion

All the programs (except the alternative viewpoint in CHR) are working efficiently for almost all puzzles. The experiments show that the different viewpoints can have an impact on the propagation behavior and thus also the performance. These effects are however also depending on the puzzle itself considered. While all programs were overall equally performant, the standard viewpoint in ECLiPSe was the most natural and easy way to solve the Sudoku problem, as anticipated at the start. Moreover, the first-fail heuristic is the best approach (intuitively, this is also the way most Sudoku's are manually solved).

2 Maximum Density Still Life

2.1 Discussion

2.1.1 ECLiPSe

The formulation of the problem is as explained in [2, 3] almost a trivial exercise in constraint programming. The main constraints of the problem are shown below (additionally, we have to add constraints such that the border cells are dead and stay dead):

```
% current cell and its neighbours
Nbs #= ( Board[X-1,Y-1]  + Board[X,Y-1] + Board[X+1,Y-1] +
   Board[X-1,Y]    +        Board[X+1,Y]   +
   Board[X-1,Y+1]  + Board[X,Y+1] + Board[X+1,Y+1] ),
% live cell must be kept alive
Board[X,Y] => ( Nbs #>= 2 and Nbs #=<3 ),
% dead cell must stay dead
neg(Board[X,Y]) => Nbs #\= 3
```

This Still Life can easily be used for the MDSL solver by maximizing the number of cells that are alive (or equivalently, minimizing the cells that are dead).

The symmetry breaking in [2, 3] can also be easily written. Additionally, the coupling with external solvers for the linear programming (using the eplex library) allows to use a hybrid solver, which should perform considerably better.

2.1.2 CHR

We used the given starting program skeleton in CHR. While the complete program is also very readable, the approach taken does require a complete different way of thinking. Also in general, the symmetry breaking constraints seem much more difficult to write, as does the coupling with an linear programming interface.

2.1.3 JESS

In essence, the same solution as for Sudoku could be used: Java for input and "guessing"/backtracking; and JESS would make use of rules to propagate as much as possible. Many libraries exist for Java for optimization so a combination with a linear programming package would be possible, but the full integration with the rule handling would not be so trivial.

2.1.4 Alternatives

The MDSL problem has been studied intensively and various specialized optimizations have been proposed. Various relaxations are discussed in [5]. The use of Binary Decision Diagrams (BDD) have been discussed in [4]

2.2 Experiments

Experiments in ECLiPSe for MDSL were performed on the PC's of the computer labo's; for CHR again the same configuration as above.

2.2.1 ECLiPSe

The experiments with (a subset of) the different approaches as discussed in [3] are shown in Table 4. 'CP' is the standard constraint programming technique, 'CP (symm)' breaks the symmetry by considering only board where the density of the left halve is at least that of the right halve[2] (this is an ad-hoc approach); 'CP (symm2)' uses the symmetry breaking discussed at the end of the paper (i.e., symmetry breaking by selective ordering[3]). Finally, the hybrid 'CP/IP' solver is also tested, first without symmetry constraints and then with the second symmetry breaking approach (for the hybrid approach we only added the death by overcrowding constraint as this gave the best results in the paper).

n	CP	CP (symm)	CP (symm2)	CP/IP	CP/IP (symm2)
2	0.00	0.00	0.00	0.00	0.00
3	0.00	0.00	0.00	0.00	0.00
4	0.02	0.03	0.01	0.01	0.01
5	0.01	0.01	0.00	0.01	0.00
6	1.49	1.55	1.34	1.51	1.50
7	11.77	13.06	10.56	14.06	13.83
8	259.60	260.03	236.26	272.45	258.30

Table 4: Experiments MDSL in ECLiPSe. Times shown are in seconds.

The problem with sizes $n \geq 9$ took more than 15 minutes for all solvers. While the pure CP approaches are perfectly in line with the timings shown in the paper [3], the CP/IP hybrid approaches do not seem to improve drastically; they even perform (albeit slightly) worse for some instances. We also tried both the death by overcrowding and the death by isolation together, but these performed even more drastically worse (for $n = 8$, it took almost 400 s).

We don't have the full explanation, but it might be that the linear programming solver used cannot "communicate" efficiently with the CP solver, either because of wrong settings or because of the implementation (we also observed that these timings differed greatly from one system te another, so in general, it is quite difficult to point out the exact problem). It might be that the software used in the paper (OPLStudio) supports a better hybrid search, or that the authors of the paper also used specific extra heuristics/optimizations which were not mentioned.

[2] Likewise this can be done for the upper and lower halve, either as an extra constraint or independently. We only tested the lower versus right density constraint.

[3] The assignment of cells is forced to respect an ordering on the values that occur in corner entries of the board.

2.2.2 CHR

We'll just show the traces of the CHR output as these should be self-explanatory.

```
?- time(mdsl(2)).
trying_label(4)
0000
0110
0110
0000
% 35,617 inferences, 0.010 CPU in 0.014 seconds (71% CPU, 3561700 Lips)
label(4)
true.

?- time(mdsl(3)).
trying_label(6)
00000
01100
01010
00110
00000
% 87,504 inferences, 0.020 CPU in 0.024 seconds (82% CPU, 4375200 Lips)
label(6)
true.

?- time(mdsl(4)).
trying_label(11)
trying_label(10)
trying_label(9)
trying_label(8)
000000
011000
010100
001010
000110
000000
% 13,781,406 inferences, 2.290 CPU in 2.298 seconds (100% CPU, 6018081 Lips)
label(8)
true.

?- time(mdsl(5)).
trying_label(16)
0000000
0110110
0110110
```

```
0000000
0110110
0110110
0000000
% 289,178 inferences, 0.050 CPU in 0.060 seconds (83% CPU, 5783560 Lips)
label(16)
true.

?- time(mdsl(6)).
trying_label(22)
trying_label(21)
trying_label(20)
trying_label(19)
trying_label(18)
00000000
01100000
01010110
00110110
00000000
00110110
00110110
00000000
% 937,169,006 inferences, 184.612 CPU in 186.195 seconds (99% CPU, 5076437 Lips)
label(18)
true.

?- time(mdsl(7)).
trying_label(29)
trying_label(28)
000000000
011011010
001010110
010010000
011101110
000010010
011010100
010110110
000000000
% 2,931,062,192 inferences, 574.505 CPU in 581.200 seconds (99% CPU, 5101893 Lips)
label(28)
true.
```

2.2.3 Conclusion

CHR is not able to solve instances greater than $n = 7$ under 15 minutes ($n = 8$ under 2 hours), ECLiPSe can solve $n = 8$ in about 4 minutes. However, instances with $n \geq 9$ are still taking very large amounts of time to solve in ECLiPSe, even with the hybrid approach and/or symmetry breaking. The CP approach complies fully with the results shown in the paper, and it is also clear that the ad-hoc symmetry breaking does not help much the CP search. In our experiments, the CP with the second symmetry breaking outperformed the other methods.

References

[1] A. Bartrlett, T. Chartier, A. Langville, and T. Rankin. An integer programming model for the sudoku problem. *J. Online Math. & Its Appl.*, 8(May 2008), 2008.

[2] R. Bosch and M. Trick. Constraint programming and hybrid formulations for Life. *CP'01*, 2001. Workshop on Modelling and Problem Formulation (Formul '01).

[3] R. Bosch and M. Trick. Constraint programming and hybrid formulations for three life designs. *Annals of Operations Research*, 130:41–56, 2004. 10.1023/B:ANOR.0000032569.86938.2f.

[4] K. Cheng and R. Yap. Applying ad-hoc global constraints with the `case` constraint to still-life. *Constraints*, 11:91–114, 2006. 10.1007/s10601-006-8058-9.

[5] G. Chu, P. J. Stuckey, and M. G. De La Banda. Using relaxations in maximum density still life. In *Proceedings of the 15th international conference on Principles and practice of constraint programming*, CP'09, pages 258–273, Berlin, Heidelberg, 2009. Springer-Verlag. `http://ww2.cs.mu.oz.au/~pjs/still-life/` [last accessed: May 31, 2011].

[6] R. Lewis. Metaheuristics can solve Sudoku puzzles. *Journal of Heuristics*, 13:387–401, August 2007.

[7] I. Lynce and J. Ouaknine. Sudoku as a SAT problem. In *Proceedings of the 9th International Symposium on Artificial Intelligence and Mathematics, AIMATH 2006, Fort Lauderdale*. Springer, 2006.

[8] A. Rossberg. Alice constraint tutorial. sudoku viewpoints., 2011. `http://www.ps.uni-saarland.de/alice/manual/cptutorial/node20.html` [last accessed: May 31, 2011].

[9] H. Simonis. Sudoku as a constraint problem. In *Modelling Workshop CP2005*, 2005.

[10] Sudoku of the day. Diabolical puzzle for saturday 21st may, 2011. `http://www.sudokuoftheday.com/pages/s-o-t-d.php?day=7&level=5` [last accessed: May 30, 2011].

[11] tarek. The hardest Sudoku's, 2011. `http://forum.enjoysudoku.com/the-hardest-sudokus-new-thread-t6539.html` [last accessed: May 30, 2011].

[12] Wikipedia. Sudoku — Wikipedia, the free encyclopedia, 2011. `http://en.wikipedia.org/wiki/Sudoku` [last accessed: May 30, 2011].