数理解析研究所講究録
第1148巻 2000年 11-16

知的自動証明機の提案と実装

Noriko H. Arai
Ryuji Masukawa
Hiroshima City University
4-3-1 Ozuka-higashi, Asaminami-ku, Hiroshima 731-31 JAPAN
narai@cs.hiroshima-cu.ac.jp,
masukawa@log05.logic.cs.hiroshima-cu.ac.jp

Abstract

In Arai (1996) [1], we introduced a new system for propositional calculus, which gives a natural framework for combinatorial reasoning using "without loss of generality" argument and brute force induction. In this paper, we implement this system, *Simple Combinatorial Reasoning* as a ground theorem prover. We adopt tableau and DLL expressed as sequent calculi for the base systems and implement a symmetry rule on it. We show that our prover successfully finds symmetries in many elementary combinatorial problems, which are known to be exponentially hard for resolution and tableau, and automatically produce polynomial-size proofs. Furthermore, our prover distinguishes those formulas which contain symmetries and those which do not with high possibility without loosing much time. As a result, the performance of our prover on randomly generated formulas is as good as that of existing resolution or tableau provers.

1 Introduction

Since Haken found the first hard example for resolution [13], many others were added to the list of tautologies which require superpolynomially long proofs for resolution and analytic tableau [8]. Actually most of the interesting combinatorial problems were found hard for these proof systems. It was a depressing news for the society of automated theorem proving since many of automated provers adopt either resolution or analytic tableau as their engines. However, it was a quite natural consequence when we ponder how we human being reason. We use different reasoning for different types of problems; algebraic approach to the problems related to counting or linear algebra, combinatorial approach to those related to graphs. If we always take only one approach, which is purely logical analysis in case we adopt resolution and analytic tableau, it is very likely that we end up with exponentially long proofs.

What we suggest in this paper is to give-up "only-one" approach and to adopt different approaches to different types of problems in ground theorem proving. The prover we designed in this paper features two theorem prover. One is a DLL-like sequent calculus and the other is Simple Combinatorial Reasoning. Introduced by Arai (1996), Simple Combinatorial Reasoning is a propositional proof system designed exclusively for combinatorial problems. It features the symmetry rule which allows the exploitation of symmetries present in a problem. It polynomially proves the pigeonhole principle, the mod-k principles, Bondy's theorem, Clique-Coloring problem and many other combinatorial problems; all of them are known to be hard for both resolution and tableau.

Although quite number of researchers share Slaney's opinion: "I consider symmetry to be one of the most important topics of current research in ground theorem proving" [15], not much effort was done to design a theorem prover exploiting symmetries. One reason why people were not so enthusiastic in adopting symmetries in the real prover is that finding symmetries seemed to be as time consuming as exhaustive search anyway. When a formula contains n variables, the most naive program to search for symmetries will check all the permutations on n variables; $n!$ permutations all together. The second reason is that symmetry rule does not seem to make any progress to shorten proofs for randomly generated formulas; the implementation of symmetry rule does not seem to improve the average time complexity. To make the situation worse, it was proved that finding a permutation of the longest orbit in a given formula is NP-complete, and asking two given formulas are symmetric is as hard as the graph isomorphism problem, which is conjectured not in the class P [11]. However, we should not mislead these evidences to conclude that symmetry rules is effective only in theory, but not in practice. These evidences only tell us that we cannot always find the symmetries hidden in formulas, and symmetries will not give us much when we focus on the randomly generated formulas.

In this paper, we set our goal to design a ground theorem prover so that

1. it finds symmetries in a propositional formula as long as human being can find the symmetries in the corresponding first order formula, and
2. it can quickly decide whether symmetry rule is worth trying; it distinguishes formulas with a lot of symmetries from those without them.

Notice that our goal does not contradict to any of the pessimistic evidences.

The symmetry rule can be added to resolution, tableau or sequent calculus. Since Krishnamurthy first pointed out that the symmetry rule is effective to shorten resolution refutations, the researchers had focused on the symmetry rule in resolution [6][7]

[14]. It was Benhamou and Sais who first presented an algorithm how to implement the symmetry rule on resolution [6]. Their strategy was to find a permutation of the largest orbit in the given formula before the machine started resolution procedure. It was pointed out in [11] that finding a permutation of the longest orbit is an NP-complete problem, but Benhamou and Sais allowed machine to backtrack only for fixed amount of time, therefore their algorithm has polynomial-time complexity. They demonstrated their SLDI resolution prover with symmetry rule can automatically produce polynomial-size refutations for the pigeonhole principle. Unfortunately, Benhamou-Sais algorithm did not overwhelm other techniques without the symmetry rule mainly because of the following two reasons.

1. B-S algorithm heavily depends on the form of the input clauses, and it does not work when we disturb its symmetries by throwing in some unnecessary clauses or additional variables.

2. It does not feature any subroutine whether we should run the subroutine to find symmetries; it tries to find symmetries always. [1] Consequently, we end up with poor average time-complexity although it may run dramatically fast for a small class of interesting formulas.

To overcome these deficiencies, we implemented our prover as a sequent-calculus-type backward search prover, called Godzilla in [4]. Godzilla does not try to find symmetries in the input formula, but it finds them while breaking down the formula. Godzilla almost always finds symmetries and produces proofs of size linear to the size of inputs for the pigeonhole principle, the mod-k principle, the clique-coloring problem without increasing the time-complexity much.

However, the performance of original Godzilla turned out to be much poorer than existing DLL provers for randomly generated formulas. One reason is that DLL is theoretically faster than tableau, and another is that Godzilla did not use any heuristic favor for randomly generated 3-CNF formulas. Another criticism against Godzilla was that the performance of Godzilla on the combinatorial formulas seemed to rely on how nicely the input formulas were formulated. In this paper, we adopt both tableau and DLL as the basis for new Godzilla so that we can choose either of them according to the conditions satisfied by the input formula. As a good by-effect, new Godzilla proves some combinatorial problems which old model was not able to produce short proofs. We discuss the detail in section 4. As a result, the performance of Godzilla is improved considerably. We experimented whether or not Godzilla can appropriately find symmetries when we shuffle the input clauses.

This paper is organized as follows. In section 2, we analyze proofs for elementary combinatorial problems. In section 3, we define a deterministic algorithm to simulate elementary combinatorial proofs line by line, and implement it as a theorem prover, Godzilla. In section 4.1, we demonstrate how Godzilla produces proofs for the set of the clauses of size n on n variables, the pigeonhole principle and the clique-coloring problem, which surprisingly resemble to human proofs. In section 4.2, we shuffle the input clauses of the pigeonhole principle and see whether Godzilla can still find symmetries. In section 4.3, we examine the performance of Godzilla on randomly generated formulas. It is a key for the success of Godzilla not to increase time-complexity when it is attacking an tautology having no combinatorial model.

[1]A hard example for B-S algorithm can be found in [5]

2 Simple Combinatorial Proofs

In this section, we informally define what *elementary combinatorial proofs* are, and discuss how to find the symmetries hidden in problems and how to exploit them to obtain short proofs. By analyzing proofs for simple combinatorial problems step by step, we try to extract the reason why these problems are so straightforward for us while they are exponentially hard for many automatic provers.

The pigeonhole principle is one of the most elementary combinatorial principle. The pigeonhole principle states that there is no one-one mapping from the set of $n+1$ objects into the set of n objects. This principle is known to be hard for tableau, resolution and even for bounded depth Frege systems, although the truth of the principle is clear for us. The best thing we can do to prove the principle in resolution is to go over all the possible cases, $n!$ cases all together, that is slightly better than the truth table.

An elementary proof of the pigeonhole principle uses mathematical induction on the number, n, of objects in the domain; we assume that the pigeonhole principle holds for n, and show that it also holds for $n+1$.

(Informal proof of the pigeonhole principle)
Let f be a mapping from $\{0,\ldots,n+2\}$ to $\{0,\ldots,n+1\}$. Without loss of generality, we can assume that $f(n+2)=n+1$. If there exists an $i \neq n+2$ such that $f(i)=n+1$, we are done. Suppose otherwise. Then the function f restricted to $\{0,\ldots,n+1\}$ is a mapping to $\{0,\ldots,n\}$. By the induction hypothesis, it is not one-to-one, and so is not f (q.e.d.).

The novelty of the proof given above is the line, "Without loss of generality ...". Here, we understand that the situation of $f(n+2)=i$ $(i=0,\ldots,n)$ is merely a variant of the situation of $f(n+2)=n+1$; we save time by representing (exponentially) many cases by just one case.

We give another example which has slightly different proof structure. We define $\Pi(n)$ by the set of all clauses of length n in n variables. $\Pi(n)$ is an unsatisfiable set of clauses. D'Agostino proved that this problem is hard for analytic tableau [10]: it requires the proof of size at least $n!$, which is superpolynomial of 2^n. $\Pi(n)$ is informally proved as follows.

(Informal proof of $\Pi(n)$)
Let $p_1,\ldots,p_n$ denote the list of variables appearing in $\Pi(n)$. $\Pi(n)$ is true if and only if both $\Pi(n)|_{p_1=T}$ and $\Pi(n)|_{p_1=\bot}$ are true. However, both of the formulas are equivalent to $\Pi(n\text{-}1)$. By the induction hypothesis, $\Pi(n\text{-}1)$ is true. (q.e.d.)

In this proof, again, we understand that $\Pi(n)$ with the assumption p_1 being true and that with the assumption p_1 being false are isomorphic in structure. Consequently, we represent exponentially many cases by just one case.

The main structure of these proofs is summarized as follows.

1. The statement to be proved is a big disjunction of subcases,
$$\bigvee_{1\le i\le h} A_i$$
where A_i and A_j $(1 \le i,j \le h)$ are isomorphic each other.

2. The formula A_1 is reducible (using pure logic) to an induction hypothesis, or A_1 has a short proof.

We define *elementary combinatorial proofs* by those having the structures satisfying the conditions (1) and (2) given above. Many combinatorial principles are known to have elementary combinatorial proofs; the mod-k principle, the non-unique endnode principle and Bondy's theorem are few examples.

Now we try to simulate elementary combinatorial proofs in the propositional setting. We assume that formulas are expressed as CNF; the input formula is expressed as a set of clauses; $A = C_1 \wedge \cdots \wedge C_n$ and $C_i = l_1^i \vee \cdots \vee l_{m_i}^i$. The first task is to understand the given formula, A, as a big disjunction of subcases.

In the case of the pigeonhole principle, there exists a clause C_i $(1 \le i \le n)$ such that

$$\begin{array}{rcl} A & = & (C_1 \wedge \cdots C_{i-1} \wedge l_1^i \wedge C_{i+1} \wedge \cdots \wedge C_n) \\ & \vee \cdots \vee & (C_1 \wedge \cdots C_{i-1} \wedge l_{m_i}^i \wedge C_{i+1} \wedge \cdots \wedge C_n) \end{array}$$

and each $(C_1 \wedge \cdots C_{i-1} \wedge l_j^i \wedge C_{i+1} \wedge \cdots \wedge C_n)$ is isomorphic to the induction hypothesis.

The proof structure for $\Pi(n)$ is different; there exists a variable p such that A with the assumption p and that with the assumption $\bar{p}$ are both isomorphic to the induction hypothesis.

The first kind of reasoning is most naturally expressed as tableau-like sequent calculus, on the other hand DLL-like sequent calculus is more suitable to express the second kind.

The old prover we designed in [4] to simulate elementary combinatorial proofs was equipped only with tableau. Hence, it failed to find second kind of symmetries discussed above. To overcome this deficiency, we design our prover so that it can choose either tableau or DLL according to the type of the input formula.

3 Theorem prover: Godzilla

3.1 Algorithm

In this subsection, we implement a ground theorem prover, *Godzilla*, to simulate elementary combinatorial proofs discussed in the previous section. The algorithm of Godzilla consists of three parts. The first part is a tableau-like sequent calculus, the second is a DLL-like sequent calculus, and the third part takes care of the restricted permutation rule.

Each of the first and the second part consists of two subparts: *Simplification* and *Branching*. Simplification consists of three subroutines. The first subroutine checks whether or not a given set of clauses contains an axiom. We delete unnecessary clauses as much as possible in the second subroutine. The third subroutine is the unit propagation (unit resolution). During unit propagation, the set of clauses is reordered. Branching divides the given set of clauses to several subsets. Obviously, Branching is the main cause to blow-up the size of proofs.

The third part of the algorithm checks whether or not given two formulas are isomorphic. This procedure is called *Musical-Chair*. It is quite important not to play Musical-Chair when it is hopeless that two given formulas are isomorphic, otherwise the average performance of our prover will be quite poor comparing to the existing Davis-Putnam based theorem prover. For this purpose, we inserted a procedure called *Checker* to examine whether we should try musical-chair or not.

Now we explain the flow of the algorithm. For a technical reason, we describe the algorithm so that the machine produces proofs for several sets of clauses stored in a database. Each set of clauses is expressed as a sequence of clauses, called a *sequent*. Each sequent S is labeled with the number of clauses in S, denoted by $len(S)$, and a sequence of integers $seq(S)$ of length $len(S)$ such that the ith element in $seq(S)$ is the size of the ith clause in S. We call $seq(S)$ the characteristic sequence of S. Suppose that a sequent S is of the form

$$p_1p_2p_3, p_4p_5, \bar{p}_1\bar{p}_4, \bar{p}_3\bar{p}_5.$$

Then, $len(S) = 4$ and $seq(S)$ is $3, 2, 2, 2$.

We first run the subroutine *Simplification* for all the sequents in the database in pararell. Next, we send the database to *Musical-Chair*. In *Musical-Chair* new databases are formed. At last, we send the new databases to *Branching*, and back to *Simplification*.

1. First, check if all the elements in the database are proper sets of clauses.

2. (*Simplification*) Simplification consists of three subroutines.

   (a) (*Subroutine 1*) For each sequent S in the database, check whether or not S contains an axiom: for some literal l, both $\{l\}$ and $\{\bar{l}\}$. If S contains an axiom, write "S is unsatisfiable". If S is an empty sequent, stop the whole procedure immediately and output "(The input is) satisfiable". If not, go to next subroutine

   (b) (*Subroutine 2*) For each S in the database, find a literal l such that there exists a clause containing l but there is none containing $\bar{l}$. Delete all the clauses containing l. If there is no such l, go to the next subroutine.

   (c) (*Unit Propagation*) For each sequent S in the database, find a unit clause $\{l\}$. Delete the clauses containing l. Move all the clauses containing $\bar{l}$ to the head of S, delete the occurrences of $\bar{l}$, and go back to Subroutine 1. Otherwise, this is the end of Simplification, and go to *Checker*.

3. In *Checker*, we divide the given database into disjoint subdatabases.

   (a) (*Checker 1*) First, partition the given database into databases consisting of the same length of sequents. When a database consists of a single sequent, send it to *Branching*.

   (b) (*Checker 2*) Next, partition the given database into databases consisting of sequents so that S and S' are in the same database iff $seq(S) = seq(S')$. When a database consists of a single sequent, send it to *Branching*.

4. (*Musical Chair*) Now we are looking at a database consisting of sequents having the same characteristic sequences. Pick two sequents S_1 and S_2 in the database. (Note that the number of combination is $O(n^2)$.) S_1 and S_2 are expressed as follows.

$$\begin{array}{c} C_1, \ldots, C_n \, (= S_1) \\ D_1, \ldots, D_n \, (= S_2) \end{array}$$

   $size(C_k) = size(D_k)$ for every $1 \leq k \leq n$. Suppose that C_1 is a clause of the form $l_1 \cdots l_m$ and D_1 is of the form $t_1 \cdots t_m$. Without loss of generality, we can assume that C_1 and D_1 are disjoint sets of literals. Define a permutation π by a product of transpositions as follows.

$$\pi = (\ l_1 \quad t_1\ ) \cdots (\ l_m \quad t_m\ )$$

   Extend π so that $\pi(l) = t$ if $\pi(\bar{l}) = \bar{t}$. Rename literals in S_1 according to π. If $\{\pi(C_2), \ldots, \pi(C_n)\} = \{D_2, \ldots, D_n\}$ as sets of clauses, then delete $C_1, \ldots, C_n$ from the database because it is reducible to $D_2, \ldots, D_n$ by using a symmetry rule. Otherwise, move all the clauses containing a literal in $(C_1 - \pi(C_1))$ or its negation to the end of the sequent in S_1. On the other hand, move the clauses which contains a literal in $(\pi(D_1) - D_1)$ or its negation to the end of the sequent in S_2. Send the obtained two sequents back to *Musical-Chair*. If we are still playing on the same two sequents after running this procedure n times, pick different combination of sequents. When we finish checking all the combinations, it is the end of *Musical-Chair*. For each sequent left in the database, form a new database consists of the sequent, and send them to *Branching*.

5. When we receive a database from *Musical-Chair*, it always consists of a single sequent S. *Branching* has two subroutines called *Tableau* and *DLL*. In this paper, we adopt the following condition as our heuristic to decide which we should apply *Tableau* or *DLL*. If there exists a clause C such that no literals in C appears in other clauses in S, go to *Tableau*. Otherwise go to *DLL*.

   (a) (*Tableau*) Without loss og generality, we assume that S is of the form

$$C_1, C_2, \ldots, C_n$$

   where no literals in C_1 appears in other clauses. Delete S from the database, and add new sequents

$$\{l\}, C_2, \ldots, C_n$$

   for each $l \in C_1$. Send the new database back to *Simplification*.

   (b) (*DLL*) Pick a variable p of most occurrences in S. Delete S from the database, and add two new sequents $\{p\} \cup S$ and $\{\bar{p}\} \cup S$, to the database. Send the new database back to *Simplification*.

6. If every sequent in every database is unsatisfiable, output "(The input is) unsatisfiable".

When we input a set of unsatisfiable clauses, $\{C_1, \ldots, C_n\}$, Godzilla produces a elementary combinatorial proof expressed as a directed acyclic graph so that every leaf of P is labeled by an axiom. When we input a set of satisfiable clauses, $\{C_1, \ldots, C_n\}$, Godzilla stops immediately when it finds a satisfying valuation.

Time-Complexity All of the subroutines are accomplished in time $O(n^4)$ where n is the size of the input. Hence, if the size of the obtained proof is polynomially bounded, the time complexity to obtain the proof is also polynomially bounded. Hence, we can assess the efficiency of Godzilla by the size of proofs generated by Godzilla. The obvious upper bound for the size of proof is $k \cdot 2^n$, where n is the number of variables contained in a given formula, and k the size of the formula.

Memory The number of sets of clauses stored in the memory is bounded by (max clause length)$\times$(number of variables). The size of each sets of clauses is bounded by that of the input set.

4 Experimental results

4.1 How Godzilla simulates human reasoning

We first demonstrate how Godzilla acts on combinatorial problems; $\Pi(n)$, the pigeonhole principle, and the clique-coloring problem.

The *clique-coloring problem*, denoted by k-Test(n), states that if a graph contains a k-clique, the graph cannot be properly colored by (k-1) different colors. To express the clique-coloring problem in the propositional calculus, we introduce three types of variables; one to express the clique function, one to express whether there exists a edge between given two vertices, and another to express the coloring function. In this model, how we should permute "coloring variables" is determined by the permutation of "edge variables", which is determined by that of "clique variables"; finding an appropriate permutation for the clique-coloring problem is a lot harder than that for the pigeonhole principle.

Figure1, 2 and 3 shows comparison of performance of Godzilla with and without permutation rule for Π(n), PHP(n) and (n-1)-Test(n) in CPU time. [2]

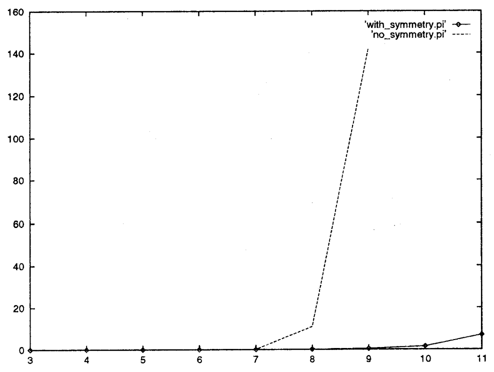

Figure 1: Godzilla w/ vs. w/o symmetries on Π(n)

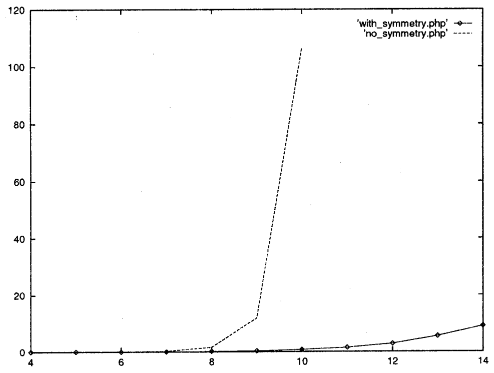

Figure 2: Godzilla w/ vs. w/o symmetries on PHP(n)

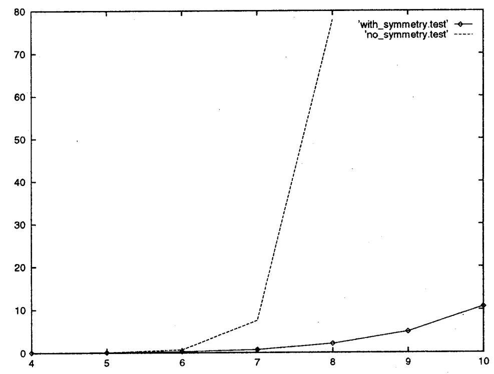

Figure 3: Godzilla w/ vs. w/o symmetries on (n-1)-Test(n)

As mentioned in the previous section, the cost of musical-chair has time complexity $O(n^4)$.

[2]Run times are in seconds and are for C version of Godzilla running on a AMD-K6-2 300MHz processor with 128 MB ram.

Table 1 shows the number of leaves of the proofs generated by Godzilla for these problems.

Table 1: Number of leaves in proofs; Godzilla

	n=3	n=4	n=5	n=6	n=7	n=8	n=9
Π(n)	2	2	2	2	2	2	2
PHP(n)	2	2	2	2	2	2	2
Test(n)	2	2	4	2	4	4	4

Godzilla almost always finds necessary permutations for these elementary combinatorial problems. Furthermore, Godzilla applied DLL for Π(n) and Tableau for PHP(n) and (n-1)-Test(n), which were appropriate decisions. The number of nodes in proofs for Π(n) produced by Godzilla is about square root of that produced by its old model.

4.2 Can Godzilla find symmetries when input clauses are shuffled?

One of the main criticism against Godzilla in [4] was that Godzilla seemed to find symmetries only when the input were formulated nicely. Table 2 shows how the size of proofs increases when we shuffle the order of clauses in the pigeonhole principle. We varied the number of pigeons from 5 to 13. For each n, we ran Godzilla on 20 shuffled PHP(n) and take the average number of leaves in the proofs generated by Godzilla. The result shows that the chance for Godzilla to find symmetries is much worse than the results in the previous subsection. It should be worth noting that even if a prover fails to recognize two formulas are isomorphic only 1 out of 100 cases, the size of proofs may still blow-up exponentially. We need more techniques to improve the ability to find symmetries when the input formula is not formulated nicely. Analyzing the proof produced by Godzilla on a shuffled pigeonhole principle with 7 pigeons, we observed that when the sequents become longer, it is hard for Godzilla to recognize two given sequents are isomorphic after shuffling; it is hard to find symmetries in the beginning of the proof. However, reordering process in the unit propagation helped Godzilla to find the symmetries, and the possibility to find symmetries increases towards the end of the proof.

Table 2: Number of leaves for shuffled PHP(n); Godzilla with vs. without symmetry rule

n	4	5	6	7	8	9
DLL	6	24	120	720	5040	40320
Godzilla	2	4	14	62	353	2278
course-of-values ind.	2	3	4	4	7	9

When we extend Godzilla so that it can simulate course-of-values induction, we obtain better performance in finding symmetries for shuffled formulas or more complicated problems. However, in return, the performance for randomly generated 3-CNF drops severely because of the search-space blow-up.

4.3 How Godzilla acts on randomly generated formulas

It is important for Godzilla being successful as a theorem prover that it does not loose much time when it is attacking a problem which has little hope to contain any symmetries, such as randomly generated formulas. As described in section 3, Godzilla is endowed with a subroutine called *Checker* so that it stops immediately when the given sequent seems to be asymmetric. Our preliminary experiments showed that when randomly generated 3-CNF formulas are broken into several sequents by Branching, only 3 cases out of 1000 passed Checker; Checker seems to be quite effective to detect which sequents contain symmetries and which do not.

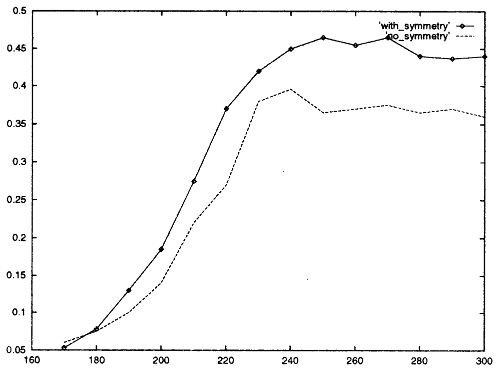

Figure 4: Godzilla w/ vs. w/o symmetries on random 3-CNF

Figure 4 shows a comparison of Godzilla with and without symmetry rule on 50 variable randomly generated 3-CNF in CPU time. We varied the number of clauses from 170 to 300. Godzilla only lost 16% of time by having the symmetry rule. [3]

5 Conclusion

Our theoretical results show that permutation rule (or symmetry rule) has dramatic impact on reducing the lengths of proofs for many combinatorial problems, which are hard for both resolution and tableau. Moreover, our experimental results show that finding symmetries in a given formula is not as hard as it was believed when we adopt the sequent calculus for the base system.

It is a key for the intellectual theorem proving how accurately and how easily the machine can recognize which field of mathematics a given problem falls in. Then, we can apply algebraic technique, for example the cutting planes, for algebraic problems, symmetry rules for elementary combinatorial problems, and common resolution for randomly generated 3CNF's. In this paper, we used naive heuristics to distinguish whether we should apply the symmetry rule or not, that worked quite successfully in the restricted setting. We will need more delicate heuristic functions when we extend our technique to prove problems which have various types of mathematical models.

[3]Godzilla is about 60 times faster than its old model [4] on randomly generated 3CNF's.

References

[1] N.H. Arai, "Tractability of cut-free Gentzen type propositional calculus with permutation inference", *Theoretical Computer Science*, Vol. 170 (1996) 129-144.

[2] N.H. Arai, "Tractability of cut-free Gentzen type propositional calculus with permutation inference II", to appear in *Theoretical Computer Science*.

[3] N.H. Arai, "No feasible monotone interpolation for simple combinatorial reasoning", to appear in *Theoretical Computer Science*.

[4] N.H. Arai and R. Masukawa, "How to find symmetries hidden in combinatorial problems", submitted.

[5] N.H. Arai and A. Urquhart, "Local symmetries in propositional logic", manuscript.

[6] B. Benhamou and L. Sais, "Tractability through symmetries in propositional calculus", *Journal of Automated Reasoning*, Vol. 12 (1994) 89-102.

[7] W. Bibel, "Short proofs of the pigeonhole formulas based on the connection method", *Journal of Automated Reasoning*, Vol. 6 (1990) 287-297.

[8] V. Chvátal and E. Szemerédi, "Many Hard Examples for Resolution",*Journal of the Association for Computing Machinery*, Vol. 35 (1988) 759-768.

[9] J. Crawford, L. Auton, "Experimental results on the crossover point in satisfiability problems", *Proc. of the 11th AAAI* (1993) 21-27.

[10] M. D'Agostino, "Are tableaux an improvement on truth-tables?", *Journal of Logic, Language and Information*, Vol.1 (1992) 235-252.

[11] T. De la Tour and S. Demri, "On the complexity or extending ground resolution with symmetry rules", *Proc. of the 14th IJCAI* (1995) 289-295.

[12] J. H. Gallier, *Logic for Computer Science*, John Wiley & Sons, New York (1987).

[13] A. Haken, "The intractability of resolution", *Theoretical Computer Science*, 39 (1985) 297-308.

[14] B. Krishnamurthy, "Short proofs for tricky formulas, *Acta Informatica*, Vol. 22 (1985) 253-275.

[15] J. Slaney, "The crisis in finite mathematics: automated reasoning as cause and cure", in *CADE-12, Nancy*, ed. A Bundy, Springer Verlag, LNAI 814 (1994) 1-13.