Constraint Processing Assignment

Hendrik Van Hove (r0341925)

December 5, 2016

1 Magic Square: 2x2

Given the magic square of order 2, we know the following:

1. Numbers in the squares must be from 1 to 4.
2. All rows, columns and diagonals must have equal sums.

By some simple mental gymnastics, we can already form a hypothesis that there will be no solution. Let's check if this is correct. We first show how we put the variables on the grid.

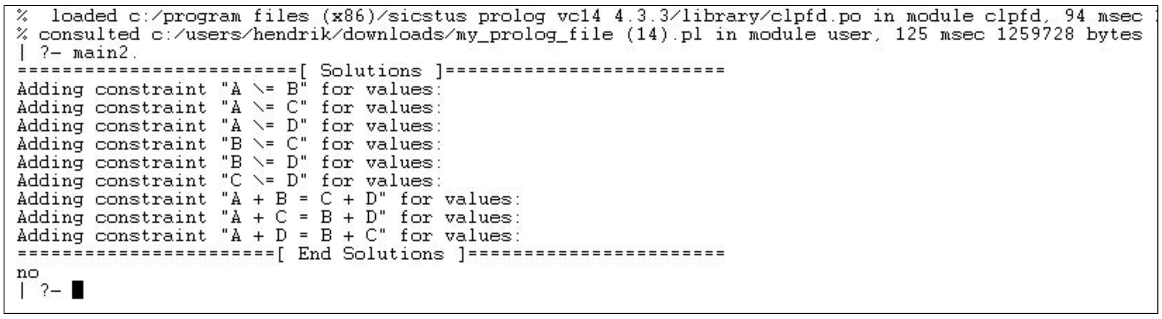
A	B
C	D

Now we show our variables and constraints in the code below. Remember the 2 properties of magic squares we outlined earlier, they are used to make the constraints. The variable ranges are not listed because they are not indexed here.

```
% Variables
Name: A, domain: 1..4
Name: B, domain: 1..4
Name: C, domain: 1..4
Name: D, domain: 1..4

% Variables cannot take the same value
Constraint: A \= B
Constraint: A \= C
Constraint: A \= D
Constraint: B \= C
Constraint: B \= D
Constraint: C \= D

% Rows, columns and diagonals need to have equal sums
Constraint: A + B = C + D
Constraint: A + C = B + D
Constraint: A + D = B + C
```



```
%  loaded c:/program files (x86)/sicstus prolog vc14 4.3.3/library/clpfd.po in module clpfd, 94 msec
% consulted c:/users/hendrik/downloads/my_prolog_file (14).pl in module user, 125 msec 1259728 bytes
| ?- main2.
========================[ Solutions ]========================
Adding constraint "A \= B" for values:
Adding constraint "A \= C" for values:
Adding constraint "A \= D" for values:
Adding constraint "B \= C" for values:
Adding constraint "B \= D" for values:
Adding constraint "C \= D" for values:
Adding constraint "A + B = C + D" for values:
Adding constraint "A + C = B + D" for values:
Adding constraint "A + D = B + C" for values:
======================[ End Solutions ]======================
no
| ?- 
```

Figure 1: Solutions for the 2x2 Magic Square

Subsequently, we feed these constraints into the constraint processing tool. We show our result in Figure 1.

The system did not manage to find any solutions and so our hypothesis is confirmed. There is no such thing as a magic square of order 2.

2 Magic Square: 3x3

In this question we are looking to do the same as in question 1, but for a magic square of order 3. We are facing the same constraints here as in the first question.

1. Numbers in the squares must be from 1 to 9.
2. All rows, columns and diagonals must have equal sums.

However, we ought to use an indexed variable V this time, with index i ranging from 1 to 8 in an order such as in the table below.

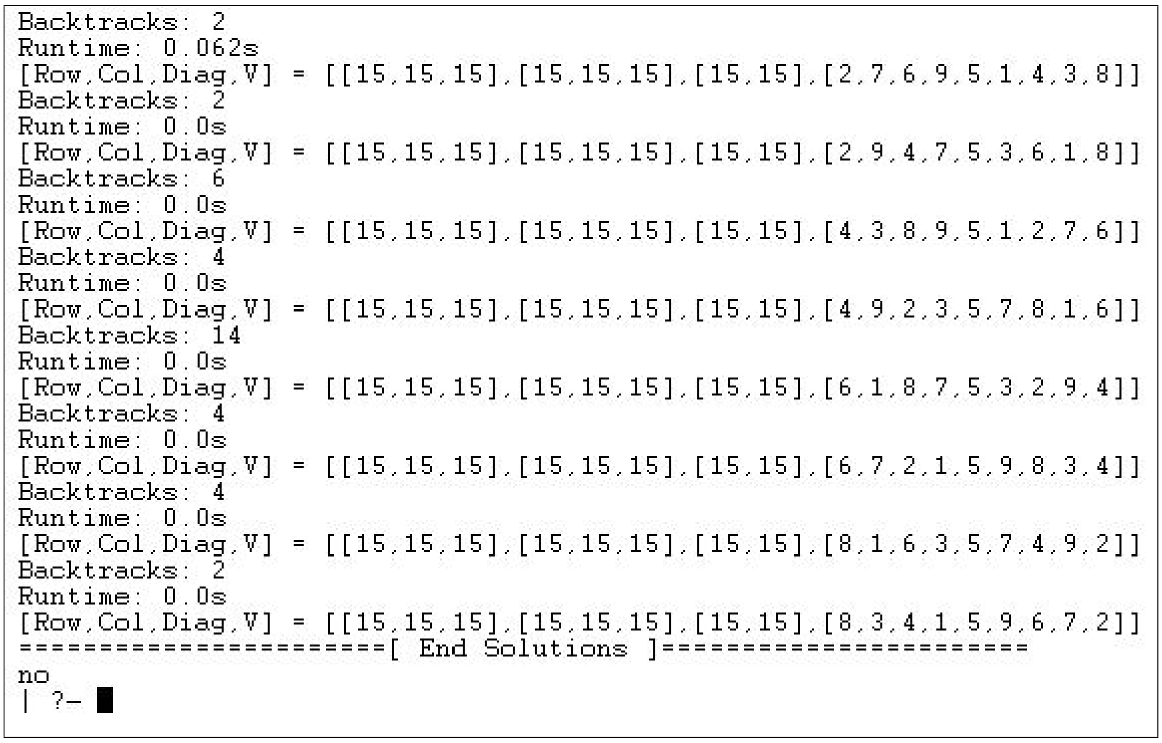
V(0)	V(1)	V(2)
V(3)	V(4)	V(5)
V(6)	V(7)	V(8)

We show our variables and constraints in the code outlined below.

```
% Variables
Name: Row(i), index: 1..3, domain: 15..15
Name: Col(i), index: 1..3, domain: 15..15
Name: Diag(i), index: 1..3, domain: 15..15
Name: V(i), index: 0..8, domain: 1..9

% Variables cannot take the same value
Constraint: V(i) \= V(j), range: i\=j

% Rows, columns and diagonals need to have different sums
```



```
Backtracks: 2
Runtime: 0.062s
[Row,Col,Diag,V] = [[15,15,15],[15,15,15],[15,15],[2,7,6,9,5,1,4,3,8]]
Backtracks: 2
Runtime: 0.0s
[Row,Col,Diag,V] = [[15,15,15],[15,15,15],[15,15],[2,9,4,7,5,3,6,1,8]]
Backtracks: 6
Runtime: 0.0s
[Row,Col,Diag,V] = [[15,15,15],[15,15,15],[15,15],[4,3,8,9,5,1,2,7,6]]
Backtracks: 4
Runtime: 0.0s
[Row,Col,Diag,V] = [[15,15,15],[15,15,15],[15,15],[4,9,2,3,5,7,8,1,6]]
Backtracks: 14
Runtime: 0.0s
[Row,Col,Diag,V] = [[15,15,15],[15,15,15],[15,15],[6,1,8,7,5,3,2,9,4]]
Backtracks: 4
Runtime: 0.0s
[Row,Col,Diag,V] = [[15,15,15],[15,15,15],[15,15],[6,7,2,1,5,9,8,3,4]]
Backtracks: 4
Runtime: 0.0s
[Row,Col,Diag,V] = [[15,15,15],[15,15,15],[15,15],[8,1,6,3,5,7,4,9,2]]
Backtracks: 2
Runtime: 0.0s
[Row,Col,Diag,V] = [[15,15,15],[15,15,15],[15,15],[8,3,4,1,5,9,6,7,2]]
======================[ End Solutions ]======================
no
| ?-
```

Figure 2: Solutions for the 3x3 Magic Square

```
% Rows
Constraint: V(i) + V(j) + V(k) = Row(l), range: i=0, j=1, k=2, l=1
Constraint: V(i) + V(j) + V(k) = Row(l), range: i=3, j=4, k=5, l=2
Constraint: V(i) + V(j) + V(k) = Row(l), range: i=6, j=7, k=8, l=3

% Columns
Constraint: V(i) + V(j) + V(k) = Col(l), range: i=0, j=3, k=6, l=1
Constraint: V(i) + V(j) + V(k) = Col(l), range: i=1, j=4, k=7, l=2
Constraint: V(i) + V(j) + V(k) = Col(l), range: i=2, j=5, k=8, l=3

% Diagonals
Constraint: V(i) + V(j) + V(k) = Diag(l), range: i=0, j=4, k=8, l=1
Constraint: V(i) + V(j) + V(k) = Diag(l), range: i=2, j=4, k=6, l=2
```

We run this in the constraint programming tool and the result in Figure 2 is returned. Interestingly enough, we find eight different solutions for a magic square of order 3. All of these solutions seem to have a 5 in the middle position and even numbers on the edges of the matrix.

Table 1: Sudoku 1

4	6	1	5	2	9	8	3	7
7	2	5	8	3	1	6	9	4
3	9	8	6	4	7	1	5	2
8	3	6	2	1	5	4	7	9
9	5	2	7	8	4	3	6	1
1	7	4	3	9	6	5	2	8
5	4	3	8	7	8	2	1	6
6	1	9	4	5	2	7	8	3
2	8	7	1	6	3	9	4	5

3 Sudoku

3.1 Represent the Sudoku as a constraint problem

The problem faces the following constraints:

1. Every position must be filled with a number from 1..9
2. Every number can only be used once each row, column, or 3x3 subsquares as seen on the larger grid

For variables, we will just use a single variable X_i with an index $i = 1..80$ and a domain of 1..9. In the code below we formulated our variables and constraints.

```
% Variables
Name: X(i), index: 1..80, domain: 1..9

% A number can only occur once every row
Constraint: X(i) \= X(j), range: i / 9 = j / 9 /\ i \= j


% A number can only occur once every column
Constraint: X(i) \= X(j), range: i mod 9 = j mod 9 /\ i \= j

% A number can only occur once every adjacent 3x3 subsquare
Constraint: X(i) \= X(j), range: (i mod 9)/3 = (j mod 9)/3 /\ i / 27 = j / 27 /\ i \= j
```

3.2 Solve the provided puzzles

We do this by adding the code from the webtool to our Prolog code. In Table 1 to 6, I have entered the solutions that were yielded by the SICStus Prolog system.

In addition, we show an overview of results in table 7.

We obtain only one solution that is found for every sudoku in the assignment. Some solutions took marginally longer, but the differences in time are marginal. In general, it seems like the solver has more trouble with the harder sudokus because it needs to do more backtracking in those cases.

Table 2: Sudoku 2

1	3	9	5	8	2	6	7	4
4	8	6	9	7	1	3	5	2
7	2	5	4	6	3	9	1	8
9	4	1	6	5	8	7	2	3
2	5	7	3	4	9	1	8	6
8	6	3	2	1	7	5	4	9
5	9	4	7	2	6	8	3	1
6	7	8	1	3	4	2	9	5
3	1	2	8	9	5	4	6	7

Table 3: Sudoku 3

7	4	2	1	8	3	6	9	5
6	1	3	5	4	9	8	7	2
8	9	5	7	6	2	1	4	3
2	5	7	6	1	4	9	3	8
1	8	9	3	5	7	2	6	4
3	6	4	2	9	8	5	1	7
4	2	6	9	7	5	3	8	1
9	3	8	4	2	1	7	5	6
5	7	1	8	3	6	4	2	9

Table 4: Sudoku 4

1	6	5	7	8	3	2	9	4
9	3	2	6	5	4	8	7	1
4	7	8	2	9	1	3	5	6
8	9	7	3	4	6	5	1	2
3	5	1	9	7	2	4	6	8
6	2	4	8	1	5	9	3	7
7	1	9	4	3	8	6	2	5
5	8	6	1	2	9	7	4	3
2	4	3	5	6	7	1	8	9

Table 5: Sudoku 5

4	2	6	5	1	7	8	3	9
9	7	1	4	8	3	2	5	6
5	8	3	9	6	2	4	7	1
7	1	9	2	5	4	3	6	8
6	5	8	3	9	1	7	4	2
3	4	2	6	7	8	9	1	5
8	9	5	7	4	6	1	2	3
1	3	4	8	2	5	6	9	7
2	6	7	1	3	9	5	8	4

Table 6: Sudoku 6

7	2	3	9	5	6	4	1	8
5	8	4	3	1	7	2	6	9
9	1	6	2	4	8	3	7	5
3	5	8	4	6	1	7	9	2
6	4	9	7	8	2	5	3	1
1	7	2	5	3	9	6	8	4
8	9	7	6	2	4	1	5	3
4	3	1	8	7	5	9	2	6
2	6	5	1	9	3	8	4	7

Table 7: Overview of Solutions - Leftmost

Sudoku #	Solutions Found	Backtracks	Runtime
1	1	0	0.125s
2	1	0	0.141s
3	1	0	0.077s
4	1	20	0.234s
5	1	59	0.125s
6	1	1	0.234s

This is logical, because in the harder sudokus, less numbers have been pre-filled in the grid meaning that the program will have to try more different combinations which in turn increases the chance of backtracking.

3.3 Compare the leftmost to the first-fail + constraint count heuristics, and discuss

Leftmost is a variable-choice heuristic that selects the first unassigned variable from a statically ordered list. First-fail is a dynamic variable choice heuristic that selects the first variable with the current smallest domain. We can extend the first-fail heuristic by counting constraints. This will use the number of constraints on a given variable as a tie-break when two variables have the same domain size. The constraint count here is not necessary though because every variable has the same constraints

In Table 8 we run the code for each sudoku again, but this time with the first-fail + constraint count heuristic. We notice indeed that because of domain reduction, the amount of backtracks has been reduced for the hard sudoku's with 7, 47 and 1 respectively.

3.4 Two out of three rule

The idea of the two out of three rule seems to be very simple for the human mind to understand, but harder to put into constraints. The reasoning for the rule should be the following:

- 2 out of 3 rule in the rows

Table 8: Overview of Solutions - First-Fail + Constraint Counting

Sudoku #	Solutions Found	Backtracks	Runtime
1	1	0	0.116s
2	1	0	0.125s
3	1	0	0.11s
4	1	13	0.11s
5	1	12	0.172s
6	1	0	0.157s

1. Scan 3 adjacent rows for equal values X(i) and X(j)
2. X(i) and X(j) cannot be in the same row
3. X(i) and X(j) cannot be in the same 3x3 subsquare
4. so there must be an X(h) that is
   (a) in the same 3 adjacent rows
   (b) but not in the same row as X(i) and X(j)
   (c) and in a different 3x3 subsquare as X(i) and X(j)
   (d) and taking the leftmost square of 3 remaining possibilities after our other constraints (more info below)

- 2 out of 3 rule in the columns
  1. Scan 3 adjacent columns for equal values X(i) and X(j)
  2. X(i) and X(j) cannot be in the same column
  3. X(i) and X(j) cannot be in the same 3x3 subsquare
  4. so there must be an X(h) that is
     (a) in the same 3 adjacent columns
     (b) but not in the same columns as X(i) and X(j)
     (c) and in a different 3x3 subsquare as X(i) and X(j)
     (d) and taking the uppermost square of 3 remaining possibilities after our other constraints (more info below)

Let's show this process in constraints here below:

```
%
% Row constraints
X(i) = X(j) => X(i)=X(h) \/ X(h+1) = X(i) \/ X(h+2) = X(i),

range: i/9 \= j/9 /\ j/9 \= h/9 /\ i/9 \=h/9 /\
i/27 = j/27 /\ j/27 = h/27 /\

((i mod 9)/3 \= (j mod 9)/3 /\ (j mod 9)/3 \= (h mod 9)/3) /\ (i mod 9)/3 \= (h mod 9)/3 /\

h mod 3 = 0

%
```

```
% Column constraints
X(i) = X(j) => X(i)=X(h) \/ X(h+9) = X(i) \/ X(h+18) = X(i),
range: i mod 9 \= j mod 9 /\ j mod 9 \= h mod 9 /\ i mod 9 \=h mod 9 /\

i/27 \= j/27 /\ j/27 \= h/27 /\ i/27 \= h/27 /\

((i mod 9)/3 = (j mod 9)/3 /\ (j mod 9)/3 = (h mod 9)/3) /\

(h/3) mod 9 = 0
```

Now let's explain the process behind these constraints. For the row constraints, we can again use integer division to make sure variables don't occur in the same rows. We use integer division by 27 to make sure variables occur in the same 3 adjacent rows (keep in mind we need 1 less term here, because these terms are equal instead of unequal). Then, we use the modulo of 9 divided by 3 to check for the 3x3 subsquares. Values cannot take up the same position by making $i < j$ and we hardcode h mod 3 to 0. Why do we do this? Well, because in our constraint, we say X(i) could be equal to either X(h) or X(h+1) or X(h+2). This means we only want to have the leftmost square and then add indexes to it. This is easily solved by taking the mod 3 of h and setting it to 0.

For the column constraints, the approach is exactly the same, only now we have to tackle the process in column approach rather than in row approach. So we must do some things differently. We now say that the modulo 9 of the indexes cannot be equal so they are in different columns Next, we can take the 3-adjacent columns by making the index modulo 9/3 equal to one another and finally we can divide these into different 3x3 subsquares by saying that the integer division by 27 must be different.

3.5 Does this extension affect the solutions you find? Discuss

This extension does not affect the solution we find. We already knew there was only one solution from our earlier computations. Changing the algorithm to find that one solution is not going to change the solution we will find.

3.6 Does this extension affect the time needed to find a solution? Discuss

The time needed to find a solution increases drastically when employing the 2 out of 3 rule. This makes sense when looking at the constraints. Instead of having to iterate thrice over i and j with possibilities ranging from 0..80 without the 2 out of 3 rule, the algorithm now needs to additionally iterate twice over i, j and h. This additional variable is the cause for a lot of extra computations, which increases the compute times significantly.

3.7 Influence of employed heuristic? What about difficulty level? Explain

The amount of backtracks and runtime were written down in table 7, 8 and 9. We can clearly notice that the difference in heuristics in the tests without the 2 out of 3 rule are quite different. Backtracking was reduced significantly in all cases where it occurred.

However, this effect seems to fade away when using the two out of three rule. Because of the added constraints, the algorithm will now be less likely to go into backtracking even when executing the leftmost heuristic. The 2 out of 3 rule does help however when it comes to the more difficult

Table 9: Comparison of heuristics with 2 out of 3 rule

	Leftmost + 2/3 Rule			FFC + 2/3 Rule		
Sudoku #	Solutions Found	Backtracks	Runtime	Solutions	Backtracks	Runtime
1	1	0	29.077s	1	0	29.084s
2	1	0	29.028s	1	0	29.026
3	1	0	28.829s	1	0	29.535s
4	1	0	29.374s	1	2	30.079s
5	1	5	30.187s	1	1	29.778s
6	1	0	29.658s	1	0	30.910s

puzzles. The difference between the harder and easier puzzles has been reduced noticeably.

We can conclude that actually, we are using implied constraints to sudoku with the two out of three rule. Given our sudoku constraint problem, if after the initial constraint building, the problem still takes a lot of time, we can refine our search by adding new constraints. These new constraints cannot change the solutions but they increase the propagation capabilities of the algorithm. These "implied constraints" can drastically reduce solving time by reducing the search space while not compromising the outcome. The two out of three rule is such an implied constraint, and as we have noted before that using this strategy was more successful in the harder sudoku problems to reduce solving time. There is only the issue of increased processing time that arises, as we have noticed as well for this sudoku solver.