International Journal of Mathematics and Computer Research

ISSN: 2320-7167

Volume 13 Issue 08 August 2025, Page no. – 5520-5524

Index Copernicus ICV: 57.55, Impact Factor: 8.615

DOI: 10.47191/ijmcr/v13i8.03

Bimodality of the Proof Complexity on Boolean Ring

Ruo Ando[1], Yoshiyasu Takeuji[2]

[1]National Institute of Informatics, 2 Chome-1-2 Hitotsubashi, Chiyoda City, Tokyo 101-8430

[2]Musashino University, 3 Chome-3-3 Ariake, Koto City, Tokyo 135-8181

ARTICLE INFO

Published Online: 05 August 2025

Corresponding Author: Ruo Ando

ABSTRACT

In this paper, we present an experiment of complex effects of LRPO on the proof complexity in term orderings on algebraic structures. In experiment, we yield 10! lex patterns that are derived from the symbols that make up Boolean rings (+, −,×, #,∧, 0, 1, a, b, c). Also, the given clause algorithm is used to measure the proof complexity by counting the number of clauses generated. Curiously, the experimental results showed that the distribution the proof complexity was not unimodal, but bimodal. We discuss three possible reasons of bimodality of proof complexity in algebraic structures.

KEYWORDS: Proof Complexity, LRPO, Boolean ring, Given clause algorithm, Bimodality

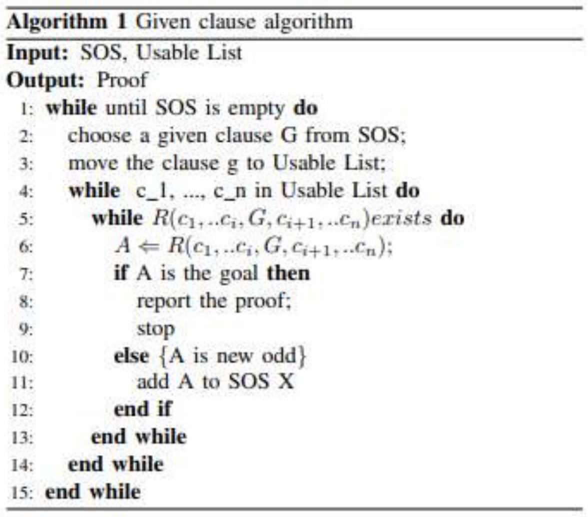
I. INTRODUCTION

In this paper, the complexity of a proof is defined as the number of generated clauses. The generated clauses are calculated using the given clause algorithm. The proof derives the distributive bundle from the rules of the Boolean ring. A Boolean ring and a distributive lattice are closely related mathematical structures. In fact, a Boolean ring always forms a distributive lattice.

```
0 [] a#b*c!= (a#b)* (a#c).
```

OTTER [1] adopts given-clause algorithm in which the program attempts to use any and all combinations from axioms in given clause. In other words, the combinations of clause are generated from given clauses which has been focused on.

Figure 1 shows the procedures of the given clause algorithm. It takes a set of support and a usable list. In line 2, the prover picks up G (given clause) from SoS(Set of Support). Two while loops are started in lines 4 and 5to attempt any and all combinations extracted from the given clause and usable list. Readers are encouraged to check [2] for the basic design of given clause algorithm. In anutshell, five steps are taken in the given clause algorithm asfollows:1) Choose a clause as the given clause from the clauselist of the set of support.2) Append the given clause to the usable list.3) Using the inference rule or rules for the inference ofall clauses.4) Test the retention of newly inferred clause5) Add each yielded new clause to the SoS.OTTER selects a clause G from the clause set which hasbeen focused on in SoS. In this sense, clause G is called agiven clause or focal clause

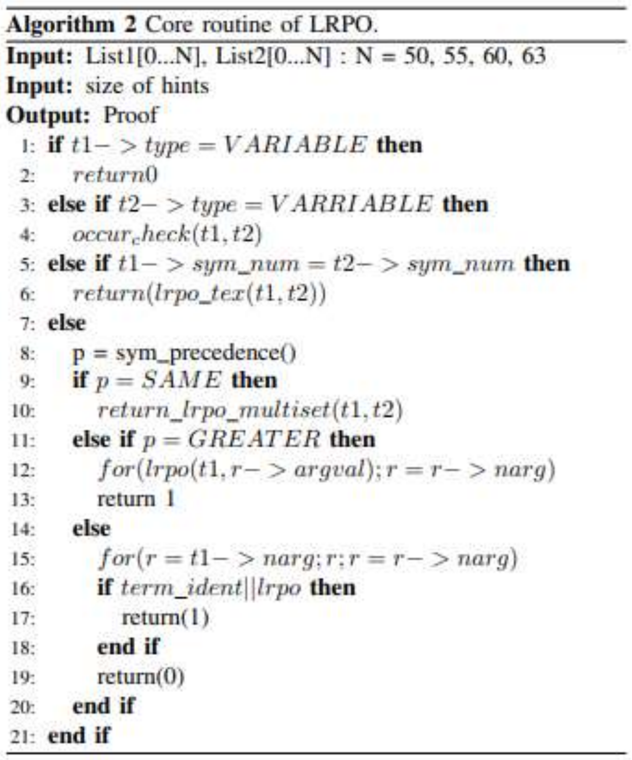


```
Algorithm 1 Given clause algorithm
Input: SOS, Usable List
Output: Proof
 1: while until SOS is empty do
 2:   choose a given clause G from SOS;
 3:   move the clause g to Usable List;
 4:   while c_1, ..., c_n in Usable List do
 5:     while $R(c_1, ..c_i, G, c_{i+1}, ..c_n) exists$ do
 6:       $A \Leftarrow R(c_1, ..c_i, G, c_{i+1}, ..c_n)$;
 7:       if A is the goal then
 8:         report the proof;
 9:         stop
10:       else {A is new odd}
11:         add A to SOS X
12:       end if
13:     end while
14:   end while
15: end while
```

Figure 1. Given clause algorithm

II. LRPO

LRPO [3] is a technique for guaranteeing termination in term rewriting systems. It is necessary to check that the lefthand side of all rewrite rules is greater than the right-hand side, and this judgment is directly linked to the proof of the overall system's terminability. If the number of rewrite rules is k, the depth of terms is d, and the number of arguments is n, the overall computational complexity is O(k n d) at worst. Therefore, in order to use LRPO efficiently, it is necessary to appropriately limit the depth of terms and the number of arguments, and to manage the computational complexity.



```
Algorithm 2 Core routine of LRPO.
Input: List1[0...N], List2[0...N] : N = 50, 55, 60, 63
Input: size of hints
Output: Proof
 1: if $t1->type = VARIABLE$ then
 2:     $return0$
 3: else if $t2->type = VARRIABLE$ then
 4:     $occur_check(t1, t2)$
 5: else if $t1->sym\_num = t2->sym\_num$ then
 6:     $return(lrpo\_tex(t1, t2))$
 7: else
 8:     p = sym_precedence()
 9:     if $p = SAME$ then
10:         $return\_lrpo\_multiset(t1, t2)$
11:     else if $p = GREATER$ then
12:         $for(lrpo(t1, r->argval); r = r->narg)$
13:         return 1
14:     else
15:         $for(r = t1->narg; r; r = r->narg)$
16:         if $term\_ident||lrpo$ then
17:             return(1)
18:         end if
19:         return(0)
20:     end if
21: end if
```

Figure 2. LRPO

Figure2 depicts the algorithm of LRPO. From line 1 to line 4, the process applies when T1 is a variable. In this case, variables are considered smaller than regular terms, so the return value in line 1 is 0. In line 4, if the term being compared is a variable, an occurrence check is performed. In line 5, when T1 and T2 share the same symbol, a function is called to compare them argument by argument. Lines 1 to 6 describe the handling of variables and cases where terms share the same symbol. From line 7 onward, the main comparison process is executed. In line 8, symbols are compared, and in line 9, if the symbols follow the same order, a function for multiset comparison is invoked. Lines 11 to 20 contain the core processing procedure. If P is greater than T, this indicates that T1 is greater than T2, so the function returns 1. In other words, if all arguments of T1 dominate those of T2, the function returns 1. Lines 14 to 20 handle cases where P is not "greater." In line 16, if all arguments of T1 are greater than those of T2, this means that T1 dominates T2, and the function returns 1. Otherwise, if T1 is smaller than T2, the function returns 0.

III. BOOLEAN RING

In mathematics, a Boolean ring [4], R is a ring for which x2 = x for all x in R, that is, a ring that consists of only idempotent elements. An example is the ring of integers modulo 2. Every Boolean ring gives rise to a Boolean algebra, with ring multiplication corresponding to conjunction or meet , and ring addition to exclusive disjunction or symmetric difference (not disjunction, which would constitute a semiring). Conversely, every Boolean algebra gives rise to a Boolean ring. Boolean rings are named after the founder of Boolean algebra, George Boole.

Key Result

Since a + a = 0, each element serves as its own additive inverse. The addition operation in a Boolean ring corresponds to the exclusive OR (XOR) operation in Boolean algebra, denoted as:

$a + b = a \oplus b.$

The multiplication operation corresponds to the logical AND operation, expressed as:

$a \cdot b = a \wedge b.$

If a unit element 1 exists in the ring, it aligns with the standard Boolean algebra identity element.

Example

The ring Z/2Z = {0, 1} serves as a typical example of a Boolean ring, where addition behaves as XOR, and multiplication follows the AND operation. Thus, a Boolean ring connects the principles of Boolean algebra and ring theory, playing a crucial role in logic computation and digital circuit design.

IV. EXPERIMENTS

The proof derives the distributive bundle from the rules of the Boolean ring.

```
0 [] a#b*c!= (a#b)* (a#c).
```

This equation represents a case where the distributive property does not hold. In other words, it indicates that the specific combination of the operations # and ∗ does not satisfy the distributive law.

```
list(usable).
0 [] x=x.
0 [] (x+y)+z=x+y+z.
0 [] 0+x=x.
0 [] (-x)+x=0.
0 [] x+y=y+x.
0 [] (x*y)*z=x*y*z.
0 [] x* (y+z)=x*y+x*z.
0 [] (x+y)*z=x*z+y*z.
end_of_list.
```

Sos list has Idempotent Law and identity law. Usable list has Identity Law, Associative Law, Inverse Law, Commutative Law, Associative Law and Distributive Law.

```
list(sos).
0 [] x*x=x.
0 [] 1*x=x.
0 [] x*1=x.
0 [] x#y=x+y+x*y.
0 [] x^ =1+x.
end_of_list.
```

Figure 3 shows the historam of the number of generated clauses. Regarding Figure 3, it is labeled Proof Complexity. The X-axis represents the amount of calculation calculated by OTTER specifically, it is the number of theories generated by each inference process that finished within 15 seconds for each pattern. And the Y-axis is the number of generated clauses for each. The key point is that this is a bimodal plot, not a unimodal one. To be more specific, the largest number of generated theories is around 120,000, which is roughly 335,000 patterns. The next largest number of generated theories is around 110,000, which is about 25,000 patterns. And the third largest number of generated theories is around 30,000, which is roughly 180,000 patterns. As you can see, the peak in the number of generated theories is divided into two groups, or two peaks, and does not form a single peak. There are two possible reasons why a single peak is not guaranteed here. Firstly, the influence of each sign is not homogeneous, secondly, there are complex dependencies between signs, and thirdly, there is no universal ordering method.

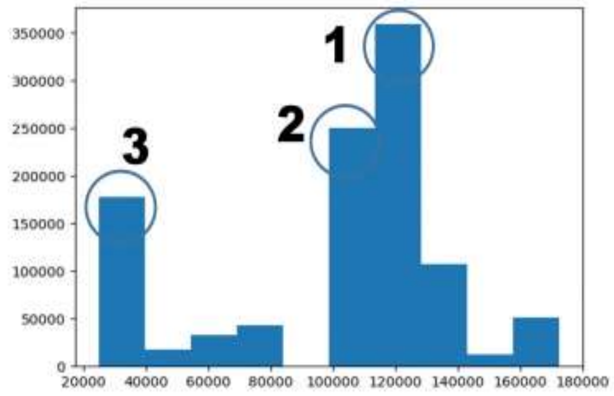


Figure 3. Proof Complexity. Histogram of the number of clauses generated.

The explanation regarding Figure 4 illustrates the relationship between symbols and computational complexity. The horizontal axis represents each part of the symbols, while the vertical axis shows the computational complexity. Overall, the plotted relationship between computational complexity and symbols indicates that the density is not uniformly distributed. Some areas are denser than others, suggesting that denser regions correspond to types of programs that can be computed. Next, when the four symbols and their computational complexities are divided into two parts and categorized into eight blocks (A to H), it becomes clear that the computational complexity for each symbol is not entirely symmetrical. Particularly for Symbol 1, significant asymmetry is observed in regions such as A and D. As a general trend, the first symbol achieves higher computational complexity when installed earlier, whereas the third symbol reduces computational complexity when installed later. However, the overall distribution remains irregular, and this irregularity is likely due to interdependencies between the symbols. Individually, installing the first symbol at the beginning results in higher computational complexity and longer execution times. On the other hand, the third symbol tends to reduce computational complexity when delayed, even though this reduces the number of executable programs. This trade-off explains the uneven histogram observed. In terms of block combinations, the histogram in Figure 1 can be explained using blocks A to H. Specifically, blocks B and C contribute to the largest peak, while the second peak is likely attributed to another combination of blocks. In any case, these eight blocks are neither uniform nor symmetrical, which accounts for the non-uniform distribution of computational complexity.

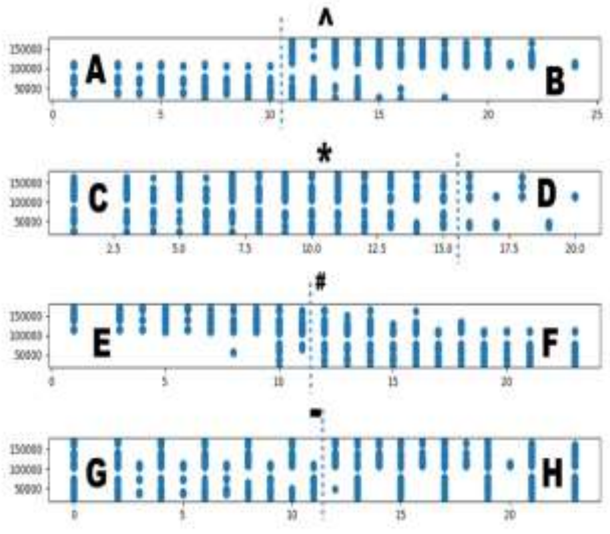


Figure 4. Experimental results of LRPO. X is the position of signs. Y is the number of clauses generated.

The explanation regarding Figure 4 illustrates the relationship between symbols and computational complexity. The horizontal axis represents each part of the symbols, while the vertical axis shows the computational complexity. Overall, the plotted relationship between computational complexity and symbols indicates that the density is not uniformly distributed. Some areas are denser than others, suggesting that denser regions correspond to types of programs that can be computed. Next, when the four symbols and their computational complexities are divided into two parts and categorized into eight blocks (A to H), it becomes clear that the computational complexity for each symbol is not entirely symmetrical. Particularly for Symbol 1, significant asymmetry is observed in regions such as A and D. As a general trend, the first symbol achieves higher computational complexity when installed earlier, whereas the third symbol reduces computational complexity when installed later. However, the overall distribution remains irregular, and this irregularity is likely due to interdependencies between the symbols. Individually, installing the first symbol at the beginning results in higher computational complexity and longer execution times. On the other hand, the third symbol tends to reduce computational complexity when delayed, even though this reduces the number of executable programs. This trade-off explains the uneven histogram observed. In terms of block combinations, the histogram in Figure 1 can be explained using blocks A to H. Specifically, blocks B and C contribute to the largest peak, while the second peak is likely attributed to another combination of blocks. In any case, these eight blocks are neither uniform nor symmetrical,

which accounts for the non-uniform distribution of computational complexity.

V. DISCUSSION

The experimental results of LRPO turned out to be highly complex. Here, we discuss the number of proof clauses, focusing on whether the structure exhibits unimodality or multimodality in this case, bimodality. There are several reasons for this. First, since the placement of symbols interacts with each other, it affects the complexity of the proof. Therefore, rather than the position of symbols independently influencing the proof complexity, the mutual interaction of symbol positions contributes to forming such a complex multimodal structure. Additionally, as is commonly observed, the computational complexity and density of proof complexity are not uniform across symbols. Due to the superposition of distributions with different densities, the structure does not remain unimodal but instead exhibits multimodality, and in this case, bimodality. Consequently, a linear search approach is unlikely to be effective in identifying the optimal lex pattern. In other words, since symbols within the lex pattern interact with each other, a simple linear search is not particularly effective. To determine the optimal symbol placement, that is, the arrangement of symbol positions that minimizes proof complexity, some form of heuristic approach is likely required.

VI. RELATED WORK

These are the manuscript preparation guidelines used as a standard template. Author must follow these instructions and ensure that the manuscript is carefully aligned with these guidelines including headings, figures, tables and references. Manuscripts with poor or no typesetting are not preliminary approved and consider for review. The study presented in this paper is closely related to several research fields. First, the quantification of proof complexity through the number of generated clauses aligns with computational complexity theory and proof theory. Notable related works include Cook’s ”The complexity of theorem-proving procedures” [5] and Buss’s ”Handbook of Proof Theory” [6], which explore foundational aspects of proof complexity. Second, the research on Labeled Recursive Path Ordering (LRPO) and term rewriting systems provides a significant background for the efficient handling of computational processes. Key studies, such as Dershowitz’s ”Termination of rewriting” [7] and Jouannaud and Lescanne’s ”On multiset orderings” [8], offer insights into methods ensuring termination and efficiency in rewriting systems, directly influencing the techniques discussed in this paper. Third, the study leverages properties of Boolean rings and their algebraic structures. Seminal works like Halmos’s ”Boolean algebras” [9] and Huntington’s ”Sets of independent postulates for the algebra of logic” [10] investigate the relationship between Boolean rings and Boolean algebras, providing a mathematical foundation for the algebraic characteristics examined in the paper. Fourth, the use of the given-clause algorithm, as implemented in the OTTER automated theorem-proving system, forms a cornerstone of this study. Foundational works such as McCune’s ”OTTER 3.0 Reference Manual and Guide” [11] and Sutcliffe and Suttner’s ”The CADE ATP System Competition” [12] detail the algorithm’s framework and its application in automated reasoning. Fifth, the analysis of distributions and complexities, particularly bimodal distributions, is rooted in statistical methodologies. Mitzenmacher’s ”A brief history of generative models for power law and lognormal distributions” [13] provides a relevant perspective on the emergence of non-unimodal distributions, aiding the interpretation of the paper’s experimental findings. Finally, research on the optimization of symbol orderings and their impact on algorithm efficiency is directly relevant. Baader and Nipkow’s ”Term Rewriting and All That” [14] discusses the role of symbol orderings in improving computational processes, mirroring the focus of this paper on the complexity associated with arranging ten symbols. These areas collectively establish a robust theoretical and methodological foundation for the paper’s investigation into proof complexity, symbol orderings, and algebraic structures, while also offering directions for further exploration

REFERENCES

1. McCune, W.: 1994b, Otter 3.0 Reference Manual and Guide, Technical Report ANL-94/6 Technical report, Argonne National Laboratory, Argonne, Illinois.
2. William McCune, ”Experiments with Given-Clause Algorithm in Otter,”Journal of Automated Reasoning (JAR), March 1997
3. Gerard Huet, Jean-Marie Hullot, ”Lexicographic Path Orderings and Applications to Term Rewriting Systems,” Journal of Symbolic Computation, June 1982
4. Garrett Birkhoff, ”Boolean Rings and Their Applications,” Transactions of the American Mathematical Society, April 1935
5. Cook, S. A. (1971). ”The complexity of theorem-proving procedures.” Proceedings of the third annual ACM symposium on Theory of computing, pp. 151?158.
6. Buss, S. R. (1998). Handbook of Proof Theory. North-Holland. Dershowitz, N. (1987). ”Termination of rewriting.” Journal of SymbolicComputation, 3(1-2), 69?116.
7. Jouannaud, J.-P., and Lescanne, P. (1982). ”On multiset orderings.” Information Processing Letters, 15(2), 57?63.
8. Dershowitz, N. (1987). ”Termination of rewriting.” Journal of Symbolic
9. Halmos, P. R. (1963). Boolean algebras. Springer.

10. Huntington, E. V. (1904). ”Sets of independent postulates for the algebra of logic.” Transactions of the American Mathematical Society, 5(3), 288-309.
11. McCune, W. (1994). OTTER 3.0 Reference Manual and Guide. Argonne National Laboratory
12. Sutcliffe, G., and Suttner, C. (1998). ”The CADE ATP System Competition.” Automated Deduction?CADE-14, pp. 128?143. Springer.
13. Mitzenmacher, M. (2004). ”A brief history of generative models for power law and lognormal distributions.” Internet Mathematics, 1(2), 226?251.
14. Baader, F., and Nipkow, T. (1998). Term Rewriting and All That.Cambridge University Press.