Katholieke
Universiteit
Leuven

**Department of
Computer Science**

APLAI

Assignment 2018-2019

Sander Van Driessche (r0623896)
Toon Willemot (r0596433)
Academic year 2018–2019

Contents

1 Task 1: Sudoku

1.1 A different viewpoint

For a different viewpoint on sudoku puzzles one can take as variable X_{ij}, the position (column) of the number i in the j^{th} row. Indexing starts from the top left. An example is illustrated in 1.

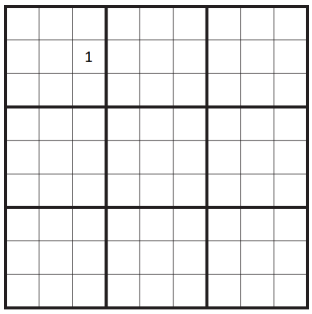

Figure 1: $X_{12} = 3$ - the position of the number 1 in the second row is the third column

In the classical viewpoint of the Sudoku puzzle, there are 81 variables, take Z_{ij} with i the row number and j the column number. The channeling constraints between the classical viewpoint and the new one are:

$$Z_{ij} = k \equiv X_{ki} = j$$

1.2 Most Challenging constraint

The most challenging constraint with this new viewpoint is the block constraint. In the classical view, we can just get all the variables belonging to a block and impose the `alldifferent` constraint. In this new viewpoint, we can formulate the block constraint like the following example. If X_{11} gets the value 1 - meaning the 1 in the first row is on column 1 - the 1 in the second row and the 1 in the third row cannot be the column 1, 2 or 3. This amounts to X_{11}, X_{12} and X_{13} having a different block number. This block number can be calculated as $(X_{ij} - 1)//3$, where // represents integer division. So, it is in the range 0 to 2 and is not unique for a block in the sudoku. Figure 2 shows a division of the Xs board into groups where the block number needs to be different.

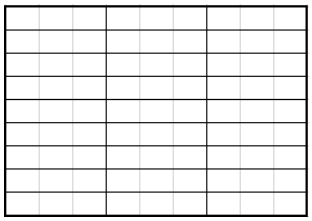

Figure 2: The blocks of Xs that need to have a different block number

1.2.1 ECLiPSe

Since in ECLiPSe integer division constraints are not really supported outside of coroutine mode, these constraints are done via multiplication. The constraint exists of two parts. The first part of the new block constraint is simply a loop over all 81 different X_{ij} where for each position, the constraint that the X_{ij} is in the block Y_{ij} is formulated as 2 lesser than constraints[1]:

[1] In the code snippets all array and matrix indexing starts from 1.

```
    ( multifor([I,J],1,N,1), param(Ys,Xs,NN) do
        Xs[I,J] - NN * Ys[I,J] #=< 3,
        Xs[I,J] - NN * Ys[I,J] #>= 1
    ),
```

where `NN` is the width of a single block (3 in classic puzzles).

The second part of the new block constraints uses the `alldifferent` constraint on these block numbers, in the groups given by fig. 2.

```
    (for(I,1,N), param(Ys,NN,N) do
        (for(J,1,N - NN + 1,NN), param(Ys,NN, I) do
                alldifferent(Ys[I,J..J + NN - 1])
        )
    ).
```

1.2.2 CHR

Because the solver is self-made in CHR, it is possible to handle the block constraint differently. The CHR constraint `inDifferentBox` is set for each pair of X_{ij} in the groups shown in fig. 2. The following code shows one part of the implementation. The other part, the symmetric counterpart for $X1$ and $X2$, is not written for brevity.

```
inDifferentBox(X1,X2,NN), X2 in L <=> nonvar(X1) |
    groupCols(X1,NN,GroupCols),subtract(L,GroupCols,LNew),X2 in LNew.
```

When X1 gets instantiated, the group columns of X1 get calculated and these group columns get removed from the domain of X2. This takes care of the block constraint for CHR.

Example: If X1 gets assigned value 7 - meaning the value is in column 7 - the group columns are 7,8,9.

1.3 Channeling approach

1.3.1 ECLiPSe

In ECLiPSe, channeling is possible via the reified constraints. Since the solver can deal with equalities the next code snippet handles the channeling constraints.

```
(multifor([I,J,K],1,N), param(Board,Xs) do
    #=(Board[I,J],K,B),
    #=(Xs[K,I],J,B)
).
```

1.3.2 CHR

For CHR, the channeling constraints are self-implemented. Both directions (to and from the second viewpoint) get their own constraint, since using a free variable in `nth1` actually binds this and creates a choice point. Setting channeling then creates constraints in both directions. The relevant code:

```
channelToXs(Xs,X,Y,K) <=> nonvar(K) | get(Xs,Y,K,X).
channelToBoard(Board,I,J,K) <=> nonvar(K) | get(Board,K,I,J).

setChannelings(Board,Xs):-
    length(Board,N),
    for(1,N,1,
        for(1,N,1,
            setChanneling(Board,Xs)
        )
    ).
setChanneling(Board,Xs,X,Y):-
```

```
get(Board,X,Y,K),
channelToXs(Xs,X,Y,K),
get(Xs,Y,X,E),
channelToBoard(Board,Y,X,E).
```

1.4 Experiments

1.4.1 ECLiPSe

For the benchmarking, the benchmarking code in the file was used. For each puzzle and method, the runtimes were captured three times and the average was taken. The predicate `statistics/2` with argument runtime was used to measure CPU time.

Comparing the two viewpoints reveals that neither of them is really better. Both have puzzles where they outperform the other viewpoint. This is to be expected, since both viewpoints have the same constraint power, and it depends on the puzzle which one outperforms the other.

Figure 3: ECLiPSe benchmark

From the results, it is clear that both using the `alldifferent/1` predicate from the `ic_global` library and using channeling drastically decreases the amount of backtracks. This is to be expected, since either increasing the constraints or increasing the propagation leads to less wrong guesses. The effect on the runtime is not as positive though, as both have puzzles where one performs much better than the other. This is the consideration of backtracking versus propagation.

Another conclusion is that, generally, the first fail heuristic decreases both the number of backtracks and the runtime. There are exceptions, though. This is also expected. The input order heuristic is pretty much random in its efficiency. At the start both a very good choice or a very bad one can be chosen. If the first thing happens, first fail

has a hard time beating input order. However, most of the time, first fail is better. This indicates that first fail is a better heuristic in general.

Adding the channeling constraints further decreases the number of backtracks. This is also normal, as added constraints remove wrong guesses. In general, the channeling approach performs the best of the three approaches. This is likely because it combines the propagation of both approaches. Again, there are exceptions though. This is when the added constraints don't lead to removal of (the right) options, and only serve as extra weight.

1.4.2 CHR

1.4.2.1 Variable and value heuristics

For the variable heuristic, the used heuristic is input_order. This follows from the definition of the `enum` predicate that initiates the search. Consequently, the first value is taken.

For the value heuristic, the used heuristic corresponds to in_domain. Values are tried from low to high, but not removed on backtracking.

1.4.2.2 Comparison of viewpoints

In almost every puzzle, the alternative viewpoint outperforms the classical one, both in time and in inferences. This is probably because of the implementation of the block constraints in the second puzzle. This decreases the amount of CHR constraints considerably, since with the classic viewpoint, 35 `ne` constraints are needed per block. In the alternative viewpoint, this decreases to 9. The constraint removes more values from the domains. Since there are less constraints, the CHR search times and wake times are decreased, thus decreasing the overall time needed.

This is confirmed by the performance of the channeling constraints: more like the classic viewpoint. Note that we removed the row and column constraints of the second viewpoint in the channeling approach, thus keeping the amount of additive constraints low. Adding these would increase the runtime and amount of inferences even more. The benchmarks are available in figure 4.

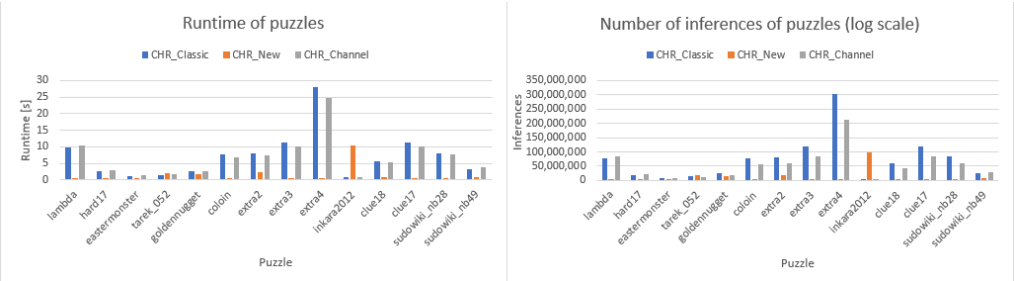

Figure 4: CHR benchmark

2 Task 2: Hashiwokakero

2.1 ECLiPSe

2.1.1 The load_puzzle predicate

The `load_puzzle(Id, Puzzle, NbIslands, Sink)` predicate translates between the input format used in `benchmarks.pl` and the one used in the provided partial solution. It is also used to pass the amount of islands *NbIslands* in the puzzle and assign the role of sink to an island *Sink*. Currently this will always be the top leftmost island.

2.1.2 Basic constraints

The distinction between celltypes (non/island) is made based on the variable *Sum* of the cell. It is the total number of bridges that are connected to the island. So, only islands have a *Sum* value greater than zero. All of the following constraints are active as they alter the domains of the constraint variables.

- **Bridges can only run horizontally or vertically in one straight line**

  These same constraints also express that bridges must be connected to islands on both sides, as the number of connections to a border is 0.

```
% Connections on one side of a cell imply the same amount of connections
  on the opposite side of the adjacent cell. (for all cells)
  (I>1    -> BN #= NESW[I-1,J,3] ; BN=0)
  (I<Imax -> BS #= NESW[I+1,J,1] ; BS=0)
  (J>1    -> BW #= NESW[I,J-1,2] ; BW=0)
  (J<Jmax -> BE #= NESW[I,J+1,4] ; BE=0)
% Only for non-island cells:
  BN=BS, BE=BW
```

- **Bridges cannot cross other bridges or islands**

```
% Only for non-island cells:
(BN#=0) or (BE#=0)
```

- **At most two bridges connect a pair of islands**

```
% For island cells: (indirectly also for other cells)
[BN,BE,BS,BW] #:: 0..2
```

- **The number of bridges connected to each island must match the number on that island**

```
% For island cells:
BN+BE+BS+BW #= Sum
```

2.1.3 Connectedness constraints

A flow-network approach is taken to implement the connectedness constraint. Flow is represented in a similar fashion as the bridges[2]. An additional four constraint variables are created for each grid cell (`FN, FE, FS, FW`). A *Sink* is appointed and the total number of islands *NbIslands* is passed on. Connectedness is then implemented by the following active constraints:

- **The net flow arriving at the *Sink* is equal to *NbIslands*-1**

```
FE+FN+FS+FW #= NbIslands-1
```

[2] Flow convention: positive = incoming; negative = outgoing

- **Every non-sink island generates 1 unit of flow**

```
FE+FN+FS+FW #= -1
```

- **A non-island cell generates no net flow**

```
FN #= -FS, FE #= -FW
```

- **If a non-island cell has a flow of n going in a given direction, then the neighboring cell in that direction should have a flow of n going the opposite direction**

```
(I>1    -> FN #= -NESW[I-1,J,7] ; FN=0)
(I<Imax -> FS #= -NESW[I+1,J,5] ; FS=0)
(J>1    -> FW #= -NESW[I,J-1,6] ; FW=0)
(J<Jmax -> FE #= -NESW[I,J+1,8] ; FE=0)
```

- **You can't have any flow in a cell which contains no connections (If a flow is present, there must be a connection)**

```
(BN #\= 0) or (FN #= 0)
(BE #\= 0) or (FE #= 0)
(BW #\= 0) or (FW #= 0)
(BS #\= 0) or (FS #= 0)
```

It is important to realise that while the absence of a connection implies the absence of flow, the presence of a connection does not necessarily imply the presence of flow. Also, a valid solution can have different valid flow configurations. This is illustrated in fig. 5. For this reason a cut is applied after obtaining a solution. (The program can thus only return a single solution, even if multiple exist, which is the case for puzzle 1.)

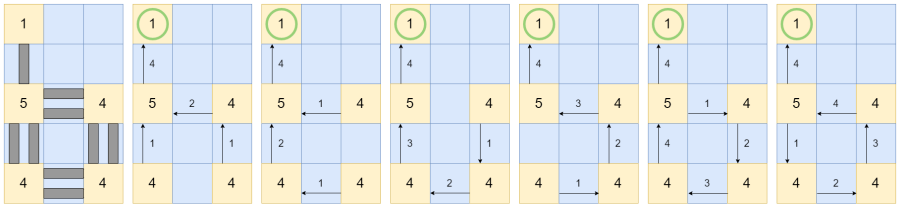

Figure 5: Different flow configurations for the same solution. Also note that not all bridges direct flow.

2.1.4 Search strategies

A couple of configurations of the generic `search/6` predicate (see `eclipse-documentation/search-6`) were tested. They can be found in section *search strategies* of the code . The best results were obtained with the `first_fail` selection method. The worst results were obtained with the `input_order` selection method. These results are visualised in fig. 6.

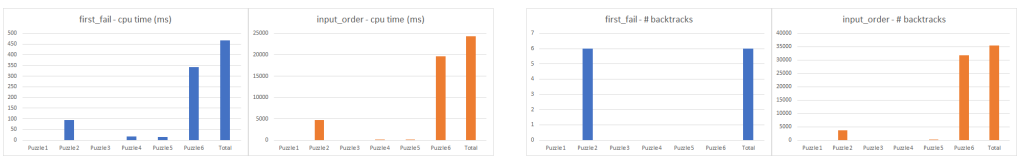

Figure 6: First-fail selection method vs. Input-order selection method.

The huge difference in performance can most likely be attributed to the fact that all the constraints are active. When a value is assigned, a lot of propagation will occur. When selecting a variable with the smallest domain, it is more likely that a valid assignment is in fact correct, eliminating the need for backtracking.

2.2 CHR

2.2.1 Data representation

Each cell of the board is represented by a `board(X,Y,Sum,BN,BE,BS,BW)` constraint. The parameters can be interpreted the same way as in the ECLiPSe representation. This time, no flow algorithm is used for the connectedness constraint. Instead, an approach based on segment isolation is used.

2.2.2 A basic solver

2.2.2.1 The flow of the solver

We use several predicates to control the flow of our constraint store. First, `load_puzzle/1` generates the correct `board/7` constraints and some additional constraints which will be discussed later on. This includes assigning the correct *Sum* to the various islands. After these constraints are generated we use `bridge_constraints/0` to activate the basic constraints. Next, we use `additional_constraints/0` to activate the additional constraints. Then `make_domains/0` sets the remaining domains. Meanwhile, the constraint propagation constraints discussed in §2.2.2.3 fire whenever possible. This process results in a partially solved board (which sometimes is already solved!). Finally, `search/0` will start the finite domain solver and resolve the final unknowns while verifying connectedness after each assignment through `no_isolated_segment/0`.

2.2.2.2 Basic constraints (active)

These constraints are very much the same as in the *ECLiPSe* implementation. However, a couple of additional constraints are also defined. These additional constraints are inspired by *hashi techniques*.

Bridge constraints

- The total number of connections must equal the island number.

```
bridge_constraints, board(_,_,Sum,BN,BE,BS,BW) ==> Sum > 0 |
    S in 0..4,  add(BN, BE, S),
    Ss in 0..4, add(BS, BW, Ss),
    add(S, Ss, Sum).
```

- Non-island cells have the same amount of connections on adjacent sides.

```
bridge_constraints, board(_,_,0,BN,BE,BS,BW)
    ==> BN = BS, BE = BW.
```

- Non-island cells have either horizontal or vertical connections. Not both.

```
board(_,_,0,BN,BE,_,_) ==> number(BN), BN > 0 | BE = 0.
board(_,_,0,BN,BE,_,_) ==> number(BE), BE > 0 | BN = 0.
```

- Adjacent cells have the same number of connections on opposite sides.

```
bridge_constraints, board(X,Y,_,BN,_,_,_), board(Xx,Y,_,_,_,BS,_)
    ==> Xx =:= X-1 | BN eq BS.
bridge_constraints, board(X,Y,_,_,BE,_,_), board(X,Yy,_,_,_,_,BW)
    ==> Yy =:= Y+1 | BE eq BW.
```

- Connections to the borders are not possible.

```
bridge_constraints, board(1,_,_,BN,_,_,_)
    ==> BN = 0.
bridge_constraints, board(_,1,_,_,_,_,BW)
    ==> BW = 0.
bridge_constraints, board(_,Border,_,_,BE,_,_), border(Border)
    ==> BE = 0.
bridge_constraints, board(Border,_,_,_,_,BS,_), border(Border)
    ==> BS = 0.
```

Additional constraints These additional constraints are based on the isolation techniques which state that some connections directly create isolated segments which of course violates the connectedness constraint. By constraining these before the search starts we can reduce even more domains, lowering the need for search.

- Isolation of a two-island segment. This entails `1-1` and `2=2` connections.

```
% 1-1 connections are impossible
neighbours(X,Y,Xx,Y), island(Xx,Y,1), board(X,Y,1,_,_,BS,_)
    ==> var(BS) | BS = 0.
neighbours(X,Y,X,Yy), island(X,Yy,1), board(X,Y,1,_,BE,_,_)
    ==> var(BE) | BE = 0.
% 2=2 connections are impossible
neighbours(X,Y,Xx,Y), island(Xx,Y,2), board(X,Y,2,_,_,BS,_)
    ==> var(BS) | BS in 0..1.
neighbours(X,Y,X,Yy), island(X,Yy,2), board(X,Y,2,_,BE,_,_)
    ==> var(BE) | BE in 0..1.
```

Two examples where these additional constraints help are puzzle 1 (`1-1 isolation`) and puzzle 2 of the benchmarks (`2=2 isolation`).

2.2.2.3 Propagation constraints

The `in/2, eq/2` and `add/3` constraints are used for propagation. These propagation constraints are based primarily on slides 6.64 to 6.70 (*CHR finite domain solver*). The domain constraint `X in A..B` means that the variable `X` takes its value from the given finite domain `[A..B]`. These domains are then reduced by the `add/3, eq/2` constraints.

```
?- 3 in 3..3, Y in 0..5, Z in -10..10, add(3,Y,Z).
add(3, Y, Z),
Z in 3..8,
Y in 0..5,
3 in 3..3.
```

Figure 7: An example of active constraint propagation by `add/3` and `in/2`

Naturally, once a domain is reduced to the form `[A..A]`, one can conclude `X = A`. See the *constraint solving expressions* section of the attached code for the implementation[3].

2.2.2.4 Connectedness constraints (passive)

Whereas the additional constraints actively manipulate some domains, they can only check 2-island segment isolation. This idea of segment isolation is extended upon now. That is, during the search phase, segment isolation is checked after each assignment using `no_isolated_segment/0`. If an assignment leads to an isolated segment, backtracking is forced right then and there. This is not an active constraint method as it does not

[3]This is not included in the report as it is copied from the slides

directly affect any variable domains. It does however significantly increase the speed of the program.

The `additional_constraints` discussed previously obviously enforce the same ideology, but in an active way. Thus reducing the need of searching for variable assignments.

2.2.3 Search strategy

The `enum/1` constraint handles the search procedure.

```
% 2 alternative value selection methods were experimented with
enum(X), X in A..B <=> smallest_first_between(A, B, X).
enum(X), X in A..B <=> largest_first_between(A, B, X).
```

The `search/0` constraint activates this constraint. Inspired by the results of the ECLiPSe implementation, the `first_fail` selection method was simulated. Assigning values to the variables with certain domains in order (smaller → larger).

```
% Example variable selection order
search, X in 0..1 ==> enum(X).
search, X in 1..2 ==> enum(X).
search, X in 0..2 ==> enum(X).
% input\_order variable selection order
search, X in A..B ==> enum(X).
```

Playing around with this order grants the results in table 1.

Puzzle	0..1-1..2-0..2	1..2-0..1-0..2	0..2-0..1-1..2	0..2-1..2-0..1	0..1-0..2-1..2	1..2-0..2-0..1
1	145	141	146	139	139	140
2	3136	6273	3616	3621	3053	6272
3	65	65	65	65	67	68
4	153	144	142	144	145	143
5	474	487	367	357	470	486
6	28874	14479	13398	13428	29284	14387

Puzzle	0..1-1..2-0..2	1..2-0..1-0..2	0..2-0..1-1..2	0..2-1..2-0..1	0..1-0..2-1..2	1..2-0..2-0..1
1	140	139	139	139	137	142
2	13579	9811	11007	11030	13624	9722
3	67	66	66	69	65	64
4	145	143	144	146	149	142
5	779	386	560	562	776	385
6	27437	31396	30483	30406	27450	31495

Table 1: CHR puzzle execution time (ms): smallest_value selection (top) and largest_value selection (bottom). For each puzzle the best result is highlighted.

From these tables we can conclude that our CHR implementation is easily outperformed by the ECLiPSe implementation. This can mainly be attributed to the fact that the ECLiPSe implementation uses active constraints for connectedness. The segment method which we implemented in CHR requires a lot of rule firing and checking at many points in time and backtracking is still used alot (whereas the best ECLiPSe implementation barely backtracks).

Table 1 shows that the best results for puzzles which actually require search (puzzles 2, 5 and 6) were obtained by selecting variables with domains starting at 0 and using the `smallest_first_between` value selection method. This makes sense. In general, knowing whether there is *any* connection at all provides the most information and this is checked the fastest by testing the absence of a connection (1 check rather than 2).

Comparing these orderings with `input_order` variable selection shows that backtracking occurs much faster with our order imposed. That is, noticeably fewer substantial backtracks over multiple variables occur.

3 Task 3: Scheduling meetings

3.1 Code

The code with the required predicate is attached to this report. Aside from the given constraints, it also uses a symmetry breaking constraint for meetings of equal rank, same weekend allowance and no precedence constraints. This decreases the runtime, because the added constraints remove the symmetry in the solution space, and thus decrease its size.

3.2 Weekend constraints

The weekend constraint implementation starts with a simple `toWeekDay` predicate. This predicate works with constraints to get the weekday (0 - 6, as mentioned in the assignment), from the day and the starting day, or sets the correct constraints if used in reverse.

The `setWeekendConstraint` predicate sets the weekend constraint for a single meeting. If the meeting is going on in a weekend, the `AllowedInWeekend` term must be set to 1 or the constraint will fail.

This predicate is then performed for all the meetings and sets the weekend constraint using the previous predicate.

The mentioned predicates are described below.

```
setWeekendConstrait(Start,Duration,StartingDay,0):-
    WeekDay :: 0 .. 6,
    WeekDay #= Start + StartingDay - _ * 7,
    5 - WeekDay #>= Duration.
setWeekendConstrait(_,_,_,1).
```

3.3 Cost

As the cost function must give precedence to the ending time and there is a maximum number of violations, it is possible to 'interleave' the ending time with the number of precedence violations. Thus, if a single day of ending time is worth more than the maximum number of precedence violations, the cost function is correct. By first calculating the maximum violations numbers, and then updating the cost function, this function remains correct.

```
% Calculate the maximum number of violations
maxNbOfViols(Ranks,I,Acc,Max) :-
    length(Ranks,Len),
    Len >= I,
    nth_value(Ranks,I,Own),
    getNbLarger(Ranks,Own,0,Nb),
    NewI is I + 1,
    NewAcc is Acc + Nb,
    maxNbOfViols(Ranks,NewI,NewAcc,Max).

maxNbOfViols(Ranks,I,Acc,Acc) :-
    length(Ranks,Len),
    Len < I.

getNbLarger([],_,Acc,Acc).
getNbLarger([L|Tail],E,Acc,Nb):-
    L > E,
    NewAcc is Acc + 1,
    getNbLarger(Tail,E,NewAcc,Nb).
getNbLarger([L|Tail],E,Acc,Nb):-
```

```
    L =< E,
    getNbLarger(Tail,E,Acc,Nb).

meeting(...)
    ...
    maxNbOfViols(RankList,1,0,Max),
    MaxViols is Max + 1,
    ...
    Cost #= MaxStart * MaxViols + Viol,
    minimize(labeling(Start),Cost).
```

From the above code, the cost function is `MaxStart * MaxViols + Viol`. Note that `MaxViols` is one higher than the maximum number of violations, such that a planning with N days and the maximum number of violations is always worse than a planning with N - 1 days and any number - even the maximum amount - of violations.

3.4 Comparison of constraints

The runtimes (CPU time) of each benchmark for the branch and bound disjunctive and the self made disjunctive is given in the following table in ms.
Some benchmarks are only run once because of the long runtime. Two tests did not finish within two hours, and have no data. It is clear that the self-made constraint is a lot slower because of the much weaker propagation of the constraints. This constraint can be found in the code, added to this report.

benchmark	run1	run2	run3	avg	run1 (self)	run2 (self)	run3 (self)	avg (self)
bench1a	250	266	266	260.67	438	469	453	453.33
bench1b	93	79	78	83.33	281	281	281	281
bench1c	938	937	938	937.67	1578	1531	1547	1552
bench2a	24031	36171	24078	28093.33	61063	60906	62594	61521
bench2b	15625	15672	15656	15651	2053172			2053172
bench2c	219	203	219	213.67	4687	4688	4687	4687.33
bench3a	219	218	219	218.67	219	219	234	224
bench3b	313	328	344	328.33	313	313	343	323
bench3c	4171	4204	4171	4182	5407	5375	5375	5385.66
bench3d	4016	4047	4062	4041.67	10437	10453	10422	10437.33
bench3e	48469	48437	48438	48448	222703			222703
bench3f	3559000			3559000	7793156			7793156
bench3g	2110	2125	2109	2114.67	4693625			4693625

4 Conclusions

We divide the conclusions into part according to the assignment they stem from.

4.1 Sudoku

From the Sudoku assignment, it is apparent that the implementation has a dramatic impact on the runtimes. Even when the viewpoint is very good, if the implementation is not efficient the result will not be workable.

A first example of this is the amount of variables. In a first iteration, not presented in the report, the block constraint was implemented very differently in CHR, but more along the lines of ECLiPSe. This was a very inefficient approach for CHR because the number of variables, and along with it the number of constraints woken up each time, was double the current value. This had the effect that some puzzles would have a very long runtime. This has now been changed to remove the values immediately from the domains of the necessary variables, and is now a strong point of the solution.

A second example of this phenomenon is the constraint propagation in ECLiPSe. When another CHR implementation was written, attempts were made to do the same in ECLiPSe and work with integer division. But, as was discovered quickly, constraints with integer division were only supported in coroutine mode, and these propagated very slowly. So, the multiplication approach was kept, and can be considered as a strong point of the ECLiPSe implementation.

A further conclusion from the Sudoku assignment was the effect of heuristics on the solving time. Not using these heuristics generally means a longer runtime, but there are exceptions, and that is why they are called heuristics: They can be wrong.

A last conclusion is that stronger propagation does not always lead to a shorter runtime, certainly when this propagation itself costs a lot of time, as is evident from the solving times.

In the CHR implementation we don't use heuristics and simply use input order, both for value selection and variable selection. This is a significant weakpoint.

4.2 Hashiwokakero

The most notable conclusion from the ECLiPSe implementation is the importance of heuristics during the search phase. As all constraints are active, a single assignment gives rise to a lot of constraint propagation. By picking variables with the smallest domains first, we are more likely to invalidate wrong values through the constraint propagation more quickly, preventing a lot of backtracking.

The ECLiPSe implementation vastly outperforms the CHR implementation. This is probably mainly due to the fact that our CHR implementation does not have an active connectedness constraint method. However, by verifying connectedness whilst backtracking we have saved a lot of execution time.

We must conclude that our CHR implementation is rather weak, and a flow based approach might have been more efficient. However, making the comparison was also interesting.

4.3 Planning

The last assignment makes the impact of the size of the search space on the runtimes very clear. E.g. benchmark 3g has the same number of meetings as benchmark 3f, but 3f allows for meetings in weekends. This increases the search space size, and thus the run time dramatically.

Another conclusion is the large impact of the propagation strength on the runtimes, as evidenced by the runtimes with the self designed disjunctive constraint versus the ones using the predefined one.

A first strong point of our solution is the symmetry breaking constraint for meetings with no precedence constraints and the same priority. As it doesn't matter in which order they are planned, removing these options dramatically decrease the search space size. The result, in accordance with the first conclusion of this assignment, is a much shorter runtime.

Another strong point of the implementation is the way to find the maximum number of days, essentially by simulating a dumb solution. This was improved from assigning a whole week to every meeting smaller than 7 days, 2 weeks for meetings with a duration from 7 to 14 etc, and brought a much needed improvement in runtimes.

This brings us the the weakpoint: even with the clever tricks described above, the runtime is still long, and definitely not suited for production level products.

A Workload of the project

The workload of the project was OK. Mainly thanks to the late deadline. An earlier deadline would clash with deadlines such as software architecture and Multi Agent Systems. Another option is a stronger division into parts, with a different deadline for each. This relieves the workload from the end of the semester.

B Task allocation

As we both take a variety of different courses, each with their own deadlines, it was tough to find the time to work on the project together. For this reason we decided to split the tasks between us, working on them whenever we found the time and asking each other questions whenever we ran into problems. This way, we both worked intensively with ECLiPSe as well as CHR, and we both encountered issues from which we learned.

- Sander Van Driessche:
  - Sudoku assignment
  - Planning assignment
  - Report on Sudoku and Planning assignment
  - Check and revise report on Hashiwokakero assignment
  - Tolinto registration for oral examination
- Toon Willemot:
  - Hashiwokakero assignment
  - Report on Hashiwokakero assignment
  - Check and revise report on Sudoku assignment
  - Check and revise report on Planning assignment
  - Toledo submission of the report and attached code
  - Hand in of the printed version