Constraint Satisfaction Problems

B. Nebel, C. Becker-Asano, S. Wölfl
Wintersemester 2014/15

University of Freiburg
Department of Computer Science

Project: Part 1

Due: 2.12.2014
(2 + 10 + 3 points)

Task 1. Setting up your git repository (2 points). If you want you may form groups of two. The code submission will be handled by the git source code management system. For this purpose we will host a git repository for each group. To access the repository *everyone* is required to create a private/public key pair.
On Linux (e.g. in the computer pool) create a private/public key pair by running:

```
ssh-keygen -f csp1415yourname
```

(replace yourname with your name). This will give you the files `csp1415yourname` (the private key) and `csp1415yourname.pub` (the public key).

- Copy the *private key* into `~/.ssh/`
- Send the *public key* to Stefan Wölfl (woelfl@informatik.uni-freiburg.de) . We will then set up a repository for your group.

Add the following entries to your ssh config file (`~/.ssh/config`):

```
Host git-csp1415
HostName gkipool4.informatik.uni-freiburg.de
User csp1415
IdentityFile ~/.ssh/csp1415yourname
```

We will notify you by email once we have created the repository for your group. When the repository has been set up by us you can start to use the repository. First *clone* the git repository by running (replace GROUPNAME with your group name):

```
git clone git-csp1415:GROUPNAME
```

This will give you the directory GROUPNAME which already contains some simple example problems. A list of further basic git commands is provided at the end of this file. If you use branches in git: we will expect your submission in the master branch of your repository.

Task 2: Reading problem descriptions (10 points). In this and the next project assignments you should implement standard algorithms used to solve constraint satisfaction problems. The programming language for this and the follow-up assignments is **Python 3 (version 3.4.2)**. The task is to read CSP instances and to set up the appropriate data types (the topics of the *next* assignments will be backtracking search and local consistency methods).

Your program is expected to read constraint networks from an XML file and provide some output as specified in the next sections. You should already implement the basic structure

of the data types later used by your solver (e.g. Constraint, Network). We do not impose the way you represent constraint networks. For example, you can represent constraints by allowed tuples, forbidden tuples, etc.
We expect that all your Python code is contained in some top-level directory of your repository, except a single Python-Script *solver.py.* To run the script on an input file *some_path/input.xml* we will use the call *python3 solver.py –print some_path/input.xml* in the root directory of your repository.

The XML format for input. The CSP instances will be encoded in the XCSP 2.1 format. A description of the format is available at the following address:

http://arxiv.org/pdf/0902.2362v1.

We nevertheless apply some restrictions:

- We are only concerned with standard CSP instances (no Weighted CSPs and no Quantified CSP instances);
- We are (initally) only concerned with extensional constraints (no intensional constraints and no global constraints). Note that extensional constraints can be given in the form of allowed tuples (*'supports'*) or forbidden tuples (*'conflicts'*)

On the following page an XCSP description of the 4-queens problem is listed, which will help to understand the format. As one can see, the format is explicit and there are no particular difficulties. You find the following sections:

domains: here the domains in the problem are specified. They will be linked with variables in the "variables" section.

variables: here the variables are specified and associated with their respective domain.

relations: here the relations are described (as sets of forbidden or allowed tuples). If the constraints describe allowed tuples the "semantics" attribute is set to "supports". If the constraints describe forbidden tuples the "semantics" attribute is set to "conflicts".

constraints: here the constraints are specified, i.e. one can link variables (through the scope attribute) and a relation (through the reference attribute).

The 4-queens problem is also already available as an example in the repository.

The text format for output. In order to test that your XML parser correctly reads the XML description, your program should print out the variables and constraints of the input network to standard output. The format is as follows:

- First, the variables are given separated by commas. A semicolon ends the line.
- Then, all the constraints are described here. They are all given in the form of forbidden tuples. So, if the input format specifies some constraints under the form of allowed tuples you will need to translate it. This does not imply how to store this information, you can do either way. If no constraints hold between some sets of variables, there is nothing to write on the output format. For example, there is no constraint of arity 3 in the 4-queens problem, so there is no constraints of size 3 in the output given below. First the scope is given (with variables separated by commas), then a pipe symbol and afterwards all the forbidden tuples are given separated by commas. A semicolon ends the line.

Example 1. *XCSP specification of the 4-queens problem*

```
<instance>
  <presentation name="4queens" maxConstraintArity="2"
                format="XCSP 2.1"/>
  <domains nbDomains="1">
    <domain name="D0" nbValues="4">1..4</domain>
  </domains>
  <variables nbVariables="4">
    <variable name="c1" domain="D0"/>
    <variable name="c2" domain="D0"/>
    <variable name="c3" domain="D0"/>
    <variable name="c4" domain="D0"/>
  </variables>
  <relations nbRelations="3">
    <relation name="e1" arity="2" nbTuples="6"
              semantics="supports">
      1 3 | 1 4 | 2 4 | 3 1 | 4 1 | 4 2
    </relation>
    <relation name="e2" arity="2" nbTuples="8"
              semantics="supports">
      1 2 | 1 4 | 2 1 | 2 3 | 3 2 | 3 4 |
      4 3 | 4 1
    </relation>
    <relation name="e3" arity="2" nbTuples="10"
              semantics="supports">
      1 2 | 1 3 | 2 1 | 2 3 | 2 4 | 3 1 |
      3 2 | 3 4 | 4 2 | 4 3
    </relation>
  </relations>
  <constraints nbConstraints="6">
    <constraint name="r12" arity="2" scope="c1 c2"
                reference="e1"/>
    <constraint name="r23" arity="2" scope="c2 c3"
                reference="e1"/>
    <constraint name="r34" arity="2" scope="c3 c4"
                reference="e1"/>
    <constraint name="r13" arity="2" scope="c1 c3"
                reference="e2"/>
    <constraint name="r24" arity="2" scope="c2 c4"
                reference="e2"/>
    <constraint name="r14" arity="2" scope="c1 c4"
                reference="e3"/>
  </constraints>
</instance>
```

Example 2. *Expected output for the 4-queens problem explained in Example* **??**.

```
c1, c2, c3, c4;
c1, c2 | 1 1, 1 2, 2 1, 2 2, 2 3, 3 2, 3 3, 3 4, 4 3, 4 4;
c2, c3 | 1 1, 1 2, 2 1, 2 2, 2 3, 3 2, 3 3, 3 4, 4 3, 4 4;
c3, c4 | 1 1, 1 2, 2 1, 2 2, 2 3, 3 2, 3 3, 3 4, 4 3, 4 4;
c1, c3 | 1 1, 1 3, 2 2, 2 4, 3 1, 3 3, 4 1, 4 4;
c2, c4 | 1 1, 1 3, 2 2, 2 4, 3 1, 3 3, 4 1, 4 4;
c1, c4 | 1 1, 1 4, 2 2, 3 3, 4 1, 4 4;
```

Task 3: Primal constraint graphs (3 points). The task is here to generate the primal constraint graph of an input constraint network. The graph should be printed to standard output in DOT-format. See the user manual

http://www.graphviz.org/pdf/dotguide.pdf

which contains many examples of graphs in this format. The output graphs should have a labelling of the vertices. Moreover the edges should be labelled: list all the names of the constraints (if available) on which the respective arc depends.
We will call your solver by *python3 solver.py –primal-graph some_path/input.xml* to test this functionality.

A minimal guide to git. The only necessary actions to use the git repository are listed here. Adding files to git:

```
git add file
```

This tells git to keep track of the given file. Simply put, only tracked files can be uploaded to us (see `commit` and `push`). Tracked files must be all your *source files*. Untracked files will not be seen by us, this might be appropriate for temporary files or binaries from your build process. Adding also works on files within directories, e.g.:

```
git add path/file
```

will retain the path within the repository as well. Similarly, you can remove ("untrack") files from git by

```
git rm file
```

This will not affected previous uploads (or so-called commits) but causes future uploads and updates to not contain this file. To store a snapshot of the current state of the tracked files run

```
git commit -a -m "Commit message"
```

This *commit* contains the current version of all tracked files. The message serves as a label for this snapshot. Finally, to upload all your commits to us run

```
git push
```

Without `git push` we cannot access your work. Conversely, you can download and apply the changes from your group's repository by

```
git pull
```

This will automatically merge the latest version from the repository with your local snapshots (i.e. commits). More information and help on git can be found on e.g. `http://git-scm.com/`.