Principles of Knowledge Representation and Reasoning

B. Nebel, S. Wölfl, F. Lindner — University of Freiburg
Winter Semester 2015/2016 — Department of Computer Science

Exercise Sheet 3

Due: November 11th, 2015

Exercise 3.1 (Formula Game and Reduction, 2 + 5)

(a) The Formula Game is a two-player game played on a given quantified Boolean formula (in prenex normal form) $Q_1 p_1 \dots Q_k p_k \psi$. The rules are simple: If the outermost unassigned variable p_i is universally (existentially) quantified, it is the turn of player U (player E resp.) who assigns a truth value to that variable p_i. Thus both players finally construct a truth assignment I to the variables occurring in the matrix formula ψ. Player E wins the game if $I(\psi) = 1$; otherwise, player U wins the game.

Check whether one of the players U or E has a strategy for winning the formula game for the following formulae:

(a) $\forall p \forall q \exists r \forall s \big((p \wedge r) \to (q \wedge s)\big)$

(b) $\forall p \exists q \exists r \big((p \to q) \wedge (q \to \neg r) \wedge (r \vee \neg p)\big)$

(b) We consider the following two-player game $\mathcal{G}$ played on a directed graph $\langle V, A\rangle$ with a designated start node $v_0 \in V$. Player 1 and player 2 choose in turn some arc in the graph such that each chosen arc starts in the head of the previously chosen arc. Player 1 begins with choosing an arc starting in node v_0. A player looses the game if s/he is unable to choose an arc to a not yet visited node in the graph.

Show that the following problem is PSPACE-complete.

Instance: A directed graph $\langle V, A\rangle$, a start node v_0.

Question: Does Player 1 have a strategy for winning $\mathcal{G}$?

Hint: Existence of a winning strategy in the formula game (see exercise 3.1) is known to be PSPACE-complete even for QBF of the following form:

$$\exists x_1 \forall x_2 \exists x_3 \forall x_4 \dots \exists x_{2k-1} \forall x_{2k} \exists x_{2k+1} \psi,$$

where ψ is a 3-CNF formula. For the reduction construct for a given formula of this form a directed graph. The following subgraphs will be useful:

- For each propositional variable introduce a subgraph with four nodes that represents that a variable has been assigned a truth value.

$$\begin{tikzcd}
& \boxed{\mathcal{I}(x_i)\text{ undefined}} \arrow[dl] \arrow[dr] & \\
\boxed{\mathcal{I}(x_i) = F} \arrow[dr] & & \boxed{\mathcal{I}(x_i) = T} \arrow[dl] \\
& \boxed{\mathcal{I}(x_i)\text{ defined}} &
\end{tikzcd}$$

The current player will have to decide on the truth value of the next unassigned variable x_i. Note that the node corresponding to the chosen assignment may not be revisited in the game.

- Furthermore introduce nodes for each clause c_i of ψ and the literals l_{i_1}, $\dots, l_{i_3}$ occurring in it. For example, if $c_i = x_{i_1} \vee \neg x_{i_2} \vee x_{i_3}$:

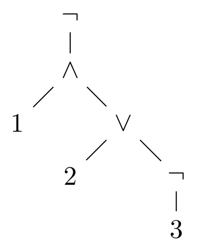
$$\begin{tikzcd}
& \boxed{c_i} \arrow[dl] \arrow[d] \arrow[dr] & \\
\boxed{x_{i_1}} \arrow[d] & \boxed{\neg x_{i_2}} \arrow[d] & \boxed{x_{i_3}} \arrow[d] \\
\boxed{\mathcal{I}(x_{i_1}) = T} & \boxed{\mathcal{I}(x_{i_2}) = F} & \boxed{\mathcal{I}(x_{i_3}) = T}
\end{tikzcd}$$

Finally discuss the size of your graph and relate the winning strategies in the games.

Exercise 3.2 (Project: Handling Propositional Formulae, 1+3+2+4)

This exercise is a project. You are asked to write a small program that parses propositional logic formulae, translates them to CNF, and decides their satisfiability by using an existing satisfiability solver. Please submit your code to `lindner@informatik.uni-freiburg.de` until **November 25th**. You are free to use Python, Java, or Scheme.

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.

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

We consider both the standard CNF translation given in the lecture as well as the labeling CNF conversion which is a faster CNF conversion that preserves satisfiability. The labeling CNF conversion is explained in the document you can obtain from `http://eprints.biblio.unitn.it/1573/1/A_SAT-based_tool_for_solving_configuration_problems.pdf` (see Section 2.3.2). Roughly, the main idea is to recursively label non-trivial subformulas θ_n with a new variable b_n, and to represent the original formula as a conjunction of terms of the form $\theta_n \leftrightarrow b_n$. In the document you also find an extensive example (Example 2.3.1.). Try to understand it first!
Moreover, in the document an improved version of the labeling CNF conversion, which takes the polarity of subformulas into account, is discussed. You do not need to implement this one.
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.
Your tasks:

(a) Write a parser for the given postfix format that generates a binary tree for a given formula (or use the python parser provided on the web).

(b) Write two functions to convert an arbitrary formula to CNF — one using the standard conversion and the other using the labeling CNF conversion.

(c) Write a function which can generate random DNFs with conjunctions of size exactly 3. 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 first in CNF, then in DIMACS. Discuss the efficiency 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 the output as a MiniSat's input.