DIMACS CNF Format

Resolution theorem proving seeks to answer the question "Is the knowledge base plus the negated hypothesis satisfiable?" There is an intense research effort underway for high-performance satisfiability solvers. Each year, there is a satisfiability software competition. The most common problem input format is the DIMACS CNF Format. Recall the liars problem from daily practice 7. We have the KB

1. $\neg A \vee \neg B$
2. $A \vee B$
3. $\neg B \vee \neg C$
4. $B \vee C$
5. $\neg A \vee \neg C$
6. $\neg B \vee \neg C$
7. $A \vee B \vee C$
8. C (when we try to prove $\neg C$, we add C to the KB)

In DIMACS CNF Format, these clauses might be encoded as follows:

```
c Truth-teller and liar problem
c 1:  Amy is a truth-teller.
c 2:  Bob is a truth-teller.
c 3:  Cal is a truth-teller.
c
p cnf 3 8
-1 -2 0
2 1 0
-2 -3 0
3 2 0
-3 -1 0
-3 -2 0
1 2 3
0 3 0
```

According to the 2004 SAT competition documentation[1],

[1]See URL http://satlive.org/SATCompetition/2004/format-benchmarks2004.html

The file can start with comments, that is, lines beginning with the character `c`.

Right after the comments, there is the line `p cnf nbvar nbclauses` indicating that the instance is in CNF format; `nbvar` is an upper bound on the largest index of a variable appearing in the file; `nbclauses` is the exact number of clauses contained in the file.

Then the clauses follow. Each clause is a sequence of distinct non-null numbers between `-nbvar` and `nbvar` ending with 0 on the same line; it cannot contain the opposite literals `i` and `-i` simultaneously. Positive numbers denote the corresponding variables. Negative numbers denote the negations of the corresponding variables.

Each atomic sentence (called a *variable* above) is associated with a positive integer. Thus, literals are represented by integers other than 0, which is used to terminate clauses.

The output when running the SAT solve zChaff[2] on our example is as follows:

```
Z-Chaff Version:  Chaff II
Solving example.cnf ......
CONFLICT during preprocess
Instance Unsatisfiable
Random Seed Used 0
Max Decision Level 0
Num.  of Decisions 0
( Stack + Vsids + Shrinking Decisions )
Original Num Variables 3
Original Num Clauses 8
Original Num Literals 16
Added Conflict Clauses 0
Num of Shrinkings 0
Deleted Conflict Clauses 0
Deleted Clauses 0
Added Conflict Literals 0
Deleted (Total) Literals 0
Number of Implication 3
Total Run Time 0
RESULT: UNSAT
```

The contradiction was found quickly during the preprocessing stage of the solver and it reports that the clauses are unsatisfiable (UNSAT).

zChaff

You'll want to get zchaff working either on a lab machine or on your home computer. For this you will need a `C++` compiler. Some instructions that may or may not work for you are as follows:

Installing `C++` on your own computer

[2]See URL `http://www.princeton.edu/~Echaff/zchaff.html`

- OS X: Install the Xcode developer tools. Choose the appropriate version depending on which version of OS X you're running. You'll then be able to access the `g++` compiler from a terminal.

- Windows: Install Cygwin, which is an environment that gives you a prompt and software that looks and feels just like UNIX. On the Cygwin page, click "Install Cygwin now" on the top right. This points you towards an installation executable. Run it, and choose all defaults for the first few screens. You'll then need to arbitrarily choose a mirror site from which to download. Finally, on the "select packages" screen, expand the "Devel" category, scroll down, and select the packages "`gcc-g++`" and "make". Feel free to skim the list of packages to see if you see any other favorite open source tools you'd like to have around. In particular, look at the "Editors" category, as you might find an editor there you wish to use. When done, click the Next button at the bottom. Once installed, you'll get a shortcut on your Start Menu and on your Desktop that will give you a UNIX-like prompt. Note that your home directory within Cygwin is actually the Windows folder `C:\cygwin\home\username`. You'll need to know where this is if you want to find your files using traditional Windows navigation, such as when you want to submit them.

- Linux: Any self-respecting Linux distribution already has the `g++` compiler installed.

Installing zchaff

- Navigate to the zchaff website (`http://www.princeton.edu/ chaff/zchaff.html`), select the current 32 bit version, and download the zip file to a location of your own choosing. Unzip the contents of the file somewhere that you can access from your command prompt.

- Using your command prompt (terminal window or cygwin window, depending on which operating system you are using), navigate to the directory where you unzipped the zchaff files. Type `make` at the prompt to compile all of the code.

- When complete, you will have an executable zchaff file (called `zchaff` on Mac/Linux, or `zchaff.exe` on Windows). Copy that file somewhere into your path where it can be automatically found. To do this, in your terminal window, type `echo $PATH` which will show you a list of all directories that are automatically searched whenever you execute a command. Pick on of these to copy your zchaff executable into.

- If the above worked, you should be able to navigate to any random directory you like, type `zchaff`, and you should see the program respond.

SATSolver

In order to aid Python programmers in working with zChaff (written in `C++`) we have support Python code called `SATSolver`, described in this section. You can find the `SATSolver.py` file on Blackboard.

Clauses in `SATSolver` are represented in lists. A clause is a sequence of non-zero integers much like the DIMACS CNF format clause lines, but without 0 termination.

`SATSolver` provides two functions for invoking zChaff:

```
testKb(clauses)
testLiteral(literal, clauses)
```

`testKb` is used for testing to see if a particular knowledge base is satisfiable. It writes a DIMACS CNF query file, and executes zChaff as an external process. The result returned will be `True` if the combined clauses are satisfiable, and `False` otherwise.

Often, the user will simply want to know at a given point if a literal can be proved true/false or not relative to a previously consructed knowledge base. For this, we use the the `testLiteral` function can be used. It returns one of the symbols `True`, `False`, and `None`, which represent whether a particular literal is true, false, or unprovably neither relative to the knowledge base.

Here we provide a test case in which we take our liar and truth-teller example and apply the SAT-solver code.

```
clauses = [[-1,-2],[2 ,1],[-2,-3],[3,2],[-3,-1],[-3, -2],[1,2,3]]
print ‘Knowledge base is satisfiable:’,testKb(clauses)
print ‘Is Cal a truth-teller?’,
result = testLiteral(3,clauses)
if result==True:
print ‘Yes.’
elif result==False:
print ‘No.’
else:
print Unknown.
```

This code may be used to efficiently solve the exercises on homework #2. You may want to try it out on your own to make sure you have it working.