Spring Semester, 2008 **B.Y. Choueiry**
CSCE 421/821: Foundations of Constraint Processing

Homework 1

Large Generic Data Structures for Storing CSPs

Assigned: Monday, January 28, 2008

Due: Wednesday, February 6, 2008

Total value: 80 points

Contents

1 Goal of the homework **1**

2 General indications **2**

3 Basic data structures **2**
3.1 Main data structures . . . 2
3.2 Main functions/methods . . . 4

4 Loading and initializing a CSP instance **4**

---

1 Goal of the homework

The goal of this exercise is to write code for generating the data structures and accessor functions that will allow you to implement (most) CSPs, 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 CSPs 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 exercise has two parts:

1. Creating the basic data structures. (50 points)
2. Generating the encoding of a CSP instance by (30 points)
   reading and parsing the data from an XML file written in XCSP 2.0 format.

2 General indications

Please read carefully before you start working on the homework:

- Please make sure that *your code and your files are protected*. Your name, date, and course number must appear in each file of code 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.
- This homework must be done individually. If you collaborate with a colleague, please inform the instructor immediately.
- *If you receive help from anyone, you must clearly acknowledge it.*
- Always acknowledge sources of information (URL, book, class notes, etc.).
- You can use the programming language of your choice. We are not able to provide any programming/debugging help. You are check *your results* with colleagues.
- Please inform instructor quickly about typos or other errors that may appear in the specifications or the files made available online.

3 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 these 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.

3.1 Main data structures

1. *Problem instance*. Create a record-like data-structure for storing a CSP instance. (For lispers, use `defstruct`.) This data structure should have the following attributes:

- `name`: stores the name of the problem, generates one randomly at initialization if none is provided. (Hint for Lisp users: use `(gentemp "PROBLEM-")`.)
- `variables`: to be filled with a list of pointers to the *data structures* corresponding to the variables. Alert: this is not a list of the names of the variables.
- `constraints`: to be filled with a list of pointers to the data structures corresponding to the constraints. This list can be implemented as a hash-table where a key is the list of variables in the scope of a given constraint and the value is a pointer to the definition of the constraint. No matter how you choose to implement this, you may consider to list the names of the variables in the scope of the constraint sorted in alphabetical order in order to facilitate access.
- (optional) `values`: a list of pointers to all the values in the problem. Such a list is usually useful in scheduling and resource allocation applications to store all resources such as machines.

2. *Variable*. Create a record-like data-structure for storing a CSP variable. This data structure should have the following attributes:
   - `name`: the variable name or identifier (e.g., a string, an integer).
   - `initial-domain`: to be filled with the list of values in the domain of the variable.
   - `constraints`: to be filled with a list of pointers to the data structures corresponding to the constraints defined over the variable.
   - `neighbors`: to be filled with a list of pointers to the data structures of the CSP variables that share a constraint with the variable.

3. *Constraint*. Use a *class* to declare and store a CSP constraint. Declare three sub-classes of the constraint class to represent:
   (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.

   The constraint data structure should have the following attributes:
   - `variables`: a list of pointers to the variables in the scope of the constraint.
   - `definition`: to store the definition of a constraint, defined in intension or in extension, both as a set of supports or a set of conflicts.

     For a constraint whose definition is given in extension, the list of tuples can be stored as a (linked) list of values that are allowed by the constraint in the lexicographical order of the variables. For instance:

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

Alternatively, you could store the allowed tuples in an array of two dimensional array of size $d^2 \times 2$, where d is the max domain-size (more generally, the size is $d^k \times 2$, where k is the max arity). Each row in this array is a tuple allowed by the constraint. (This is similar to a table in a relational database.)

3.2 Main functions/methods

Further, create the following methods that apply to this constraint data-structure:

- `arity`: is a function that returns the number of variables in the scope of the constraint.
- `unary-p`: is a predicate that determines whether or not the constraint applies exactly to one variable.
- `binary-p`: is a predicate that determines whether or not the constraint applies to exactly two variables.
- `check`: is declared as a method on a constraint. It etermines whether a tuple, given in input, is consistent. You may choose to specify the tuple given in input in one of two ways:
  - As a list, or a vector, of variable-value pairs. For example: $((V_1, v_1), (V_2, v_2), \ldots, (V_k, v_k))$. Or
  - As a tuple of values. For example: $(v_1, v_2, \ldots, v_k)$, assuming that it is clear which value is for which variable (e.g., because the scope of the constraint is sorted alphabetically).

  We assume that the tuples specify values for *all* the variables in the scope of the constraints. Note that `check` should be implemented for all three constraint sub-classes: intension, extension as conflicts, and extension as supports.

4 Loading and initializing a CSP instance

You are requested to read description of a CSP instances specified in the XML XCSP 2.0 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. Useful web pointers:

- Most of the resources are available from http://www.cril.univ-artois.fr/~lecoutre/
- XCSP 2.0 format:
  http://www.cril.univ-artois.fr/~lecoutre/research/benchmarks/benchmarks.html#format
- Online parsers:
  http://www.cril.univ-artois.fr/~lecoutre/research/tools/tools.html#parsing

- Online tools:
  http://www.cril.univ-artois.fr/~lecoutre/research/tools/tools.html

- Benchmark problems:
  http://www.cril.univ-artois.fr/~lecoutre/research/benchmarks/benchmarks.html

The tasks here are to:

1. Use or write a parser that loads any file of a CSP in the above format. (10 points)

2. Load the 12 problem instances in XCSP 2.1 provided by the instructor: (20 points)
   http://cse.unl.edu/~choueiry/CSPTestInstances/

3. Test your code by loading various benchmark problems. (0 points)