Advanced Programming Languages for Artificial Intellingence: Project

Toon Nolten r0258654, Joren Verspeurt r0258417

June 2014

1 Introduction

In this assignment we evaluate the usefulness of different constraint programming systems when applied to logic puzzles, specifically sudoku and slitherlink (both by Nikoli). This report discusses the improved version of the solutions we previously handed in for this assignment.

2 Sudoku

2.1 Discussion

ECLiPSe The *ECLiPSe* system provides the programmer with many useful features to solve constraint programming problems. These features include a core language that is largely backwards compatible with (several dialects of) Prolog, a large set of libraries and a development environment. The Prolog language is extended with additional basic data types, arrays and structures, and a versatile iteration construct. Though it may seem strange to use iteration constructs like *for* loops in a logic programming context it does make working with the arrays a lot more efficient for the programmer.

When specifically considering solving Sudoku puzzles in *ECLiPSe* the following aspects of the system are especially useful:

- The built-in array syntax provides a very natural way to reason about the Sudoku board and the cells are easily accessible through the multifor iteration construct.
- Representing the possible ways of filling in a place on the grid as a variable with a finite integer domain is very elegant.
- The ic (_global) library contains many constraints that allow the rules of the Sudoku puzzle to be represented in a concise way, only taking a couple of lines of code, such as the *alldifferent* and *occurrences* constraints.

- The search predicate is powerful and dynamic, allowing for an easy way to test out many different search strategies in quick succession.

However, *ECLiPSe* also has some significant disadvantages:

- Many of the ways of dealing with arrays involve first getting a list representation of the array and then performing operations on this list, which can become very confusing for programmers not yet completely familiar with the system.
- There is no way (that we know of) to get any kind of information about the propagation that is happening inside the solver, and most importantly the propagation that lead to an inconsistency, thus making the program fail. This can make debugging *ECLiPSe* applications a very arduous process, as while it is possible to find out after which constraint the program fails, for example by use of the debugger, but as failure in this case is usually caused by inconsistencies in a combination of constraints this doesn't usually provide enough information to solve the problem.
- There are libraries that contain predicates that clash with those in the standard library. Also some of these are "context- dependent", what this means isn't explained anywhere as far as we know.

CHR Chr allows for constraints regarding the values on a sudoku grid to be easily expressed as chr rules. This is great for problems that can be solved by relying solely on constraints. However for problems like sudoku there is no known set of constraints that solves all puzzles. These kind of problems require search which is hard to do in chr and very hard to do well.

Jess Jess is a rule engine, therefore the rules regarding sudoku should be easily expressible in jess. However, like chr jess's weakpoint is problems like sudoku where rules alone do not suffice. Jess has even less support for search as does chr, the standard way of doing it is to delegate it to java, the host language.

2.2 Viewpoints and Programs

Alternative Viewpoint Instead of considering a map of coordinates to values, we considered a map of values to sets of coordinates. Each of these sets will have size nine and contain only coordinates that do not lie on the same row, in the same column or in the same box as any other in that set.

In chr this is accomplished through a constraint "val_set/3" which holds the value as an identifier, a list of positions that certainly belong to the set and a list of positions which may belong to the set. (In the end we were unable to get this working and we used a different viewpoint in chr: for every row and every value there are 9 possible columns.)

In our *ECLiPSe* implementation we did something similar, we used an array where the row index corresponded to a value to be filled in, the column index

to the actual column in the sudoku grid and the domain of the variables to the possible rows.

2.3 Experiments

In general *ECLiPSe* is much faster than CHR, we are unsure whether this is inherent to these two systems or due to our inexperience.

ECLiPSe Experiments run with our *ECLiPSe* implementation revealed a couple of things:

- The second viewpoint and channeling constraints didn't help the performance much, at least not when using a simple labeling method as search. To the contrary, the extra constraints actually slowed the solution down depending on the order of the constraints, doubling the calculation time from about 3.3 seconds to 7 seconds for the hardest problem (sudowiki_nb28 in this case).
- The search strategy matters a lot for the performance: the time for the puzzle that was the hardest for labeling went down from 3.3 seconds to 0.08 seconds when using the "most constrained" variable selection strategy. The value selection strategy matters as well: among the results for the experiments that used the "most constrained" strategy the times still range between 0.08 seconds and 1 second.

To determine the best variable selection and variable assignment strategies we set up a series of experiments. Every combination of selection and assignment strategies from a selection of strategies is used in turn solving every puzzle given as part of the assignment and the time needed to solve the puzzle is recorded. The strategy combination that gives the best average result for all puzzles is the combination of the "most constrained" and "indomain" strategies. The strategy combination that gives the worst average results is "occurrence"-"indomain_max".
The following variable selection strategies were used:

- *input_order*: Variables are considered in the order that they were passed to the *search* predicate.
- *first_fail*: Variables with the smallest domain are prioritized.
- *smallest*: Variables with the smallest elements in their domain are prioritized.
- *largest*: Variables with the largest elements in their domain are prioritized.
- *occurrence*: Variables that have the largest amount of constraints associated with them are prioritized.

- *most_constrained*: Picks the variable with the smallest domain and if multiple of those exist the one with the most associated constraints is chosen.

The following variable assignment strategies were used:

- *indomain*: Values are tried in order. Failed values are not removed.
- *indomain_max*: Values are tried in decreasing order. Failed values are removed.
- *indomain_middle*: Values are tried from the middle out. Failed values are removed.
- *indomain_min*: Values are tried in increasing order. Failed values are removed.
- *indomain_random*: Values are selected from the domain at random.
- *indomain_split*: Values are selected by splitting the domain and trying the lower half first. On failure the interval is removed entirely.

A comparison between the different puzzles for all of the methods is displayed below. It lists every puzzle that was part of the assignment and the average time it took for the different search strategies to find the solution, the number of times it was the easiest puzzle for a given method and the number of times it was the hardest puzzle for a given method.

Name	Average time	Easiest	Hardest
clue17	0.036	3	0
clue18	0.767	0	5
coloin	1.372	0	7
eastermonster	0.209	0	3
expert	0.017	3	0
extra1	0.024	2	0
extra2	0.018	4	0
extra3	0.019	3	0
extra4	0.016	1	0
goldennugget	0.330	0	0
hard17	0.089	2	0
inkara2012	0.289	0	0
lambda	0.017	10	0
sudowiki_nb28	1.223	0	17
sudowiki_nb49	0.578	0	1
symme	0.689	0	3
tarek_052	0.265	0	0
verydifficult	0.017	8	0

It is clear from these results that *lambda* and *verydifficult* are the easiest puzzles and *sudowiki_nb28* and *coloin* are the hardest puzzles for our implementation. These results seem slightly contradictory to the intuition that having more information to start with should make the puzzle easier to solve: lambda is one of the puzzles with the fewest amounts of given numbers and sudowiki_nb28 has relatively many numbers filled in.

The next table shows just the results for the best search strategy: the combination of "most constrained" and "indomain".

Name	Time	Name	Time
clue17	0.020	clue18	0.090
coloin	0.180	eastermonster	0.190
expert	0.020	extra1	0.020
extra2	0.010	extra3	0.020
extra4	0.020	goldennugget	0.020
hard17	0.020	inkara2012	0.140
lambda	0.010	sudowiki_nb28	0.080
sudowiki_nb49	0.040	symme	0.060
tarek_052	0.180	verydifficult	0.010

For comparison the last table shows the results with only the constraints belonging to the 1st viewpoint: that all of the numbers in every row and column are different and that in the 3×3 blocks all of the numbers are different as well.

Name	Time	Name	Time
clue17	0.040	clue18	0.070
coloin	0.190	eastermonster	0.060
expert	0.010	extra1	0.030
extra2	0.040	extra3	0.290
extra4	0.070	goldennugget	0.080
hard17	0.010	inkara2012	0.130
lambda	0.280	sudowiki_nb28	0.500
sudowiki_nb49	0.040	symme	0.050
tarek_052	0.100	verydifficult	0.010

From the tables it's clear that the difference is quite minimal. Generally for this search strategy we've seen that adding more constraints than necessary slows the solution down (no numerical results given).

CHR I consider only the five sudoku that are mentioned in table 1, most of the others were comparable with "verydifficult", some were almost as hard as "ex-

tra2" or "sudowiki_nb28" but those two were the hardest, I'v included "lambda" because the channeling solution improves dramatically on its performance.

The classic viewpoint is represented by cell/2 constraints which have a coördinate and a list of numbers that could still fit in the cell. A filled cell is just a cell with a list of length 1. The rules of sudoku and a search strategy are expressed as chr rules:

- A set of alldifferent rules which fail when a single row, column or box contains the same value twice.
- A set of eliminate rules which eliminate a value from other cells in the same row, column or box when it is filled in in a certain cell.
- A rule implementing a fail first strategy, a cell with the least possible remaining values is selected and one of the values is attempted.

Other constraints such as "when a value occurs only in one cell in a row, column or box as a possibility, that cell has to be filled in with that value" where tried but these slowed down the performance of the program. As can be seen in table 1, this program performs well although the performance is nowhere near that of the *ECLiPSe* solution.

The alternative viewpoint we implemented is very similar to the classic viewpoint except, one of the coördinates is actually the value that is filled in in the cell and the list of values is a list of possible columns for the specific value on a specific row. This is represented by rvc/2 constraints. The rules are as follows:

- A set of alldifferent rules very similar to the previous one.
- A set of constraints to eliminate options from other rvc's when one is filled in. In this viewpoint the rule that no two cells in the same box can have the same value is much harder to express, therefore that rule was left out. Short experiments with the classic viewpoint showed that this rule is not essential but it might speed up some of the more complex puzzles.
- A first fail search strategy that is very similar to the last one, except now a column is chosen for the rvc that has the fewest remaining possibilities.

Because this viewpoint is so similar we expected nearly the same results, however that is not nearly the case. The difference is so large that the only puzzles I have any timings for are "test"(which is almost trivial), "verydifficult" which takes 40000 times as long. The lack of the elimination rule for boxes cannot explain such a huge difference. It is possible that solving a sudoku row by row (which is what the search does) is incredibly inefficient but the difference in performance still seems too big. The other possibility is that something is wrong with our implementation of the rules, we haven't found a mistake yet but the difference is so large that this is the most probable explanation.

The combination of these viewpoints was reasonably straightforward, the alldifferent rules are shared between the two viewpoints, the one for boxes from the classic viewpoint, the ones for rows and columns from the second viewpoint.

Both sets of elimination rules are used because this leads to better performance than mixing or using only one. And both search strategies are used which makes the most difference in performance, this way either a cell is chosen with fewest possible values or an rvc is chosen which can only be filled in in the fewest possible columns. Two constraints are added so updates to the cells and rvc's are easier: single/3 and remove/3. The single constraint fills in a value in a cell and the right column in the corresponding rvc, the remove constraint removes a certain value from a cell and the right column from the corresponding rvc. This implementation performs very similar to the first one for the easiest puzzles. It outperforms the first one by a large margin on "lambda" and especially "extra2" but it loses by a lot on "sudowiki_nb28". We expected this method to perform better overall because it has a more complete fail first strategy, however we cannot explain why it performs so much slower on the "sudowiki_nb28" puzzle.

	verydifficult	expert	lambda	extra2	sudowiki_nb28
classic vp	0.024	0.343	6.299	29.855	28.135
	154,077	2,144,591	38,687,501	187,954,363	176,091,128
other vp	949.641		8634.513		9797.167
	5,954,999,111		40,506,063,752		48,362,688,821
channeling	0.184	0.360	0.217	0.741	82.326
	1,455,227	3,098,789	1,875,723	6,854,189	759,734,989

Table 1: CHR: Performance solving sudoku (time in s, #inferences)

3 Slitherlink

3.1 *ECLiPSe*

In the case of Slitherlink *ECLiPSe*'s power is limited by the possible representations for the Slitherlink puzzle as *ECLiPSe* arrays. Perhaps an object oriented system would have some advantages here, as one could represent the grid as squares having multiple different attributes. This could make addressing the different edges and crossings less confusing and thus improve programmer efficiency. The best ways of dealing with reified constraints and the logical connections between them also takes some getting used to. As will be explained later we thought of representing the parts of the single cycle as a list of lists of dots but we couldn't find any way of elegantly representing this in a form that was useful in *ECLiPSe* data structures. The solution we ended up going for turned out to be quite strange as a result.
The main advantage of *ECLiPSe* is that it is relatively easy to come up with a working solution in relatively little time. The copious amount of built-in functionality is quite useful.
The main disadvantage of *ECLiPSe* is the fact that its arrays of domain variables are pretty much the only data structures that are available. We feel that the availability of more sophisticated data structures would make it significantly

easier to model this problem. Also the system is geared towards expressing absolute truths as consequences of certain conditions in stead of possibilities and impossibilities. For example it would be natural to write a constraint that limits the domain of a dot variable based on the value of a nearby edge value but this doesn't seem to work well in *ECLiPSe*, perhaps this was due to our incompetence but we tried several different ways of expressing this and it took quite a lot of time to get it into a form that actually did what we expected it would. As a rule of thumb the more things you can express as a sum and an equality the better. Which is rather frustrating, because this can make the resulting code quite confusing.

Primary Viewpoint In *ECLiPSe* the primary viewpoint we used represented the puzzle grid as a combination of dots and edges, where the dot can have a certain set of values between 0 and 12. These values are determined by assigning a power of 2 to each edge connected to it as a value and finding all possible values their weighted sum can have, given that the weights are 0 or 1 depending on whether the edge is colored or not. The fact that only 2 edges connected 2 a point can be colored at the same time further constricts these values. Of course the domains of the dots at the edges are limited even further.

Alternative Viewpoint In *ECLiPSe* the first alternative viewpoint represents the board as a set of dots that are grouped into segments. Originally we wanted to make the presentation as a list of lists of dots but there was no practical way we could find to do this, so it ended up as the following: every dot on the grid has a variable associated with it that can have a value between 0 and (an loose approximation of) the largest possible amount of cycles in that grid. We then constrain the values of the variables that correspond to dots that belong to the same cycle to be equal. In this way we know that at the end only 2 possible values are possible: 0 and an arbitrary other value in the domain, the 0 being assigned to dots that are not in a cycle. To make this a complete viewpoint one might associate edges to the segments as well, constrain them with the values in the squares on the grid and mark edges that could possibly extend the segment. Search on this representation could potentially be very efficient but as only the necessary parts of the viewpoint's representation are implemented this was not explored. As a final remark about this viewpoint another advantage of a complete implementation could be that edge-colorings that close a loop could be detected instantly this way.
The second alternative viewpoint represents the grid as squares and edges in stead of dots and edges. We tried adding a couple of constraints that fill in edges in places where we know they have to be, for example between two adjacent threes etc, but couldn't get it to work properly.
Introducing the "squares" viewpoint makes it possible to perform search on the variables in this viewpoint in stead of the ones in the standard viewpoint, the effect this has on the performance is discussed in the "experiments" part.

Experiments Experiments were performed comparing times for different alternatives for the following options:

- Whether to use the "squares" viewpoint or not
- Given that the "squares" viewpoint is used whether to perform search on its variables or on the primary viewpoint's variables.
- Whether to enforce the single cycle rule with constraints or by checking after the search.
- Which variable selection strategy to use for the search.
- Which value assignment strategy to use for the search.

Generally the results depend very strongly on the puzzle. Some options provide significant benefits for one puzzle but adversely affect performance for other puzzles. This is illustrated in the following figures: Figure 1 and Figure 2. As

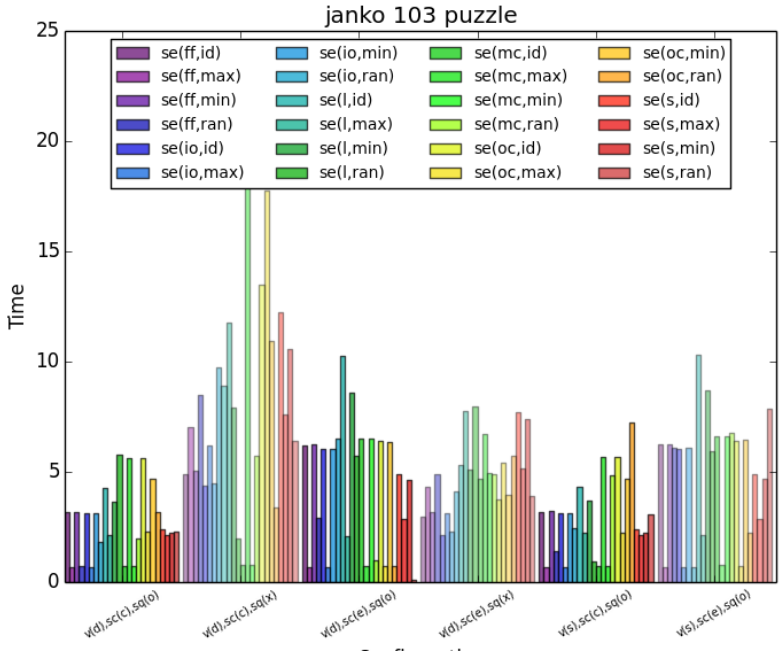

Figure 1: A comparison for puzzle 103 from the Janko website

can be seen from Figure 2 the use of the "squares" viewpoint (the entries with sq(o) as their x label) negatively affects performance, almost quadrupling the worst case time. However, looking at Figure 1 it can be remarked that for that puzzle the "squares" viewpoint has no negative impact, even a slightly positive

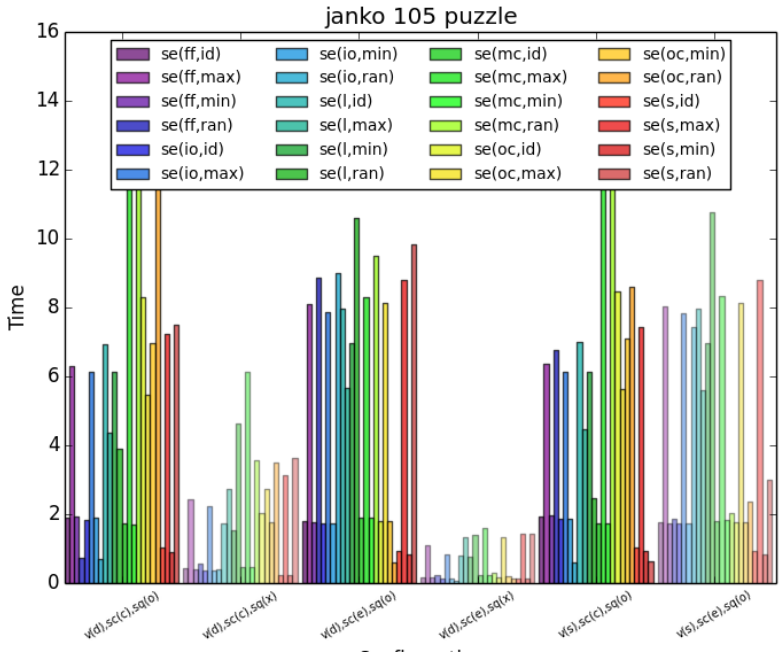

Figure 2: A comparison for puzzle 105 from the Janko website

impact. For most puzzles it has a negative impact on performance.
The biggest factor in the execution is, naturally, the size of the puzzle. Our solution can solve every size 5 puzzle in less than 0.04s and the size 10 puzzles in 5 seconds or less. There is a size 12 puzzle that is solvable in a couple of minutes but there is also a size 12 puzzle that is not solvable with a timeout of 2 hours.

Name	Time	Name	Time
a2	0.04	b2	0.01
a3	0.02	b3	0.01
a5	0.02	b5	0.06
x5	0.04	b7	0.04
a10	0.43	b10	4.79

3.2 Slitherlink in CHR

For the representation we used the one in the paper that is mentioned in the assignment. A c/3 constraint represents a cell with a number, once a cell's edges have been filled in according to the number in the cell the 'c' constraint is

removed, this makes it easy to check that in the end all the numbered cells have been satisfied. An h/3 and a v/3 constraint are used to represents horizontal and vertical edges respectively, the third value is whether an edge is drawn or not, this is the only ungrounded term in a constraint so it can be used to propagate choices. To make it easier to state the constraint that every point has either degree 0 or 2, edges are added outside of the actual grid (the edges that are perpendicular to the border of the grid), these edges always have 0 as fill. There is a segment/2 constraint so we can implement the first hint from the assignment.

The rules were implemented with CHR rules as follows. We used the first hint from the assignment to implement the single cycle constraint. Every edge that is filled creates a segment, segments are joined and in the end only one segment should remain as a loop between a point and itself. If any edges haven't been ground when the cycle is closed they are filled with 0's (so, not filled), if a cycle is closed but another segment remains it is a failure, if a cycle is closed but there are still 'c' constraints it is a failure because those constraints haven't been fulfilled and can never be fulfilled because that would imply creating another segment and a segment together with a cycle is a failure. The cell number constraints were implemented so that a guess is made at how the edges should be drawn if the choice turns out to be wrong another one is made. Then there is one rule for the degree on a point, this rule has guards so it only fires when either three edges are known or two edges are known to be filled. If this rule was not guarded like this it would activate for every point on the grid and make a guess and this resulted in a lot of backtracking because often a choice made very early on turned out to be wrong and this repeated because every time a choice was made for every point on the grid, this worked very well for z1 through b3 (a1 is not considered to be a valid slitherlink puzzle because only 1 valid puzzle can be made with a cell numbered 4) but takes a long time (we did not actually let it finish, so it may not have worked at all). The last rules are to implement the depth first search strategy, the first rule searches at one side of a segment because most of the time those are the best places to search.

Additional constraints were not implemented. Most constraints we could think of, for example when three 3's neighbour each other the edges have to follow a kind of snake pattern, were already covered by the cell number rules because they immediately make a choice for a cell, so they would choose for one 3 then make a compatible choice for the next 3, until either each 3 was satisfied or a failure happened and choices had to be revised. There was one constraint regarding 2's in corners which could be interesting, when there's a 2 in any corner the two edges which 'extend' the corner have to be drawn. But we did not implement this because it only works when there's a 2 in a corner our search would pick those edges without guessing because they're the only possibilities so it would not speed things up a lot.

Alternative Search Strategy Because our rules make a choice for every numbered cell immediately, the alternative search strategy is not easily implemented. We did not attempt implementing this strategy.

Experiments Our basic solution for slitherlink is not fast but it seems to be correct. In table 2 the performance on the easiest puzzles is listed. A few puzzles stand out because they take longer than a second: a5, x5, janko5e. b7 takes 16 seconds, so the larger puzzles seem to take longer. However, janko5b takes 50s and janko5e takes 620s while janko5c takes only 0.651s this seems to indicate that the time it takes is proportional to the length of the cycle and seeing as our search strategy searches along the cycle, adding one segment each time, this is very logical. We believe the complexity of this search strategy is exponential making it feasible only for small problemsizes.

	basic solution
z1	0.002
ar2	0.004
a2	0.012
b2	0.012
a3	0.024
b3	0.034
a5	3.750
b5	0.254
x5	3.729
b7	16.034
a10	500.210
janko5b	49.826
janko5c	0.651
janko5d	619.872
janko5e	4.666

Table 2: CHR: Performance solving slitherlink (time in s)

4 Conclusion

We learned that constraint programming allows solving a non-trivial problem with a very short program. However this succinctness comes at a price, you have to be very precise when formulating your constraints because they do not always mean what you think they mean. Constraint programming requires a lot more focus and attention than imperative programming but it does often lead to elegant solutions.

5 Appendix

We each worked on this project for 50 hours.

The extra Slitherlink puzzles we tested on came from janko.at.