Solving Epistemic Logic Programs Using Generate-and-Test with Propagation

Jorge Fandinno[1], **Lute Lillo**[1,2]
[1]University of Nebraska Omaha, Omaha, NE, USA
[2]University of Vermont, Burlington, VT, USA
jfandinno@unomaha.edu
elillopo@uvm.edu

Abstract

This paper introduces a general framework for generate-and-test-based solvers for epistemic logic programs that can be instantiated with different generator and tester programs. It proves sufficient conditions on those programs for the correctness of the solvers built using this framework. It also introduces a new generator program that incorporates the *propagation of epistemic consequences* and shows that this can exponentially reduce the number of candidates that need to be tested while only incurring a linear overhead. We implement a new solver based on these theoretical findings and experimentally show that it outperforms existing solvers by achieving a $\sim$3.3x speed-up and solving 91% more instances on well-known benchmarks.

Introduction

Answer Set Programming (ASP) is a declarative programming language well-suited for solving knowledge-intensive search problems, which allows users to encode problems such that the resulting output of the program (called stable models or answer sets) directly corresponds to solutions of the original problem (Gelfond and Lifschitz 1988, 1991; Lifschitz 2008; Brewka, Eiter, and Truszczyński 2011; Schaub and Woltran 2018; Lifschitz 2019). Epistemic logic programs (ELPs) extend the ASP language by allowing the use of *subjective literals* in the body of rules (Gelfond 1991; Gelfond and Przymusinska 1993; Gelfond 1994; Fandinno, Faber, and Gelfond 2022). As an example, rules

$$felon \leftarrow \mathbf{K}\, break\_rule$$
$$suspect \leftarrow \neg\mathbf{K}\, break\_rule, \neg\mathbf{K}\neg break\_rule$$

encode that a person is a $felon$ if we can determine that she broke a rule, and she is a $suspect$ if it cannot be determined that she has broken it nor that she has not done so (Son et al. 2017). ELPs allow us to naturally represent problems that involve reasoning about the lack of knowledge of agents—as illustrated by the above example—as well as problems laying on the second level of the polynomial hierarchy such as conformant planning (Son et al. 2017; Kahl et al. 2020; Cabalar, Fandinno, and Fariñas del Cerro 2019b, 2021), action reversibility (Faber, Morak, and Chrpa 2021) or reasoning about attack trees and graphs (Fandinno, Faber, and Gelfond 2022). It is worth noting that the computational complexity of ELPs is higher than that of ASP: the problem of determining whether an ELP has a solution (called *world-view*) is Σ_3^P-complete (Truszczynski 2011), one level higher in the polynomial hierarchy than the complexity deciding whether an objective program has a stable model (Dantsin et al. 2001). Hence, the development of efficient tools to solve ELPs, called ELP solvers, is crucial for exploiting the high expressivity capabilities of ELPs in practice. Existing ELP solvers can be classified in three categories: *post-processing*-based, *generate-and-test*-based, and *translational*-based. Early solvers were post-processing-based as they use an ASP solver to compute all stable models of some objective (or non-epistemic) program that are post-processed to obtain its worldviews (see Leclerc and Kahl 2018 for a survey). Generate-and-test-based solvers on the other hand use two instances of an ASP solver—one to generate candidate worldviews and another to check whether they are indeed worldviews—avoiding the generation of all stable models in most practical cases (Son et al. 2017; Cabalar et al. 2020). Translational-based solvers translate ELPs into an alternative language that matches the computational complexity of ELPS such as non-ground objective programs with large rules (Bichler, Morak, and Woltran 2020) or ASP with Quantifiers (Faber and Morak 2023). Existing generate-and-test and translational based solvers outperform post-processing based solvers in existing benchmarks (Leclerc and Kahl 2018; Cabalar et al. 2020), while the literature lacks clear evidence when comparing generate-and-test and translational based solvers. Our own experiments show that existing generate-and-test-based solvers outperform existing translational-based solvers (See the Experimental Evaluation Section below).

In this paper, (i) we introduce a general framework for generate-and-test-based solvers that can be instantiated with different generator and tester programs, and we prove sufficient conditions on those programs for the correctness of the solvers built using this framework; (ii) we instantiate this framework with a new generator program that incorporates the *propagation of epistemic consequences* and prove that this can exponentially reduce the number of candidates that need to be tested while only incurring in a linear overhead; and (iii) we implement a new solver based on the two

previous points and experimentally demonstrate that it can solve 91% more instances than the best performing available solver with a ~3.3x speed-up.

Background

We assume some familiarity with the answer set semantics for disjunctive logic programs (Gelfond and Lifschitz 1991). Given a set of atoms At, an *objective literal* is either an atom or an atom preceded by one or two occurrences of the negation symbol "$\neg$." An *extended objective literal* is either an objective literal or a truth constant[1]. An expression of the form $\mathbf{K}l$ with l being an extended objective literal is called *subjective atom*. A *subjective literal* L is a subjective atom possibly preceded by one or two occurrences of the negation symbol. A *literal* is either an extended objective literal or a subjective literal. A *rule* is an expression of the form:

$$a_1 \vee \cdots \vee a_n \leftarrow L_1, \ldots, L_m \tag{1}$$

with $n \geq 0$ and $m \geq 0$, where each $a_i \in At$ is an atom and each L_j a literal. The left- and right-hand sides of (1) are respectively called the *head* and *body* of the rule. When $n = 0$, the rule is called a *constraint* and we usually write $\bot$ as its head. A *choice rule* is an expression of the form:

$$\{a\} \leftarrow L_1, \ldots, L_m \tag{2}$$

and it is understood as a shorthand for the rule:

$$a \leftarrow L_1, \ldots, L_m, \neg\neg a \tag{3}$$

An *epistemic program* (or epistemic specification) is a finite set of rules. Given an epistemic program Π, we define $At(\Pi)$ as the set of all atoms that occur in program Π. An epistemic program without subjective atoms is called an *objective program*. Similarly, a rule without subjective atoms is called *objective rule*. In many descriptions of epistemic logic programs, a second epistemic operator $\mathbf{M}$ is also introduced. Though interesting from a knowledge representation perspective, this operator is not necessary from the solving perspective because it can be replaced by $\neg\mathbf{K}\neg$ (Fandinno, Faber, and Gelfond 2022). Therefore, we focus on programs with the operator $\mathbf{K}$ only.

An interpretation $I \subseteq At$ is a set of atoms. We say that an interpretation I satisfies an extended objective literal L if L is $\top$, or L is an atom and $L \in I$, or L is of the form $\neg a$ and $a \notin I$, or L is of the form $\neg\neg a$ and $a \in I$. An interpretation I *satisfies* an objective rule if it satisfies some atom in the head of the rule whenever it satisfies all the literals in its body. An interpretation I is a *model* of an objective program if it satisfies all rules of the program. We say that a literal is *negated* if it is of the form $\neg A$ or $\neg\neg A$ for some (subjective) atom A. The reduct of an objective program Π with respect to an interpretation I, in symbols Π^I, is obtained by removing all rules with a body that contains a negated literal L that is not satisfied by I, and by removing all negated literals from the remaining rules. An interpretation I is a *stable model* of an objective program Π if I is a $\subseteq$-minimal model of the reduct Π^I. SM$[\Pi]$ denotes the stable models of Π.

An *assumption* is a set of objective literals A of the form a or $\neg a$ such that it does not contain both a and $\neg a$ for any atom a. A set of atoms M is a *stable model* of an objective program Π *under assumption* A if M is a stable model of the program

$$\Pi \cup \{\bot \leftarrow \neg a \mid a \in A\} \cup \{\bot \leftarrow a \mid \neg a \in A\}.$$

By SM$[\Pi; A]$ we denote the stable models of Π under the assumption A. By CC$[\Pi; A]$ we denote the set of of *cautions consequences* of Π *under the assumption* A, that is,

$$\text{CC}[\Pi; A] \quad = \quad \bigcap \text{SM}[\Pi; A]$$

Let $\mathbb{W}$ be a set of interpretations. We write $\mathbb{W} \models \mathbf{K}l$ if objective literal l is satisfied by every interpretation I of $\mathbb{W}$, and $\mathbb{W} \models \neg\mathbf{K}l$ otherwise.

Definition 1 (Subjective reduct; Fandinno, Faber, and Gelfond 2022)**.** *The* subjective reduct *of an epistemic program* Π *with respect to a set of interpretations* $\mathbb{W}$*, written* $\Pi^{\mathbb{W}}$*, is obtained by replacing each subjective literal* L *by* $\top$ *if* $\mathbb{W} \models L$ *and by* $\bot$ *otherwise.* □

Note that the subjective reduct of an epistemic program does not contain subjective literals and, thus, it is an objective program. Therefore, we can collect its stable models.

Definition 2. *A* belief interpretation $\mathbb{W}$ *is a non-empty set of interpretations. A* worldview *of an epistemic program* Π *is a belief interpretation* $\mathbb{W}$ *such that* $\mathbb{W} = \text{SM}[\Pi^{\mathbb{W}}]$.

This definition of worldview is a rephrasing of the original definition by Gelfond (1994), usually denoted G94. In this work, we focus on obtaining worldviews of epistemic programs under this semantics. From a solving viewpoint, this is not a limitation because other semantics can use G94 as their basis. There are simple linear-time reductions from the G11 (Gelfond 2011) and the K15 (Kahl et al. 2015) semantics to G94 (Fandinno, Faber, and Gelfond 2022), while the worldviews according to the S16 (Shen and Eiter 2016, 2017) and the C19 (Cabalar, Fandinno, and Fariñas del Cerro 2019a, 2020) semantics are a selection of the K15 (Kahl, Leclerc, and Son 2016; Son et al. 2017) and the G94 worldviews, respectively.

Normal Form

To develop algorithms for solving epistemic logic programs, it is interesting to have a normal form that simplifies the number of cases the algorithm must consider. This also significantly simplifies the presentation of the formal results. We say that an epistemic program is in *normal form* if negation does not occur in the scope of the $\mathbf{K}$ operator, that is, every subjective literal is of the forms $\mathbf{K}a$, $\neg\mathbf{K}a$ or $\neg\neg\mathbf{K}a$ with a an atom. We can transform any epistemic program Π into a corresponding program in normal form by

- replacing every subjective literal $\mathbf{K}\neg a$ by $\mathbf{K}na$ and adding rule $na \leftarrow \neg a$ to Π;
- replacing every subjective literal $\mathbf{K}\neg\neg a$ by $\mathbf{K}nna$ and adding rule $nna \leftarrow \neg\neg a$ to Π;

[1]For a simpler description of program transformations, we allow truth constants in rules, where $\top$ denotes true and $\bot$ denotes false. These constants can be easily removed.

Algorithm 1: Generate-and-test computation of n worldviews of a program Π in normal form.

Input Generate program $G(\Pi)$
Input Test program $T(\Pi)$
Input Number of requested worldviews n
Output Set Ω containing at most n worldviews of Π

```
Let Ω = ∅.
for M in SM[G(Π)] do
    if Test(T(Π), M) then
        𝕎 = BuildWorldView(M)
        Ω = Ω ∪ {𝕎}
        if |Ω| ≥ n then
            return Ω
        end if
    end if
end for
return Ω
```

We assume that for every atom a in Π, atoms na and nna do not occur in Π. By $NF(\Pi)$ we denote the program obtained from Π by applying these two steps. In the following result $\mathbb{W}_{|At(\Pi)}$ stands for the set of interpretations obtained from $\mathbb{W}$ by removing all atoms that do not occur in Π from all the interpretations in $\mathbb{W}$.[2]

Theorem 1. *There is a one-to-one correspondence between the worldviews of* Π *and* $NF(\Pi)$ *s.t.* $\mathbb{W}$ *is a worldview of* $NF(\Pi)$ *if and only if* $\mathbb{W}_{|At(\Pi)}$ *is a worldview of* Π.

Worldviews by Generate-and-Testing

As mentioned in the introduction, generate-and-test-based solvers rely on using two objective programs: a generator program that provides *candidates* and a tester program that checks whether a candidate is a worldview. Algorithm 1 summarizes how to compute n worldviews using this approach, where $G(\Pi)$ and $T(\Pi)$ respectively are a *generator* and a *tester program* for Π. Different generator and tester programs can be used to compute the worldviews of a program. Here we provide general definitions for these programs with sufficient conditions to ensure that Algorithm 1 correctly computes the worldviews of a program.

We assume that, for each subjective literal $\mathbf{K}a$ in Π, there is a new fresh objective atom ka that does not occur anywhere else in the program. For simplicity, we assume that no atom occurring in the program starts with the letter k and, thus, all atoms that start with this letter are newly introduced. We denote by $K(\Pi)$ the set of these newly introduced atoms. Given an interpretation $M \subseteq At(\Pi) \cup K(\Pi)$, we define $k(M) = M \cap K(\Pi)$ as the set of atoms of the form $ka \in K(\Pi)$ that belong to M with $a \in At(\Pi)$. For a worldview $\mathbb{W}$, by $k(\mathbb{W})$ we denote the set of atoms of $ka \in K(\Pi)$ such that $\mathbb{W} \models \mathbf{K}a$. In our characterization, each worldview $\mathbb{W}$ of a program Π is associated with a stable model M of an objective program such that $k(\mathbb{W}) = k(M)$ satisfying some extra conditions that we introduce below.

[2]The proof can be found in arXiv (Fandinno and Lillo 2024).

Using this terminology, we can define our general notions of generator and tester program as follows.

Definition 3 (Generator Program)**.** *We say that an objective program* $G(\Pi)$ *is a* generator program *for a program* Π *if it satisfies the following conditions:*

- *for every worldview* $\mathbb{W}$ *of* Π*, there is* $M \in \text{SM}[G(\Pi)]$ *satisfying* $k(\mathbb{W}) = k(M)$ *and* $(M \cap At(\Pi)) \in \mathbb{W}$.
- *every stable model* $M \in \text{SM}[G(\Pi)]$ *and every set of interpretations of* $\mathbb{W}$ *satisfying* $k(\mathbb{W}) = k(M)$ *also satisfy* $(M \cap At(\Pi)) \in \text{SM}[\Pi^{\mathbb{W}}]$.

The first condition in Definition 3 ensures that, for every worldview of the original program, the generator program has a stable model that represents that worldview. The second condition ensures that such a stable model corresponds to a non-empty set, that is, a belief interpretation. Recall, that empty sets of interpretations cannot be worldviews.

Before providing our general notion of a tester program, let us introduce an example of a tester program. The intuitive idea of a tester program is that it behaves as the subjective reduct of the original program modulo the auxiliary atoms when used in combination with the proper assumptions. For any rule r of the form of (1), by $k(r)$ we denote the result of replacing each subjective literal $\mathbf{K}a$ by ka. For any program Π, by $k(\Pi)$ we denote the program obtained from Π by replacing each rule r of Π by $k(r)$. Clearly, $k(\Pi)$ is an objective program over $At(\Pi) \cup K(\Pi)$. By $T_0(\Pi)$ we denote the program obtained from $k(\Pi)$ by adding a choice rule $\{ka\}$ for each atom $ka \in K(\Pi)$. The following result shows that $T_0(\Pi)$ behaves as the subjective reduct of Π modulo the auxiliary atoms under the proper assumptions.

Proposition 1. *Let* Π *be a program and* $\mathbb{W}$ *be a set of interpretations. Then,*

$$\{M \cup k(\mathbb{W}) \mid M \in \text{SM}[\Pi^{\mathbb{W}}]\} \;=\; \text{SM}[T_0(\Pi); k(\mathbb{W})] \quad (4)$$

$$\text{SM}[\Pi^{\mathbb{W}}] \;=\; \text{SM}[T_0(\Pi); k(\mathbb{W})]_{|At(\Pi)} \quad (5)$$

The following definition generalizes the above example of a tester program.

Definition 4 (Tester Program)**.** *We say that an objective program* $T(\Pi)$ *is a* tester program *for a program* Π *if it safisfies the following condition:*

- $\mathbb{W} = \text{SM}[T(\Pi); k(\mathbb{W})]_{|At(\Pi)}$ *holds for every worldview* $\mathbb{W}$ *of* Π*, and*
- *every non-empty set of interpretations* $\mathbb{W}$ *that is not a worldview of* Π *satisfies* $\mathbb{W} \neq \text{SM}[T(\Pi); k(\mathbb{W})]_{|At(\Pi)}$

Using the second equality of Proposition 1, it is easy to see that program $T_0(\Pi)$ is a tester program for Π because any non-empty set of interpretations $\mathbb{W}$ is a worldview iff

$$\mathbb{W} \;=\; \text{SM}[\Pi^{\mathbb{W}}] \;=\; \text{SM}[T(\Pi); k(\mathbb{W})]_{|At(\Pi)}$$

Definition 4 does only require that $T(\Pi)$ behaves as the subjective reduct on actual worldviews. For non-worldviews, we only require that it does not produce false positives. This relaxation may allow the introduction of further optimizations in the future.

Definition 5 (Generate and test worldviews)**.** *If $G(\Pi)$ and $T(\Pi)$ respectively are a generator and a tester program for Π, by $wv(G(\Pi), T(\Pi))$ we denote the set of all stable models M of $G(\Pi)$ such that*

$$k(M) \quad = \quad \{ka \in K(\Pi) \mid a \in \text{CC}[T(\Pi); k(M)]\}. \quad (6)$$

The following theorem shows the correspondence between the worldviews of Π and members of $wv(G(\Pi), T(\Pi))$.

Theorem 2. *Let $G(\Pi)$ be any generator program for Π and $T(\Pi)$ be any tester program for Π. Then, there is a one-to-many correspondence between the worldviews of Π and the members of $wv(G(\Pi), T(\Pi))$ such that*

1. *if $\mathbb{W}$ is a worldview of Π, then there is a stable model M of $G(\Pi)$ with $k(\mathbb{W}) = k(M)$ satisfying* (6)*;*
2. *if M is a stable model of $G(\Pi)$ satisfying* (6)*, then the set of stable models $\text{SM}[T(\Pi); k(M)]$ is a worldview of Π.*

Theorem 2 shows that, if we have any pair of generator and tester programs, then we can compute the worldviews using Algorithm 1 where the fuction `Test`$(T(\Pi), M)$ amounts to check conditon (6), and function `BuildWorldView`(M) amounts to compute the stable models of the tester program under assumptions $k(M)$. When compared with existing generate-and-test-based solvers such as `EP-ASP` (Son et al. 2017) and `eclingo` (Cabalar et al. 2020), function `Test`$(T(\Pi), M)$ is simpler as it does not require an extra call for the computation of the brave consequences of the tester program. This is thanks to computing the normal form before invoking Algorithm 1[3].

Basic Generator Programs

In this section, we introduce two basic generator programs that can be used to compute the worldviews of a program. These two programs are inspired by the generator program used by `EP-ASP` and `eclingo`, respectively.

Tester program as a generator. We start by showing that the tester program $T_0(\Pi)$ is also a generator program for Π.

Proposition 2. *$T_0(\Pi)$ is a generator program for Π.*

Proposition 2 shows that we can use the tester program $T_0(\Pi)$ as a generator program for Π. With some preprocessing and the differences due to the different semantics implemented in `EP-ASP`—K15 vs G94—this is the strategy followed by this solver.

Generator with consistency constraints. Cabalar et al. (2020) noted that some stable models of $T_0(\Pi)$ can never correspond to a worldview of Π. Consider, for instance, the single rule program

$$b \leftarrow \mathbf{K}a \quad (\Pi_1)$$

and its corresponding $T_0(\Pi_1)$ program

$$b \leftarrow ka \qquad \{ka\} \leftarrow$$

which has two stable models $\emptyset$ and $\{b, ka\}$. The second stable model cannot correspond to a worldview $\mathbb{W}$ of the original program with $k(\mathbb{W}) = \{ka\}$ because, for every non-empty set of interpretations $\mathbb{W}$ such that $k(\mathbb{W}) = \{ka\}$, the empty set is the only stable model of $\Pi^{\mathbb{W}}$. Hence, to be a worldview $\mathbb{W}$ must satisfy $\mathbb{W} \not\models \mathbf{K}a$, which is a contradiction with $k(\mathbb{W}) = \{ka\}$. Using this observation, Cabalar et al. (2020) introduced a more refined generator program that reduces the number of candidates by adding constraints that ensure that the stable models of the generator program that contain ka must also contain a.

By $G_0(\Pi)$ we denote the program obtained from $T_0(\Pi)$ by adding a constraint of the form

$$\bot \leftarrow ka \wedge \neg a \quad (7)$$

for each atom $ka \in K(\Pi)$. Continuing with our running example, we can see that $\{b, ka\}$ does not satisfy (7) and, thus, it is not a stable model of $G_0(\Pi_1)$.

Proposition 3. *$G_0(\Pi)$ is a generator program for Π.*

Proposition 3 shows that we can reduce the number of candidates by adding consistency constraints of the form of (7) to the generator program. If we consider programs in normal form, Algorithm 1 with generator $G_0(\Pi)$ and tester $T_0(\Pi)$ is the core of the epistemic solver `eclingo`. However, `eclingo` does not compute the normal form of a program, so its algorithm is slightly more complicated than Algorithm 1. Hence, this is a proof `eclingo` is correct for programs in normal form[4].

Corollary 1. *Algorithm 1 with generator $G_0(\Pi)$ and tester $T_0(\Pi)$ correctly computes the worldviews of Π.*

Generator with Epistemic Propagation

In a generate-and-test approach, the tester needs to check every stable model of the guess program. Each of these checks requires a linear amount of calls to an answer set solver to compute the corresponding cautions consequences. Each of these calls is Σ_2^P-complete. Hence, reducing the number of tester checks is crucial to achieve a fast solver. In the previous section, we saw how we can reduce the number of candidates by adding consistency constraints to the generator program. In this section, we introduce a new generator program that can exponentially reduce the number of candidates by propagating the consequences of epistemic literals in the generator program.

Example 1. *Let us consider the following program:*

$$\begin{aligned} a_i &\leftarrow \neg\mathbf{K}na_i \quad \textit{for } 0 \leq i \leq n \\ na_i &\leftarrow \neg a_i \quad \textit{for } 0 \leq i \leq n \\ g &\leftarrow a_i \quad \textit{for } 0 \leq i \leq n \\ \bot &\leftarrow \mathbf{K}g \end{aligned} \quad (\Pi_2^n)$$

The two rules first are the result of normalizing rule $a_i \leftarrow \neg\mathbf{K}\neg a_i$, usually used as a kind of subjective choice that generates worldviews $[\emptyset]$ and $[\{a_i\}]$. This program has

[3]Faber and Woltran (2011) describe an alternative way of checking when a candidate is a worldview using manifold programs. The manifold program is quadratic in size and we are not aware of any tool implementing this idea.

[4]Cabalar et al. (2020) described the algorithm of `eclingo` but they did not show its correctness.

a unique worldview $[\emptyset]$. *Let us now consider its guess program* $G_0(\Pi_2^n)$*:*

$$\begin{aligned} a_i &\leftarrow \neg kna_i && \textit{for } 0 \leq i \leq n \\ na_i &\leftarrow \neg a_i && \textit{for } 0 \leq i \leq n \\ g &\leftarrow a_i && \textit{for } 0 \leq i \leq n \\ \bot &\leftarrow kg \\ \{kna_i\} &\leftarrow && \textit{for } 0 \leq i \leq n \\ \{kg\} &\leftarrow \\ \bot &\leftarrow kna_i \wedge \neg na_i && \textit{for } 0 \leq i \leq n \\ \bot &\leftarrow kg \wedge \neg g \end{aligned}$$

This candidate generator program has 2^n *stable models of the form*

$$M \cup \{na_i \mid a_i \notin M\} \cup \{kna_i \mid a_i \notin M\} \cup \{g \mid M \neq \emptyset\}$$

with $M \subseteq \{a_1, \ldots, a_n\}$*. Only the stable model corresponding to* $M = \emptyset$ *leads to a worldview of the original program, while the other* $2^n - 1$ *stable models do not correspond to any worldview.*

We can achieve an exponential speed-up if we can identify the candidates that do not pass the tester check within the generator program. We can observe that, in every worldview $\mathbb{W}$ of Π_2^n, if $\mathbb{W} \models \neg \mathbf{K} na_i$ for any $1 \leq i \leq n$, then every $I \in \mathbb{W}$ must satisfy $I \models a_i$ because of the first rule. Hence, every $I \in \mathbb{W}$ must satisfy $I \models g$ because of the third rule, and we obtain $\mathbb{W} \models \mathbf{K} g$. This contradicts the constraint $\bot \leftarrow \mathbf{K} g$ and, thus, no worldview can satisfy $\neg \mathbf{K} na_i$. We can implement this reasoning within the generator program by adding rules:

$$\begin{aligned} pka_i &\leftarrow \neg kna_i && \text{for } 1 \leq i \leq n \\ pkg &\leftarrow pka_i && \text{for } 1 \leq i \leq n \\ \bot &\leftarrow pkg \wedge \neg kg \end{aligned}$$

The new fresh atoms pka_i and pkg are respectively used to represent that $\mathbf{K} a_i$ and $\mathbf{K} g$ are a consequence propagated from the choices made in the generator program. The meaning of pkg is different from the meaning of atom kg that represents the choice made in the generator program rather than an inferred consequence. The unintended candidates are removed by the constraint that ensures that if we conclude pkg, then kg must also hold.

Let us now describe the general idea of the propagation of epistemic literals by introducing a new guess program $G_1(\Pi)$. For each atom $a \in At(\Pi)$, we introduce new fresh atoms pka and $pkna$ that respectively represent that $\mathbf{K} a$ and $\mathbf{K} \neg a$ are consequences propagated from the choices made in the generator program. We also introduce a new fresh atom $pknr_j$ for each rule r_j of Π that represents that the body of r_j is not satisfied by any interpretation of the worldview corresponding to the current candidate. We use the following notation. If L is a literal, then $\overline{L}$ is the complement of L, that is $\overline{L}$ is $\neg L$ if L is an atom or a subjective atom, and $\overline{L}$ is L' if L is of the form $\neg L'$. Furthermore, if L is a subjective literal, then kL is a literal obtained by replacing every occurrence of the form $\mathbf{K} a$ by ka.

By $pk(\Pi)$, we denote the program containing rule

$$pka_1 \leftarrow pka_2, \ldots, pka_k, pkna_{k+1}, \ldots, pkna_m, kL_{m+1}, \ldots, kL_n \tag{8}$$

for every rule $r_i \in \Pi$ of the form

$$a_1 \leftarrow a_2, \ldots, a_k, \neg a_{k+1}, \ldots, \neg a_m, L_{m+1}, \ldots, L_n \tag{9}$$

with each L_j being a subjective literal; plus rules of the form

$$\begin{aligned} pknr_i &\leftarrow pkna_j && \text{for } 1 \leq j \leq k \\ pknr_i &\leftarrow pka_j && \text{for } l+1 \leq j \leq m \\ pknr_i &\leftarrow \overline{kL_j} && \text{for } m+1 \leq j \leq n \end{aligned}$$

for every rule $r_i \in \Pi$ of the form

$$H \leftarrow a_2, \ldots, a_k, \neg a_{k+1}, \ldots, \neg a_m, L_{m+1}, \ldots, L_n \tag{10}$$

with H of the form $a_1 \vee \ldots \vee a_l$ and each L_j a subjective literal; plus a rule of the form

$$pkna \leftarrow pknr_1, \ldots, pknr_k \tag{11}$$

for each atom $a \in At(\Pi)$ and where $r_1, \ldots, r_k$ are the rules of Π that have a in the head. By $G_1(\Pi)$ we denote the program obtained from $G_0(\Pi)$ by adding the rules of $pk(\Pi)$ plus a *consistency constraint* of the form

$$\bot \leftarrow pka \wedge \neg ka \tag{12}$$

for each atom $ka \in K(\Pi)$.

Lemma 1. *Let* $\mathbb{W}$ *be a non-empty set of interpretations, and* M *be an interpretation such that* $k(\mathbb{W}) = k(M)$*. Then,*

- *If* M *is a stable model of* $G_0(\Pi)$*, then there is a unique stale model* M' *of* $G_0(\Pi) \cup pk(\Pi)$ *of the form* $M' = M \cup M^p \cup M^n$ *with*

$$M^p \subseteq \{pka \mid \text{SM}[\Pi^{\mathbb{W}}] \models \mathbf{K} a\} \tag{13}$$

$$M^n \subseteq \{pkna \mid \text{SM}[\Pi^{\mathbb{W}}] \models \mathbf{K} \neg a\} \tag{14}$$

- *If* M' *is a stable model of program* $G_0(\Pi) \cup pk(\Pi)$*, then* $M' \cap At(G_0(\Pi))$ *is a stable model of* $G_0(\Pi)$*.*

Main Theorem. $G_1(\Pi)$ *is a generator program for* Π *and, thus, Algorithm 1 with generator* $G_1(\Pi)$ *and tester* $T_0(\Pi)$ *correctly computes the worldviews of* Π*.*

Proof. Condition 1 for generator program. Pick a worldview $\mathbb{W}$ of Π. By Proposition 3, there is a unique stable model M of $G_0(\Pi)$ satisfying $k(\mathbb{W}) = k(M)$ and $M_{|At(\Pi)} \in \mathbb{W}$. By Lemma 1, there is a unique stable model M' of $G_0(\Pi) \cup pk(\Pi)$ of the form $M' = M \cup M^p \cup M^n$ with M^p and M^n satisfying (13) and (14). Furthermore, $M'_{|At(\Pi)} = M_{|At(\Pi)} \in \mathbb{W}$ and $k(M) = k(M')$. It only remains to be shown that M' is a stable model of $G_1(\Pi)$, that is, that it satisfies the consistency constraints of the form of (12). Pick $pka \in M^p$. Then, $\text{SM}[\Pi^{\mathbb{W}}] \models \mathbf{K} a$ and, thus, $\mathbb{W} \models \mathbf{K} a$ and $ka \in M$ and, thus, the constraints is satisfied.

Condition 2 for generator program. Pick now a stable model of $G_1(\Pi)$ and let $\mathbb{W}$ be any set of interpretations such that $k(\mathbb{W}) = k(M)$. By Lemma 1, $M \cap At(G_0(\Pi))$ is a stable model of $G_0(\Pi)$ and, since $G_0(\Pi)$ is a generator program for Π, it follows that $M_{|At(\Pi)} \in \text{SM}[\Pi^{\mathbb{W}}]$.

Therefore $G_1(\Pi)$ is a generator program and, by Theorem 2, Algorithm 1 with generator $G_1(\Pi)$ and tester $T_0(\Pi)$ correctly computes the worldviews of Π. □

This Theorem shows that we can use $G_1(\Pi)$ as a generator program for Π instead of $G_0(\Pi)$. Let us show now that using $G_1(\Pi)$ instead of $G_0(\Pi)$ is actually beneficial when using Algorithm 1.

Proposition 4. *The following properties hold:*

1. $|\text{SM}[G_1(\Pi)]| \leq |\text{SM}[G_0(\Pi)]|$ *for every program* Π,
2. *for every program* Π *we can compute a stable model of* $G_1(\Pi)$ *from a stable model of* $G_0(\Pi)$ *in linear time, and*
3. *there is a family of programs* $\Pi_1, \Pi_2, \ldots$ *such that* $2^i|\text{SM}[G_1(\Pi_i)]| \leq |\text{SM}[G_0(\Pi_i)]|$ *and* $\mathcal{O}(s(\Pi_i)) = i$.

where $s(\Pi)$ *denotes the size of program* Π.

The first two properties show that using $G_1(\Pi)$ instead of $G_0(\Pi)$ only incurs a linear overhead in computing the stable models of the generator program. In particular, the first property ensures that the loop between lines 2 and 10 of Algorithm 1 is never required to do more iterations when using $G_1(\Pi)$ instead of $G_0(\Pi)$. The third property shows that using $G_1(\Pi)$ can reduce the number of candidates that need to be checked exponentially. As an example of such a family of programs, consider programs $(\Pi_2^n)_{n\geq 1}$ of Example 1.

Lemma 1 also implies an interesting property that allows us to skip some of the tester checks. By (13) and (14), it follows that any stable model M of $G_1(\Pi)$ that satisfies the following two conditions corresponds to a worldview of Π and, thus, the tester check can be skipped:

1. every $ka \in M$ satisfies $pka \in M$, and
2. every $ka \notin M$ satisfies $pkna \in M$.

Hence, when using $G_1(\Pi)$ as a generator program, function `Test`$(T(\Pi), M)$ first checks these two conditions, and only if any of them fails it checks condition (6). Checking these two conditions is a linear time operation while checking condition (6) requires a linear amount of calls to a Σ_2^P-complete oracle.

Implementation and Experimental Evaluation

Implementation. We implement a new solver for epistemic logic programs based on Algorithm 1. Our implementation is built on top of version 5.7 of the ASP solver `clingo` (Gebser et al. 2019) using Python and ASP. We extend the language of `clingo` by allowing literals of the form `&k`$\{L\}$ in the body of rules, with L being a literal of the forms A, `not` A, or `not not` A for some atom A in the usual syntax of `clingo`. An interesting side effect of using `clingo` in this way is that our solver accepts most of the features of `clingo` such as aggregates, choice rules, intervals, pools, etc. Another important aspect of our implementation is that it easily allows us to use different generator and tester programs. To achieve this, we rely on *metaprogramming*, where the ground program is reified as a set of facts and we can use an ASP program to produce the generator and tester programs (Kaminski et al. 2023). As a result, we can change the generator and tester programs by only changing the ASP metaprogram. This is released as a new version of `eclingo` (https://github.com/potassco/eclingo).

	G_1		G_0	
Benchmark	`#solved`	`time`	`#solved`	`time`
Eligibility	145	4.03	145	5.38
Yale	7	0.009	7	0.112
Bomb	131	245.66	13	558.40
All	283	132.912	166	300.435

Table 1: Instances for each benchmark solved by the different versions of the generator program.

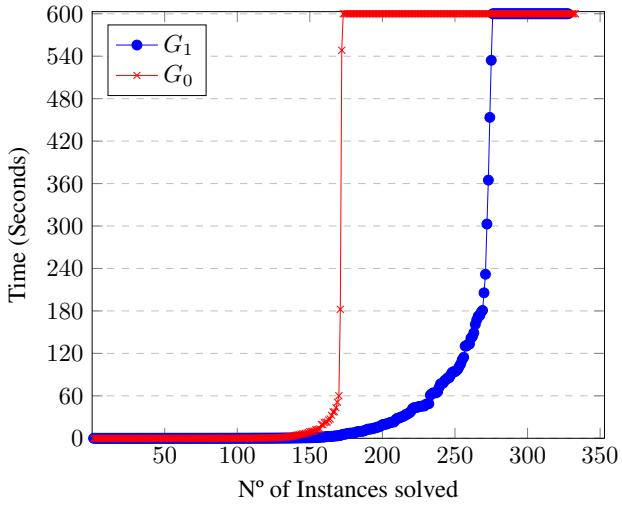

Figure 1: Comparison of our implementation of Algorithm 1 with two different versions of the generator program.

Experimental Evaluation. For the experimental evaluation, we use the well-established benchmark suite by Son et al. (2017). It consists of three problems: the *Eligibility* problem that represents reasoning in disjunctive databases (Gelfond 1991), and the *Yale Shooting* and *Bomb in the Toilet* problems that are instances of conformant planning. The original suite consists of 58 instances (25 of the Eligibility problem, 7 of the Yale problem and 26 of the Bomb problem). We expand this suite by adding 268 new instances for a total of 326 instances (145 Eligibility problems, 7 Yale problems and 174 for Bomb problems). The new instances are automatically generated by increasing the number of students in the Eligibility problem and the number of packages and toilets in the Bomb problem. We set a timeout of 600s and ran our experiments on a machine powered by an Intel Core Processor (Broadwell) with 12 CPUs running at 2095.078 MHz and 100GB of RAM. The OS is Red Hat Enterprise Linux Server 7.9. The code to repeat the benchmarks can be found at https://github.com/krr-uno/eclingo-benchmark.

Results. We first compare our implementation of Algorithm 1 with two different versions of the generator program, denoted G_0 and G_1. Table 1 shows the number of instances solved within the timeout and the average solving time by each version of the generator program. On computing average times, instances that did not solve within the timeout were assigned a time of 600s. Figure 1 reports the cactus

	G_1		G_0		eclingo		EP−ASP		selp		elp2qasp		EP−ASP^{se}	
Benchmark	#solved	time	#solved	time	#solved	time	#solved	time	#solved	time	#solved	time	#solved	time
Eligibility	145	17.982	145	18.621	124	134.244	28	523.102	60	380.366	6	577.303	-	-
Yale	7	0.377	7	0.469	7	0.157	7	30.377	5	223.296	7	1.926	7	0.109
Bomb	131	246.039	15	556.396	17	561.52	45	471.759	8	572.809	8	572.663	86	327.900
All	283	139.328	166	306.350	148	359.42	80	485.118	73	479.708	21	562.472	121	407.684

Table 2: Instances for each benchmark solved by the different epistemic solvers

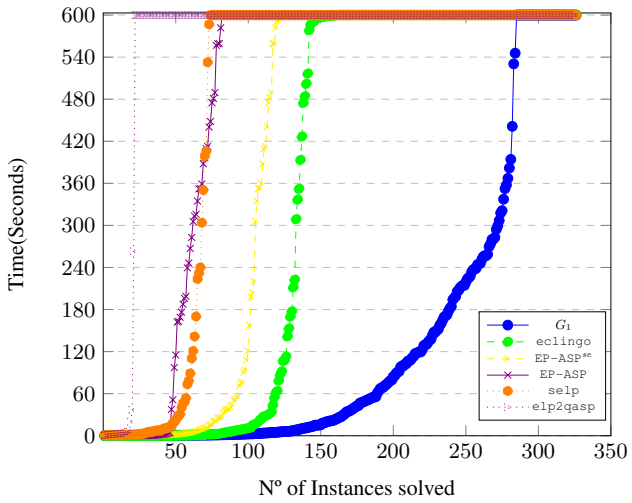

Figure 2: Comparison of all Epistemic Solvers Eligible, Yale and Bomb Problems.

plot for these two versions of our solver. Using G_1 solves more instances than using G_0 with any timeout greater than 0.546 seconds. There are 49 easy instances solved under that timeout where using G_0 outperforms G_1.

We also compare our implementation with the state-of-the-art epistemic solvers EP−ASP (Son et al. 2017), the previous version of eclingo[5] (Cabalar et al. 2020), selp[6] (Bichler, Morak, and Woltran 2020), and elp2qasp[7] (Faber and Morak 2023). For EP−ASP, we use the version[8] created by Faber and Morak (2023) to run with clingo 5.4. Table 2 shows the number of instances solved within the timeout and the average expended time by each solver. In this case, expended time comprises both grounding and solving because we do not have access to the solving time of all solvers. Our implementation of Algorithm 1 with the generator program G_1 solves the most instances, dominating the other solvers across all benchmarks. It solves 91% more instances than eclingo and more than double than EP−ASP. When looking at average times, our solver achieves a speed-up of ∼3.3x and ∼3.8x when compared with the previous version of eclingo and EP−ASP, respectively[9]. When looking at individual benchmarks, we can see that eclingo solves more instances than EP−ASP mostly thanks to the Eligibility problem, where it solves more than 4 times the number of instances. When looking at the Bomb problem things reverse and EP−ASP solves ∼2.6x more instances than eclingo. Our new solver successfully solves all 145 instances of the Eligibility problem and almost ∼7.7x more instances than EP−ASP in the Bomb problem. When looking at average times, our solver is ∼7.5 times faster than eclingo in the Eligibility problem and ∼2.4x faster than EP−ASP in the Bomb problem. The Yale Problem seems quite easy for state-of-the-art solvers, with all but selp solving all 7 instances, and both versions of eclingo solving every instance in less than a second. It is also worth mentioning that among state-of-the-art solvers, generate-and-test-based solvers outperform translational-based solvers, with the three top solvers being generate-and-test-based and the two bottom ones being translational-based. Finally, EP−ASP has a conformant planning mode that uses domain-specific heuristics that gives it a significant advantage on the Yale and Bomb problems. This is reported in Table 2 and Figure 2 as EP−ASP^{se}. Even with this advantage, our solver is ∼1.5x faster in the Bomb problem and it solves ∼50% more of its instances.

Conclusions

We present a general framework to study generate-and-test-based solvers for epistemic logic programs (ELPs). In this framework, we first compute the normal form of the epistemic program. Then, we check all the stable models of a generator program by computing the cautious consequences of a tester program. We provided sufficient conditions on the generator and tester programs for the correctness of the solvers built using this algorithm. We instantiate this framework with generator programs corresponding to the existing generate-and-test-based solvers (EP−ASP and eclingo) and introduce and new generator program based on the idea of propagating epistemic consequences. We formally prove that this new generator program can exponentially reduce the number of candidates while only incurring a linear overhead. We experimentally evaluate our new solver and show that it outperforms existing solvers by achieving a ∼3.3x speed-up and solving 91% more instances on well-known benchmarks. Even when comparing with EP−ASP^{se} on conformant planning problems our solver is ∼1.5x faster and solves ∼50% more instances, despite the latter using domain-specific heuristics and our solver being based only on general-purpose algorithms for ELPs. For future work, we plan to study the impact of splitting (Cabalar, Fandinno, and Fariñas del Cerro 2021) in worldview computation and the use of general-purpose and domain-specific heuristics to further improve the performance of our solver. For domain-specific heuristics, we will focus on conformant planning and adapt the heuristics used by EP−ASP^{se} and the one used for classical planning in ASP (Gebser et al. 2013).

[5]https://github.com/potassco/eclingo/releases/tag/v0.2.1

[6]https://dbai.tuwien.ac.at/proj/selp/

[7]elp2qasp was shared with us by its authors.

[8]https://github.com/mmorak/EP-ASP

[9]The speed-up is computed by dividing the total time of the solver divided by the total time of our solver. Only instances solved by at least one solver are considered.

Acknowledgements

This research is partially supported by NSF CAREER award 2338635. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

Balduccini, M.; Lierler, Y.; and Woltran, S., eds. 2019. *Proceedings of the Fifteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'19)*, volume 11481 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag.

Balduccini, M.; and Son, T., eds. 2011. *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning: Essays Dedicated to Michael Gelfond on the Occasion of his 65th Birthday*, volume 6565 of *Lecture Notes in Computer Science*. Springer-Verlag.

Bichler, M.; Morak, M.; and Woltran, S. 2020. selp: A Single-Shot Epistemic Logic Program Solver. *Theory and Practice of Logic Programming*, 20(4): 435–455.

Brewka, G.; Eiter, T.; and Truszczyński, M. 2011. Answer set programming at a glance. *Communications of the ACM*, 54(12): 92–103.

Cabalar, P.; Fandinno, J.; and Fariñas del Cerro, L. 2019a. Founded World Views with Autoepistemic Equilibrium Logic. In (Balduccini, Lierler, and Woltran 2019), 134–147.

Cabalar, P.; Fandinno, J.; and Fariñas del Cerro, L. 2019b. Splitting Epistemic Logic Programs. In (Balduccini, Lierler, and Woltran 2019), 120–133.

Cabalar, P.; Fandinno, J.; and Fariñas del Cerro, L. 2020. Autoepistemic Answer Set Programming. *Artificial Intelligence*, 289: 103382.

Cabalar, P.; Fandinno, J.; and Fariñas del Cerro, L. 2021. Splitting Epistemic Logic Programs. *Theory and Practice of Logic Programming*, 21: 296–316.

Cabalar, P.; Fandinno, J.; Garea, J.; Romero, J.; and Schaub, T. 2020. eclingo: A solver for Epistemic Logic Programs. *Theory and Practice of Logic Programming*, 20(5): 834–847. https://github.com/potassco/eclingo.

Dantsin, E.; Eiter, T.; Gottlob, G.; and Voronkov, A. 2001. Complexity and Expressive Power of Logic Programming. *ACM Computing Surveys*, 33(3): 374–425.

Faber, W.; and Morak, M. 2023. Evaluating Epistemic Logic Programs via Answer Set Programming with Quantifiers. In *Proceedings of the Thirty-seventh National Conference on Artificial Intelligence (AAAI'23)*, 6322–6329. AAAI Press.

Faber, W.; Morak, M.; and Chrpa, L. 2021. Determining Action Reversibility in STRIPS Using Answer Set and Epistemic Logic Programming. *Theory and Practice of Logic Programming*, 21(5): 646–662.

Faber, W.; and Woltran, S. 2011. Manifold Answer-Set Programs and Their Applications. In (Balduccini and Son 2011), 44–63.

Fandinno, J.; Faber, W.; and Gelfond, M. 2022. Thirty years of Epistemic Specifications. *Theory and Practice of Logic Programming*, 22(6): 1043–1083.

Fandinno, J.; and Lillo, L. 2024. Solving Epistemic Logic Programs using Generate-and-Test with Propagation. *CoRR*, abs/2410.22130.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2019. Multi-shot ASP solving with clingo. *Theory and Practice of Logic Programming*, 19(1): 27–82.

Gebser, M.; Kaufmann, B.; Otero, R.; Romero, J.; Schaub, T.; and Wanko, P. 2013. Domain-specific Heuristics in Answer Set Programming. In desJardins, M.; and Littman, M., eds., *Proceedings of the Twenty-seventh National Conference on Artificial Intelligence (AAAI'13)*, 350–356. AAAI Press.

Gelfond, M. 1991. Strong Introspection. In Dean, T.; and McKeown, K., eds., *Proceedings of the Nineth National Conference on Artificial Intelligence (AAAI'91)*, 386–391. AAAI Press.

Gelfond, M. 1994. Logic Programming and Reasoning with Incomplete Information. *Annals of Mathematics and Artificial Intelligence*, 12(1-2): 89–116.

Gelfond, M. 2011. New Semantics for Epistemic Specifications. In Delgrande, J.; and Faber, W., eds., *Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11)*, volume 6645 of *Lecture Notes in Artificial Intelligence*, 260–265. Springer-Verlag.

Gelfond, M.; and Lifschitz, V. 1988. The Stable Model Semantics for Logic Programming. In Kowalski, R.; and Bowen, K., eds., *Proceedings of the Fifth International Conference and Symposium of Logic Programming (ICLP'88)*, 1070–1080. MIT Press.

Gelfond, M.; and Lifschitz, V. 1991. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9: 365–385.

Gelfond, M.; and Przymusinska, H. 1993. Reasoning on Open Domains. In Pereira, L.; and Nerode, A., eds., *Proceedings of the Second International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'93)*, volume 928 of *Lecture Notes in Artificial Intelligence*, 397–413. Springer-Verlag.

Kahl, P.; Leclerc, A.; and Son, T. 2016. A Parallel Memory-efficient Epistemic Logic Program Solver: Harder, Better, Faster. In Bogaerts, B.; and Harrison, A., eds., *Proceedings of the Ninth Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP'16)*.

Kahl, P.; Watson, R.; Balai, E.; Gelfond, M.; and Zhang, Y. 2015. The language of epistemic specifications (refined) including a prototype solver. *Journal of Logic and Computation*.

Kahl, P.; Watson, R.; Balai, E.; Gelfond, M.; and Zhang, Y. 2020. The language of epistemic specifications (refined) including a prototype solver. *Journal of Logic and Computation*, 30(4): 953–989.

Kaminski, R.; Romero, J.; Schaub, T.; and Wanko, P. 2023. How to Build Your Own ASP-based System?! *Theory and Practice of Logic Programming*, 23(1): 299–361.

Leclerc, A.; and Kahl, P. 2018. A survey of advances in epistemic logic program solvers. arXiv:1809.07141.

Lifschitz, V. 2008. What Is Answer Set Programming? In Fox, D.; and Gomes, C., eds., *Proceedings of the Twenty-third National Conference on Artificial Intelligence (AAAI'08)*, 1594–1597. AAAI Press.

Lifschitz, V. 2019. *Answer Set Programming*. Springer-Verlag.

Schaub, T.; and Woltran, S. 2018. Special Issue on Answer Set Programming. *Künstliche Intelligenz*, 32(2-3): 101–103.

Shen, Y.; and Eiter, T. 2016. Evaluating Epistemic Negation in Answer Set Programming. *Artificial Intelligence*, 237: 115–135.

Shen, Y.; and Eiter, T. 2017. Evaluating Epistemic Negation in Answer Set Programming (Extended Abstract). In Sierra, C., ed., *Proceedings of the Twenty-sixth International Joint Conference on Artificial Intelligence (IJCAI'17)*, 5060–5064. IJCAI/AAAI Press.

Son, T.; Le, T.; Kahl, P.; and Leclerc, A. 2017. On Computing World Views of Epistemic Logic Programs. 1269–1275.

Truszczynski, M. 2011. Revisiting Epistemic Specifications. In (Balduccini and Son 2011), 315–333.