Project sheet 2: Backtracking and Consistency (30 points)

Prof. Dr. Bernhard Nebel, Dr. Stefan Wölfl
Dr. Julien Hué and Matthias Westphal

May 23, 2012

This work is due on 15.06.2012

Updates of the git repository

For this Project Sheet, we will provide some more elaborate CSP instances. They will be directly put into your git repository. The command to update your repository is the following:

```
git pull
```

Note that you will get the following error message if you try to push without having pulled necessary changes:

```
! [rejected] master -> master (non-fast forward)
error: failed to push some refs to 'csp2012@csp2012:GROUP'
To prevent you from losing history, non-fast-forward updates were rejected
Merge the remote changes before pushing again. See the 'non-fast forward'
section of 'git push --help' for details.
```

As usual, the source and Makefile should be in the git repository before the due date. We require the source to compile. If you handed in your code, you can ask us by mail to check whether the code is compiling and working.

This project sheet will only cover binary CSPs as the consistency algorithms introduced in the lecture so far only cover binary networks. Extensions to non-binary network will be the subject of future project sheets.

1 Backtrack algorithm (6 points)

Implement the main Backtrack algorithm given in Figure 1.

The variable selection (line 03) must be made inside a distinct class (or function). For this time, the variable may be picked according to the lexicographic ordering, but note that it will be modified in future projects. The same applies to the selection of values (line 06); create a distinct class (or function) for value selection and (for now) implement a lexicographic ordering.

```
Backtrack(C,a)

Input: a constraint network C = <V,D,c>
      a partial solution a of C
      (possible: the empty instantiation a={})
Output: a solution of C or inconsistent

01.  IF a is defined for all variables in V THEN
02.   RETURN a
03.  ELSE select a variable vi for which a is not defined
04.   Di’ := Di
05.   WHILE Di’ is not empty
06.    select and delete a value x from Di’
07.    set vi to x
08.    a’ := a + vi
09.    IF a’ is consistent THEN
10.     a’’ := Backtrack(C,a’)
11.    ENDIF
12.    IF a’’ is not inconsistent THEN
13.     RETURN a’’
14.    ENDIF
15.   ENDWHILE
16.   RETURN inconsistent
17.  ENDIF
```

Figure 1: Backtrack Algorithm

After the search terminates your solver must print whether a problem instance is satisfiable (SAT) or unsatisfiable (UNSAT) and the time the solver used, e.g.:

```
SAT
Time: 213.46s
```

Be careful with line 10 of the algorithm; you need to pass the constraint network to the next recursive invocation. If you copy the network each time you will be short on memory very soon. Since this simple algorithm is not altering the network, you can pass the argument by reference.

On future project sheets we will have algorithms that refine the network during search, which will require you to implement passing the network in a smarter way, e.g., passing by reference and manually keeping track of chances to the network while undoing them upon backtracking. For this project sheet simply passing by reference will do.

2 Consistency

Now that the backtrack algorithm has been implemented, you can see how inefficient it is by testing it with the instances we are providing (note the running times, see Section 2.2). We now consider improvements to this solver in the form of preprocessing networks once before the search.

2.1 Arc Consistency and Path Consistency (3×6 points)

Implement the following functions inside your solver (6 points each):

- AC3
- AC2001
- PC2

These algorithms are given in the Lecture (Chapter 4). AC3 and PC2 are also described in Rina Dechter's book (Constraint Processing) on pages 58 and 65. These different consistency algorithms are here to be used as preprocessing before the Backtrack algorithm is invoked.

If the name of your executable file is *solver*, the instance is named *instance.xml* and the consistency algorithm to use is *algo*, the user may be able to select which Consistency algorithm he wants to use according to the following command line:

```
./solver --consistency=algo instance.xml
```

Therefore, the following command lines would preprocess the instance with the associated algorithm and then apply the backtrack search algorithm:

```
./solver --consistency=none instance.xml
./solver --consistency=ac3 instance.xml
./solver --consistency=ac2001 instance.xml
./solver --consistency=pc2 instance.xml
```

The command without any consistency algorithm specified, such as:

```
./solver instance.xml
```

should pick the consistency algorithm of your choice (see Section 2.2).

2.2 Comparing the consistency algorithms (6 points)

Add into your repository the file *results.txt* (do not forget *git add results.txt*). This file must contain a table with the running times of the instances we provided according to the consistency algorithm implemented (AC3, AC2001, PC2, and none) together with the information if the instance has a solution (satisfiable/unsatisfiable). We suggest to put a time limit of 5 minutes to your tests (i.e., abort the solver if the runtime exceeds 5 minutes). If you cannot solve one of the instances mark it as indeterminate in the table. On Linux this can be achieved with first running the *ulimit* command before starting the solver. The syntax of the *ulimit* command may vary according to the shell you are using (typically, it is `ulimit -t 300`).

The table must have the following format:

```
instance01.xml SAT
none 100.21s
ac3 20.54s
ac2001 10.76s
pc2 15.54s
instance02.xml UNSAT
none 50.32s
ac3 1.64s
ac2001 1.76s
pc2 2.74s
...
instance10.xml SAT
none INDETERMINATE
ac3 100.43s
ac2001 80.95s
pc2 20.13s
```

Below the table, write a short text (400-600 characters) commenting and comparing the results in the table.

3 Feedback (2 bonus points)

Tell us what you think about the lecture, exercises and projects. Simply add a *feedback.txt* in your repository with your comments.