Katholieke
Universiteit
Leuven

**Department of
Computer Science**

PROJECT

Advanced Programming Languages for A.I. (H02A8a)

Deruyttere-Halilovic

Thierry Deruyttere (r0660485)
Armin Halilovic (r0679689)

Academic year 2016-2017

Contents

Introduction **2**

Task 1: Sudoku **3**
1.1 Alternative Viewpoint . . . 3
1.2 Criteria to judge if a viewpoint is good or not . . . 4
1.3 Channeling . . . 4
1.4 Experiments . . . 6
1.4.1 ECLiPSe . . . 6
1.4.2 Impact of the different search strategies and alldifferent . . . 8
1.4.3 Analysis of differences . . . 8
1.4.4 CHR . . . 9

Task 2: Hashiwokakero **12**
2.1 ECLiPSe . . . 12
2.1.1 Basic solver . . . 12
2.1.2 Improvements . . . 13
2.1.3 Experiments . . . 14
2.2 CHR . . . 15
2.2.1 Data representation . . . 15
2.2.2 Constraints representation . . . 15
2.2.3 Propagation and search . . . 17
2.2.4 Connectivity constraint . . . 18
2.2.5 Improvements . . . 19
2.2.6 Experiments . . . 21

Conclusion **23**
3.1 Weak points . . . 23
3.2 Strong points . . . 23
3.3 Lessons learned . . . 23

Appendix **24**

Introduction

In this report we discuss different solvers we have created for Sudoku and Hashiwokakero puzzles. They are the result of a lot of work and a lot of backtracking 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 backtrack on our work since we often felt that we were doing things in a non-declarative way. We often tried to do things in a procedural way, which means we lost quite some time rethinking how we could write things in a more declarative way.

For the solvers 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 lecture on Jess, we felt 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 with all the parentheses used in its syntax.

Task 1: Sudoku

1.1 Alternative Viewpoint

For the alternative viewpoint in Sudoku we decided to use the following representation. Each possible sudoku number must occur in N positions which are represented as tuples (X,Y) (with N the size of the sudoku board). Therefore, we keep track of arrays of length N for each possible sudoku number. The arrays hold positions on which the numbers occur. Each of these positions (X,Y) can only be used once by any of the numbers. Within one such array, all of the X's and all of the Y's must be different, so that a number does not occur multiple times on a row or column on the board. In the arrays, we use the indices to represent the different X values. The values at those indices represent the Y values of positions. These values are the search variables in this viewpoint.

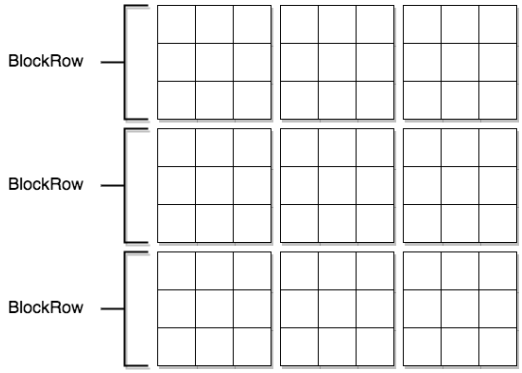


Figure 1.1: Illustration of the definition of a block row

The block constraints can be expressed with a bit of maths using this representation. For a specific number, we already know each X of all of the different positions, so we can check in which block row (see illustration) the position is. First, we can split the board up in block rows. For example, if you take a 9x9 board like in the illustration above, you know that rows 1,2,3 belong to block row 1, rows 4,5,6 belongs to block row 2 and rows 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, since we had some trouble at first to find a good alternative viewpoint. We believe that the following criteria can be used to judge whether a viewpoint is good or not.

- Ease of expressing what is needed with the viewpoint. Obviously, if you lose a lot of time implementing a viewpoint then it may be worth it to look for an alternative viewpoint.
- How easy is it to understand the viewpoint? In other words, is it a logical representation for the problem?
- Lastly, think about the amount of constraints necessary with the viewpoint. Is it the lowest amount possible?

.

1.3 Channeling

The channeling constraint for the two viewpoints, the classical one and the one explained above, are in fact not that hard to express in words. If in the classical viewpoint you know which value there has to be at a certain (X,Y) position, you can immediately set the position in the array of the value in the alternative viewpoint. For the alternative viewpoint, if you know an (X,Y) value for a number, then you can enter the number at that position in the classic viewpoint. This of course is trivial in words but finding the correct way to express it in ECLiPSe was not that straightforward.

ECLiPSe channeling constraints

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

Listing 1.1: Channel constraints in ECLiPSe

The crucial bit of code for the channeling constraints are inside the multifor. There you can see 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[X, Y]. The second constraint is the channeling constraint for the alternative viewpoint. There you can see that we index with `Number` and `X`. Together these constraints expresses that if Y is known in the alternative viewpoint then we can fill in Board[X, Y] with `Number` or if we know which number is at [X, Y] in the classic viewpoint then we can fill in the Y of the alternative viewpoint. 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 true, then the other should be true as well.

CHR channeling constraints

```
1  % the search variable for board is Value
2  channel, board(X, Y, BlockIndex, Value), board_viewpoint_2(Value, X, Y2, B2)
3      ==> number(Value), var(Y2), var(B2) |
4          Y2 = Y,
5          B2 = BlockIndex.
6
7  % the search variable for board_other_viewpoint is Y
8  channel, board_viewpoint_2(Value, X, Y, BlockIndex), board(X, Y, BlockIndex, V2)
9      ==> var(V2), number(Y), number(BlockIndex) |
10         V2 = 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 solver sets up the channel constraints at the right moment. Expressing the channeling constraints in CHR was a bit easier than doing it in ECLiPSe. The first rule expresses that if Value of board and board_viewpoint2 are the same, and if Value is a number, and if Y2 and B2 are still variables then we can say that Y2 equals Y and B2 equals BlockIndex. The important part here was that Y2 and B2 are still variables since this means that we don't know their values yet. Why do we initialise Y2 and B2 instead of just removing the old board_viewpoint2 from the constraint store? If we would just remove board_viewpoint2 then this would mean that other constraints would never be fired! When Y2 and B2 get their values, other rules which could not be fired before might be fired now.

For the second constraint we need to know that both Y and BlockIndex are numbers and V2 is a free variable. Since we know both Y and BlockIndex, we can just fill in the Value of that position in board.

1.4 Experiments

1.4.1 ECLiPSe

input order as value heuristic, alldifferent from ic_global

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

input order as value heuristic, alldifferent from ic

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

first fail as value heuristic, alldifferent from ic_global

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

first fail as value heuristic, alldifferent from ic

Puzzle	Classical Viewpoint		Alternative viewpoint		Channeling	
	ms	backtracks	ms	backtracks	ms	backtracks
lambda	0.091	977	10.42	36846	1.2	1961
hard17	0.031	419	7.06	23036	0.139	386
eastermonster	0.01	101	6.71	16122	0.11	122
tarek_052	0.019	130	1.11	3178	0.05	45
goldennugget	0.051	358	9.609	23741	0.261	310
coloin	0.01	83	0.29	929	0.22	227
extra2	0.49	7690	0.3	1044	0.0	10
extra3	0.08	977	10.89	36846	1.34	1961
extra4	0.19	2097	289.22	986497	0.32	720
inkara2012	0.03	273	5.36	13509	0.09	108
clue18	0.05	439	88.58	256741	0.06	52
clue17	0.03	270	140.96	440342	0.01	15
sudowiki_nb28	0.21	2221	20.19	60024	0.65	864
sudowiki_nb49	0.07	655	11.9	32895	0.19	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 sometimes are large differences between the two tables. For example, 'extra2' for the alternative 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 the alternative viewpoint we see that `input order` is faster this time! 197306 backtracks for `input order` vs 858267 backtracks for `first fail`. This is quite odd, since 'extra4' is just puzzle 'extra3' (where `first fail` outperforms `input order` by more than 50%) with an extra hint.

When we look at the tables where we use alldifferent from the ic, 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 the alternative 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

To be able to explain these behaviors, we first have to know the exact difference between the two search strategies. `Input order` takes the first domains that has been created, while `first fail` starts from the smallest domain. We think that the difference for 'extra4' can explained as follows: As you always take the smallest domain, if you make a wrong decision early on, you will need to do a lot of backtracks in a wrong branch of the tree. Since you always took the smallest ones first you get a tree branch that is very wide. If your first choice was a mistake you need to traverse this whole subtree first before you can backtrack to your first choice. We believe that the extra hint resulted in this situation. It has probably reduced a domain which creates these problems. `Input order` doesn't suffer from this since it just takes the first entered domains. On the other hand, sometimes taking the smallest domain gives you the best result because you have less choices to make here, which would explain the behavior for 'extra2'.

alldifferent

The difference with the alldifferent versions 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.

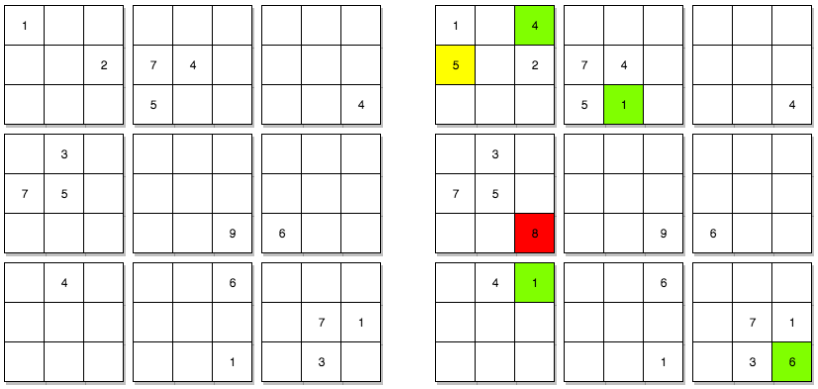
[1] `http://eclipseclp.org/doc/bips/lib/ic_global/alldifferent-1.html`

1.4.4 CHR

For the CHR implementation of the two viewpoints we had a lot of trouble getting something that performs well. When we run the two viewpoints on harder boards it takes easily a couple of minutes. Obviously this isn't good, so we improved the performance of both our implementations using the heuristics explained below.

Heuristic 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 in the assignment.



(a) Lambda board (b) Lambda board after heuristic

Figure 1.2: Lambda board pre vs post heuristic

If we look at 1.2a we can see that in position (9,9), the only possible number we can play there is a 6, since it cannot be put anywhere else in the block because of the two sixes of the neighboring blocks. This information is critical to be able to solve sudoku puzzles in a smarter way. This is what we tried to do with our likelihood heuristic. Our heuristic will state that it is very likely that 6 should be put on position (9,9). In fact it is 100% 'likely' that 6 should be played there.

The way we do this is by checking the domain of (9,9) and comparing it with to the domains of all the other squares in the block. When doing this comparison, we check for the numbers that are in our domain but not in the domain of the square we're comparing with. If a number is not present in the domain of the other square, e.g. number 6, then we know that the likelihood that this 6 is at our position increases. Thus, 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 based on the count. 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 equals to 9, you know that we are in 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. 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, that 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.

When we look at figure 1.2b , 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 found the red one. The other green ones are 'easy' because for the (9,9) example you see that this position is obviously the only place a 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.

Heuristic alternative viewpoint

To increase the search speed for the alternative viewpoint we decided to do something about the search space itself. Consider the following example: Let's say we want to play

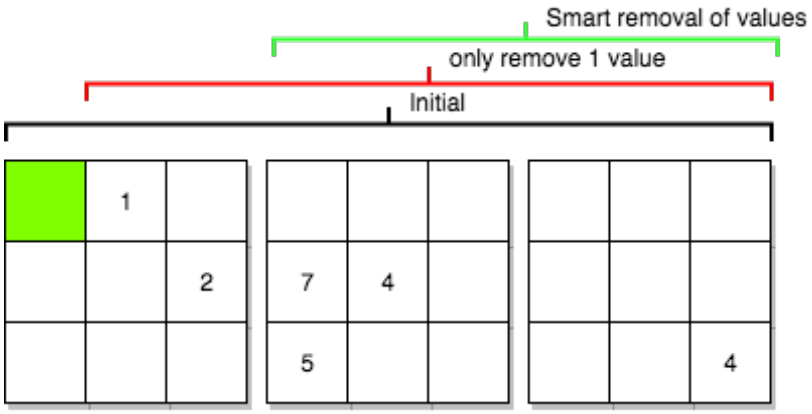


Figure 1.3: Smart domain changes

a 3 at the green square on figure 1.3. Our initial domain would be from 1 to 9. If we play the 3 in the green square, then we know that we cannot put aby other 3's in the first column. So the in the domains of other 3's, the first column is removed. But this play actually contains more information which we try to exploit here. Since we played in the first column, we know that the other 3's in this block row cannot be played anywhere in block 1 anymore! So for their domains, instead of just removing the first column, we remove all the columns in the first block. This means that from 9 values, we go to 6, 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.

Experiments

When we compare both tables, we see that for the alternative viewpoint there is a good improvement. Although when we look at the impact of the heuristic for the classic viewpoint, we see that it is in most cases quite a bit slower! However, for the harder boards like hard17 and clue18 we can see a big speed up! The impact of channeling here can be seen in the last column. Sometimes it does bring a serious improvement compared

Puzzle	Classical Viewpoint s	Alternative viewpoint s	Channeling s
lambda	240.556s	31.546s	268.08s
hard17	1410.554s	111.446s	581.685s
eastermonster	2.626s	52.918s	3.78s
tarek_052	9.705s	47.722s	19.056s
goldennugget	13.731s	40.311s	20.638s
coloin	92.932s	21.009s	99.832s
extra2	14.038s	553.181s	19.242s
extra3	235.877s	31.142s	269.242s
extra4	91.517s	69.132s	128.158s
inkara2012	92.476s	294.721s	114.792s
clue18	1854.41s	1373.993s	2743.256s
clue17	6.168s	142.121s	9.855s
sudowiki_nb28	289.829s	11.816s	278.37s
sudowiki_nb49	47.434s	14.049s	64.556s

Table 1.1: Sudoku in CHR without heuristics

Puzzle	Classical Viewpoint s	Alternative viewpoint s	Channeling s
lambda	294.197s	4.47s	141.152s
hard17	218.107s	0.225s	30.291s
eastermonster	52.796s	11.054s	31.252s
tarek_052	1.351s	56.398s	1.81s
goldennugget	16.791s	26.212s	18.233s
coloin	67.885s	25.439s	46.873s
extra2	360.367s	37.646s	26.468s
extra3	277.144s	4.452s	143.193s
extra4	128.014s	5.107s	65.539s
inkara2012	7.399s	161.194s	7.062s
clue18	774.288s	312.615s	479.363s
clue17	16.193s	0.534s	7.414s
sudowiki_nb28	994.815s	3.527s	477.672s
sudowiki_nb49	103.807s	14.097s	112.585s

Table 1.2: Sudoku in CHR with heuristics

to a standalone technique. Sometimes it is even worse than the two techniques separately for example clue18 in the table without heuristics. We we look to the table with heuristics however we see that there has been a very big increase in speed for clue18!

Task 2: Hashiwokakero

2.1 ECLiPSe

2.1.1 Basic solver

Constraints

The ECLiPSe Hashiwokakero solver is based on the stackoverflow post[2] given in the assignment. We have used the proposed data structure and constraints in the post, and have added constraints for the connectivity of the islands. The resulting constraints are:

- For every position on the board it holds that the amount of bridges going in one direction (e.g. N) equals the amount of bridges going in the opposite direction (e.g. S) from the next position in the original direction (N in this example). If the position is on an edge of the board, the amount of bridges in the direction that would go outside of the board is zero.

```
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 ),
```

- The sum of all bridges connected to an island must be equal to the number of the island.

```
1  [N, E, S, W] #:: 0..2,
2  N + E + S + W #= Amount
```

- The amount of bridges going in one direction equals the amount of bridges going the opposite direction and bridges cannot cross each other.

```
1 N = S, E = W,
2 (N #= 0) or (E #= 0)
```

- The connectivity constraint is implemented by checking that all islands can be visited starting from a certain island. This is done after the search. fill_set_visit is called with a certain starting island. Then, the Visited list is filled with the islands that can be reached from the starting island. Afterwards, the length of Visited must equal the amount of islands on the board. If the length of Visited does not equal the amount of islands on the board, this means that not all islands can be visited from a certain island, which then means that the bridges do not connect the islands into a single connected set.

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

```
1  board_connected_set(Board) :-
2      board_islands(Board, Islands),
3      length(Islands, N),
4      length(Visited, N),
5      % make the island be member of current set
6      nth1(1, Islands, [X, Y]),
7      % set position to visited
8      nth1(1, Visited, 1),
9      % travel to the neighbors of the current position and fill the Islands/Visited
           set
10     fill_set_visit(Board, X, Y, Islands, Visited),
11     % if all free variables in Visited have been bound, then all islands form a
           connected set
12     count_nonvars(Visited, N).
```

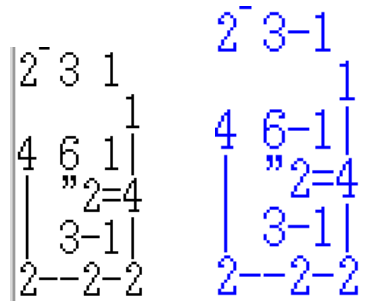
All the constraints are active, except for the connectivity constraints. The connectivity constraint does not directly make changes to domains of variables so it a passive constraint. The other constraints do make changes to domains.

2.1.2 Improvements

When one thinks about improvements for solving Hashiwokakero there are a couple of obvious ones that pop up. For example, if there is a 4 in one of the four corners, the only way to get 4 bridges is by placing 2 bridges in the two possible directions. The same idea goes for a 6 at one of the edges of the board. Because one of it's directions is blocked, the only way to get 6 briges is by placing 2 bridges in all 3 of the possible directions. For an island with an 8, the only option is to place 2 bridges in every direction. These seem like good improvements, but actually they were already done implicitly by ECLiPSe's domain solving. Since these were not really improvements to our solver, we had to look for other improvements. The following improvements are inspired by `http://www.conceptispuzzles.com/index.aspx?uri=puzzle/hashi/techniques`.

1. The first improvement we made was stating that a 1 and another 1 cannot be connected directly to each other. If there was a bridge between them, we would immediately have an invalid solution, as all of the islands in the end result must form a single connected set. Another situation that corresponds with this idea is a 2 and another 2. These can never have 2 bridges between each other since then you would also create an isolated segment. By stating that there cannot be a bridge between islands with 1 and that there cannot be two bridges between islands with 2, we have added more knowledge before the search starts.
   Figure **??** shows the result of this improvement. It shows two boards before the search phase. The board with the improvement contains two more briges before the search phase. This means that the during the search less backtracking will occur in the worst case.

2. The second improvement builds further on the idea of avoiding isolated segments. Assume there is an island with a 2 that has three neighbors, two of which are islands with 1. If you were to place a bridge from the 2 to both of the 1's, then you would end up with an isolated segment. In this case, we know that the third neighbor will have at least one bridge going to the 2. This is implemented by stating that the amount of bridges going from the 2 to the third neighbor cannot be zero. This adds more knowledge to start with before the search.



(a) board without improvement

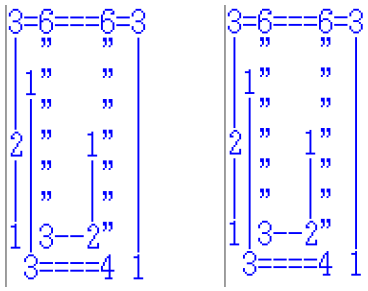
(b) board with improvement

Figure 2.4: boards without and with improvement

3. The third improvement avoids isolated segments for islands with a 3 that have 3 neighbors, among which one neighbor has a 1 and another has a two. If you connect one bridge with the 1 and 2 bridges with the 2, you get an isolated segment. In this case, we know for sure that the third neighbor of the island with 3 has at least one bridge going to the 3. This is implemented by stating that the amount of bridges going from the 3 to the third neighbor cannot be zero. Again, this adds more knowledge to start with before the search.

All three of the improvements have an impact on the connectivity constraint. Because the improvements avoid isolated segments before the search starts, the amount of backtracks (in the wost case) caused to satisfy the connectivity constraint will be reduced.

2.1.3 Experiments



(a) board pre search

(b) board post search

Figure 2.5: Some of the boards are ready even before the search (board 5)

ECLiPSe Hashiwokakero solutions using input order

Table 2.3 shows the results of searching with the input order heuristic.

ECLiPSe Hashiwokakero solutions using first fail

Table 2.4 shows the results of searching with the first fail heuristic.

Puzzle	Without improvements		With improvements	
	ms	backtracks	ms	backtracks
1	0	1	0	1
2	130	60	81	42
3	0	0	0	0
4	0	0	0	0
5	0	0	0	0
6	10	1	10	1
7	0	0	0	0
8	0	0	0	0
9	0	1	0	1
10	49	160	61	160
11	0	0	0	0
12	0	1	0	1
13	10	0	0	0
14	0	1	0	1
15	20	4	20	4
16	21	28	20	28
17	10	1	9	0
18	0	1	0	1
19	0	0	0	0
20	0	2	0	1
21	0	1	0	0
22	10	9	10	9

Table 2.3: ECLiPSe Hashiwokakero solutions using input order

2.2 CHR

2.2.1 Data representation

We used the same data representation as in the ECLiPSe solution. The data is stored in 'board/7' facts with variables X, Y, Am, N, E, S, W, where (X, Y) represents the position of the board, Am the amount of bridges that must be connected this position. N, E, S, and W represent the amount of bridges that go in a certain direction from position (X,Y).

2.2.2 Constraints representation

- The amount of bridges going in one direction equals the amount of bridges going in the opposite direction on the next position.

```
1 bridge_constraints, board(X, Y, _, N, _, _, _), board(X2, Y, _, _, _, S, _)
2     ==> X > 1, X2 is X-1 |
3     N eq S.
4
5 bridge_constraints, board(X, Y, _, _, E, _, _), board(X, Y2, _, _, _, _, W)
6     ==> Y2 is Y+1 |
7     E eq W.
```

- There are no crossing bridges. In other words, if the amount of bridges going in one direction is greater than zero, then the amount of bridges going in the perpendicular position must be zero.

Puzzle	Without improvements		With improvements	
	ms	backtracks	ms	backtracks
1	0	1	0	1
2	170	61	20	61
3	0	0		0
4	0	0	0	0
5	0	0		0
6	10	1		1
7	0	0		0
8	0	0		0
9	0	1	0	1
10	110	152	69	152
11	0	0	0	0
12	0	1	10	1
13	0	0	0	0
14	10	1	0	1
15	80	4	90	4
16	50	28	61	28
17	0	1	10	0
18	0	1	0	1
19	0	0	0	0
20	0	2	0	1
21	0	1	0	0
22	10	9	10	9

Table 2.4: ECLiPSe Hashiwokakero solutions using first fail

```
1 board(_, _, 0, N, E, _, _) ==> number(N), N > 0 | E = 0.
2 board(_, _, 0, N, E, _, _) ==> number(E), E > 0 | N = 0.
```

- Bridges cannot go outside of the board. The amount of bridges going in a direction that would leave the board are set to zero here.

```
1 bridge_constraints, board(1, _, _, N, _, _, _) ==> N = 0.
2 bridge_constraints, board(_, YMax, _, _, E, _, _), ymax(YMax) ==> E = 0.
3 bridge_constraints, board(XMax, _, _, _, _, S, _), xmax(XMax) ==> S = 0.
4 bridge_constraints, board(_, 1, _, _, _, _, W) ==> W = 0.
```

- If on a certain position there is no island, then the amount of bridges that come in from a certain direction equals the amount of bridges that leave in the opposite direction.

```
1 bridge_constraints, board(_, _, 0, N, E, S, W) ==> N = S, E = W.
```

- The amount of bridges connected to an island equals that island's amount. Three addition constraints are used here to represent

$$Amount = N + E + S + W$$

```
1 bridge_constraints, board(_, _, Amount, N, E, S, W) ==> Amount > 0 |
2     Sum in 0..4,
3     Sum2 in 0..4,
4     add(N, E, Sum),
5     add(S, W, Sum2),
6     add(Sum, Sum2, Amount).
```

2.2.3 Propagation and search

Constraint propagation is handled by the 'in/2', 'add/3' and 'eq/2' rules. The 'in/2' rule is used to set the domain of a variable. The domain of a variable is a list of integers represented by two integers A and B and is noted as 'A..B'. When the domain of a variable only contains one element (when A equals B), then the value of the variable is known. The 'add/3' and 'eq/2' rules are used to represent the rules for addition and equality in the program. additionally, their job is to reduce the domains of variables (see lines 9, 10, and 15 in the code below). This makes these constraints active constraints. An example of the domain reduction goes as follows. Assume the following facts are known. 'X in 0..4', 'Y in 0..4', 'Z in 0..2', add(X, Y, Z). We know that $X + Y$ must equal Z, and that Z must lie between 0 and 2. Therefore, we can reduce the domains of X and Y from '0..4' to '0..2'. This kind of behaviour is implemented by line 15 in the code excerpt below.

```
1  % remove duplicate indomain constraints
2  X in A..B \ X in A..B <=> var(X) | true.
3  % when a variable's domain gets reduced to 1 number, set the value of the variable
4  X in A..A <=> var(X) | X = A.
5
6  % equality constraint
7  X eq Y <=> number(X), number(Y) | X == Y.
8  % equality domain constraint solving
9  X eq Y \ X in A..B, Y in C..D <=> A \== C | L is max(A, C), X in L..B, Y in L..D.
10 X eq Y \ X in A..B, Y in C..D <=> B \== D | U is min(B, D), X in A..U, Y in C..U.
11
12 % addition constraint
13 add(X, Y, Z) <=> number(X), number(Y), number(Z) | Z is X + Y.
14 % addition domain constraint solving
15 add(X, Y, Z) \ X in A..B, Y in C..D, Z in E..F <=>
16     not( ( A >= E-D, B =< F-C, C >= E-B, D =< F-A, E >= A+C, F =< B+D ) ) |
17         NewA is max(A, E-D), NewB is min(B, F-C), X in NewA..NewB,
18         NewC is max(C, E-B), NewD is min(D, F-A), Y in NewC..NewD,
19         NewE is max(E, A+C), NewF is min(F, B+D), Z in NewE..NewF.
```

Listing 2.3: Constraint propagation and domain solving

The actual search for values of variables is handled by the 'search/0' and 'enum/1' rules. When 'enum/1' is fired with a variable that has a domain constraint ('X in A..B'), then it will cause X to take a value in its domain. Values are assigned to X by Prolog's 'between(A, B, X)' procedure. Each time X is assigned a value by this procedure, other rules may be fired, depending on what is in the constraint store at that time. Among those other rules could lie equality and adition constraints which can now further reduce domains of other variables because the domain of X has been removed. If this causes backtracking to happen back up to the 'between' procedure, the next value in the domain will be assigned to X.

```
1 % assign values to variables X. X must lie between A and B
2 enum(X) <=> number(X) | true.
3 enum(X), X in A..B <=> between(A, B, X).
4
5 % search for constraint variables
6 search, X in _.._ ==> var(X) | enum(X).
```

Listing 2.4: Search

2.2.4 Connectivity constraint

The constraint that in the end result, the bridges must connect all islands into a single connected group, is handled in two different ways.

Version 1

The first version implements the connectivity constraint by checking whether or not all islands are reachable from a certain island. 'reachable/2' is used to represent which positions on the board are reachable. The reachable set can be expanded whenever a new bridge between two islands ('connection/2') is detected. The actual connectivity constraint is represented by lines 10 and 11 in the excerpt below. If, after the search phase is over, there is an 'island/3' fact without an accompanying 'reachable/2' fact, then the islands are not connected into a single group. This then causes backtracking to happen which causes search to try other values for variables.

```
1  % put first island in reachable set
2  island(X, Y, _) \ pick_first_island <=> reachable(X, Y).
3
4  % build up reachable set
5  reachable(X, Y) \ connected([X, Y], [A, B]) <=> reachable(A, B).
6  reachable(X, Y) \ connected([A, B], [X, Y]) <=> reachable(A, B).
7  reachable(X, Y) \ reachable(X, Y) <=> true.
8
9  % connectivity constraint: each island fact needs to have an accompanying reachable
       fact
10 connected \ island(X, Y, _), reachable(X, Y) <=> true.
11 connected, island(_, _, _) <=> false.
```

Listing 2.5: Connectivity constraint propagator 1

Version 2

The second connectivity constraint propagator follows a similar idea, but implements it by using a disjoint-set-like data structure. After the puzzle board is loaded, each island forms its own 'connected set' (or segment). A counter 'connected_sets_counter/1' represents the amount of connected sets that are on the board. 'connected_set/3' represents which island belongs to which connected set. After a bridge has been formed between two islands ('connected/2'), their connected sets are merged together (lines 1, 4, and 10 below) by changing the identifier of the smaller connected set into the identifier of the larger one. After the sets have been merged, the 'connected_sets_counter/1' is decremented. The actual connectivity constraint is represented by the last line in the code excerpt. If, after the search phase is over, there exists more than one connected set, then the solution is not valid, and backtracking is forced.

Compared to version 1, this connectivity constraint propagator is actually worse. This is because it is more costly to keep track of all of the connected sets during search than to just check whether or not all island are reachable from a certain island. However, subsection 2.2.5 contains an improvement for this version which makes it perform better than the previous one.

```
1  connected_set(A, B, Set1), connected_set(C, D, Set2), connected_set_counter(Set1,
       Count1)
2      \ connected([A, B], [C, D]), connected_set_counter(Set2, Count2) <=> Set1 \==
           Set2, Count2 =< Count1 |
3          connected_sets_union(Set1, Set2).
4  connected_set(A, B, Set1), connected_set(C, D, Set2), connected_set_counter(Set1,
       Count1)
5      \ connected([C, D], [A, B]), connected_set_counter(Set2, Count2) <=> Set1 \==
           Set2, Count2 =< Count1 |
6          connected_sets_union(Set1, Set2).
7  connected_set(A, B, Set), connected_set(C, D, Set) \ connected([A, B], [C, D]) <=>
       true.
8  connected_set(A, B, Set), connected_set(C, D, Set) \ connected([C, D], [A, B]) <=>
       true.
9
10 connected_sets_union(Set1, Set2) \ connected_set(X, Y, Set2),
       connected_set_counter(Set1, Count) <=> NCount is Count + 1 |
11     connected_set(X, Y, Set1),
12     connected_set_counter(Set1, NCount).
13 connected_sets_union(_, _), connected_sets_counter(Count) <=> NCount is Count - 1 |
       connected_sets_counter(NCount).
14
15 connected, connected_sets_counter(C) ==> C > 1 | fail.
```

Listing 2.6: Connectivity constraint propagator 2

2.2.5 Improvements

Improvement 1

Our first improvement in the CHR solution follows the same logic as the first improvement in the ECLiPSe solution. Islands with value 1 cannot be connected to one another and islands with value 2 cannot be connected by 2 bridges (because then they would form isolated segments). This is not detected automatically by the domain solving rules of 'eq/2' and 'add/3', so we have added 4 rules which reduce the domains of variables before the search phase starts.

```
1  % improvement A: island with 1 cannot be connected to other island with 1, so set
       variable to 0
2  make_domains, board(X1, Y1, 1, N, _, _, _), neighbors(X1, Y1, 'N', X2, Y2),
       island(X2, Y2, 1) \ N in _.._ <=> var(N) | N = 0.
3  make_domains, board(X1, Y1, 1, _, E, _, _), neighbors(X1, Y1, 'E', X2, Y2),
       island(X2, Y2, 1) \ E in _.._ <=> var(E) | E = 0.
4  % improvement B: 2 cannot be connected to 2 by 2 bridges, so make domain A..1
5  make_domains, board(X1, Y1, 2, N, _, _, _), neighbors(X1, Y1, 'N', X2, Y2),
       island(X2, Y2, 2) \ N in A..2 <=> var(N) | N in A..1.
6  make_domains, board(X1, Y1, 2, _, E, _, _), neighbors(X1, Y1, 'E', X2, Y2),
       island(X2, Y2, 2) \ E in A..2 <=> var(E) | E in A..1.
```

Listing 2.7: Connectivity constraint propagator 2

This improvement removes a lot of the guess work done by 'between/3', because the reduction of some of the domains can cause a domino effect to occur in which a lot of domains can subsequently be reduced. On some boards there isn't even a need for the search phase any more because the board can be solved by the domain solving of 'eq/2' and 'add/3'.
This improvement also has an impact on the connectivity constraint, as isolated sets like 1-1 and 2=2 cause the connectivity constraint to not be satisfied. This is particularly impactful for example on boards for which the solver would guess an isolated set like 1-1 right at the beginning of the search phase. In that case, backtracking would need to go over all choice points made after the variables included in the 1-1 set.

Improvement 2

The biggest flaw in the two connectivity constraint propagators was that checking the connectivity constraint only happens after a value has been assigned to all variables. The connected sets of the second connectivity constraint propagator allows us to detect isolated segments earlier during the search phase. First, 'no_isolated_segments/0' is added to the constraint store after a new value has been assigned to a variable. 'no_isolated_segments/0' is used to add 'connected_set_not_isolated/1' into the constraint store. This then makes sure that each connected set currently on the board is not isolated. A connected set is not isolated when one of the islands in the set can still create a bridge (and this is possible when one of the directions in the 'board/7' is still a variable). If no such island is found in a set, then backtracking is forced (last line in the excerpt below).

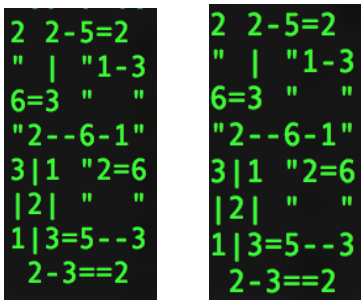
```
1  search, X in _..._ ==> var(X) | enum(X), no_isolated_segments.
2
3  ...
4
5  no_isolated_segments, connected_set_counter(Set, _) ==>
       connected_set_not_isolated(Set).
6  no_isolated_segments <=> true.
7
8  connected_sets_counter(1) \ connected_set_not_isolated(_) <=> true.
9  connected_set(X, Y, Set1), board(X, Y, _, N, _, _, _) \
       connected_set_not_isolated(Set1) <=> var(N) | true.
10 connected_set(X, Y, Set1), board(X, Y, _, _, E, _, _) \
       connected_set_not_isolated(Set1) <=> var(E) | true.
11 connected_set(X, Y, Set1), board(X, Y, _, _, _, S, _) \
       connected_set_not_isolated(Set1) <=> var(S) | true.
12 connected_set(X, Y, Set1), board(X, Y, _, _, _, _, W) \
       connected_set_not_isolated(Set1) <=> var(W) | true.
13 connected_set_not_isolated(_) <=> false.
```

Listing 2.8: Connectivity constraint propagator 2

This improvement has a large impact on the connectivity constraint, because now the connectivity constraint is checked during the search phase instead of after the search phase. If an isolated segment were to be formed early in the search phase, this improvement would detect it and cause backtracking to happen instantly, whereas without this improvement backtracking would need to go over all choice points made after the creation of the isolated set. The effect of this improvement is most visible on board 2. Its solving time went from 516s with improvement 1 to 317s with improvement 1 and 2 (see tables

in section 2.2.6). This improvement did not have a great impact on most other boards. This is because most other boards had smaller and fewer (or zero in some cases) domains to chose from during the search phase because of the domain solving done by 'eq/2' and 'add/3'.

2.2.6 Experiments



(a) board pre search

(b) board post search

Figure 2.6: Even for CHR some of the boards are completely filled in before the search (In this example: board 4)

board number	no improvements	isolation improvements
1	0.049s	0.044s
2	aborted after 1h30m	497.33s (8.3m)
3	0.026s	0.03s
4	0.054s	0.057s
5	0.05s	0.049s
6	0.121s	0.114s
8	0.006s	0.004s
9	0.006s	0.007s
10	26.642s	1.344s
11	0.044s	0.049s
12	0.05s	0.05s
13	0.056s	0.056s
14	0.097s	0.066s
15	1.584s	1.527s
16	1.11s	1.113s
17	2.152s	0.222s
18	0.035s	0.051s
19	0.01s	0.011s
20	0.034s	0.024s
21	0.012s	0.013s
22	0.028s	0.027s

Table 2.5: Improvements on CHR solver with connectivity propagator 1

board number	no improvements	isolation improvements	connectivity constraint improvement	all improvements
1	0.05s	0.053s	0.054s	0.048s
2	unknown	515.92 (8.6m)	unknown	316.37s (5.3m)
3	0.028s	0.028s	0.028s	0.026s
4	0.065s	0.059s	0.057s	0.062s
5	0.058s	0.052s	0.051s	0.051s
6	0.125s	0.119s	0.127s	0.116s
8	0.007s	0.004s	0.006s	0.006s
9	0.007s	0.01s	0.007s	0.006s
10	27.415s	1.356s	50.548s	1.34s
11	0.046s	0.047s	0.048s	0.048s
12	0.052s	0.048s	0.053s	0.052s
13	0.059s	0.059s	0.06s	0.06s
14	0.093s	0.077s	0.093s	0.076s
15	1.587s	1.613s	1.604s	1.555s
16	1.127s	1.226s	1.132s	1.143s
17	2.221s	0.232s	2.585s	0.234s
18	0.047s	0.049s	0.06s	0.039s
19	0.01s	0.011s	0.012s	0.011s
20	0.024s	0.023s	0.024s	0.029s
21	0.012s	0.013s	0.014s	0.012s
22	0.03s	0.037s	0.024s	0.026s

Table 2.6: Improvements on CHR solver with connectivity propagator 2

Conclusion

3.1 Weak points

There are a couple of weak points in our report:

- The speed of our Sudoku implementations in CHR are not that great. We did try to improve it with the heuristics, but we feel it is still too slow. We think this was more due to the fact that we were still novices with how CHR worked as our Hashiwokakero CHR implementation is quite fast.
- The implementation of the connectivity constraint in ECLiPSe for Hashiwokakero is also a rather weak point. Ideally we should have implemented it using disjoint sets like in CHR but we couldn't figure out how to do this correctly.
- We think that implementing a Hashiwokakero solver using a graph representation is a more logical approach than the one we have used, but we had problems with ECLiPSe and how to express the no crossing bridges constraint.

3.2 Strong points

The strong points in our report are:

- Our Hashiwokakero CHR implementation is quite fast thanks to our improvements set.
- We think we have good heuristics for Sudoku in CHR. Certainly the heuristic for the alternative viewpoint has made good improvements on the speed of the search.

3.3 Lessons learned

We have learned a lot of things during this project due to the fact we often had to re-track our steps. If we would now have to do an other project with these systems we think we would have a better idea for how to start and what to think about. We spent quite some time with both ECLiPSe and CHR so we now have a much better idea of what the strong points are and the weak points are for each system.

Appendix

We started working on this project before the Easter holiday. In the beginning 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 task and the ECLiPSe part of hashiwokakero before the end of the holiday. We then had to halt our work for a while since we had a deadline for a ridiculously large project for another course. As the semester was coming to an end other deadlines and an exam were coming up so we had to manage those first. Thus it was only at the start of the study period that we could continue working. 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 individual changes when they had time but most of our work was done online using Hangouts and its screen sharing functionality. 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. We have each individually put more than 100 hours in this project.