Constraint Processing Assignment

The deadline for submitting your report is **Monday 5 December, noon**. We expect you to deliver a printout of your report in the student mailbox (located in the printer room of 200A).

In this project you are presented with a number of constraint problems. You should solve these problems using the constraint solver that is part of *SICStus Prolog*. This solver employs a hybrid algorithm that combines *backtracking* with *arc consistency checking*. Conceptually, it solves a constraint problem as follows. Initially, an arc consistency algorithm such as AC3 is run. When this fails to find a solution, a free variable is assigned one of the remaining elements in its domain, after which AC3 is run once more. These steps are iterated until a solution is found. If an inconsistency is raised (for example, a variable ends up having an empty domain), the system *backtracks* over the assignments just as described in the course lectures on *forward checking* and *lookahead checking*.

In contrast to the conventions we observed throughout the course material, this solver imposes no restrictions on the number of variables that may appear within a constraint. Because of this, you can and may describe constraints involving more than two variables. Note that this implies that the algorithms employed by the solver are not the same ones as described in the course material, but rather extensions.

All the necessary software and its manual can be found at the following URL.

`https://dtai.cs.kuleuven.be/education/ai/Projects/constraints/`

- The web page of the constraint processing tool has on the bottom right a field labelled Submitted variables & constraints. Your report should *include the content of this field for each exercise* (ensure that you first clicked on Show variables and constraints).
- You should *explicitly mention the number of solutions for each exercise.* Unless the number of solutions is far too great, you should also include the output of the tool (though make sure the Debugging check box is turned off).
- If the exercise requires you to discuss something, you should ensure that you clearly separate your observations from your explanation of those observations.

- Your project report should contain natural language descriptions that capture the meaning of the non-trivial constraints you used when modelling the problems. For instance, for a constraint in an n-queens puzzle, you could state: Two queens should not be on the same column, nor should they be on the same diagonal.

In case of technical problems with the software you can contact: `dries.vandaele@cs.kuleuven.be`.

1 Magic square: 2x2

A magic square of order n is an nxn square in which the numbers 1 to n^2 are positioned in such a way that the sum of the columns, rows, and diagonals is one and the same. The figure below is an example of a magic square of order 5 for which the sum is 65.

11	24	7	20	3
4	12	25	8	16
17	5	13	21	9
10	18	1	14	22
23	6	19	2	15

Each 1x1 square trivially satisfies the requirements of a magic square. Do there exist magic squares of order 2? Answer this question using the tool. Employ the variables A, B, C en D, and make them represent the content of the square as follows:

A	B
C	D

2 Magic square: 3x3

Now search for a magic square of order 3. Rely on an indexed variable V, that represents the content of the square as follows:

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

3 Sudoku

Sudoku is a popular puzzle. The objective is to fill a 9x9 grid with the numbers 1 to 9 while satisfying a number of constraints. Concretely, each number can only be used once in each row, column, and in each of the nine non-overlapping 3x3 squares that are contained within the larger grid. Appendix A contains a selection of Sudoku puzzles divided in two groups (easy, difficult) based on their perceived complexity. Use these in the following exercise:

1. Represent Sudoku as a constraint problem and encode your representation within the constraint solving tool.

2. Solve the puzzles we have provided. Do you observe any differences when solving puzzles that have a different difficulty level? **Note:** The tool returns the *complete* run-time; this is both the time it took to iterate over all the indices in order to impose the constraints, as well as the time it took to satisfy those constraints. A better way to determine how challenging a constraint problem is to our system involves considering the number of times a conflict occurred (i.e. a variable ends up with an empty domain resulting in *backtracking*) during the solving step. This figure is also reported by the tool. Can you explain your observations using your understanding of constraint solving? Discuss.

3. The tool offers a number of heuristics. Compare the `leftmost` to the `first-fail + constraint count` heuristics, and discuss.

4. When solving a Sudoku, a human player will often apply a number of strategies. The following website:

   `http://www.sudokudragon.com/sudokustrategy.htm`

   showcases a number of these strategies. The fourth strategy is the "*two out of three rule.*". Extend your formalisation with a constraint that explicitly encodes this strategy.

5. Does this extension affect the solutions you find? Discuss.

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

7. After including this extension, can you detect the influence of the employed heuristic? What about the difficulty level of the puzzle? Explain your observations to the best of your abilities.

Hints

The constraint solving tool only allows a single level of indexing. This means that you can not reference a square using its row and column index like: $(4, 5)$, but instead you have to assign each square a 1-dimensional coordinate. An example is shown in the following figure.

0	1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16	17
18	·	·	·					
				·	·	·	79	80

When you use this type of indexing, you can derive the row number of a square using:

$Row(i) = i \ / \ 9$ (where / is the operation that performs integer division);

The column number is derived using:

$Column(i) = i \bmod 9$ (where mod returns the remainder after division).

Once you have a general encoding for the Sudoku puzzle, you can of course load a concrete puzzle by equating certain variables to particular values. Because this is a fairly tiresome process, we offer a web service that assists you in this task:

`https://dtai.cs.kuleuven.be/education/ai/Projects/constraints/sudoku.php`

The usage instructions are provided on the web page. **Note that this particular tool only works if you perform your variable indexing in the exact same manner as described above!**

A Example Sudokus

A.1 Easy Sudokus

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

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

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

A.2 Difficult Sudokus

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

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

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