Andreas Sekine
Math 480A
Final Project

Boolean Formulas (Propositional Calculus) in Sage

1 Background

1.1 Simplification Methods

There are a few widely used algorithms for minimizing Boolean formulas, most notable of which are the **Quine-McCluskey** algorithm, and the **Espresso** method. Both work by taking a Boolean formula and compute an equivalent expression using simpler operators and/or fewer terms. Both methods return a solution expressed in sum-of-products terms, a formula involving *and* terms *or*'ed together.

The computational and spatial complexity of Quine-McCluskey is exponential, and can quickly become unreasonable for complex formulas involving many variables. It is guaranteed to find a solution with the minimum number of terms.

The Espresso method uses heuristics to minimize a formula much more quickly, however it does not guarantee the simplest form of the expression. It typically finds solutions that are relatively short, and much more quickly than Quine-McCluskey.

2 Logic in Sage (The `logic` Module)

2.1 Current Release

As of version 3.0.1, the logic module relies the `SymbolicLogic` class to deal with Boolean logical statements and expressions. It has several function stubs, such as `prove, simplify,` and `combine`.

While `SymbolicLogic` did implement basic Boolean operations and functions, there were some poor design choices and holes left that left much to be desired. For example, it used the strings *"OPAREN"* and *"CPAREN"* to internally store parentathes, and used the name `eval` for evaluating a statement, which is generally bad style since python has a built in eval function.

2.2 Active Development

The logic module in sage is currently undergoing active development. The sage ticket `http://sagetrac.org/sage_trac/ticket/545` shows the current progress. There were many significant changes to the logic module:

- The code was refactored into several files. Before it all resided in *logic.py*, and now is sepp up according to functionality. For example, the parsing of the statement now is the responsibility of *logicparser.py*, and *boolformula.py* holds the related class
- The names were changed. Instead of being called through `SymbolicLogic`, a boolean function is initialized through the `propcalc` module, and uses the `BooleanFormula` class.
- Boolean formulas are stored internally in a tree structure now

2.2.1 Simplification

`BooleanFormula` objects in sage have a `simplify` method that can be used to reduce expressions to their minterms (sum-of-products form) by using the Quine-McCluskey algorithm. This is accomplished using the **boolopt** package developed by Michael Greenberg (`http://www.weaselhat.com/boolopt/`).

2.2.2 Suggested Improvements

Ticket 545 contains a patch that completely reworked the logic module, and is currently awaiting review. After inspection it seems that the patch still issues that need to be addressed.

- It seems like there should be a way to evaluate a given `BooleanFormula` with specified inputs. It may possible by calling *booleval.eval_formula)*, however there should probably be an interface for such basic functionality, without having to resort to making an entire truth table.
- `propcalc.formula` includes the following code:

  ```
  #verify the syntax
  f.truthtable(0, 1)
  ```

  The truthtable construction doesn't do any (direct) error checking, and won't throw any helpful error messages. Meanwhile the logicparser does plenty of syntax checking for valid input – is the truthtable call necessary?
- Currently *boolopt.py* is imported by propcalc by calling `sys.path.append('boolopt-1.1')`, and then importing the module. Applying the patch directly didn't work, and python complained about not finding the boolopt import. I had to fix it by copying *boolopt.py* to the parent directory (sage/logic).
- BooleanFormula._latex_ doesn't work properly, as it doesn't escape any latex reserved characters. Particularly the symbols `&, ^  and ~`.

  It may be possible to enter the latex formula in verbatim block, however this should probably be mentioned in the docstring if it's to be the case
- `BooleanFormula__eq__` has very naive behavior - it just performs literal comparison of trees, and you get behavior like the following:

  ```
  sage: p = propcalc.formula("a|b")
  sage: q = propcalc.formula("b|a")
  sage: p == q
  False
  ```

  This may be the desired behavior, but then maybe there should be a method `equivalent` that can find out if something is logically equivalent. This presents certain challenges, such as
- The docstring examples for `BooleanFormula.convert_expression` and `BooleanFormula.convert_cnf` are the same, and yet the functions do different things. Both contain the following:

  ```
  EXAMPLES:
      sage: import propcalc
      sage: s = propcalc.formula("a^b<->c")
      sage: s.convert_cnf_recur()  #long time
      (a|a)&(b|a)&(a|c)&(b|c)
  ```

- There are a few methods that would be nice for a logic module to have, such as `is_tautology` or `is_contradition`. These would not be hard to implement, just an enumeration over possible inputs until a counterexample is found, or all inputs have been exhastively searched.
- *booleval.py* contains a few very poorly written methods: `eval_ifthen_op` and `eval_iff_off`. The body of eval_ifthen_op, for example, is:

```
if(lval == False and rval == False):
    return True
elif(lval == False and rval == True):
    return True
elif(lval == True and rval == False):
    return False
elif(lval == True and rval == True):
    return True
```

which could be simplified to `(not lval) or rval`

- `BooleanFormula.simplify` would sometimes cause system hangs, even on small numbers of variables. I Was able to establish that this was only when the formula involed `<->, ^, and ->`. It's possible my test code was incorrect, but then one would hope the parser would have caught that, and rejected my attempt at created a BooleanFormula object. It could also be the `BooleanFormula.reduce_op` method, which is responsible for converting all `<->, ^, and ->` into `&, | and ~`.

3 The Quine-McCluskey Algorithm

The Quine-McCluskey Algorithm (also known as the **Prime Implicant** method is guaranteed to find a minimized equivalent form of a boolean expression. Minimized means the smallest number of sum-of-product terms, that is, a series of *and* terms *or*'ed together. This method is often chosen for automation because it is a relatively simple process. The steps are as follows:

1. Iterate over all possible inputs to the boolean expression, and build up a table of the combinations that result in an evaluation to *true*, grouping the table entries by the number of *true*'s they have as in. These combinations are known as *minterms*.

   Note: because each input variable has 2 possible inputs, this means there are a possible 2^n input combinations for n variables.

2. Compare each element of each group with the elements of the neighboring group. If there are any elements in a neighboring group that only differs by a single input, you know you can simplify away that variable, since leaving it out of the term covers both cases (if the input had been T or F). This variable entry is replaced with a "$-$" as a place holder, which is known as a *dont-care*.

3. Iterate over the entire column, putting terms created with don't cares in the next column, until there are no more combinations to be formed.

4. Repeat this process for the newly created column: look for terms that differ by only 1 input (the dont-care's have to be matched too – only one variable of input is allowed to differ), and comtinue to make new columns that include more dont-care terms.

5. Once no more terms can be combined, create a list out of every term that was unable to combine with any other terms (regardless of which column they ended up in). These are known as *prime implicants*. Mark in the table which minterms are covered by which prime implicants.

6. Create a *prime implicant table*, with the prime implicants along the left (the first column), and the minterms along the top.

7. Find the smallest number of implicants that will cover all the minterms - this is done by first figuring out which (if any) prime implicants exclusively provide cover for any minterms. Mark these as necessary for your final solution; these are known as *essential prime implicants*. Then successively try combinations of the remaining prime implicants until the smallest combination is found that covers all of the minterms