SELP - A System for Studying Strong Equivalence between Logic Programs

Yin Chen[1,2], Fangzhen Lin[3] and Lei Li[2]

[1] Department of Computer Science, South China Normal University, China
[2] Software Institute, Sun Yat-sen University, China
[3] Department of Computer Science, Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong

Abstract. This paper describes a system called **SELP** for studying strong equivalence in answer set logic programming. The basic function of the system is to check if two given ground disjunctive logic programs are equivalent, and if not, return a counter-example. This allows us to investigate some interesting properties of strong equivalence, such as a complete characterization for a rule to be strongly equivalent to another one, and checking whether a given set of rules is strongly equivalent to another, perhaps simpler set of rules.

1 Introduction

The notion of strongly equivalent logic programs was proposed by [5]. It has been found useful for tasks such as program simplification (e.g. [2]). In this paper, we describe a system called **SELP** that can help us answer questions regarding this notion, from simple ones such as "are P and Q strongly equivalent" to more involved ones such as "exactly what kind of rules are strongly equivalent to the empty set".

The core of the system is checking whether two disjunctive logic programs are strongly equivalent. This is based on [6], which provides a simple mapping from logic programs to propositional theories that reduces strong equivalence to entailment in classical propositional logic. Thus, the problem of strong equivalence checking can be translated into a satisfiability problem in propositional logic, and solved using SAT solvers like zChaff [10].

In addition, when two programs are not strongly equivalent, we may want to find some witnesses. For example, $P_1 = \{(a_1 \leftarrow a_2), (a_1 \leftarrow not\, a_2)\}$ and $P_2 = \{a_1 \leftarrow\}$ are not strongly equivalent, and $P = \{a_2 \leftarrow a_1\}$ is a witness (counter-example): it is easy to see that $\{a_1, a_2\}$ is an answer set of $P_2 \cup P$, but not of $P_1 \cup P$. It is often hard to find a witness manually, but **SELP** can do this automatically: when two programs P_1 and P_2 are found not to be strongly equivalent, it will return a program P, such that $P_1 \cup P$ and $P_2 \cup P$ have different answer sets.

The most interesting part of **SELP** is that it allows us to study some properties of the notion of strong equivalence. In [7], we described some results on classes of strongly equivalent logic programs discovered using the system. In this paper, we shall show how the system can help us answer questions of the following form: Given a set P of rules, is there another set of rules of certain property that is strongly equivalent to P?

In [4], a system called LPEQ was developed that can check if two normal programs are strongly equivalent, and was implemented using the answer set logic programming system *smodels*. Besides being implemented using a different technique, our system can deal with normal as well as disjunctive logic programs. Furthermore, our system can construct a counter-example when two programs are not strongly equivalent.

The remainder of this paper is organized as follows. In the next section, we review some key concepts and notations in answer set logic programming. In section 3, we describe the core of the system, i.e. how to check if two programs are strongly equivalent. In section 4, we describe how to discover classes of strongly equivalent programs using **SELP**. In section 5, we show how to find all programs that are strongly equivalent to a given one. Finally, we conclude the paper in section 6.

2 Preliminaries

Let L be a propositional language, i.e. a set of atoms. In this paper we shall consider logic programs with rules of the following form:

$$h_1; \cdots; h_k \leftarrow p_1, \cdots, p_m, not\, p_{m+1}, \cdots, not\, p_n \tag{1}$$

where h_i's and p_i's are atoms in L. So a logic program here can have default negation (not), constraints (when $k = 0$), and disjunctions in the head of its rules. In the following, if r is a rule of the above form, we write Hd_r to denote the set $\{h_1, ..., h_k\}$, Ps_r the set $\{p_1, ..., p_m\}$, and Ng_r the set $\{p_{m+1}, ..., p_n\}$. Thus a rule r can also be written as $Hd_r \leftarrow Ps_r, not\, Ng_r$.

The semantics of these programs are given by answer sets as defined in [3]. As conventional in logic programming, we identify interpretations of L with sets of atoms in L. Let I be a set of atoms, and P a logic program. We say that I is closed under P if for any rule r in P, we have that $Hd_r \cap I \neq \emptyset$ whenever $Ps_r \subseteq I$ and $Ng_r \cap I = \emptyset$. Now if P is a program without negation, a set I of atoms is an answer set of P iff I is a minimal set of atoms that is closed under P. Generally, I is an answer set of P iff I is an answer set of P^I, where P^I, the reduct of P on I, is obtained from P as follows: for any rule of form (1), if there is an atom p_i, $m + 1 \leq i \leq n$, such that $p_i \in I$, then delete this rule; otherwise, delete all the literals of the form $not\, q_i$ from this rule.

Two logic programs P_1 and P_2 in L are said to be *equivalent* if they have the same answer sets, and *strongly equivalent* [5] (in the language L) if for any logic program P in L, $P \cup P_1$ and $P \cup P_2$ are equivalent. For instance, $\{p \leftarrow q\}$ and $\{(p \leftarrow q), (q \leftarrow p)\}$ are equivalent but not strongly equivalent: they both have the unique answer set $\emptyset$, but when we add the rule $p \leftarrow$ to them, the first one will have the answer set $\{p\}$, while the latter $\{p, q\}$.

The notion of strongly equivalent logic programs is interesting for a variety of reasons. For instance, as Lifschitz *et al.* [5] noted, if two sets of rules are strongly equivalent, then one can be replaced by the other in any logic program regardless of the context. Thus knowing whether two sets of rules are strongly equivalent is a useful exercise that may have applications in program simplification.

In the following, for convenience, when we say a rule is strongly equivalent to the empty set, we mean the set that contains exactly this rule is strongly equivalent to the

empty set. Similarly, when we say two rules are strongly equivalent, we mean that the two sets of rules, each consisting of exactly one of the rules, are strongly equivalent.

3 Checking strong equivalence between two logic programs

Lifschitz, Pearce, and Valverde [5] showed that checking for strong equivalence between two logic programs can be done in the logic of here-and-there, a three-valued non-classical logic somewhere between classical logic and intuitionistic logic. Turner [11] provided a model-theoretic characterization of strong equivalence in terms of pairs of sets of atoms. Lin [6] provided a mapping from logic programs to propositional theories and showed that two logic programs are strongly equivalent iff their corresponding theories in propositional logic are equivalent. This result will be the basis that we are using in this paper for checking if two logic programs are strongly equivalent, and we repeat it here.

Let P_1 and P_2 be two finite logic programs, and L the set of atoms in them.

Theorem 1. *[6] P_1 and P_2 are strongly equivalent iff in the propositional logic, the following two entailments hold:*

$$\{p \supset p' | p \in L\} \cup \Delta(P_1) \models \Delta(P_2), \tag{2}$$
$$\{p \supset p' | p \in L\} \cup \Delta(P_2) \models \Delta(P_1). \tag{3}$$

where for each $p \in L$, p' is a new atom, and for each program P, $\Delta(P) = \{\Delta(r) \mid r \in P\}$, where for each rule r of the form (1), $\Delta(r)$ is the conjunction of the following two sentences:

$$p_1 \wedge \cdots \wedge p_m \wedge \neg p'_{m+1} \wedge \cdots \wedge \neg p'_n \supset h_1 \vee \cdots \vee h_k, \tag{4}$$
$$p'_1 \wedge \cdots \wedge p'_m \wedge \neg p'_{m+1} \wedge \cdots \wedge \neg p'_n \supset h'_1 \vee \cdots \vee h'_k. \tag{5}$$

Notice that if $m = n = 0$, then the left sides of the implications in (4) and (5) are considered to be $true$, and if $k = 0$, then the right sides of the implications in (4) and (5) are considered to be $false$.

Theorem 1 makes it possible to check the strong equivalence between two logic programs using a SAT solver. For instance, to verify (2), it is sufficient to check that both of the following two formulas are satisfied for all $r \in P_2$:

$$\{p \supset p' \mid p \in L\} \cup \Delta(P_1) \cup \{p \mid p \in Ps_r\} \cup \{\neg p' \mid p \in Ng_r\} \cup \{\neg p \mid p \in Hd_r\},$$
$$\{p \supset p' \mid p \in L\} \cup \Delta(P_1) \cup \{p' \mid p \in Ps_r\} \cup \{\neg p' \mid p \in Ng_r \cup Hd_r\}.$$

Algorithm 1 makes precise this idea, and was implemented using the SAT solver zChaff [10].

If P_1 and P_2 are not strongly equivalent, then zChaff will return an assignment that is a counter-example to either (2) or (3), and from this assignment, we can construct a program P such that $P \cup P_1$ and $P \cup P_2$ are not equivalent, i.e. P is a witness of the fact that P_1 and P_2 are not strongly equivalent.

1: **procedure** SELP(LP P_1, LP P_2)
2: $F = \emptyset$ ▷ the formula to be checked
3: **for all** atom p in $\mathcal{L}$ **do**
4: $F = F \cup \{\neg p \vee p'\}$
5: **end for**

6: $\Delta(P_1) = \emptyset$
7: **for all** rule $r : h_1; \ldots; h_k \leftarrow p_1, \ldots, p_m, p_{m+1} \ldots, p_n$ in P_1 **do**
8: $\Delta(P_1) = \Delta(P_1) \cup \{h_1 \vee \cdots \vee h_k \vee \neg p_1 \vee \cdots \vee \neg p_m \vee p'_{m+1} \vee \cdots \vee p'_n\} \cup \{h'_1 \vee \cdots \vee h'_k \vee \neg p'_1 \vee \cdots \vee \neg p'_m \vee p'_{m+1} \vee \cdots \vee p'_n\}$
9: **end for**

10: **for all** rule $r : h_1; \ldots; h_k \leftarrow p_1, \ldots, p_m, p_{m+1} \ldots, p_n$ in P_2 **do**
11: $G = F \cup \Delta(P_1) \cup \{\neg h_1, \ldots, \neg h_k, p_1, \ldots, p_m, \neg p'_{m+1}, \ldots, \neg p'_n\}$
12: **if** G is satisfiable **then** ▷ this is implemented in **SELP** by calling zChaff
13: **return** FALSE
14: **end if**
15: $G = F \cup \Delta(P_1) \cup \{\neg h'_1, \ldots, \neg h'_k, p'_1, \ldots, p'_m, \neg p'_{m+1}, \ldots, \neg p'_n\}$
16: **if** G is satisfiable **then**
17: **return** FALSE
18: **end if**
19: **end for**

20: Exchange P_1 and P_2 do step 6 to step 19

21: **return** TRUE
22: **end procedure**

Theorem 2. *Let P_1 and P_2 be two programs, M a model of $\{p \supset p' | p \in L\} \cup \Delta(P_1)$, and not $\Delta(P_2)$. Let M_L and $M_{L'}$ be the two sets of atoms defined as follows:*

$$M_L = \{p \mid p \in L \text{ and } M \models p\}, \tag{6}$$
$$M_{L'} = \{p \mid p \in L \text{ and } M \models p'\}. \tag{7}$$

Then we have

(1) If $M_{L'}$ is not closed under P_2, then $P_1 \cup P$ and $P_2 \cup P$ is not equivalent, where $P = \{p \leftarrow \mid p \in M_{L'}\}$.
(2) If $M_{L'}$ is closed under P_2, then $P_1 \cup P$ and $P_2 \cup P$ is not equivalent, where $P = \{p \leftarrow \mid p \in M_L\} \cup \{p \leftarrow q \mid p, q \in M_{L'} \setminus M_L, p \neq q\}$.

Proof. Follows from the proofs of Theorem 1 in [5] and Theorem 1 in [6].

For example, given $P_1 = \{(a \leftarrow not\, b), (b \leftarrow not\, a)\}$ and $P_2 = \{a; b \leftarrow\}$, the answer of **SELP** will return the counter-example $P = \{(a \leftarrow b), (b \leftarrow a)\}$. However, **SELP** will confirm that $P_1 \cup \{\leftarrow a, b\}$ and $P_2 \cup \{\leftarrow a, b\}$ are strongly equivalent. As another example, consider $P_3 = \{(a \leftarrow b, c), (a \leftarrow b, not\, c)\}$ and $P_4 = \{a \leftarrow b\}$. They are not strongly equivalent, and $\{(a \leftarrow c), (c \leftarrow a), (b \leftarrow)\}$ is the counter-example returned by **SELP**.

While the basic function of our system **SELP** is to check whether two given logic programs are strongly equivalent, and if not provides a witness, we do not envision its use this way. Rather, we consider it a tool to systematically study the notion of strong equivalence. In the following, we discuss its use in discovering classes of strongly equivalent logic programs [7], and in searching for simpler sets of rules that are strongly equivalent to a given one.

4 Discovering general theorems

As we have mentioned, one possible use of strongly equivalent logic program is in program simplification. For instance, if a rule is strongly equivalent to an empty set, then we can delete this rule in any program without changing the answer sets of the program. It is well-known that a rule r is strongly equivalent to empty set if $Hd_r \cap Ng_r \neq \emptyset$. But is this the only case, i.e. is this both a sufficient and necessary condition for a rule to be strongly equivalent to the empty set?

In [7], we described a methodology and proved some general theorems for discovering theorems like this. More precisely, we were interested in the following so-called k-m-n theorem-discovery problem: Find some computationally effective conditions under which a set $\{r_1, ..., r_k, u_1, ..., u_m\}$ of $k + m$ rules is strongly equivalent to a set $\{r_1, ..., r_k, v_1, ..., v_n\}$ of $k + n$ rules. Notice that these two sets share k rules. This is to capture the so-called conditional strong equivalence. For instance, as we mentioned above, if r is strongly equivalent to $\emptyset$, then we can delete r from any logic program. However, if we want to delete r in a logic program only under the condition that another rule r' is in the program, then what we need is to check that $\{r', r\}$ and $\{r'\}$ are strongly equivalent.

We will not go into details of how we addressed the k-m-n problems (see [7]). The basic idea is to first find a condition that captures the strong equivalence between $\{r_1, ..., r_k, u_1, ..., u_m\}$ and $\{r_1, ..., r_k, v_1, ..., v_n\}$ when all the rules are from a small language, say with only three atoms. To do this, we generate all possible triples of the form (S_1, S_2, S_3), where S_1,S_2 and S_3 are sets of k number, m number and n number, respectively, of rules, and run **SELP** to check whether $S_1 \cup S_2$ is strongly equivalent to $S_1 \cup S_3$. The following are some of the experimental results that we obtained using **SELP**.

Lemma 1. *If r mentions at most three distinct atoms, then r is s.e. to $\emptyset$ iff $(Hd_r \cup Ng_r) \cap Ps_r \neq \emptyset$.*

Lemma 2. *For any two rules r_1 and r_2 that mentions at most four atoms, $\{r_1, r_2\}$ and $\{r_1\}$ are strongly equivalent iff one of the following two conditions is true:*

1. *$\{r_2\}$ is strongly equivalent to $\emptyset$.*
2. *$Ps_{r_1} \subseteq Ps_{r_2}$, $Ng_{r_1} \subseteq Ng_{r_2}$, $Hd_{r_1} \subseteq Hd_{r_2} \cup Ng_{r_2}$.*

Lemma 3. *For any three rules r_1, r_2 and r_3 that mentions at most six atoms, $\{r_1, r_2, r_3\}$ and $\{r_1, r_2\}$ are strongly equivalent iff one of the following four conditions is true:*

1. *$\{r_3\}$ is strongly equivalent to $\emptyset$.*
2. *$Ps_{r_1} \subseteq Ps_{r_3}$, $Ng_{r_1} \subseteq Ng_{r_3}$, $Hd_{r_1} \subseteq Hd_{r_3} \cup Ng_{r_3}$.*
3. *$Ps_{r_2} \subseteq Ps_{r_3}$, $Ng_{r_2} \subseteq Ng_{r_3}$, $Hd_{r_2} \subseteq Hd_{r_3} \cup Ng_{r_3}$.*
4. *there is an atom p such that:*
   - *4.1 $p \in (Ps_{r_1} \cup Ps_{r_2}) \cap (Hd_{r_1} \cup Hd_{r_2} \cup Ng_{r_1} \cup Ng_{r_2})$*
   - *4.2 $Hd_{r_i} \setminus \{p\} \subseteq Hd_{r_3} \cup Ng_{r_3}$ and $Ps_{r_i} \setminus \{p\} \subseteq Ps_{r_3}$ and $Ng_{r_i} \setminus \{p\} \subseteq Ng_{r_3}$, where $i = 1, 2$*
   - *4.3 If $p \in Ps_{r_1} \cap Ng_{r_2}$, then $Hd_{r_1} \cap Hd_{r_3} = \emptyset$*
   - *4.4 If $p \in Ps_{r_2} \cap Ng_{r_1}$, then $Hd_{r_2} \cap Hd_{r_3} = \emptyset$*

Some general theorems are proved in [7] that help us verifying that Lemmas 1-3 in fact hold in the general case.

Theorem 3. *Lemma 1-3 hold in the general case, without any restriction on the number of atoms.*

An important consequence of this theorem is the following theorem that solves the 0-1-1 problem:

Theorem 4 (0-1-1). *For any two rules r_1 and r_2, $\{r_1\}$ and $\{r_2\}$ are strongly equivalent iff one of the following two conditions is true:*

1. *$\{r_1\}$ and $\{r_2\}$ are both strongly equivalent to $\emptyset$.*
2. *$Ps_{r_1} = Ps_{r_2}$, $Ng_{r_1} = Ng_{r_2}$, and $Hd_{r_1} \cup Ng_{r_1} = Hd_{r_2} \cup Ng_{r_2}$.*

Proof. By Theorem 1, it is easy to see that $\{r_1\}$ and $\{r_2\}$ are strongly equivalent iff $\{r_1, r_2\}$ and $\{r_1\}$ are strongly equivalent and $\{r_1, r_2\}$ and $\{r_2\}$ are strongly equivalent.

Notice that for a language with six atoms, there are in principle $(2^6)^3-1 = 262,143$ possible rules. So for Lemma 3, which is about the 2-1-0 problem, there would be $262,143^3$ cases to check, which would be impossible to do using currently available computers. Fortunately, we can cut the numbers down significantly. First, by Theorems 3 and 4, we only need to consider rules where each atom occurs at most once: for any rule r, if there is an atom p that occurs more than once in r, then one of the following two cases applies:

1. $p \in Ps_r \cap (Ng_r \cup Hd_r)$, in this case, r is strongly equivalent to the empty set;
2. $p \in Ng_r \cap Hd_r$, in this case, r is strongly equivalent to $(Hd_r \backslash Ng_r) \leftarrow Ps_r, not\, Ng_r$.

In the following, we call rules in which each atom occurs at most once *non-redundant* rules. Second, we do not have to consider isomorphic rules: if there is a one-to-one onto function from L to L that maps $\{r_1, \ldots\}$ to $\{r'_1, \ldots\}$, then these two sets of rules are essentially the same except the names of atoms in them.

In general, we have the following theorem, which says that for checking a condition about the k-m-n problem in a language with N atoms, it is sufficient to consider $\binom{4^{k+m+n}+N-1}{N}$ cases. For instance, for Lemma 3, there are $\binom{69}{6} = 119,877,472$ cases to check, which took about 10 hours on a Solaris server consisting of 8 Sun UltraSPARC III 900Mhz CPUs with 8GB RAM.

Theorem 5. *Let L be a language with N atoms $a_1, \ldots, a_N$. For any $n_1, n_2, n_3 \geq 0$, we can generate a set S of less than $\binom{4^M+N-1}{N}$ triples (R_1, R_2, R_3), where $M = n_1 + n_2 + n_3$ and for each $1 \leq i \leq 3$, R_i is a set of no more than n_i number of rules in L, such that for any sets W_1, W_2, W_3 of n_1, n_2, n_3 rules, respectively, in L, there is a triple (R_1, R_2, R_3) in S and an automorphism f of L such that $R_1 \cup R_2$ and $f(W_1) \cup f(W_2)$ are strongly equivalent, and $R_1 \cup R_3$ and $f(W_1) \cup f(W_3)$ are strongly equivalent, where for any set W of rules, $f(W)$ is the result of replacing each atom a in every rule of W by $f(a)$.*

The proof of this theorem is omitted, and we give some intuitive explanations here. The problem is to generate M rules using N atoms. As mentioned above, we consider only non-redundant rules. For example, when $M = 1$, the problem is to generate one rule using N atoms. There are 4 cases of the occurrence of an atom in a non-redundant rule:

(i) in head of the rule,
(ii) in the body of the rule positively,
(iii) in the body of the rule negatively,
(vi) none of above.

Let r and r' be two rules, and, for both of them, there are x_0 atoms of case (i), x_1 atoms of case (ii), x_2 atoms of case (iii), and x_3 atoms of case (vi), where $x_0+x_1+x_2+x_3 = N$. We can easily define an automorphism of L and map r and r' to the same rule. So r and r' are isomorphic rules. Thus, for $M = 1$, the triples in S is less than the non-negative integer answers of equation $x_0 + x_1 + x_2 + x_3 = N$.

For $M = 2$, there are $4^2 = 16$ cases of the occurrence of an atom in two rules r_1, r_2:

(i) in head of r_1 and in the head of r_2,
(ii) in head of r_1 and in the body of r_2 positively,
(iii) in head of r_1 and in the body of r_2 negatively,
(vi) in head of r_1 and not in r_2,
(vi) in the body of r_1 positively and in head of r_2,
... ...
(xvi) not in r_1 and not in r_2

So, for $M = 2$, the triples in S is less than the non-negative integer answers of equation $x_0 + x_1 + x_2 + \cdots + x_{15} = N$. In general, there are 4^M cases of the occurrence of an atom in M rules, and the triples in S is less than the non-negative integer answers of equation $x_0 + x_1 + x_2 + \cdots + x_{4^M-1} = N$. Notice that there are exactly $\binom{4^M+N-1}{N}$ non-negative integer answers of $x_0 + x_1 + x_2 + \cdots + x_{4^M-1} = N$.

5 Finding strongly equivalent logic programs

Our second application of **SELP** is about finding out whether there is another, preferably simpler logic program that is strongly equivalent to a given one. For instance, we have seen that the self-loops (loops of length one) like $p \leftarrow p, q$ are strongly equivalent to $\emptyset$. A natural follow-up question is then: what about loops of length two, like $\{(a \leftarrow b), (b \leftarrow a)\}$?

Given a program P in the language L, an obvious way to look for another program in L that is strongly equivalent to P would be to generate all possible programs in L, and call **SELP** on them one by one. This is clearly infeasible even for a program with only three or four atoms. Fortunately, there is a much better way of doing it. Instead of considering all possible sets of rules, we can first find all possible rules that are redundant in the presence of P, i.e. all rules r such that $P \cup \{r\}$ is strongly equivalent to P, and consider sets of these rules only, as the following theorem says.

Theorem 6. *Let P be a logic program in L, and S the set of rules defined as follows:*

$$S = \{r \mid r \text{ is in } L, \text{ and } P \cup \{r\} \text{ and } P \text{ are strongly equivalent } \}.$$

For any program Q in L, if P and Q are strongly equivalent, then $Q \subseteq S$.

Proof. Follows from Theorem 1.

Notice that the set S in the theorem includes "trivial" rules like $p \leftarrow p$. As we mentioned in Section 4, by Theorem 3 and Theorem 4, we need to consider only rules where each atom occurs at most once, i.e. non-redundant rules.

Corollary 1. *Let P be a logic program in L, and S_P the set of rules defined as follows:*

$$S_P = \{r \mid r \text{ is a non-redundant rule in } L, \text{ and } P \cup \{r\} \text{ and } P \text{ are strongly equivalent } \}.$$

For any program Q in L, if P and Q are strongly equivalent, then $Q' \subseteq S_P$, where Q' is obtained from Q by deletion rules that are strongly equivalent to $\emptyset$, and replace each remaining rule r by $Hd_r \setminus Ng_r \leftarrow Ps_r, not\ Ng_r$.

Using this corollary, our system **SELP** finds all programs that are strongly equivalent to a given logic program in two steps:

- generate all possible non-redundant rule r, and check if P is strongly equivalent to $P \cup \{r\}$, thus computing the set S_P,
- for each subset of S_P, check if it is strongly equivalent to P.

For our example loop with length two, $P_1 = \{(a \leftarrow b), (b \leftarrow a)\}$, S_{P_1} consists of the following rules:

$$a \leftarrow b \qquad b \leftarrow a \qquad \leftarrow a, not\, b \qquad \leftarrow b, not\, a$$

As it turned out, there is no subset of S_{P_1} that is strongly equivalent to P_1 yet does not contain P_1, i.e. P_1 cannot be simplified using strong equivalence.

As another example, consider

$$P_2 = \{(a_1 \leftarrow not\, a_2), (a_2 \leftarrow not\, a_3), (a_3 \leftarrow not\, a_1)\}.$$

This is a program with an odd cycle [8, 9], i.e. there is a simple cycle in the dependency graph of the program that has an odd number of negative edges. Odd cycles in a logic program act as constraints [9]. For instance, we have already seen that $a \leftarrow not\, a$, the odd cycle with length one, is strongly equivalent to the constraint $\leftarrow not\, a$. The hope is that odd cycles of greater lengths, like P_2, can be similarly reduced to a set of constraints. Unfortunately, like P_1, P_2 cannot be simplified using strong equivalence either. This means that how rules in P_2 act as a constraint depends on other rules, thus cannot be determined locally.

The details of our experiment with P_2 is as follows. **SELP** returned S_{P_2} as the set of following rules:

$\leftarrow not\, a_1, not\, a_2$	$\leftarrow a_3, not\, a_1, not\, a_2$	$\leftarrow not\, a_1, not\, a_3$
$\leftarrow a_2, not\, a_1, not\, a_3$	$\leftarrow not\, a_2, not\, a_3$	$\leftarrow a_1, not\, a_2, not\, a_3$
$\leftarrow not\, a_1, not\, a_2, not\, a_3$	$a_1 \leftarrow not\, a_2$	$a_1 \leftarrow a_3, not\, a_2$
$a_1 \leftarrow not\, a_2, not\, a_3$	$a_2 \leftarrow not\, a_3$	$a_2 \leftarrow a_1, not\, a_3$
$a_2 \leftarrow not\, a_1, not\, a_3$	$a_3 \leftarrow not\, a_1$	$a_3 \leftarrow a_2, not\, a_1$
$a_3 \leftarrow not\, a_1, not\, a_2$	$a_1; a_2 \leftarrow not\, a_3$	$a_1; a_3 \leftarrow not\, a_2$
$a_2; a_3 \leftarrow not\, a_1$		

The following are programs that consist of rules from S_{P_2}, do not contain P_2 as a subset, and are strongly equivalent to P_2.

$a_1 \leftarrow a_3, not\ a_2$	$a_2 \leftarrow a_1, not\ a_3$	$a_1 \leftarrow a_3, not\ a_2$
$a_2 \leftarrow a_1, not\ a_3$	$a_3 \leftarrow a_2, not\ a_1$	$a_3 \leftarrow a_2, not\ a_1$
$a_3 \leftarrow a_2, not\ a_1$	$a_1; a_2 \leftarrow not\ a_3$	$a_1; a_3 \leftarrow not\ a_2$
$a_1; a_2 \leftarrow not\ a_3$	$a_2; a_3 \leftarrow not\ a_1$	$a_2; a_3 \leftarrow not\ a_1$
$a_1; a_3 \leftarrow not\ a_2$	$a_1 \leftarrow not\ a_2$	$a_2 \leftarrow not\ a_3$
$a_2; a_3 \leftarrow not\ a_1$		
$a_3 \leftarrow a_2, not\ a_1$	$a_1 \leftarrow a_3, not\ a_2$	$a_2 \leftarrow a_1, not\ a_3$
$a_2; a_3 \leftarrow not\ a_1$	$a_2 \leftarrow a_1, not\ a_3$	$a_1; a_2 \leftarrow not\ a_3$
$a_1 \leftarrow not\ a_2$	$a_1; a_2 \leftarrow not\ a_3$	$a_1 \leftarrow not\ a_2$
$a_2 \leftarrow not\ a_3$	$a_1; a_3 \leftarrow not\ a_2$	$a_3 \leftarrow not\ a_1$
	$a_3 \leftarrow not\ a_1$	
$a_1 \leftarrow a_3, not\ a_2$		
$a_1; a_3 \leftarrow not\ a_2$		
$a_2 \leftarrow not\ a_3$		
$a_3 \leftarrow not\ a_1$		

As one can see, none of them contain constraints. Actually, they can all be explained by the theorems in [7]. For example, consider $Q = \{(a_1 \leftarrow not\ a_2), (a_2 \leftarrow a_1, not\ a_3), (a_1; a_2 \leftarrow not\ a_3), (a_3 \leftarrow not\ a_1)\}$. Q is strongly equivalent to P because the first two rules are already in P, and the set of the last two rules is strongly equivalent to the last rule in P.

6 Concluding remarks and future work

We develop a tool **SELP** for studying strong equivalence between logic programs. It can check if two programs are strongly equivalent, as well as, it is helpful for us to find some general theorems on strong equivalence. We will try to find some more theorems by **SELP**. It is also an interesting works to find some general theorems on uniform equivalence [1], and to find the difference between two kinds of equivalence.

References

1. Thomas Eiter and Michael Fink. Uniform equivalence of logic programs under the stable model semantics. In Catuscia Palamidessi, editor, *ICLP*, volume 2916 of *Lecture Notes in Computer Science*, pages 224–238. Springer, 2003.
2. Thomas Eiter, Michael Fink, Hans Tompits, and Stefan Woltran. Simplifying logic programs under uniform and strong equivalence. In Vladimir Lifschitz and Ilkka Niemelä, editors, *LPNMR*, volume 2923 of *Lecture Notes in Computer Science*, pages 87–99. Springer, 2004.
3. Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.
4. Tomi Janhunen and Emilia Oikarinen. Lpeq and dlpeq - translators for automated equivalence testing of logic programs. In Vladimir Lifschitz and Ilkka Niemelä, editors, *LPNMR*, volume 2923 of *Lecture Notes in Computer Science*, pages 336–340. Springer, 2004.

5. V. Lifschitz, D. Pearce, and A. Valverde. Strongly equivalent logic programs. *ACM Transactions on Computational Logic*, 2(4):526–541, 2001.
6. Fangzhen Lin. Reducing strong equivalence of logic programs to entailment in classical propositional logic. In *Proceedings of the Eighth International Conference on Principles of Knowledge Representation and Reasoning (KR2002)*, pages 170–176, 2002.
7. Fangzhen Lin and Yin Chen. Discovering classes of strongly equivalent logic programs. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI–05)*, 2005. To appear.
8. Fangzhen Lin and Jia-Huai You. Abduction in logic programming: A new definition and an abductive procedure based on rewriting. *Artificial Intelligence*, 140(1/2):175–205, 2002.
9. Fangzhen Lin and Xishun Zhao. On odd and even cycles in normal logic programs. In *Proceedings of the 19th National Conference on Artificial Intelligence (AAAI–2004), AAAI Press, Menlo Park, CA.*, pages 80–85, 2004.
10. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *DAC*, pages 530–535. ACM, 2001.
11. Hudson Turner. Strong equivalence for logic programs and default theories (made easy). In *Proceedings of LPNMR'2001*, pages 81–92, 2001.