Principles of Knowledge Representation and Reasoning

University of Freiburg
Winter Semester 2013/2014 Department of Computer Science

Exercise Sheet 4

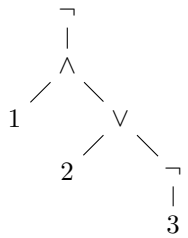
Due: November 20th, 2013

Exercise 4.1 (Project: Handling Propositional Formulae, 1+3+2+4)

The aim of this exercise is to write a small program that parses propositional logic formulae, translates them to CNF, and decides their satisfiability by using an existing satisfiability solver. You can use any programming language you like (given that it is usable under Ubuntu 12). It must includes a makefile for compilation.

Source code **must be submitted** to: `hue@informatik.uni-freiburg.de`.

We restrict ourselves to postfix notation[1] to keep parsing of the formula simple. In these formulae propositional variables are written as (non-zero) positive integer numbers. Only the following propositional connectives are used: `not` (unary), `or` (binary), `and` (binary). After parsing, a formula is internally represented as a binary tree (Figure 1 gives an example) and must be converted to CNF.



Postfix notation: `3 not 2 or 1 and not`

Formula: $\neg\big(1 \wedge (2 \vee \neg 3)\big)$

NNF: $\neg 1 \vee (\neg 2 \wedge 3)$

CNF: $(\neg 1 \vee \neg 2) \wedge (\neg 1 \vee 3)$

Figure 1: Example for postfix notation, formula, CNF, binary tree.

We consider both the standard CNF translation given in the lecture as well as the labeling CNF conversion which is a polynomial CNF conversion that preserves satisfiability. This conversion is achieved from the NNF by labeling every non-trivial subformulas with a new variable.

If b is a new variable the subformula and $l_1, ..., l_n$ the literals in the formula. The encodings are:

- $(l_1 \vee l_2) \leftrightarrow b$ is encoded into $(\neg l_1 \vee b) \wedge (\neg l_2 \vee b) \wedge (l_1 \vee l_2 \vee \neg b)$
- $(l_1 \wedge l_2) \leftrightarrow b$ is encoded into $(l_1 \vee \neg b) \wedge (l_2 \vee \neg b) \wedge (\neg l_1 \vee \neg l_2 \vee b)$

If we consider the example above, we denote by b_1 the formula $\neg 1 \vee (\neg 2 \wedge 3)$ and b_2 the formula $\neg 2 \wedge 3$. We have as a translation: $b_1 \wedge (b_1 \vee 1) \wedge (b_1 \vee \neg b_2) \wedge (\neg 1 \vee b_2 \vee \neg b_1) \wedge (\neg 2 \vee b_2) \wedge (3 \vee b_2) \wedge (1 \vee \neg 3 \vee b_2)$.

More explanations can be found on `http://eprints.biblio.unitn.it/1573/1/A_SAT-based_tool_for_solving_configuration_problems.pdf`

[1] `http://en.wikipedia.org/wiki/Postfix_notation`

For evaluating CNF formulae you can use an existing propositional satisfiability solver (SAT solver), e.g., the MiniSat solver `http://minisat.se/`. Virtually all SAT solvers accept as input the simple DIMACS format:

```
p cnf 5 2
1 -2 3 0
-1 2 5 4 0
```

The first line specifies that it is a CNF problem (`p cnf`) and gives the number of variables and clauses (in this case 5 variables; atoms $1, \ldots, 5$ and 2 clauses). Each of the following lines specifies one clause: positive integers represent positive literals, negative integers represent negative literals with 0 terminating the clause/line.

(a) Write a parser for the given postfix format that generates a binary tree for a given formula (or use the one provided for the java language).

(b) Write a function to convert an arbitrary formula to CNF both using the standard binary tree representation and the labeling CNF conversion.

(c) Write a function which can generate random 3-DNFs. The function must take as input: the number of conjunctions and the number of variables.

(d) Write a function to output the CNF in DIMACS format and test the satisfiability of randomly generated DNFs by converting them in CNF then in DIMACS. Discuss the efficciency of the two translations.

Your program should take as argument one filename to read the formula from and output the CNF in DIMACS on the standard output. It should not write anything else than the DIMACS format in order to pipe it as a minisat's input.