Artificial Intelligence 1
Lab 3

Roeland Lindhout (s2954524) & Younes Moustaghfir (s2909758)
AI2/AI3

June 6, 2016

1 Constraint Satisfaction Problems

Solving a small set of equations

The program took quite some time, and states, to calculate the variables. The only solution that the program found, was A = 3, B = 7 and C =2. This was found in 63254 states. The most effective heuristic is the arc heuristic, which solves the problem in 4 states. Besides this, only forward checking showed an improvement, to 254 states. Arc consistency quickly takes away the options that are not possible when a value is chosen, and it does this more quickly than forward checking, which explains why it is the quickest.

Market

We used three variables, each with a domain from 0 to 100. The three variables represented the number of fruits. Added to each other, they had to equal 100. The second constraint considered the prices: $A * 88 + B * 99 + C * 102 = 10000$. Wherein A represented the oranges, B the grapefruit and C the melons. The reason that they have to be equal to 10000, is that the prices are given in cents.

There were 5 solutions, in 10308 states. We checked the solutions, and they were all correct. The solutions were as follows:

```
### Solution 1 ###
A =    1
B =    62
C =    37

### Solution 2 ###
A =    4
B =    48
C =    48

### Solution 3 ###
```

```
A =    7
B =    34
C =    59

### Solution 4 ###
A =    10
B =    20
C =    70

### Solution 5 ###
A =    13
B =    6
C =    81
```

For the heuristics, the same can be said as with the 42 problem.

Chain of trivial equations

Without an extra constraint, many states were visited, and 100 solution were found. The solutions were of course all the integers from 0 up to and including 99. With the constraint of A being 42, 2502 states were visited, and only 1 solution was found, namely all are 42. With the constraint of Z being 42, only 27 states were visited. The reason that this is so much lower, is that if $A = 42$, you have yet to determine all other variables, so you have to test all of them. When $Z = 42$, you can backtrack using the known final value, because each variable gets the value of the next variable.

None of the heuristics have a positive effect, which makes sense because the constraints are simply to simple to be able to be done more quickly

Constraint graph

For this problem, there are 5 variables, namely A to E. Each of these variables has a domain from 1 up to and including 4. We had to introduce an extra variable, to be able to represent the constraint $C \neq D + 1$. This variable ranged from 2 to 4, because it is always bigger than 1, since 0 was not in the original domain. 5 (i.e. 4 + 1) also didn't have to be included, because it is impossible for C to be 5, so when D = 4, $C \neq D + 1$ is always true.

The problem was solved in 29 states, and 2 solutions were found:

```
### Solution 1 ###
A =    3
B =    3
C =    4
D =    2
E =    1
F =    3

### Solution 2 ###
```

```
A =    4
B =    4
C =    2
D =    3
E =    1
F =    4
```

By making the problem arc-consistent, 17 states are visited instead of 29, because the search space is limited, by removing all impossible options at the start. By keeping the problem arc-consistent, even fewer states are visited (13). Forward checking reduces the amount of states to 15. Arc-consistency is more efficient, because it takes a long time to notice where the mistakes are with Forward Checking.

Cryptarithmetic puzzles

For the SEND+MORE=MONEY-problem, we had to define 7 variables with a domain ranging from 0 to 9 and 4 variables ranging from 0 to 1. The first 7 variables were all the letters of those words. In this case that were: S, E, N, D, M, O, R, Y. SEND and MORE are like a sum, so carry-over variables are needed. In this case we named them: X1 to X4, because the total length of the words SEND and MORE is 4, therefore there are 4 places where a carry-over could take place. Most of the constraints are quite logical, but we chose the value 1 for M, because the values of S and M cannot be higher than 20(because they are both maximally 9). Therefore M cannot be higher than 1. M can also not be 0, because numbers do not start with a zero.

For the UN+UN+NEUF = ONZE-problem, we had to make a small adjustment compared to the code for the first problem described in this subsection, because we did not have two, but three words, so it became possible that the carry-over could also be 2. As a heuristic, we figured that X3, which was the last carry-over, could not be bigger than 1, because at that point, we no longer had 3 letters. We did not need a fourth carry-over, because there is no further addition in the final step. There is only one solution. Namely:

```
U =    8
N =    1
E =    9
F =    7
O =    2
Z =    4
X1 =   0
X2 =   2
X3 =   1
```

The following solution was found for the ONE + NINE + TWENTY + FIFTY = EIGHTY problem, again after some adjustments. For this one, we used quite a lot of tricks, to make the constraints more efficient, because it had

quite a large search space. All the first letters had a possible domain from 1 to 9, because the numbers will not start with a 0. Y could only be an even number, because E + E is always even. The first and the second carry-over could not be larger than 2, because of the amount of variables, and they could not be 0 as well, because then there would have to be negative values, after crossing out duplicate variables on both sides of the equation. The other carry-overs were also logically limited. Also, W + F + X4 had to be bigger than 10, because else X5 would be 0, which would imply that T = E. This is the solution:

```
O =    9
N =    8
E =    4
I =    5
T =    3
W =    6
Y =    2
F =    7
G =    0
H =    1
X1 =   1
X2 =   2
X3 =   3
X4 =   2
X5 =   1
```

Lastly, the last puzzle was solved. This was a lot easier than the one before this, so not a lot of tricks had to be used. We only used that the first letters could not be 0. The following solution was found:

```
I =    5
G =    2
U =    6
E =    8
S =    1
T =    4
H =    7
R =    9
X1 =   2
X2 =   1
X3 =   1
X4 =   1
```

The minimal remaining values heuristic helped, because the program will start with the letters that are at the beginning of words, since they have a smaller domain. Arc-consistency is once again the most effective heuristic, but Forward Checking also greatly reduces the amount of states.

Finding the first 21 primes

For prime numbers, two rules can be applied. For the low numbers, we check if they are divisible by the numbers 1 -9. If they are, they are clearly composite numbers and therefore not prime numbers. The high prime numbers can be found by dividing the numbers by one of the low numbers, and seeing what remains. This remainder is a prime number. This is looked at in a forall loop. Because either of these cases can be true, the any-quantifier is used.

The program only uses 22 states to find the 21 primes, so heuristics can't really be applied.

Solving Sudoku's

The sudoku problem is a rather simple one, but it is quite time-consuming. We used a 2D-array to represent the sudoku. In the domain-section, we stated the known places. After this, we just had to make sure that everything was different, according to the rules of sudoku. For the line and row differences, we used a forall-loop, but we could not figure out how to do this for the blocks, so we wrote that out entirely. The left sudoku was solved much more quickly, which is because there are much more known numbers. Below are the solutions of respectively the left and the right sudoku.

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

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

The Forward-Checking heuristic was the most useful, because the errors that can be made are easily and quickly spotted. The minimal remaining value heuristic is also very efficient. This is a lot like how people fill-in sudokus, because when most places are known, it is easy to see what the number should be, because of the low number of options left. Also, arc-consistency at the beginning helps a lot, by removing all the impossible values.

n-queens

For the n-queens problem, we choose to use n-variables, all with a range from 1 to n. The variables represented the columns, and their value the rows (or vice versa). Then, the constraints were that they are all different, and that the row-numbers minus the corresponding column values are not the same, to simulate that the queens cannot be in the same diagonal. We did not manage to this in a good, scalable way, which resulted in a lot of constraints for the higher n's. We found the following amount of solutions, not taking rotation etc. into account.

```
4 queens : 2
5 queens : 10
6 queens : 4
7 queens : 40
8 queens : 92
9 queens : 352
10 queens : 724
```

The only heuristics that had an effect were arc-consistency and forward checking. Once again, arc-consistency is more efficient.

Magic squares

We used a 4*4 array to represent the square, with a domain of 1 to 16. Then we said that all entries had to be different, and defined the values of the rows and the diagonals. We used the trick that the values in a magic square of 4, is always 34. We found 7040 solutions. Because the search-space is so big, we had to use a lot of heuristics. We choose to use the arc-heuristic, because it limits the search space more efficiently and quickly than forward checking. MRV is also very helpful, because when a row is almost filled, it is very easy to fill in the last one for example, because only a limited number of combinations will fit to the sum. The degree-heuristic only slowed down the process, because it only looks at how many neighbors, while it is all very dependent on each other, further than just the amount of neighbors.

Boolean satisfiability

To solve this problem, we used 5 variables, each with a possible value of either 1 or -1. We then used the max operator to simulate the disjunction symbol. One of the variables in each of the conjuncts always had to be true, i.e. 1, to make sure that everything is true. This was done by saying that the max of one of the three variables is 1. For a not-variable, we simply negated that variable. Since the variables are interchangeable, it did not matter in what order they are in the max-operator. We found 9 solutions:

```
### Solution 1 ###
x1 =   -1
x2 =   1
```

```
x3 =   -1
x4 =   -1
x5 =   -1

### Solution 2 ###
x1 =   1
x2 =   1
x3 =   -1
x4 =   -1
x5 =   -1

### Solution 3 ###
x1 =   1
x2 =   1
x3 =   1
x4 =   -1
x5 =   -1

### Solution 4 ###
x1 =   1
x2 =   -1
x3 =   1
x4 =   1
x5 =   -1

### Solution 5 ###
x1 =   -1
x2 =   -1
x3 =   -1
x4 =   -1
x5 =   1

### Solution 6 ###
x1 =   1
x2 =   -1
x3 =   -1
x4 =   -1
x5 =   1

### Solution 7 ###
x1 =   1
x2 =   1
x3 =   -1
x4 =   -1
x5 =   1

### Solution 8 ###
x1 =   -1
x2 =   -1
x3 =   -1
```

```
x4 =   1
x5 =   1

### Solution 9 ###
x1 =   1
x2 =   -1
x3 =   1
x4 =   1
x5 =   1
```

The program was already quite effective, with 35 states visited. Both arc and forward checking reduce this to 29 states. There was not much more room for improvement. All the other heuristics had no effect.

2 Logic

Model checking in propositional logic

```
int checkAllModels(int modelSize) {
       /* return 1 if KB entails INFER, otherwise 0 */
       inferred = 0;
       int i,j;
       int cnt = 0;
       while (cnt < pow(2,modelSize)) {
              i = cnt;
              j = 1;
              while (i != 0) {
                     model[modelSize-j] = i%2;
                     i = i/2;
                     j++;
              }
              if (evaluateExpressionSet(inferSize, infer)) {
                     return inferred = 1;
              }
              cnt++;
       }
       showModel(modelSize);
       return inferred;
}
```

Resolution in propositional logic

- Initialization function

```
void init(clauseSet *s) {
       clause c;
       char i;
       int j = 0;
```

```
        char *arr = malloc(20*sizeof(char));
        makeEmptyClauseSet(s);
        scanf("%c" , &i);
        while (i != ' ') {
                if (i == '[') {
                        while (i != ']') {
                                if ( i != '[') {
                                        arr[j] = i;
                                        j++;
                                }
                                scanf("%c", &i);
                        }
                        arr[j] = '\0';
                        makeClause(&c, arr);
                        insertInClauseSet(c,s);
                        j = 0;
                }
                scanf("%c", &i);
        }
        free(arr);
}
```

- Proof Print
  This piece of code does not work, but we did not manage to fix the problem that caused the segmentation fault.

```
void recursivePrintProof(int idx, clauseSet kb) {
        int i,j;
        for (i=0; i < kb.size; i++) {
                for (j=i+1; j < kb.size; j++) {
                clauseSet resolvents;
                resolveClauses(kb.clauses[i], kb.clauses[j],
                    &resolvents);
                   if(areEqualClauses(resolvents.clauses[0],
                       kb.clauses[idx]) ||
                       areEqualClauses(resolvents.clauses[1],
                       kb.clauses[idx])) {
                          recursivePrintProof(i,kb);
                          recursivePrintProof(j,kb);
                          printClause(kb.clauses[idx]);
                          printf("is inferred from");
                          printClause(kb.clauses[i]);
                          printf(" and ");
                          printClause(kb.clauses[j]);
                          printf(".\n");
                }
                freeClauseSet(resolvents);
                }
        }
```

```
20                  printf("\n");
21          }
```

- Proof
  We could not find a good proof, because we did not manage to get the program to work correctly.

Prolog

Biblical family

1. ?- grandfather(X, lot).

   X = terach .

2. ?- findall(X, (son(X,Y), son(Y, terach)), Z).

   Z = [isaac, lot].

Arithmetic with natural numbers

1. ?- plus(s(s(s(0))), s(s(0)), s(s(s(s(s(0)))))).

   true.

2. ?- plus(s(s(s(0))), s(s(0)), s(s(s(s(s(0)))))).

   false.

3. Even and odd

```
1       % even(X) is true if X is even
2       even(0) :- isnumber(0).
3       even(s(s(X))) :- even(X).
4
5       % odd(X) is true if X is odd
6       odd(s(0)) :- isnumber(s(0)).
7       odd(s(s(X))) :- odd(X).
```

4. Integer Division

```
1       % div2(X,Y) is true if X div 2 is Y
2       div2(0, 0) :- isnumber(0).
3       div2(1, 0) :- isnumber(0).
4       div2(X, Y) :- plus(Y, Y, X), even(X).
5       div2(X, Y) :- plus(s(Y), Y, X), odd(X).
```

5. Integer Division with predicate times

```
divi2(0, 0) :- isnumber(0).
divi2(1, 0) :- isnumber(0).
divi2(X, Y) :- even(X), times(s(s(0)), Y, X).
divi2(X, Y) :- odd(X), times(s(s(0)), Y, Z1), plus(s(0),
    Z1, X).
```

6. Log

```
% log(X, B, N) is true if B ^ N = X
log(X, B, 0) :- isnumber(s(0)).
log(X, B, N) :- pow(B, N, X).
```

7. Fibonacci

```
% fib(X, Y) is true if fib(X) = Y
fib(0, 0).
fib(s(0), s(0)).
fib(X, Y) :- minus(X, s(0), X1), minus(X, s(s(0)), X2),
    plus(Y1,Y2,Y), fib(X1, Y1), fib(X2, Y2).
```

8. Power

```
% power(X,N,Y) is true if X^N = Y
power(X, N, Y) :- even(N), div2(N,C), pow(X, s(s(0)), D),
    pow(D, C, Y).
power(X, N, Y) :- odd(N), minus(N, s(0), Q), pow(X,Q,L),
    times(X, L, Y).
```

Lists

```
% member(X, L) is true if X is a member of list L
member(X, [X|_]).
member(X, [_|T]) :- member(X, T).

% concat(L, X, Y) is true if L is the concatenation of the lists
    X and Y
concat(L, X, Y) :- append(X, Y, L).

% reverse(L, R) is true if R is the reversal of L
reverse([], []).
reverse([H|T], R) :- reverse(T, RT), append(RT, [H], R).

% palindrome(L) returns true if L is a palindrome.
palindrome(L) :- reverse(L, L).
```

Maze

```
edge(a,b).
edge(b,f).
edge(f,e).
edge(f,j).
edge(j,k).
edge(k,g).
edge(g,c).
edge(c,d).
edge(d,h).
edge(h,l).
edge(l,p).
edge(i,m).
edge(m,n).
edge(n,o).

path(X, Y) :- edge(X, Y).
path(X, Y) :- path(Z, Y), edge(X, Z).
```

When we ask the system: path(a,p), the system will return true. When we ask the system path(a,m), the system will run out of memory, which is not a surprising action, because there is no possible path from a to m.

3 Appendix

CSP's

- Market

```
variables:
A,B,C : integer;

domains:
A,B,C <- [0..100];

constraints:
A + B + C = 100;
A*88 + B*99 + C*102 = 10000;

solutions: all
```

- Chain of trivial equations

```
variables:
A, B, C, D, E, F, G, H, I, J, K, L ,M, N, O , P, Q, R, S,
    T, U , V, W, X, Y, Z : integer;

domains:
A, B, C, D, E, F, G, H, I, J, K, L ,M, N, O , P, Q, R, S,
    T, U , V, W, X, Y, Z <- [0..99];

```

```
constraints:
A = B;
B = C;
C = D;
D = E;
E = F;
F = G;
G = H;
H = I;
I = J;
J = K;
K = L;
L = M;
M = N;
N = O;
O = P;
P = Q;
Q = R;
R = S;
S = T;
T = U;
U = V;
V = W;
W = X;
X = Y;
Y = Z;

solutions: all
```

- Constraint Graph

```
variables:
A,B,C,D,E,F : integer;

domains:
A,B,C,D,E <- [1..4];
F <- [2..4];

constraints:
A > D;
B >= A;
D > E;
C > E;
alldiff(C,A);
alldiff(B,C);
alldiff(C,D);
F = D + 1;
alldiff(C,F);

```

```
19          solutions: all
```

- Cryptharithmetic puzzles

```
1           SEND + MORE = MONEY
2           variables:
3           S,E,N,D,M,O,R,Y : integer;
4           X1, X2, X3, X4 : integer;
5
6           domains:
7           S,E,N,D, O,R,Y <- [0..9];
8           M <- [1..9];
9           X1, X2, X3, X4 <- [0,1];
10
11          constraints:
12          alldiff(S,E,N,D,M,O,R,Y);
13          D + E = Y + 10 *X1;
14          X1 + N + R = E + 10 * X2;
15          X2 + E + O = N + 10 * X3;
16          X3 + S + M = O + 10 * X4;
17          X4 = M;
18
19          solutions: all
20
21          UN + UN + NEUF = ONZE
22          variables:
23          U,N,E,F,O,Z : integer;
24          X1, X2, X3 : integer;
25
26          domains:
27          U,N,E,F, Z <- [0..9];
28          O <- [1..9];
29          X1, X2 <- [0..2];
30          X3 <- [0,1];
31
32          constraints:
33          alldiff(U,N,E,F,O,Z);
34          N + N + F = E + 10 *X1;
35          X1 + U + U + U= Z + 10 * X2;
36          X2 + E = N + 10 * X3;
37          X3 + N = O;
38
39          solutions: all
40
41          ONE + ONE + NINE + TWENTY + FIFTY = EIGHTY
42          variables:
43          O, N, E, I, T, W, Y, F, G, H : integer;
44          X1, X2, X3, X4, X5 : integer;
45
```

```
domains:
I, W, G, H <- [0..9];
O, N, T, F, E <- [1..9];
Y <- [0,2,4,6,8];
X1, X2 <- [1,2];
X3 <- [0..3];
X4 <- [0..2];
X5 <- [1];

constraints:
W + F + X4 > 10;
alldiff(O, N, E, I, T, W, Y, F, G, H);
E + E + Y =10 *X1;
X1 + N + N + T= 10 * X2;
X2 + O + I + N + F = H + 10 * X3;
X3 + N + E + I = G + 10 * X4;
X4 + W + F = I + 10 * X5;
X5 + T = E;

solutions: all

I + GUESS + THE + TRUTH = HURTS
variables:
I, G, U, E, S, T, H, R : integer;
X1, X2, X3, X4 : integer;

domains:
U, E, S, R <- [0..9];
I, G, T, H <- [1..9];
X1 <- [0..3];
X2, X3 <- [0..2];
X4 <- [0..1];

constraints:
alldiff(I, G, U, E, S, T, H, R);
I + S + E + H = S + 10 * X1;
X1 + S + H + T = T + 10 * X2;
X2 + E + T + U = R + 10 * X3;
X3 + U + R = U + 10 * X4;
X4 + G + T = H;

solutions: all
```

- Finding the first 21 primes

```
variables:
A : integer;

domains:
```

```
A <- [2..75];

constraints:
forall(i in [2..9])
any((A mod i),(i div A));
end

solutions: all
```

- Solving Sudoku's

```
Sudoku 1
variables:
s[9][9] : integer;

domains:
s <- [1..9];
s[0][2] <- [8]; s[0][3] <- [6]; s[0][4] <- [3]; s[0][5] <-
    [2]; s[0][6] <- [4];
s[1][1] <- [4]; s[1][7] <- [1];
s[2][0] <- [5]; s[2][3] <- [9]; s[2][5] <- [4]; s[2][8] <-
    [6];
s[3][0] <- [8]; s[3][8] <- [5];
s[4][0] <- [6]; s[4][8] <- [4];
s[5][0] <- [1]; s[5][2] <- [7]; s[5][6] <- [9]; s[5][8] <-
    [2];
s[6][0] <- [4]; s[6][3] <- [7]; s[6][4] <- [5]; s[6][5] <-
    [1]; s[6][8] <- [3];
s[7][1] <- [6]; s[7][7] <- [2];
s[8][2] <- [5]; s[8][3] <- [8]; s[8][4] <- [2]; s[8][5] <-
    [6]; s[8][6] <- [7];

constraints:
#line/row configuration
forall(i in [0..8])
alldiff(s[i][0], s[i][1], s[i][2], s[i][3],s[i][4],s[i][5],
    s[i][6], s[i][7],s[i][8]);
alldiff(s[0][i], s[1][i], s[2][i], s[3][i],s[4][i],s[5][i],
    s[6][i], s[7][i],s[8][i]);
end

#first column of blocks
alldiff(s[0][0],s[0][1], s[0][2], s[1][0],s[1][1],
    s[1][2],s[2][0],s[2][1],s[2][2]);
alldiff(s[3][0],s[3][1], s[3][2], s[4][0],s[4][1],
    s[4][2],s[5][0],s[5][1],s[5][2]);
alldiff(s[6][0],s[6][1], s[6][2], s[7][0],s[7][1],
    s[7][2],s[8][0],s[8][1],s[8][2]);

```

```
29          #second column of blocks
30          alldiff(s[0][3],s[0][4], s[0][5], s[1][3],s[1][4],
                s[1][5],s[2][3],s[2][4],s[2][5]);
31          alldiff(s[3][3],s[3][4], s[3][5], s[4][3],s[4][4],
                s[4][5],s[5][3],s[5][4],s[5][5]);
32          alldiff(s[6][3],s[6][4], s[6][5], s[7][3],s[7][4],
                s[7][5],s[8][3],s[8][4],s[8][5]);
33
34          #third column of blocks
35          alldiff(s[0][6],s[0][7], s[0][8], s[1][6],s[1][7],
                s[1][8],s[2][6],s[2][7],s[2][8]);
36          alldiff(s[3][6],s[3][7], s[3][8], s[4][6],s[4][7],
                s[4][8],s[5][6],s[5][7],s[5][8]);
37          alldiff(s[6][6],s[6][7], s[6][8], s[7][6],s[7][7],
                s[7][8],s[8][6],s[8][7],s[8][8]);
38
39          solutions: all
40
41          Sudoku 2
42          variables:
43          s[9][9] : integer;
44
45          domains:
46          s <- [1..9];
47          s[0][7] <- [1]; s[0][8] <- [2];
48          s[1][8] <- [3];
49          s[2][2] <- [2]; s[2][3] <- [3]; s[2][6] <- [4];
50          s[3][2] <- [1]; s[3][3] <- [8]; s[3][8] <- [5];
51          s[4][1] <- [6]; s[4][4] <- [7]; s[4][6] <- [8];
52          s[5][5] <- [9];
53          s[6][2] <- [8]; s[6][3] <- [5];
54          s[7][0] <- [9]; s[7][4] <- [4]; s[7][6] <- [5];
55          s[8][0] <- [4]; s[8][1] <- [7]; s[8][5] <- [6];
56
57          constraints:
58          #line/row configuration
59          forall(i in [0..8])
60          alldiff(s[i][0], s[i][1], s[i][2], s[i][3],s[i][4],s[i][5],
                s[i][6], s[i][7],s[i][8]);
61          alldiff(s[0][i], s[1][i], s[2][i], s[3][i],s[4][i],s[5][i],
                s[6][i], s[7][i],s[8][i]);
62          end
63
64          #first column of blocks
65          alldiff(s[0][0],s[0][1], s[0][2], s[1][0],s[1][1],
                s[1][2],s[2][0],s[2][1],s[2][2]);
66          alldiff(s[3][0],s[3][1], s[3][2], s[4][0],s[4][1],
                s[4][2],s[5][0],s[5][1],s[5][2]);
67          alldiff(s[6][0],s[6][1], s[6][2], s[7][0],s[7][1],
                s[7][2],s[8][0],s[8][1],s[8][2]);
```

```
68
69          #second column of blocks
70          alldiff(s[0][3],s[0][4], s[0][5], s[1][3],s[1][4],
                s[1][5],s[2][3],s[2][4],s[2][5]);
71          alldiff(s[3][3],s[3][4], s[3][5], s[4][3],s[4][4],
                s[4][5],s[5][3],s[5][4],s[5][5]);
72          alldiff(s[6][3],s[6][4], s[6][5], s[7][3],s[7][4],
                s[7][5],s[8][3],s[8][4],s[8][5]);
73
74          #third column of blocks
75          alldiff(s[0][6],s[0][7], s[0][8], s[1][6],s[1][7],
                s[1][8],s[2][6],s[2][7],s[2][8]);
76          alldiff(s[3][6],s[3][7], s[3][8], s[4][6],s[4][7],
                s[4][8],s[5][6],s[5][7],s[5][8]);
77          alldiff(s[6][6],s[6][7], s[6][8], s[7][6],s[7][7],
                s[7][8],s[8][6],s[8][7],s[8][8]);
78
79          solutions: all
```

- n-queens

  This is only one of the n-queens problems, namely n = 7. The others are solved similarly.

```
1           variables:
2           q1, q2, q3, q4, q5, q6, q7 : integer;
3
4           domains:
5           q1, q2, q3, q4, q5, q6, q7 <- [1..7];
6
7           constraints:
8           alldiff(q1,q2,q3,q4,q5, q6, q7);
9           abs(q1-q2) <> abs(1-2);
10          abs(q1-q3) <> abs(1-3);
11          abs(q1-q4) <> abs(1-4);
12          abs(q1-q5) <> abs(1-5);
13          abs(q1-q6) <> abs(1-6);
14          abs(q1-q7) <> abs(1-7);
15          abs(q2-q3) <> abs(2-3);
16          abs(q2-q4) <> abs(2-4);
17          abs(q2-q5) <> abs(2-5);
18          abs(q2-q6) <> abs(2-6);
19          abs(q2-q7) <> abs(2-7);
20          abs(q3-q4) <> abs(3-4);
21          abs(q3-q5) <> abs(3-5);
22          abs(q3-q6) <> abs(3-6);
23          abs(q3-q7) <> abs(3-7);
24          abs(q4-q5) <> abs(4-5);
25          abs(q4-q6) <> abs(4-6);
26          abs(q4-q7) <> abs(4-7);
```

```
        abs(q5-q6) <> abs(5-6);
        abs(q5-q7) <> abs(5-7);
        abs(q6-q7) <> abs(6-7);

        solutions: all
```

- Magic Square

```
        variables:
        A[4][4] : integer;

        domains:
        A <- [1..16];

        constraints:
        alldiff(A);
        forall(i in [0..3])
        A[i][0] + A[i][1] + A[i][2] + A[i][3] = 34;
        A[0][i] + A[1][i] + A[2][i] + A[3][i] = 34;
        end
        A[0][0] + A[1][1] + A[2][2] + A[3][3] = 34;
        A[0][3] + A[1][2] + A[2][1] + A[3][0] = 34;

        solutions: all
```

- Boolean satisfiability

```
        variables:
        x1,x2,x3,x4,x5 : integer;

        domains:
        x1,x2,x3,x4,x5 <- [-1,1];

        constraints:
        max(max(x1,x2),-x3) = 1;
        max(max(-x1,-x2),-x4) = 1;
        max(max(x1,-x2),-x5) = 1;
        max(max(-x1,x3),-x4) = 1;
        max(max(x1,-x3), x5) = 1;
        max(max(x1,-x4),x5) = 1;
        max(max(x2,x4),x5) = 1;
        max(max(-x3,x4),-x5) = 1;

        solutions: all
```