CMPT 383: Assignment #3

Anders Miltner
miltner@cs.sfu.ca

Due Nov 15

Introduction

In this assignment, we will be using Rust to make a SAT solver.

Given a boolean formula, a SAT solver determines whether there exists an assignment of variables to true or false, such that the boolean formula evaluates to true.

SAT solving is proven to be NP-complete, so in worst case, we do not have any algorithms that solve it in less than exponential time in the worst case. There is a polynomial-time reduction from arbitrary boolean formulas to boolean formulas in *conjunctive normal form*. These boolean formulas are written as:

```
Atom    :=   v
           | not v

Clause  := Atom_1 \/ ... \/ Atom_n

Formula := Clause_1 /\ ... /\ Clause_n
```

Formulas are a list of "Clauses" and-ed together. Clauses are lists of "Atoms" and-ed together. Atoms are either variables, or the negation of variables.

In the context of Rust, Variables are represented as chars. Atoms are represented as the enum `enum Atom {Base(Variable), Not(Variable)`. Clauses are represented by vectors of atoms. Full formulas are represented by vectors of clauses.

Thus, the Rust data: `vec![vec![Base(b),Not(c)],vec![Base(c)]]` represents the boolean formula $(b \vee \neg c) \wedge c$.

What does an empty formula represent? The Rust data `vec![]` represents the formula $true$. What does an empty clause represent? The Rust data `vec![vec![Base(a),Base(b)],vec![]]` represents the formula $(a \vee b) \wedge false$.

While we do not have polynomial time algorithms, there exist algorithms that do relatively well in the average case, that operate over CNF boolean formulas. One of the first such algorithms is DPLL. In this assignment, we will be writing a DPLL algorithm.

The fundamental DPLL algorithm works as follows:

```
procedure DPLL($\Phi$)
    UnitPropogate($\Phi$)
    AssignPureVars($\Phi$)
    If $\Phi$ is empty, return true
    If $\Phi$ contains an empty clause, return false
    $v \leftarrow$ ChooseVariable($\Phi$)
    return DPLL($\Phi \wedge \{v\}$) or DPLL($\Phi \wedge \{\neg v\}$)
```

Note that DPLL returns $true$ when Φ is empty because an empty Φ is just true. Note that DPLL returns *false* when Φ contains an empty clause because an empty clause is just false, so the conjunction of anything and false is false.

In this assignment, you will write some portions of the UnitPropogate procedure, some portions of the AssignPureVars procedure, and the full DPLL loop.

You've been provided a partial test suite. You can run these tests by running cargo test. You can also build your own formula, and get a result of whether or not it is satisfiable by running cargo run.

1 UnitPropogate

The UnitPropogate algorithm consists of two parts. First is finding a "unit clause," and then second is "propogating" that unit clause. This continues until there are no more unit clauses.

A unit clause is a clause with exactly one atom. If a unit clause exists, the variable in the atom *must* be assigned according to the atom. If the clause is `[Not(v)]`, then `v` must be assigned to false. If the clause is `[Base(v)]`, then `v` must be assigned to true.

The first function you must write is `find_propogatable`. If there is a singleton clause `[Base(v)]`, then `find_propogatable(f)` should return `Some(v,true)`. If there is a singleton clause `[Not(v)]`, then `find_propogatable(f)` should return `Some(v,false)`. If there are no singleton clauses, then `find_propogatable(f)` should return `None`.

The next function you must write is `propogate_unit`. This function will go through each clause, and update them according to the unit clause assignment.

If a variable `v` is set to `true`, then all clauses containing `Base(v)` can immediately be removed – they are already satisfied. If a clause contains `Not(v)`, then `Not(v)` should be removed from that clause – the `Not(v)` atom can never be satisfied.

If a variable `v` is set to `false`, then all clauses containing `Not(v)` can immediately be removed – they are already satisfied. If a clause contains `Base(v)`, then `Base(v)` should be removed from that clause – the `Base(v)` atom can never be satisfied.

For example, if `propogate_unit(f,v,true)` is called, then every clause containing the atom `Base(v)` should be removed. Furthermore, every atom `Not(v)` should be removed from the clause. Note: it may be useful to build a helper function.

The provided function, `unit_propogate(f)` will repeatedly call `find_propagatable` and `propogate_unit` until there are no more propogatable units.

2 AssignPureVars

The AssignPureVars algorithm consists of two parts. First is finding a "pure variable," and then second is "assigning" that pure variable. This continues until there are no more pure variables.

A pure variable is a variable that has the same "polarity" throughout the entirety of the formula. If a variable `v` always shows up as either `Base(v)` or as `Not(v)`, then that variable is "pure." If a variable is pure, it can safely be assigned to either true or false. Then, all clauses containing that variable can be removed.

The first function you must write is `find_pure_var`. If there are any pure variables `v` in `f`, then `find_pure_var(f)` should return `Some(v)`. If there are not any pure variables, then `find_pure_var(f)` should return `None`. The functions `get_vars(f)` and `is_pure(f,v)`, defined in `cnf_formula.rs`, will be useful for this function.

The next function you must write is `assign_pure_var(f,v)`. This function will go through each clause in `f`. If that clause contains `v`, then the clause can be safely removed.

The provided function, `assign_pure_vars(f)` will repeatedly call `find_pure_var` and `assign_pure_var` until there are no more pure variables.

3 DPLL

The function DPLL is described in the Introduction. Now you should implement `dpll` in your code. Note that `unit_propogate` and `assign_pure_vars` both modify the provided formula. Because of this, when recursively calling `dpll`, you cannot simply edit the provided `f` and provide that to each recursive call. Rather, you should clone `f` with `f.clone()`, and edit that, when adding the singleton clauses `[Base(v)]` and `[Not(v)]` to the formula.