15-150: Functional Programming- Project 1

Kaustubh Iyer (kkiyer)

Description of the Problem:

Logical formulae have been largely used all over the world in mathematical proofs. A recurring theme has been the problem of finding an assignment of variables(true or false) in a propositional formula, such that, the formula would evaluate to true.
This problem has grown in popularity over the years, and has gained the nickname "SAT", which stands for 'satisfiability' of a formula. The reason this problem is so popular is mainly because it is part of the set of problems that are "NP-Complete", that is, any algorithm that would solve them is slow (exponential).

Description of the Solution:

It is true that there is no 'fast' way to solve SAT, some algorithms do exist that, on average, don't have to explore every single case to solve the problem. Thus, on average, these algorithms are faster, because, in most cases, they find the answer early.
One such algorithm that can solve SAT is called DPLL. This project aim to implement this algorithm to solve SAT.
DPLL works in the following way:

- We start by converting the proposition to CNF form, i.e., of the form:

$$(a_1 \vee a_2...) \wedge (b_1 \vee b_2...) \wedge ... \wedge (n_1 \vee n_2...)$$

- Now we get a list of all the unique variables in this formula.
- We travel over this list, and as we go, we give a truth value to each variable. We start by suggesting it's true.
- After suggesting its true, we reduce the formula using this value. We do this because we know that if one literal in Or of a bunch of literals is true, then the whole branch is true.
- We then carry on assigning true for the rest of the list, one by one.
- In case we find out that reducing the formula gave me a formula that's definitely false, then I say that making this variable true didn't work so I try making it false.
- If making a variable false doesn't work either, then I go back to the previous variable and try to make it false.
- If making a variable false worked, I try to go on by making the next true.

It can be seen that this method would compute all possible assignments which are potentially useful, while throwing away the bad ones. If it reduces a formula to true, then it succeeds, and if it exhausts all outcomes, then it fails.

Usage Instructions:

Once you have downloaded this package, extract its contents to a directory. If you would like to use a type of your own to represent propositional variables, you can change the type t on prop.sml to whatever you'd like to represent variables. To run the SAT solver, you must:

- Open a command prompt at the directory of the solver files.

- Run the SML repl with the command:
  >> sml
- Run the following command on the repl:
  >> CM.make "sources.cm";
- Initialize an instance of a satsolver in the following way:
  >> structure a = SatSolver(PropChar);

  Here, 'a' is a variable that can be called anything you'd like. If you created your own Prop structure and would like to use that instead, then replace "PropChar" with this structure's name.
- Now you may create a form. Below is an illustration through a simple example:
  >> val formula = a.Impl(a.And(PVar(#"x"),Not(PVar(#"y"))),a.Or(PVar(#"x"),Not(PVar(#"y"))));
- Notice that a.And, a.Or and a.Impl take two arguments and a.Not takes one. PVar(¡char¿) represents a single propositional variable.
- Now to get results, you may call any of the given functions on this formula:
  >> val assignment = a.sat(formula);
  >> val isunsat = a.isUnsat(formula);
  >> val isvalid = a.isValid(formula);

  The repl prints out the answers. For a.sat it gives you NONE if there's no satisfiable assignment of variable, and SOME of an assignment, if there is.
  a.isUnsat tells you if a propositional formula is unsatisfiable, i.e., no set of truth assignments make it true.
  a.isValid checks if a formula is valid, i.e., if any truth assignment would make it true.

Implementation Description:

- **Step 1: Conversion to CNF form-**
  Functions Used:
  - **removeimpl:** Given a formula, removes the implications, by replacing a => b with $\neg$ a $\vee$ b. Maintains the truth value of the formula, by not making any other changes.
  - **pushnegs:** Given a formula, push the negatives down to the literal level. This is done recursively, by eliminating Not(Not(x)) as x, and De-Morgans' law to push the negations down the formula.
  - **convert:** This does the final step, by applying the properties of distributivity to pull And's to the top level. a $\vee$ (b $\wedge$ c) = (a $\vee$ b) $\wedge$ (a $\vee$ c), (b $\wedge$ c) $\vee$ a = (b $\vee$ a) $\wedge$ (c $\vee$ a). We must be careful to case on the result of one distribute, and look if it results on another application of distribute.
  - **toCNF:** Applies the above functions sequentially, in the order- removeipl, pushnegs, and then convert, which results in the output of a valid CNF form.
  - **cnfify, and uncnfify:** Converts to and from a form to my dedicated structure for CNF forms.

  I use CNF forms because the DPLL algorithm works on certain special properties of these forms.
- **Step 2: Getting list of literals-**
  Functions:
  - **getvlist:** Takes a cnf form and generates a list of every literal used. Duplicates may exist.
  - **isin:** Checks if a literal occurs at least once in a given list of literals.

- **removeduplicates:** This functions removes all duplicate instances of a variable in a variable list.

This is where, while generating a variable list, a heuristic could possibly be applied to sort this list, but I don't go through this process.

- **Step 3: Solving SAT- helpers-** Functions:
  - **reduce:** This function reduces a propositional formula in cnf form by setting arg2 (a variable) to true in the equation. It uses DPLL techniques to get rid of disjunction chains.
  - **fixcnf:** This function makes sure the cnf structures are ok by seeing if the ors operate on only one element, and if so, pull that element out.
  - **satsolver:** This function accepts a cnf formula and a list of cnf literals, and returns either SOME of a list of assignments that make the form evaluate to true, or NONE if the formula cannot be satisfied. It recurses on the variable list, by first reducing the formula with the variable set to true, and then attempting to find an assignment for the formula that's left over, and if it is unsatisfiable, then attempting to set the negation of the literal to true and repeating the procedure. If the negation fails as well, the given formula is unsatisfiable.
    This uses the specifics of DPLL to reduce, and also to check if the form is in a position where it cannot be reduced further. If one of the literals in the conjunction is set to false, then the whole assignment is unsat, and so it returns NONE in this case.
- The remaining functions are the requested ones.
- sat takes in a form, converts it to cnf, and extracts a list of unique variables, and calls satsolver on it.
- isUnsat calls sat, and, if it returns NONE then returns true, else false.
- isValid checks if the negation of a formula is unsatisfiable (contrapositive of the condition for validity, and returns true if it is, and false otherwise.

Description of my CNF representation of formulae:

To be able to effectively apply DPLL to solve SAT of inputted forms, I converted them to CNF form- "Conjunctive normal form".
Here is a representation of a generic CNF form:

$$(a_1 \vee a_2...) \wedge (b_1 \vee b_2...) \wedge ... \wedge (n_1 \vee n_2...)$$

toCNF does this, by sequentially calling three recursive functions on my formula. These functions were quite simple- one to remove implications, one to push the negations down to literals and one to Bring the and operation out to global and operate on ors or variables.
Post this, I made a new datatype of my own that is dedicated to working with cnfs, called a "P.t cnf". It's definition is as follows:

```
datatype 'a cnf =
        L of 'a
        | Neg of 'a cnf
        | Andl of 'a cnf list
        | Orl of 'a cnf list
```

This makes and and or operate over a list rather than on two inputs, which is more convenient for reducing the formula with an assignment.

Heuristic for order of assignments of variables

My implementation of the DPLL has no special heuristic, it just assigns truth value based of the order of appearance of variables in the formula. The process is, hence, just an ordinary implementation of the DPLL with no special heuristics.

A simple, but effective way to cut off a large portion of the search space would be to assign true/false to the most frequently occurring variable first, and follow up with the second most frequent, and so on. This would, at each stage, cut a large portion of the formula.
Also, giving greater weight to more appearances of a literal in a smaller disjunction, since these would lead to the revelation of a contradiction faster, would also be an effective strategy in vastly reducing the search space of DPLL.

Soundness, completeness statements

Statement of completeness:

Let F be the set of all formulae. Let F' be a subset of F such that $\forall f \in F'$, f has at least one satisfiable truth assignment. Then:

$$\forall f \in F, f \in F' => sat(f) = \text{SOME x, where x is an assignment list}$$

Statement of soundness:

$$\forall f \in F, f \in F - F' => sat(f) = \text{NONE}$$

Why Completeness can be compromised, but not Soundness

We can sacrifice on the Completeness of a sat solver, but never on the soundness, because of complexity reasons. The reason is because improvements in average runtime of the algorithm generally involve completeness being sacrificed. Rendering the procedure unsound is a bad idea because that means that you aren't really improving the runtime of the algorithm, while sacrificing correctness, for seemingly no particular reason.