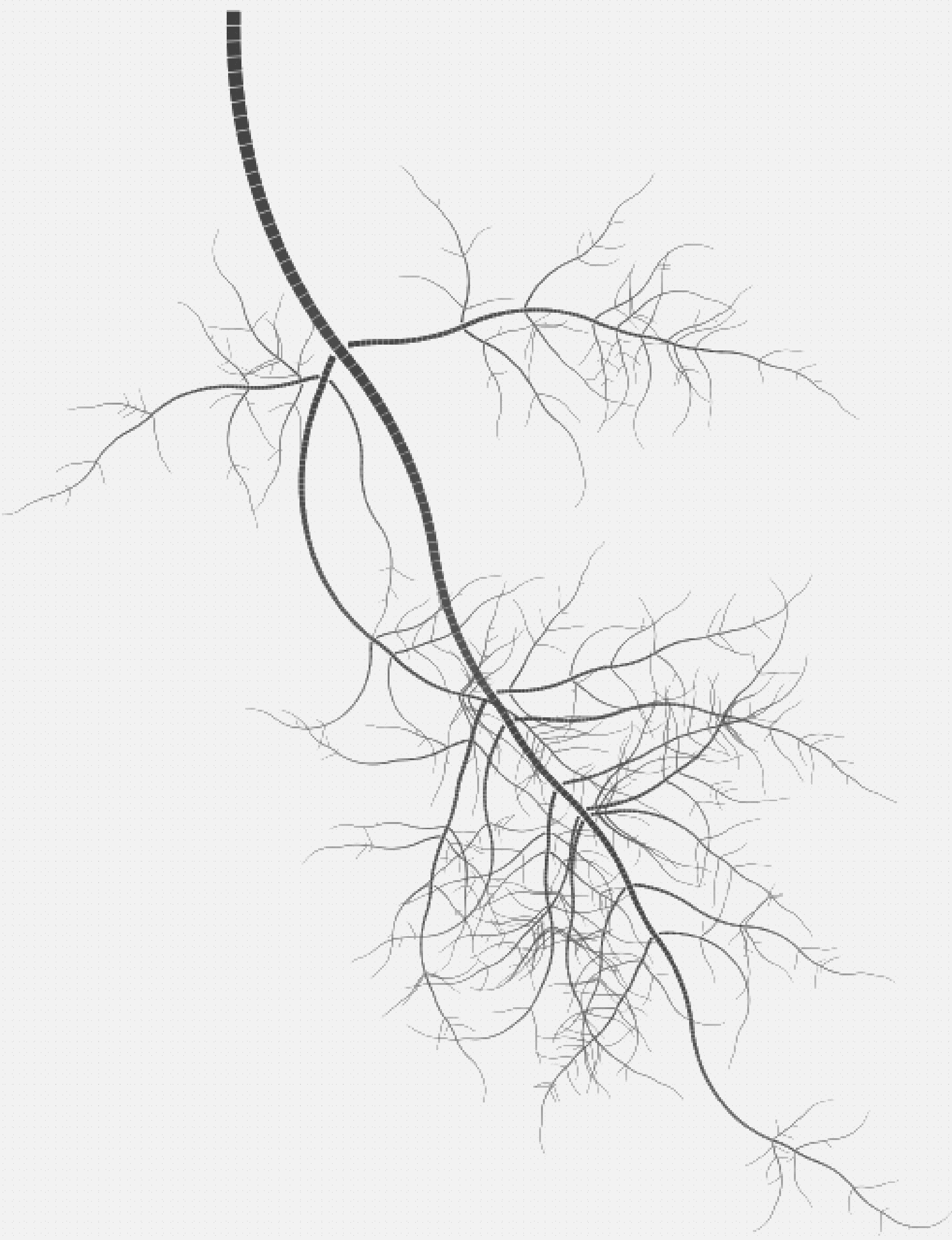
Advanced Programming Languages for A. I.

Michaël Dooreman & Bruno Vandekerkhove



Academic Year 2019

H02A8A: Advanced Programming Languages for A. I.

Contents

In the past decades much research has been done on the characterization and resolution of constraint satisfaction - and constraint optimization problems. This report discusses three challenges ; Sudoku puzzles, Hashiwokakero and a scheduling problem.

1 Sudoku

Sudoku is a well-known puzzle game which needs no introduction. It is typically modelled as a constraint satisfaction problem through the use of `all_different` constraints on rows, columns and blocks. Such global inequalities tend to improve upon the use of binary inequalities. The constraint generating code[1] is fairly trivial and needn't be detailed here.

There are several other ways one could model Sudoku. The widely cited study by Helmut Simonis [10] and subsequent studies provide some ideas. Four *'dual'* models, two approaches based on a boolean characterisation, a combination of models provided by Laburthe, a model enforcing the singular occurrence of every value in every row, column and block, as well as a model with nothing but channeling constraints were considered[2]. Tests were run on the provided puzzles[3] as well as some minimum puzzles provided by Gordon Royle[4]. These are puzzles with a minimal amount of pre-filled cells (17 to be precise [7]), which does not mean that they are harder to solve.

The dual models hold a $N \times N$ array with all the decision variables. Whereas in the classic viewpoint the rows, columns and values of this array correspond to those of the input puzzle, every one of the four dual models changes their roles. The first two switch the role of rows or columns with those of values. In the third model every row and column of the array corresponds to a block and a position. In the fourth dual model every row represents a block, every column a value and every value a position within a block. For each of them it was harder to implement the necessary constraints, usually necessitating the use of auxiliary variables together with appropriate channeling constraints.

In one of his works Laburthe discusses various rules that can be used to resolve Sudoku puzzles, after which he details three models that he associates with the rules [3]. He ends up proposing a model for every level of

[1] `ECLiPSe` and `CHR` implementations are available in `/sudoku/model/classic.pl` and in `/sudoku/chr/model/classic.pl`.

[2] These are implemented with ECLiPSe in the `/sudoku/eclipse/model/` directory, and some CHR versions are in `/sudoku/chr/model/`.

[3] `/sudoku/benchmarks/benchmarks.pl` provides automatic benchmarking code.

[4] These are available online.

difficulty of the input puzzle. An attempt was made at implementing his recommendation for ‘difficult’ puzzles. It decreased the average number of backtracks but increased the runtime.

The boolean models include the natural combined model [8] and a more intuitive characterisation resembling an integer programming - or a SAT model [6] (using `occurrences/3` instead of sums, disjunctions and conjunctions). Both of them have $N \times N \times N$ boolean variables b_{rcv} which are true if the cell at row r and column c holds the value v. The natural combined model was cumbersome to implement and performed badly. It was introduced together with an algorithm which was tailored after it, and a constraint for unequality of lists isn't really supported by ECLiPSe[5].

Note that it is usually not recommended to use a boolean model when integers can be used instead (as pointed out by Rossi [4]).

The last two models were found to be the most performant. The first makes use of the previously mentioned `occurrences/3` constraint to make every value occur just once in every row, column and block.

The second one generates nothing but channeling constraints. It has been demonstrated that this can provide good results despite such constraints being less ‘tight’ than `all_different` constraints[6]. When Dotú discussed it he was considering QuasiGroups [2]. This was extended[7] to Sudoku puzzles by making use of three instead of two dual models (since blocks need to be considered as well). The variant in which channeling constraints between all models (one primal, three dual) are generated performed better than the one in which only channeling constraints between the primal and every dual model are applied. These variants are analogous to what Dotú referred to as *‘trichanneling’* and *‘bichanneling’*.

1.1 Experiments

Number of backtracks and running time for most of the models are displayed in table 1. Removal of *‘big’* (`all_different`) constraints in the classic model[8] led to an increase in runtime which corroborates Demoen's experiences [1].

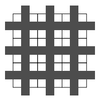


Figure 1: *Missing(6)* model

An interesting observation is that the two most performant models also have the same number of backtracks. One of them is the model with nothing but channeling constraints, the other uses the `occurrences/3` constraint which enforces arc-consistency. The first will detect when for a given *row-column*, *row-value*, *column-value* or *block-value* combination only one possible *value*, *column*, *row* or *position* remains. It will remove this value, column, ... from the domains of the other primal or dual variables[9]. The second can do the same ; it can propagate unequalities when the domain of a variable is reduced to a singleton but also knows when a value can be put in only one particular cell of a row, column or block. It is probably slower because the constraint itself is more generic than reification constraints.

A model combining the classic viewpoint with the fourth dual model was set up. Number of backtracks and runtimes are displayed in table 3.

The `first_fail` heuristic generally outperforms `input_order`. It considers variables with the smallest domain first, rather than considering variables in the order they were given. This increases pruning power as removing a value from the domain will remove a bigger part of the search tree because the branching count

[5] A custom-made implementation as well as the `~=/2` constraint which checks if two terms can be unified were tried. Channeling back to integers with `ic_global:bool_channeling/3` worked better (ironically).

[6] *“The reason for this difference is that the primal not-equals constraints detect singleton variables (i.e. those variables with a single value), the channelling constraints detect singleton variables and singleton values (i.e. those values which occur in the domain of a single variable), whilst the primal all-different constraint detects global consistency (which includes singleton variables, singleton values and many other situations)"*[5]

[7] The code lies in `sudoku/model/channeling.pl` in which a flag called `extended` can be used to opt for one of two variants.

[8] In his study Demoen gives several *Missing(6)* examples, models in which 6 of the `all_different` constraints are removed. *Missing(7)* models aren't Sudoku, and because of the stark rise in number of backtracks no further experimentation with the removal of ‘*small*’ constraints was done. The `eliminate_redundancy` flag can be used to toggle redundancy elimination on and off.

[9] The difference between unequality constraints and channeling is explained in more detail in [5]. Adding unequalities to the implementation slows down the search procedure because the channeling constraints already do this propagation and more.

Code Snippet 1: Channeling constraints for the combined viewpoint model

```
(multifor([Row,Column,Value], 1, N), param(Primal, Dual, K) do
    #=(Primal[Row,Column], Value, Bool), % reification
    block(K, Row, Column, Block), % calls utility function to convert R × C → B
    position(K, Row, Column, Position), % calls utility function to convert R × C → P
    #=(Dual[Block,Value], Position, Bool) % reification
)
```

increases as you go down the tree (which is not necessarily the case when using `input_order`). This tends to cause the number of backtracks to decrease. Unless the solution lies at the left side of the search tree generated by the `input_order` heuristic performance will increase in comparison.

As for the `all_different` constraints, more constraint propagation can be done when making use of `ic_global` since it enforces bound consistency [9] rather than enforcing arc-consistency on the corresponding inequalities. This decreases the number of backtracks for every puzzle but may increase runtime for some of them (such as *sudowiki_nb49*) because the propagation takes time. The global constraints do tend to perform a little better on average (about 11.2 versus 11.5 seconds total runtime).

Combining two viewpoints adds redundancy, leading to more propagation. The number of backtracks ends up decreasing because of this while the runtime may not necessarily decrease due to the larger number of constraints that have to be dealt with. As seen in table 3 the runtimes of the combined model generally laid somewhere in between those of the two original viewpoints.

1.2 CHR

Some of the viewpoints considered previously were implemented in `CHR`. The runtimes are shown in table 4. Initially the `first_fail` variable heuristic and the `indomain_min` value heuristic were used. These are easy to implement and generally perform quite nicely. Based on previous experiments with ECLiPSe it was concluded that (at least for the given benchmarks) `indomain_max` would be a better choice for the classic model. It cut runtime by about half, yet increased the runtime of the `dual4` model which shows that indeed, models and heuristics interact.

Because the channeling-only viewpoint performed best in the ECLiPSe experiments an attempt was made at implementing it in `CHR`. The approach that was used led to complicated code and - ironically - the slowest runtimes seen yet. After a related failed experiment[10] the focus was laid on keeping the code more simple and adding redundant constraints. A few of the rules discussed by Laburthe [3] were tested out. The *x-wing* rule was of no use, but adding a *single-position* rule decreased the runtime.

After this experiment a second attempt was made at implementing some sort of channeling-only model, or at least a model which would detect when values can only be put in one cell in a given row, column or block. By applying all the *single-position* rules, basically. While technically speaking no channeling is done in the implementation of this experimental model, the number of backtracks decreased starkly[11] and the total runtime decreased in comparison with the classic model.

In example code Gonzalez & Christiansen use a 4-coordinate approach ($a \times b \times c \times d$ with $a \times b$ designating a block, c a row - and d a column within that block). This looked like an elegant trick to try out as it makes it unnecessary to convert from *row-column* combinations to block numbers. A form of pre-processing which did decrease runtime (for the classic model) by a few seconds. Applying it in the experimental model reduced total runtime to about 4 seconds.

As in the previous section, two of the `CHR` viewpoints were combined in one single model. For every puzzle it generally performed at least as well as the worst of the two other models - with a few exceptions. Sometimes it performed better than both, because the propagation achieved by the separate models and reduction in backtracks quickened the search enough to compensate for the increased complexity of the model. Total runtime

[10] At some point a constraint `pos/4` was used to convert from *row-column* combinations to the corresponding blocks and positions (e.g. `pos(2,3,1,6)`). This slowed down the algorithm a lot because it increased the size of the constraint store. Encoding positions with 4 coordinates worked better.

[11] It's a little harder to count the number of backtracks in `CHR`. Because the channeling-only model solves the *extra2* puzzle with zero backtracks the `CHR` model was tested on that one, and it also did zero backtracking.

clearly is highest in the dual model, lower in the classic model, even lower in the combined model and lowest in the experimental model discussed previously.

	classic	dual1	dual2	dual3	boolean	laburthe	member	channeling
Total/Average	1981/142	2694/193	3467/248	2178/156	1998/143	14270/1020	1267/91	688/50
Total/Average (minimum)	2628/27	3183/32	3148/32	2612/27	2715/27	59340/594	1909/20	2026/20

Table 1: Runtimes for the various viewpoints (ECLiPSe) - shown in milliseconds. The `first_fail` and `indomain_min` heuristics were used.

	`input_order`	`first_fail`	`input_order` (global)	`first_fail` (global)
	classic - dual	classic - dual	classic - dual	classic - dual
lambda	495/4712 - 162/1715	149/977 - 72/523	22/3 - 17/0	22/3 - 19/0
hard17	132/873 - 87/389	92/419 - 68/198	20/1 - 27/1	27/1 - 29/1
eastermonster	62/119 - 75/125	58/101 - 84/155	237/51 - 236/49	169/33 - 256/66
tarek_ 052	72/193 - 68/200	62/130 - 59/137	370/61 - 324/63	224/35 - 180/34
goldennugget	103/520 - 98/441	87/358 - 90/281	687/104 - 517/72	334/76 - 264/44
coloin	368/2209 - 246/2288	61/83 - 55/80	249/88 - 675/194	79/8 - 63/10
extra2	305/4652 - 172/2894	641/7690 - 392/4232	18/0 - 18/0	19/0 - 21/0
extra3	530/4712 - 176/1715	166/977 - 93/523	22/3 - 17/0	22/3 - 19/0
extra4	1445/15116 - 409/6087	234/2097 - 139/1031	23/4 - 19/0	22/3 - 19/0
inkara2012	21/50 - 49/72	91/273 - 75/199	42/3 - 97/3	97/17 - 106/15
clue18	220/1838 - 161/1977	93/439 - 96/493	272/69 - 207/47	12/8 - 62/6
clue17	523/5520 - 329/3509	78/270 - 72/201	17/0 - 18/0	22/0 - 18/0
sudowiki_ nb28	324/2851 - 501/4555	322/2221 - 455/3015	1083/413 - 1292/421	638/297 - 875/353
sudowiki_ nb49	186/1078 - 127/749	88/655 - 137/672	246/48 - 201/40	224/58 - 275/62
Total/Average (ms)	4782/342 - 2654/190	2216/159 - 1880/135	3302/236 - 3660/262	2013/144 - 2201/157
Total/Average (backtracks)	44443/3175 - 26716/1909	16690/1192 - 11740/838	848/61 - 890/64	542/39 - 591/42

Table 2: Results for the classic and alternative `dual4` viewpoint (ECLiPSe) - shown as milliseconds/backtracks. In both models the `input_order` heuristic outperforms `first_fail` for puzzle *extra2*. This is because the latter tends to direct the search *'down the wrong alleys'* hence the relatively large number of backtracks. Since the constraints are the same the runtime increases proportionally.

	`input_order`	`first_fail`	`input_order` (global)	`first_fail` (global)
lambda	160/507	122/188	29/3	36/3
hard17	138/264	116/155	26/1	36/1
eastermonster	79/102	163/218	351/51	395/78
tarek_ 052	137/170	137/136	625/61	219/39
goldennugget	217/286	50/10	1261/104	30/0
coloin	381/1009	55/19	363/87	67/8
extra2	115/263	120/253	25/0	28/0
extra3	152/507	113/188	32/3	60/3
extra4	266/1111	153/330	40/4	38/3
inkara2012	390/36	201/177	54/3	131/19
clue18	37/730	90/62	351/69	71/8
clue17	182/462	40/1	24/0	24/0
sudowiki_ nb28	866/2581	400/713	1667/412	247/60
sudowiki_ nb49	250/564	17/252	287/49	198/46
Total/Average (ms)	3369/241	1918/137	5127/37	1575/198
Total/Average (backtracks)	8592/613	2702/193	847/61	268/19

Table 3: Results for the combined (classic + `dual4`) viewpoint (ECLiPSe) - shown as milliseconds/backtracks.

	classic	dual	combined	experiment
lambda	64	23	13	67
hard17	283	115	35	63
eastermonster	258	657	228	223
tarek_052	273	2596	604	72
goldennugget	100	3438	624	467
coloin	433	5764	633	648
extra2	2426	18	40	56
extra3	46	25	137	54
extra4	20	22	43	52
inkara2012	404	1875	248	306
clue18	35	139	619	762
clue17	913	58	44	47
sudowiki_nb28	141	1378	66	68
sudowiki_nb49	55	1289	553	169
Total/Average	5452/389	17397/1242	4004/286	3702/264

Table 4: Runtimes for the CHR Sudoku solver - shown in milliseconds. Dual corresponds to `dual4` in the ECLiPSe version. The experimental viewpoint combines the classic viewpoint with all *single-position* rules. The fact that the classic viewpoint outperforms the dual model by a large margin can partly be explained by the use of two little redundant rules that were added to quicken matching.

2 Hashiwokakero

Hashiwokakero is another Japanese logic puzzle published by the same company in which islands have to be connected by bridges. Six constraints are to be respected, the last one being the connectedness constraint, i.e. that all islands have to be connected. What follows is a discussion of an implementation of two solvers of Hashiwokakero puzzles. One written in ECLiPSe, the other in `CHR`.

2.1 ECLiPSe implementation

A partial solution by ECLiPSe's creator Joachim Schimpf was provided. It did not enforce the connectedness constraint. Joachim defines four variables for each of the input puzzle's cells. They represent the number of bridges for each of the cell's directions (north, east, south, west). Then he enforces the five first constraints :

1-2. Bridges run in one straight line, horizontally or vertically. This is enforced with equality constraints, making sure that the number of bridges for a given direction of a given cell equals the number of bridges in the opposite direction of a neighbouring cell. A total of four equality constraints for every cell except those on the border, which may only have two or three neighbours[12].

3. Bridges cannot cross other bridges or islands. This is enforced by making sure that any cell that does not represent an island either has no horizontal or no vertical bridges.

4. At most two bridges connect a pair of islands. Joachim imposes this constraint by declaring the domains of the variables to be $[0 \ldots 2]$.

5. The number of bridges connected to an island must match the number X on that island. A simple sum constraint ($N + E + S + W \ \#= X$) suffices to enforce this one.

The connectedness constraint was enforced through the use of an analogous set of four variables (FN, FE, FS, FW) per cell, denoting the *flow* for each of the cell's directions. Say the island at the upper left is said to be the sink, then if a flow can be assigned to all islands such that the sink's incoming flow equals the total number of islands minus one, the islands are sure to be connected. The net flow for each island needs to be one, for each cell it should be zero, and empty cells should have no flow. Most of these constraints can be implemented with equality constraints (the `ic` library enforces bound consistency for these), some of the others were implemented with the use of the $\Rightarrow$ ('implication') constraint.

Code Snippet 2: Constraint stating that the flow in a bridge should be zero.

```
N + E + S + W #\= 0 => FN #= -(FS) and FE #= -(FW) and FN + FE +FS + FW #= 0
```

[12]It can be noted that Joachim's code enforces both $A\#{=}B$ and $B\#{=}A$ in several cases. It has no effect on the runtimes.

All of these constraints are active, meaning that when variables are, in a sense, 'woken up', the domain of associated variables is updated accordingly.

The provided benchmarks are solved rather quickly by the solver. It generally takes a few milliseconds, even for the biggest board. If one makes use of the `most_constrained` or the `occurrence` variable heuristic, runtimes increase. The `largest` or `smallest` heuristics perform even worse. This is due to the fact that these heuristics are more likely to target flow variables first. Flow variables have larger domains and most of the values in their domain cannot partake in a solution. As a result, backtracks increase and runtimes do too.

2.2 CHR implementation

A `CHR` solver was also created. Because of the results of the previous experiments no special heuristic (such as `occurrence` or `most_constrained`) was made use of. The solver generates `island/7` and `cell/4` constraints which associates islands and cells with their variables (representing the number of bridges in a given direction) and their number (in the case of islands). Every island has a corresponding `sum/3` constraint which gets updated every time any of the variables in its list is assigned. This corresponds to forward checking. The variables represent the number of bridges for a given direction and a given cell (or island). Instead of defining all these variables separately and enforcing $V_1 = V_2$ equality constraints whenever they're supposed to be equal, shared variables are defined instead. This is done in the pre-processing step when the board is read. Because of this decision all but the second and the fifth constraint have to be enforced. The sum constraint was just described. The way the second constraint is dealt with is shown in code snippet 3. As can be seen, the `in/2` constraint (and operator) was used to update domains.

Code Snippet 3: Enforcing that no bridges can cross in `CHR`.

```
assign(Val,X), cell(_,_,X,Y) # passive ⟹ Val > 0 | Y in [0].
assign(Val,X), cell(_,_,Y,X) # passive ⟹ Val > 0 | Y in [0].
```

Two additional constraints were added to speed up the solver. The first is a generalised version of the *'4 in the corner, 6 on the side and 8 in the middle'* technique. If an island's number equals twice its number of neighbours then it should be connected with each of these neighbours by a pair of bridges. This covers some of the special cases as *'neighbour'* is defined more broadly as *'any island to which the island can still be connected with at a certain point'*[13]. As expected, it sped up the solver for all puzzles, almost cutting the total runtime by half.

The second constraint prevents isolation of some islands by stating that any two neighbouring islands, both with the number one or two, cannot be connected by that same number of bridges[14]. Only in the sixth puzzle did this speed up the search by about half a second due to the reduction in number of backtracks. Interestingly the constraint slows down puzzle two, as enforcing it changes the variable ordering leading to a stark increase in number of backtracks[15].

Some redundant constraints were already part of the basic solver. For example, if an island only has one neighbour, its sum constraint only contains one variable which can readily be assigned.

As for the connectedness constraints, both passive - and active versions were implemented. Because any initial solution generated without enforcing connectedness still tends to be connected anyways, any constraint propagation relating to flow tends to slow down the search procedure. Because of this the passive versions of the connectedness constraints outperform the active versions (having a total runtime of about 6 seconds for all six puzzles).

In the case of puzzle six no solution is found in a reasonable amount of time when the active connectedness constraints are used. This is due to the fact that the domains of the flow variables are quite large because the number of islands equals 140. While for most of these islands the flow can easily and uniquely be determined,

[13] Let's give a concrete example. Say, an island has the number four and three neighbors. Nothing could be concluded before doing any searching. If at any point during search it becomes clear that one of the three neighbors can't be connected to the island, then two pairs of bridges need to be added to form connections with the remaining neighbours.

[14] Note that in the case of trivial boards with nothing else than two such islands enforcing this constraint would prevent the solution from being found.

[15] Experiments with value heuristics (`indomain_min` versus `indomain_max`) showed that changing the heuristic can have a large impact on the number of backtracks. The same holds true for variable heuristics. Part of the reason why ECLiPSe is a whole lot faster is that it enforces bound consistency for the sum constraints. In the sum constraints of the basic `CHR` solver only forward checking is done.

```
              5 = = 4 = = 4 - - 3 = = 3
              |           |           |
              3 = 3 - 3 = 5           |
                                      3
                                      X
                                      X
          5 - - 2                     3
          X     |                     |
          5 = = 6 - - 6               |
                      X               2
                      X               |
                      6               |
                      X               3
            3 - 4     X               X
            X   X     X               X
            6 = 6 = = 7               4
                      |               X
4 - 4                 |               4
X   |                 |               X
4 = 5                 4 = 3 - 5 = 6 = 4
```

Figure 2: Puzzle 6's remnant structure after removing all *'tails'* for which the value of the flow variables is trivial to determine.

37 flow sum constraints are less trivial. Generally speaking, when a bunch of islands are connected in a circle (meaning that there is no *'tail'*, i.e. every island is connected to at least 2 other islands) the solution isn't unique. It's what happens in puzzle six, where after cutting all the *'tails'* a few circles remain (see figure 2).

If the sum constraints relating to flow would enforce bound consistency (as is the case in ECLiPSe), wrong values could quickely be filtered out.

3 Scheduling Meetings

The last challenge is the scheduling of some meetings, taking into account the preferences of the various persons involved. A constraint optimization problem where the cost is a function of the end time of the last meeting and the number of '*violations*' (people of lower rank having their meeting after that of people of higher rank). This number of violations is of secondary importance.

Weekend constraints are generated first. If a person doesn't want to meet on weekends then his or her meeting is not allowed to overlap with the first weekend that follows :

$$((S + StartingDay) \texttt{ mod } 7) + D < 5$$

In the above constraint S and D represent the start and duration of the person's meeting. Making direct use of `mod/3` leads to an instantiation error, necessitating the use of an auxiliary variable representing the result of the modulo operation.

Code Snippet 4: Weekend constraints

```
X :: 0..6, % Weekend constraints
_Q * 7 + X #= Start + StartingDay,
X + Duration #< 6
```

Precedence constraints and constraints assuring that no meetings overlap are generated last. The corresponding code is fairly trivial[16]. The fact that the meeting with the minister should come last is equivalent to adding $N - 1$ precedence constraints with N the total number of persons.

The cost function is defined as $(V_{max} \times E) + V$ where V_{max} is the maximum number of rank violations, E is the end time of the meeting with the minister and V is the actual number of rank violations for a given solution. This ensures that whenever two solutions have a different E, the solution with the smallest E will have the lowest cost (whatever the number of violations V). Yet if two solutions have the same end time E, then it's the number of violations V that will determine what solution is best.

[16] All code for this third challenge can be found in `/src/scheduling/scheduling.pl`.

Code Snippet 5: Calculating the number of violations. The cost function is defined as $MaxViolations*(StartTime_{minister} + Duration_{minister}) + Violations$

```
violations(N, Ranks, StartTimes, Violations, MaxViolations) :-
    (for(I, 1, N-1),
    fromto([], InViolations, OutViolations, ViolationList),
    param(StartTimes, N, Ranks) do
        Rank is Ranks[I],
        (for(J, I+1, N-1),
        fromto([], In, Out, List),
        param(Rank, Ranks, StartTimes, I) do
            OtherRank is Ranks[J],
            (Rank < OtherRank -> Out = [(StartTimes[I] #> StartTimes[J])|In] ;
            (Rank > OtherRank -> Out = [(StartTimes[I] #< StartTimes[J])|In] ;
            Out = In))
        ),
        append(InViolations, List, OutViolations)
    ),
    length(ViolationList, MaxViolations),
    Violations #= sum(ViolationList).
```

An additional constraint was used for the cost function, stating that it cannot be smaller than $V_{max} \times D_{tot}$ with D_{tot} the sum of all meeting durations. This makes a difference[17].

Some implied constraints were added to increase performance. In case two persons have a different rank but the same meeting duration and weekend preferences, a corresponding order on their start times can safely be imposed. This mustn't override the precedence constraints.

Table 5 shows the runtime for each benchmark. Two versions are considered ; one ensures that no two meetings overlap by imposing a $(S_1+D_1 \leq S_2$ `or` $S_2+D_2 \leq S_1)$ constraint for every such pair, the other version uses a global version of these same constraints provided by the `ic_edge_finder` library. It's clear that the global version outperforms the other one. The time it takes to propagate the constraints is usually compensated for by the reduction in nodes having to be considered due to the pruning of the search tree.

Instead of making use of implied constraints one can also tinker with the various heuristics provided by the `search/5` procedure. Some of those lend themselves to some benchmarks but not to others.

The `indomain_min` heuristic performed better than `indomain_max` as it is an optimisation problem after all, meaning that selecting the minimum starting time selects solutions with a smaller cost first. A solution with a lower cost will prune the search tree more.

	`input_order`	`input_order` + `ic_edge_finder`
bench1a	207	326
bench1b	135	180
bench1c	314	269
bench2a	3636	2110
bench2b	19269	2043
bench2c	371	366
bench3a	143	138
bench3b	386	389
bench3c	286	309
bench3d	201	333
bench3e	405	371
bench3f	506	413
bench3g	568	479
Total/Average	26422/2033	7721/594

Table 5: Benchmark results for the Scheduling Meetings challenge, shown in milliseconds.

[17] In our tests the total runtime was reduced by a factor of 4 (not when making use of the `ic_edge_finder` version).

Appendix

Reflection

When implementing the Sudoku solver in CHR we took a glimpse at Thom's implementation. It's given in his book as a solution to some exercise. Once we understood his approach we had the tendency to implement the viewpoints by making use of the same strategy ; constraints representing variables, constraints representing values, a simple implementation of the `first_fail` heuristic and forward checking. This worked well, but it wasn't the most creative thing to do. Further experimentation with other models, heuristics, redundant constraints, ... was our way to (hopefully) compensate for this.

We also would have liked to implement bound consistency in the Hashiwokakero solver for all sum constraints. Making better use of the tracer would have saved us a lot of time which could have made this possible.

Workload

Some questions were asked online after all the code had been written. There were four of them, all about Sudoku. One on how to enforce equality of lists (we already had the solution but wanted to be sure there was no better alternative). One on looping through a list which is a subscript of an array (we found the appropriate solution ourselves). One on the inner workings of `occurrences/3`. And a final one on memory usage. Aside from a quick experiment with `~=/2` no code was rewritten as a result. None of the questions mentioned Sudoku. All the viewpoints were either thought of by ourselves or come from the literature that was cited in this report.

We started working in mid-April and finished a few days before the deadline, each having worked about ??? hours in total. This includes reading (parts of) the educational material, researching, tinkering, programming, debugging and writing the report.

Overview of the Code

Folder	File	Description
`/src/sudoku/`	`utils.pl`	*Utility functions for Sudoku (CHR & ECLiPSe)*
`/src/sudoku/benchmarks/`	`benchmarks.pl`	*Automatic benchmarking code*
`/src/sudoku/benchmarks/puzzles/`	*	*Sudoku benchmarks*
`/src/sudoku/chr/`	`solver.pl`	*Sudoku solver (CHR)*
`/src/sudoku/chr/model/`	*	*Sudoku viewpoints (CHR)*
`/src/sudoku/eclipse/`	`solver.pl`	*Sudoku solver (ECLiPSe)*
`/src/sudoku/eclipse/model/`	*	*Sudoku viewpoints (ECLiPSe)*
`/src/hashiwokakero/`	`jschimpf.pl`	*Hashiwokakero solution*
`/src/hashiwokakero/benchmarks/`	*	*Hashiwokakero benchmarks*
`/src/scheduling/`	`scheduling.pl`	*Scheduling meetings solution*

Table 6: Overview of the source code.

References

[1] Bart Demoen and Maria Garcia de la Banda. Redundant sudoku rules. *TPLP*, 14:363–377, 2014.

[2] Iván Dotú, Alvaro del Val, and Manuel Cebrián. Redundant modeling for the quasigroup completion problem. In *CP*, 2003.

[3] N. Jussien F. Laburthe, G. Rochart. Évaluer la difficulté d'une grille de Sudoku à l'aide d'un modèle contraintes. *Proceedings of JFPC'06, p. 239-248*, 2006.

[4] T. Walsh F. Rossi, P. Van Beek. *Handbook of constraint programming*. 2006.

[5] Brahim Hnich, Toby Walsh, and Barbara M. Smith. Dual modelling of permutation and injection problems. *J. Artif. Intell. Res.*, 21:357–391, 2004.

[6] Inês Lynce and Joël Ouaknine. Sudoku as a SAT Problem. In *ISAIM*, 2006.

[7] Gary McGuire, Bastian Tugemann, and Gilles Civario. There is no 16-clue sudoku: Solving the sudoku minimum number of clues problem. *Experimental Mathematics*, 23:190–217, 2012.

[8] T. Pay and J. L. Cox. Encodings, consistency algorithms and dynamic variable-value ordering heuris- tics for multiple permutation problems. *International Journal of Artificial Intelligence, 15(1):33-54*, 2017.

[9] Jean-Francois Puget. A fast algorithm for the bound consistency of alldiff constraints. pages 359–366, 01 1998.

[10] Helmut Simonis. Sudoku as a constraint problem. In *CP Workshop on Modeling and Reformulating Constraint Satisfaction Problems*, 2005.