Advanced programming languages for A.I. Assignment 2017-2018

Chinjung Kuo, Kaiyuan Wei, MAI

June 1st 2018

1 Sudoku

1.1 Classical Viewpoints and Programs

The classical viewpoint for Sudoku states that all numbers in a row must be all different, and that all numbers in a column must be all different, and all numbers in a block must be all different. According to these rules, the constraints were generated by $all\_different/3$

1.1.1 Impact of variables heuristic and $all\_different/1$ from the libraries ic and ic_global.

The runtime is the summation of the computation time of ***heuristic*** and the searching time until a solution is found. On the other hand, input order will solve the problem by trying to ground the first variable in the variable list. The first fail calculates the heuristic and then will start with the variable with smallest-remaining-domain. Generally, forcing the program to fail first could have less number of backtracks. Hence, the first fail could have better performance.

IC_global ensures more reasoning between different constraints. It could consider all numbers in a row, in a column, in a block at the same time to comply with the constraints. Three dimensional perspectives suddenly become one dimensional perspective. This point of view makes reasoning more efficient and the domain reduce faster. From counting the backtrackings, the observation shows that the number of backtracks largely decreases for 1-2 orders.

There are pros and cons between the two libraries. Firstly, $alldifferent/3$ in IC_global has a higher cost on reasoning, because it shrinks the domain of each variable before assigning the value. However, reasoning globally sometimes can slow down the program. If it is known that there are lots of backtrackings, it would be better to spend computation time on global reasoning instead of searching all the backtrack points. In general, library(IC_global) is still preferred to apply in most of the cases as this could result in good combination of speed and reasoning.

1.1.2 Impact of input order vs. first fail using IC on puzzle extra2

From Table. 1, the backtracking comparison is input order (4652) v.s. first fail (7690). Part of simulation process is shown below:

Initial condition: `A = [[_, _, 1, _, 2, _, 7, _, _], [],]`

Searching domain: `A = [[_{[3, 4, 6, 8, 9]}, _{[3, 4, 6]}, 1,...`

Real solution: `A = [[3, 6, 1, 9, 2, 8, 7, 5, 4], [],]`

The solution of the first two variables happens to be the first number in its domain. Coincidentally, by following input order, the variable can be grounded faster to the real solution. Thus, in this case, the backtracking was only half by using input order than using first fail. As a result, wasting computation time on heuristic(first fail) induced longer runtime.

1.1.3 Explain the effect of IC_global on the run-times and the number of backtracks for sudowikinnb49

From Table 1, library(IC_global) shows that after reasoning globally there were 48 backtrackings. On the contrary, instead of reasoning globally, library(IC) reason locally with 1078 backtracks. However, the runtime of IC is even better, 0.11s to 0.16s. This is because IC_global spend more time on global reasoning in each backtracking. Due to the number of backtracks not shrinking enough for the compensation from time spending on reasoning, the runtime was longer for IC_global. T

The same observation happened in puzzle-eastermonster with the same reason. The number of backtrackings under using IC_global library (33) was still 3 times less than using IC library (101), but the runtime was longer. Consequently, if using the heuristic can significantly reduce the number of backtrackings, it is worth to make the calculation. Otherwise the calculation may cost much more than the backtracking. Since the calculating time is a linear cost and the time for backtracking can be exponential increased, the IC_global library is still preferred.

Sudoku	**inputorder or first fail**	**IC_global**		**IC**	
		Runtime(s)	**#backtrack**	**Runtime(s)**	**#backtrack**
lambda	input_order	**0.00**	3	0.28	4712
	first_fail	**0.00**	3	0.06	977
extra2	input_order	**0.02**	0	0.16	4652
	first_fail	**0.02**	0	0.42	7690
sudowiki_nb49	input_order	0.16	48	**0.11**	1078
	first_fail	0.16	58	**0.06**	655
extra4	input_order	**0.02**	4	0.86	15116
	first_fail	**0.02**	3	0.19	2097
hard17	input_order	**0.02**	1	0.06	873
	first_fail	**0.00**	1	0.05	419
eastermonster	input_order	0.19	51	**0.02**	119
	first_fail	0.09	33	**0.02**	101

Table 1: The comparison between IC and IC_global on six puzzles and searching in two different heuristics. The marked letters are the better ones.

1.2 Task 2: Sudoku in alternative viewpoint

1.2.1 Classical Viewpoints and Programs

A new matrix with a size of 81, Q, represented the new viewpoint: Q[Number, Block]=Index. The Index is the variable, which represents the position in the puzzle. The Number is the value filled in this position, and the Block represented which block it belonged to. The goal was to assign all the indices for each number and block.

Because all 81 Q, Q should have a different index, the first constraint is all the indices needed to be different and in domain `[1..81]`.

The second constraint came from the block. For each Q, the Block's value can also reduce the range of its index. This constraint was implemented by the code, in which K and L represented the Row and Column of the block:

$(multifor([K,L],1,3), param(Q)$

do

$AC\ is\ Q[*,K*3+L-3], AC\ ::\ [$

$K*27+L*3-29, K*27+L*3-28, K*27+L*3-27,$

$K*27+L*3-20, K*27+L*3-19, K*27+L*3-18,$

$K*27+L*3-11, K*27+L*3-10, K*27+L*3-9])$

For the third constraint, as index can represent the row and column, all the Q[Number,Block] with the same Number should be in different rows and different columns. The columns and rows were calculated by the following equation:

$Q[I,J]\# = Column*Size+Row-Size$

The implementation of this constraint turns out to be very challenging. If the Q is still a variable, this will result in an instantiation error. However, if it is implemented after all the Q are grounded, it will become a backtrack point and the rest constraints proved too weak to provide enough speed to get solution.

The solution for this issue emerged by creating two extra matrices, Clist and Rlist which both have the same size as Q, to provide indirect constraints. Each element in Q matched with the variables in both Clist and Rlist at the same matrix position. These two variables represented the row and the column of the index in Q. This constraint was implemented as followed:

$(multifor([I,J],1,N), param(Q,N,Clist,Rlist)$

do

$Q[I,J]\# = Clist[I,J]*N+Rlist[I,J]-N)$

Hence each $Q[I,J]$ with same I (Number) should be in different rows and columns, this constraint is achieved by applying $all\_different$ on each I for $Clist[I,*]$ and $Rlist[I,*]$.

1.2.2 The changes between two viewpoints V1 vs. V2

First, the impact of input order versus first fail using ic on puzzle *extra2*. From Table 2, the backtrack comparison in input order is 235907 v.s. 6438 in first fail . The number of backtracks was almost 40 times more than the first fail variable heuristic, which is opposite to V1, first fail has more backtracks.

Moreover, the domain is much larger than the classic viewpoint. In contrast to V1, now the domain is 1 to 81, after searching the first variable, the domain of the rest variables will not reduce a lot. There will be more backtracking and the computation time will grow exponentially.

More interestingly, for the puzzle *extra2*, both using first fail, compared to V1, even V2 had 1000 less backtrackings. But the runtime in V2 was still more than two times longer than V1. There is one reason to explain this. In V1, the domain is only [1..9], then the domain is easier to converge to one potential candidate for the real solution. In V2, because of the larger domain, it will search deeper before fail. Therefore, the V1 had better performance.

sudoku	**inputorder or first fail**	**IC_global**		**IC**	
		Runtime(s)	**#backtrack**	**Runtime(s)**	**#backtrack**
lambda	input_order	0.0	30	6.0	20869
	first_fail	0.1	63	0.1	466
extra2	input_order	0.2	172	84.9	235907
	first_fail	0.0	43	1.0	6438
sudowiki_nb49	input_order	7.4	11872	3.4	13498
	first_fail	13.7	18434	5.9	22830
extra4	input_order	0.0	42	16.7	62474
	first_fail	0.1	97	0.3	1932
hard17	input_order	4.2	5163	8.8	37086
	first_fail	1.0	1728	0.8	3454
eastermonster	input_order	2.6	2782	1.4	2982
	first_fail	1.5	1647	0.7	2298

Table 2: The comparison between IC and IC_global searching in two different heuristic by V2.

1.2.3 What is the effect of adding the channeling constraints?

For the channeling constraints, since the first viewpoint has an array P as its variables, the second viewpoint has an array Q as its variables, these two variables have to match with each other to reduce the range of variables. First the P was flattened to one dimension array, $Pflat$, and converted to a list, $Plist$. Then, for each element in Q, it has:

$Q[I,J]\# = Pos, gfd : element(Pos, Plist, I)$

Which means when the $Q[I,J]$ has a value of Pos, the Pos^{th} variable in $Plist$ should be equal to I. The reason to use $element/3$ from library(gfd) is that it is the only one that can take variables in both first and third arguments, and change it to a constraint.

Theoretically, due to reasoning in two different viewpoints can provide stronger constraints, the channeling could improve the runtime, or at least as efficient as before. See Table 3, the green color means after channeling is better and vice versa. In most cases, results after channeling are better. However, there were some cases which did not turned out as expected. In those cases, the number of backtracks after channeling kept the same or only became a bit less. Therefore, the runtime could be slightly worse after channeling.

Generally, the channeling indeed eliminated the backtracks. After one of the viewpoints grounded a variable, the variables in the other viewpoint can reduce domain as well.

Sudoku	Channeling			
	IC_global		IC	
	Runtime(s)	#backtrack	Runtime(s)	#backtrack
lambda	0.0	0	0.2	538
	0.0	0	0.0	52
extra2	0.0	0	1.6	4652
	0.0	0	0.4	468
sudowiki_nb49	0.2	41	0.6	1078
	0.1	23	0.3	547
extra4	0.0	0	0.5	1403
	0.0	0	0.2	206
hard17	0.0	1	0.5	873
	0.0	1	0.0	22
eastermonster	0.2	34	0.1	119
	0.0	0	0.1	97

Table 3: The runtime and backtracks after channeling. The green and red color means the performance better and worse after channeling respectively.

1.3 CHR

1.3.1 Describe the variable and value heuristics implemented in the program

For classic viewpoint, the matrix was distributed into 9 rows, 9 columns, 9 blocks respectively, in the format of CHR goals. After collecting all these goals, a simplification rule will be fired to apply $all\_different/1$ to these variables.

For the second viewpoint, the same constraints as in ECLiPSe were implemented. For the constraint that same Number should be in different rows and in different columns, extra constraints were added:

$q\_empty(Num, \_, Index1), q\_empty(Num, \_, Index2) ==>$
$ground(Index1), var(Index2)|$
$RNum1\ is\ (Index1+8)//9, CNum1\ is\ 1+((Index1-1)\ mod 9),$
$diffrow(RNum1, Index2, 1), diffcolumn(CNum1, Index2, 1).$

When any of the variables were grounded, these constraints will be fired. e.g. For known index=1, could get (row,col)=(1,1), the unknown index could not have same row from index 1 to 9, and same column from index 9*(N-1), N is 1 to 9. Moreover, the rule of all different in a block was guaranteed while generating Q[Number,Block,Index].

1.3.2 Compare the two viewpoints for the given Sudoku puzzles and discuss the impact of channeling

From table 4, the red color represents the cases where the performance was worse. The observation showed that in most cases the runtime of V1 is faster than V2. This could be explained by V1 having stronger constraints than V2.

CHR Runtime(s)			
	Perspective		
Puzzles	1st	2nd	Channeling
lambda	11.4	8.7	0.2
extra2	10.3	68.2	0.2
sudowiki_nb49	2.9	15.8	11.4
extra4	31.8	15.8	0.2
hard17	2.2	23.8	0.1
eastermonster	0.4	10.3	1.7

Table 4: The CHR runtime table for 1st, 2nd viewpoint and channeling. The green and red color show a better and worse performance respectively. The yellow color means the performance decreased after channeling.

The channeling in CHR was implemented by active constraints. When one of the variables in a viewpoint was grounded, the corresponding variable in the other viewpoint could also be influenced:

$p\_empty(Number,B,Index), q\_empty(Number,B,Index2) ==>$
$ground(Number), var(Index2)|Index2\ is\ Index.$
$p\_empty(Number,B,Index), q\_empty(N,B,Index) ==>$
$ground(Index), var(Number)|Number\ is\ N.$

By these channeling constraints, the variables in two viewpoints can be highly interactive. As a result, the runtime after channeling became much shorter than any single viewpoint.

2 Hitori

In this section, the solution for Hitori problem is discussed. The solutions implement in two ways: ECLiPSe and CHR in SWI-Prolog.

2.1 ECLiPSe

In ECLiPSe implementation, a basic solver which can find a solution for Hitori problem is first presented. Then, some additional constraints were added to the program in order to get a better performance.

2.1.1 Basic solver

First of all, in Hitori, the rules of the game defined 3 basic constraints:

1.No repetitive number in each row or column.

2.No black cells adjacent.

3.All white cells connected.

For this basic solver, only these three basic constraints were used. Importantly, since there can be several answers to one question, therefore only the optimal one is considered. This optimal means there are no cell be blacked out unnecessarily, that all the rules still satisfied without blacking it out.

The Hitori puzzles from Toledo are used as tasks. Before implementing the constraints, the expression of puzzle was first discussed. The puzzles from puzzles.pl are some fully filled square matrix with different size S (width and height), each point in the matrix is a single number. The range of each number is from 1 to S, S is the size of puzzle.

Because the only change allowed to do is to black out some numbers, so the number had been blacked out need to be distinguish from others. This distinguishing should be easily identify in the program level.

Firstly, the Boolean was considered, which means that Black cells marked as false and white cells marked as true. But to store this Boolean, an extra matrix will be needed. This created unnecessary complexity. In order to keep the data clear and simple, and the original number in the cell became meaningless after blacked out, replacing the cell by specific number was considered. Because the original number's range was from 1 to size S, a black cell need to have a number larger than S.

In Hitori, the reason to black out a cell is that it occurred more than once in its row or column. So after blacked out, the specific number should guarantee no repetitive. This could be achieved by giving different numbers to different blacked cells. So the number can be related to its position index, and start with S+1 to avoid repetitive. The number represents blacked out for each cell is calculated by this equation:

$Number = Index + Size,\ Index = Row * Size + Column - Size$

The next step was about how to represent the variables. Because the feature of Prolog, the puzzle was already a fact filled with ground values. These facts can not be changed. So another new matrix called Pnew was created. Pnew had same size as puzzle matrix but all elements were variables. This Pnew was also going to express the result in the end.

Because only repetitive numbers need to be blacked out, the numbers in Pnew depends on the original puzzle matrix. For each cell, check its repetition:

$Ele\ is\ P[I,J],$

$AC\ is\ P[I,*], array\_list(AC, LColum),\ AR\ is\ P[*,J], array\_list(AR, LRow),$

$((delete(Ele, LColum, Temp), member(Ele, Temp);$

$delete(Ele, LRow, Temp), member(Ele, Temp)) - >$

The using of $- >$ ensure if the number was unique in its row and column, it was copied from puzzle to Pnew, $New[I,J]\# = P[I,J]$. Also if the number was repetitive, the Pnew will have a constraint here and that made Pnew a variable version of the puzzle matrix:

$\# = (New[I,J], P[I,J], B1),\ \# = (New[I,J], I * S + J, B2),\ B1 + B2\# = 1$

After all the representation of the data were decided, the representation of the constraints is discussed in next part.

First of all, the basic constraint: all numbers are different in each row and column. It provided a basic constraint to Pnew's variables. Beside that, more constraints were added to satisfy all the game rules. To speed up the program, two active constraints were implemented.

One constraint was realized by the rule 2, no black cells adjacent. This constraint was added to each cell:

$New[I,J]\# > S,$

$(I = 1; I\backslash = 1, New[I-1,J]\# =< S),\ (I = S; I\backslash = S, New[I+1,J]\# =< S),$

$(J = 1; J\backslash = 1, New[I,J-1]\# =< S),\ (J = S; J\backslash = S, New[I,J+1]\# =< S)$

It is an active constraint because once the New[I,J]'s number is grounded to be black, it can immediately reduce the value range of the cells around it.

Then, the constraint about rule 3 was implemented. A predicate was named instantiate/2, to memorize those instantiation error ever appeared. The idea was to create an extra matrix,*TempArray* and mapping it to Pnew. It had the same size but filled with the index numbers. And for all the black cells, 0 was filled instead of the index:

$\# =< (New[I,J], S, B),\ TempArray[I,J]\# = B * (I * S + J - S)$

The *TempArray* was flatted to one dimension array, *CheckArray*, annd changed to list, *CL*. All the 0 inside *CL* was deleted to get *CLnew*. Check all connected by *CLnew*: Started with an empty list [], named *Done* and put the head of *CLnew* into it. Afterwards, only connected elements can be added to *Done*. This was done by using *delete*/3 to choose one element from *CLnew*, and checked all the cells around it, which means +1, -1, +S, -S on index. Use *memberchk*/2, which can take variables, to check at least one of these 4 was inside the *Done*. If yes, added this element to *Done* otherwise backtracking to get another element.

The speed of different search strategies were tested. The result shows in table 5. The input_order sometimes faster than the first_fail. This is because when using input order, the correct answer was coincidentally chose very early.

Table 5: Solving speed with input_order or first_fail, base on the questions in puzzles.pl with basic solver.

Strategy	7	8	9	10	11	12	14	15
input_order	0.328	437.8	0.109	1.435	0.718	2.293	0.156	155.1
first_fail	0.343	457.3	0.109	1.435	0.764	2.278	0.172	149.5

2.1.2 Solver with 4 additional improvements

The most important extra constraint was defined by the author. Actually it is very simple but easy to be ignored. It is a part of rule 3, all white cells connected. This rule 3 includes a weaker constraint: all white cells should connect to at least one white cell. This weaker constraint was very easy to implement and had a huge effect:

$New[I,J]\# =< S,$
$(I = 1, B6 = 0; I\backslash = 1, \# =< (New[I-1,J], S, B6)),$
$(I = S, B7 = 0; I\backslash = S, \# =< (New[I+1,J], S, B7)),$
$(J = 1, B8 = 0; J\backslash = 1, \# =< (New[I,J-1], S, B8)),$
$(J = S, B9 = 0; J\backslash = S, \# =< (New[I,J+1], S, B9)),$
$B6 + B7 + B8 + B9\# >= 1$

And because the condition to trigger this constraint was opposite to the constraint from rule 2, they were putted together, with ; (or) in between.

After implemented this constraint, the speed of different search strategies is shown in table 6, test on the questions in puzzles.pl. All 15 puzzles were tested and the puzzles cost 0.00s do not show here. Because the time was too short to compare, the only conclusion is both strategies can sometimes be better than another. This depends on the correct answer is in the beginning of the domain of not. Also after adding additional constraints, the time reduced significantly.

Table 6: Solving speed with input_order or first_fail, base on the questions in puzzles.pl with additional solver.

Strategy	7	8	9	10	14	15
input_order	0.062	0.172	0.0	0.047	0.016	0.281
first_fail	0.078	0.187	0.015	0.047	0.031	0.265

Besides, 3 more constraints were added: sandwich triple, quad corner, triple corner. They all come from the inspiration website: http://www.menneske.no/hitori/methods/eng/. Their help are very little and only works to specific puzzle. For example, sandwich triple works for puzzle 1 and triple corner works for puzzle 4. However, it is not very obvious, since the time cost was already 0.0s.

2.2 CHR

For the CHR implementation, the same idea as the ECLiPSe implementation was applied. The differences are the way to present the variables and the way to give constraints.

2.2.1 Basic solver

To represent the data, the same way as previous was applied. Copy the original matrix and change all the competitive number to variables. This was done by converting the prolog facts to CHR goals. In this project, a predicate $genblk/3$ was used. $genblk(P,1,1)$ was the starting point to convert this P. Finally this P can convert all cells of it to CHR goals, $cell/3$, which has row index, column index and value.

Because in CHR there is no matrix that can directly get value from the row and column, so another predicate $generateR\_C/1$ was used. After run $generateR\_C(P)$, it can create CHR goals, $column\_org/2$ and $row\_org/2$, which present each row of original puzzle with column index, and column of original puzzle with row index.

With $cell/3$, $column\_org/2$ and $row\_org/2$, these goals fired under the same logic as previous implementation, and generated $newcell/3$, equal to the previous Pnew. When a new_cell can be blacked out, its value was set by in/2 to has a range of [Original number, Black number].

All the constraints were written as CHR rules.

First all $newcell/3$ fired to get $checkc/4$ and $checkr/4$ and, used simplification to apply $all\_different/1$ in the end.

For no black cells adjacent, each $newcell/3$ can fire when it is black, and used propagation to generate a new goal $blackcell/2$, and each $blackcell/2$ can reduce the range of all the $newcell/3$ around it.

For all white cells connected, after one $newcell/3$ was grounded, it used propagation to generate $whitecell/2$, and the first $whitecell/2$ can be fired together with a goal $checkwhite(1)$, use simplification, generated a $concell/2$. Then each white $whitecell/3$ can do simpagation with any $concell/2$ and switched to a new $concell/2$. Finally after doing search, another goal $checkwhite(4)$ was generated, and it checked if all $whitecell/2$ had been fired and deleted.

The constraints for all black are active constraints, because after generate a $blackcell/2$, it can immediately set the number in the nearby cells.

The $all\_different/1$ is also an active constraint, the $diff/2$ inside it can be fired when a variable become grounded, and reduce another variable's range.

The basic solver used input_order searching method only. The time performance will be discussed together after adding additional improvements.

2.2.2 Solver with 2 additional improvements

In this part, two additional constraints were added: sandwich triple and triple corner.

Both of these two constraints were triggered when the *cell*/3 satisfy the condition. After the constraints fired, a *sandwich*/3 or a *triple*/3 can be generated. These goals immediately set the value in the same position in *newcell*/3.

So the speed was compared and the result shows in table 7. The CHR option of optimize can increase the speed a lot. And in most cases the additional solver was slower than the basic solver. It is because these two additional constraints did not work for all puzzles, so when there are no these specific condition in the puzzle, it will slow down the program. Also, the author tried to implement the constraint about all white cells should connected to at least one white cells, as in the ECLiPSe implementation. But according to the special feature of CHR firing, the implementation became very complicated so the speed slowed down a lot. Finally in CHR implementation that constraint was abandoned.

Table 7: The speed comparasion between basic solver and additional solver, also with or without CHR optimize option.

Strategy	1	4	7	8	9	10	14	15
basic	0.008	0.009	0.055	1.202	0.139	0.610	1.281	6.077
additional	0.007	0.010	0.050	1.156	0.134	0.651	1.403	5.707
basic+opt	0.006	0.005	0.024	0.319	0.080	0.422	1.074	4.381
addition+opt	0.006	0.006	0.032	0.344	0.076	0.486	1.104	4.367

3 conclusion

During this project, we have met a lot of problems. Some in logical level, like how to present a good viewpoint, how to give constraints, some in programming level, for example the one we mentioned in the previous section, the instantiate error. We even found a bug in ECLiPSe and reported it to the developer (the *element*/3 in ic library).

Fortunately, we have finished all the requests and the performance are acceptable, the slowest question can be solved in 5 seconds. Our strong points are some reasonable structures to store the variables. Moreover, a real alternative viewpoint (total different angle of view on variables) not only just play some tricks (like change all the decimal numbers to binary), but also add some strong constraints.

But we understand the program is not optimal, especially in CHR part. Because all the basic functions, *in*/2, $all\_different/1$, $diff/2$ and almost every predicates were written by ourselves, the solving speed can not be as fast as ECLiPSe. Also, we were going to write a heuristic search that can achieve first_fail, but we gave up. The idea was too complicated and will definitely slow down the program (about sorting the searching variables based on the size of its range and the number of $diff/2$ it is involved with).

From this course, we have learned a lot on the programming level. For the programming in CHR part, indeed gave us the idea about how the constraints was set, how the constraints were reduced, and how the searching handle the variables. But more important thing is the skill to think in logical way, and present it. We understand much better about what the logic programming language and constraint programming is.

Appendices

A Workload and tasks division

There were two parts in this project, Sudoku and Hitori. We discussed the logical thinking and possible constraints together with brainstorming. After we had a brief idea about the question, we started by Chinjung Kuo writing Sudoku and Kaiyuan Wei writing Hitori. It is more efficient if one person start with one question, and also ensures everyone knows how to write both ECLiPSe and CHR. Moreover, the discussion between us was continuously, since we worked together and discussed every difficult questions. Then we both learn the constraints programming properly. Also from sharing our opinions, we learned a lot from each other and we earned some precious experience to work together as a team. The table A shows the approximate schedule of us. In summation, we spent 150 hours per person on these project and report.

	Chinjung Kuo	Kaiyuan Wei
1 Apr	✎ Go through slides from lecture ✎ Start to understand the question and study the ECLiPSe ✎ Make a table for possible command from ECLiPSe	✎ Install ECLiPSe on mac and on virtual meachine.
3 Apr	✎ Study all the examples on the lecture and textbook, get familiar with ECLiPSe	✎ Study all the examples in teacher's link.
6 Apr	✎ Research on how to give index in an array ✎ Transfer list into array ✎ Finish the viewpoint 1 for Sudoku	✎ Implement the representation on Hitori. ✎ Implement the basic constraints. ✎ Search first and check all white connected as backtracking. Too slow to solve puzzle 8.
9 Apr	✎ Come up with the idea of viewpoint 2 for Sudoku ✎ Implement 4 constraints on V2, but not efficient, could not converge in proper time. ✎ Discuss the idea of Hitori and how to implement it	✎ Improve the efficiency of constraints on V2 in Sudoku.
22 Apr	✎ Implement little function *checking if white all connect together* for Hitori	✎ Finish improving Sudoku V2. The most difficult puzzle can be solved in 1 second.
3 May	✎ Go through the channeling material and explain the concept to teammate ✎ Implement channeling, but *instantiation error* could not be solved	✎ Change check all white connected before search. The 8th puzzle solved in 470s. ✎ Add additional constraints to Hitori. The 8th puzzle can be solved within 0.2 seconds.
8 May	✎ Discuss the possibility with teammate about channeling ✎ Go through the slides of CHR	✎ Solve the channeling by using gfd library. ✎ Report the bug in the ECLiPSe library.
10 May	✎ Discuss and share CHR concept with teammate	✎ Write and improve the CHR basic functions.
15 May	✎ Finish the Sudoku with V1 in CHR ✎ Write the Sudoku with V2 in CHR	✎ Write Hitori in CHR.
12 May	✎ Continue V2, but instead of following V2 in Eclipse, I try different way, however, it's easier with row and column are all different but it's difficult to implement all 81 position are all different	✎ Finish the Hitori in CHR. 8th puzzle can be solved in 3 seconds
20 May	✎ It works, but could not get solution if the puzzle is too complicated	✎ Rewrite V2 with CHR and finish it. The first puzzle can be solved in 7 seconds.
26 May	✎ Finish the Channeling in CHR	✎ Improve inefficient codes. The Hitori in CHR became 3 times faster.
30 May	✎ Debug and maintain the format of code ✎ Run the Experiment ✎ Start to write the report	✎ Start to write report. ✎ Do small improvement on codes.
31 May	✎ Write the report	✎ Write the report.
1 June	✎ Revise the paper with teammate.	✎ Revise the paper with teammate.