Artificial Intelligence – Lab Assignment 2

UNIZG FER, academic year 2011/12

Handed out: April 18. Due: April 28 at 23:59

The topic of this lab assignment is automated reasoning with propositional logic. The task is to implement a theorem prover using refutation resolution and the set-of-support strategy. Your solution should be structured modularly as described below.

Loading and parsing a formula

Implement a function for reading a propositional logic formula from the standard input. The formula must then be parsed and converted into an appropriate tree structure. You can use any parsing technique to accomplish this. If the input formula is not a well-formed formula, the function should signal an error. The propositional variables should be represented with upper-case letters (`P`, `Q`, ...), while logical operators should be presented by symbols `&` (operator $\wedge$), `|` (operator $\vee$), `~` (operator $\neg$), `->` (operator $\rightarrow$), and `<->` (operator $\leftrightarrow$). Some examples of formulae:

```
(P -> Q) & ~R
((P | Q) | ~R) & S
(P & Q) <-> (R | S)
```

Note: If this makes it easier for you, you may parse the formula in prefix notation. If you don't know how to parse a formula, you may skip this task (you will lose 1/3 of the points). In that case you can work with predefined data structures (you can hard-code the formulas used in the examples).

Converting a formula into clausal form

Implement a function *cnfConvert* for converting a formula of propositional logic into the conjunctive normal form (CNF) in four sequential steps, and then into the clausal form (set of sets of literals). Pay attention to involution and factorization, which should be applied in every step. Also note that each step should be repeated until the result no longer changes.

Note: You may skip this task (you will lose 1/3 of the points). In that case, modify the following task so that the input to the resolution procedure is a CNF formula.

Refutation resolution

Implement a theorem prover in propositional logic based on refutation resolution (function *plResolution*). Use the set-of-support strategy as the resolution strategy. The function takes as input a set of premises $F_1, \ldots, F_n$ and a goal formula G. The function returns *true* if the goal formula is deductively follows from the premises, and *false* otherwise. Conversion of formulas into clausal form should be done inside the *plResolution* function

using the *cnfConvert* function. The function should also output the number of steps in the deduction process (number of times the resolution rule was applied) as well as the maximum number of clauses that were stored in memory at one time. Additional parameters of the procedure are the maximum number of steps and the maximum number of clauses to be stored in memory. If these limitations are exceeded, the proof procedure should terminate and signal an error.

Be sure that you never resolve the same pair of clauses more than once and that you don't generate clauses that are already generated.

You should also implement a simplification strategy as follows. The strategy should eliminate all redundant and irrelevant clauses from the set of clauses after each application of the resolution rule. Removing redundant clauses is based on the absorption equivalence $F \wedge (F \vee G) \equiv F$. Assuming that clauses are represented as sets of literals, if the set of clauses contains a pair of clauses C_1 and C_2 for which $C_1 \subseteq C_2$, then clause C_2 may be removed from the set. Removing irrelevant clauses amounts to removing all clauses that are valid formulae. A clause is valid iff it contains a complementary pair of literals.

Write a wrapper function that will read in a set of premises and a goal formula from standard input (each formula should be in its own line, the target formula is the one that is given last), as well as the desired proof strategy (`0` – set-of-support strategy, `1` – set-of-support strategy plus simplification strategy). The function should print out whether the deduction is proven, the number of steps of the proof procedure, the maximum number of clauses in memory at a time, and the proof sequence.

Examples

Input:

```
(A&B)<->C
C->B
1
```

Output:

```
Proven
Number of steps: 4
Maximum number of clauses: 6
1. ~A v ~B v C
2. A v ~C
3. B v ~C
- - - - - - - - - -
4. C
5. ~B
--------------------
6. A    (2, 4)
7. B    (3, 4)
8. ~C   (3, 5)
9. NIL  (8, 4)
```

Input:

```
((~P & Q) <-> (R|S)) | (~P -> S)
```

```
((~S & R) -> (Q & P)) | ((P & R) | Q)
1
```

Output:

```
Proven
Number of steps: 77
Maximum number of clauses: 11
1. P v ~Q v R v S
2. P v Q v ~R v S
- - - - - - - - -
3. ~P v ~R
4. R
5. ~S
6. ~P v ~Q
7. ~Q
--------------------
8. ~P   (3, 4)
9. Q v ~R v S   (2, 3)
10. P v Q v S   (2, 4)
11. P v Q v ~R  (2, 5)
12. P v ~R v S  (2, 7)
13. Q v S       (10, 8)
14. P v Q       (10, 5)
15. P v S       (10, 7)
16. Q v ~R      (8, 11)
17. ~R v S      (8, 12)
18. P v ~R      (11, 7)
19. ~R  (18, 8)
20. P   (18, 4)
21. S   (15, 8)
22. Q   (13, 5)
23. NIL (22, 7)
```