Logic: unit propagation

In this practical you will undertake tasks related to logical inference. You will also gain further experience in C programming.

This practical is worth 25% of the continuous assessment portion of this module and is due 21:00 on Wednesday April 8th.

Background

As discussed in the lectures, the DPLL algorithm can determine whether a set of clauses leads to a contradiction. Its complexity is exponential in general. Here we will implement only one element of the DPLL algorithm, namely unit propagation. There are two rules of unit propagation:

1. If we have a unit clause x and a clause c in which $\overline{x}$ occurs, then we remove $\overline{x}$ from c. (As in the lectures, x is a positive or negative literal, and $\overline{x}$ denotes the converse of x.)

2. If we have a unit clause x, then we remove all clauses in which literal x occurs.

For a given set of clauses we can now apply unit propagation exhaustively, to obtain a set of unit clauses, both those present in the given set of clauses, and those that result from unit propagation. It is also possible that the empty clause is obtained, which signals a contradiction.

1 Implementation

Implement unit propagation in C, to output unit clauses that can be obtained from a given set of clauses.

The initial set of clauses is read from standard input. Each line represents one clause, as a list of tokens separated by spaces. A token represents a literal. If the token starts with '-', then it is a negative literal, with the rest of the token representing the variable. If the token does not start with '-', then the token as a whole is a variable. Variable names consist of one or more characters not including '-' or space ' '. An empty line represents the empty clause.

The output is one line, listing all unit clauses (literals) that are obtainable through unit propagation (or that were already included in the input) separated by a single space; the line does not end on a space. The literals are to appear in the lexicographical order of the variables. Each variable should be listed at most once.

If unit propagation ever produces the empty clause (or if the input set of clauses already included the empty clause) then the output line consists of a single '-'. Examples of required output for given input are included in the stacscheck directory.

Hints:

- For parsing the input, you may want to learn about `strtok`.
- For lexicographically sorting a list of literals, you may want to learn about `qsort` and `strcmp`. Note that this requires conversion of whatever your internal representation is to an array.
- As always, a difficult problem becomes manageable by splitting it up into smaller and easier problems. My advice is to start by writing routines to parse the input, and to store the literals and clauses in customised datastructures, by defining appropriate `struct` types and auxiliary routines for manipulating such types.
- It is not good practice to make a priori assumptions about e.g. the maximum number of variables, the maximum length of clauses, or the maximum number of clauses.
- Do your routines free malloc-ed memory afterwards?

In your report, you would at a minimum have the usual sections about design, implementation, and testing.

2 Time complexity

In this practical we are not concerned with the part of DPLL that makes it have an exponential time complexity, and your implementation is likely to run in polynomial time. But what exactly is the time complexity? Analyse this in your report.

Would it be possible to implement the same functionality (i.e. exhaustive unit propagation) in linear time? Outline in the report how this would be done. For the basic part of this assignment, it is not required to implement a linear-time solution.

3 Beyond a basic solution

A linear-time complexity for this problem is not easy to achieve in C. Efforts in this direction can be rewarded with marks in the highest band, provided the basic part of the assignment is done to a high standard.

Submission

Submit a zip file containing your report as a PDF and a directory called `LogicDir` containing the code, including a `Makefile` with targets `clean` (which deletes executables and object files) and `main`. The latter should produce an executable `main`, which can be run as e.g.:

```
main < myfile.in > myfile.out
```

The output redirection to a file is something you would normally omit during code development. It is required though for automated testing.

You can test stacscheck by:

```
stacscheck /cs/studres/CS2002/Practicals/W09-Logic/stacscheck
```

Marking

0-8 Work that fails to demonstrate understanding of unit propagation.

9-13 An attempt at a working implementation, which failed for lack of understanding of C, with nonetheless an informative report about some of the relevant issues.

14-17 A correctly working implementation, and a report that addresses the main concerns of this practical, including discussion of time complexities along the lines of Section 2.

18-20 All of the above, and in addition efforts towards a linear-time implementation.

Rubric

There is no fixed weighting between the different parts of the practical. Your submission will be marked as a whole according to the standard mark descriptors published in the Student handbook at:

`https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/feedback.html#General_Mark_Descriptors`

You should submit your work on MMS by the deadline. The standard lateness penalties apply to coursework submitted late, as indicated at:

`https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/assessment.html#lateness-penalties`

I would remind you to ensure you are following the relevant guidelines on good academic practice as outlined at:

`http://www.st-andrews.ac.uk/students/rules/academicpractice`