Spring Semester, 2011 **B.Y. Choueiry**

CSCE 421/821: Foundations of Constraint Processing

Homework 1

Generic Data Structures for Storing CSPs

Assigned: Monday, January 24, 2011

Due: Wednesday, February 2, 2011

Total value: 80 points

Contents

---

1 Help

If you have any questions, comments, or concerns, please let us know:

- Email us to cse421 AT cse.unl.edu.
- Office hours of Shant (Student Resource Center): Thursday 9:00 a.m. to – 11:00 a.m., or by appointment.
- Office hours of Robert (Student Resource Center): Thursday 5:00 p.m. to 6:00 p.m. & Friday 9:00 a.m. to 10:00 a.m., or by appointment.

2 Goal of the homework

The goal of this homework is to write code for generating the data structures and accessor functions that will allow you to implement (most) CSP instances, and to test your implementation by reading examples from files. This implementation may need to be refined as we progress in the course, but at least the basic building-blocks for initializing, storing, and manipulating CSP instances should be available. It is very important that you take this task seriously and do the implementation as clearly and neatly as possible: Future homework will build upon this code. If you notice any errors in the design or the description, please quickly mention them to the instructor as you encounter them. This homework has two parts:

1. Creating the basic data structures. (50 points)
2. Generating the encoding of a CSP instance by reading and parsing the data from an XML file written in XCSP 2.1 format. (30 points)

Alert: If you are using the existing Java parser
`http://cse.unl.edu/~choueiry/CSPTestInstances/Tools2008.zip`,
most of the work is already done for you. Your task will then simply be to learn to use the code and write whatever pieces are missing. If you are using the existing C++ parser
`http://www.cril.univ-artois.fr/~lecoutre/software.html`,
you may have to do more work.

This document is perhaps more complicated and longer than the work that you actually have to do.

3 General indications

Please read carefully the general indications below before you start working on the homework. They apply for the entire semester.

- Make sure that *your code and your files are protected*. Your name, date, and course number must be listed on the top of each file that you submit.
- All programs must be compiled, run and tested on `cse.unl.edu`. Programs that do not run correctly in this environment will not be accepted.
- A README file must be submitted. Otherwise, the entire homework is declared invalid. The README file should describe the content and purpose of the submitted files, and have ALL the necessary steps to compile and run the program.
- This homework must be done individually. *If you receive help from anyone, you must clearly acknowledge it* in the README file. If you collaborate with a colleague or retrieve the code from some other source, clearly state this information in your README file and clearly state all your sources. Always acknowledge sources of information (URL, book, class notes, etc.)

in the README file. You will not be penalized if you state your sources, you will if you do not.

- You can use the programming language of your choice. If this language is not C, C++, or JAVA, you must inform the instructor before you get started. We are not able to provide much programming/debugging help. You must check *your results* with colleagues by listing your results on the wiki and comparing them with those of your colleagues.
- Inform instructor quickly about typos or other errors that may appear in the specifications or the files made available online.

4 Basic data structures

Below we specify (as best we can) the data structures needed for storing the encoding of a CSP. When generating an encoding, we will generate instances of those general data structures. Note that:

- We will restrict ourselves to binary CSPs. Students looking for a real challenge may want to consider generalizing their code, now or later, to non-binary CSPs.
- If you have a better idea for implementing your data structures, then you should experiment with it, and consider the data structures below *as mere recommendations*.
- It is very important that you do not mix the data structures used for storing a CSP instance with the data structures of the solver (in this case, search) used for solving the CSP.

4.1 Main data structures

The design requires primarily the choice of a linked list or an array to hold the variables, constraints and the domains of the variables. This decision may have a drastic impact on the performance of the algorithms. Use your judgement to make your choice of the data structures. Depending on the design of your algorithms, you might revise your decisions at a later stage.

1. *Problem instance*. Create a data-structure for storing a CSP instance. This data structure should have the following fields:
   - `name`: the name of the problem, generates one randomly at initialization if none is provided.
   - `variables`: pointer to the structure holding the variables.
   - `constraints`: pointer to the structure holding the constraints.
2. *Variables*. Create a data structure for storing the CSP variables.

   Each variable must have the following fields:

- `name`: the variable name or identifier.
- `initial-domain`: values in the domain of the variable.
- `constraints`: pointers to the constraints that apply to the variable (i.e., constraints in the scope of which that variable appears).
- `neighbors`: pointers to the other variables that share a constraint with this variable.

3. *Constraints*. Data structure holding all the constraints in the CSP. Again, we will focus on binary constraints. If you like a challenge, you may want to generalize your code, now or later to non-binary constraints. Each constraint may require different fields depending on its specification, as in extension or intension:

   (a) A constraint defined in extension as a set of allowable tuples (a.k.a. supports).

   (b) A constraint defined in extension as a set of no-goods (a.k.a. conflicts).

   (c) A constraint defined in intension, as a function to be implemented by a method. We will restrict ourselves to those predicate functions that we will encounter in the test instances. You will implement those predicates as you encounter them.

   The constraint should have the following fields:

   - `name`: the name of the constraint.
   - `variables`: pointers to the variables in the scope of the constraint.
   - `definition`: stores the definition of a constraint. If the constraint is defined in intension, this field should point to the function that defines the constraint, that is, implements the predicate.

     If the constraint is defined in extension, then it has a list of tuples that are either supports or conflicts. For instance:

     $$C_{V_1,V_2} = \{(1,2),(3,5),(2,3)\} \tag{1}$$

     is a constraint with three tuples: (1, 2), (3, 5), (2, 3). These tuples may be stored in a list, in a two dimensional array, a bit matrix, a decision diagram, etc. The bitmap should be of size d^2 (binary constraint), where d is the maximum domain-size (more generally, the size is d^k, where k is the maximum constraint arity). The values of the bitmap should be 0's in all positions except for the positions: (1, 2), (3, 5), (2, 3), which should be 1's.

4.2 Main functions/methods

Provide methods to print on the screen the variables and the constraints. These functions can be used for debugging puposes.

For each variable the following information should be printed:

1. The name of the variable.

2. The name(s) of the constraint(s) that apply to the variable.

3. The values in the domain of the variable.

For each constraint print the following information:

1. The name of the constraint.

2. The names of the variables that are in the scope of the constraint.

3. The way the constraint is defined, i.e. extension-supports, extension-conflicts, or intension.

4. The list of tuples in the definition of the constraint or the name of the function in case of intension.

5 Loading and initializing a CSP instance

The tasks here are to:

1. Use or write a parser that loads any file of a CSP instance in the XCSP 2.1 format (see below).

2. Load the 16 problem instances in XCSP 2.0 provided by the instructor:
`http://cse.unl.edu/~choueiry/CSPTestInstances/`.

3. Print the variables and constraints as specified above.

4. Test your code by loading various benchmark problems, which are in XCSP 2.1 format:
`http://cse.unl.edu/~choueiry/CSPTestInstances/`.

You are requested to read description of a CSP instances specified in the XML XCSP 2.1 format and generate an encoding of it according to the data structures you have defined above. You can interface your code with one of the two parsers available online or write your own parser. Familiarize yourself with the content of the following web pointers, which are very useful:

- Most of the resources are available from:
`http://www.cril.univ-artois.fr/~lecoutre/`.

- XCSP 2.1 format:
`http://arxiv.org/pdf/0902.2362v1`.

- Online parsers:
`http://www.cril.univ-artois.fr/~lecoutre/software.html`, click XCSP Tools.

- Online tools:
`http://www.cril.univ-artois.fr/~lecoutre/software.html`, click XCSP Tools.

- Benchmark problems:

  `http://cse.unl.edu/~choueiry/CSPTestInstances/`.

  `http://www.cril.univ-artois.fr/~lecoutre/benchmarks.html`.

  You may notice that parsers are provided for Java and C/C++. If you chose an alternative programing language, you may either consider using the parser to parse the XML file to an intermediate representation of your convenience and then load it to your program, or write a parser using XML parsing tools in the language you are using.