Propositional Semantics for Disjunctive Logic Programs

Rachel Ben-Eliyahu
Cognitive Systems Laboratory
Computer Science Department
University of California
Los Angeles, California 90024
rachel@cs.ucla.edu

Rina Dechter
Information & Computer Science
University of California
Irvine, California 92717
dechter@ics.uci.edu

Abstract

In this paper we study the properties of the class of head-cycle-free extended disjunctive logic programs (HEDLPs), which includes, as a special case, all nondisjunctive extended logic programs. We show that any propositional HEDLP can be mapped in polynomial time into a propositional theory such that each model of the latter corresponds to an answer set, as defined by stable model semantics, of the former. Using this mapping, we show that many queries over HEDLPs can be determined by solving propositional satisfiability problems.

Our mapping has several important implications: It establishes the NP-completeness of this class of disjunctive logic programs; it allows existing algorithms and tractable subsets for the satisfiability problem to be used in logic programming; it facilitates evaluation of the expressive power of disjunctive logic programs; and it leads to the discovery of useful similarities between stable model semantics and Clark's predicate completion.

1 Introduction

Stable model semantics for logic programs [BF87, Fin89, GL91] successfully bridges the gap between two lines of research — default reasoning and logic programming. Gelfond and Lifschitz [GL91] pointed out the need for explicit representation of negated information in logic programs and accordingly defined *extended logic programs* as those that use classical negation in addition to the *negation-as-failure* operator. They then generalized stable model semantics for the class of extended logic programs.

One advantage of stable model semantics is that it is closely related to the semantics of Reiter's default logic [Rei80], and within the framework of default logic an extended logic program may be viewed as a default theory with special features. This relationship implies that insights, techniques, and analytical results from default logic can be applied in logic programming, and vice versa. The work presented here puts this observation into practice. We use techniques developed for answering queries on default theories [BED91, BED92] to compute answer sets for disjunctive logic programs according to stable model semantics. We also show how stable model semantics can be given an interpretation in propositional logic.

Specifically, we show that a large class of *extended disjunctive logic programs* (EDLPs) can be compiled in polynomial time into a propositional theory such that each model of the latter corresponds to an answer set of the former. Consequently, query answering in such logic programs can be reduced to deduction in propositional logic. This reduction establishes the NP-completeness of various decision problems regarding query answering in such logic programs and suggests that any of a number of existing algorithms and heuristics known for solving satisfiability become applicable for computing answer sets. Moreover, known tractable classes for satisfiability lead to the identification of new tractable subsets of logic programs. As an example we introduce several new tractable subsets of logic programs which correspond to tractable subsets of *constraints satisfaction problems* (CSPs).

Aside from the computational ramifications, our translation provides an alternative and, we believe, clearer representation of stable model semantics, expressed in the familiar language of propositional logic. This facilitates evaluation and comparison of various semantic proposals and allows investigation of their expressive power. In particular, it highlights useful similarities between stable model semantics and Clark's predicate completion.

Our translation does not apply to the full class of EDLPs but only to a subclass (albeit a large one) of *head-cycle-free* extended disjunctive logic programs (HEDLPs). Note that this class contains any extended disjunction-free logic program, thus including the majority of logic programs.

The rest of the paper is organized as follows: In Section 2 we review the definition of HEDLPs and their stable model semantics and present some new characterizations of answer sets for such programs. Section 3 shows how an HEDLP can be mapped into a propositional theory and discusses the properties of our mapping. Section 4 illustrates four outcomes of the translation concerning the expressiveness of the language, its complexity, its tractability, and its relationship to Clark's predicate completion. In Section 5 we explain the relevance of our work to Reiter's default logic, in Section 6 we mention relevant work by others, and in Section 7 we provide concluding remarks. Most of the proofs appear in the Appendix.

2 Extended Disjunctive Logic Programs

Extended disjunctive logic programs (EDLPs) are disjunctive logic programs with two types of negation: negation by default and classical negation. They were introduced by Gelfond and Lifschitz [GL91], who defined an EDLP as a set of rules of the form

$$L_1 | ... | L_k \longleftarrow L_{k+1}, ..., L_{k+m}, \mathit{not}\ L_{k+m+1}, ..., \mathit{not}\ L_{k+m+n} \tag{1}$$

where each L_i is a literal and *not* is a negation-by-default operator. The symbol '|' is used instead of '$\vee$' to distinguish it from the classical $\vee$ used in classical logic. A literal *appears positive* in the body of a rule if it is not preceded by the *not* operator. A literal *appears negative* in the body of a rule if it is preceded by the *not* operator.[1]

Example 2.1 *Suppose we know that a baby called Adi was born. We also know that a baby, if there is no reason to believe that it is abnormal, is normal and that normal babies are either boys or girls. This information could be encoded in a disjunctive logic program as follows:*

[1]Note that *positive (negative) literal* and *a literal that appears positive (negative) in a body of a rule* denote two different things (see Example 2.1).

Baby(Adi) $\longleftarrow$
Normal_baby(x) $\longleftarrow$ *Baby(x), not Abnormal(x)*
Boy(x) | *Girl(x)* $\longleftarrow$ *Normal_baby(x).*

The literal Abnormal(x) appears negative in the body of the second rule. The literal Normal_baby(x) appears positive in the body of the third rule.

Gelfond and Lifschitz have generalized stable model semantics so that it can handle EDLPs. We next review this semantics with a minor modification: while Gelfond and Lifschitz's definition allows inconsistent answer sets, ours does not. Given a disjunctive logic program Π, the set of answer sets of Π under this modified semantics will be identical to the set of *consistent* answer sets under Gelfond and Lifschitz's original semantics. With slight changes the results in this paper all apply to their semantics as well.

First, Gelfond and Lifschitz define an *answer set* of an EDLP Π without variables and without the *not* operator. Let $\mathcal{L}$ stand for the set of grounded literals in the language of Π. A *context* of $\mathcal{L}$, or simply "context", is any subset of $\mathcal{L}$.

An *answer set* of Π is any minimal[2] context S such that

1. for each rule $L_1|...|L_k \longleftarrow L_{k+1}, ..., L_{k+m}$ in Π, if $L_{k+1}, ..., L_{k+m}$ is in S, then for some $i = 1, ..., k$ L_i is in S, and

2. S does not contain a pair of complementary literals.[3]$\Box$

Remark 2.2 *The definition of an answer set for this simplified subclass of EDLPs can be viewed as an extension of the definition of an S-model due to Rajasekar and Minker [MR90, Section 3.4], the difference being that S-models include also clauses and are not required to be minimal.*

Suppose Π is a variable-free EDLP. For any context S of $\mathcal{L}$, Gelfond and Lifschitz define Π^S to be the EDLP obtained from Π by deleting

1. all formulas of the form *not* L where $L \notin S$ from the body of each rule and

[2]Minimality is defined in terms of set inclusion.

[3]Under Gelfond and Lifschitz's semantics, this item would be "If S contains a pair of complementary literals, then $S = \mathcal{L}$".

2. each rule that has in its body a formula *not* L for some $L \in S$.

Note that Π^S has no *not* , so its answer sets were defined in the previous step. If S happens to be one of them, then we say that S is an *answer set* of Π. To apply the above definition to an EDLP with variables, we first have to replace each rule with its grounded instances.

Consider, for example, the grounded version of the program Π above:

Baby(Adi) $\longleftarrow$
Normal_baby(Adi) $\longleftarrow$Baby(Adi), *not* Abnormal(Adi)
Boy(Adi) | Girl(Adi) $\longleftarrow$Normal_baby(Adi).

We claim that the context $S = \{Baby(Adi), Normal\_baby(Adi), Girl(Adi)\}$ is an answer set. Indeed, Π^S is

Baby(Adi) $\longleftarrow$
Normal_baby(Adi) $\longleftarrow$Baby(Adi)
Boy(Adi) | Girl(Adi) $\longleftarrow$Normal_baby(Adi),

and S is a minimal set satisfying the conditions set above for programs without the *not* operator. Note that Π has two answer sets; the context $\{Baby(Adi), Normal\_baby(Adi), Boy(Adi)\}$ is the other answer set of Π.

We will assume from now on that all programs are grounded and that their dependency graphs have no infinitely decreasing chains[4]. The *dependency graph* of an EDLP Π, G_Π, is a directed graph where each literal is a node and where there is an edge from L to L' iff there is a rule in which L appears positive in the body and L' appears in the head[5]. An EDLP is *acyclic* iff its dependency graph has no directed cycles. An EDLP is *head-cycle free* (that is, an HEDLP) iff its dependency graph does not contain directed cycles that go through two literals that belong to the head of the same rule. Clearly, every acyclic EDLP is an HEDLP. We will also assume, without

[4]More formally: For a directed graph G, let $\overline{G}$ denote the graph obtained by reversing the direction of every arc in G. We say that G has an infinitely decreasing chain iff there is a node p in $\overline{G}$ such that there is an infinite acyclic directed path which starts at p. For example, the dependency graph of the grounded version of the program $\{Q(f(A)), P(x) \longleftarrow P(f(x))\}$ has an infinitely decreasing chain.

[5]Note that our dependency graph ignores the literals that appear negative in the body of the rule.

losing expressive power, that a literal appears only once in the head of any rule in the program.

We next present new characterizations of answer sets. The declarative nature of these characterizations allows for their specification in propositional logic in such a way that queries about answer sets can be expressed in terms of propositional satisfiability.

We first define when a rule is satisfied by a context and when a literal has a proof w.r.t. a program Π and a context S.

A context S satisfies the body of a rule δ iff each literal that appears positive in the body of δ is in S and each literal that appears negative in the body of δ is not in S. *A context S satisfies a rule* iff either it does not satisfy its body or it satisfies its body and at least one literal that appears in its head belongs to S.

A *proof* of a literal is a sequence of rules that can be used to derive the literal from the program. Formally, a literal L has a *proof* w.r.t. a context S and a program Π iff there is a sequence of rules $\delta_1, ..., \delta_n$ from Π such that

1. for each rule δ_i, one and only one of the literals that appear in its head belongs to S (this literal will be denoted $h_S(\delta_i)$),
2. $L = h_S(\delta_n)$,
3. the body of each δ_i is satisfied by S, and
4. δ_1 has an empty body and, for each $i > 1$, each literal that appears positive in the body of δ_i is equal to $h_S(\delta_j)$ for some $1 \leq j < i$.

Note that the above definition of a proof is an extension of the definition of a *default proof* that was introduced by Reiter [Rei80, Section 4]. The following theorem clarifies the concept of answer sets:

Theorem 2.3 *A context S is an answer set of an HEDLP Π iff*

1. *S satisfies each rule in Π,*
2. *for each literal L in S, there is a proof of L w.r.t Π and S, and*
3. *S does not contain a pair of complementary literals.* $\Box$

Remark 2.4 *The above theorem will not necessarily hold for programs having head cycles. Consider, for example, the program*

$$P|Q \longleftarrow$$
$$P \longleftarrow Q$$
$$Q \longleftarrow P.$$

The set $\{P, Q\}$ is an answer set but it violates condition 2 of the theorem, since neither P nor Q has a proof w.r.t. the answer set and the program.

Remark 2.5 *One might think that the requirement for a proof can be replaced with a requirement for minimality; in other words, that the following claim follows quite immediately from Theorem 2.3:*

"A context S is an answer set of an HEDLP Π iff S is a minimal context such that

1. S satisfies each rule in Π,
2. for each literal L in S, there is a rule δ in Π such that
   - (a) the body of δ is satisfied by S,
   - (b) L appears in the head of δ, and
   - (c) all the literals other than L in the head of δ are not in S,

   and
3. S does not contain a pair of complementary literals."

However, the following example by Brewka and Konolige [BK] demonstrates that this is not the case:

Example 2.6 (minimality cannot replace the requirement for a proof) *Consider the logic program*

$$a \longleftarrow a$$
$$b \longleftarrow not\ a.$$

The contexts $\{a\}$ and $\{b\}$ are both minimal sets satisfying conditions 1-3 of the claim above, but only $\{b\}$ is an answer set of this program.

So is there an easy way to verify that each literal has a proof? It turns out that for an acyclic EDLP the task is easier:

Theorem 2.7 *A context S is an answer set of an acyclic EDLP Π iff*

1. *S satisfies each rule in Π,*
2. *for each literal L in S, there is a rule δ in Π such that*
   - *(a) the body of δ is satisfied by S,*
   - *(b) L appears in the head of δ, and*
   - *(c) all the literals other than L in the head of δ are not in S,*

   and
3. *S does not contain a pair of complementary literals.* $\Box$

To identify an answer set when Π is *cyclic*, we need to assign indexes to literals that share a cycle in the dependency graph.

Theorem 2.8 *A context S is an answer set of an HEDLP Π iff*

1. *S satisfies each rule in Π,*
2. *there is a function $f : \mathcal{L} \mapsto N^+$ such that, for each literal L in S, there is a rule δ in Π such that*
   - *(a) the body of δ is satisfied by S,*
   - *(b) L appears in the head of δ,*
   - *(c) all literals in the head of δ other than L are not in S, and,*
   - *(d) for each literal L' that appears positive in the body of δ, $f(L') < f(L)$,*

   and
3. *S does not contain a pair of complementary literals.* $\Box$

While the rest of the paper refers only to finite grounded logic programs (i.e., propositional logic programs), we would like to emphasize that Theorems 2.3-2.8 are valid for infinite grounded logic programs as well.

The above characterizations of answer sets are very useful. In addition to giving us alternative definitions of an answer set, they facilitate a polynomial time compilation of any finite[6] grounded HEDLP into a propositional theory, such that there is a one-to-one correspondence between answer sets of the former and models of the latter. The merits of this compilation will be illustrated in the sequel.

3 Compiling Disjunctive Logic Programs into a Propositional Theory

Each answer set of a given logic program represents a possible world compatible with the information expressed in the program. Hence, given an EDLP Π, the following queries might come up:

Existence: Does Π have an answer set? If so, find one or all of them.

Set-Membership: Given a context V, is V contained in *some* answer set of Π?

Set-Entailment: Given a context V, is V contained in *every* answer set of Π?

Disjunctive-Entailment: Given a context V, is it true that for each answer set S of Π there is at least one literal in V that belongs to S? (This amounts to asking whether the logic program Π implies the disjunction of the literals in V.)

In this section we will show algorithms that translate a finite HEDLP Π into a propositional theory T_Π such that the above queries can be expressed as satisfiability problems on T_Π.

The propositional theory T_Π is built upon a new set of symbols $\mathcal{L}_\Pi$ in which there is a new symbol I_L for each literal L in $\mathcal{L}$. Formally,

[6]The translation we provide is also appropriate for the infinite case where each cycle in the dependency graph is finite and each literal appears in the head of a finite number of rules.

$$\mathcal{L}_\Pi = \{I_L | L \in \mathcal{L}\}.$$

Intuitively, each I_L stands for the claim "The literal L is *I*n the answer set", and each valuation of $\mathcal{L}_\Pi$ represents a context, which is the set of all literals L such that I_L is assigned **true** in the valuation. What we are looking for, then, is a theory over the set $\mathcal{L}_\Pi$ such that each model of the theory represents a context that is an answer set of Π.

Consider algorithm *translate-1* below, which translates an HEDLP Π into a propositional theory T_Π.

translate-1(Π)

1. For each body-free rule $L_1|...|L_k \longleftarrow$ in Π, add $I_{L_1} \vee ... \vee I_{L_k}$ into T_Π.
2. For each rule

   $$L_1|...|L_k \longleftarrow L_{k+1}, ..., L_{k+m}, \mathit{not}\ L_{k+m+1}, ..., \mathit{not}\ L_{k+m+n} \tag{2}$$

   with no empty body add

   $$I_{L_{k+1}} \wedge ... \wedge I_{L_{k+m}} \wedge \neg I_{L_{k+m+1}} \wedge ... \wedge \neg I_{L_{k+m+n}} \longrightarrow I_{L_1} \vee ... \vee I_{L_k}$$

   into T_Π.
3. For a given $L \in \mathcal{L}$, let S_L be the set of formulas of the form

   $$I_{L_{k+1}} \wedge ... \wedge I_{L_{k+m}} \wedge \neg I_{L_{k+m+1}} \wedge ... \wedge \neg I_{L_{k+m+n}} \wedge \neg I_{L_1} \wedge ... \wedge \neg I_{L_{j-1}} \wedge \neg I_{L_{j+1}} \wedge ... \wedge \neg I_{L_k}$$

   where there is a rule (2) in Π in which L appears in the head as L_j.

   For each L in $\mathcal{L}$ such that the rule "$L \longleftarrow$" is not in Π add to T_Π the formula $I_L \longrightarrow [\vee_{\alpha \in S_L} \alpha]$ (note that if $S_L = \emptyset$ we add $I_L \longrightarrow$**false** to T_Π).
4. For each two complementary literals L, L' in $\mathcal{L}$, add $\neg I_L \vee \neg I_{L'}$ to T_Π.
   $\square$

The reader has probably noticed that the propositional theory T_Π, produced by the above algorithm, simply states the conditions of Theorem 2.7 in propositional logic: The first and second steps of algorithm *translate-1* express condition 1 of the theorem, the third step expresses condition 2, and the last step describes condition 3. Hence:

Theorem 3.1 *Procedure* translate-1 *transforms an acyclic EDLP* Π *into a propositional theory* T_Π *such that* θ *is a model for* T_Π *iff* $\{L|\theta(I_L) = \textbf{\textit{true}}\}$ *is an answer set for* Π. $\Box$

What if our program is cyclic? Can we find a theory such that each of its models corresponds to an answer set? Theorem 2.8 suggests that we can do so by assigning indexes to the literals.

When we deal with finite logic programs, the fact that each literal is assigned an index in the range $1...n$ for some n and the requirement that an index of one literal will be lower than the index of another literal can be expressed in propositional logic. Let $\#L$ stand for "L is associated with one and only one integer between 1 and n", and let $[\#L_1 < \#L_2]$ stand for "The number associated with L_1 is less than the number associated with L_2". These notations are shortcuts for formulas in propositional logic that express these assertions (see Appendix).

The size of the formulas $\#L$ and $[\#L_1 < \#L_2]$ is polynomial in the range of the indexes we need. It is clear that we need indexes only for literals that reside on cycles in the dependency graph. Furthermore, since we will never have to solve cyclicity between two literals that do not share a cycle, the range of the index variables is bounded by the maximum number of literals that share a common cycle. In fact, we can show that the index variable's range can be bounded further by the maximal length of an acyclic path in any *strongly connected component* in G_Π (the dependency graph of Π).

The strongly connected components of a directed graph are a partition of its set of nodes such that, for each subset C in the partition and for each $x, y \in C$, there are directed paths from x to y and from y to x in G. The strongly connected components can be identified in linear time [Tar72]. Thus, once again we realize that if the HEDLP is acyclic, we do not need any indexing.

The above ideas are summarized in the following theorem, which is a restricted version of Theorem 2.8 for the class of finite HEDLPs.

Theorem 3.2 *Let Π be a finite HEDLP and let r be the length of the longest acyclic directed path in any component of G_Π. A context S is an answer set of an HEDLP Π iff*

1. *S satisfies each rule in Π,*
2. *there is a function $f : \mathcal{L} \mapsto 1, ..., r$ such that, for each literal L in S, there is a rule δ in Π such that*
   - (a) *the body of δ is satisfied by S,*
   - (b) *L appears in the head of δ,*
   - (c) *all literals other than L in the head of δ are not in S, and,*
   - (d) *for each literal L' that appears positive in the body of δ and shares a cycle with L in the dependency graph of Π, $f(L') < f(L)$,*

   and
3. *S does not contain a pair of complementary literals.* $\Box$

Procedure *translate-2* expresses the conditions of Theorem 3.2 in propositional logic. Its input is any finite HEDLP Π, and its output is a propositional theory T_Π whose models correspond to the answer sets of Π. T_Π is built over the extended set of symbols $\mathcal{L}_\Pi{}' = \mathcal{L}_\Pi \bigcup \{[L = i] | L \in \mathcal{L}, 1 \leq i \leq r\}$, where r is the size of the longest acyclic path in any component of G_Π. Steps 1, 2, and 4 of *translate-2* are identical to steps 1, 2, and 4 of *translate-1*, so we will show only step 3.

translate-2(Π)-step 3

3. Identify the strongly connected components of G_Π. For each literal L that appears in a component of size > 1, add $\#L$ to T_Π.

For a given $L \in \mathcal{L}$, let S_L be the set of all formulas of the form

$$I_{L_{k+1}} \wedge ... \wedge I_{L_{k+m}} \wedge \neg I_{L_{k+m+1}} \wedge ... \wedge \neg I_{L_{k+m+n}} \wedge \neg I_{L_1} \wedge ... \wedge \neg I_{L_{j-1}} \wedge \neg I_{L_{j+1}} \wedge ... \wedge \neg I_{L_k} \\ \wedge [\#L_{k+1} < \#L] \wedge ... \wedge [\#L_{k+r} < \#L]$$

such that there is a rule in Π

$$L_1 | ... | L_k \longleftarrow L_{k+1}, ..., L_{k+m}, not\ L_{k+m+1}, ..., not\ L_{k+m+n}$$

in which L appears in the head as L_j and $L_{k+1},...,L_{k+r}$ ($r \leq m$) are in L's component.

For each L in $\mathcal{L}$ such that the rule "$L\longleftarrow$" is not in Π, add to T_Π the formula $I_L \longrightarrow [\vee_{\alpha \in S_L} \alpha]$ (note that if $S_L = \emptyset$ we add $I_L \longrightarrow$**false** to T_Π). □

Note that if *translate-2* gets as an input an acyclic HEDLP it will behave exactly the same as *translate-1*, thus it is a generalization of *translate-1*. The following proposition states that the algorithm's time complexity and the size of the resulting propositional theory are both polynomial:

Proposition 3.3 *Let* Π *be an HEDLP. Let* $|\Pi|$ *be the number of rules in* Π*,* n *be the size of* $\mathcal{L}$*, and* r *the length of the longest acyclic path in any component of* G_Π*. Algorithm* translate-2 *runs in time* $O(|\Pi|n^2r^2)$ *and produces* $O(n + |\Pi|)$ *sentences of size* $O(|\Pi|nr^2)$*.*

Proof: Let m be the maximal number of literals in a rule and t the maximal number of rules with a certain literal L in the head. Steps 1 and 2 of algorithm *translate-2* take $O(|\Pi|m)$ time and produce $O(|\Pi|)$ sentences of size $O(m)$. In step 3, for each literal L, S_L includes at most t formulas of size $O(mr^2)$ (remember that we can express inequality in a propositional sentence of size $O(r^2)$). Since $|\mathcal{L}| = n$, step 3 takes $O(ntmr^2)$ time and produces $O(n)$ sentences of size $O(tmr^2)$. Step 4 takes $O(n)$ time and produces $O(n)$ sentences of size $O(2)$. So the entire algorithm takes $O(ntmr^2 + |\Pi|m) \leq O(|\Pi|n^2r^2)$ time and produces $O(n + |\Pi|)$ sentences of size $O(tmr^2) \leq O(|\Pi|nr^2)$. □

The following theorems summarize the properties of our transformation. In all of them, T_Π is the set of sentences resulting from translating a given HEDLP Π using *translate-2* (or *translate-1* when the program is acyclic).

Theorem 3.4 *Let* Π *be an HEDLP. If* T_Π *is satisfiable and if* θ *is a model for* T_Π *, then* $\{L|\theta(I_L) = \textbf{\textit{true}}\}$ *is an answer set for* Π*.* □

Theorem 3.5 *If* S *is an answer set for an HEDLP* Π*, then there is a model* θ *for* T_Π *such that* $\theta(I_L) = \textbf{\textit{true}}$ *iff* $L \in S$*.* □

Corollary 3.6 *An HEDLP* Π *has an answer set iff* T_Π *is satisfiable.* □

Corollary 3.7 *A context* V *is contained in some answer set of an HEDLP* Π *iff there is a model for* T_Π *that satisfies the set* $\{I_L|L \in V\}$*.* □

Corollary 3.8 *A literal L is in every answer set of an HEDLP Π iff every model for T_{Π} satisfies I_L.* $\Box$

Corollary 3.9 *Given a context V, each answer set S of an HEDLP Π contains at least one literal from V iff every model for T_{Π} satisfies $\vee_{L \in V} I_L$.*

The above theorems suggest that we can first translate a given HEDLP Π to T_{Π} and then answer queries as follows: To test whether Π has an answer set, we test satisfiability of T_{Π}; to see whether a set V of literals is a member in some answer set, we test satisfiability of $T_{\Pi} \bigcup \{I_L | L \in V\}$; to see whether V is included in every answer set, we test whether $T_{\Pi} \models \wedge_{L \in V} I_L$; and to check whether Π implies the disjunction $L_1 \vee ... \vee L_r$, we check whether $T_{\Pi} \models \{\vee_{L \in V} I_L\}$.

Example 3.10 *Consider again the program Π from the previous section ("Ba" stands for "Baby(Adi)", "Bo" stands for "Boy(Adi)", and each of the other literals is represented by its initial):*

$$\begin{aligned} Ba &\longleftarrow \\ N &\longleftarrow Ba, \mathit{not}\ A \\ Bo|G &\longleftarrow N. \end{aligned}$$

The theory T_{Π}, produced by algorithm translate-2 *(and also by algorithm* translate-1*, since this program is acyclic), is:*

$\{$*(following step 1)*

$$I_{Ba}$$

(following step 2)

$$I_{Ba} \wedge \neg I_A \longrightarrow I_N,\ I_N \longrightarrow I_{Bo} \vee I_G$$

(following step 3)

$$I_N \longrightarrow I_{Ba} \wedge \neg I_A,\ I_{Bo} \longrightarrow I_N \wedge \neg I_G,\ I_G \longrightarrow I_N \wedge \neg I_{Bo},\ \neg I_A$$

(no sentences will be produced in step 4 since there are no complementary literals in $\mathcal{L}$)

$\}$.

This theory has exactly two models (we mention only the atoms to which the model assigns ***true****):*

1. *$\{I_{Ba}, I_N, I_G\}$, which corresponds to the answer set { Baby(Adi), Normal_baby(Adi), Girl(Adi) }, and*

2. *$\{I_{Ba}, I_N, I_{Bo}\}$, which corresponds to the answer set { Baby(Adi), Normal_baby(Adi), Boy(Adi) }.* $\square$

4 Outcomes of Our Translation

4.1 NP-completeness

It has been shown that there is a close relationship between default theories and logic programs interpreted by stable model semantics [BF87, GL91]. This observation allows techniques and complexity results obtained for default logic to be applied to logic programming, and vice versa. For example, the complexity results obtained by Kautz and Selman [KS91] and Stillman [Sti90] for default logic show that the satisfiability problem is polynomially reducible to deciding answer set existence and membership in a subset of extended logic programs[7] and that entailment in propositional logic is polynomially reducible to entailment for a subset of extended logic programs. These results establish the NP-hardness of the existence and membership problems and the co-NP-hardness of the entailment problem for the class HEDLPs.

In view of these results, the polynomial transformation to satisfiability that we have presented in the last section implies the following:

Corollary 4.1 *The existence problem for the class HEDLPs is NP-complete.*

Corollary 4.2 *The membership problem for the class HEDLPs is NP-complete.*

Corollary 4.3 *The entailment problem for the class HEDLPs is co-NP-complete.*

Note that the above results extend the results of Marek and Truszczyński [MT91a, Section 6], who showed that the existence problem for the class of normal finite propositional logic programs is NP-complete.

[7]We remind the reader that the class of extended logic programs (ELPs) includes all programs that do not allow disjunction in the heads of rules and therefore the class ELPs are a subset of the class HEDLPs.

Until recently, the question of whether stable model semantics for the class of *all* extended disjunctive logic programs (the class EDLPs) can be expressed in propositional logic has been regarded as an open problem. However, Eiter and Gottlob [EG92] have shown that the existence and set-membership problems for the class EDLPs is Σ_2^P complete and that set entailment for this class is Π_2^P complete. Therefore, Eiter and Gottlob conclude:

Corollary 4.4 *[EG92] Unless $\Sigma_2^P = NP$ (respectively, $\Pi_2^P = coNP$), stable model semantics of finite propositional EDLPs (or even EDLPs with no type of negation) cannot be expressed in propositional logic in polynomial time.*

4.2 Tractability

The results obtained in the last subsection suggest that in general the decision problems posed in the beginning of Section 3 are NP-hard. Our mapping of HEDLPs into propositional theories suggests a new dimension along which tractable classes can be identified. Since our transformation is tractable, any subset of HEDLPs that is mapped into a tractable subset of satisfiability is tractable as well.

Among other possibilities, we can apply techniques developed in the *constraints satisfaction* literature (for a survey, see [Dec92]) to solve satisfiability and to identify tractable classes.

A *constraint satisfaction problem (CSP)* consists of a set of n variables $X_1, ..., X_n$, their respective domain values $R_1, ..., R_n$, and a set of constraints $C_{i_1}, ..., C_{i_t}$. A constraint $C_i(X_{i_1}, ..., X_{i_j})$ is a subset of the Cartesian product $R_{i_1} \times ... \times R_{i_j}$ that specifies which values of the variables are compatible with each other. A solution is an assignment of a value to each variable that satisfies all the constraints, and the tasks are to determine whether a solution exists, to find one or all solutions, and so on.

The satisfiability of a propositional theory can be regarded as a CSP. The set of variables is the set of propositional symbols, the domain of each variable is the set $\{\mathbf{true}, \mathbf{false}\}$, and each constraint is the truth table associated with one sentence of the theory.

Example 4.5 *The theory $\{(A \vee B), (B \vee C \vee D) \wedge (A \vee D)\}$ is a CSP where the variables are the symbols A, B, C, D, their respective domains are the set $\{$**true**, **false**$\}$, and there are three constraints, one for each clause. The*

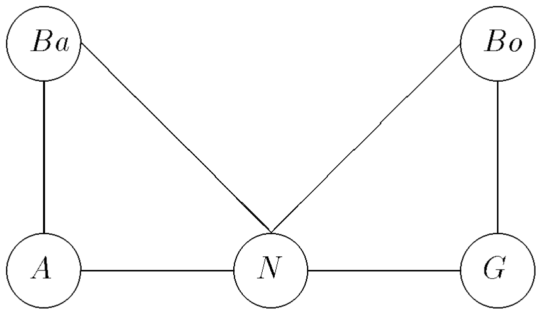

Figure 1: Constraint graph

constraint associated with the clause $A \vee B$ is given by

$$C(A, B) = \{(\boldsymbol{true}, \boldsymbol{false}), (\boldsymbol{true}, \boldsymbol{true}), (\boldsymbol{false}, \boldsymbol{true})\}.$$

Some constraint satisfaction techniques exploit the structure of the CSP through its *constraint graph.* In a *constraint graph* of a CSP, variables are represented by nodes and arcs connect any two nodes residing in the same constraint. Similarly, the constraint graph of a propositional theory associates a node with a propositional symbol and arcs connect nodes appearing in the same propositional sentence.

Example 4.6 *Consider the theory T_Π produced in Example 3.10. The constraint graph of this problem is shown in Figure 1. In this graph each variable I_L is represented by L. Note that $\neg I_G$ and I_G are represented by the same node, G.*

Notice that a constraint graph is an abstraction of a formula. Namely, it indicates that some relationship between the connected variables must be enforced but it says nothing about the exact structure of the formula. Consequently, many different theories can be associated with the same constraint graph. For instance, the theory $\{(I_{Ba} \vee I_A \vee I_N) \wedge (I_{Bo} \vee I_N \vee I_G)\}$ would also have the constraint graph in Figure 1. The value of the graph is in highlighting common dependency features between theories through graph connectivity and in allowing algorithms that exploit graph properties to find a solution to the CSP problem. An extreme, but useful case is when the constraint graph is a tree (no cycles). In this case, we can find whether there is a solution in time $O(nk^2)$ where k is the maximal domain size and n is the number of variables [DP88]. The efficiency by which a tree problem can be solved serves as the basis for identifying many topologically based tractable classes of CSPs.

It was shown that various other graph parameters are indicators of the complexity of solving CSPs. These include the *clique width*, the *size of the cycle-cutset*, the *depth of a depth-first-search spanning tree* of the graph, and the *size of the non-separable components* [Fre85, DP88, Dec90]. It can be shown that the worst-case complexity of solving CSPs is polynomially bounded by any one of these parameters. (For definitions and summary see [Dec92]).

While determining the minimum value of most of these parameters (e.g., the minimum size of a cycle-cutset in a graph) is NP-hard, a reasonable bound can be recognized polynomially using efficient graph algorithms. Consequently, these parameters can be used for assessing tractability ahead of time.

In this paper we choose to demonstrate the effectiveness of graph-based methods through two algorithms: one known as *tree-clustering* [DP89] and the other as *cycle-cutset* decomposition [Dec90]. The algorithms use different approaches for extending the class of tree-like problems.

The following two subsections briefly describe the algorithms and quote relevant results. To avoid duplicating existing literature, we give the intuition and flavor of the algorithms and refer the reader to the relevant articles for details.

4.3 Tree-clustering

If many queries are to be processed over the same set of constraints, it may be advisable to invest effort and memory space in restructuring the problem in order to facilitate more efficient query answering routines. Tree-clustering is such a restructuring technique. It guarantees that a large number of queries can be answered swiftly, either by sequential backtrack-free procedures or by distributed constraint propagation methods. The general idea is to utilize the merit of tree topologies in non-tree CSPs by forming clusters of variables such that the interactions between the clusters in the constraint graph are tree-structured and then to solve the problem by the efficient tree algorithm. This amounts to:

1. *clustering,* namely, deciding which variables should be grouped together,
2. *solving cluster,* that is, finding and listing the internally consistent values in each cluster, and
3. *solving the tree,* namely, processing each cluster as singleton variables in a tree.

The first step is performed by a linear graph algorithm; the second step requires solving (by some brute-force algorithm) the subproblems defined by each cluster. This step is worst-case exponential in the size of the resulting clusters. The two steps together produce a tree-like CSP that can now be solved linearly by the tree algorithm.

It can be shown that the most costly aspect of the three steps is generating and keeping all the solutions of each cluster (step 2). Consequently, the structuring process in *tree-clustering* is equipped with heuristics for generating clusters that are as small possible. Finding the optimal clustering scheme is known to be NP-hard [ACP87], but good tractable approximations are available.

Step 1 works by embedding the constraint graph of the problem within a *chordal graph*, because the maximal cliques of a chordal graph interact in a tree-like fashion. It can be shown that the size of the clusters generated in this way can be bounded by the *clique width* of the graph.

Definition 4.7 (clique width) *A* chord *of a cycle is an arc connecting two nonadjacent nodes in the cycle. A graph is* chordal *iff every cycle of length at least* 4 *has a chord. The* clique width *of a graph G is the minimum size of a maximal clique in any chordal graph that embeds G*[8]*.*

[8]A graph $G\prime$ embeds graph G iff $G \subseteq G\prime$ when we view graphs as sets of nodes and arcs.

Example 4.8 *The graph in Figure 1 is already chordal, so its clique width is equal to the size of its maximal clique, which is* 3*.*

It was shown that a CSP whose constraint graph has clique width of size q can be embedded in a tree of cliques of maximum size q. Therefore, the complexity of tree-clustering depends on the clique width of the constraint graph of the problem:

Theorem 4.9 *[DP89] The complexity of tree-clustering is $O(pnk^q)$, where p is the size of the constraints given as input, n is the number of variables, q is the size of the clique width, and k the maximal number of values each variable can assume.*

For a propositional theory T, this means that if the clique width of its constraint graph is q, then we can decide T's satisfiability and find one of its models (if there is one) in time $O(|T|n2^q)$, where $|T|$ is the size of the theory.

We will next show how these results apply to our class of logic programs. We will characterize the tractability of HEDLPs as a function of the topology of their *interaction graphs*. The interaction graph of an HEDLP Π will be defined so as to coincide with the constraint graph of its corresponding propositional theory T_Π.

Definition 4.10 (interaction graph) *The* interaction graph *of an HEDLP Π is an undirected graph where each literal in the language of Π is associated with a node and for every literal L, the set of all literals that appear in rules that have L in their heads are connected as a clique.*

Lemma 4.11 *The interaction graph of an HEDLP Π is identical to the constraint graph of its propositional theory T_Π.*

Proof: Immediate.

Example 4.12 *The interaction graph of the logic program Π from Example 3.10 is shown in Figure 1. Note that it has exactly the same structure as the constraint graph of T_Π.*

In the following theorems, $|\Pi|$ stands for the number of rules in Π and r stands for the number of literals along the longest acyclic directed path in any component of the dependency graph. Note that if the theory is acyclic, $r = 1$.

Theorem 4.13 *For an HEDLP whose interaction graph has a clique width q, existence, membership, and entailment can be decided in $O(n^4(2r)^{q+2})$ steps if $n \geq |\Pi|$ and in $O(|\Pi|^4(2r)^{q+2})$ steps if $|\Pi| \geq n$.* $\Box$

Proof: We answer the above queries in two stages. We first translate the logic program Π into a propositional theory T_Π and then, using Corollaries 3.6-3.8, answer the above queries by solving satisfiability of a theory that is about the same size as T_Π and has the same constraint graph. By Proposition 3.3, the translation stage takes $O(|\Pi|n^2r^2)$ time and the size of T_Π is $O(|\Pi|nr^2(|\Pi| + n))$. For the next stage of solving satisfiability, we will consider two cases:

Case Π *is acyclic*: In this case, the interaction graph has exactly the same topology as the constraint graph of T_Π, and hence, from Theorem 4.9, we can determine whether T_Π is satisfiable and can find a model for T_Π (if there is one) in time $O(|T_\Pi|n2^q)$. Consequently, the overall complexity of answering these queries when the theory is acyclic is $O(|\Pi|n^2 + |T_\Pi|n2^q)$.

Case Π *is cyclic*: In this case, we will solve satisfiability by representing the problem as a general CSP. Each literal will represent a variable, as before, but this time the domain of each variable will be a pair (t, m), where t can assume the value **true** or **false** and m is a number in the range $1, ..., r$ (intuitively, a variable having a value (t, m) has truth value t and index m). So the domain of each variable is of size $\leq 2r$. The sentences of T_Π represent the set of constraints. The interaction graph of Π has the same topology as the constraint graph of its CSP. Note that the fact that the domains are larger is not reflected in the constraint graph. Again, from Theorem 4.9, this problem can be solved in time $O(|T_\Pi|nr^q)$. So the overall complexity of answering these queries for cyclic theories is $O(|\Pi|n^2r^2 + |T_\Pi|n(2r)^q) \leq O(|\Pi|^2n^2(2r)^{q+2} + |\Pi|n^3(2r)^{q+2})$. $\Box$

We believe, however, that tree-clustering is especially useful for programs whose interaction graphs have a repetitive structure. Programs for temporal

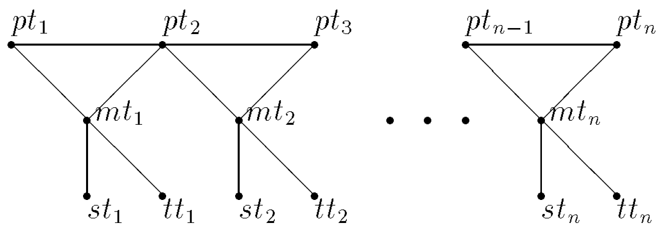

Figure 2: Constraint graph for Example 4.9

reasoning, where the temporal persistence principle causes the knowledge base to have a repetitive structure, are a good example. We next demonstrate the usefulness of tree-clustering for answering queries on such programs.

Example 4.14 *Suppose I park my car in the parking lot at time t_1. If it was not removed from the lot by being stolen or towed between time t_1 and t_n, I expect my car to be there during any time t_i between t_1 and t_n. This can be expressed in the following propositional logic program, where we have n rules of the form*

$$\begin{aligned} parked(t_{i+1}) &\longleftarrow parked(t_i),\ not\ moved(t_i) \\ moved(t_i) &\longleftarrow stolen(t_i) \\ moved(t_i) &\longleftarrow towed\text{-}away(t_i) \end{aligned}$$

The interaction graph of this program is shown in Figure 2. We can see that this graph is chordal and its clique width is 3. Thus, as n, the number of time intervals, increases, the complexity of answering queries about coherence, membership, and entailment using the tree-clustering algorithm increases polynomially.

4.4 Cycle-cutset decomposition

The *cycle-cutset scheme* is, as the name suggests, a decomposition method based on identifying a *cycle-cutset* of a graph, that is, a set of nodes that, once removed, would render the constraint graph cycle-free. This method exploits the fact that variable instantiation changes the effective connectivity of the constraint graph [Dec90].

When the group of instantiated variables constitutes a cycle-cutset, the remaining network is cycle-free and can be solved by the *tree algorithm*. In most practical cases, it would take more than a single variable to cut all the cycles in the graph. Thus, a general way of solving a problem whose constraint graph contains cycles is to find a consistent instantiation of the variables in a cycle-cutset and solve the remaining problem by the tree algorithm. If a solution to the restricted problem is found, then a solution to the entire problem is at hand. If not, another instantiation of the cycle-cutset variables should be considered until a solution is found. Thus the complexity of the *cycle-cutset* algorithm is exponentially related to the size of its cycle-cutset. It has been shown [Dec90] that a CSP whose constraint graph has a cycle-cutset of size c can be solved in time $O(pnk^{c+2})$. Consequently, if the cycle-cutset of the constraint graph of the propositional theory T is of size c, we can decide its satisfiability and find a model for the theory (if there is one) in time $O(|T|n2^c)$.

Example 4.15 *If we remove the node N from the graph in Figure 1, we are left with a cycle-free graph. So the cycle-cutset of this constraint graph is of size 1, and so this problem belongs to a class of problems for which cycle-cutset decomposition is efficient.*

The next theorem extends this idea to the interaction graph of an HEDLP. We omit the proof since it is similar to the proof of Theorem 4.13.

Theorem 4.16 *For an HEDLP (D, W) whose interaction graph has a cycle-cutset of cardinality c, existence, membership, and entailment can be decided in $O(n^4(2r)^{c+4})$ steps if $n \geq |\Pi|$ and in $O(|\Pi|^4(2r)^{c+4})$ steps if $|\Pi| \geq n$.*

4.5 Expressiveness

Are disjunctive rules really more expressive than nondisjunctive rules? Can we find a nondisjunctive theory for each disjunctive theory such that they have the same answer sets/extensions? This question has been raised by Gelfond *et al.* [GPLT91]. They consider translating a disjunctive logic program Π into a nondisjunctive program Π' by replacing each rule of the form (1) (see page 3 above) with k new rules:

$$L_1 \longleftarrow L_{k+1}, ..., L_{k+m}, not\ L_{k+m+1}, ..., not\ L_{k+m+n}, not\ L_2, ..., not\ L_k,$$

$$\vdots$$

$$L_k \longleftarrow L_{k+1}, ..., L_{k+m}, not\ L_{k+m+1}, ..., not\ L_{k+m+n}, not\ L_1, ..., not\ L_{k-1}.$$

Gelfond *et al.* show that each extension of Π' is also an extension of Π, but not vice versa. They gave an example where Π has an extension while Π' does not. So, in general, Π' will not be equivalent to Π. We can show, however, that equivalence does hold for HEDLPs.

Theorem 4.17 *Let Π be an HEDLP. S is an answer set for Π iff it is an answer set for Π'.* $\Box$

The reader can verify the above theorem by observing that our translation will yield the same propositional theory for both Π and Π'. The theorem implies that under stable model semantics no expressive power is gained by introducing disjunction unless we deal with a special case of recursive disjunctive logic programs, namely, disjunctive logic programs that use rules that are not head-cycle-free.

4.6 Relation to Clark's predicate completion

Clark [Cla78] made one of the first attempts to give meaning to logic programs with negated atoms in a rule's body ("normal programs"). He shows how each normal program Π can be associated with a first-order theory $COMP(\Pi)$, called its *completion.* His idea is that when a programmer writes a program Π, the programmer actually has in mind $COMP(\Pi)$, and thus all queries about the program should be evaluated with respect to $COMP(\Pi)$. So a formula Q is implied by the program iff $COMP(\Pi) \models Q$.

For the comparison between Clark's work and ours, we consider only normal propositional programs, that is, a set of rules of the form

$$Q \longleftarrow P_1, ..., P_n, not R_1, ..., not R_m \tag{3}$$

where Q, $P_1, ..., P_n$, and $R_1, ..., R_m$ are atoms.

Given a propositional logic program Π, $COMP(\Pi)$ is obtained in two steps:

Step 1: Replace each rule of the form (3) with the rule

$$Q \longleftarrow P_1 \wedge ... \wedge P_n \wedge \neg R_1 \wedge ... \wedge \neg R_m. \tag{4}$$

Step 2: For each symbol Q, let Support(Q) denote the set of all clauses with Q in the head. Suppose Support(Q) is the set

$$Q \longleftarrow Body_1 \tag{5}$$
$$. \tag{6}$$
$$. \tag{7}$$
$$. \tag{8}$$
$$Q \longleftarrow Body_k. \tag{9}$$

Replace it with a single sentence,

$$Q \longleftrightarrow Body_1 \vee ... \vee Body_k. \tag{10}$$

Note two special cases: If "$Q \longleftarrow$" in Support(Q), simply replace Support(Q) by Q. If Support(Q) is empty, replace it with $\neg Q$.

Example 4.18 *Consider the following program* Π*:*

$$P \longleftarrow Q, notR \tag{11}$$
$$P \longleftarrow V \tag{12}$$
$$R \longleftarrow S \tag{13}$$
$$V \longleftarrow \tag{14}$$

The completion of Π *is the following propositional theory:*

$$P \longleftrightarrow [Q \wedge \neg R] \vee V \tag{15}$$
$$R \longleftrightarrow S \tag{16}$$
$$V \tag{17}$$
$$\neg S \tag{18}$$
$$\neg Q. \tag{19}$$

There are interesting similarities between $COMP(\Pi)$ and the translation we provide for the same logic program. If we take the program in the previous example and translate it using algorithm *translate-1*, we get that T_Π is the following theory (note that Π is acyclic according to our definitions):

$$I_V \tag{20}$$

$$I_Q \wedge \neg I_R \longrightarrow I_P \tag{21}$$
$$I_S \longrightarrow I_R \tag{22}$$
$$I_V \longrightarrow I_P \tag{23}$$
$$I_P \longrightarrow I_Q \wedge \neg I_R \vee I_V \tag{24}$$
$$I_R \longrightarrow I_S \tag{25}$$
$$\neg I_S, \neg I_Q, \{\neg I_{\neg L} | L \in \{P, Q, R, S, V\}\} \tag{26}$$
$$\{I_L \wedge I_{\neg L} \longrightarrow \textbf{false} | L \in \{P, Q, R, S, V\}\}. \tag{27}$$

Combining sentences (21), (23), and (24) and sentences (22) and (25) and replacing each symbol of the form I_L, where L is positive, with L, we get the following equivalent theory (compare to (15)-(19)):

$$\begin{aligned} P &\longleftrightarrow [Q \wedge \neg R] \vee V \\ R &\longleftrightarrow S \\ &V \\ &\neg S \\ &\neg Q \\ &\{\neg I_{\neg L} | L \in \{P, Q, R, S, V\}\} \\ &\{L \wedge I_{\neg L} \longrightarrow \textbf{false} | L \in \{P, Q, R, S, V\}\}. \end{aligned}$$

It is easy to see that each model for the above theory is a model of the completion of the program and that each model of the completion of the program can be extended to be a model for this theory. The above example can easily be generalized to a proof of the following theorem, which was also proved independently by Fages [Fag92]:

Theorem 4.19 *Let* Π *be a normal acyclic propositional logic program. Then* M *is a model for* $COMP(\Pi)$ *iff* $\{I_P | P \in M\}$ *is a model for* T_Π. □

Proof: (sketch) Let Π be an acyclic normal logic program, $\mathcal{L}$ the language of Π, and T'_Π the theory obtained from T_Π by replacing each occurrence of the atom I_P, where P is an atom in $\mathcal{L}$ with the symbol P. It is easy to see that the set of models of T'_Π projected on $\mathcal{L}$ is equivalent to the set of models of $COMP(\Pi)$. □

Corollary 4.20 *Let* Π *be an acyclic normal propositional logic program.* Π *has a stable model iff* $COMP(\Pi)$ *is consistent. Furthermore,* M *is a model for* $COMP(\Pi)$ *iff* M *is an answer set for* Π. $\Box$

Proof: Follows from the above theorem and Theorems 3.4 and 3.5. $\Box$

Corollary 4.21 *Let* Π *be an acyclic normal propositional logic program. An atom* P *is in the intersection of all the answer sets of* Π *(as defined by stable model semantics) iff* $COMP(\Pi) \models P$. $\Box$

Corollary 4.22 *Let* Π *be an acyclic normal propositional logic program. An atom* P *does not belong to any of the answer sets of* Π *(as defined by stable model semantics) iff* $COMP(\Pi) \models \neg P$. $\Box$

It is already known that each stable model for a normal logic program is a model of its completion [MS89] and that if an atom is implied by the completion of a locally stratified normal program, then it belongs to its (unique) answer set [ABW88, Prz89]. We believe that the above observations are new because they identify the class of acyclic normal propositional logic programs as a class for which stable model semantics (under "skeptical reasoning"[9]) is *equivalent* to Clark's predicate completion.

It is well known that Clark's predicate completion has problems handling positive recursion, while stable model semantics handles it in an intuitive manner. The translation that we provide explains this difference: stable model semantics distinguishes between cyclic and acyclic programs, while Clark's predicate completion is the same for both types.

5 Relation to Default Logic

While stable model semantics for nondisjunctive logic programs is very closely related to Reiter's default logic [Rei80], it does not agree with default logic when disjunctive logic programs are under consideration. In this section we will discuss the difference between the way default logic handles disjunctive information and the way stable model semantics handles disjunction in logic programs. We will also briefly review a generalization of default

[9] "Skeptical reasoning" means that a program entails an atom iff the atom belongs to all of the program's answer sets.

theories called *disjunctive default theories*, originally presented in Gelfond *et al.* [GPLT91], and show that the results developed in this paper apply to this class as well. We will consider only default theories that are defined over a propositional language, since this is sufficient for our case.

Let $\mathcal{L}$ be a propositional language. Reiter defines a *default theory* as a pair (D, W) , where D is a set of *defaults* and W is a set of well-formed formulas in $\mathcal{L}$. A *default* is a rule of the form

$$\frac{\alpha : \beta_1, ..., \beta_n}{\gamma},$$

where $\alpha, \beta_1, ..., \beta_n$, and γ are formulas in $\mathcal{L}$. The intuition behind a default can be "If α is believed and there is no reason to believe that one of the β_i is false, then γ can be believed".

The set of defaults, D, induces an *extension* on W. Intuitively, an extension is a maximal set of formulas that can be deduced from W using the defaults in D. An extension of a default theory corresponds to an answer set of a logic program. Let E^* denote the logical closure of E in $\mathcal{L}$. We use the following definition of an extension ([Rei80], Theorem 2.1):

Definition 5.1 *Let $E \subseteq \mathcal{L}$ be a set of formulas, and let (D, W) be a default theory. Define*

1. *$E_0 = W$ and,*
2. *for $i \geq 0$, $E_{i+1} = E_i{}^* \bigcup \{\gamma | \frac{\alpha:\beta_1,...,\beta_n}{\gamma} \in D$ where $\alpha \in E_i$ and $\neg\beta_1, ..., \neg\beta_n \notin E\}$.*

E is an extension for (D, W) iff $E = \bigcup_{i=0}^{\infty} E_i$ for some ordering (note the appearance of E in the formula for E_{i+1}). $\Box$

It turns out that Reiter's default logic has some difficulties when dealing with disjunctive information. This has motivated Gelfond *et al.* [GPLT91] to propose a generalization that improves the way default logic handles disjunctive information. They define a *disjunctive default theory* as a set of disjunctive defaults. A *disjunctive default* is an expression of the form

$$\frac{\alpha : \beta_1, ..., \beta_n}{\gamma_1 | ... | \gamma_m}, \tag{28}$$

where $\alpha, \beta_1, ..., \beta_n$, and $\gamma_1, ..., \gamma_m$ are formulas in $\mathcal{L}$. Gelfond *et al.* define an extension for a disjunctive default theory D to be one of the minimal deductively closed set of sentences E' satisfying the condition[10] that, for any default (28) from D, if $\alpha \in E'$ and $\neg\beta_1, ..., \neg\beta_n \notin E$, then for some $1 \leq i \leq m$, $\gamma_i \in E'$.

Let us now consider the subset of disjunctive default theories that we call *disjunctive default programs.* A *disjunctive default program* is a set of defaults of the form

$$\frac{L_{k+1} \wedge ... \wedge L_{k+m} : L_{k+m+1}, ..., L_{k+m+n}}{L_1 | ... | L_k}, \tag{29}$$

in which each L_i is a literal or the constant **true** and $n > 0$. Each such disjunctive default program D can be associated with a disjunctive logic program Π_D by replacing each default of the form (29) with the rule

$$L_1 | ... | L_k \longleftarrow L_{k+1}, ..., L_{k+m}, \mathit{not}\ \overline{L_{k+m+1}}, ..., \mathit{not}\ \overline{L_{k+m+n}},$$

where $\overline{L}$ is the literal complementary to L. Similarly, each disjunctive logic program Π can be associated with a disjunctive default program D_Π by replacing each rule of the form

$$L_1 | ... | L_k \longleftarrow L_{k+1}, ..., L_{k+m}, \mathit{not}\ L_{k+m+1}, ..., \mathit{not}\ L_{k+m+n} \tag{30}$$

with the default

$$\frac{L_{k+1} \wedge ... \wedge L_{k+m} : \overline{L_{k+m+1}}, ..., \overline{L_{k+m+n}}}{L_1 | ... | L_k}.$$

According to Gelfond *et al.* , we have the following theorem:

Theorem 5.2 *[GPLT91] Let D be a disjunctive default program. E^* is an extension of D iff E is an answer set of* Π_D*.*

Let Π *be a disjunctive logic program. E is an answer set of* Π *iff E^* is an extension of* D_Π*.*

This theorem implies that all the techniques and complexity results established in this paper with respect to disjunctive logic programs also apply to disjunctive default programs.

[10]Note the appearance of E in the condition.

6 Related Work

Elkan has shown [Elk90] that stable models of *normal* logic programs can be represented as models of propositional logic. Our results extend his work to a more expressive class of logic programs and provide an explicit propositional theory characterizing those models. We have also shown how the propositional characterization clarifies the semantical difference between various formalisms and how it can be used to transfer computational techniques from propositional satisfiability to logic programs.

An approach similar to ours was developed independently by Marek and Truszczynski [MT91b] in the context of autoepistemic logic. They have shown how questions of membership in expansions of an autoepistemic theory can be reduced to propositional provability. In this paper we provide an explicit algorithms for processing queries on stable models of disjunctive logic programs, together with complexity guarantees. We believe that similar results could be obtained for autoepistemic logic using Marek and Truszczyński's analysis.

Computationally, the most relevant work is that of Bell *et al.* and Subrahmanian *et al.* [BNNS91, SNV92], which implements linear and integer programming techniques in order to compute stable models (among other nonmonotonic logics). However, because their mapping of logic programs into linear integer programs is not "perfect" — not every solution of the set of constraints corresponds to a stable model[11]— it is difficult to assess the merit of their approach relative to ours in terms of both complexity and semantics. We do note that in order to find whether an atom belongs to all answer sets of a given logic program, they have to generate all the answer sets, while because we solve this problem using classical deduction, we may not need to generate all answer sets.

7 Conclusion

This paper provides several characterizations of answer sets for head-cycle-free extended disjunctive logic programs (HEDLPs) according to stable model semantics. It shows that any grounded HEDLP can be mapped in polynomial time into a propositional theory such that models of the latter and answer

[11]A similar observation and the term "perfect" was given to us by Mirek Truszczyński.

sets of the former coincide. This allows techniques developed for solving satisfiability problems to be applied to logic programming problems. It also enables an evaluation of the expressive power of EDLPs, identification of their tractable subsets, and discovery of useful similarities between stable model semantics and Clark's predicate completion.

We have also shown the relevance of our work to the work done on disjunctive default theories [GPLT91]. One of the possible drawbacks of stable model semantics is that it entails multiple answer sets. The approach proposed in this paper suggests that in order to compute whether a literal belongs to one or all answer sets we do not need to compute or count those sets. Thus, multiplicity of answer sets may not in itself be a severe computational obstacle to the practicality of disjunctive logic programming.

We are currently investigating extensions of our work to a class of ungrounded logic programs.

Acknowledgments

This work was supported in part by grants NSF IRI-88-21444 and AFOSR 90 0136 and by an IBM Graduate Fellowship to the first author and grant NSF IRI-91573636 to the second author. The authors thank Michael Gelfond, Vladimir Lifschitz, Jack Minker, Stott Parker, Teodor Przymusinski, Mirek Truszczyński, and Carlo Zaniolo for helpful discussions on the relations between logic programming and default reasoning. Michael Gelfond and Vladimir Lifschitz pointed out to us the difference between the use of disjunction in default logic and in logic programs and drew our attention to the close connection between our translation and Clark's predicate completion. We also thank Judea Pearl and Halina Przymusinska for valuable comments on earlier drafts of this paper.

References

[ABW88] K.R. Apt, H.A. Blair, and A. Walker. Towards a theory of declarative knowledge. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programs*, pages 89–148. Morgan Kaufmann, 1988.

[ACP87] S. Arnborg, D.G. Corneil, and A. Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM J. Algebraic Discrete Method*, 8(2):277–284, 1987.

[BED91] Rachel Ben-Eliyahu and Rina Dechter. Default logic, propositional logic and constraints. In *AAAI-91: Proceedings of the 9th national conference on artificial intelligence*, pages 379–385, Anaheim, CA, USA, July 1991.

[BED92] Rachel Ben-Eliyahu and Rina Dechter. Inference in inheritance networks using propositional logic and constraints networks techniques. In *AI-92: Proceedings of the 9th Canadian conference on AI*, pages 183–189, Vancouver, British Columbia, Canada, May 1992.

[BF87] N. Bidoit and C. Froidevaux. Minimalism subsumes default logic and circumscription in stratified logic programming. In *LICS-87: Proceedings of the IEEE symposium on logic in computer science*, pages 89–97, Ithaca, NY, USA, June 1987.

[BK] Gerhard Brewka and Kurt Konolige. Personal communication, May 1992.

[BNNS91] C. Bell, A. Nerode, R.T. Ng, and V.S. Subrahmanian. Computation and implementation of non-monotonic deductive databases. Technical Report CS-TR-2801, University of Maryland, 1991.

[Cla78] Keith L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, New York, 1978.

[Dec90] Rina Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41:273–312, 1990.

[Dec92] Rina Dechter. Constraint networks. In Stuart C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, pages 276–285. John Wiley, 2nd edition, 1992.

[DP88] Rina Dechter and Judea Pearl. Network-based heuristics for constraint satisfaction problems. *Artificial Intelligence*, 34:1–38, 1988.

[DP89] Rina Dechter and Judea Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, 38:353–366, 1989.

[EG92] Thomas Eiter and Georg Gottlob. Complexity results for disjunctive logic programming and application to nonmonotonic logics. Technical Report CD-TR-92/41, Inst. fuer Informations systeme, TU-Wien, A-1040 Wien, Austria, 1992.

[Elk90] Charles Elkan. A rational reconstruction of nonmonotonic truth maintenance systems. *Artificial Intelligence*, 43:219–234, 1990.

[Eve79] Shimon Even. *Graph Algorithms*. Computer Science Press, 1979.

[Fag92] Francois Fages. Consistency of clark's completion and existence of stable models. *Methods of Logic in Computer Science*, 2, April 1992.

[Fin89] Kit Fine. The justification of negation as failure. *Logic, Methodology and Philosophy of Science*, 8:263–301, 1989.

[Fre85] E.C. Freuder. A sufficient condition for backtrack-bounded search. *Journal of the ACM*, 32(4):755–761, 1985.

[GL91] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.

[GPLT91] Michael Gelfond, Halina Przymusinska, Vladimir Lifschitz, and Miroslaw Truszczyński. Disjunctive defaults. In *KR-91: Proceedings of the 2nd international conference on principles of knowledge representation and reasoning*, pages 230–237, Cambridge, MA, USA, 1991.

[KS91] Henry A. Kautz and Bart Selman. Hard problems for simple default logics. *Artificial Intelligence*, 49:243–279, 1991.

[MR90] Jack Minker and Arcot Rajasekar. A fixpoint semantics for disjunctive logic programs. *Journal of Logic Programming*, 9:45–74, 1990.

[MS89] Wiktor Marek and V.S. Subrahmanian. The relationship between logic program semantics and non-monotonic reasoning. In *Logic Programming: Proceedings of the 6th international conference*, pages 600–617, Lisbon, Portugal, June 1989. MIT Press.

[MT91a] Wiktor Marek and Miroslaw Truszczyński. Autoepistemic logic. *Journal of the ACM*, 38:588–619, 1991.

[MT91b] Wiktor Marek and Miroslaw Truszczyński. Computing intersection of autoepistemic expansions. In *Logic programming and non-monotonic reasoning: Proceedings of the 1st international workshop*, pages 37–50, Washington, DC, USA, July 1991.

[Prz89] Teodor Przymusinski. On the declarative and procedural semantics of logic programs. *Journal of Automated Reasoning*, 5:167–205, 1989.

[Rei80] Raymond Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.

[SNV92] V.S. Subrahmanian, Dana Nau, and Carlo Vago. WFS + branch and bound = stable models. Technical Report CS-TR-2935, University of Maryland, July 1992.

[Sti90] Jonathan Stillman. It's not my default: The complexity of membership problems in restricted propositional default logics. In *AAAI-90: Proceedings of the 8th national conference on artificial intelligence*, pages 571–578, Boston, MA, USA, 1990.

[Tar72] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1:146–160, 1972.

Appendix

A Expressing Indexes in Propositional Logic

Suppose we are given a set of symbols L to each of which we want to assign an index variable within the range $1 - m$.

We define a new set of symbols, $L' = \{P, P = 1, P = 2, ..., P = m | P \in L\}$, where $P = i$ for $i = 1, ..., m$ denotes a propositional letter that intuitively means "P will get index i". For each P in $\mathcal{L}_\Pi$, let N_P be the following set of sentences:

$$\begin{aligned} P = 1 \vee P = 2 \vee ... \vee P = m \\ P = 1 \longrightarrow [\neg(P = 2) \wedge \neg(P = 3) \wedge ... \wedge \neg(P = m)] \\ P = 2 \longrightarrow [\neg(P = 3) \wedge \neg(P = 4) \wedge ... \wedge \neg(P = m)] \\ . \\ . \\ . \\ P = m - 1 \longrightarrow \neg(P = m). \end{aligned}$$

The set N_P simply states that p must be assigned one and only one number. We will sometimes denote the theory N_P by $\#P$.

For each P and Q in $\mathcal{L}_\Pi$, let $[\#P < \#Q]$, which intuitively means "The number of P is less than the number of Q", denote the *disjunction* of the following set of sentences:

$$\begin{aligned} P = 1 \wedge Q = 2, P = 1 \wedge Q = 3, ..., P = 1 \wedge Q = m \\ P = 2 \wedge Q = 3, ..., P = 2 \wedge Q = m \\ . \\ . \\ . \\ P = m - 1 \wedge Q = m. \end{aligned}$$

Thus, for each symbol P to which we want to assign an index, we add N_P to the theory, and then we can use the notation $[\#P < \#Q]$ to express the order between indexes.

B Proofs

Some useful definitions and lemmas

Lemma B.1 *Let Π be an EDLP. If S is an answer set of Π, then there is no proper subset of S that is also an answer set of Π.*

Proof: Suppose conversely that $S' \subset S$ and that both S and S' are answer sets of Π. We will show that S' satisfies each rule in Π^S, which is a contradiction to the minimality of S. This is done by the observation that since $S' \subset S$, $\Pi^S \subseteq \Pi^{S'}$. $\Box$

Definition B.2 (superstructure) *[Eve79] Let $C_1, ..., C_m$ be the strongly connected components of a directed graph G. The* superstructure *of G, G_S, is defined as follows:*

1. *The set of all vertexes of G_S is the set $C_1, ..., C_m$.*
2. *There is an edge from C_i to C_j iff x and y are vertexes in G, x belongs to C_i, y belongs to C_j, and there is an edge from x to y in G.*

Lemma B.3 *Let Π be a $\underline{positive}$[12] HEDLP and let S be a context satisfying all the rules in Π. Suppose that there is a context $E \subset S$ and a set of rules Δ such that the following conditions hold:*

1. *Δ is the set of all rules from Π not satisfied by E.*
2. *For each rule in Δ there are at least two different literals in its head that belong to S.*

Then S is not minimal, i.e., there is a proper subset of S which satisfies all the rules in Π.

[12]A program is called *positive* if the *not* operator is not used in the program.

Proof: Let Π be a positive HEDLP and let S be a context satisfying all the rules in Π. Assume that there is a context $E \subset S$ that satisfies conditions 1-2 of the lemma. We will show that there is a proper subset of S which satisfies all the rules in Π.

Let $\Delta \subseteq \Pi$ be the set of all rules not satisfied by E and suppose that for each rule in Δ there are at least two different literals in its head that belong to S. Let H be the set of all literals in S that appear in the head of some rule from Δ. Let G be the directed graph defined as follows:

1. There is a node in the graph for each literal in H and there are no other nodes.
2. There is an edge from L_1 to L_2 iff there is a directed path from L_1 to L_2 in G_Π (the dependency graph of Π).

Let G_S be the superstructure of G. If G_S is empty then H is empty and so Δ must be empty too, which means that E satisfies all the rules in Π. Otherwise, G_S must have a source (note that we assume that the dependency graph has no infinitely decreasing chains and hence, even though G_S might be infinite, it must have a source). Let F be the set of all literals that appear in a source of G_S. Note that for every $p, q \in F$ there is a positive cycle between p and q in the dependency graph of Π. Consider the context S', defined as follows:

$$\begin{aligned}
S_0 = \quad & E, \\
S_{i+1} = \quad & S_i \bigcup \{p | p \in S - F, \text{there is a rule } \delta \text{ in } \Pi \text{ such that the body of } \delta \\
& \text{is satisfied by } S_i \text{ and } p \text{ is in the head of } \delta\}, \\
S' = \quad & \textstyle\bigcup_{i=1}^{\infty} S_i.
\end{aligned}$$

Clearly, S' is a proper subset of S. It is also easy to see that for every literal p in $S' - E$, there must be a path in the dependency graph of Π from some literal in $H - F$ to p. Let δ be an arbitrary rule in Π. We will show that S' satisfies δ. The proof will go by induction on the minimum i such that the body of δ is satisfied by S_i.

case $i = 0$ In this case the body of δ is satisfied by E. If E satisfies δ, then clearly S' satisfies δ as well. If E does not satisfy δ, then

$\delta \in \Delta$, which means that there are at least two different literals, say p and q, in the head of δ which belong to S. It can't be that both p and q belong to F because Π is head-cycle-free. So assume WLG that $p \in S - F$. So p must be in S_1, which means that S' satisfies δ.

case $i > 0$ Suppose that the body of some rule δ is satisfied by S_i. Since S satisfies δ, there must be some literal q in the head of δ which belongs to S. We will show that q belongs to $S - F$. Since i is a minimum, there must be a literal p in the body of δ such that $p \notin S_{i-1}$. So p must be in $S' - E$ and therefore, as we have observed above, there must be a path in the dependency graph from some literal in $H - F$ to p, and so there is a path in the dependency graph from some literal in $H - F$ to q. So it can't be that q is in F, because F is a source in G_S.

□

Lemma B.4 *Let* Π *be a* *positive* *HEDLP and let* S *be an answer set of* Π*, then each* $L \in S$ *has a proof w.r.t.* S *and* Π*.*

Proof: Consider the context E defined as follows:

$E_0 = \ \{L |\ L \in S,\ \delta = L_1|...|L_k \longleftarrow \ \in \Pi,\ L = h_S(\delta)\}$,

$E_{i+1} = E_i \bigcup \{L |\ L \notin E_i,\ L = h_S(\delta),\ \delta = L_1|...|L_k \longleftarrow L_{k+1}, ..., L_{k+m}$, for all $k < i \leq k + m\ L_i \in E_i\}$,

$E = \bigcup_{1 \leq \infty} E_i$.

Clearly, each literal in E has a proof w.r.t. E and Π. We will prove that $S = E$.

Clearly, $E \subseteq S$. Assume conversely that $E \subset S$. Consider the set Δ of all rules from Π that are not satisfied by E. By the way it was constructed, E satisfies all the rules such that there is only one literal in the rule's head that belongs to S. So each rule in Δ must have at least two literals in its head that belong to S. By Lemma B.3, S is not minimal, which is a contradiction to S being an answer set of Π. □

Lemma B.5 *Let* Π *be an HEDLP and let* S *be an answer set of* Π*, then each* $L \in S$ *has a proof w.r.t.* S *and* Π*.*

Let Π be an HEDLP and let S be an answer set of Π. Since S is an answer set of Π^S and Π^S does not contain *not* , we know by Lemma B.4 that, given $L \in S$, L has a proof $\delta_1, ..., \delta_n$ w.r.t. S and Π^S. For each $\delta_i = L_1|...|L_k \longleftarrow L_{k+1}, ..., L_{k+m}$ in the proof, there is a rule δ_i' in Π such that $\delta_i' = L_1|...|L_k \longleftarrow L_{k+1}, ..., L_{k+m}, not\ L_{k+m+1}, ..., not\ L_{k+m+n}$, and, for each $k+m+1 \leq j \leq k+m+n$, L_j is not in S. So $\delta_1', ..., \delta_n'$ is a proof of L w.r.t. Π and S. $\Box$

Proof of Theorem 2.3 Let Π be an HEDLP and let S be one of its answer sets. Clearly S satisfies conditions 1 and 3. By Lemma B.5, S satisfies condition 2 as well.

Now, suppose S satisfies conditions 1-3 of the theorem. We will show that it is an answer set of Π. It is enough to show that S is an answer set of Π^S. S satisfies all the rules from Π^S since it satisfies all the rules from Π. To show that S is minimal, it is enough to show that for each $A \subseteq S$, there is a rule in Π^S that will not be satisfied by $S - A$. Let A be an arbitrary nonempty subset of S, and let $L \in A$ be such that L's proof length is minimal w.r.t. the proofs of all the literals in A. Let δ be the last rule in the proof of L. By minimality of L's proof, all the literals in the body of δ must belong to $S - A$, and so δ is not satisfied by $S - A$. $\Box$

Proof of Theorem 2.7 Suppose Π is acyclic and W is a context satisfying conditions 1-3 of the theorem. We will show that W is an answer set. By Theorem 2.3, it is enough to show that each $L \in S$ has a proof. We will build the "proof graph" G as follows: There is a node in G for each literal in S. For each $L \in S$, let δ be a rule from Π that satisfies condition 2 for L. We draw an edge to L from each literal that appears in the body of δ without the *not* operator. Since Π is acyclic, G is acyclic, so it induces a partial order on its nodes. It is easy to see that we can prove that each L has a proof by induction on that partial order. $\Box$

Proof of Theorem 2.8 Suppose S is a context satisfying the conditions of the theorem with respect to a given HEDLP Π. We will show that

S is an answer set of Π. By Theorem 2.3, it is enough to show that each $L \in S$ has a proof w.r.t S and Π. If S is empty, the claim clearly holds. Otherwise we can prove the claim by induction of the index of L ($f(L)$).

Suppose S is an answer set of an HEDLP Π. By Theorem 2.3, it must satisfy conditions 1 and 3. It is left to show that it satisfies condition 2. Let f be a function that assigns to each literal L the minimal number of rules used in any proof of L w.r.t. Π and S. It is easy to see that f and the last rule in any minimal proof of L satisfy subconditions (a)-(d) in condition 2 w.r.t. L. $\Box$

Proof of Theorem 3.2 Suppose S is a context satisfying the conditions of the theorem with respect to a given HEDLP Π. We will show that S is an answer set of Π. By Theorem 2.3, it is enough to show that each $L \in S$ has a proof w.r.t S and Π. We will define a total ordering on the literals in $\mathcal{L}$ and prove it by induction on that ordering.

Let $C_1, ..., C_m$ be the strongly connected components of G_Π. Following [Eve79], we define G^*, the *superstructure* of G_Π, as follows:

$$\begin{aligned} V^* &= \{C_1, ..., C_m\}, \\ E^* &= \{(C_i, C_j) \mid i \neq j, \text{ there is an edge from } L \text{ to } L' \text{ in } G_\Pi, \\ &\qquad\qquad\qquad L \in C_i, \text{ and } L' \in C_j\}. \end{aligned}$$

Clearly, G^* is acyclic.

Let $C_1, ..., C_m$ be the topological ordering of the components of G_Π induced by G^*. Let $\prec$ be any total ordering on the literals of $\mathcal{L}$ satisfying the following:

- If $L \in C_i$, $L' \in C_j$, and $i < j$, then $L \prec L'$.
- If L and L' are in the same component and $f(L) < f(L')$, then $L \prec L'$.

We will proceed by showing that for each L if $L \in S$ then there is a proof $\delta_1, ..., \delta_j$ of L in S such that for all $i \in \{1, ..., j\}$ if L' appears positive in the body of δ_i then $L' \prec h_S(\delta_i)$.

We prove this by induction on $@L$, the place of L in the ordering.

Case $@L = 1$: Since $L \in S$, by condition 2 there must be a rule δ in Π such that

1. the body of δ is satisfied by S,
2. L appears in the head of δ,
3. all literals in the head of δ other than L are not in S, and
4. for each literal L' that appears positive in the body of δ and shares a cycle with L in the dependency graph of Π, $f(L') < f(L)$.

Since $@L = 1$, there must be no positive literal in the body of δ. So it is easy to see that δ is a proof of L w.r.t S and Π.

Case $@L > 1$ is done in a similar way, since we realize that all the literals that appear positive in the rule δ that satisfies the above conditions for L must have a lower place in the ordering and therefore have a proof by the induction hypothesis.

To prove the other direction, suppose S is an answer set of an HEDLP Π. By Theorem 2.3, it must satisfy conditions 1 and 3. It is left to show that it satisfies condition 2.

First we show how we assign a number to each literal:

- By Theorem 2.3, each L in S has a proof w.r.t. S and Π. We first build G, the "proof graph" of S, as follows:
  1. If all literals that are true in S are in G, then exit.
  2. Pick $L \in S$ such that $L \notin G$.
  3. Let $\delta_1, ..., \delta_n$ be a minimal proof of L in S.
  4. For each δ_i do the following in the order $\delta_1, ..., \delta_n$:
     If $h_S(\delta_i)$ is not already a node in G, add it as a node and add an edge to $h_S(\delta_i)$ from each literal that appears positive in the body of δ_i.
  5. Goto step 1.
- We claim that G is acyclic because each proof is acyclic and because there are no arrows from literals introduced in one proof to literals introduced in a previous proof.

Let $L_1, ..., L_n$ be the topological ordering induced on the literals in S by G.

Based on this ordering, we will give each L_i a number ($\#L_i$) as follows:

If there is a $j < i$ such that L_j and L_i are in the same component in G_Π (the dependency graph of Π) and there is a path from L_j to L_i in the proof graph, then let r be a maximal such j. Assign $\#L_i = \#L_r + 1$.

If there is no such j, then $\#L_i = 0$.

The following proposition indicates that for each literal L we will get that $\#L \leq r$, where r is the length of the longest acyclic path in any component of G_Π.

Proposition B.6 *If in the proof graph there is a directed path with k nodes from some component C in* $G_\Pi (k > 0)$*, then there is a directed acyclic path of length* $\geq k - 1$ *in that component with those k nodes along that path.*

Proof: Given an arbitrary component C in G_Π and a path in the proof graph, the proof goes by induction on the number of nodes from C along the path. □

Now we want to show that S satisfies condition 2 of the theorem. Let $L \in S$, and let δ be the last rule in the proof of L that was used when the proof graph was constructed. If there is a L' that appears positive in the body of δ that is in a same component as L, it is assigned a lower number by our numbering method. So condition 2 is satisfied. □