# Artificial Intelligence 1
# Lab 3a:

Albert Dijkstra (s3390071)
Niels Bugel (s3405583)

LC[7] Team Supreme

May 27, 2018

## Programming assignments

For each of these problem, we check if the amount of visited states is close to the (number of variable assignments)*(number of solutions). It can be less if the solutions are very much alike or it can be a little bit more (around the amount of solution states) if the solution is completely different.

In the following problems arc is usually faster than fc due to the nature of the problems, because with all of the problems there are a lot of ”hard” connections(connections that rule out a lot of possibilities) between the variables, which makes it easier for arc to rule out a lot of values.

## Exercise 1: Heuristic and propagation technique analysis

### Solving a small set of equations

The solver finds only one solution for this problem.
We use the -arc flag to solve this problem.
The reason that keeping the problem arc consistent works so well is because with the last constraint of A*B*C=42, we can eliminate a lot of possibilities. With this information we can keep reducing the amount of possible values until we only have a very small selection left. However, if we now randomly pick one of these possibilities for a variable, the amount of possibilities for the next variable is still very large.

If we again make the problem arc consistent, then the amount of possibilities gets again reduced to a very small amount. This is the reason why -arc works and -iconst 2 does not. Since there is only one solution and 3 variables and we have 4 states visited, we cannot reduce the amount of states visited by adding more flags. So adding a heuristic will not do anything in this case.

One of the reasons -fc doesn't work as well as -arc is because in the beginning arc consistency can already eliminate 80 % of all the possibilities while fc first has to try them to find out they don't work. Node consistency doesn't work because at the start there are no constraint's to limit the amount of possibilities.

## Solving a small set of equations

There are five solutions.
We can rewrite this problem to make it similar to the problem we saw in the previous one. Therefore the reason that -arc works is the same.
We see here that we have visited 19 states with this problem, which is very close to the minimum. So adding more heuristics/propagation techniques will not solve the problem in less states visited.

## Chain of trivial equations

There are 100 solutions.
When we run the problem without any flags, the number of states is 2601. Adding any flags does nothing to improve the amount of states visited. The reason is that every variable ranges from A to Z. The amount of solutions is 100. So the amount of states we visited is very close to 100*26. So we cannot reduce this by using other heuristics or propagation techniques.

ChainA and ChainZ both have 1 solution. However the main difference is that ChainA always visits 2502 states (except when we use -iconst 2) and ChainZ only visits 27 states.

the solver goes through each constraint from the bottom to the top. So with ChainA, it tries every value for Z, but when it arrives at A, it sees that a solution is not possible. Because A is already assigned, we do not need to do this. Therefore the amount of states visited is 2601-99=2502.

We can reduce this by adding the -iconst 2 flag. This way it immediately assigns A to 42 and then it only visits 27 states in total. Except for the very first state, there is no need to keep the problem arc consistent, since it cannot eliminate any more possibilities after the first state. This is why -arc does not work as good as -iconst 2.

With ChainZ it immediately assigns 42 to Z. Then it simply checks the constraints and assigns it to the other variables as well. This is why it always takes only 27 states.

## 1.4 Cryptarithmetic puzzles

When we run the program for SEND+MORE=MONEY, we get the solution:
S = 9
E = 5
N = 6
D = 7
M = 1
O = 0
R = 8
Y = 2

c1 = 1
c2 = 1
c3 = 0

This is indeed the same solution as provided in the exercise.
With -arc we can again eliminate a lot of possibilities. With -mrv, we can reduce the amount of states visited even more. e.g. If we have two variables x1 and x2. X1 has two values left to be chosen from, while x2 has 5 left.

By using -mrv, we continue our problem with x1. This is because, if we find an inconsistency with x1, we can write off 50% of the values, while if we had chosen x2, it would only be 20%. This is why both -arc and -mrv are very good for this problem. With the other puzzles it is the same.

## Finding the first 20 primes

The program gives one solution.
We only use the flag -iconst 2.
The reason for this is because in the constraints, it checks whether the number is divisible by either 1 or itself. With -iconst 2, we check this before the program runs(which is why we use -iconst 2 and not -arc). So then we remove all of the non prime numbers already, which is why this is the best to use. With -iconst 2 it visits 21 which is very close to 1*20.

## Solving Sudokus

Again the program finds one solution.
It is obvious that we need to use both -iconst 2 and -arc, since this is the whole idea of solving Sudokus: keep removing possibilities.
If we arrive at a point where we have to choose, we want to make a smart choice. For this reason we include the -mrv flag. That way more possibilities are eliminated. If we run it with -iconst 2 and -arc it visits 4099 states, while it visits only 2383 states if we include the -mrv flag(for the hard Sudoku).

The difference between the hard Sudoku and the smiley Sudoku is pretty big(4099 states visited as opposed to 82 states visited). This is probably because with the hard Sudoku, the solver has to choose between values at a certain point. While with the smiley Sudoku, it can keep removing possibilities and there will always be a cell in the Sudoku with only one possibility left.

## n-queens problem (again)

solutions for n queens:
n = 4: 2 solutions, 9 states visited
n = 5: 10 solutions, 46 states visited
n = 6: 4 solutions, 41 states visited
n = 7: 40 solutions, 240 states visited
n = 8: 92 solutions, 655 states visited
n = 9: 352 solutions, 2622 states visited
n = 10: 724 solutions, 7401 states visited

For this we only used the flag -arc. -arc works best here, because we eliminate a lot of possible positions of the next queens to be placed.
We see that initially, there is nothing on the board, so -iconst 1 and -iconst 2 are useless here. -mrv and -deg do not help a lot as well, this is because of the nature of the problem: generally it does not help a lot to pick a queen that is most restricted in its movement for example.

## Exercise 2: Writing CSP descriptions

### Constraint graphs

There are 2 different solutions Solution 1:
A = 3
B = 3
C = 4
D = 2
E = 1

Solution 2:
A = 4
B = 4
C = 2
D = 3
E = 1

We see that this problem has a lot of constraints. So we can eliminate a lot of possibilities by running this program with the -arc flag. If we run this program with -arc, we get 10 states visited. This is the minimum amount of states we can visit, because it has 5 variables and 2 solutions, and 5*2=10. Therefore it is not necessary to run it with any more flags, because it cannot get any better.

Using only -iconst 2, it visits 13 states. This is because before the program runs, not all possibilities can be eliminated such that the program does not have to choose randomly. Maybe for bigger cases it would be useful to combine both -iconst 2 and -arc, but for this problem there is no difference in states visited, because we are already at the minimum.

### Magic squares

There are in total 7040 solutions for n=4.
To solve this we use the following flags: -mrv, -deg, -arc.

We use arc to eliminate a lot of possibilities (partly because of alldiff).
We use -mrv to pick a certain set of possibilities which is most likely to eliminate the most amount of states.
It is possible that there are still multiple possibilities after we run -mrv. Which is why we use -deg to pick the best one out of those, instead of picking a random one.

### Boolean satisfiability (SAT)

There were in total 9 solutions, one of the solutions is:
x1 = 1

x2 = 0
x3 = 1
x4 = 1
x5 = 1

We run this program with -arc because if you assign one value you can use resolution to exterminate other values, this is also the reason why -iconst 2 doesn't work that well (because it first needs to pick a random one before making it arc consistent).

## Program files for Constrain Graph

### ConstraintGraph.csp

```
#The set of variables of the CSP
variables:
   A,B,C,D,E : integer;
#Here the domains are defined
domains:
   A,B,C,D,E <- [1, 2, 3, 4];
#Here are the constraints:
constraints:
   B >= A;
   A > D;
   C <> A;
   B <> C;
   C <> D + 1;
   C <> D;
   D > E;
   C > E;
# Here you can specify in how many solutions
# you are interested (all, 1, 2, 3, ...)
solutions: all
```

## Program files for Magic Squares

### MagicSquares.csp

```
#The set of variables of the CSP
variables:
   square[16] : integer;
#Here the domains are defined
domains:
   square <- [1..16];
#Here are the constraints:
constraints:
   alldiff(square);
   #rows and collums
   forall (i in [0,1,2,3])
            square[i*4] + square[i*4+1] + square[i*4 + 2] +
                square[i*4 + 3] = 34;
            square[i] + square[i+4] + square[i+8] + square[i+12] = 34;
   end

   #diagonals
   square[0] + square[5] + square[10] + square[15] = 34;
   square[3] + square[6] + square[9] + square[12] = 34;
# Here you can specify in how many solutions
# you are interested (all, 1, 2, 3, ...)
solutions: all
```

## Program files for SAT

### SAT.csp

```
#The set of variables of the CSP
variables:
    x1,x2,x3,x4,x5 : integer;
#Here the domains are defined
domains:
    x1,x2,x3,x4,x5 <- [0,1];
#Here are the constraints:
constraints:
    max(max(x1, x2),abs(x3 - 1)) = 1;
    max(max(abs(x1 -1),abs(x2-1)),abs(x4-1)) = 1;
    max(max(x1, abs(x2-1)),abs(x5-1)) = 1;
    max(max(abs(x1 - 1), x3), abs(x4 -1)) = 1;
    max(max(x1, abs(x3 -1)), x5) = 1;
    max(max(x1, abs(x4 -1)), x5) = 1;
    max(max(x2, x4),x5) = 1;
    max(max(abs(x3 -1), x4), abs(x5 -1)) = 1;
# Here you can specify in how many solutions
# you are interested (all, 1, 2, 3, ...)
solutions: all
```