CSC322 Spring 2019

Project – Using SAT to find valid Entailments

In this project, you will write a simple program to translate Boolean formulas into CNF using the translation given in class with the goal of using a SAT solver to determine whether, given Boolean formulas $\mathcal{A}_1, \ldots, \mathcal{A}_k, \mathcal{A}$, whether $\mathcal{A}_1, \ldots, \mathcal{A}_k \models \mathcal{A}$ (for the basic task, $k = 0$.)

You should work in groups of 3. You may use any language you want for your implementation, but it should be possible for the grader to run and test your code on `linux.csc.uvic.ca`, without installing any new software. The `minisat` system is available on `linux.csc.uvic.ca`.

Task 1

(Worth 10/15 of grade.)

To complete the basic task, you must write code that will translate a single formula $\mathcal{A}$ into a CNF $\mathcal{C}$ such that $\mathcal{C}$ is satisfiable iff $\not\models \mathcal{A}$ (or, in other words, $\mathcal{C}$ is unsatisfiable iff $\mathcal{A}$ is valid. This is exactly the translation described on pp. 7-10 of the course notes. You will need to determine how to do this translation for all the connectives we are using, as outlined below under **Parsing** below.) The CNF you produce should be in DIMACS format, suitable as input to the `minisat` solver. Your code should invoke `minisat` to determine whether C is satisfiable, and use this result to decide whether $\mathcal{A}$ is valid.
So, for example, for input `A1 v ∼A1` your program should return `VALID` while for input `A1 v A2` is should return `NOT VALID`.

Task 2

(Worth 2/15 of grade.)

For the next task, you should extend the code from Task 1, so that in the case that $\mathcal{A}$ is not valid, an assignment which gives $\mathcal{A}$ the value `F` is returned. So for the second formula above it would return `NOT VALID : A1 = F, A2 = F`.

Task 3

(Worth 2/15 of grade.)

For this task, you should extend the validity checker to check entailments as described above. Not that in class we did not discuss how to extend the method of translating a formula to CNF to translating an entailment to a CFN C such that $\mathcal{A}_1, \ldots, \mathcal{A}_k, \mathcal{A}$, whether $\mathcal{A}_1, \ldots, \mathcal{A}_k \not\models \mathcal{A}$ iff C is satisfiable. Note that in the case when the entailment does not hold, you should given a assignment that makes each of $\mathcal{A}_1, \ldots, \mathcal{A}_k, \mathcal{A}$, whether $\mathcal{A}_1, \ldots, \mathcal{A}_k$ evaluate to `T` and makes $\mathcal{A}$ evaluate to `F`.

Task 4

(Worth 1/15 of grade.)
For this task, you should try to compare the performance of the translation-based checker to a brute-force checker that just tries all assignments to variables. Note that `minisat` has a verbose option (`-verb=2`) which will report CPU time. For the brute-force checker you will have to collect timing information yourself. Clearly CPU time is only a rough estimate of performance. Can you notice any difference?

Details

Your code will roughly fall into three major parts: (1) Parsing Boolean formula input into an internal tree representation (2) Using the internal representation to produce CNF in DIMACS format (3) Using the CNF as input to `minisat` to prodcuce the required output.

Parsing

We will use Boolean formulas without $\leftrightarrow$ or $\perp$. Formulas will use `~` for negation, `&` for conjunction, `v` for disjunction and `->` for implication. Variables will have the form `x` followed by a string of digits (not starting with `0`.)

More specifically, we have the following token classes, with regular expressions that define them:

```
NEGOP = ~
ANDOP = &
OROP = v
IMPOP = ->
LPAREN = (
RPAREN = )
VAR = A[1-9][0-9]*
```

For specifying the syntax of formulas, we will use EBNF (extended BNF). This will make writing the parser easier, in particular dealing with the fact that conjunction and disjunction are left-associative. Here is the grammar:

```
SENT ::= DISJ | DISJ IMPOP SENT
DISJ ::= CONJ{OROP CONJ}
CONJ ::= LIT{ANDOP LIT}
LIT ::= ATOM | NEGOP ATOM
ATOM ::= VAR | LPAREN SENT RPAREN
```

We write the grammar in this form to make it particularly easy to write a recursive descent parser.

Here some notes on writing a recursive-descent parser. Instructions on how to support EBNF are given near the end. You might also want to refer to this. You need to build a tokenizer that provides the stream of tokens to your parser. Python has particularly nice support for using regular expressions to build a tokenizer. You can read more about this here.

The grammar is written as it is in EBNF to simplify recursive descent for the left-associative and and or operators. The implication is right-recursive, and so the usual recursive BNF rule works fine.

The output of your parser should be an internal data-structure which gives a representation of the abstract syntax tree corresponding nodes. Internal nodes will just need to contain the operator which is the principle connective of the subformula rooted at the node. Leaf nodes should contain the number of the corresponding variable. You might actually want to include more information in your tree to simplify the process of producing the CNF (see below)

Translation to CNF

This is relatively straightforward once you have the abstract syntax tree (AST) but there is one consideration that will make your life easier: the translation we gave in class introduces a new variable for each subformula. It would be nice to generate these variables as you create the corresponding node during parsing. But how to do this in a way that does not conflict with the actual variable of the formula, which you won't have yet? The key thing to note is that the internal representation of variables are just numbers (fortunately this is also how they are also reprsented in DIMACS format.) You can use odd numbers for the variables corresponding to subformulas. What to do about the actual propositional variables in your formula? Just multiply the number of the variable by 2. Note that this convention will make task 2 pretty easy. `minisat` will give an assignment for all the variables in the CNF you give it, if the CNF is satisfiable. Now it will be easy to get the variables from the original formula – just look for the even ones, and divide by 2 to get back the original variable number.

The result of this translation should be suitable for input to `minisat`, that is seqence of lines of the form:

```
p cnf <# variables> <# clauses>
<list of clauses>
```

Here `<# variables>` actually means the largest number you use to represent a variable. E.g., if you use variables `1, 2, 5` then you must put `5` for `<# variables>`.
Each clause is given by a list of non-zero numbers terminated by a `0`. Each number represents a literal. Positve numbers `1,2,...` are unnegated variables. Negative numbers are negated variables. Comment lines preceded by a `c` are allowed. For example the CNF formula

A_1, A_3, A_4
$\overline{A_1}, A_2$
$\overline{A_3}, \overline{A_4}$ would be translated as

```
c A sample DIMACS CNF
p cnf 4 3
1 3 4 0
-1 2 0
-3 -4 0
```

Interface to the SAT solver

`minisat` just reads from `STDIN`, so it is easy enough to pass it the output from the previous phase. You will then need some simple code to extract the relevant information from the output. Note that if you use the even-number-variable convention described above you will know which variables correspond to the original atomic variables from $\mathcal{A}$ – you just have to divide by 2 to get the original variable number. One problem with this approach is that there might not be an exact match between the number of variables that you actually use and the value that you give on the `p cnf` line. This isn't the end of the world, but it is a bit unfortunate.

Deliverables and Detailed Grade Breakdown

Your submission should include

1. Your code, with documentation on how to use it. Your code should produce *Linux commands* for each task (except Task 4). They should be called `vcheck1`, `vcheck2`, and `vcheck3`. The first two read a single Boolean formula from `STDIN` and output the result on `STDOUT`. The third should take a comma-separated list of formulas from `STDIN` and output the result on `STDOUT`
2. A `README` file describing the entire contents of the submission as well as any details you feel are relevant. Make sure you put the name and Student ID of *all* group members here.

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. 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.)

DO NOT submit any executable of `minisat`

The grader will test your submission as follows: after executing `tar xvzf subid.tar.gz`, the grader should either be able to find the command executables described above in the top level of directory `subid` or be

Task 1	
Parser code	4
CNF generation code	3
Interfacing code	2
Documentation	1
Remaining Tasks	
Task 2 code	2
Task 3 code	2
Task 4 report and code	1

Table 1: Grade Breakdown

able to execute `make clean`, followed by `make target` to create them. If you are not able to use `make` or include the excutables, you should give clear instructions on how to build your commands.

`For Task 4` you should submit the code of your brute-force solver, but the grader will not run it. You should experiment with this yourself and include a write-up of your findinings (comparing the *minisat* to brute-force approach.)

You only need to provide one submission for your group.