Why Are There So Many Loop Formulas?

VLADIMIR LIFSCHITZ
University of Texas at Austin
and
ALEXANDER RAZBOROV
Institute for Advanced Study[1]

A theorem by Lin and Zhao shows how to turn any nondisjunctive logic program, understood in accordance with the answer set semantics, into an equivalent set of propositional formulas. The set of formulas generated by this process can be significantly larger than the original program. In this note we show (assuming $P \not\subseteq NC^1/poly$, a conjecture from the theory of computational complexity that is widely believed to be true) that this is inevitable: any equivalent translation from logic programs to propositional formulas involves a significant increase in size.

Categories and Subject Descriptors: F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*Logic and Constraint Programming*; I.2.4 [**Artificial Intelligence**]: Knowledge Representation Formalisms and Methods—*Representation Languages*; F.1.3 [**Computation by Abstract Devices**]: Complexity Measures and Classes—*Reducibility and Completeness*

General Terms: Theory

Additional Key Words and Phrases: Answer sets, loop formulas, P-completeness, stable models

1. INTRODUCTION

A theorem by Lin and Zhao [2002] shows how to turn any nondisjunctive logic program, understood in accordance with the answer set (stable model) semantics [Gelfond and Lifschitz 1988], into an equivalent set of propositional formulas. This set consists of two parts: (i) the completion of the program in the sense of [Clark

[1]on leave from Steklov Mathematical Institute, Moscow, Russia

This submission was handled by the previous Editor-in-Chief.
Vladimir Lifschitz was partially supported by the National Science Foundation under Grant IIS-9732744 and by the Texas Higher Education Coordinating Board under grant 003658-0322-2001. Alexander Razborov was supported by the State of New Jersey, the Bell Companies Fellowship, the James D. Wolfensohn Fund, and the Ellentuck Fund.
Authors' addresses: V. Lifschitz, Department of Computer Sciences, University of Texas, 1 University Station C0500, Austin, TX 78712; e-mail: vl@cs.utexas.edu; A. Razborov, Institute for Advanced Study, School of Mathematics, Einstein Drive, Princeton, NJ 08540; e-mail: razborov@ias.edu.

1978] and (ii) a set of "loop formulas." For instance, the completion of the program

$$\begin{array}{l} p \leftarrow not\ q \\ q \leftarrow not\ p \\ r \leftarrow p \end{array} \tag{1}$$

can be written as

$$p \equiv \neg q,\ q \equiv \neg p,\ r \equiv p. \tag{2}$$

The set of loop formulas of (1) is empty. Consequently, (2) is the Lin/Zhao translation of program (1). The answer sets $\{p, r\}$, $\{q\}$ for program (1) are identical to the satisfying assignments of (2), provided that we identify an assignment with the set of atoms that get the value $\mathbf{t}$. As another example, replace the last rule $r \leftarrow p$ in (1) with $r \leftarrow r$. The Lin/Zhao translation of the modified program consists of its completion

$$p \equiv \neg q,\ q \equiv \neg p,\ r \equiv r$$

and the loop formula $\neg r$.

The completion of any program Π has approximately the same size as Π, but the number of loop formulas of Π can be exponentially larger. The need for so many formulas seems puzzling: the problem of deciding whether a program has an answer set is NP-complete [Marek and Truszczyński 1991], like SAT, and we do not expect a reduction of one NP-complete problem to another to involve exponential growth. A partial explanation of this phenomenon was offered in [Lee and Lifschitz 2003], where the concept of a loop formula is shown to be applicable to disjunctive programs; deciding whether a disjunctive program has an answer set is Σ_2^P-complete [Eiter and Gottlob 1993]. Still, the main question about the number of loop formulas remained unanswered: Is it possible, in the nondisjunctive case, to modify the definition of a loop formula, or perhaps to replace the Lin/Zhao construction by an entirely different translation from logic programs to propositional formulas, so that the translation will be of approximately the same size as the original program, and the answer sets for the program will coincide with the satisfying assignments of its translation?

In this note we show that the answer to this question is no, assuming that polynomial time computations cannot be simulated by short propositional formulas (this is one of the most widely believed assumptions in the theory of computational complexity). The following logic programs Π_n turn out to be "difficult" for translating into the language of propositional formulas:

$$\begin{array}{l} p_{ijk} \leftarrow not\ q_{ijk} \\ q_{ijk} \leftarrow not\ p_{ijk} \\ r_1 \\ r_k \leftarrow r_i, r_j, p_{ijk} \end{array}$$

$(i, j, k \in \{1, \dots, n\})$.

After the review of the necessary background material in the next two sections, the main theorem is stated in Section 4, and a lemma that the theorem easily follows from is proved in Section 5. The discussion of the theorem in Section 6 includes, in particular, a comparison of logic programs with propositional formulas in terms of their expressiveness.

2. BACKGROUND: LOGIC PROGRAMMING

In this note, a *rule* is an expression of the form

$$a \leftarrow b_1, \ldots, b_m, not\ c_1, \ldots, not\ c_n \tag{3}$$

where $a, b_1, \ldots, b_m, c_1, \ldots, c_n$ are (propositional) atoms, and m, n are nonnegative integers. (The symbol *not* is called negation as failure.) If $m = n = 0$ then the arrow after a can be dropped. A *(logic) program* is a finite set of rules.

We define when a set X of atoms is an answer set for a program Π, as follows. Remove from Π all rules (3) such that at least one of the atoms $c_1, \ldots, c_n$ belongs to X. In each of the remaining rules (3), remove the part $not\ c_1, \ldots, not\ c_n$. The result is a set of rules of the form

$$a \leftarrow b_1, \ldots, b_m. \tag{4}$$

Consider the smallest set Y of atoms that is closed under each of these rules (which means that if $b_1, \ldots, b_m \in Y$ then $a \in Y$). If $Y = X$ then X is an *answer set* for Π.

Let us check, for instance, that $\{p, r\}$ is an answer set for program (1). We remove from the program the second rule, because p belongs to $\{p, r\}$, and then remove $not\ q$ from the first rule. The resulting set of rules (4) is

$$\begin{array}{l} p \\ r \leftarrow p, \end{array}$$

and the smallest set of atoms closed under these rules is indeed the set $\{p, r\}$ that we started with.

Semantically, logic programs as defined above are similar to propositional formulas: the meaning of a program is given by the collection of its answer sets, which is a set of sets of atoms, that is to say, a truth table. On the other hand, the semantics of propositional formulas is monotonic, in the sense that if T_1 and T_2 are sets of propositional formulas and T_1 is a subset of T_2 then the set of models of T_1 is a superset of the set of models of T_2. For logic programs, the corresponding condition does not hold. Consider, for instance, the logic programs

$$p \leftarrow not\ q$$

and

$$\begin{array}{l} p \leftarrow not\ q \\ q. \end{array}$$

The only answer set for the former is $\{p\}$; the only answer set for the latter is $\{q\}$. The importance of nonmonotonicity for representing declarative knowledge was first recognized around 1980. The logic programming language defined above is, in fact, a subset of one of the nonmonotonic formalisms proposed at that time—default logic [Reiter 1980].

Software systems that compute answer sets for a given logic program, such as SMODELS[2] and DLV[3], are called answer set solvers. Such systems have been successfully used for solving combinatorial search problems; this is called answer set

[2] `http://www.tcs.hut.fi/Software/smodels/` .
[3] `http://www.dbai.tuwien.ac.at/proj/dlv/` .

programming. The possibility of transforming a logic program into an equivalent set of propositional formulas, mentioned in the introduction, has been used for creating answer set solvers ASSAT[4] and CMODELS.[5]

3. BACKGROUND: COMPLEXITY

For a systematic treatment of the complexity hierarchy below the class P we refer the reader to the excellent monograph [Greenlaw et al. 1995]; here we review only a few basic definitions and some notation necessary to formulate and understand our results.

P is the class of all languages decidable by polynomial time Turing machines. NC^1 is the class of languages recognizable by a uniform[6] family of Boolean circuits $\{C_n : n = 1, 2, \ldots\}$ of logarithmic depth $O(\log n)$ (the circuit C_n has n inputs and must correctly answer the queries $x \overset{?}{\in} L$ for all binary words x of length n). Similarly, FNC^1 is the class of all *functions* $f : \{0,1\}^* \to \{0,1\}^*$ computable by (many output) uniform families of logarithmic depth circuits.

$NC^1/poly$ is the analogue of the class NC^1 in which no uniformity conditions on the family $\{C_n\}$ are imposed. A classical result in circuit complexity [Spira 1971] states that $NC^1/poly$ is equal to the class of languages decidable by polynomial *size* propositional *formulas*. This class of languages is uncountable, so that "relatively few" languages from $NC^1/poly$ belong to P (or are recursive, for that matter). On the other hand, most complexity theorists firmly believe that P is not a subset of $NC^1/poly$ either, and finding a rigorous proof of this conjecture constitutes an open problem in Theoretical Computer Science, second in importance only to the $P \overset{?}{=} NP$ question.

A language L is NC^1 *many-one reducible* to another language L' if there is an FNC^1 function $f : \{0,1\}^* \to \{0,1\}^*$ such that $x \in L$ if and only if $f(x) \in L'$. This notion of reducibility is transitive. A language $L \in P$ is *P-complete* if every $L' \in P$ is NC^1 many-one reducible to L. P-complete languages have the property that if any one of them can be recognized by propositional formulas of size $S(n)$, then *every* language in P can be recognized by propositional formulas of size $S(n)^{O(1)}$. In particular, if at least one P-complete language were in $NC^1/poly$ then P would be a subset of $NC^1/poly$, contrary to the conjecture mentioned above.

We need the following (canonical) example of a P-complete language from [Cook 1974].[7] Let V be a finite set of *vertices*, $S \subseteq V$ and $R \subseteq V \times V \times V$. The vertices *admissible from S in R* are the least set $A \subseteq V$ such that $S \subseteq A$ and such that if $v_1, v_2 \in A$ and $R(v_1, v_2, v)$, then $v \in A$. Intuitively, the triples $\langle v_1, v_2, v\rangle$ in R can be thought of as "two-premise rules" allowing us to derive v from v_1 and v_2, and then A is the set of vertices that can be generated by these rules from the elements of S.

[4]`http://assat.cs.ust.hk/` .
[5]`http://www.cs.utexas.edu/users/tag/cmodels/` .
[6]A family of Boolean circuits $\{C_n : n = 1, 2, \ldots\}$ is called (log-space) *uniform* if there exists a logarithmic space Turing machine computing from 1^n the description of C_n.
[7]See also [Greenlaw et al. 1995, Problem A.6.8]. Here we slightly change the notation (in particular, we switch the roles of S and T).

Path Systems (PATH)
Given: A *path system* (V, R, S, T), where V is a finite set, $S \subseteq V$, $T \subseteq V$, and $R \subseteq V \times V \times V$.
Problem: Is there a vertex in T that is admissible from S in R?

Every problem (= language) in P to which PATH is NC^1 many-one reducible is also P-complete.

4. STATEMENT OF RESULTS

In connection with the sequence Π_n of logic programs defined in the introduction, we consider the following algorithmic problem.

Answer Set for Π_n (AS-Π_n)
Given: An integer n and a subset X of the atoms

$$\{r_i : i \in \{1, \dots, n\}\} \cup \{p_{ijk}, q_{ijk} : i, j, k \in \{1, \dots, n\}\}.$$

Problem: Is X an answer set for Π_n?

LEMMA 1. *The problem AS-Π_n is P-complete.*

This fact is proved in the next section.

THEOREM 2. *If $P \not\subseteq NC^1/poly$ (that is, not all languages in P can be recognized by polynomial size propositional formulas), then it is impossible to find a sequence of propositional formulas $F_1, F_2, \dots$ such that*

(i) for every n, the satisfying assignments of F_n are identical to the answer sets for Π_n, and
(ii) the sizes of the formulas F_n are bounded from above by a polynomial in n.

PROOF. Assume that a sequence of formulas with these properties exists. Then, by Spira's theorem quoted in Section 3, AS-Π_n is in $NC^1/poly$. By Lemma 1, it follows that $P \subseteq NC^1/poly$, contrary to the assumption from the statement of the theorem. □

If we adopt here the stronger assumption that P cannot be recognized by any *sub-exponential* $(\exp(n^{o(1)}))$ size propositional formulas, also widely believed by complexity theorists, then we can respectively relax the requirement (ii) in this theorem. In either form, this result strongly indicates that whenever we try to translate programs Π to propositional formulas $F(\Pi)$, (at least) one of the following two will inevitably happen:

(1) the correspondence between answer sets for the program Π and satisfying assignments of its propositional translation $F(\Pi)$ will not be given by the identity mapping (and, more generally, by any NC^1 many-one mapping);
(2) the size of $F(\Pi)$ will be significantly larger than the size of Π.

From the lemma we can conclude that the problem of recognizing answer sets for arbitrary logic programs is P-complete as well, because programs Π_n are easy to construct by uniform logarithmic depth circuits. This corollary has an easier proof that does not even require the use of negation as failure in programs.[8] The assertion of the theorem, however, would be incorrect if we replaced the programs Π_n by any sequence of programs without negation as failure. Indeed, a program that does not contain negation as failure has a unique answer set, and consequently can be represented in propositional logic by a conjunction of literals.

5. PROOF OF THE LEMMA

Clearly, AS-Π_n is in P. It is sufficient to construct an NC^1 many-one reduction from PATH to AS-Π_n, and we break the construction into two pieces using the following minor modification of PATH:

Propagation in Path Systems (PathProp)
Given: (V, R, s), where V is a finite set, $R \subseteq V \times V \times V$, and $s \in V$.
Problem: Are all vertices in V admissible from $\{s\}$ in R?

Reduction from PATH to PathProp (cf. the proof of the NL-completeness of STRONG (DIRECTED) CONNECTIVITY by Laaser presented in [Jones 1975]). Let $P = (V, R, S, T)$ be an instance of PATH. If T is empty then P is not in the language PATH, and we translate it into an arbitrary fixed negative instance of PathProp. In the case $T \neq \emptyset$, we translate P into an instance $P' = (V', R', s)$ of PathProp as follows. Let $V' = V \dot\cup \{s\}$, where s is a new vertex not appearing in V. Let

$$R' = R \dot\cup \{\langle s, s, s'\rangle : s' \in S\} \dot\cup \{\langle t, t, v\rangle : t \in T, v \in V\}.$$

It is clear that this transformation is in NC^1.

Let A be the set of vertices in V admissible from S in R. If $A \cap T = \emptyset$, then $A \cup \{s\}$ is closed under the set of rules R' (and thus no vertex in T is admissible from $\{s\}$ in R'), and if $t \in A \cap T$ then we use the last rule $\langle t, t, v\rangle$ to see that all vertices in V' become admissible. Thus, our transformation indeed establishes the desired reducibility.

Reduction from PathProp to AS-Π_n. Let $P = (V, R, s)$ be an instance of PathProp. We may assume w.l.o.g. that $V = \{v_1, \ldots, v_n\}$ and $v_1 = s$. We translate P into the following instance $X(P)$ of AS-Π_n:

$$X(P) = \{p_{ijk} : R(v_i, v_j, v_k)\} \cup \{q_{ijk} : \neg R(v_i, v_j, v_k)\} \cup \{r_i : 1 \leq i \leq n\}.$$

Clearly, this translation is in NC^1, and it is easy to check that $X(P)$ is an answer set for Π_n if and only if all vertices in V are admissible from $\{v_1\}$. Thus, our translation is an NC^1 many-one reduction from PathProp to AS-Π_n.

Lemma 1 is proved by combining the two reductions above.

6. DISCUSSION

The need to use long formulas when logic programs are translated into propositional logic can be established, under plausible assumptions, for some programs that look

[8] We are grateful to Hudson Turner for this observation.

even simpler than Π_n. Consider, for instance, the logic programming definition of reachability in arbitrary directed graphs:

$$\begin{array}{l} p_{ij} \leftarrow not\ q_{ij} \\ q_{ij} \leftarrow not\ p_{ij} \\ r_1 \\ r_j \leftarrow r_i, p_{ij} \end{array}$$

$(i, j \in \{1, \ldots, n\})$. For these programs, we can prove an assertion similar to Theorem 1, with the assumption $P \not\subseteq NC^1/poly$ replaced by the stronger, but still very plausible conjecture that some languages recognizable by nondeterministic log-space Turing machines are not recognizable by polynomial size formulas.

We should emphasize that these results are about propositional formulas *equivalent* to the given logic programs, but not about formulas that *extend them conservatively*. Any logic program Π can be converted, in polynomial time, into a propositional formula F that may contain "new atoms" so that the answer sets for Π can be obtained by removing these atoms from the models of F (an economical way of doing this is described in [Lin and Zhao 2003]).

The problem discussed in this note exemplifies the "comparative linguistics" approach to knowledge representation formalisms as described in [Gogic et al. 1995]:

> *How succinctly can the formalism express the set of models that it can?* ...[W]e consider formalism A to be stronger than formalism B if and only if any knowledge base in B *has an equivalent knowledge base in* A *that is only polynomially longer, while there is a knowledge base in* A *that can be translated to* B *only with an exponential blowup.*

Let us compare, in the sense of this criterion, the language of logic programs under the answer set semantics with the language of propositional formulas. Answer sets for the class of programs defined in Section 2 have the anti-chain property: an answer set cannot contain another answer set for the same program. Consequently, many propositional formulas do not have equivalent logic programs. The situation changes, however, as soon as we make the syntax of logic programs a little more general and allow, for instance, negation as failure in the heads of rules [Inoue and Sakama 1998], or nested expressions [Lifschitz et al. 1999], or choice rules [Niemelä et al. 2002]. In the language of logic programs with nested expressions, any propositional formula F can be represented by the program consisting of the rules

$$p \leftarrow not\ not\ p$$

for all atoms p occurring in F and the constraint

$$\leftarrow not\ F$$

(with the negations in F written as *not*). The discussion in Section 4 shows, on the other hand, that (under some plausible assumptions) there are logic programs that can be translated into propositional logic only with an exponential blowup.

The strength, in the sense of [Gogic et al. 1995], of the language of logic programs is its second advantage over the language of propositional formulas as a knowledge representation formalism, besides the fact that it is nonmonotonic.

ACKNOWLEDGMENTS

We would like to thank Selim Erdoğan, Paolo Ferraris, Michael Gelfond, Georg Gottlob, Joohyung Lee, Miroslaw Truszczyński and Hudson Turner for comments and discussions.

REFERENCES

Clark, K. 1978. Negation as failure. In *Logic and Data Bases*, H. Gallaire and J. Minker, Eds. Plenum Press, New York, 293–322.

Cook, S. A. 1974. An observation on time-storage trade-off. *Journal of Computer and System Sciences 9,* 3, 308–316.

Eiter, T. and Gottlob, G. 1993. Complexity results for disjunctive logic programming and application to nonmonotonic logics. In *Proceedings of International Logic Programming Symposium (ILPS)*, D. Miller, Ed. 266–278.

Gelfond, M. and Lifschitz, V. 1988. The stable model semantics for logic programming. In *Proceedings of International Logic Programming Conference and Symposium*, R. Kowalski and K. Bowen, Eds. 1070–1080.

Gogic, G., Kautz, H., Papadimitriou, C., and Selman, B. 1995. The comparative linguistics of knowledge representation. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*. 862–869.

Greenlaw, R., Hoover, H. J., and Ruzzo, W. L. 1995. *Limits to parallel computation: P-completeness theory.* Oxford University Press.

Inoue, K. and Sakama, C. 1998. Negation as failure in the head. *Journal of Logic Programming 35*, 39–78.

Jones, N. D. 1975. Space-bounded reducibility among combinatorial problems. *Journal of Computer and System Sciences 11*, 68–85.

Lee, J. and Lifschitz, V. 2003. Loop formulas for disjunctive logic programs. In *Proceedings of International Conference on Logic Programming (ICLP)*. 451–465.

Lifschitz, V., Tang, L. R., and Turner, H. 1999. Nested expressions in logic programs. *Annals of Mathematics and Artificial Intelligence 25*, 369–389.

Lin, F. and Zhao, J. 2003. On tight logic programs and yet another translation from normal logic programs and propositional logic. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*. 853–864.

Lin, F. and Zhao, Y. 2002. ASSAT: Computing answer sets of a logic program by SAT solvers. In *Proceedings of National Conference on Artificial Intelligence (AAAI)*. 112–117.

Marek, V. and Truszczyński, M. 1991. Autoepistemic logic. *Journal of the ACM 38*, 588–619.

Niemelä, I., Simons, P., and Soininen, T. 2002. Extending and implementing the stable model semantics. *Artificial Intelligence 138*, 181–234.

Reiter, R. 1980. A logic for default reasoning. *Artificial Intelligence 13*, 81–132.

Spira, P. M. 1971. On time-hardware complexity tradeoffs for Boolean functions. In *Proceedings of the 4th Hawaii Symposium on System Sciences.* Western Periodicals Company, North Hollywood, 525–527.

Received February 2004; accepted April 2004