Katholieke
Universiteit
Leuven

**Department of
Computer Science**

PROJECT

Advanced Programming Languages for A.I. (H02A8a)

Halilovic-Deruyttere

Armin Halilovic (r0679689)
Thierry Deruyttere (r0660485)

Academic year 2016-2017

Contents

Introduction

In this report we will discuss the different approaches we tried to eventually come to the solutions we have now for Sudoku and Hashiwokakero. The solutions we got are the result of a lot of work and a lot of back tracking on our previously done work. We often came in situations where we got stuck because of the limitations of the ECLiPSe and CHR systems but we also often had to back track on our work since we were often feeling that we were doing things in a non declarative way. We often tried to do things in a procedural way when we first started with Sudoku which means we lost quite some time here since we often had to rethink how we could write things in a more declarative way. For the Hashiwokakero part of the project things went a bit better but we lost quite some time here with the fact that ECLiPSe doesn't support constraints in conditionals. We will discuss this further in chapter 1.4.4. The fact that we often had to backtrack on our work was according to us due to the fact that we are still novices with prolog since this is the first time we used this programming language.

In our solutions we decided to only use ECLiPSe and CHR. This was partially due to the fact that when we started this assignment we still hadn't seen Jess in class. Once we did have the class about Jess we found that since we were still novices at declarative programming languages, it would be a good exercise to continue using the more declarative systems to gain more experience with them since Jess can also be used to program in a more procedural way. Another reason for not using Jess was that the Jess syntax looked less appealing than the syntax CHR was offering us by all the parenthesizes used in its syntax.

Part 1: Sudoku

1.1 Other Viewpoint

For our other viewpoint for Sudoku we decided that each number has N positions (with N the size from one side of the sudoku board). Each of these (X,Y) positions must be different as well as each X must be different from the other positions as well as the Y. This representation has as a benefit that you know that each value from 1 to N must be in X, so we can already fill this in. The only thing left is to find the right Y.

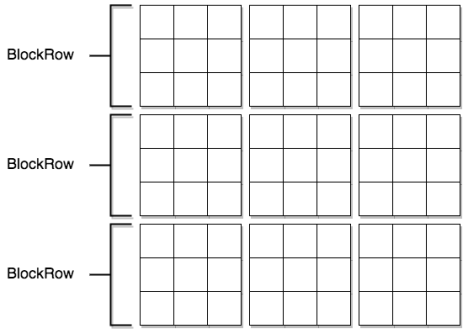


Figure 1.1: Illustration of the definition of a block row

The block constraints can be expressed with a bit of maths thank to this representation. We know each X of all the different positions of a number so we can check in which block row (see illustration) the position is. For example if you take a 9 x 9 board like in the illustration, you know that the rows 1,2,3 belong to block row 1, the rows 4,5,6 belongs to block row 2 and 7,8,9 belongs to block row 3. To express the block constraints we do the following:

1. Take all the positions of a number
2. Check which positions are in the same block row (there are exactly $\sqrt{N}$ positions of a number in the same block row).
3. Express that each position in the same block row should be in a different block in that block row.

1.2 Criteria to judge if a viewpoint is good or not

This is really difficult in our opinion since we had some problems at first to find a good other viewpoint. We think that a good criteria to judge if a viewpoint is good or not is how easy is it to express what you need with this viewpoint, obviously if you lose a lot of time with implementing an other viewpoint then it may be worth it to look for an other viewpoint. Another criteria is how easy is it to use this different viewpoint? In other words, is it a logical representation for the problem? Our final criteria for a good viewpoint is to think about the amount of constraints you need with this viewpoint. Is it really the lowest amount possible?

1.3 Channeling

The channeling constraint for the two viewpoints we are using, the classical one and the one explained above, are in fact not that hard to express. If for the classical viewpoint you know which value there has to be in a certain cell, you also immediately know it's (X,Y) values. For the other viewpoint, if you know the (X,Y) values of a number then obviously you can enter the number at that position.

ECLiPSe channeling constraints

```
1 channel(NumbersPositions, Board):-
2     dim(Board, [N, N]),
3     dim(NumbersPositions, [N, N, 2]),
4 (    multifor([Number, Position, Y], 1, N), param(NumbersPositions, Board, N) do
5         #=(Board[Position, Y], Number, B),
6         #=(NumbersPositions[Number, Position, 2], Y, B)
7 )    .
```

Listing 1.1: Channel constraints in ECLiPSe

The crucial bit of code for the channeling constraints can be found inside the multifor. There you can find two different constraints that links the two viewpoints together. The first one is the channeling constraint for the classical viewpoint. It lays a constraint on the value of Board[Position,Y]. The second constraint is the channeling constraint for our viewpoint. There you can see that we index with `Number`, `Position` and 2. The reason we're only interested in the second value is because the first value is already filled in as stated in section 1.1. Together these constraints expresses that if Y is know in our viewpoint then we can fill in the board with `Number` or if we know which number is at [Position,Y] in the classic viewpoint then we can fill in the Y of our viewpoint. (Or just refer to intro of 1.3 instead of repeat?) This is done with the B at the end of both of the constraints. This B is crucial in the channeling constraints as it expresses that if one of the constraints is correct, then the other should be correct as well!

CHR channeling constraints

```
1  % Viewpoint 1 (thus board) is board(X,Y, BlockIndex, Value),
2  % Viewpoint 2 (thus board_other_viewpoint) is board(Value, X,Y, BlockIndex),
3
4  % The search variable for board is Value1
5  channel, board(X,Y, BlockIndex, Value)
6      , board_other_viewpoint(Value, X, Y2, B2) ==> number(Value), var(Y2), var(B2) |
7          Y2 is Y,
8          B2 is BlockIndex.
9
10 % The search variable for board_other_viewpoint is the Y index
11 channel,board_other_viewpoint(Value, X, Y, BlockIndex),
12     board(X,Y, BlockIndex, V2) ==> var(V2), number(Y), number(BlockIndex) |
13         V2 is Value.
```

Listing 1.2: Channel constraints in CHR

For the channeling constraints in CHR we decided to use an extra predicate `channel`. This is used just so that the system waits for the right moment to start setting up the channel constraints. Expressing the channeling constraints in CHR was a bit easier than doing it in ECLiPSe. Let's start with the first constraint. This expresses that if Value of board and board_other_viewpoint are the same and a are a number and Y2 and B2 are still variables then we can say that Y2 is Y and B2 is BlockIndex. We can say that Value has to be a number for board_other_viewpoint since this is trivial since we insert a number here at construction. The important part was that Y2 and B2 are still variables since this means that we still don't know their values. Why do we use 'is' instead of just removing the old board_other_viewpoint from the constraint store? If we would just remove the constraint then this would mean that other constraints would never be fired! That's why we use is! If Y2 and B2 gets their values this means that maybe other constraints can be solved too. If you remove this constraint from the store you lose this link.

For the second constraint we need to know that both Y and BlockIndex are numbers and V2 is still a var. Since we know both X, Y and BlockIndex then it is very easy to just fill in the Value of that position in board. We use 'is' for the same reason as mentioned before. If V2 finally get's a value then this means that there might be other propagations that can fire.

1.4 Experiments

1.4.1 ECLiPSe

input order as value heuristic, alldifferent from ic_global

Puzzle	Classical Viewpoint (input order)		Our Viewpoint (input order)		Channeling (input order)	
	ms	backtracks	ms	backtracks	ms	backtracks
lambda	0.01s	3	74.669s	78859	0.031s	2
hard17	0.01s	1	84.319s	130134	0.041s	0
eastermonster	0.21s	51	3.29s	3278	0.139s	24
tarek_052	0.34s	59	0.19s	166	0.03s	0
goldennugget	0.671s	104	12.79s	11171	0.63s	70
coloin	0.22s	88	6.469s	4717	1.071s	178
extra2	0.0s	0	691.25s	662099	0.05s	4
extra3	0.011s	3	70.23s	78859	0.03s	2
extra4	0.01s	4	174.23s	197306	0.03s	3
inkara2012	0.03s	3	4.23s	5273	0.19s	34
clue18	0.26s	69	162.28s	156554	0.17s	20
clue17	0.01s	0	180.65s	191972	0.01s	0
sudowiki_nb28	1.03s	413	39.82s	40477	3.36s	832
sudowiki_nb49	0.19s	48	9.86s	10771	0.2s	37

input order as value heuristic, alldifferent from ic

Puzzle	Classical Viewpoint (input order)		Our Viewpoint (input order)		Channeling (input order)	
	ms	backtracks	ms	backtracks	ms	backtracks
lambda	0.34s	4712	36.93s	99470	1.359s	1155
hard17	0.07s	873	53.289s	153383	0.781s	1187
eastermonster	0.01s	119	2.709s	4361	0.131s	85
tarek_052	0.02s	193	0.15s	201	0.01s	6
goldennugget	0.06s	520	8.81s	16938	0.42s	389
coloin	0.19s	2209	3.82s	6998	0.76s	676
extra2	0.18s	4652	477.78s	959808	6.57s	13979
extra3	0.33s	4712	34.6s	99470	1.21s	1155
extra4	0.95s	15116	89.85s	253873	1.19s	1540
inkara2012	0.0s	50	3.78s	8033	0.26s	196
clue18	0.14s	1838	118.8s	268974	3.19s	3604
clue17	0.33s	5520	98.43s	227622	0.15s	177
sudowiki_nb28	0.21s	2851	41.91s	76081	2.33s	2187
sudowiki_nb49	0.11s	1078	5.89s	14424	0.18s	232

first fail as value heuristic, alldifferent from ic_global

Puzzle	Classical Viewpoint (first fail)		Our Viewpoint (first fail)		Channeling (first fail)	
	ms	backtracks	ms	backtracks	ms	backtracks
lambda	0.02s	3	24.07s	32384	0.03s	2
hard17	0.01s	1	6.28s	9183	0.02s	0
eastermonster	0.12s	33	10.69s	11599	0.17s	25
tarek_052	0.16s	35	1.389s	1502	0.071s	6
goldennugget	0.29s	76	19.94s	19870	0.39s	59
coloin	0.04s	8	0.48s	527	0.32s	56
extra2	0.0s	0	0.68s	909	0.02s	0
extra3	0.01s	3	23.88s	32384	0.03s	2
extra4	0.01s	3	585.84s	858267	0.03s	3
inkara2012	0.08s	17	8.33s	10942	0.08s	15
clue18	0.04s	8	175.63s	221669	0.02s	0
clue17	0.01s	0	262.53s	336082	0.019s	0
sudowiki_nb28	0.549s	297	41.25s	46872	1.25s	344
sudowiki_nb49	0.21s	58	23.021s	24536	0.219s	38

first fail as value heuristic, alldifferent from ic

Puzzle	Classical Viewpoint (first fail)		Our Viewpoint (first fail)		Channeling (first fail)	
	ms	backracks	ms	backracks	ms	backracks
lambda	0.091s	977	10.42s	36846	1.2s	1961
hard17	0.031s	419	7.06s	23036	0.139s	386
eastermonster	0.01s	101	6.71s	16122	0.11s	122
tarek_052	0.019s	130	1.11s	3178	0.05s	45
goldennugget	0.051s	358	9.609s	23741	0.261s	310
coloin	0.01s	83	0.29s	929	0.22s	227
extra2	0.49s	7690	0.3s	1044	0.0s	10
extra3	0.08s	977	10.89s	36846	1.34s	1961
extra4	0.19s	2097	289.22s	986497	0.32s	720
inkara2012	0.03s	273	5.36s	13509	0.09s	108
clue18	0.05s	439	88.58s	256741	0.06s	52
clue17	0.03s	270	140.96s	440342	0.01s	15
sudowiki_nb28	0.21s	2221	20.19s	60024	0.65s	864
sudowiki_nb49	0.07s	655	11.9s	32895	0.19s	241

1.4.2 Impact of the different search strategies and alldifferent

Input order vs first fail

When we compare the two tables for `input order` and `first fail` with the alldifferent from ic_global we see that actually for most of the puzzles in all the three columns, the `first fail` heuristic does a better job than `input order`! We can see that there are sometimes really big differences between the two tables, for example 'extra2' for our viewpoint in the `input order` table has 662k backtracks but when you look over at the `first fail` table you see that it only has 909 backtracks! Though, it is not always `first fail` that is faster. If we look at 'extra4' for yet again our viewpoint we see that `input order` is faster this time! 197306 backtracks `input order` for vs 858267 backtracks `first fail`. This is quite odd behavior since 'extra4' is just puzzle 'extra3', where `first fail` outperforms `input order` by more than 50%, with an extra hint.

When we look to the alldifferent from the ic library tables, we see exactly the same behavior as mentioned before. We see that most of the times `first fail` is faster than `input order` but yet again we observe the same odd behavior for 'extra4'.

ic_global vs ic

When we compare the alldifferent from ic_global vs ic we see that the version from ic_global is faster/does less backtracking! For our viewpoint the increase is not that big but when you look at the channeling column and the classical viewpoint, then you see that the increases in backtracks here are quite big!

1.4.3 Analysis of differences

input order vs first fail

alldifferent

The difference with the alldifferent is due to the fact that the alldifferent from ic_global has been programmed with stronger propagation behavior [1] than the alldifferent from the ic library as explained in class. The ic_global implementation has a global view of the constraints and can thus make smarter decisions about it.

[1] `http://eclipseclp.org/doc/bips/lib/ic_global/alldifferent-1.html`

1.4.4 CHR

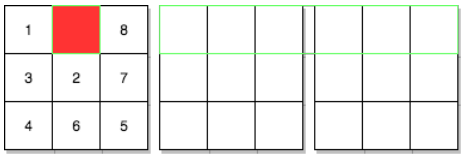
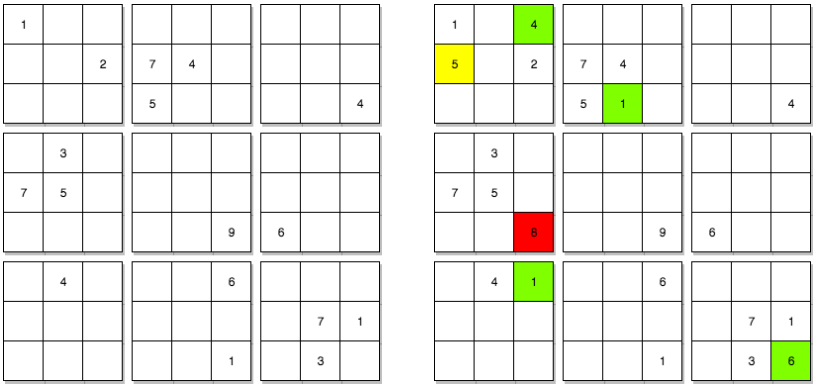


Figure 1.2: Classic (red) vs our viewpoint (green). Figure shows only one block row.

For the CHR implementation of the two viewpoints we had a lot of troubles getting something performant. When we run the two viewpoints on harder boards it takes easily a couple of minutes. Obviously this isn't good so we tried to improve the performance of both our implementations. The first classical viewpoint has some advantages over our other viewpoint since when you look at a block and there is only one spot left you know that whatever number is missing you should input it there. This can be seen on figure 1.2. The red colored box is the only box left where 9 can be inputted from the classical point of view. By seeing this, we made our system quite faster! For the other viewpoint however, the only thing it can see are the squares bounded by the two green rectangles. The correct place is still available but it also sees unnecessary other spaces at this time.

Heuristics classical viewpoint

For the classical viewpoint we found that there was one obvious heuristic that we could use to try to improve the search. To demonstrate this let's take for example the 'lambda' board from the puzzles provided on toledo.



(a) Lambda board

(b) Lambda board after heuristic

Figure 1.3: Lambda board pre vs post heuristic

When we look at 1.3(a) we for example see that in the bottom left block, at the position (9,9), the only possible number that we can play there is a 6 since it cannot be put anywhere else in the block because of the two sixes of the blocks next to us. This information is critical to be able to solve sudoku's in a smart way. This is what we tried to do with our heuristic. Our heuristic will try to let (9,9) know that it is very likely that 6 should be put there. In fact it is 100% 'likely' that 6 should be played there.

The way we do this is we check the domain of (9,9) and we compare it with the domains of all the other squares in the block. When doing this comparison we check for the numbers that is in our domain but not in the domain of the square we're comparing. If a number is not present in the domain of the comparing square, let's say 6 for example, then we know that the likelihood that this 6 is at our position increases. So what we do is we keep track of all the differences between the other squares and we count how many times we've seen a number. We then sort our domain according to this in a descending order. Obviously we want to try things that are very likely first and things that are less likely at a later moment. One important observation to this is that if the count of a number is equals to 9 you know that we are the only position that can play this number else we would never see this number appear 8 times from the differences. We get to 9 because we use our domain as a starting point, so this means that each number in our domain immediately gets a count of 1. We do this because it is not certain that when checking all the differences with the other squares, you will get all the values from your domain. So if we would only use the numbers from the differences we might for example never see a 2 while actually 2 was the only answer that could be played at that position in the end.

This scenario happens when for example all the other squares had also a 2 in their domain so you will never see 2 in the difference set and since we reset the current domain to the likelihood domain this would mean that the 2 would be forever gone which might mean that the system keeps on backtracking to try everything until it sees that nothing is left anymore just to fail. This is obviously something that we want to prevent, this is why we start from our original domain. When we look at figure 1.3(b), we can see the effect of our likelihood heuristic just before the start of the search. We see that already 5 different squares are filled in (the green and red ones). The green ones are some obvious plays but we were actually surprised when it played the red one. The other green ones are 'easy' because for example for the (9,9) example you see that this position is obviously the only place 6 can be played because of the other sixes. But for the red one we don't see a single other 8 on the board and yet it knew that it had to play an 8 there. We have to admit though that our system is a bit flawed since this heuristic cannot be used during the search since too many calculations are made. For example with everything we have filled in we could deduct that for the yellow square, 5 is the only possible play. But this is not feasible since it is way to slow to calculate this. This is why we ordered the domain according to likelihood. If we look at the domain of (2,1) sorted with likelihood, we see that the domain is [5, 3, 6, 8, 9]. So even though we're not fully sure that the 5 had to come there before we filled in the 4 in our block, our system captures the likelihood of a 5 at (2,1) still quite well. Sadly enough this heuristic doesn't really improve the search a lot.

Heuristics our viewpoint

For our viewpoint it was a bit more difficult to find a heuristic and in fact we weren't really able to find a heuristic for it. The only thing that we found obvious to do is to make the domains smaller when we played a number. Let's

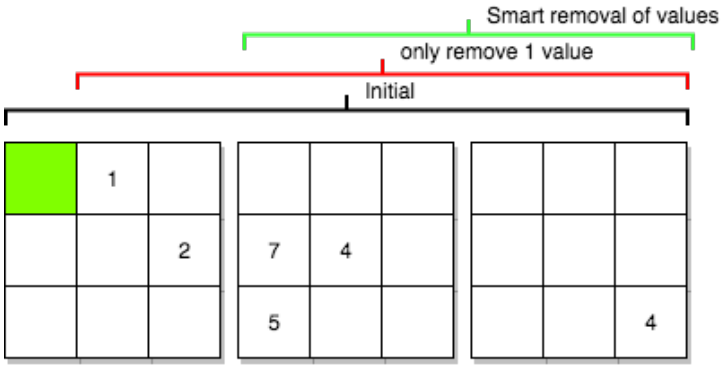


Figure 1.4: Smart domain changes

say we want to play a 3 at the green square on figure 1.4. Our initial domain would be from 1 to 9. If we play the 3 in the green square then we know that for all our other 3's that we cannot put them in the first column anymore. So the other domains for playing 3 will remove the first column from their domain. But this play actually contains more information which we try to exploit with our system. Since we played in the first column, we know that for the other 3's in this block row they cannot be played anywhere in block 1 anymore! So for their domain, instead of just removing the first column, we remove all the columns from the first block from their domain! This means that from 9 values we go to 6 immediately and if we play another 3 in this block row then the last 3 only has a search space of 3 left! By doing this small change we effectively speed up our system quite a bit. Before implementing this we recorded a time of 32s for one of our test boards, with these domain adjustments we went to 14s! So we effectively made our system at least twice as fast with this.

Part 2: Hashiwokakero

2.1 ECLiPSe

2.1.1 Our basic solver

Constraints

Our Hashiwokakero solver in ECLiPSe is based on the stackoverflow post[2] given in the assignment. Therefore we use the constraints mentioned in the post but we also add constraints for connectivity. All our constraints are thus:

- If the islands is not at any of the borders of the board then a bridge that comes from a certain direction, needs to go out on the opposite direction. This means that any bridge coming from the North also leaves from the South or vice versa. Any bridge coming from the West leaves from the East or vice versa. If following one of the directions would mean that we would leave the board, put a 0 on that direction.
- The sum of all bridges coming in/going out from the islands must be equal to the number of the islands.
- Bridges cannot cross each other
-

2.1.2 Optimization

When one thinks about optimizing hashiwokakero there are some obvious things you could do. Like for example if there is a 4 in one of the four corners, you know that you need to have 2 bridges in the two directions that are left. The same goes for a 6 at one of the sides of the board, since one of it's directions is blocked, you know that there has to be 2 bridges in the 3 leftover directions. For an 8 the only possible thing you can do here is that you need to have 2 bridges in each direction. These seem like good improvements, but actually the eclipse system already did these implicitly for us. This is done by

```
1 ( X > 1 -> N #= Board[X-1, Y, 4] ; N = 0 ),
2 (  Y < YMax -> E #= Board[ X, Y+1, 5] ; E = 0 ),
3 (  X < XMax -> S #= Board[X+1, Y, 2] ; S = 0 ),
4 (  Y > 1 -> W #= Board[ X, Y-1, 3] ; W = 0 ),
5
6  % if this position requires an amount of bridges,
7  % make the sum of all bridges equal this amount
```

[2] https://stackoverflow.com/questions/20337029/hashi-puzzle-representation-to-solve-all-solutions-with-prolog-re

```
8  (  Amount > 0 ->
9   % (Amount == 8 ->
10  % Not needed since if the amount is 8
11  % Eclipse knows that everything needs to be 2
12  [N, E, S, W] #:: 0..2,
13  N + E + S + W #= Amount
14  ; % else make sure bridges don't cross each other
15  N = S, E = W,
16  ( N #= 0) or (E #= 0)
17  ) ,
```

Listing 2.3: implicit improvements

Another thing that these constraints do is that if you only have one neighbor then you know that all your possible bridges can only be with them. So we felt like these were not really improvements anymore since they were implicitly done. This is why we decided to look for other improvements.

1. The first improvement we made was noticing that a 1 and another 1 cannot be connected directly to each other. If this would be true we would immediately have a disconnection in our board. Another thing that corresponds with this idea is a 2 and another 2. These can never have 2 bridges between each other since else you would yet again create a disconnection in the set.

2.

2.2 CHR

2.2.1 Optimization

board number	time without improvements	time with improvements
1	0.054s	0.047s
3	0.03s	0.031s
4	0.059s	0.06s
5	0.057s	0.054s
6	0.127s	0.117s
8	0.004s	0.006s
9	0.007s	0.006s
10	27.567s	1.389s
11	0.046s	0.046s
12	0.055s	0.051s
13	0.066s	0.058s
14	0.092s	0.075s
15	1.614s	1.533s
16	1.186s	1.143s
17	2.254s	0.229s
2	aborted after 6405s	1001s

Table 2.1: CHR with connectivity propagator 1

board number	time without improvements	time with improvements
1	0.053s	0.049s
3	0.029s	0.026s
4	0.056s	0.061s
5	0.053s	0.056s
6	0.128s	0.118s
8	0.007s	0.004s
9	0.007s	0.008s
10	28.369s	1.364s
11	0.053s	0.047s
12	0.048s	0.052s
13	0.06s	0.057s
14	0.087s	0.076s
15	1.701s	1.573s
16	1.219s	1.156s
17	2.292s	0.23s
2	unknown	847s

Table 2.2: CHR with connectivity propagator 2

Conclusion

3.1 Weak points

3.2 Strong points

3.3 Lessons learned

Never use this system ever again.

Appendix

We started working on this project before the easter holiday. During our start we often lost quite some time since we didn't really know how the systems worked. During the second week of the easter holiday we continued to work on the project each evening and we finished the Sudoku part at the end of the holiday. We then had to pause our work since we had an enormous deadline for an other course so sadly enough we could only continue working on the project after this deadline (10th may). As the semester was coming to an end other deadlines also started to come closer and closer so we had to manage these first, so it is only at the start of the study period that we could continue our work. From the start of the study period we tried to spend around 6 hours of work each day for this project. The work was not really divided since we were doing pair programming most of the time. Sometimes someone made some individual changes when they had time but most of our work has been made in group over hangouts by using screensharing. We could argue that by doing pair programming we lost quite some time, which is true, but by doing this we worked very closely together and we learned quite a lot.