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. 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.

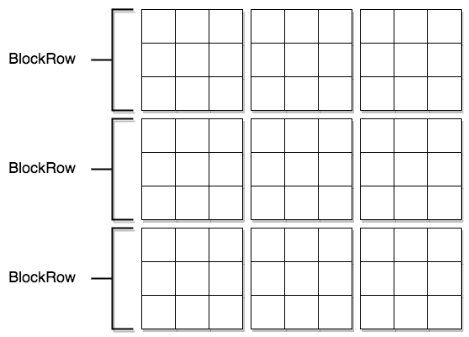

Figure 1.1: Illustration of the definition of a block row

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

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

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

```
% Viewpoint 1 (thus board) is board(X,Y, BlockIndex, Value),
% Viewpoint 2 (thus board_other_viewpoint) is board(Value, X,Y, BlockIndex),

% The search variable for board is Value1
channel, board(X,Y, BlockIndex, Value)
    , board_other_viewpoint(Value, X, Y2, B2) ==> number(Value), var(Y2), var(B2) |
       Y2 is Y,
       B2 is BlockIndex.

% The search variable for board_other_viewpoint is the Y index
channel,board_other_viewpoint(Value, X, Y, BlockIndex),
    board(X,Y, BlockIndex, V2) ==> var(V2), number(Y), number(BlockIndex) |
       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

Part 2: Hashiwokakero

2.1 ECLiPSe

2.1.1 Optimization

Optimization for 4 in corner, 6 at one of the sides or 8 on the board is already fixed implicitly by the (see comments). Single neighbor is also already fixed.

2.2 CHR

2.2.1 Optimization

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.