Solving Constraint Satisfaction Problems

Adapted from: UBC, Giuseppe Carenini, David Poole, & CPSC322 Team (Kevin Leyton-Brown. Mark Crowley, Erik P. Zawadzki, David R.M. Thompson, Alan Mackworth)

In this project, you will implement basic CSP solving algorithms and heuristics and try them on three different kinds of CSPs: Sudoku, a spatial layout problem, and a scheduling problem.

You may use the utility code provided here, which is in Java,
OR
you may complete the project using a language of your choice, in which case you might need to write some of your own utility functions.

Pro tip: For ALL of the following parts, it will make your life a LOT easier if you work out a concrete example on paper ***first****, and that will give you something to check the step-by-step results of your algorithms against.*

Part 1. Sudoku

Sudoku is a game played on a square grid of 9 large blocks, each containing its own 3x3 grid of cells. Here is an example:

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

The rules are simple:

- each cell contains a number from 1-9
- the number in a given cell (i, j) cannot match the number in any other cell in row i or column j
- each block can only contain one entry of each number from 1-9

Some cells are initially filled in and the goal is to find the unique assignment of numbers to the remaining cells that satisfies the above rules. This game can be seen a straightforward system of constraints.

Note: You can assume that the given Sudoku board is valid, that is, it has a solution.

(a) Before you start writing any code, design the CSP you will use to solve a Sudoku problem. Describe the variables, their domains and the constraints needed. We encourage you to use only binary constraints, it will make the implementation easier.

(b) Implement a Sudoku solver that uses basic backtracking search without any kind of constraint propagation. You can base your implementation off the **backtracking search algorithm pseudocode in Russell&Norvig** (Figure 6.5 in 4th edition). Your algorithm should call subfunctions for selecting the next variable and next value, and these subfunctions should just iterate through the possibilities in a fixed order.

- How well does your algorithm perform on various Sudoku problems? (Several example problems are given as .sud files.)

(c) Now, modify your variable-selection subfunction to use the **minimum-remaining-values (MRV)** heuristic and the **degree heuristic**.

- How much (if any) does performance improve on various Sudoku problems?

(d) Now add **forward checking** to your algorithm. How much (if any) does performance improve on various Sudoku problems?

- How much (if any) does performance improve on various Sudoku problems?

(e) Now add two preprocessing steps: verifying **node consistency** and verifying **arc consistency**. For arc consistency, you can see the **AC-3 algorithm in Russell&Norvig** (Figure 6.3 in 4th edition).

- With these pre-processing steps, how much (if any) does performance improve on various Sudoku problems?

Part 2. Spatial Layout CSP

CSP techniques are useful in solving complex configuration and allocation problems. Consider the following spatial layout problem. (Note, there is no utility code with this part.)

You are given the task of allocating four developments in a new land site. You have to place a housing complex, a big hotel, a recreational area and a garbage dump. The area for development can be represented as 3x3 grid (three rows 0,1,2 and three columns 0,1,2) and you need to place each development in one cell of the grid. Unfortunately there are some practical constraints on the problem that you need to take into account. In the following, A is "close to" B if A is in a cell that shares an edge with B.

- There is a cemetery in cell 0,0.
- There is a lake in cell 1,2.
- The housing complex and the big hotel should not be close to the cemetery.
- The recreational area should be close to the lake.
- The housing complex and the big hotel should be close to the recreational area.
- The housing complex and the big hotel should not be close to the garbage dump.

(a) Represent this problem as a CSP. Be as precise as you can in specifying the constraints. Also do not forget some basic constraints that are inherent in allocating objects in space but are not listed above.

(b) Apply the various versions of the CSP solver you have developed for the previous part to solve this CSP. Describe how performance changes with each version of your solver.

Part 3. Scheduling CSP

Consider the following problem of scheduling university exams, in which:
There are a fixed number of dates (five) and timeslots per day (four) when exams can be run. There are a limited number of rooms that the exams can run in (though this is variable). An exam slot specifies a date, time and room. No two exams can be run in the same exam slot. There are a variety of students taking different combinations of courses. No student can be in two rooms at once. This can be represented as a CSP as follows:

Variables - There is one variable per exam.
Domains - The domain of each variable is the set of exam slots.
Constraints - There is a constraint between every pair of variables, that no two take the same value. There are also a number of constraints between pairs of variables that the two exams can't have simultaneous exam slots if any student has to attend both.

You will be provided with code that can generate instances of this scheduling problem.

(a) Using a variety of test cases (you should generate as many as you need), repeat your experiments using your various solver versions, and examine how performance changes.

(b) Finally, you will implement a very different type of CSP solver, called the Min-Conflicts algorithm. This algorithm is a stochastic local search algorithm, meaning that it generates a complete schedule, checks to see how "good" it is, and then changes one assignment to try to make it a little bit better. **So: It is akin to a hill-climbing optimization algorithm.**

You can implement **min-conflicts using the Russell&Norvig pseudocode** (Fig 6.9 in 4th edition). Note that instead of counting up the conflicts yourself, you can use the Evaluator class that already had this implemented for you.

How does min-conflicts do on this problem? (Remember that min-conflicts is not guaranteed to terminate, nor to find a completely valid solution if one exists.)

Guide to utility code for this part:
- SchedulingProblem.java: this class stores a scheduling problem.
- Student.java, Room.java, Course.java: components of the scheduling problem.
- Generator.java: generates a random instance of the scheduling problem.
- ScheduleChoice.java: This class stores the scheduling information for one exam (i.e., an exam slot). A possible world of a particular instance of the scheduling problem is an array of these with a ScheduleChoice for every exam.
- Evaluator.java: A scoring function that scores a schedule (i.e., a possible world of a particular instance of the scheduling problem). You can use it to guide your local search algorithms and also to assess the quality of your solutions. It penalizes schedules that schedule two exams at the same room at the same time and applies a smaller penalty for each student scheduled to be in two exams at the same time.
- Driver.java: A test driver that you can use to evaluate your solutions.
- Scheduler.java: scheduler interface that your algorithms have to implement.
- Scheduler1.java, Scheduler2.java: stubs for your code.