CSC322 Spring 2019

Project 2 – Adding preprocessing to a Prolog-based SAT solver

In this project, you will write add preprocessing steps to the Prolog-based SAT solver described in the paper *A Pearl on SAT Solving in Prolog.*

You should work in groups of 2, but groups of 1 or 3 are also accpetable if this works better for you (3 is the maximum.) You should use the `SWI Prolog` system, accessible on `linux.csc.uvic.ca` via the `prolog` command. Your code should be compatible with the provided SAT solver code.

Task 1

(Worth 8/10)

To complete the basic task, you must write code that will do a pure literal elimination preprocessing step. Recall the a literal ℓ is *pure* with respect to a CNF formula $\mathcal{C}$ if every occurence of ℓ in $\mathcal{C}$ has the same polarity. The *polarity* of ℓ is *negative* if ℓ is a negated variable and *positive* otherwise.
Pure literal elimination requires the following steps:

1. Identify pure literals
2. Remove clauses containing occurrences of any pure literal
3. Set each variable corresponding to a pure literal to the polarity of the literal

Due to the extremely efficient nature of DPLL with unit propagation using the watched literal method, modern SAT solvers only apply pure literal elimination once as a preprocessing step.
Your implementation of pure literal elimination will use the representation of clauses discussed in the paper. In particular, a clause is a list of lists of literals. Each literal is a *pair* consisting of a *polarity* and a *variable*. Here is a polarity is either `true` or `false` and a variable is just a Prolog variable, e.g., `X`. Pairs in Prolog are represented using the `-` constructor. So the literals corresponding to the variable `X` would be `true-X` and `false-X`. The CNF $\{\{\overline{x}, y\}, \{x, \overline{y}, z\}\}$ would have the representation

```
[[false-X,true-Y],[true-X,false-Y,true-Z]]
```

You should define a predicate `pure_literal_eliminate/4`. If `Clauses` is a list representing a CNF and `Vars` is the list of variables appearing in `Clauses`,

```
?- pure_literal_eliminate(Clauses,Vars,El_Clauses,El_Vars)
```

should succeed with `El_Clauses` being bound to the list of clauses representing the resulting CNF (as described in (2) above) and `El_Vars` being the list of variables with any corresponding to a pure variable being bound to its polarity. For example, we have:

```
?- pure_literal_eliminate([[false-X,true-Y],[true-X,false-Y,true-Z]],[X,Y,Z],El_Clauses,El_Vars).
El_Clauses = [[false-X, true-Y]],
El_Vars = [X, Y, true] .
```

Note here that `Z` is the only variable that occurs as a pure literal (with positive polarity,) so the one clause containing it is removed, and it is set to `true` in the list of variables.

Some ideas Prolog has built-in predicates `flatten/2` which turns a list-of-lists into a list, and `list_to_set/2` which remains duplicate entries from a list. You can use these predicates to go from a CNF formula in the given representation to a list containing all the literals that appear in the formula (without repetition.)
Once you have the set of all literals, it is relatively easy to define a predicate `purify/2` such that if `Literals` is the set of all literals, then `purify(Literals,PureLiterals)` will bind `PureLiterals` to the set of all literals which occur with only one polarity in `Literals`. Note that `purify` can take advantage of the fact that `list_to_set` already removed any duplicate literals (i.e., we don't have to worry about the particular polarities in the literals.)
Once we have `PureLiterals`, we can define predicates `eliminate_clauses/3` and `set_vars/3` that take case of steps (2) and (3) above. In short, we can define:

```
pure_literal_eliminate(Clauses,Vars,El_Clauses,El_Vars) :-
        flatten(Clauses,FlattenedClauses),
        list_to_set(FlattenedClauses,AllLiterals),
        purify(AllLiterals,PureLiterals),
        eliminate_clauses(PureLiterals,Clauses,El_Clauses),
        set_vars(PureLiterals,Vars,El_Vars).
```

Complications It would be nice at this point if we could use built-in Prolog predicates such as `select` (to remove duplicates) and `intersection` (to see whether a pure literal occurs in a clause.) Unfortunately, things are complicated by the representation we are using for CNF formulas, in particular, using Prolog variables to represent variables in the formula. The predicates described above consider two terms to be equal if they can be *unified*, but for our purposes we need syntactic equality, e.g., to decide whether variables are equal to each other. Fortunately, Prolog provides the predicates `==` for syntactic equality, and `\==` for syntactic inequality. If this isn't clear, try the following:

```
?- X = Y.
X = Y.

?- X \= Y.
false.

?- X == Y.
false.

?- X \== Y.
true.
```

Single equality `=` means unifiability. Clearly we can unify the variables `X` and `Y`, that is why the first query succeeds (we can just set `X` to `Y`) and the second fails. For syntactic equality `==`, the third query faiis. So when you are testing for an occurrence of a variable (or overlap between a clause and the set of pure literals) you will need to use `==`, when you are testing for nonoccurrence (or disjointness) you will need to use `\==`, etc.

Task 2

(Worth 1/10)

Implement one of the preprocessing steps described in the paper *Effective Preprocessing in SAT through Variable and Clause Elimination.*

Task 3

(Worth 1/10)
The goal of this task is to evaluate the effectiveness of pure literal elimination as a preprocessing step. Using the testing harness provided with this distribution of the solver, compare the performance of the solver using pure literal elimination versus static variable ordering.

Deliverables and Detailed Grade Breakdown

For Task 1, you only need to submit a single `.pl` file named pure_literal_elminate.pl (start with the supplied template.) For task 2, submit a separate, appropriately named source file, as well as a documentation file. For task 3, do not submit any code, just a summary report. Submit everything as a single `.tar.gz` file. The name of the file should be the conneX ID of the group member who submits the file (only one submission per group is required.) The submitted file should extract to a single directory with the same name as the `.tar.gz` file (even if you only do Task 1 and are submitting a single file, put it in a directory. More specifically, to create the submission, you should be in a directory which contains the directory `subid`, and execute the following:

```
tar cvzf subid.tar.gz subid
```

where `subid` is the conneX ID of the submitter, as described above.)

The grader will test your submission as follows: after executing `tar xvzf subid.tar.gz`, the grader should be able to find the Prolog source files and the report for Task 3, if you are submitting this. Include a README with a listing of all the files in the directory, as well as names and Vnumbers of all group members.

For Task 1, the grader will run your code on a selection of test cases (I will being a subset of test cases when they are ready.)

For Task 2, the grader will not necessarily try to execute your code. You should provide a separate file explaining your code, including a transcript of its behaviour on a representative sample of inputs.

For Task 3, the grader will read your report.

You only need to provide one submission for your group.

Supplied Files

The assignment page has links for the following files:

`flops.pdf` – the paper *A Pearl on SAT Solving in Prolog*

`sat_solver_when.pl` – a version of the solver that will work with SWI Prolog