Technical Report: Enumerating Theorems

Michael Wehar
University at Buffalo
mwehar@buffalo.edu

May 22, 2015

Abstract

Using a simple unification based algorithm, we explored a technique for enumerating sets of propositional theorems. Then, we tested a small set of seemingly hard theorems on three different theorem provers that were implemented by Christian Gottschall. The results suggest that no one prover is better than the rest which justifies using our larger sets of theorems as an informal benchmark for propositional theorem provers. Finally, we implemented a propositional theorem enumerator in Prolog and compared it with our previous implementation in Java.

1 Prior Work

1.1 The Logic System

Consider a Hilbert style system of propositional calculus with connectives not ($\neg$) and implication ($\rightarrow$) and the inference rule modus ponens. There are many valid axiomatizations of this system. For example, you could use Frege's axioms:

1) $A \rightarrow (B \rightarrow A)$
2) $(A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C))$
3) $(A \rightarrow B) \rightarrow (\neg B \rightarrow \neg A)$
4) $\neg\neg A \rightarrow A$
5) $A \rightarrow \neg\neg A$

Or, you could simply use a single axiom such as Łukasiewicz and Tarski's axiom:

1) $((A \rightarrow (B \rightarrow A)) \rightarrow (((\neg C \rightarrow (D \rightarrow \neg E)) \rightarrow ((C \rightarrow (D \rightarrow F)) \rightarrow ((E \rightarrow D) \rightarrow (E \rightarrow F)))) \rightarrow G)) \rightarrow (H \rightarrow G)$

Just looking at these two axiomatizations of propositional calculus, it's not clear that they are equivalent. However, using automated techniques I was able to show that nearly all of the axiomatizations found in [8] are equivalent.

1.2 Theorem Enumeration

The naive way of enumerating theorems is to enumerate proofs. However, this is very inefficient because most strings of length n don't represent proofs. Even if you had an efficient way of enumerating proofs, it's better to dynamically prove new theorems from the theorems you already have rather than reconstructing new proofs from scratch.

We explore a method using the unification algorithm. Given two schema X and $Y \to Z$, if X matches the subschema Y, then there is a most general theorem you can prove in one step. Find the minimum substitution for X and Y using unification, then apply this substitution to Z. The result is the most general theorem you can get in one step.

Our algorithm keeps a minimal list that represents all theorems you can prove in n steps and then uses the unification approach to generate a minimal list for $n + 1$ steps and so on.

The algorithm was implemented in Java. More detail on prior work and results can be found in [7]. In addition, a related approach was applied to enumerating all Turing machine tape configurations that can be obtained by compressed one-tape Turing machines with at most n states in n steps from a blank tape input.

2 Present Work

2.1 Theorem Provers

While pursuing an independent study, I encountered many proof procedures in [2]. Further, I went on to investigate theorem provers and their performance such as the leanCoP_small prover [5]. The performance results and comparisons led me to run my own performance tests. I made significant adjustments to the theorem enumeration system that I previously developed in Java in order to generate and output a large list of propositional theorems that are sorted by complexity. The adjustments consisted of: (1) Adding randomness into the system to generate theorems of varying complexity, (2) picking an axiomatization that generates theorems with high complexity, and (3) refining parameters to yield the best results.

Randomness was simply added into the system by flipping a weighted coin to determine whether or not we would apply the unification approach to two given theorem schema. We needed to introduce randomness because without it, we would generate so many theorems of low complexity that it would take days before generating theorems of high complexity. Further, it appeared that the Łukasiewicz and Tarski's axiomatization mentioned in the previous section generated theorems of the highest complexity. I measured complexity by the number of variables and the depth of the shortest proof that generated the theorem. The parameters that mattered the most were the weight of the weighted coin and the number of theorems we would generate before halting. For example, during one of the better simulations, the weight was 0.2 and the number of theorems was 50,000.

I ran small tests on three basic online theorem provers by Christian Gottschall [3]. The three provers are called: Advanced Tableau, Basic Tableau, and Benson Mates. In my submission, I included ten seemingly complex instances that I ran the three provers on. The performance was measured in seconds. I found that at least one of the provers could handle nearly all of the instances. However, there were many instances that caused the Advanced Tableau prover to time-out. The results of the small tests suggest that (1) the Advanced Tableau prover is the slowest, (2) the Basic Tableau prover is the most reliable, (3) the Benson Mates can be very fast on some instances and very slow other instances, and (4) no prover always performs better than the others.

2.2 Prolog and Counting Theorems

In addition, we implemented a simplified version of the Java theorem enumerator in Prolog. The Prolog implementation is only a few lines while the Java program is over a thousand lines of code. See the basic implementation below.

```
1) isThm(Axiom, 0).
2) isThm(B,n+1) :- isThm(B,n).
3) isThm(B,n+1) :- isThm(impl(A,B), n), isThm(A, n).
```

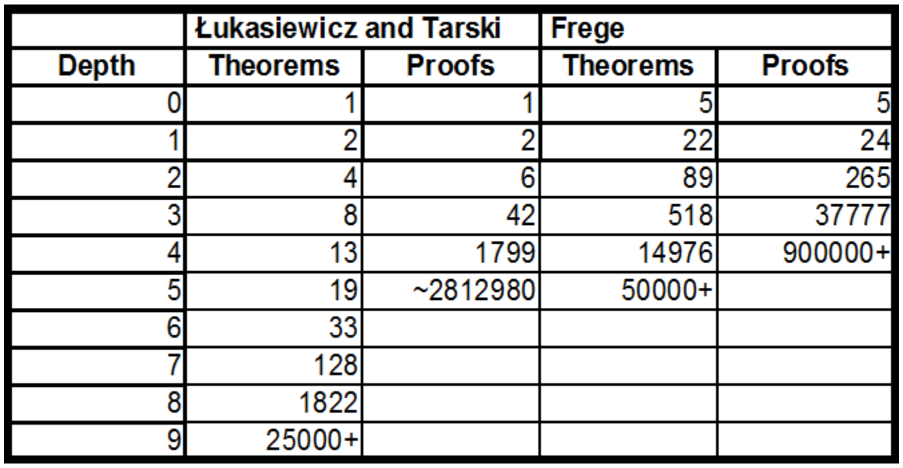
Through some informal testing it appears that the Prolog implementation runs roughly four times as fast as the central unification and enumeration component of the Java implementation. Although the Prolog implementation seems favorable, the basic implementation doesn't condense the list of theorems so it's quite inefficient when you make a query such as `setof(D, isThm(D,7), X)`. The Java implementation efficiently condenses the list of theorems and includes other subtle optimizations that allow one to make such queries yielding results efficiently.

Further, we implemented several more advanced Prolog programs that use tabling to condense the list of theorems. However, these implementations only work in XSB Prolog. For example, we could simply add the following line to the beginning of the basic implementation to enable tabling.

```
0) :- table isThm/2.
```

When we ran this example, it appeared to run more efficiently on queries such as `setof(D, isThm(D,n), X)` for very small n. However, the XSB system runs out of memory when n is larger than 5.

Further, we thoroughly tested all implementations on the Łukasiewicz and Tarski and the Frege axiomatizations that were mentioned in Section 1. We discovered that condensing the list of theorems is essential for yielding meaningful results. For example, the Łukasiewicz and Tarski axiom yields 2.8 million proofs of the 19 theorems of depth 5. This means that the basic Prolog implementation runs out of memory on theorems of depth 5 while the Java implementation can continue generating theorems up to depth 9. See Figure 1 below.



	Łukasiewicz and Tarski		Frege	
Depth	Theorems	Proofs	Theorems	Proofs
0	1	1	5	5
1	2	2	22	24
2	4	6	89	265
3	8	42	518	37777
4	13	1799	14976	900000+
5	19	~2812980	50000+	
6	33			
7	128			
8	1822			
9	25000+			

Figure 1: Counting theorems and proofs of the specified depth.

All implementations in XSB Prolog with tabling ran out of memory on small depths. In particular, our implementations ran out of memory on theorems of depth 6 for Łukasiewicz and Tarski's axiom and theorems of depth 2 on Frege's axiomatization.

3 Future Work

The small tests that I ran were just preliminary runs to get a feel for the sorts of comparisons and runtimes I may expect when I start to run my benchmarks consisting of 50,000+ propositional theorems on high performance theorem provers. There is a nice platform already set-up at [6] to efficiently compare such provers. Further, I recently got in contact with Jeffrey Shallit who (with colleagues) has been successfully applying methods from automata theory to automatedly verify (or even decide) short yet complex statements in number theory [1]. The prover that he uses is called Walnut [4]. I look forward to applying the knowledge that I obtained this semester to future research and collaborations in mathematical logic and theorem proving.

Acknowledgments.

The core algorithm of the propositional theorem enumerator was made while at Carnegie Mellon University for undergraduate research under the advisement of Klaus Sutner. Further collaboration was done with fellow student Joseph Swernofsky. The tests that we discussed in this report were performed at University at Buffalo for an independent study under the advisement of Jan Chomicki.

References

[1] Chen Fei Du, Hamoon Mousavi, Luke Schaeffer, and Jeffrey Shallit. Decision algorithms for fibonacci-automatic words, with applications to pattern avoidance. *CoRR*, abs/1406.0670, 2014.

[2] Melvin Fitting. *First-order Logic and Automated Theorem Proving.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[3] Christian Gottschall. Automated theorem prover for classical predicate logic, 2012. [Online; accessed 12-May-2015].

[4] Hamoon Mousavi. Walnut prover, 2015. [Online; accessed 11-May-2015].

[5] Jens Otten and Wolfgang Bibel. leancop: Lean connection-based theorem proving. *J. Symb. Comput.*, 36(1-2):139–161, July 2003.

[6] Geoff Sutcliffe and Christian Suttner. The tptp problem library for automated theorem proving, 2001. [Online; accessed 12-May-2015].

[7] Michael Wehar. Mathematical logic tools and games, 2014. [Online; accessed 12-May-2015].

[8] Wikipedia. List of logic systems, 2014. [Online; accessed 12-May-2015].