INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES RENNES

Computer Science Department

CH8: CONSTRAINT (LOGIC) PROGRAMMING

A BRIEF INTRODUCTION
(NOT COVERED DURING LECTURES)

Prof. Mireille Ducassé

Last update: April 2023

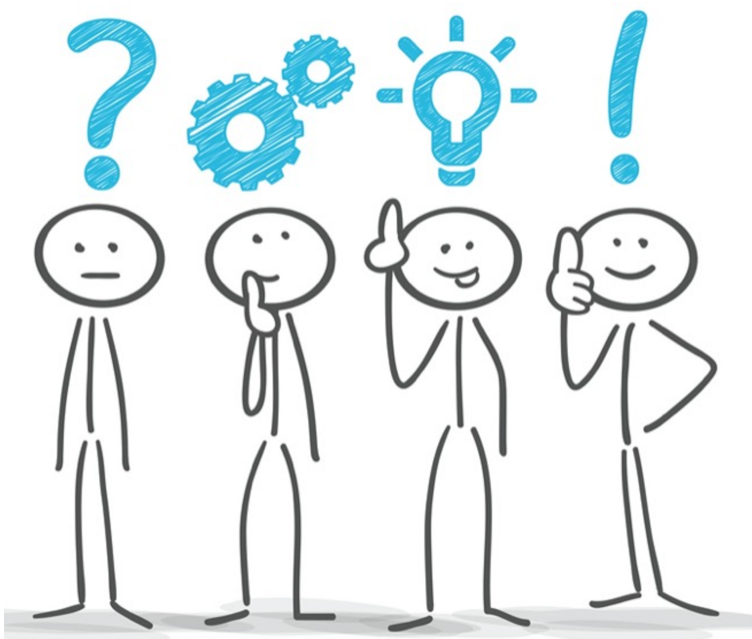
Sudoku 1/14

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

The challenge is to fill the grid with numbers from 1 to 9 such that every row, every column, and every 3x3 sub-grid contains the digits 1 to 9.

- Fill in **1 slot**, **explain why this is a valid step**

Allow yourself some time to search before looking at solutions 😀



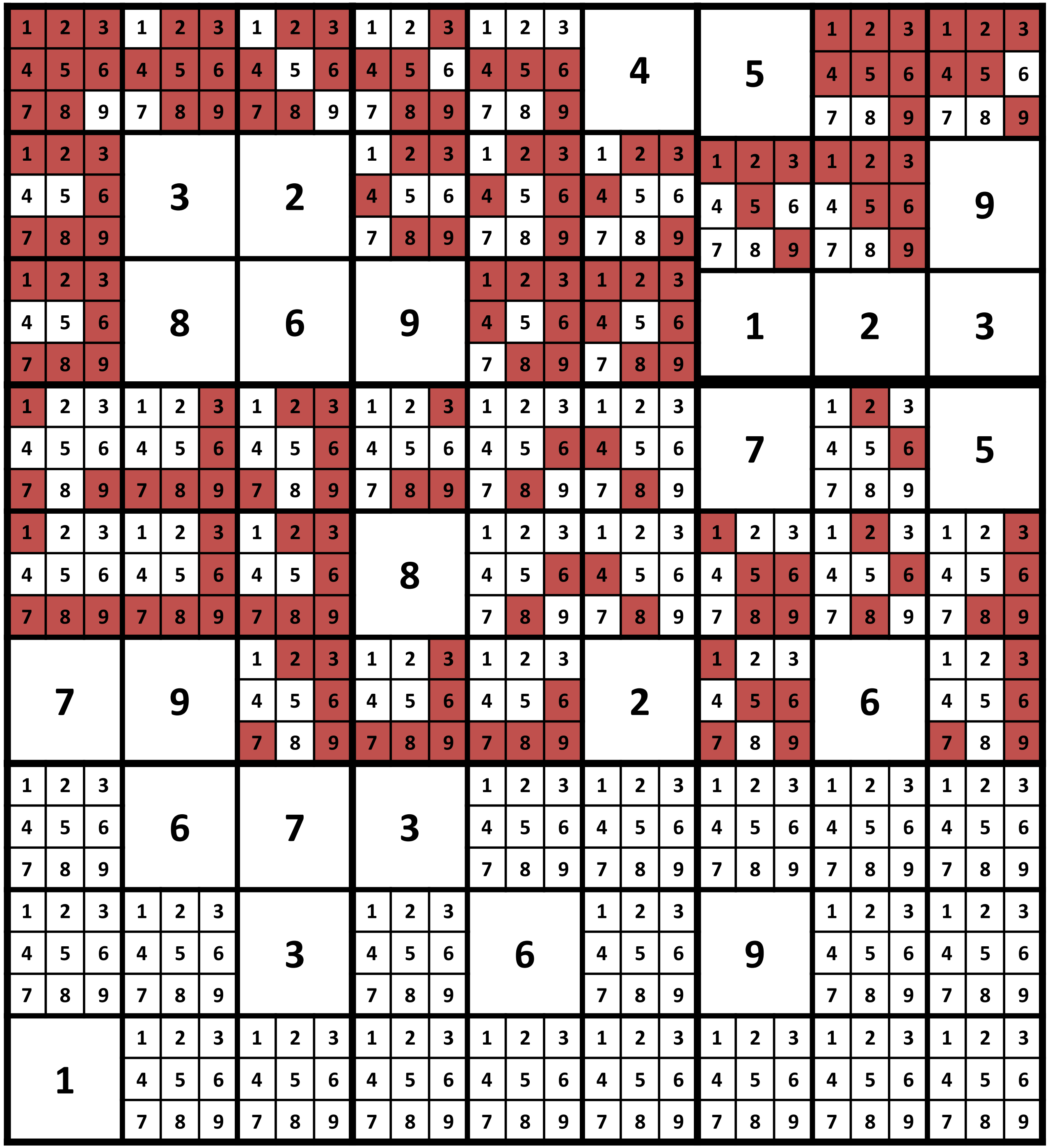
Sudoku 1bis/14

	7				4	5		
	3	2						9
4	8	6	9			1	2	3
						7		5
			[illegible]					
7	9				2		6	
	6	7	3					
		3		6		9		
1								

- Column constraints : 7
- Line constraint + block constraint : 4

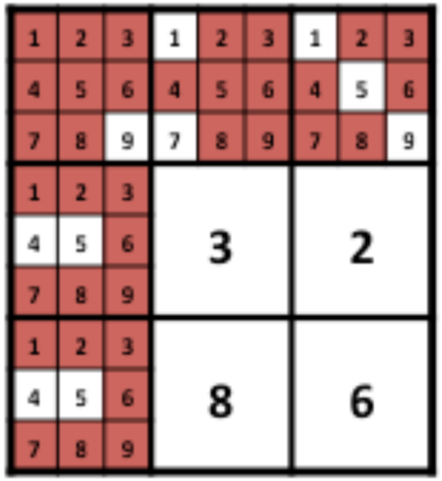
Sudoku 2/14

1. In the next grid, finish to remove the impossible values due to the initially given values
   – Has the order an impact on the amount of values removed ?
2. On what to reason next ?
   – A square ? A line ? A column ?
   – Which one ? And why ?



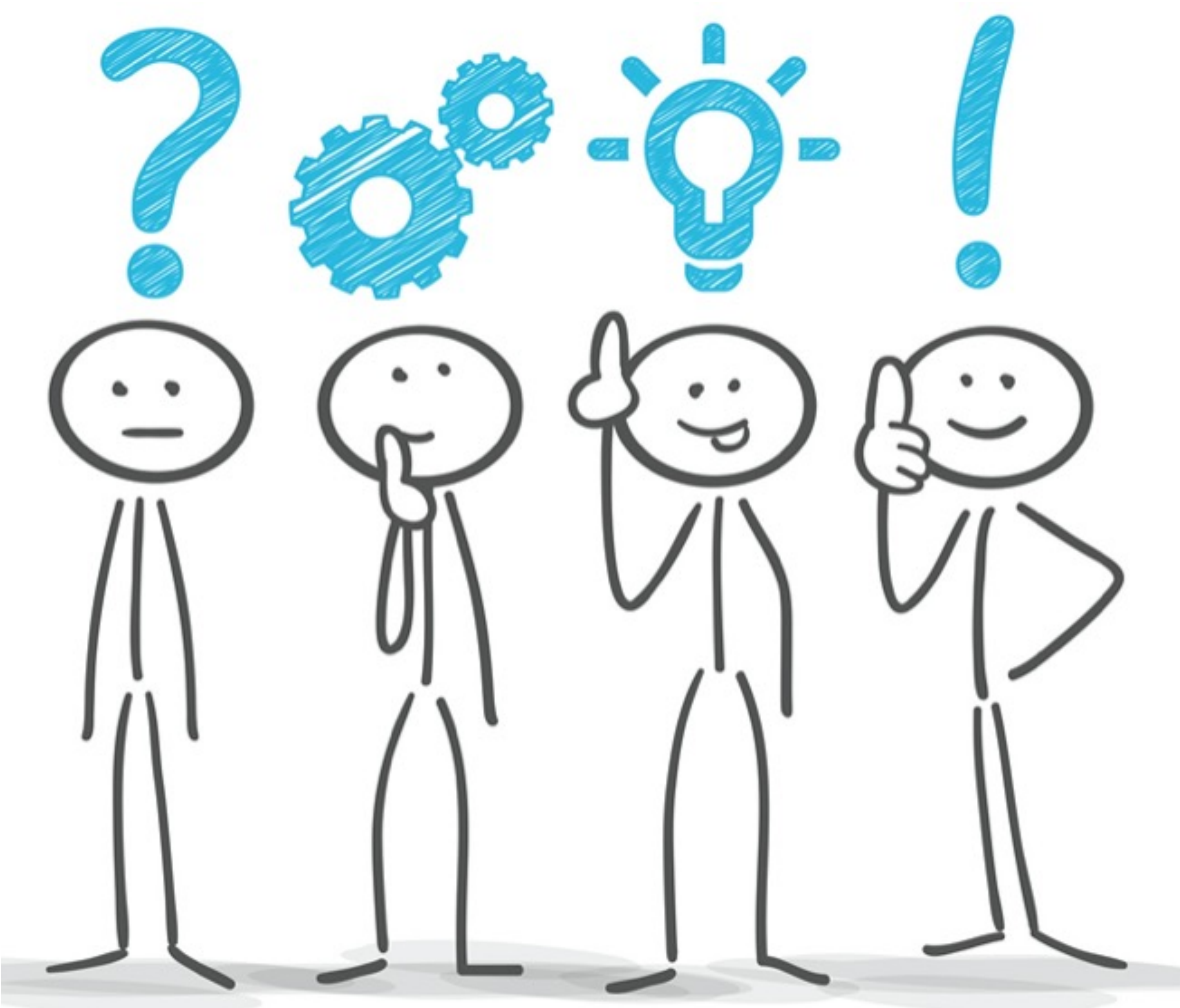
3/14

Sudoku 4/14

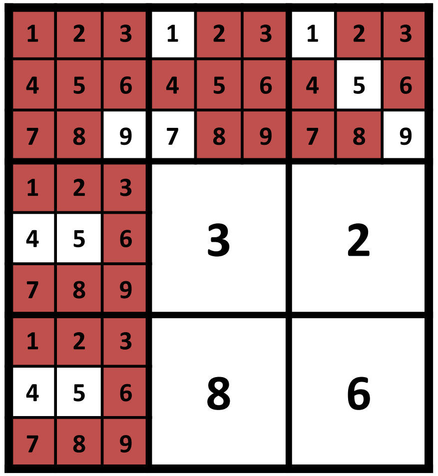


1. Assume we take the upper left most square, what can be deduced ?
    1. .
    2. .
    3. .
    4. .
    5. .

Allow yourself some time to search before looking at solutions 😀



Sudoku 4bis/14



What can be deduced ?

1. l1.c1 can only contain 9
   - remove 9 from the rest of line 1, col 1, square 1
2. 1 can only be in the first line
   - remove 1 from line 1 in other squares
3. 7 can only be in l1.c2
   - 1 can thus not be in l1.c2
   - remove 7 from the rest of line 1, col 2
4. 1 can only be in l1.c3
   - 5 cannot be in l1.c3
   - remove 1 from the rest of c3
5. 4 and 5 can only be in col 1
   1. remove 4 and 5 from col 1 in other squares

What would have happened if we had taken step 3 in second ?

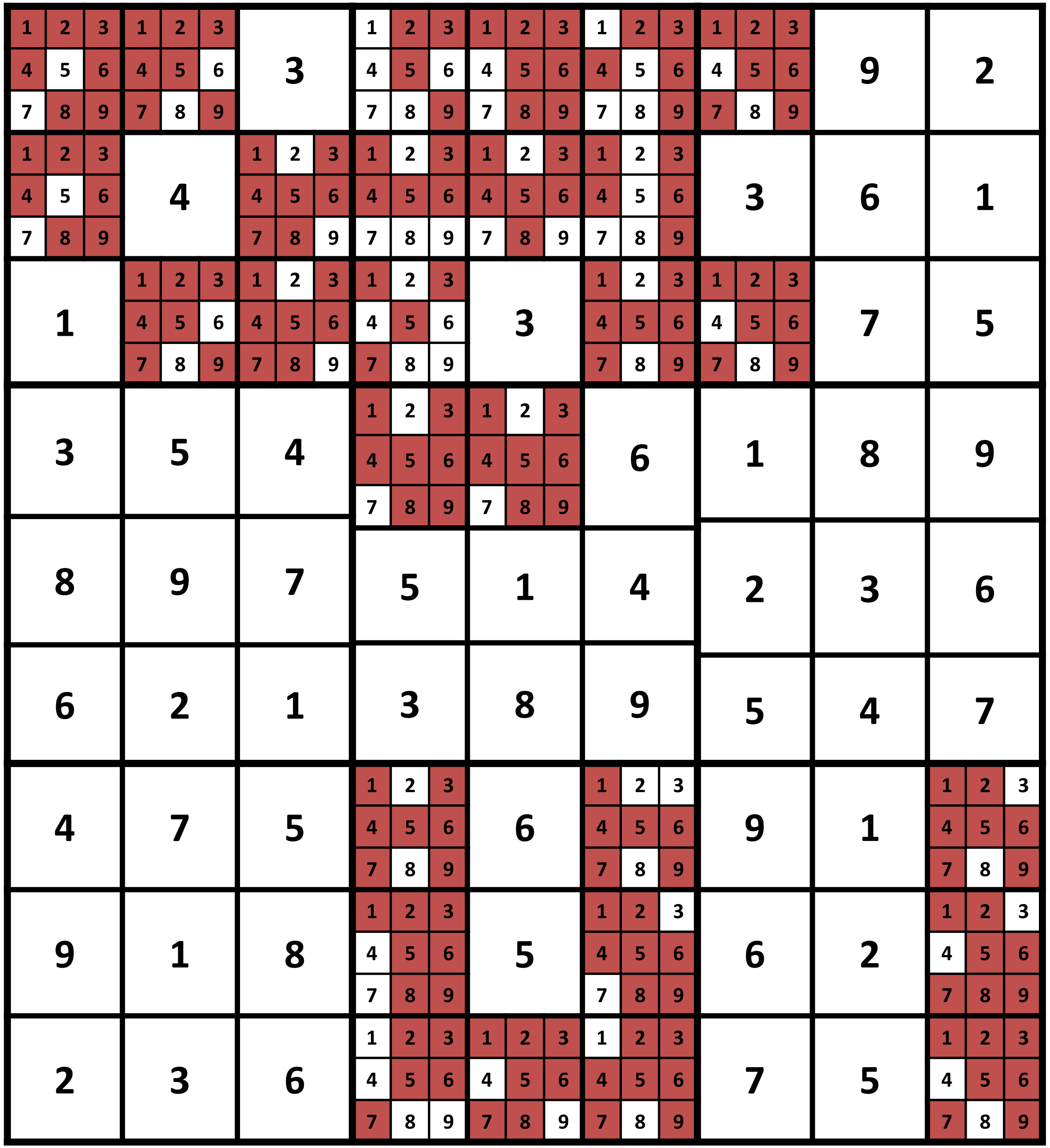
Sudoku 5/14

1. What happens if only half of the numbers are initially given ?

2. What if the given numbers are randomly changed ?

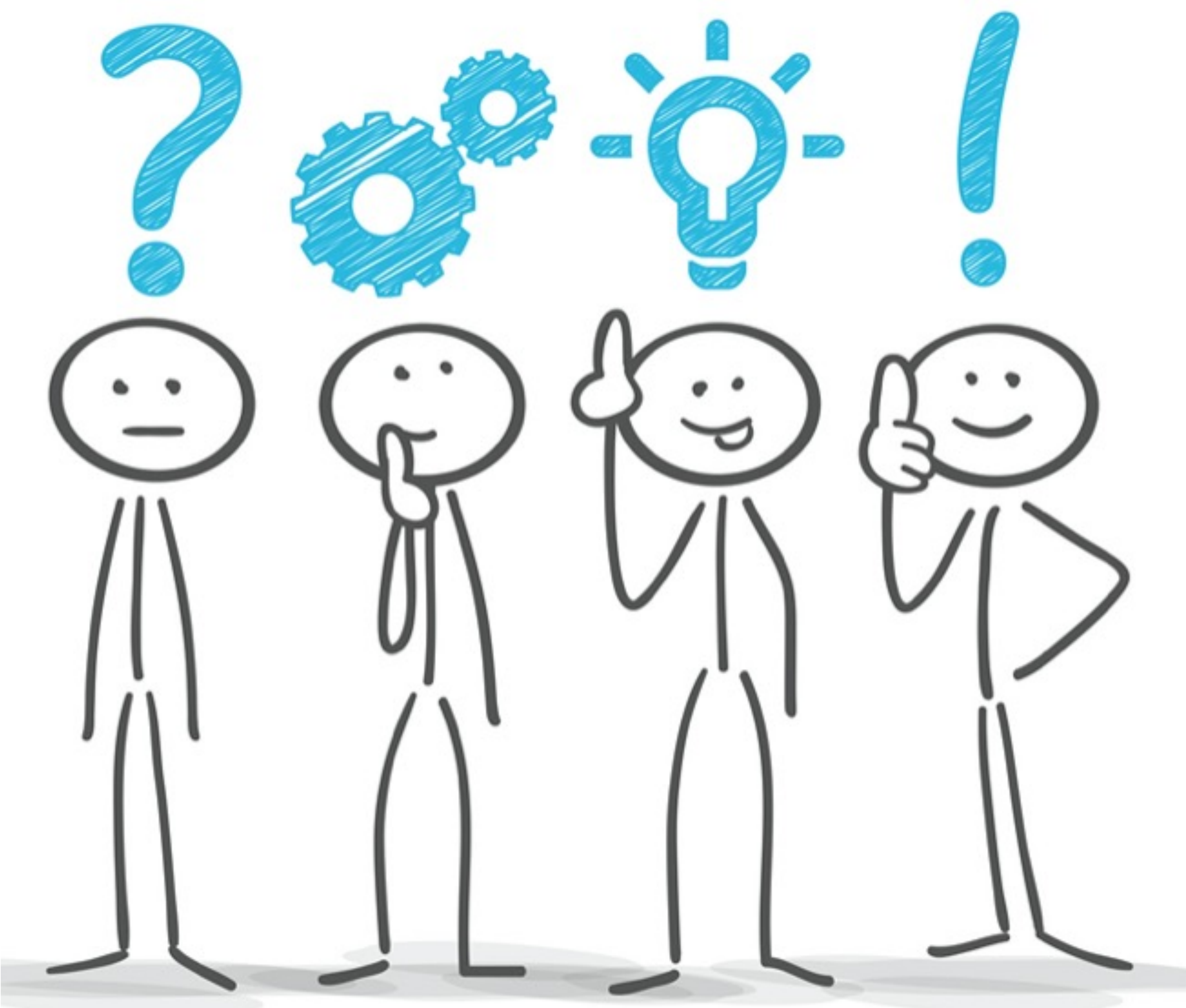
Sudoku 6/14

- In the next grid, all impossible values have been removed
- We should **try values**
  - on which cell(s) does it seem the most promising and why ?



7/14

Allow yourself some time to search before looking at solutions 😀



Sudoku 6bis/14

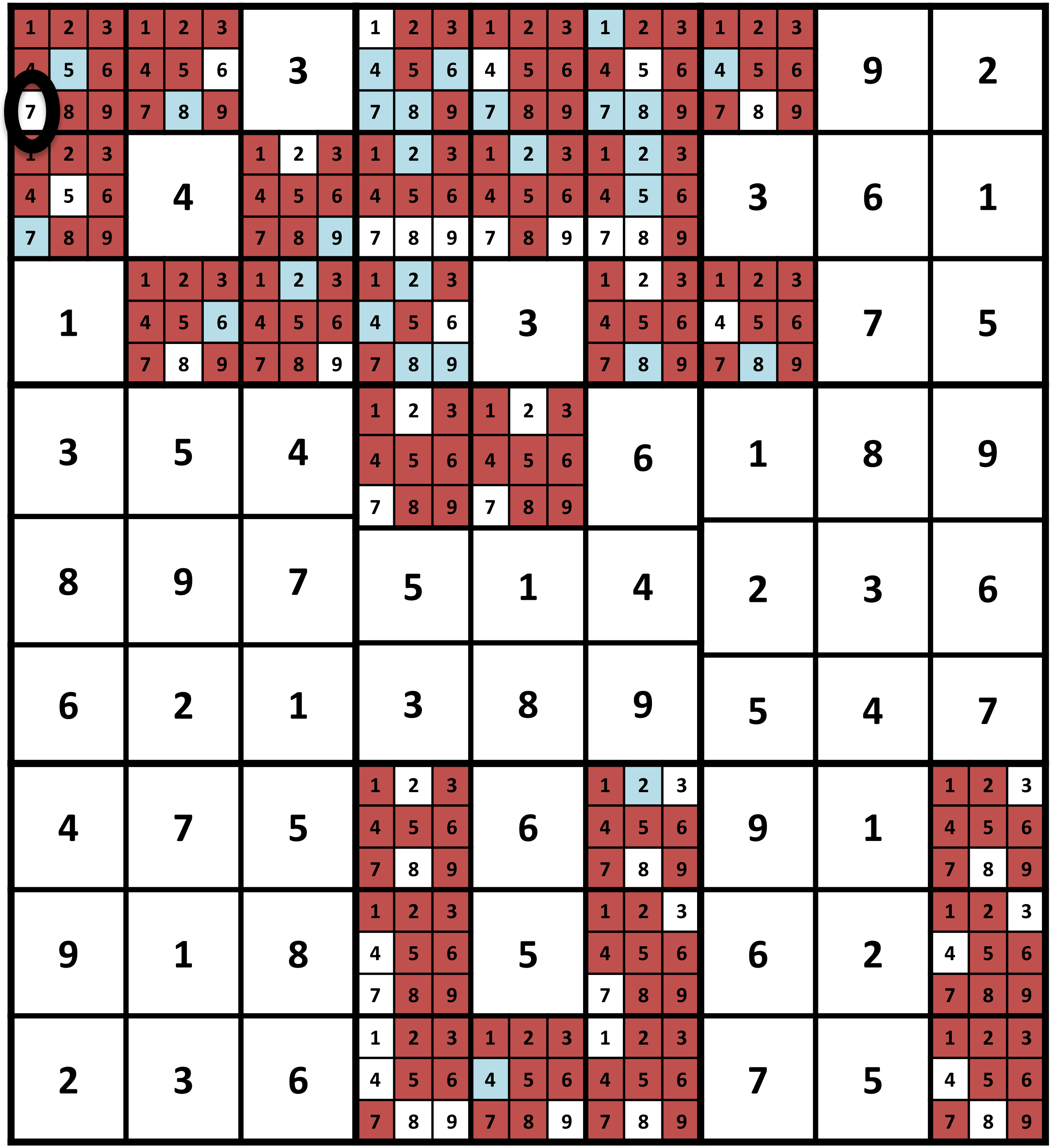
Cell(s) where it seems most promising to try values

- those with only 2 values left
- those with a value that appears often “as possible” in other cells
- ...

Sudoku 8/14

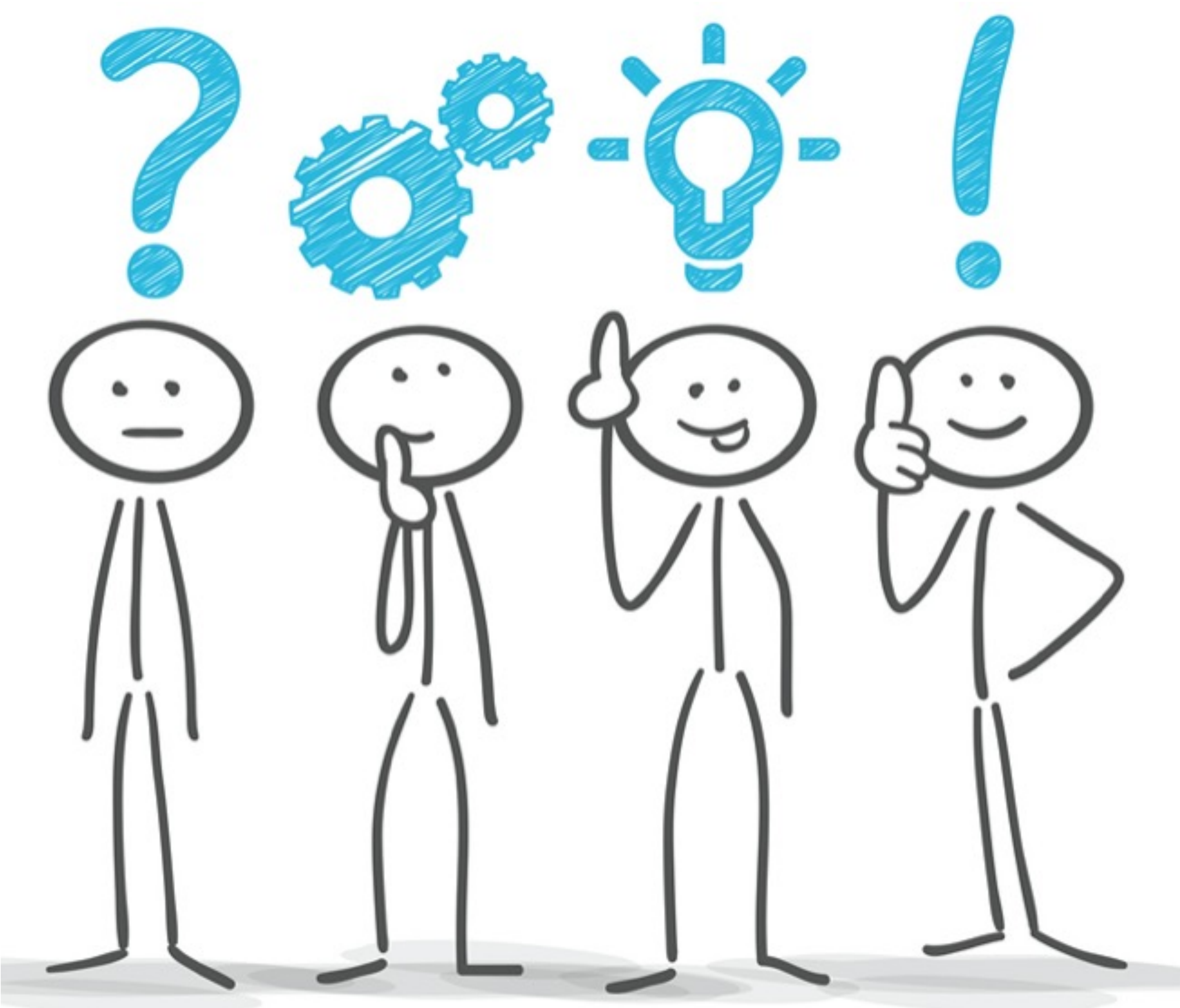
- In the next grid we have tried 7, a possible value, for l1.c1 and removed all impossible values
  - what can be noticed ?

- Try 7 at cell l2.c4 and propagate
  - what happens ?

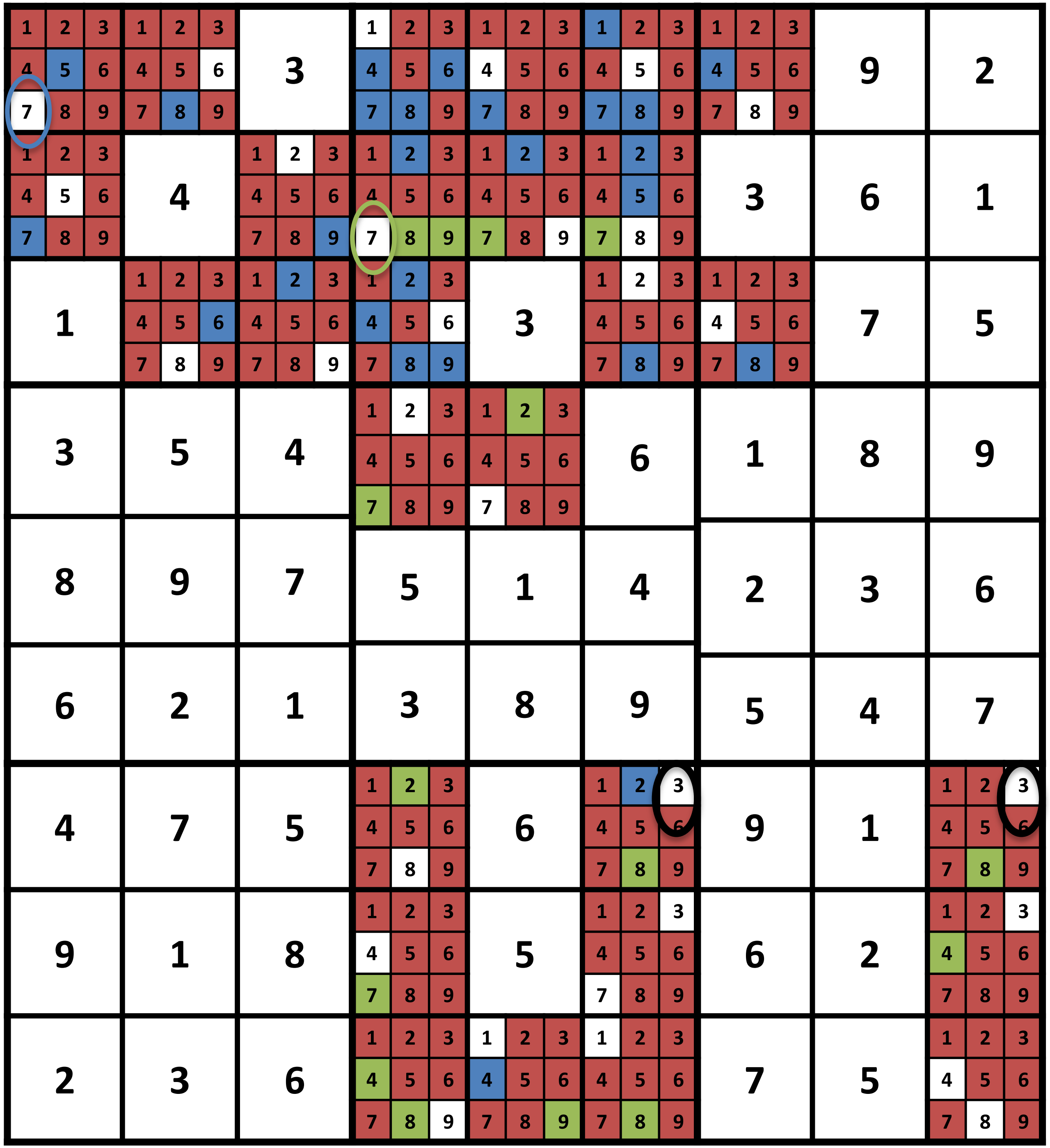


9/14

Allow yourself some time to search before looking at solutions 😀



9bis/14



Failure !
We need to backtrack

Remarks

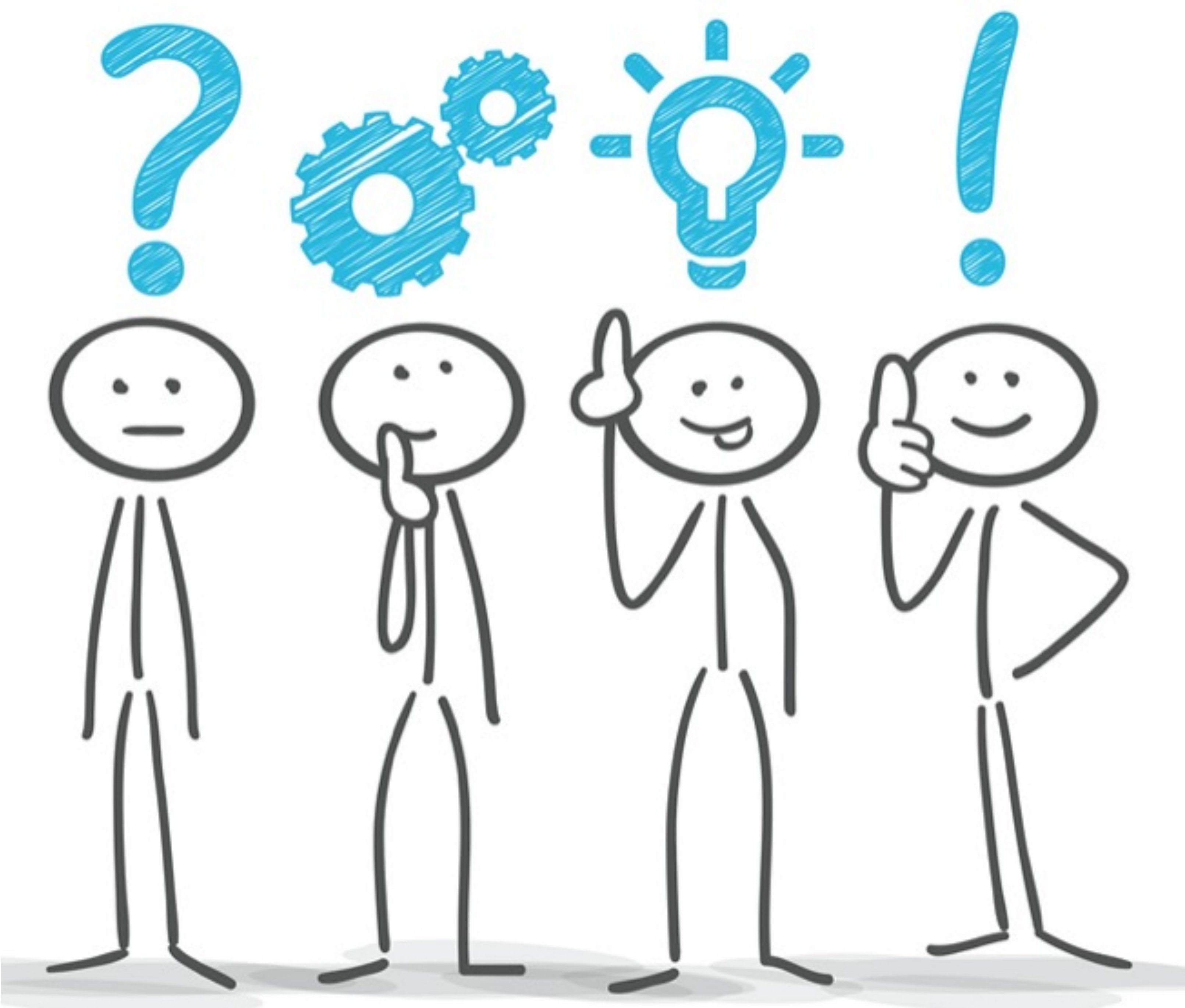
- Removing all the impossible values does not necessarily lead to a single solution
  - Failures can occur
  - Several values per slots may still be possible
    - Values have to be tried
- Trying a value is in general insufficient
  - Propagating inconsistent values is necessary

➢ **<u>In the general case, the two aspects have to be executed in turn</u>**

Sudoku 10/14

- What are the parameters of the previous reasoning ?
  - How are chosen the particular objects to reason upon ?
    1. .
    2. .
    3. ...
  - Which actions are used ?
    1. .
    2. .
    3. .
    4. .

Allow yourself some time to search before looking at solutions 😀



Sudoku 10bis/14

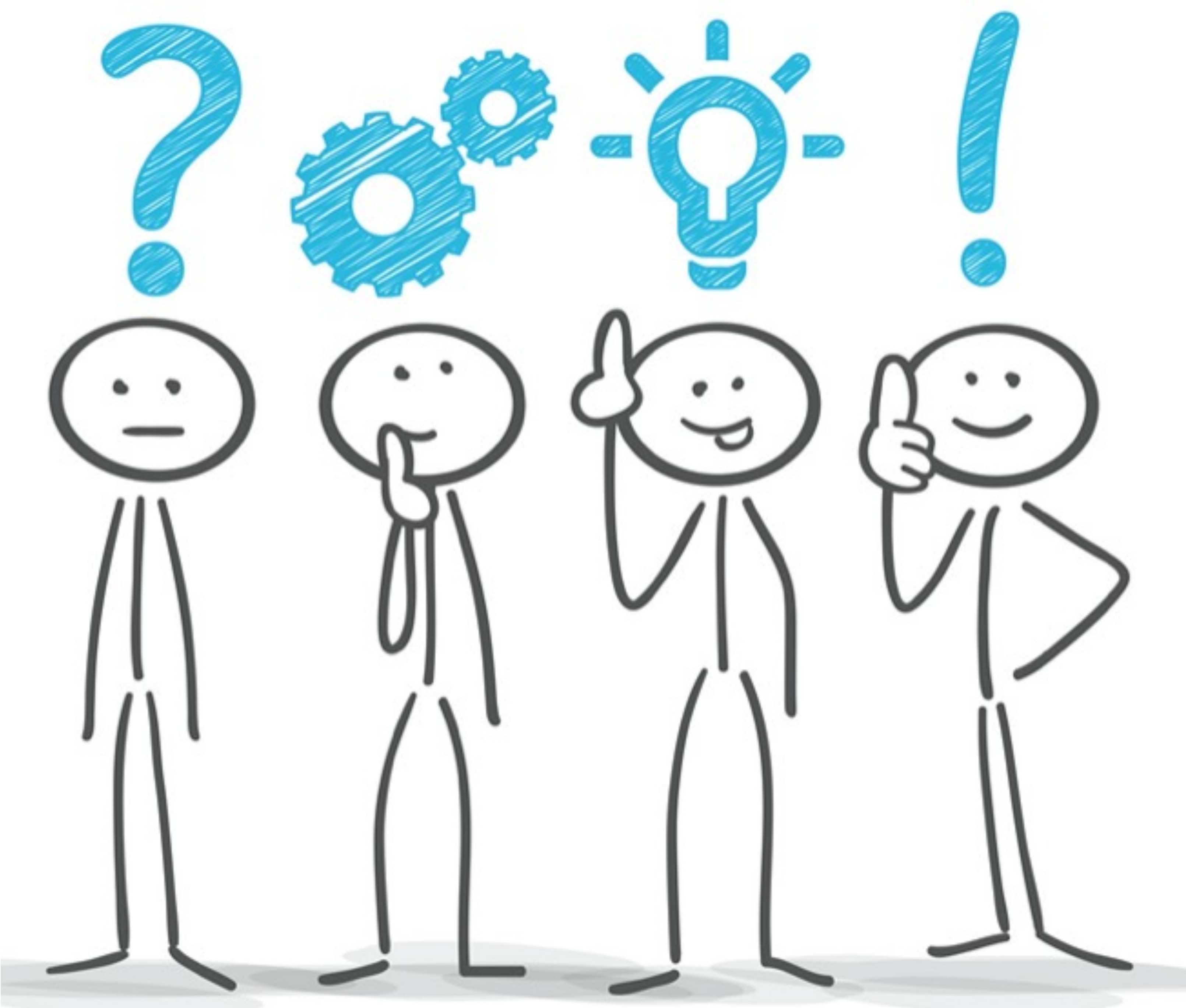
- What are the parameters of the previous reasoning ?
  - How are chosen the particular objects to reason upon ?
    1. the most constrained, or
    2. the first one in a line, or
    3. ...
  - Which actions are used ?
    1. detect only possible values at a given slot, line, column, square using different heuristics
    2. remove impossible values at a given place, line, column, square propagating constraints
    3. try a value at a given slot
    4. backtrack on failures
    5. ...

Sudoku 11/14

- What is difficult for human-beings ?

- What is difficult for computers ?

Allow yourself some time to search before looking at solutions 😀



Sudoku 11bis/14

- What is difficult for human-beings ?
  - to choose heuristics and places to reason upon
  - to apply heuristics consistently
  - to remember what has already been tried
  - to backtrack (what to undo? in which order ? until where ?)

- What is difficult for computers ?
  - to choose heuristics and places to reason upon

Sudoku: ~50 lines of a C++ program of 262 lines

```
static int disambiguate_board(int board[9][9]) {
 int game_solved = 0;
 int changed = 1;
 int invalid = 0;

 while ((changed) && (!invalid)) {
  game_solved = 0;
  changed = 0;
  for (int i = 0; i < 3; ++i) {
   for (int j = 0; j < 3; ++j) {
    int square_base_y = i*3;
    int square_base_x = j*3;
    for (int k = 0; k < 3; ++k) {
     for (int l = 0; l < 3; ++l) {
      int definite = 0;
      for (int m = 1; m <= 9; ++m) {
       if (onlies[m] == board[square_base_y+k][square_base_x+l]) definite = m;
      }
      if (definite) {
       for (int n = 0; n < 3; ++n) {
        for (int o = 0; o < 3; ++o) {
         if ((n != k) || (o != l)) {
          int before = board[square_base_y+n][square_base_x+o];
          board[square_base_y+n][square_base_x+o] &= negates[definite];
          if (before != board[square_base_y+n][square_base_x+o]) changed++;
          if (board[square_base_y+n][square_base_x+o] == 0x000) invalid++;
  }}}}}}}}
  for (int i = 0; i < 9; ++i) {
   for (int j = 0; j < 9; ++j) {
    int definite = 0;
    for (int m = 1; m <= 9; ++m) {
     if (onlies[m] == board[i][j]) definite = m;
    }
    if (definite) {
     for (int k = 0; k < 9; ++k) {
      if (k != i) {
       int before = board[k][j];
       board[k][j] &= negates[definite];
       if (before != board[k][j]) changed++;
       if (board[k][j] == 0x000) invalid++;
      }
      if (k != j) {
       int before = board[i][k];
       board[i][k] &= negates[definite];
       if (before != board[i][k]) changed++;
       if (board[i][k] == 0x000) invalid++;
  }}}}}
  for (int i = 0; i < 3; ++i) {
   for (int j = 0; j < 3; ++j) {
    for (int m = 1; m <= 9; ++m) {
     int count = 0;
     int posx = -1;
     int posy = -1;
     for (int k = 0; k < 3; ++k) {
      for (int l = 0; l < 3; ++l) {
       int y = (i*3)+k;
       int x = (j*3)+l;
       if (board[y][x] & onlies[m]) {
        posy = y;
        posx = x;
        count++;
     }}}
...
```

Note the number of nested loops !!

The **whole** ECLiPSe-CLP demo program

```
:- lib(ic).
:- import alldifferent/1 from ic_global.

solve(SudokuName) :-
        problem(SudokuName, Board),
        print_board(Board),
        sudoku(Board),
        print_board(Board).

sudoku(Board) :-
        dim(Board, [9,9]),
        Board :: 1..9,
        col_and_rows_all_diff(Board),
        sub_square_all_diff(Board),
        labeling(Board).

col_and_rows_all_diff(Board) :-
        (  for(I, 1, 9),
          param(Board)
        do
          Row is Board[I, 1..9],
          alldifferent(Row),
          Col is Board[1..9, I],
          alldifferent(Col)
        ).
```

```
sub_square_all_diff(Board) :-
        (  multifor([I, J], 1, 9, 3),
          param(Board)
        do
          (  multifor([K, L], 0, 2),
            param(Board, I, J),
            foreach(X, SubSquare)
          do
            X is Board[I+K, J+L]
          ),
          alldifferent(SubSquare)
        ).
```

Two nested loops only

```
print_board(Board) :-
        dim(Board, [9,9]),
        ( for(I,1,9), param(Board) do
           ( for(J,1,9), param(Board,I) do
                   X is Board[I,J],
             ( var(X) -> write("  _") ; printf(" %2d",
        [X]) )
           ),
           nl
        ),
        nl.

%---------------------------------------------
% Sample data

problem(sudoku1, [](
    [](_, _, 2, _, _, 5, _, 7, 9),
    [](1, _, 5, _, _, 3, _, _, _),
    [](_, _, _, _, _, _, 6, _, _),
    [](_, 1, _, 4, _, _, 9, _, _),
    [](_, 9, _, _, _, _, _, 8, _),
    [](_, _, 4, _, _, 9, _, 1, _),
    [](_, _, 9, _, _, _, _, _, _),
    [](_, _, _, 1, _, _, 3, _, 6),
    [](6, 8, _, 3, _, _, 4, _, _))).
```

Sudoku 14/14

- Which program is easier to understand ?
- Which one is easier to maintain ?
- Which one is easier to tune ?
- Which version is easier to write?
- Where does the “miracle” come from ?
  - The CLP version states **what** has to be done
    - A **solver** addresses **how** it is achieved, and in an **optimized** way
  - A lot of research and work has been invested in existing solvers : no need to re-invent the wheel

What is a constraint?

- Let X1, . . . , Xn be a ***finite*** sequence of **variables**
- each associated with a set of possible values called its **domain**, D1, . . . , Dn

- A constraint on X1, . . . , Xn is a **relation,** included in D1 × · · · × Dn

- Rm: constraint ≣ relation ≣ equation

Constraint programming

1. **Modeling**: Formulate the problem as a finite set of constraints
   – a Constraint Satisfaction Problem (CSP)
2. **Solving**: Solve the CSP
   – if possible by using a constraint programming system
3. **Mapping**: Map the solution to the CSP to a solution to the original problem

The Constraint Satisfaction Problem

- An instance of the Constraint Satisfaction Problem (CSP) consists of
  - *a finite* set of variables, X1, . . . ,Xn,
  - for each variable Xi a set of values, Di, called its domain,
  - *a finite* set of constraints. Each restricts the values that the variables can simultaneously take.
    - Examples: $x \neq y$. $x+y \leq z$
- A total assignment maps each variable to an element in its domain.
  - It is a total function
- *A* solution to an instance of the constraint satisfaction problem is a total assignment that satisfies **all** the constraints.

The Constraint Satisfaction Problem

Given an instance of CSP the goal is usually **one of**

- determine whether the instance **has any** solutions
  - In that case the CSP is said **consistent** or **feasible**
- find **any** solution
- find **all** solutions
- find a solution that **optimizes** some given objective function
- Determine that there is no solution (refutation)

- Note that in many cases finding a solution (even if not optimal) is already very useful (and can already be a challenge)

Constraint **Logic** Programming

- The advantage of Constraint **Logic** Programming is that it offers both
  - backtracking
    - traversal of the **control flow** search space
  - constraint propagation
    - filtering/pruning of the **data** space

“Constraint” problems

- Puzzles
- **Planning and Scheduling**
  - Assignment problems
  - Jobshop Scheduling
  - Warehouse location
  - Ecologist traveler
  - Covering a square with smaller squares of different sizes.
  - Scheduling players for sport tournaments
  - Computing a staff roster
  - Airline crew scheduling
  - ...

Characteristics of “constraint” problems

- There are **no general methods or algorithms**
  - NP-completeness (cf complexity course)
- Different strategies and heuristics have to be tested
  - **different input data may lead to different strategies**
- Requirements are quickly changing
  - Programs should be flexible enough to adapt to these changes rapidly

Remarks

- Solvers significantly ease the resolution of "constraint problems"
  - Especially important for large complex problems

but

- Each solver has its own algorithms and heuristics in order to propagate constraints
- These algorithms and heuristics are in general quite complex
  - Choosing proper solvers is an issue
  - Choosing a relevant model is an issue
  - Using proper strategies inside the solvers is also an issue

Background knowledge is mandatory.

- **I recommend to follow some course, for example on ECLiPSE ELearning Website**
  - video lectures, slides, handouts and other material
  - 20 (!) chapters
  - **An impressive lists of applications**
  - by Helmut Simonis
  - http://www.eclipseclp.org/ELearning/