CSci 4511 Final Project: A Resolution-Based Propositional Logic Theorem Prover

Sean Bowman

May 4, 2011

1 Introduction

Propositional logic is a simple formal logic system, allowing statements such as $p \implies q$ (p implies q), $a \vee b \iff c$ (a or b if and only if c), etc. While its simplicity and lack of generality largely precludes its sole use in any practical system, there are many important logic systems that use its big brother, first-order logic. First order logic allows such general expressions as $\forall x.Man(x) \implies Mortal(x)$ (for all x, if x is a man, then x is mortal). First-order logic proving systems have had many important applications, such as planning [2], computer program correctness proving [4], and most importantly in formal verification of logic circuits (after the infamous Pentium floating point division bug) [6].

At the core of these systems, though, often lies a simple propositional logic based theorem proving system. By "propositionalizing" the first-order knowledge base to include variables of interest, these systems are able to run a simpler propositional logic theorem proving system on it. I sought to create one of these propositional logic proving systems that is able to prove if any statement of propositional logic is true given past information told to the system.

2 Implementation Details

I chose to implement the propositional knowledge base as a resolution-based proving system, as it was familiar to me and has been shown to be highly effective in the past. I chose to implement all algorithms in Python. The main class of the system is one called KnowledgeBase, which one can interact with as follows:

```
>>> kb = KnowledgeBase()
>>> kb.tell("a => b")
>>> kb.tell("b => c")
>>> kb.ask("a => c")
True
```

All code has been submitted in a .zip file via the CS Submit utility. To use it, simply include "from KnowledgeBase import *" at the top of a python source file and use it as described above; or, run KnowledgeBase.py interactively and use it as described above. The syntax for logical expressions is very forgiving; if you think it will work, it probably will. Example: "a -> b", "a implies ~~b", and "((a=>b))" all work just fine and result in the same thing (~ represents *not*). Associativity and precedence rules are followed appropriately, as described in Russell and Norvig, page 244 [7].

Other methods that may be useful are kb.clear() to empty the knowledge base's set of knowledge, and kb.unlearn("expression") to remove all clauses in an expression from the knowledge base. This may have undesired effects, though, e.g. telling a knowledgebase "(a and b) => c" and "~a or c" and then calling kb.unlearn("a and b => c") will have the effect of emptying the database.

The propositional logic system can be broken down into three distinct parts: (1) A parser to make sense of human input, (2) A logic simplification system to reduce everything to conjunctive normal form, and (3) the resolution system itself, able to deduce truths and falsehoods from a set of logical clauses. I'll now go over each part of the system in detail.

2.1 Parser

Some logical systems require users to input the logic statements as some pre-formatted input that is very native to computer understanding, such as prefix notation, where $a \wedge b \implies c$ would be represented as $implies\left[and\left[a,b\right],c\right]$. Indeed, this makes the first stage of the system much easier to implement, and I used it in my first tries at an implementation. However, I didn't like forcing such an artificial syntax on the user, as unfamiliar syntax and an unnatural feeling interface has been cited as one of the primary impediments to the proliferation of logic verification systems [5]. Instead, I wanted to parse input in as human of a form as possible. To do so, I proceed in three steps.

1. Tokenize the input, so it becomes a list of computer recognizable chunks, or "tokens." To perform this task, I use regular expressions, which allows for a variety of input syntax and a robustness for variations of spacing, etc. For example, the regular expression corresponding to "if and only if" is "\<\-\>|\<\=\>|[iI][fF][fF]", allowing for the user to input <->, <=>, or "iff" in any case as part of a logical expression.

2. Convert the resulting set of tokens to reverse Polish notation, or postfix notation, which allows for easier processing. For example, the tokens $[a, and, b, implies, c]$ (representing $a \wedge b \implies c$) would be converted to $[a, b, and, c, implies]$. To convert the infix notation to postfix notation, I use Dijkstra's shunting yard algorithm [8], which I had to modify to accomodate unary operators (logical negation).

3. Finally, convert the list of tokens in postfix notation order to an actual logical expression. I use a simple stack-based postfix parser to convert

the postfix notation tokens into an "Expression" class that unambiguously captures the logical meaning.

The result of this parsing system is a robust and versatile system that is both intuitive to use and easily extensible for a wide variety of syntactical preferences, correctly parsing sentences from "a or b" to "a and b or c and e <-> f" (resulting prefix notation: $iff\,[or\,[and\,[a,b]\,,and\,[c,e]]\,,f]$).

2.2 Logical Simplifications

Having logical statements unambigiously represented in memory is great, however, a resolution-based system requires a very specific form of logical sentences. To apply the principle of resolution to any logical statements, it is necessary that they be in conjunctive normal form (CNF), i.e. a set of clauses anded together, where a clause is a set of literals or'd together. For example, a sentence in CNF is $(a \vee b) \wedge (c \vee d)$, and one not in CNF is $(a \wedge b) \vee c$. It is possible to convert an arbitrary logical statement to CNF in the following way [7]:

1. Eliminate all biconditionals. Clearly, biconditionals, or iff statements, cannot be present in a CNF sentence. The following equivalence allows all biconditionals to be removed: $a \iff b$ is equivalent to $(a \implies b) \wedge (b \implies a)$.

2. Eliminate all implications. Similar to biconditionals, implications are unacceptable in CNF sentences. The following rule allows implications to be removed: $a \implies b$ is equivalent to $\neg a \vee b$.

3. Require that all "not" operations be applied only to literals. Something such as $(\neg a) \vee b$ is acceptable, whereas $\neg\,(a \vee b)$ is not. This can be accomplished by repeatedly applying DeMorgan's rules, which state that $\neg\,(a \wedge b)$ is equivalent to $\neg a \vee \neg b$, and that $\neg\,(a \vee b)$ is equivalent to $\neg a \wedge \neg b$.

4. We now have a set of (possibly negated) literals linked by *or* and *and* operators in some arbitrary fashion. We can finally convert this to CNF by distributing *or* over *and* wherever possible in the following sense: $a \vee (b \wedge c)$ is equivalent to $(a \vee b) \wedge (a \vee c)$.

2.3 Resolution System

The knowledge base itself is really nothing but a list of clauses stored in set form; for example, after calling

```
kb.tell("a => b")
```

, the knowledge base can be contained entirely in the clause $\neg a \vee b$, which is represented as $\{\neg a, b\}$ (the $\vee$ is implied as all clauses are linked with *or*). Telling the knowledge base $b \implies c$ causes the knowledge base to expand to two clauses, i.e. $[\{\neg a, b\}\,, \{\neg b, c\}]$. Our theorem proving system needs to be able to determine if this set of clauses (call it KB) entails another logical sentence,

say $a \implies c$. To do so, we use a resolution based theorem proving algorithm given in Russell and Norvig [7].

The algorithm relies on the basic principle that if a sentence s is entailed by the knowledge base, the sentence $KB \wedge \neg s$ must be unsatisfiable. The algorithm begins by converting $KB \wedge \neg s$ to CNF form. Then, each clause in $KB \wedge \neg s$ is resolved with every other clause in $KB \wedge \neg s$, and the newly created clauses are added to the set of clauses if not already present. If two clauses resolve to produce the empty clause, then our original set of clauses $KB \wedge \neg s$ must be unsatisfiable. If no new clauses are produced after a round of resolution, then the KB does not entail s. For a proof of the algorithm's correctness and completeness, please see [7].

While this algorithm is complete, meaning it will always determine correctly whether s is entailed by the knowledge base, it is unfortunately very slow. As each clause must be resolved with every other clause over the course of the algorithm, the worst-case runtime of the resolution algorithm is exponential in the number of clauses, resulting in very slow runtimes for even moderately large knowledge bases. It has been proven [1] that it is impossible to perform this task in sub-exponential time no matter which algorithm you use; however, at its core, the resolution procedure is a search problem with branching factor proportional to the number of clauses in our resolving set – if we are better able to choose clauses to resolve, the branching factor would be reduced and, while the runtime would still be exponential, it would be greatly reduced.

A significant such optimization comes from the assumption that the knowledge base contains no internal contradictions. Thus, resolving clauses in KB with other clauses in KB is a fruitless activity, as it will never lead to a contradiction. Instead, we can constrain the choice of clauses to resolve all clauses in $KB \wedge \neg s$ with only those clauses in $\neg s$; this greatly reduces branching factor from $O\left(\|KB \wedge \neg s\|^2\right) = O\left(\|KB\|^2 + 2\|KB\| \cdot \|s\| + \|s\|^2\right)$ to $O\left(\|KB \wedge \neg s\| \cdot \|s\|\right) = O\left(\|KB\| \cdot \|s\| + \|s\|^2\right)$, where $\|S\|$ is the cardinality of S, i.e. the number of clauses in S. As $\|s\|$will almost always be much smaller than $\|KB\|$, this is a significant improvement.

Note that even if our knowledge base *does* contain internal contradictions, the optimized algorithm may still be preferable. If our knowledge base consists of, say, "a and ~a", no matter what we ask it will return "True." It may instead make more sense to only return True if the negated asked statement introduces any *new* contradictions into the knowledge base, which is exactly what the optimized algorithm does. Of course, it is best to never introduce contradictions in the first place; my resolution system can be told expressions with "kb.tell(s, safe=1)" to check that no contradictions are being introduced when telling the kb s.

Test #	1	2	3	4	5
t_{BF} (sec.)	0.001	4.871	4.640	0.378	2.937
t_{OPT} (msec.)	0.494	2.077	2.290	1.963	13.839
t_{BF}/t_{OPT}	2.069	2323.9	1962.314	18.44	194.3

Table 1: Run times of resolution algorithms on different input data sets. t_{BF} refers to the runtime of the brute force resolution algorithm; t_{OPT} is the runtime of the algorithm optimized as described in section 2.3.

3 Results

All algorithms as described above were implemented in Python. Everything was implemented from scratch by me; no third party libraries or other tools were used in the final knowledge base. They were tested under various conditions; some of the interactive sessions are included below. Note: the "ask" function seen below implements the optimized resolution algorithm described above; the "slow_ask" function implements the original brute force resolution algorithm.

```
>>> kb = KnowledgeBase()
>>> kb.tell("a => b")
>>> kb.tell("b => c")
>>> kb.ask("a => c")
True

>>> kb = KnowledgeBase()
>>> kb.tell("~(a <=> c) and (b <=> c) => e")
>>> kb.tell("~e")
>>> kb.ask("a => c")
True
>>> kb.slow_ask("a => c")
True

>>> kb = KnowledgeBase()
>>> kb.tell("a -> b")
>>> kb.tell("(c -> d) iff (a -> b)")
>>> kb.tell("e iff ~(c -> d)")
>>> kb.ask("~e")
True
>>> kb.ask("a")
False
```

I created several such test cases and derived their expected answers by hand to both check the correctness of the algorithms and to compare the speed of the naive resolution with the improved clause selection algorithm. All algorithms found the correct result in all cases, and the timing results are seen in table 1. It can be seen that for larger problems, the optimized version performs significantly better than the unoptimized version.

Even the optimized version of this algorithm, however, is unable to handle even moderately large knowledge bases adequately. To test how it would perform against a large knowledge base, I combined all of the tests together into one large test, telling the knowledge base everything up front and then asking it the conjunction of all the tests' corresponding assertions. This was a terrible mess, though; the very large number of clauses present in the input knowledge base and the larger number of clauses present in the asked sentence cause the state space to explode. I left the test running for over six hours, and it still did not produce an answer.

To accomodate for the larger state space, further optimizations are needed. It's probably possible to segment the search into independently solvable subproblems, similar to the method of tree decomposition used in constraint satisfaction problems [7]. This is a possible direction of future work. Additionally, there are algorithms more efficient than resolution that exploit knowledge between clauses at a level deeper than simple resolution, such as the common DPLL algorithm that is used in most industrial theorem proving systems [3].

References

[1] Cook, S. A. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing* (New York, NY, USA, 1971), STOC '71, ACM, pp. 151–158.

[2] Fikes, R., and Nilsson, N. J. Strips: A new approach to the application of theorem proving to problem solving. In *IJCAI* (1971), pp. 608–620.

[3] Furbach, U., and Obermaier, C. Applications of automated reasoning. In *KI 2006: Advances in Artificial Intelligence*, C. Freksa, M. Kohlhase, and K. Schill, Eds., vol. 4314 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2007, pp. 174–187.

[4] Hoare, C. A. R. An axiomatic basis for computer programming. *Commun. ACM 12* (October 1969), 576–580.

[5] Hoare, C. A. R. An overview of some formal methods for program design. *Computer 20* (September 1987), 85–91.

[6] Miller, S. P., and Srivas, M. Formal verification of the aamp5 microprocessor: a case study in the industrial use of formal methods. In *Proceedings of the 1st Workshop on Industrial-Strength Formal Specification Techniques* (Washington, DC, USA, 1995), WIFT '95, IEEE Computer Society, pp. 2–.

[7] Russell, S. J., and Norvig, P. *Artificial Intelligence - A Modern Approach (3rd ed.)*. Pearson Education, 2010.

[8] Wikipedia. Shunting-yard algorithm — wikipedia, the free encyclopedia, 2011. [Online; accessed 3-May-2011].