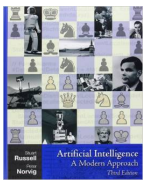
Deadline: 2014-10-31 23:59
Recitation (øvingsforelesning): 2014-10-21 16:15, G1

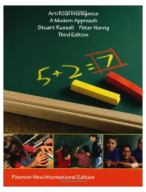
Solving Constraint Satisfaction Problems

Purpose: Gain hands-on experience with Constraint Satisfaction Problems (CSPs) by implementing a CSP solver and using it to solve Sudoku boards.

Note: The textbook, *Artificial Intelligence: A Modern Approach, 3rd edition*, exists in two versions:



"Blue version"



"Green version"

All references to chapters and figures in the textbook are given as 1.2.3 / 2.3.4, for the blue and the green version respectively.

Overview

In this assignment you will implement a general solver for Constraint Satisfaction Problems, specifically using *backtracking search* and the arc-consistency algorithm *AC-3*. You will then use this program to solve Sudoku boards of varying difficulty. Sudoku boards are easily representable as Constraint Satisfaction Problems, as discussed by the text book (Chapter 6.2.6 / 7.2.6).

You will be required to implement the CSP solver by yourself, and the source code must be well-commented and attached to your submission. To make sure you get off on the right foot, we provide you with *skeleton code* for Python and Java. This code suggests an interface for your CSP solver and initializes and populates the necessary data structures to represent the CSP. You are welcome to use and adapt this code in your submission, but it is not required.

Solving CSPs with Backtracking Search and AC-3

Using backtracking search to solve CSPs is described in Chapter 6.3 / 7.3 in the textbook, and summarized as pseudocode in Figure 6.5 / 7.5. The pseudocode references three functions that have not been specified any further by the textbook, namely SELECT-UNASSIGNED-VARIABLE, ORDER-DOMAIN-VALUES and INFERENCE. For this assignment, it is sufficient to have SELECT-UNASSIGNED-VARIABLE return any unassigned variable, and to have ORDER-DOMAIN-VALUES return any ordering of the possible values for the given variable. INFERENCE will be the arc consistency algorithm AC-3, as described in pseudocode in Figure 6.3 / 7.3.

Skeleton Code

The Python and Java skeleton code suggests which data structures to use to represent the CSP. As mentioned in the textbook in Chapter 6.1 / 7.1, a Constraint Satisfaction Problem can be represented as a tuple (X, D, C), where

- X is a set of *variables*,
- D is an assignment of a *domain* to each variable, where a domain is the set of legal values for the given variable,
- C is a set of *constraints*, where a constraint is a set of legal pairs of values for two given variables.

The skeleton code represents X, D and C as the following data structures:

- X is an *array* of variable names.
- D is a *hash table* that associates a *variable name* with its corresponding *array of legal values*.
- C is a hash table that associates a variable name i with *another* hash table, where this second hash table contains the constraints affecting variable i. Specifically, the second hash table associates the name of another variable j ($i \neq j$) with an *array of legal pairs of values* for the pair of variables (i, j).

In Python, the arrays, hash tables and legal-pairs-of-values are implemented as lists (`[]`), dictionaries (`{}`) and tuples (`()`) respectively. In Java, the arrays, hash tables and legal-pairs-of-values are implemented as `ArrayList`, `HashMap` and `Pair` objects respectively, where `Pair` is a custom class that holds two values.

The following printout from Python shows the contents of these data structures when the map coloring CSP from the textbook (Chapter 6.1.1 / 7.1.1) has been set up:

```
>>> print csp.variables
['WA', 'NT', 'Q', 'NSW', 'V', 'SA', 'T']
>>> print csp.domains
{'WA': ['red', 'green', 'blue'], 'Q': ['red', 'green', 'blue'], 'T':
['red', 'green', 'blue'], 'V': ['red', 'green', 'blue'], 'SA': ['red',
'green', 'blue'], 'NT': ['red', 'green', 'blue'], 'NSW': ['red',
'green', 'blue']}
>>> print csp.constraints
{'WA': {'SA': [('red', 'green'), ('red', 'blue'), ('green', 'red'),
('green', 'blue'), ('blue', 'red'), ('blue', 'green')], 'NT': [('red',
'green'), ('red', 'blue'), ('green', 'red'), ('green', 'blue'), ('blue',
'red'), ('blue', 'green')]}, 'Q': {'SA': [('red', 'green'), ('red',
'blue'), ('green', 'red'), ('green', 'blue'), ...lots more...
```

As seen in the printout, all seven variables have identical domains. The excerpt from `csp.constraints` shows, among other things, that WA is connected to SA and NT. The lists `csp.constraints['WA']['SA']` and `csp.constraints['WA']['NT']` show that the valid value pairs between WA–SA and WA–NT are *all possible pairs having different colors*.

Among the skeleton code, there is some sample code that sets up the map coloring CSP shown above. The map coloring CSP can be a helpful way to test your code as you implement backtracking and AC-3.

Sudoku Boards as CSPs

The skeleton code includes code to read Sudoku boards from text files and to represent these boards as CSPs. If you choose to use the skeleton code, your task is thus to implement backtracking search and AC-3 at the indicated locations in the skeleton code, and your Sudoku solver should then be ready to go.

To demonstrate that your CSP solver works, your program should solve the following four Sudoku boards, and the results should be included in your report:

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

Figure 1: Easy board (`easy.txt`)

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

Figure 2: Medium board (`medium.txt`)

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

Figure 3: Hard board (`hard.txt`)

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

Figure 4: Very hard board (`veryhard.txt`)

Deliverables

1. Well-commented source code for a CSP solver based on backtracking search and AC-3, that is able to solve Sudoku boards.
2. Your program's solution for each of the four boards shown above.
3. The number of times your BACKTRACK function was *called*, and the number of times your BACKTRACK function *returned failure*, for each of the four boards shown above. Comment briefly on these numbers for each of the four boards.