

Programing Directions for Assignment 1

CSS486/2506 - Knowledge Representation and Reasoning

Fall 2008

1 Platforms to be used

The preferred programming languages for the course are the following two:

1. **Prolog:** your program has to run under SWI-Prolog, which is available for Linux and Windows at <http://www.swi-prolog.org/> together with full documentation.

You can run SWI-Prolog in CDF (execute `p1`) and CSLAB (execute `/usr/bin/swipl` on the application machines).

2. **Scheme:** your program has to run under MIT Scheme. You can download a copy from <http://www.gnu.org/software/mit-scheme/> for Linux or Windows.

MIT Scheme is installed in both CS and CDF. Just type `scheme` to run it.

The reason for using Scheme or Prolog is that they are natural programming languages for the kind of problems we will be dealing with. Nevertheless, you can still use **Java** or **C/C++** as long as your program compiles and runs in CDF or CS machines. In the documentation, you must specify whether your program runs in CDF or CS and clearly state the exact **command used to compile** your source files.

2 Input Format

In exercise 6, you are asked to write a satisfiability procedure for propositional clauses. Here, we discuss the format of the input to your program. You **must** use this representation.

First, we will assume the following representation for *literals*:

1. **Positive literals** will be represented with **positive integers**. In this way, instead of literals p and q , we will refer to literals 2 and 5;
2. **Negative literals** are represented with **negative integers**. In this way, instead of literals $\neg p$ and $\neg q$, we will refer to literals -2 and -5 .

Second, the following is the representation for a *clause*:

- In Scheme, we will represent a clause as a Scheme list of literals. So a clause will have this format: $(l_1 l_2 \dots l_n)$ where l_i is the representation of a literal (that is, an integer different from zero). For instance, $(2 -3 4)$ stands for the clause $(2 \vee \neg 3 \vee 4)$;
- In Prolog, we will represent a clause as a Prolog list of literals. So a clause will have this format: $[l_1, l_2, \dots, l_n]$ where l_i is the representation of a literal (that is, an integer different from zero). For instance, $[2, -3, 4]$ stands for the clause $(2 \vee \neg 3 \vee 4)$.

(1 -2)	[1,-2] .
(2 -8)	[2,-8] .
(-5 -4 3)	[-5,-4,3] .
(5)	[5] .
(4 -2 -6)	[4,-2,-6] .
(-8 -7 6 -10)	[-8,-7,6,-10] .
(6 -10)	[6,-10] .
(-10 7)	[-10,7] .
(8 -5)	[8,-5] .
(-9 9)	[-9,9] .
(end)	[end] .
(a) Scheme representation	(b) Prolog representation

Figure 1: Two files containing a set of propositional clauses

Finally, a set of propositional clauses in a file will be represented as a sequence of clauses representations as described. The sequence will finish with a list containing a single element “end” meaning there are no more clauses. In particular, in the Prolog representation each clause will be followed by a dot ‘.’ symbol stating the end of the clause. For example, figure 1(a) shows a file containing a set of propositional clauses for Scheme and figure 1(b) shows a file representing the same set of for Prolog. Your program will take files of this sort as the input. If you do not use Prolog or Scheme you should choose one of the two representations and state which one you use in your documentation.

3 What your program should provide

If you use Prolog or Scheme we provide sample files (`lsSat.pl` and `lsSat.scm` respectively) which already have the code necessary for reading a set of clauses from a file. These two files define a top-level procedure called `lsSat` whose only argument is the name of a file containing a set of clauses.

Procedure `lsSat` reads the clauses in the file and calls `solve_ls_sat` while passing the corresponding set of clauses as a *list of clauses*. It is **this** procedure, `solve_ls_sat`, that you have to implement by modifying the sample file. `solve_ls_sat` takes a list of clauses in the corresponding form and prints ‘YES’ if such clauses are satisfiable, or ‘NO’ otherwise.

Notice that you do not need to program by yourself the top-level predicate `lsSat` as it is already implemented. Given a file containing a set of clauses, you should be able to test your Scheme program as follows: (`lsSat "<filename>"`). Similarly, in Prolog the command will look as follows: `lsSat(<filename>)`.

Finally, if you are using Java or C, your program itself should be named `lsSat` and it should take as its only argument the name of the file where it should read the set of clauses. A call to your program should have the following form: `lsSat <filename>`. No templates for reading input files will be provided for those languages.

Recall that you should explicitly say in your documentation the exact command used for compiling your program in either CDF or CS.

4 About your experiments

You are asked to confirm or refute the easy-hard-easy pattern. For this, you will have to generate a number N of random instances (sets of clauses), and plot their running times. The simplest way to plot this is by using a histogram graph in which the X axis is the clauses-to-variables ratio and Y is the average running time of the algorithm for the instances on each bin of the histogram. All the instances used in your experiments will have the **same number of variables**.

Since you have limited time to run your experiments you will have to make a smart decision when choosing the total number of instances, and the number of variables for the whole set of experiments. We do not require the number of variables to be high. In fact, any number over 20 will be fine. To define the number of instances you will use, run a few experiments with formulae with a clause-to-variable ratio around 4.2, and measure the running time of the algorithm. Use these values to extrapolate the total running time of all your experiments.

5 How to submit your program

You should submit a printed copy of your program together with its documentation with your assignment in class. Moreover, you should submit your code to `jabaier /AT/ cs /DOT/ toronto .edu` as an attachment file. Note that code received *after the due date* will count as a late assignment.

No matter what platform you choose, remember that your program has to run error-free in CDF or CS. So, if you worked elsewhere, we recommend you try your program in either of these machines before handing in the code.