Spring Semester, 2022 **B.Y. Choueiry**

CSCE 421/821: Foundations of Constraint Processing

Homework 1

Generic Data Structures for Storing CSPs

Assigned: Monday, January 24, 2022

Check point: Monday, January 31, 2022

Due: Monday, February 7, 2022

Value: 10 points for check point + 80 points for final homework

Contents

---

1 Help

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

- Use Piazza to communicate problems / questions with the homework.
- Discuss among yourselves and help each others, but do your own homework.
- Daniel and Tony are available for help during office hours 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 choose to use an existing parser (not required), 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. Additionally, the Abscon parser provides a built in function for evaluating the intension constraints needed in Homework 2. If you are interested in the challenge, you are free to implement your own parser from scratch. However, due to lack of TA resources, we are unable to provide any mentoring, help, or grading for this task.

Abscon (Java): `http://cse.unl.edu/~choueiry/CSPTestInstances/Tools2008.zip`,

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. You must include a Makefile with your program so that your code can be compiled by issuing 'make' while on cse. You also must include a script called 'runProgram.sh' that contains the command to run your program (e.g., if your program is called a.out, runProgram.sh would have 'a.out $@' inside of it, with $@ passing on the command line arguments to a.out).
- 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 (including if you are starting from an existing parser). 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.
- This homework must be written in Java. 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.
- To facilitate debugging and the expectations of the homework assignment, web grader is set up to quickly evaluate the correctness of your program: `https://cse.unl.edu/~cse421/grade/`. After you have files submitted through webhandin, you will be able to run the web grader.

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 to distinguish the data structures used for storing a CSP instance from those used to do search in the following sense. Search can access the data structures (variables, domains, constraint) of the CSP but should not modify them. Imagine that two or more solvers are trying to solve the same CSP. You want each solver to access the same problem definition but it should not modify it.

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 judgment 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 read from the xml, uses the xml filename 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 should have the following fields:

   - `name`: the variable name or identifier.
   - `initial-domain`: values in the domain of the variable.
   - `current-domain`: values in the domain that are still alive (this should be a *deepcopy* of the `initial-domain`).
   - `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.

   Constraints can be specified in intension or in extension. Further, the latter can be given as supports (positive tables) or as no-goods. It is best to define a general constraint class, and specialize it to intension and extension, then specialize the extension class into supports and conflicts.

   Each constraint class may require different fields:

   (a) A constraint defined in extension must store all allowable tuples (i.e.. supports) or all forbidden tuples (i.e., no-goods, conflicts).

   (b) A constraint defined in intension, is 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.

   Note: The Java parser by Lecoutre can parse the predicates defined in the XML benchmarks. If you are writing your own parser, you can create a library (piece of code) of the functions implementing the predicates and pass the `id` of the function in the library to the constraint definition, possibly manually.

   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 (e.g., 2 dimensional bitmap), 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 purposes.

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.

An example output generated by your program for the 4queens-supports.xml file:

```
Instance name: 4q-supports
Variables:
Name: V0, initial-domain: {1,2,3,4}, constraints: {C0,C1,C2}, neighbors: {V1,V2,V3}
Name: V1, initial-domain: {1,2,3,4}, constraints: {C0,C3,C4}, neighbors: {V0,V2,V3}
Name: V2, initial-domain: {1,2,3,4}, constraints: {C1,C3,C5}, neighbors: {V0,V1,V3}
Name: V3, initial-domain: {1,2,3,4}, constraints: {C2,C4,C5}, neighbors: {V0,V1,V2}
Constraints:
Name: C0, variables: {V0,V1}, definition: supports {(1,3),(1,4),(2,4),(3,1),(4,1),(4,2)}
Name: C1, variables: {V0,V2}, definition: supports {(1,2),(1,4),(2,1),(2,3),(3,2),(3,4),(4,1),(4,3)}
Name: C2, variables: {V0,V3}, definition: supports {(1,2),(1,3),(2,1),(2,3),(2,4),(3,1),(3,2),(3,4),(4,2),(4,3)}
Name: C3, variables: {V1,V2}, definition: supports {(1,3),(1,4),(2,4),(3,1),(4,1),(4,2)}
Name: C4, variables: {V1,V3}, definition: supports {(1,2),(1,4),(2,1),(2,3),(3,2),(3,4),(4,1),(4,3)}
Name: C5, variables: {V2,V3}, definition: supports {(1,3),(1,4),(2,4),(3,1),(4,1),(4,2)}
```

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) and outputs information about the CSP instance. You are required to use the following flag to specify the filename: -f <filename>.
2. Provide a README, Makefile, and runProgram.sh file for your program.
3. If you use one of the parsers by Lecoutre, you may still need to modify the data structures of the parser to fulfill the requirements of the homework.
4. Load, one at a time, the 16 problem instances in XCSP 2.1 provided by the instructor: `http://cse.unl.edu/~choueiry/CSPTestInstances/`.
5. Print the variables and constraints as specified above.

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. (Note: XCSP 3 is now released, but we are still using XCSP 2.1.) 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 parser: `http://cse.unl.edu/~choueiry/CSPTestInstances/Tools2008.zip`
- Benchmark problems: `http://cse.unl.edu/~choueiry/CSPTestInstances/`
  `http://www.cril.univ-artois.fr/~lecoutre/benchmarks.html`
  XCSP 2.1: `http://www.cril.univ-artois.fr/~lecoutre/\#/benchmarks`

6 Alert to common errors

Below is a list of common difficulties and errors:

1. *Deep copies of structures.* Create your own data structures for:
   (a) The domain of the variables
   (b) The constraints implemented in extension

Abscon uses the same, unique data structure for all variables that have the 'same' domain (i.e., domains that are equal sets). As a result, when you filter the domain of a single variable, the domains of all other variables with the same domain are modified, which is a disastrous side effect. You need to make deep copies of these data structures.

2. *Constraints defined in intension.* For constraints defined in intension, you are welcome to write your own code to implement the function, which would be a great exercise. However, for the sake of simplicity, it is acceptable that you rely on the implementation of Abscon. The snippet of code below may be useful to this end if you are using Java:

```
boolean satisfied = true;
for (Map.Entry<MyConstraint, Boolean> entry : constraint.entrySet()) {
    MyIntensionConstraint intensionConstraint = (MyIntensionConstraint) entry.getKey();

    satisfied = satisfied && (intensionConstraint.refCon.computeCostOf(tuple) == 0);

}
return satisfied;
```

Figure 1: Call to constraints defined in intension

3. *Unary constraints.* Some problem instances, such as the Zebra puzzle, have unary constraints. The Zebra puzzles has two such constraints. Do not ignore them. Usually, we do a preprocessing to enforce these constraints before we start any processing (e.g., search or enforcing consistency).

4. *Constraints with subsumed scopes.* Beware when two or more constraints have the same scope. Usually, we always include a pre-processing step to *normalize* such constraints. Constraint normalization consists of replacing all constraints whose scope is subset ($\subseteq$) of other constraints by a unique constraint that is the intersection of them all. As a result, we end up with a unique constraint defined over any given set of variables.

5. *Separately process independent components.* It is customary to run a connectivity test to check whether the constraint graph is connected or not. In case it is, each independent component is processed as an independent problem. This comment applies mainly to the search procedure in the context of the homework.

Other than the above, the instance `bqwh-15-106-0_ext has an empty constraint definition, which should be ignored. It was an 'artifact' of the instance generator.`