Program Completion as Constraint Satisfaction: Tight Logic Programs Revisited

Yun Bai and Yan Zhang
Artificial Intelligence Research Group
School of Computing, Engineering and Mathematics
University of Western Sydney
Locked Bag 1797, Penrith, NSW 2751, Australia
Email: ybai@scem.uws.edu.au, yan@scem.uws.edu.au

Abstract—Research in logic programming shows an increasing interest in studying tight logic programs because as Fages proved, each stable model of a tight logic program is identical to a logic model of a corresponding propositional theory (called the Clark’s completion of the program), and vice versa. Therefore, any algorithms for solving the satisfiability problem may be used to compute stable models of tight logic programs. Furthermore, it has been also observed that many important problems can be encoded into tight logic programs. However, it is still unclear whether we can give a better characterization on the tractability of tight logic programs although some obvious tractable subclass is easy to be recognized. In this paper, we investigate the computational complexity of propositional tight logic programs under stable model semantics. In particular, we provide explicit syntactic characterizations for various tractable subclasses of tight logic programs. Our approach is to transform the completions of tight logic programs to instances of the constraint satisfaction problem (CSP), and then apply Schaefer’s Dichotomy Theorem to obtain sufficient tractability conditions for tight logic programs. Based on this approach, we further specify an interesting subclass of tight logic programs whose computational tractability may be decided through their decompositions.

Key words: Knowledge representation, Automated problem solving, Reasoning strategies, Logic programming, Nonmonotonic logic

I. INTRODUCTION

Research in logic programming shows an increasing interest in studying tight logic programs (e.g. [1], [2], [9], [15], [17]) because as Fages proved, each stable model of a tight logic program is identical to a logic model of a corresponding propositional theory (called the Clark’s completion of the program), and vice versa. Therefore, any algorithms for solving the satisfiability problem may be used to compute stable models of tight logic programs [1], [6]. Furthermore, it has been also observed that many important problems can be encoded into tight logic programs. However, the computational tractability property of tight logic programs has yet been thoroughly studied. Specifically, it is still not clear whether we can provide some syntactic but rather general characterizations for tractable subclasses of tight logic programs. In this paper, we investigate the computational complexity of propositional tight logic programs under stable model semantics. In particular, we provide explicit syntactic characterizations for various tractable subclasses of tight logic programs. Our approach is to construct a polynomial transformation from the completions of tight logic programs to instances of the constraint satisfaction problem (CSP), and then apply Schaefer’s Dichotomy Theorem to obtain sufficient tractability conditions for tight logic programs. Based on this approach, we further specify an interesting subclass of tight logic programs whose computational tractability may be decided through their decompositions.

The paper is organized as follows. Section 2 introduces some preliminary logic concepts and notations we need in this paper. Section 3 describes a polynomial transformation from the completion of a tight logic program to a Boolean CSP-instance. Section 4 gives the main tractability result of this paper and provides a formal proof. Section 5 specifies the class of decomposable tight logic programs and depicts how these programs’ tractability may be recognized through their decompositions. Finally, section 6 discusses the related work and concludes the paper with some remarks.

II. PRELIMINARIES

The relationship between Clark’s completion semantics and stable model semantics has been intensively studied by researchers, e.g. [2], [9], [10], [12]. It has been showed that by compiling a logic program into a classical propositional theory - the completion of this program, each stable model of the program is a logical model of the theory, and under certain condition called tightness, each logical model of the propositional theory is also a stable model of this program. We are interested in the computational issue of this type of logic programs because it has been observed that many important problems can be encoded into tight logic programs.

We consider finite propositional normal logic programs in which each rule has the form:

\[ a \leftarrow b_1, \ldots, b_m, \neg c_1, \ldots, \neg c_n \]  \hspace{1cm} (1)

where \( a \) is either a propositional atom or empty, \( b_1, \ldots, b_m, c_1, \ldots, c_n \) are propositional atoms. When \( a \) is empty, rule (1)
is called a constraint. Given a rule \( r \) of the form (1), we denote \( \text{head}(r) = a, \text{pos}(r) = \{b_1, \cdots, b_m\} \) and \( \text{neg}(r) = \{c_1, \cdots, c_n\} \), and therefore, rule (1) may be represented as the form:

\[
\text{head}(r) \leftarrow \text{pos}(r), \text{not } \neg \text{neg}(r).
\]

We also use \( \text{Atom}(II) \) to denote the set of all propositional atoms occurring in program \( II \).

The stable model of a program \( II \) is defined as follows. Firstly, we consider \( II \) to be the program in which each rule does not contain negation as failure sign \( \text{not} \). A finite set \( S \) of propositional atoms is called a stable model of \( II \) if \( S \) is the smallest set such that for each rule \( a \leftarrow b_1, \cdots, b_m \) from \( II \), if \( b_1, \cdots, b_m \in S \), then \( a \in S \). Now let \( II \) be an arbitrary normal logic program. For any set \( S \) of propositional atoms, program \( S^\Psi \) is obtained from \( II \) by deleting (1) each rule from \( II \) that contains \( \text{not} \ c \) in the body and \( c \in S \); and (2) all forms of \( \text{not} \ c \) in the bodies of the remaining rules\(^2\). Then \( S \) is a stable model of \( II \) if and only if \( S \) is a stable model of \( S^\Psi \). It is easy to see that a program may have one, more than one, or no stable models at all.

**Definition 1:** [3] Given a logic program \( II \) consisting of rules of the form (1). Its completion \( \text{Comp}(II) \) is obtained in three steps:

- **Step 1:** Replace each rule of the form (1) with the formula:
  \[
  b_1 \land \cdots \land b_m \land \neg c_1 \land \cdots \land \neg c_n \supset a.
  \]
  We may use notion \( \text{body} \supset a \) to represent this formula.

- **Step 2:** For each symbol \( a \) (\( a \) is not empty), let \( \text{Support}(a) \) denote the set of formulas obtained from all rules with \( a \) in the head as showed in Step 1. Suppose \( \text{Support}(a) \) is the set:
  \[
  \text{body}_1 \supset a, \cdots, \text{body}_k \supset a.
  \]
  Replace this set with a single formula
  \[
  a \equiv \text{body}_1 \lor \cdots \lor \text{body}_k.
  \]
  If \( \text{Support}(a) = \emptyset \) then replace it by \( \neg a \).

- **Step 3:** For each constraint in \( II \):
  \[
  \leftarrow b_1, \cdots, b_m, \text{not } c_1, \cdots, \text{not } c_n,
  \]
  replace it with a formula \( \text{Cons} : \neg b_1 \lor \cdots \lor \neg b_m \lor c_1 \lor \cdots \lor c_n \).

Then \( \text{Comp}(II) \) is the set of all such \( \text{Support}(a) \) and \( \text{Cons} \).

Gelfond and Lifschitz showed that given a program \( II \), each stable model of \( II \) is also a (minimal) model of \( \text{Comp}(II) \) [12]. However, the converse is usually not true. Consider program \( II \) consisting of a single rule: \( a \leftarrow p \). Clearly, \( \text{Comp}(II) = \{p \equiv p\} \), which has two models \( \emptyset \) and \( \{p\} \). But program \( II \) only has one stable model \( \emptyset \). Fages defined a syntactic condition on logic programs that ensures the equivalence between stable model semantics and completion semantics. This is so called tightness of a program.

\(^2\) We call \( \text{Comp}^\Psi \) is the result of Gelfond-Lifschitz transformation on \( II \) with \( S \).

\(^3\) If some \( \text{body}_i \equiv \top \), then \( \text{Support}(a) = a \).

**Definition 2:** Given a program \( II \). \( II \) is tight if there exists a function \( \lambda \) from the set of all propositional atoms to ordinals such that for each rule (1) in \( II \),

\[
\lambda(a) > \lambda(b_i) \quad \text{for all } i \ (1 \leq i \leq m).
\]

Fages showed that for any tight logic program \( II \), a set of propositional atoms \( S \) is a stable model of \( II \) if and only if \( S \) is a model of \( \text{Comp}(II) \) [10]. Immediately from this statement, we have the following result.

**Proposition 1:** A tight logic program \( II \) has a stable model if and only if \( \text{Comp}(II) \) is satisfiable.

Let \( S \) be a set. The size of \( S \), denoted as \( |S| \), is defined to be the cardinality of \( S \). Then since a program \( II \) is a finite set of rules, the size of \( II \), i.e. \( |II| \), is the number of rules in \( II \). It is well known that for a given set \( S \) of propositional atoms and a logic program \( II \), deciding whether \( S \) is a stable model of \( II \) can be solved in polynomial time in terms of the size of \( S \) and \( II \). But deciding whether a logic program has a stable model is NP-complete [18], which implies that in practice it is unlikely to implement a general algorithm to compute stable models of logic programs polynomially. However, we may identify tractable subclasses of tight logic programs by investigating the tractable computation of the models of their program completions as the following proposition states.

**Proposition 2:** Deciding whether a tight logic program \( II \) has a stable model can be achieved in polynomial time if deciding the satisfiability of \( \text{Comp}(II) \) can be achieved in polynomial time.

### III. Program Completion as Constraint Satisfaction

Given a logic program, we can view the satisfiability of its completion as a constraint satisfaction problem (CSP). Formally, an instance of CSP (or say a CSP-instance) is a triple \((V,D,C)\), consisting of a finite set \( V \) of variables, a finite domain \( D \) of values, and a set \( C \) of constraints \((t,R)\) where \( t \) is a tuple \( t = (x_1, \cdots, x_m) \) of variables for some \( m \) and \( R \) is a relation on \( D \) of arity \( m \). Sometimes, we also call \( R \) a relation in the CSP-instance \((V,D,C)\). A solution to a CSP-instance is a mapping \( h : V \rightarrow D \) such that for every constraint \((t,R) \in C\), \( h(t) = (h(x_1), \cdots, h(x_m)) \in R \).

The classical Boolean satisfiability problem SAT can be viewed as a CSP-instance. For example, given a CNF formula \( \varphi = (p \lor q) \land (q \lor r \lor \neg s) \), we transform it to a CSP-instance \((V,D,C)\), where \( V = \{p,q,r,s\} \), \( D = \{0,1\} \) and \( C \) consists of two constraints \((p,q), \{0,1\}^2 \setminus \{(0,0)\}\) and \((q,r,s), \{0,1\}^3 \setminus \{(0,0,0)\}\). In this CSP-instance, for each constraint \((t,R) \in C\), relation \( R \) is a subset of \( \{0,1\}^{|t|} \), and we call it logical relation\(^4\). A CSP-instance of this form is also called a Boolean CSP-instance. Given a logic program \( II \), we can always represent its completion \( \text{Comp}(II) \) as a CNF formula, and then transform this CNF formula to a Boolean CSP-instance, as shown by the following example.

**Example 1:** Consider program \( II \) consisting of the following rules:

\[^4|t|\] denotes the cardinality of the set of variables occurring in tuple \( t \).
Then the corresponding CSP-instance is: $\text{Comp}(\Pi) = \{p, q, -s, -t, r \equiv ((p \land \neg s) \lor (q \land \neg t))\}$, which is equivalent to the following set of clauses:

- $p$
- $q$
- $\neg s$
- $\neg t$
- $r \lor \neg p \lor s$
- $r \lor q \lor \neg t$
- $\neg r \lor p \lor \neg t$
- $\neg r \lor q \lor \neg s$
- $\neg r \lor \neg s \lor \neg t$

Then the corresponding CSP-instance is: $(V, D, C)$, where $V = \{p, q, r, s, t\}$, $V = \{0, 1\}$, and $C$ consists of the following constraints:

- $(\{p\}, \{1\})$
- $(\{q\}, \{1\})$
- $(\{r, q, t\}, \{0, 1\}^3 - \{(0, 1, 0)\})$
- $(\{q\}, \{1\})$
- $(\{r, p, q\}, \{0, 1\}^3 - \{(1, 0, 0)\})$
- $(\{s\}, \{0\})$
- $(\{r, p, t\}, \{0, 1\}^3 - \{(1, 0, 1)\})$
- $(\{t\}, \{0\})$
- $(\{r, q, s\}, \{0, 1\}^3 - \{(1, 0, 1)\})$
- $(\{r, p, s\}, \{0, 1\}^3 - \{(0, 1, 0)\})$
- $(\{r, s, t\}, \{0, 1\}^3 - \{(1, 1, 1)\})$

Given a logic program $\Pi$, we call the CSP-instance constructed as described in Example 1 the $\Pi$'s Boolean CSP-instance, denoted as $\text{CSP}(\Pi)$. Then we have the following proposition.

**Proposition 3:** A tight logic program $\Pi$ has a stable model if and only if $\Pi$'s Boolean CSP-instance $\text{CSP}(\Pi)$ has a solution.

From previous discussion, we observe that for a given clause $l_1 \lor \cdots \lor l_k$, the logical relation in the corresponding constraint has a size of $2^k$. Then it seems that encoding an arbitrary CNF formula into a CSP-instance may exponentially depend on the length of clauses in the CNF formula. We should note that this actually is not a barrier of using CSP techniques to study satisfiability problem because for each fixed $k$, transforming a CNF formula, in which the length of its each clause is bounded by $k$, into a Boolean CSP-instance takes polynomial time as illustrated in [7], [14].

Similarly, to transform $\text{Comp}(\Pi)$ into $\text{CSP}(\Pi)$ polynomially, we may fix the underlying logic program’s rule length and support depth. Formally, consider a logic program $\Pi$ consisting of rules of the form $(1)$, we define the rule length of rule $r$ to be the number of atoms occurring in $r$, i.e. $r(l(r)) = 1 + m + n$. For atom $a \in \text{Atom}(\Pi)$, if its support $\text{Support}(a)$ in $\Pi$ is formula $a \equiv \text{body}_1 \lor \cdots \lor \text{body}_k$, then we define the support depth of $a$ to be the number of $\text{body}_j$ occurring in this formula, i.e. $\text{sd}(a) = k$. Now we define the rule length and support depth of the program $\Pi$ as follows:

- $rl(\Pi) = \text{Max}(r(l(r)) : r \in \Pi)$
- $sd(\Pi) = \text{Max}(\text{sd}(a) : a \in \text{Atom}(\Pi))$

**Theorem 1:** Let $\Pi$ be a logic program with fixed rule length $l$ and support depth $d$. $\Pi$’s Boolean CSP-instance $\text{CSP}(\Pi)$ can be constructed in at most $O((d2^l + l^d2^{d+1})|\Pi|)$ steps.

**Proof:** For each formula in $\text{Comp}(\Pi)$, it has one of the three forms: $a, \neg a$ or $a \equiv \text{body}_1 \lor \cdots \lor \text{body}_k$, where $k \leq d$, and each body is of the form $b_1 \lor \cdots \lor b_m \lor -c_1 \land -c_n$, where $(1 + m + n) \leq l$. It is easy to see that $|\text{Comp}(\Pi)| \leq l|\Pi|$ and $\text{Comp}(\Pi)$ contains at most $|\Pi|$ formulas of the form $a \equiv \text{body}_1 \lor \cdots \lor \text{body}_k$. This formula is equivalent to the following two formulas:

- $a \lor (\neg \text{body}_1 \land \cdots \land \neg \text{body}_k)$, and
- $\neg a \lor \text{body}_1 \lor \cdots \lor \text{body}_k$.

The first formula above can induce $k$ clauses each with a length of at most $l$. Then each relation in $\text{CSP}(\Pi)$ constructed from these clauses needs at most $2^l$ steps. Therefore, at most constructing all relations in $\text{CSP}(\Pi)$ from clauses induced from $a \lor (\neg \text{body}_1 \land \cdots \land \neg \text{body}_k)$ needs at most $k2^l \leq d2^l$ steps.

On the other hand, it is observed that formula $\neg a \lor \text{body}_1 \lor \cdots \lor \text{body}_k$ can at most induce $l^k$ clauses and each with a length of $k + 1$. So constructing relations of $\text{CSP}(\Pi)$ from these clauses needs at most $l^{k2^{d+1}} \leq l^d2^{d+1}$ steps. As $\text{Comp}(\Pi)$ contains at most $|\Pi|$ formulas of the form $a \equiv \text{body}_1 \lor \cdots \lor \text{body}_k$, altogether, constructing relations of $\text{CSP}(\Pi)$ from all clauses induced from $\text{Comp}(\Pi)$ needs at most $(d2^l + l^d2^{d+1})|\Pi|$ steps. Also note that all other formulas in $\text{Comp}(\Pi)$ are of the form $a$ or $\neg a$, and at most the number of these formulas in $\text{Comp}(\Pi)$ is $(l - 1)|\Pi|$ which is less than $(l^d2^{d+1})|\Pi|$ when $d \geq 1$. So given a logic program $\Pi$ with the fixed rule length $l$ and support depth $d$, we can construct $\text{CSP}(\Pi)$ in at most $O((d2^l + l^d2^{d+1})|\Pi|)$ steps.

We should note that in practice, constructing the Boolean CSP-instance of a given logic program needs much less steps than what Theorem 1 states, because in a program, the rule length for most rules is usually quite small, and the support depth for most atoms occurring in the program is just 1! This has been observed from many classical problem domains (e.g. N-queen problem and Hamiltonian Circuit problem) and practical applications like robotic planning studied in logic programming [3]. In the rest of the paper, without explicit declaration, we will consider all logic programs with fixed rule length and support depth. The following propositions illustrate several interesting properties of $\text{CSP}(\Pi)$ constructed from a given $\Pi$.

**Proposition 4:** Let $\Pi$ be a logic program with $rl(\Pi) = l$ and $sd(\Pi) = d$ where $l > d$. Then every relation in $\text{CSP}(\Pi)$ has at most arity $l$.

**Proposition 5:** Let $\Pi$ be a logic program with $rl(\Pi) = l$ and $sd(\Pi) = d$ where $d > l$. Then every relation in $\text{CSP}(\Pi)$ has at most arity $(d + 1)$.

**Proposition 6:** Let $\Pi$ be a logic program. If for each $a \in \text{Atom}(\Pi)$, there is a rule $r \in \Pi$ with $\text{head}(r) = a$, for any $r \in \Pi$ $rl(r) = l$, and $sd(\Pi) = 1$, then every relation in $\text{CSP}(\Pi)$ has exact arity $l$. 
IV. Tractable Subclasses of Tight Logic Programs

A. The Result

We first present the main characterization result for the computational tractability of tight logic programs.

Theorem 2: Let \( \Pi \) be an arbitrary tight logic program. Deciding whether \( \Pi \) has a stable model is solvable in polynomial time if one of the following five conditions is satisfied. In this case, we also call \( \Pi \) a tractable tight logic program.

1) For each rule \( r \in \Pi \), \( pos(r) \neq \emptyset \);
2) For each \( a \in \text{Atom}(\Pi) \), there is a rule \( r \in \Pi \) such that \( \text{head}(r) = a \), and there exists at least one rule \( r' \) with \( \text{head}(r') = a \) and \( \text{neg}(r') = \emptyset \); for each rule \( r \in \Pi \) with \( \text{head}(r) = \emptyset \) (i.e. \( r \) is a constraint), \( \text{neg}(r) \neq \emptyset \);
3) For each \( a \in \text{Atom}(\Pi) \), there is a rule \( r \in \Pi \) such that \( \text{head}(r) = a \), \( |pos(r) \cup \neg \text{neg}(r)| = 1 \) and \( \text{sd}(a) = 1 \); for each rule \( r \in \Pi \) with \( \text{head}(r) = \emptyset \), \( |pos(r) \cup \neg \text{neg}(r)| = 2 \);
4) For each rule \( r \in \Pi \) with \( \text{head}(r) \neq \emptyset \), \( \text{neg}(r) = \emptyset \) and \( |pos(r)| \leq 1 \); for each rule \( r \in \Pi \) with \( \text{head}(r) = \emptyset \), \( |pos(r)| \leq 1 \);
5) For each rule \( r \in \Pi \) with \( \text{head}(r) \neq \emptyset \), \( \text{neg}(r) \neq \emptyset \), either \( pos(r) \cup \neg \text{neg}(r) = \emptyset \), or \( |\text{sd}(\text{head}(r))| = 1 \) and \( \text{neg}(r) = \emptyset \); for each rule \( r \in \Pi \) with \( \text{head}(r) = \emptyset \), \( |\text{neg}(r)| \leq 1 \).

Example 2: Consider the following two tight logic programs:

\[
\Pi_1: \quad a \leftarrow \neg b, \quad b \leftarrow, \quad c \leftarrow, \quad d \leftarrow b, \quad e \leftarrow \neg c.
\]

\[
\Pi_2: \quad d \leftarrow \neg a, \quad a \leftarrow, \quad c \leftarrow, \quad \neg c \leftarrow a, b.
\]

It is easy to check that \( \Pi_1 \) satisfies condition 4 and \( \Pi_2 \) satisfies condition 5 in Theorem 2 respectively. In section 5, we will illustrate other program examples that satisfy conditions 1, 2, and 3 in Theorem 2.

B. The Proof of Theorem 2

As we have showed in last section, given a logic program \( \Pi \) with fixed rule length \( l \) and support depth \( d \), it can be transformed to a Boolean CSP-instance CSP(\( \Pi \)) in polynomial time. Then if \( \Pi \) is a tight logic program, and deciding whether CSP(\( \Pi \)) has a solution is solvable in polynomial time, it follows that deciding whether \( \Pi \) has a stable model is also solvable in polynomial time. Furthermore, each solution of CSP(\( \Pi \)) (if there is one) is also a stable model of \( \Pi \). The constraint satisfaction problem we considered for program completion is also called the generalized satisfiability problem for which Schaefer provided a complete characterization on its computational complexity [19]. Before we present Schaefer’s Dichotomy Theorem, we first introduce following useful concepts.

Let \( R \) be a \( k \)-ary logical relation. \( R \) is \( 0 \)-valid if tuple \((0, \cdots, 0) \in R \). \( R \) is \( 1 \)-valid if tuple \((1, \cdots, 1) \in R \). \( R \) is \textit{bijunctive} if \( R \) is the set of truth assignments satisfying some 2CNF formula. \( R \) is \textit{weakly positive} (or \textit{weakly negative}, resp.) if \( R \) is the set of truth assignments satisfying some CNF formula having at most one negative (or positive, resp.) variable in each conjunct. \( R \) is \textit{affine} if \( R \) is the set of solutions to a system of linear equations over the two-element field. That is, \( R \) is equivalent to a conjunction of formulas of the forms \( x_1' \oplus \cdots \oplus x_m' = 0 \) and \( x_1'' \oplus \cdots \oplus x_n'' = 1 \), where \( \{x_1', \ldots, x_m', x_1'', \ldots, x_n''\} \subseteq \{x_1, \ldots, x_k\} \), and \( \oplus \) denotes addition modulo 2.

Theorem 3: (Schaefer’s Dichotomy Theorem [19]) Let \((V, \{0, 1\}, C)\) be a Boolean CSP-instance. Deciding whether \((V, \{0, 1\}, C)\) has a solution is solvable in polynomial time if one of the following six conditions is satisfied, otherwise it is NP-complete.

(a) Every logical relation \( R \) in \((V, \{0, 1\}, C)\) is \( 0 \)-valid;
(b) Every logical relation \( R \) in \((V, \{0, 1\}, C)\) is \( 1 \)-valid;
(c) Every logical relation \( R \) in \((V, \{0, 1\}, C)\) is bijunctive;
(d) Every logical relation \( R \) in \((V, \{0, 1\}, C)\) is weakly positive;
(e) Every logical relation \( R \) in \((V, \{0, 1\}, C)\) is weakly negative;
(f) Every logical relation \( R \) in \((V, \{0, 1\}, C)\) is affine.

Now we consider how this theorem is used to prove Theorem 2. Basically, we will show that a tight logic program \( \Pi \) satisfying one of conditions 1-5 in Theorem 2 will imply that for CSP(\( \Pi \)), one of conditions (a)-(e) in Schaefer’s Dichotomy Theorem (Theorem 3) also holds accordingly, where condition (f) in Theorem 3 will never hold for CSP(\( \Pi \)).

Lemma 1: Given a tight logic program \( \Pi \) and its corresponding CSP(\( \Pi \)) = \((V, \{0, 1\}, C)\). Every logical relation \( R \) in CSP(\( \Pi \)) is \( 0 \)-valid if and only if only for each rule \( r \in \Pi \), \( pos(r) \neq \emptyset \).

Proof: Suppose every relation \( R \) in CSP(\( \Pi \)) is \( 0 \)-valid. From the construction of CSP(\( \Pi \)), we know that each \( k \)-ary relation \( R \) in CSP(\( \Pi \)) is constructed from a clause of the form \( l_1 \lor \cdots \lor l_k \), where \( l_i \) is either a variable in \( V \) or the negation of a variable in \( V \). Note that \( R \) is formed as \{0, 1\}^k – \{X\}, where \( X = \{\epsilon_1, \ldots, \epsilon_k\} \), \( \epsilon_i \in \{0, 1\} \) and is an assignment satisfying \( \text{Support}(a) \). Since \( R \) is \( 0 \)-valid, it means that \( X \neq \emptyset \). Consequently, it implies that \( l_1 \lor \cdots \lor l_k \) is a clause of the form \( x_1 \lor \cdots \lor x_k \) where each \( x_i \in V \). Without loss of generality, we assume that \( l_1 \lor \cdots \lor l_k \) \( \neg \) is not a clause of the form \( x_1 \lor \cdots \lor x_k \) where each \( x_i \in V \). Each solution implies the result to be trivially true. Now consider \( \text{Support}(a) = a \equiv \text{body}_1 \lor \cdots \lor \text{body}_n \), which is equivalent to the following two formulas:

\[\neg a \lor \text{body}_1 \lor \cdots \lor \text{body}_n, \quad a \lor \neg \text{body}_1 \lor \cdots \lor \neg \text{body}_n, \]

where each \( \text{body}_i \) is of the form \( b_1 \land \cdots \land b_m \land \neg c_1 \land \cdots \land c_n \). It is easy to see that each clause induced from the first formula above will contain \( \neg a \), and consequently, the relation constructed from this clause is \( 0 \)-valid. Now consider clauses...
induced from the second formula above. It is observed that each clause is of the form \( a \lor b \lor \cdots b_m \lor c_1 \lor \cdots \lor c_n \). As the relation in CSP(\( \Pi \)) constructed from this clause contains tuple \((0, \ldots, 0)\), it implies at least one of \( b_i (1 \leq i \leq m) \) should be in the clause \( a \lor -b_1 \lor \cdots \lor -b_m \lor c_1 \lor \cdots \lor c_n \). This means \( pos(r) \neq \emptyset \).

Now we assume that clause \( l_1 \lor l_2 \lor \cdots l_k \) is induced from some constraint in \( \Pi \):
\[
\langle -b_1, \ldots, -b_n, \text{not } c_1, \ldots, \text{not } c_n \rangle. \text{ Then the clause is of the form } \neg b_1 \land \cdots \land \neg b_m \lor c_1 \lor \cdots \lor c_n. \text{ Clearly, this formula is not of the form } x_1 \lor \cdots \lor x_k \text{ where each } x_i \in V. \text{ This implies } pos(r) \neq \emptyset. \text{ Combining these two cases, it concludes that to ensure every relation } R \text{ in } CSP(\Pi) \text{ is 0-valid, for each rule } r \in \Pi, \text{ it must be the case } pos(r) \neq \emptyset.
\]

The other direction can be proved in a similar way.

**Lemma 2:** Given a tight logic program \( \Pi \) and its corresponding CSP(\( \Pi \)) = \((V, \{0, 1\}, C)\). Every logical relation \( R \) in CSP(\( \Pi \)) is 1-valid if and only if for each \( a \in Atom(\Pi) \) there is a rule \( r \in \Pi \) such that \( head(r) = a \), and there exists at least one rule \( r' \in \Pi \) with \( head(r') = a \lor neg(r') = \emptyset \); for each rule \( r \in \Pi \) with \( head(r) = \emptyset \) (i.e. \( r \) is a constraint), \( neg(r) \neq \emptyset \).

**Proof:** We only prove one direction, while the other direction is proved in a similar way. Suppose each \( R \) in CSP(\( \Pi \)) is 1-valid. That is, \( R \) is constructed from a clause \( l_1 \lor l_2 \lor \cdots l_k \) from either formula \( Support(a) \) for some \( a \in Atom(\Pi) \) or \( Cons \) for some constraint in \( \Pi \). We first consider the case of \( Support(a) \). This implies that \( Support(a) \) is not of the form \( \neg a \) or \( a \). So it concludes that for each \( a \in Atom(\Pi) \), there is a rule \( r \): \( a \leftarrow pos(r), not \ neg(r), \) where \( pos(r) \lor neg(r) \neq \emptyset \). Also, it must be the case \( pos(r) \lor neg(r) \neq \emptyset \). Suppose \( sd(a) > 1 \). For instance, suppose \( sd(a) = 2 \). Then we have \( Support(a) = a \leftarrow l \lor l' \), where \( l \) and \( l' \) are atoms or negative atoms. Obviously, from \( a \leftarrow l \lor l' \), clause \( \neg a \lor l \lor l' \) is induced. This contradicts the fact that all relations in CSP(\( \Pi \)) are bijunctive.

Now suppose that \( R \) is constructed from a constraint \( r : \leftarrow pos(r), not \ neg(r) \). Obviously, the fact that \( R \) is bijunctive implies \( pos(r) \lor neg(r) = 2 \).

**Lemma 4:** Given a tight logic program \( \Pi \) and its corresponding CSP(\( \Pi \)) = \((V, \{0, 1\}, C)\). Every logical relation \( R \) in CSP(\( \Pi \)) is weakly positive if and only if for each rule \( r \in \Pi \) with \( head(r) = \emptyset \), \( neg(r) = \emptyset \) and \( \|pos(r)\| \leq 1 \); for each rule \( r \in \Pi \) with \( head(r) = \emptyset \), \( |pos(r)| \leq 1 \).

**Lemma 5:** Given a tight logic program \( \Pi \) and its corresponding CSP(\( \Pi \)) = \((V, \{0, 1\}, C)\). Every logical relation \( R \) in CSP(\( \Pi \)) is weakly negative if and only if for each rule \( r \in \Pi \) with \( head(r) = \emptyset \), either \( pos(r) \lor neg(r) = \emptyset \), or \( \|head(r)\| = 1 \) and \( neg(r) = \emptyset \); for each rule \( r \in \Pi \) with \( head(r) = \emptyset \), \( |neg(r)| \leq 1 \).

**Lemma 6:** Given a tight logic program \( \Pi \) and its corresponding CSP(\( \Pi \)) = \((V, \{0, 1\}, C)\). Then every logical relation \( R \) in CSP(\( \Pi \)) is not affine.

**Proof:** From [19], it shows that the cardinality of an affine relation is always a power of 2. However, from the construction of CSP(\( \Pi \)), we know that each relation \( R \) in CSP(\( \Pi \)) is formed as \( \{0, 1\}^{|\Pi|} - \{(\epsilon_1, \ldots, \epsilon_\ell)\} \) where \( |\ell| \geq 1 \) and \( \epsilon_\ell \in \{0, 1\} \), which has a cardinality of an odd number.

**Proof of Theorem 2** We only need to show that conditions 1-5 in Theorem 2 implies conditions (a) - (e) in Schaefer’s Dichotomy Theorem when the program is (polynomially) transformed to the corresponding Boolean CSP-instance. Lemmas 1-5 have shown such equivalence between conditions (a), (b), (c), (d), (e) in Schaefer’s Dichotomy Theorem and conditions 1, 2, 3, 4, and 5 in Theorem 2 respectively. Lemma 6, on the other hand, shows that for any program \( \Pi \), every relation in CSP(\( \Pi \)) is not affine, and hence condition (f) in Schaefer’s Dichotomy Theorem is not applicable in our case.

**V. Decomposable Tight Logic Programs**

It is important to realize that the completeness feature of Schaefer’s Dichotomy Theorem does not apply to Theorem 2. This is because our polynomial transformation from a tight logic program to its corresponding Boolean CSP-instance only ensures the program’s tractability if solving the corresponding Boolean CSP-instance is tractable. Indeed, it is not difficult to find a subclass of tight logic programs which do not satisfy any of conditions 1-5 but are still tractable. Consider the following tight logic program:

\[
\Pi:
\begin{align*}
    a & \leftarrow, \\
    b & \leftarrow, \\
    c & \leftarrow, \\
    c & \leftarrow b, d.
\end{align*}
\]

Clearly, this program does not satisfy any of conditions 1-5 in Theorem 2, but since it is a Horn program, its model can be computed in polynomial time.

Nevertheless, under certain conditions, we still can recognize a tight logic program’s tractability by using Theorem 2 even if this program does not satisfy any of conditions 1-5 in Theorem 2. To illustrate our idea, let us examine the above Horn program \( \Pi \) again. It is easy to observe that \( \Pi \) almost satisfies condition 5 in Theorem 2 except \( sd(c) = 2 > 1 \). However, we can actually decompose \( \Pi \) into two small programs:

\[
\Pi_1:
\begin{align*}
    a & \leftarrow, \\
    b & \leftarrow, \\
    c & \leftarrow a,
\end{align*}
\]

\[
\Pi_2:
\begin{align*}
    a & \leftarrow, \\
    b & \leftarrow, \\
    c & \leftarrow b, d.
\end{align*}
\]

Now note that both \( \Pi_1 \) and \( \Pi_2 \) satisfy condition 5 in Theorem 2. Furthermore, it is also observed that \( \Pi \)’s model can be obtained from some standard set operation on \( \Pi_1 \)’s and \( \Pi_2 \)’s models, (i.e. \( \{a, b, c\} = \{a, b\} \cup \{a, b\} \)). It is easy to prove that this subclass of tight logic programs like this type are still tractable. This example motivates us to specify a more general
subclass of tight logic programs whose tractability may be indirectly decided by applying Theorem 2.

Definition 3: Let $\Pi$ be an arbitrary tight logic program. $\Pi$ is decomposable if $\Pi$ can be polynomially decomposed to a sequence of tight logic programs $(\Pi_1, \ldots, \Pi_k)$ for some constant $k > 1$, such that $\Pi = \bigcup_{i=1}^{k} \Pi_i$ and $\Pi$ has a stable model if and only if each $\Pi_i$ ($1 \leq i \leq k$) has a stable model. In this case, $(\Pi_1, \ldots, \Pi_k)$ is called a decomposition of $\Pi$.

Directly from Definition 3 and Theorem 2, we can prove the following result.

Theorem 4: Suppose that $\Pi$ is a decomposable tight logic program and has a decomposition $(\Pi_1, \ldots, \Pi_k)$. Then $\Pi$ is a tractable program if each $\Pi_i$ ($1 \leq i \leq k$) satisfies one of conditions 1-5 in Theorem 2.

Now an immediate question is: under what conditions, a tight logic program is decomposable and how to obtain a decomposition of this program? Intuitively, if a program is decomposable, it may have more than one decompositions. As a trivial example, we can see that both $\{(a \leftarrow, b \leftarrow), (c \leftarrow)\}$ and $\{(a \leftarrow), \{b \leftarrow), \{c \leftarrow)\}$ are decompositions for program $\{a \leftarrow, b \leftarrow, c \leftarrow\}$. However, what our interest here is to study those decompositions that may contain useful information for deciding the tractability of the original program, e.g. as Theorem 4 indicates, whereas we fail to do so via Theorem 2 directly.

For this purpose, we first define the dependency graph for a program. Given a program $\Pi$, its dependency graph, denoted as $G(\Pi)$, is a directed graph $(\text{Atom}(\Pi), E)$, where $\text{Atom}(\Pi)$ is the set of vertices, and $E$ is the set of edges. An edge $(a, b) \in E$ iff there is a rule $r \in \Pi$ such that $a \in \text{pos}(r) \cup \text{neg}(r)$ and $b = \text{head}(r)$. Edge $(a, b)$ is labeled “positive” (+) if $a \in \text{pos}(r)$ and “negative” (−) if $a \in \text{neg}(r)$. We say that atom $a$ is positive in $G(\Pi)$ if for any path starting from $a$, no negative edge is contained in the path.

We say that $G(\Pi)$ contains a dependency subgraph of $\Pi$, if there is a subgraph in $G(\Pi)$: $(V_1, E_1)$ where $V_1 \subseteq \text{Atom}(\Pi)$ and $E_1 \subseteq E$, such that there does not exist an edge $(a, b) \in E$ with $a \in V_1$ and $b \in (\text{Atom}(\Pi) - V_1)$, or $a \in (\text{Atom}(\Pi) - V_1) \cup \text{Atom}(\Pi) - V_1$ and $b \in V_1$ (note that for each edge $(a, b) \in E_1$, $a, b \in V_1$). From this definition, we can see that $\Pi$’s dependency subgraph actually provides a partition on $\Pi$ where all rules related to the dependency subgraph are self-contained in the subgraph with no connection to other rules in $\Pi$. That is, for some $\Pi_1 \subseteq \Pi$, $V_1 = \text{Atom}(\Pi_1)$ and $\text{Atom}(\Pi_1) \cap \text{Atom}(\Pi - \Pi_1) = \emptyset$. Therefore, we can represent a dependency subgraph of $\Pi$ as $G(\Pi_1) = (\text{Atom}(\Pi_1), E_1)$, where $\Pi_1 \subseteq \Pi$. Now we have the following general result.

Theorem 5: Let $\Pi$ be a tight logic program and $G(\Pi)$ the dependency graph of $\Pi$. If $G(\Pi)$ contains dependency subgraphs $G(\Pi_1), \ldots, G(\Pi_k)$ where $\Pi = \bigcup_{i=1}^{k} \Pi_i$ ($k > 1$), then $(\Pi_1, \ldots, \Pi_k)$ is a decomposition of $\Pi$.

Example 3: Consider the following tight logic program:

$\Pi$: $r_1 : a \leftarrow b, \text{not } c$, $r_2 : d \leftarrow \text{not } e$, $r_3 : d \leftarrow f$, $r_4 : e \leftarrow$.

It is easy to check that $\Pi$ does not satisfy any of conditions 1-5 in Theorem 2. $\Pi$ has the following dependency graph $G(\Pi)$.

Clearly $G(\Pi)$ contains two dependency subgraphs of $\Pi$: $G(\Pi_1)$ and $G(\Pi_2)$ where $\Pi_1 = \{r_1\}$ and $\Pi_2 = \{r_2, r_3, r_4, r_5, r_6\}$. So according to Theorem 5, $(\Pi_1, \Pi_2)$ is a decomposition of $\Pi$. Also note that $\Pi_1$ and $\Pi_2$ satisfy conditions 1 and 2 in Theorem 2 respectively.

The following theorem provides another way to decompose a program which is especially useful to decide the tractability of a program that has a support depth greater than 1.

Theorem 6: Let $\Pi$ be a tight logic program and there is an atom $a \in \text{Atom}(\Pi)$ such that $\text{sd}(a) = d > 1$, that is $\Pi$ contains rules $r_1, \ldots, r_d$ with head$(r_i) = a$ ($1 \leq i \leq d$). If $a$ is positive in $\Pi$’s dependency graph $G(\Pi)$, then $\Pi$ has a decomposition $(\Pi_1, \ldots, \Pi_d)$, where each $\Pi_i$ ($1 \leq i \leq d$) is formed as $\Pi_i = \Pi - \{r_j : a \leftarrow \text{pos}(r_j), \text{not neg}(r_j), j \neq i\}$.

Example 4: Consider the following tight logic program:

$\Pi$: $r_1 : a \leftarrow b$, $r_2 : d \leftarrow c$, $r_3 : b \leftarrow \text{not } c$, $r_4 : c \leftarrow \text{not } b$, $r_5 : d \leftarrow a$.

Again, we observe that $\Pi$ does not satisfy any of conditions 1-5 in Theorem 2. However, since $\text{sd}(a) = 2 > 1, \text{head}(r_1) = \text{head}(r_2) = a$, and atom $a$ is positive in $\Pi$’s dependency graph $G(\Pi)$, according to Theorem 6, $(\Pi_1, \Pi_2)$ is a decomposition of $\Pi$, where $\Pi_1 = \{r_1, r_3, r_4, r_5\}$ and $\Pi_2 = \{r_2, r_3, r_4, r_5\}$. Then it is easy to see that both $\Pi_1$ and $\Pi_2$ satisfy condition 3 in Theorem 2.

Corollary 1: Every tight Horn logic program $\Pi$ with $\text{sd}(\Pi) > 1$ is decomposable and has a decomposition where each program in the decomposition satisfies condition 3 in Theorem 2.

VI. RELATED WORK AND CONCLUSION

Since Marek and Truszczynski proved that deciding whether a logic program has a stable model is NP-complete [18], the study on the computational issues of logic programs with stable model semantics has focused on two different methods: One is to investigate proper conditions under which the procedure of computing stable models may be optimized. For instance, various splitting techniques [16], [11]. The other approach is to identify tractable subclasses of logic programs so that efficient algorithms can be specifically developed for computing stable models of these tractable programs. In this paper, we followed the second approach to study the
tractability of tight logic programs by using CSP techniques. An interesting feature of our tractability characterizations is that they are purely syntax-based, and it only takes linear time to check whether a tight logic program satisfies one of these syntactic forms (see Theorem 2).

The idea of using CSP techniques to study the tractability of logic programs was previously considered by Ben-Eliyahu and Dechter. In [4], Ben-Eliyahu and Dechter studied the tractability of head cycle-free extended disjunctive logic programs (HEDLPs) and defined two tractable subclasses of HEDLPs which correspond to two tractable subsets of CSP. Basically, Ben-Eliyahu and Dechter showed that the stable model (answer set) existence of an HEDLP can be decided in polynomial time with respect to the clique width and cycle-cutset’s cardinality of the HEDLP associated interaction graph respectively. But these tractable subclasses are somehow not easy to be recognized because finding the clique width in a graph and finding the minimum size of cycle-cutset in a graph are NP-hard [7].

Ben-Eliyahu also proposed a tractable hierarchy for computing stable models [5]. In particular, Ben-Eliyahu presented a sequence of classes of programs \(\Omega_1, \Omega_2, \ldots\) such that if a program \(\Pi\) belongs to class \(\Omega_k\), then \(\Pi\) at most has \(k\) stable models and each of them can be found in time polynomial in the size of \(\Pi\) and \(k\). Further, Ben-Eliyahu proved that for an arbitrary program \(\Pi\), we can find the minimum \(k\) where \(\Pi\) belongs to \(\Omega_k\) in time polynomial in the size of \(\Pi\). However, we observe that in general these results do not have much help for achieving the tractable computation of the stable models of logic programs because for a given class \(\Omega_k\), \(k\) may be of the exponential size of programs belonging to \(\Omega_k\), and computing stable models of programs in \(\Omega_k\) may remain intractable in terms of the size of these programs.

On the other hand, Gottlob, Scarcello and Sideri recently also studied the tractability of logic programs from a fixed parameter complexity perspective [13]. They showed that all related computational problems of logic programs (including the stable model existence problem) are fixed-parameter tractable with respect to the feedback width of the undirected dependency graph of the logic program. However, this approach also has some restrictions. For example, it was observed that positive programs with large feedback width are not recognized to be tractable although they may be clearly tractable, and it is not known yet whether computing feedback vertex sets of size \(k\) is fixed-parameter tractable for directed graph [13].

Our work presented in this paper can be extended in several directions. Firstly, we may extend our results to other logic programs. For instance, we may investigate whether our tractability results can be improved to cover generalized tight logic programs which were studied by researchers from various aspects [2], [9], [17]. Also, instead of considering the completion of a logic program, we may define an alternative equivalent propositional theory for a large class of logic programs (e.g. [4]), and then obtain similar syntactic tractability characterizations by studying the tractable subclasses of the satisfiability problem through CSP techniques. The other direction is to identify more decomposable logic programs and extend this technique to first-order answer set programs [11], which will eventually simplify the computation of the underlying programs.

REFERENCES


Readers are referred to [4], [7] for the concepts of interaction graph, clique width, and cycle-cutset.