Loops: Relevant or Redundant?

Martin Gebser and Torsten Schaub

Institut für Informatik, Universität Potsdam, Postfach 900327, D–14439 Potsdam

Abstract. Loops and the corresponding loop formulas play an important role in answer set programming. On the one hand, they are used for guaranteeing correctness and completeness in SAT-based answer set solvers. On the other hand, they can be used by conventional answer set solvers for finding unfounded sets of atoms. Unfortunately, the number of loops is exponential in the worst case. We demonstrate that not all loops are actually needed for answer set computation. Rather, we characterize the subclass of *elementary loops* and show that they are sufficient and necessary for selecting answer sets among the models of a program's completion. Given that elementary loops cannot be distinguished from general ones in atom dependency graphs, we show how the richer graph structure provided by *body-head dependency graphs* can be exploited for this purpose.

1 Introduction

The success of Answer Set Programming (ASP) is largely due to the availability of efficient solvers, e.g. [1, 2]. A similar situation is encountered in the area of satisfiability checking (SAT), in which manifold solvers show an impressive performance. This has led to ASP solvers mapping answer set computation to model generation via SAT solvers [3–5]. Since the answer sets of a program form a subset of its classical models, however, additional measures must be taken for eliminating models that are no answer sets. To this end, a program is transformed via *Clark's completion* [6]. The models of the resulting completed program are called *supported models*; they are generally still a superset of the program's answer sets. However, supported models coincide with answer sets on *tight* programs, that is, programs having an acyclic positive atom dependency graph [7]. For example, the program $\{p \leftarrow p\}$ is non-tight; it admits a single empty answer set, while its completion, $\{p \equiv p\}$, has two models, $\emptyset$ and $\{p\}$. While early SAT-based ASP solvers [3] reject non-tight programs, the next generation of solvers, e.g. [4, 5], exploits the circular structures within the atom dependency graph for handling non-tight programs. As put forward in [4], the idea is to extend a program's completion by *loop formulas* in order to eliminate the supported models that are no answer sets. Loop formulas are generated from *loops*, which are sets of atoms that circularly depend upon each other in a program's atom dependency graph. Unfortunately, a program may yield exponentially many loops in the worst case [8], so that exponentially many loop formulas may be necessary for filtering out the program's answer sets.

We show that not all loops are needed for selecting the answer sets among the models of a program's completion. Rather, we introduce the subclass of *elementary loops*, whose corresponding loop formulas are sufficient for determining the answer sets of a program from its completion. Moreover, elementary loops are essential in the sense that

generally none on their loop formulas can be omitted without reintroducing undesired supported models. Given that elementary loops cannot be distinguished from general ones in atom dependency graphs, we show how the richer graph structure provided by *body-head dependency graphs* [9] can be exploited for recognizing elementary loops. Body-head dependency graphs extend atom dependency graphs by an explicit representation of rules' bodies. Their richer graph structure allows for identifying elementary loops in an efficient way. Finally, we show that the set of elementary loops lies between the set of $\subseteq$-minimal loops and the set of all loops. As a consequence, there may still be an exponential number of elementary loops, since there may already be an exponential number of $\subseteq$-minimal loops in the worst case. On the other hand, we show that there may also be exponentially fewer elementary loops than general ones in the best case.

The next section provides the background of this paper. In Section 3, we characterize elementary loops and show that they are sufficient and, generally, necessary for capturing answer sets. Section 4 introduces body-head dependency graphs as a device for recognizing elementary loops. In Section 5, we provide lower and upper bounds for programs' elementary loops. We conclude with Section 6.

2 Background

A *logic program* is a finite set of *rules* of form $a \leftarrow b_1, \ldots, b_m, not\ c_1, \ldots, not\ c_n$ where $a, b_1, \ldots, b_m, c_1, \ldots, c_n$ are atoms for $m \geq 0$, $n \geq 0$. Given such a rule r, we denote its *head* a by $head(r)$ and its *body* $\{b_1, \ldots, b_m, not\ c_1, \ldots, not\ c_n\}$ by $body(r)$. Furthermore, we let $body^+(r) = \{b_1, \ldots, b_m\}$ and $body^-(r) = \{c_1, \ldots, c_n\}$ be the *positive* and *negative body* of r, respectively. The set of bodies in logic program Π is $body(\Pi) = \{body(r) \mid r \in \Pi\}$. The set of atoms appearing in Π is given by $atom(\Pi)$. A logic program Π is *basic*, if $body^-(r) = \emptyset$ for every rule $r \in \Pi$. The smallest set of atoms closed under basic program Π is denoted by $Cn(\Pi)$. The *reduct* of a logic program Π relative to a set X of atoms is the basic program $\Pi^X = \{head(r) \leftarrow body^+(r) \mid r \in \Pi, body^-(r) \cap X = \emptyset\}$. An *answer set* of a logic program Π is a set X of atoms satisfying $X = Cn(\Pi^X)$.

The *Clark completion* of a program can be defined as follows [6]. For a logic program Π and a rule $r \in \Pi$, define

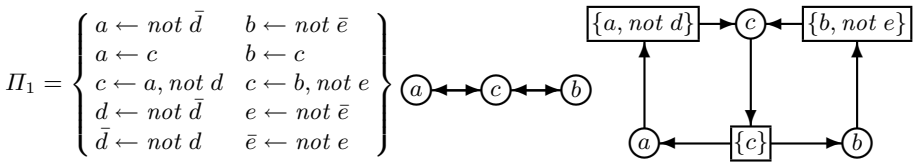
$$comp(r) = \bigwedge_{b \in body^+(r)} b \wedge \bigwedge_{c \in body^-(r)} \neg c\ ,$$
$$comp(\Pi) = \{a \equiv \bigvee_{r \in \Pi, head(r) = a} comp(r) \mid a \in atom(\Pi)\}\ .$$

An answer set of Π is also a model[1] of $comp(\Pi)$. Models of $comp(\Pi)$ are also called *supported models* of Π.

As shown in [4], answer sets can be distinguished among the supported models by means of *loops* in atom dependency graphs (cf. [10, 11]). To be precise, the *positive atom dependency graph* of a program Π is the directed graph $(atom(\Pi), E(\Pi))$ where $E(\Pi) = \{(b, a) \mid r \in \Pi, b \in body^+(r), head(r) = a\}$. A set $L \subseteq atom(\Pi)$ is a *loop* in Π, if $(L, E(\Pi, L))$ is a strongly connected subgraph[2] of the positive atom

[1] That is, an interpretation is represented by its entailed set of atoms.

[2] A (sub)graph is strongly connected, if there is a path between any pair of contained nodes.



(a) Logic program Π_1.

(b) Atom dependency graph of Π_1 (restricted to connected nodes).

(c) Body-head dependency graph of Π_1 (restricted to connected nodes).

Fig. 1. Logic program Π_1 yielding $loop(\Pi_1) = \{\{a, c\}, \{b, c\}, \{a, b, c\}\}$.

dependency graph $(atom(\Pi), E(\Pi))$ such that $E(\Pi, L) = E(\Pi) \cap (L \times L) \neq \emptyset$. Given a loop L in Π, we partition the rules whose heads are in L into two sets, namely

$$R^+(\Pi, L) = \{r \in \Pi \mid head(r) \in L,\ body^+(r) \cap L \neq \emptyset\}\ ,$$
$$R^-(\Pi, L) = \{r \in \Pi \mid head(r) \in L,\ body^+(r) \cap L = \emptyset\}\ .$$

The *loop formula* associated with loop L is

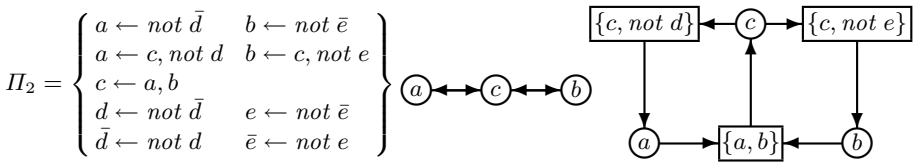
$$LF(\Pi, L) = \neg\big(\bigvee_{r \in R^-(\Pi, L)} comp(r)\big) \rightarrow \bigwedge_{a \in L} \neg a\ . \tag{1}$$

We denote the set of all loops in Π by $loop(\Pi)$. The set of all loop formulas of Π is $LF(\Pi) = \{LF(\Pi, L) \mid L \in loop(\Pi)\}$. As shown in [4], a set X of atoms is an answer set of a logic program Π iff X is a model of $comp(\Pi) \cup LF(\Pi)$.

For illustration, consider Program Π_1 in Figure 1(a). This program has four answer sets: $\{a, b, d, e\}$, $\{b, \bar{d}, e\}$, $\{a, d, \bar{e}\}$, and $\{\bar{d}, \bar{e}\}$. Apart from these, Π_1 has three additional supported models: $\{a, b, c, \bar{d}, e\}$, $\{a, b, c, d, \bar{e}\}$, and $\{a, b, c, \bar{d}, \bar{e}\}$. Observe that each additional supported model is a superset of some answer set. A closer look reveals that all of them contain atoms a, b, and c, which are the ones being involved in loops. In fact, the loops are responsible for the supported models that are no answer sets since they allow for a circular support among atoms. To see this, consider the positive atom dependency graph of Π_1 in Figure 1(b). (We omit atoms d, $\bar{d}$, e, and $\bar{e}$ since they are not involved in any positive dependencies.) We can identify three loops: $\{a, c\}$, $\{b, c\}$, and $\{a, b, c\}$. Each of them induces a strongly connected subgraph that reflects the possibility of circular derivations among these atoms (via rules in $R^+(\Pi_1, \{a, c\})$, $R^+(\Pi_1, \{b, c\})$, and $R^+(\Pi_1, \{a, b, c\})$). This circular behavior can be counterbalanced by the corresponding loop formulas

$$\begin{aligned}
LF(\Pi_1, \{a, c\}) &= \neg(\neg\bar{d} \vee (b \wedge \neg e)) \rightarrow \neg a \wedge \neg c \ \equiv\ \bar{d} \wedge (\neg b \vee e) \rightarrow \neg a \wedge \neg c\ ,\\
LF(\Pi_1, \{b, c\}) &= \neg(\neg\bar{e} \vee (a \wedge \neg d)) \rightarrow \neg b \wedge \neg c \ \equiv\ \bar{e} \wedge (\neg a \vee d) \rightarrow \neg b \wedge \neg c\ ,\\
LF(\Pi_1, \{a, b, c\}) &= \neg(\neg\bar{d} \vee \neg\bar{e}) \rightarrow \neg a \wedge \neg b \wedge \neg c \ \equiv\ \bar{d} \wedge \bar{e} \rightarrow \neg a \wedge \neg b \wedge \neg c\ .
\end{aligned}$$

While these formulas are satisfied by all answer sets of Π_1, one of them is falsified by each of the additional supported models. In this way, $LF(\Pi_1, \{a, c\})$ eliminates $\{a, b, c, \bar{d}, e\}$, $LF(\Pi_1, \{b, c\})$ excludes $\{a, b, c, d, \bar{e}\}$, and $LF(\Pi_1, \{a, b, c\})$ for-



(a) Logic program Π_2.

(b) Atom dependency graph of Π_2 (restricted to connected nodes).

(c) Body-head dependency graph of Π_2 (restricted to connected nodes).

Fig. 2. Logic program Π_2, where $eloop(\Pi_2) = \{\{a, c\}, \{b, c\}\} \subset loop(\Pi_2)$.

bids $\{a, b, c, \bar{d}, \bar{e}\}$. Observe that each loop formula prohibits a different supported model and can, thus, not be omitted (although loop $\{a, b, c\}$ contains the two other ones).

3 Elementary Loops

Our main focus lies in characterizing a set of relevant loops, whose loop formulas are *sufficient* and *necessary* for capturing a program's answer sets (together with the completed program). Sufficiency simply means that each model is an answer set. The meaning of necessity is not that straightforward and needs some clarification (see below).

Based on these preliminaries, we introduce the notion of an *elementary loop*.

Definition 1 (Elementary Loop). *Let Π be a logic program and let $L \in loop(\Pi)$.*

We define L as an elementary loop in Π, if, for each loop $L' \in loop(\Pi)$ such that $L' \subset L$,[3] we have $R^-(\Pi, L') \cap R^+(\Pi, L) \neq \emptyset$.

In words, a loop is elementary if each of its strict sub-loops possesses a non-circular support that positively depends on the loop. This characterization is inspired by the structure of loop formulas in (1), according to which non-circular supports form loop formulas' antecedents. If a sub-loop has no non-circular support from the genuine loop, its loop formula's antecedent is satisfied independently. Notably, Section 4 gives a direct characterization of elementary loops that avoids the inspection of sub-loops. As with loops, we denote the set of all elementary loops in a program Π by $eloop(\Pi)$. The set of all elementary loop formulas of Π is denoted by $eLF(\Pi) = \{LF(\Pi, L) \mid L \in eloop(\Pi)\}$. Obviously, we have $eloop(\Pi) \subseteq loop(\Pi)$ and $eLF(\Pi) \subseteq LF(\Pi)$.

Program Π_1 in Figure 1(a) yields the $\subseteq$-minimal loops $\{a, c\}$ and $\{b, c\}$. Such loops are by definition elementary. Moreover, $\{a, b, c\}$ is an elementary loop: Its strict sub-loops, $\{a, c\}$ and $\{b, c\}$, yield $R^-(\Pi_1, \{a, c\}) \cap R^+(\Pi_1, \{a, b, c\}) = \{c \leftarrow b, not\ e\}$ and $R^-(\Pi_1, \{b, c\}) \cap R^+(\Pi_1, \{a, b, c\}) = \{c \leftarrow a, not\ d\}$. The difference between elementary and non-elementary loops shows up when looking at Program Π_2 in Figure 2(a). Similar to Π_1, Program Π_2 has four answer sets: $\{a, b, c, d, e\}$, $\{b, \bar{d}, e\}$, $\{a, d, \bar{e}\}$, and $\{\bar{d}, \bar{e}\}$. Also, both programs share the same positive atom dependency

[3] We use '$\subset$' to denote the strict subset relation; that is, $L' \subset L$ iff $L' \subseteq L$ and $L' \neq L$.

graph, as witnessed by Figures 1(b) and 2(b), respectively. Hence, given that Programs Π_1 and Π_2 are indistinguishable from their positive atom dependency graphs, both programs yield the same set of loops, namely, $loop(\Pi_2) = loop(\Pi_1) = \{\{a, c\}, \{b, c\}, \{a, b, c\}\}$. Unlike this, both programs yield a different set of elementary loops. To see this, observe that for loop $\{a, b, c\}$ and its sub-loops $\{a, c\}$ and $\{b, c\}$, we have

$$R^-(\Pi_2, \{a, c\}) \cap R^+(\Pi_2, \{a, b, c\}) = \{a \leftarrow not\ \bar{d}\} \cap R^+(\Pi_2, \{a, b, c\}) = \emptyset\ ,$$
$$R^-(\Pi_2, \{b, c\}) \cap R^+(\Pi_2, \{a, b, c\}) = \{b \leftarrow not\ \bar{e}\} \cap R^+(\Pi_2, \{a, b, c\}) = \emptyset\ .$$

Thus, $\{a, b, c\}$ is not an elementary loop in Π_2, and $eloop(\Pi_2) = \{\{a, c\}, \{b, c\}\}$ is a strict subset of $loop(\Pi_2)$.

As mentioned above, we are interested in a minimal set of essential loops such that their loop formulas in addition to a program's completion capture the program's answer sets. Our next result is a step towards characterizing a sufficient set of loops.

Proposition 2. *Let Π be a logic program and let $L \in loop(\Pi)$ such that $L \notin eloop(\Pi)$. Let I be an interpretation of $atom(\Pi)$ such that $L \subseteq I$ and $I \models \neg LF(\Pi, L)$.*

Then, there is a loop $L' \in loop(\Pi)$ such that $L' \subset L$ and $I \models \neg LF(\Pi, L')$.

This shows that non-elementary loops are prone to redundancy.

Our first major result is an enhancement of [4, Theorem 1]. That is, elementary loop formulas are, in addition to a program's completion, sufficient for capturing the program's answer sets.

Theorem 3. *Let Π be a logic program and let $X \subseteq atom(\Pi)$.*

Then, X is an answer set of Π iff X is a model of $comp(\Pi) \cup eLF(\Pi)$.

Let us illustrate the two last results by Π_2 in Figure 2(a). Recall that we have

$$eloop(\Pi_2) = \{\{a, c\}, \{b, c\}\} \quad \subset \quad \{\{a, c\}, \{b, c\}, \{a, b, c\}\} = loop(\Pi_2)\ .$$

For Program Π_2, the set $loop(\Pi_2)$ of general loops induces the loop formulas

$$\begin{array}{lcll} LF(\Pi_2, \{a, c\}) & = & \neg(\neg\bar{d}) \rightarrow \neg a \wedge \neg c & \equiv\ \bar{d} \rightarrow \neg a \wedge \neg c\ , \\ LF(\Pi_2, \{b, c\}) & = & \neg(\neg\bar{e}) \rightarrow \neg b \wedge \neg c & \equiv\ \bar{e} \rightarrow \neg b \wedge \neg c\ , \\ LF(\Pi_2, \{a, b, c\}) & = & \neg(\neg\bar{d} \vee \neg\bar{e}) \rightarrow \neg a \wedge \neg b \wedge \neg c & \equiv\ \bar{d} \wedge \bar{e} \rightarrow \neg a \wedge \neg b \wedge \neg c\ . \end{array}$$

Observe that

$$LF(\Pi_2, \{a, c\}), LF(\Pi_2, \{b, c\}) \models LF(\Pi_2, \{a, b, c\})\ .$$

That is, loop formula $LF(\Pi_2, \{a, b, c\})$ is redundant and can be removed from $LF(\Pi_2)$ without any risk of producing models of $comp(\Pi_2) \cup (LF(\Pi_2) \setminus \{LF(\Pi_2, \{a, b, c\})\})$ that are no answer sets of Π_2. This outcome is directly obtained when considering elementary loop formulas because $eLF(\Pi_2) = LF(\Pi_2) \setminus \{LF(\Pi_2, \{a, b, c\})\}$.

In what follows, we consider the "necessity" of elementary loops. The problem here is that whether or not a loop formula eliminates unwanted supported models is context dependent because of possible interactions with the completed program and/or among loop formulas. To see this, consider Program $\Pi = \{a \leftarrow ; b \leftarrow a ; b \leftarrow c ; c \leftarrow b\}$.

We have $eloop(\Pi) = \{\{b, c\}\}$, but loop formula $LF(\Pi, \{b, c\}) = \neg a \rightarrow \neg b \wedge \neg c$ is not violated in the single supported model $\{a, b, c\}$ of Π because atom b is supported anyhow by rule $b \leftarrow a$. Furthermore, consider Program $\Pi' = \{a \leftarrow not\ b\,; b \leftarrow not\ a\,; c \leftarrow d, not\ a, not\ b\,; d \leftarrow c, not\ a, not\ b\}$ having elementary loop $\{c, d\}$. The supported models of Π' are $\{a\}$ and $\{b\}$ such that $LF(\Pi', \{c, d\}) = \top \rightarrow \neg c \wedge \neg d$ is not needed for inhibiting circular support among atoms c and d.

In order to capture elementary loops that really produce unwanted supported models, we introduce the notion of an *active elementary loop*.

Definition 4 (Active Elementary Loop). *Let Π be a logic program and let I be an interpretation of $atom(\Pi)$.*

We define $L \in eloop(\Pi)$ as an active elementary loop with respect to I, if

1. *for each rule $r \in R^-(\Pi, L)$, we have $I \models \neg comp(r)$, and*
2. *L is an elementary loop in $\Pi \setminus \{r \in \Pi \mid I \models \neg comp(r)\}$.*

By Condition *1.* an active elementary loop is not non-circularly supported. Condition *2.* ensures that an active elementary loop is still elementary with respect to the rules satisfied by an interpretation; i.e. the rules connecting the elementary loop are not falsified.

The distinguishing property of elementary loops that are active with respect to an interpretation I, as opposed to general loops, is that I "automatically" satisfies the loop formula of any of their sub-loops.

Theorem 5. *Let Π be a logic program, let $L \in eloop(\Pi)$, and let I be an interpretation of $atom(\Pi)$ such that L is active with respect to I.*

Then, we have $I \models \neg LF(\Pi, L)$, and, for each loop $L' \in loop(\Pi)$ such that $L' \subset L$, we have $I \models LF(\Pi, L')$.

For illustration, reconsider Programs Π_1 and Π_2 (cf. Figures 1(a) and 2(a)). Both programs yield the loops $\{a, c\}$, $\{b, c\}$, and $\{a, b, c\}$. The difference between Π_1 and Π_2 is that $\{a, b, c\}$ is an elementary loop in Π_1, but not in Π_2. For Π_1, this means that, if $\{a, b, c\}$ is active with respect to a supported model M of $comp(\Pi_1)$, M is also model of $comp(\Pi_1) \cup eLF(\Pi_1) \setminus \{LF(\Pi_1, \{a, b, c\})\}$. In fact, the supported model $M = \{a, b, c, \bar{d}, \bar{e}\}$ violates

$$LF(\Pi_1, \{a, b, c\}) \;=\; \neg(\neg\bar{d} \vee \neg\bar{e}) \rightarrow \neg a \wedge \neg b \wedge \neg c \;\equiv\; \bar{d} \wedge \bar{e} \rightarrow \neg a \wedge \neg b \wedge \neg c$$

but satisfies

$$\begin{aligned} LF(\Pi_1, \{a, c\}) \;&=\; \neg(\neg\bar{d} \vee (b \wedge \neg e)) \rightarrow \neg a \wedge \neg c \;\equiv\; \bar{d} \wedge (\neg b \vee e) \rightarrow \neg a \wedge \neg c\,, \\ LF(\Pi_1, \{b, c\}) \;&=\; \neg(\neg\bar{e} \vee (a \wedge \neg d)) \rightarrow \neg b \wedge \neg c \;\equiv\; \bar{e} \wedge (\neg a \vee d) \rightarrow \neg b \wedge \neg c\,. \end{aligned}$$

Hence, we cannot skip $LF(\Pi_1, \{a, b, c\})$ without producing a supported model that is no answer set. In contrast to this, no model of $comp(\Pi_2)$ violates $LF(\Pi_2, \{a, b, c\})$ and satisfies both $LF(\Pi_2, \{a, c\})$ and $LF(\Pi_2, \{b, c\})$. This follows directly from Theorem 3, as $\{a, b, c\}$ is not an elementary loop in Π_2.

4 Graph-Theoretical Characterization of Elementary Loops

We have seen in the previous section that elementary and non-elementary loops cannot be distinguished using atom dependency graphs (cf. Figures 1(b) and 2(b)). Furthermore, Definition 1 suggests examining all strict sub-loops for finding out whether a loop is elementary. This is intractable as a loop may have exponentially many strict sub-loops. In what follows, we show that identifying elementary loops can be done efficiently based on a refined concept of a dependency graph.

First of all, we introduce the *body-head dependency graph* of a program.

Definition 6 (Positive Body-Head Dependency Graph). *Let Π be a logic program.*

We define the positive body-head dependency graph of Π as the directed graph $(atom(\Pi) \cup body(\Pi), E_0(\Pi) \cup E_2(\Pi))$ where

$$E_0(\Pi) = \{(b, B) \mid r \in \Pi,\, b \in body^+(r),\, body(r) = B\}\,,$$
$$E_2(\Pi) = \{(B, a) \mid r \in \Pi,\, body(r) = B,\, head(r) = a\}\,.$$

Body-head dependency graphs were introduced in [9] as a formal device for characterizing answer set computation. In fact, fully-fledged body-head dependency graphs constitute the primary data structure of the `nomore` answer set solver [12]. In addition to the edges in $E_0(\Pi)$ and $E_2(\Pi)$, they contain a type of edges for negative dependencies, namely, $E_1(\Pi) = \{(c, B) \mid r \in \Pi, c \in body^-(r), body(r) = B\}$.[4] In what follows, we often drop the attribute 'positive' and simply write body-head or atom dependency graph, respectively, since loops exclusively rely on positive dependencies.

Definition 7 (Induced Subgraph). *Let Π be a logic program and let $A \subseteq atom(\Pi)$.*

We define the induced subgraph of A in Π as the directed graph $(A \cup body(\Pi, A), E_0(\Pi, A) \cup E_2(\Pi, A))$ where

$$body(\Pi, A) = \{B \in body(\Pi) \mid b \in A,\, (b, B) \in E_0(\Pi),\, a \in A,\, (B, a) \in E_2(\Pi)\}\,,$$
$$E_0(\Pi, A) = E_0(\Pi) \cap (A \times body(\Pi, A))\,,$$
$$E_2(\Pi, A) = E_2(\Pi) \cap (body(\Pi, A) \times A)\,.$$

Note that, in the induced subgraph of a set A of atoms, we only include those bodies that contain an atom in A and that also occur in a rule whose head is in A. That is, the bodies, which are responsible for edges in atom dependency graphs, are made explicit in body-head dependency graphs as nodes in-between atoms.

Figure 1(c) shows the body-head dependency graph of Π_1. As in Figure 1(b), we leave out isolated nodes, that is here, purely negative bodies and atoms not occurring in positive bodies. Unlike this, atom a is contained in the graph since it occurs in the positive body of rule $c \leftarrow a, not\ d$; accordingly, the edge $(a, \{a, not\ d\})$ belongs to the set of edges $E_0(\Pi_1)$ of the body-head dependency graph. Among the edges in $E_2(\Pi_1)$, we find $(\{a, not\ d\}, c)$ because of rule $c \leftarrow a, not\ d$. The induced subgraph of $\{a, c\}$ in Π_1 contains atoms a and c, bodies $\{a, not\ d\}$ and $\{c\}$, and their connecting edges.

As with atom dependency graphs, a set of atoms is a loop if its induced subgraph is a non-trivial strongly connected graph.

[4] The notation traces back to [13]; the sum of labels in a cycle indicates whether the cycle is even or odd.

Proposition 8. *Let Π be a logic program and let $L \subseteq atom(\Pi)$.*

L is a loop in Π iff the induced subgraph of L in Π is a strongly connected graph such that $body(\Pi, L) \neq \emptyset$.

In order to describe elementary loops graph-theoretically, we introduce the *elementary subgraph* of a set of atoms, which is more fine grained than the induced subgraph.

Definition 9 (Elementary Subgraph). *Let Π be a logic program, let $A \subseteq atom(\Pi)$, and let $(A \cup body(\Pi, A), E_0(\Pi, A) \cup E_2(\Pi, A))$ be the induced subgraph of A in Π.*

We define the elementary closure of A in Π as the set $eCl(\Pi, A)$ of edges where

$$
\begin{aligned}
eCl^0(\Pi, A) &= \emptyset\ , \\
eCl^{i+1}(\Pi, A) &= eCl^i(\Pi, A) \cup \{(b, B) \in E_0(\Pi, A) \mid \textit{ there is} \\
&\qquad \textit{a path in } (A \cup body(\Pi, A)\ , eCl^i(\Pi, A) \cup E_2(\Pi, A)) \\
&\qquad \textit{from } b \in A \textit{ to each } b' \in A \textit{ such that } (b', B) \in E_0(\Pi, A)\ \}\ ,^{5} \\
eCl(\Pi, A) &= \textstyle\bigcup_{i \in I\!N} eCl^i(\Pi, A)\ .
\end{aligned}
$$

We define the elementary subgraph of A in Π as the directed graph $(A \cup body(\Pi, A), eCl(\Pi, A) \cup E_2(\Pi, A))$.

The general purpose of elementary subgraphs is to distinguish essential from superfluous dependencies. Let us illustrate this by rule $c \leftarrow a, b$ in Π_2 and consider the body-head dependency graph of Π_2 in Figure 2(c). Here, atom c positively depends on atoms a and b through body $\{a, b\}$. In Π_2, c is unfounded if either a or b and c itself are not non-circularly supported; that is, the other atom cannot help in non-circularly supporting a and c or b and c, respectively. The situation changes if a and b take part in a loop independently from c. Then, a and b non-circularly support c if there is a non-circular support for either a or b. The elementary closure reflects these issues by stipulating that there is already a path from one to the other predecessors of a body before an edge to the body can be added. This allows for distinguishing essential dependencies from superfluous ones.

Our next major result shows that elementary subgraphs make the difference between elementary loops and non-elementary ones.

Theorem 10. *Let Π be a logic program and let $L \subseteq atom(\Pi)$.*

L is an elementary loop in Π iff the elementary subgraph of L in Π is a strongly connected graph such that $body(\Pi, L) \neq \emptyset$.

For illustrating the previous result, reconsider Figure 1(c) showing the connected part of the body-head dependency graph of Program Π_1. Observe that each contained body is reached by precisely one edge. Therefore, we have $eCl^1(\Pi_1, A) = E_0(\Pi_1, A)$ for every $A \subseteq atom(\Pi_1)$, and elementary subgraphs coincide with induced subgraphs.

The body-head dependency graph of Program Π_2 is different from the one of Program Π_1, as witnessed by Figures 1(c) and 2(c). In Figure 2(c), we see the connected

[5] Note that the path from b to b' can be trivial, i.e. $b = b'$.

part of the body-head dependency graph of Π_2, which coincides with the induced subgraph of loop $\{a, b, c\}$ in Π_2. Regarding the elementary closure of $\{a, b, c\}$, we have

$$eCl(\Pi_2, \{a, b, c\}) = eCl^1(\Pi_2, \{a, b, c\}) = \{(c, \{c, not\ d\}), (c, \{c, not\ e\})\}\ .$$

Observe that $eCl(\Pi_2, \{a, b, c\})$ does not contain edges $(a, \{a, b\})$ and $(b, \{a, b\})$. This is because a as well as b must have a path to the other atom before the respective edge can be added to the elementary closure. Since there are no such paths, none of the edges can ever be added. As a consequence, atoms a and b have no outgoing edges in the elementary subgraph of $\{a, b, c\}$ in Π_2, which is not strongly connected. This agrees with the observation made in Section 3 that $\{a, b, c\}$ is not an elementary loop in Π_2. In contrast to $\{a, b, c\}$, the elementary subgraphs of loops $\{a, c\}$ and $\{b, c\}$ in Π_2 are strongly connected, verifying $eloop(\Pi_2) = \{\{a, c\}, \{b, c\}\}$.

As observed on Program Π_1, elementary subgraphs coincide with induced subgraphs on unary programs, having at most one positive body atom. For such programs, every general loop is also an elementary one.

Proposition 11. *Let Π be a logic program such that $|body^+(r)| \leq 1$ for all $r \in \Pi$. Then, we have $eloop(\Pi) = loop(\Pi)$.*

Note that unary programs are strictly less expressive than general ones, as shown in [14].

The analysis of elementary subgraphs yields that each contained atom must be the unique predecessor of some body; otherwise, the atom has no outgoing edge in the elementary closure. Moreover, the induced subgraph of an elementary loop cannot be torn apart by removing edges to bodies, provided that each body is still reachable.

Proposition 12. *Let Π be a logic program and let $L \in eloop(\Pi)$.*

Then, the induced subgraph of L in Π, $(L \cup body(\Pi, L), E_0(\Pi, L) \cup E_2(\Pi, L))$, has the following properties:

1. *For each atom $b \in L$, there is a body $B \in body(\Pi, L)$ such that $\{b\} = \{b' \in L \mid (b', B) \in E_0(\Pi, L)\}$.*
2. *For every set $E_0^{\subseteq} \subseteq E_0(\Pi, L)$ of edges such that $\{B \in body(\Pi, L) \mid b \in L, (b, B) \in E_0^{\subseteq}\} = body(\Pi, L)$, we have that $(L \cup body(\Pi, L), E_0^{\subseteq} \cup E_2(\Pi, L))$ is a strongly connected graph.*

Although we refrain from giving a specific algorithm, let us note that the concept of elementary subgraphs allows for computing elementary loops efficiently by means of standard graph algorithms. In particular, deciding whether a set of atoms is an elementary loop can be done in linear time.

5 Elementary versus Non-Elementary Loops

This section compares the sets of a program's elementary and general loops. By Theorem 3, loop formulas for non-elementary loops need not be added to a program's completion in order to capture the program's answer sets. With this information at hand, we are interested in how many loop formulas can be omitted in the best or in the worst case, respectively.

First, we determine a lower bound on the set of a program's elementary loops. Such a bound is immediately obtained from Definition 1, because a loop is trivially elementary if it has no strict sub-loops. Thus, we have $mloop(\Pi) \subseteq eloop(\Pi)$ where $mloop(\Pi)$ denotes the set of $\subseteq$-minimal loops in a program Π. Second, the set of a program's loops constitutes an upper bound for the program's elementary loops, also by Definition 1. Thus, we have $eloop(\Pi) \subseteq loop(\Pi)$. Finally, the question is how the set of a program's loops can be bound from above. In order to answer it, we define the set of loops that are $\subseteq$-minimal for an atom $a \in atom(\Pi)$ as $aloop(\Pi, a) = \{L \in loop(\Pi) \mid a \in L$, *there is no loop* $L' \in loop(\Pi)$ *such that* $a \in L'$ *and* $L' \subset L\}$. For a program Π, we let $aloop(\Pi) = \bigcup_{a \in atom(\Pi)} aloop(\Pi, a)$.

In the worst case, any non-empty combination of loops in $aloop(\Pi)$ is a loop, and we obtain the following upper bound for a program's loops.

Proposition 13. *Let Π be a logic program.*
Then, we have $loop(\Pi) \subseteq \{\bigcup_{L \in A} L \mid A \in 2^{aloop(\Pi)} \setminus \{\emptyset\}\}$.

Taking the above considerations together, we obtain the following estimation.

Corollary 14. *Let Π be a logic program. Then, we have*

$$mloop(\Pi) \subseteq eloop(\Pi) \subseteq loop(\Pi) \subseteq \{\bigcup_{L \in A} L \mid A \in 2^{aloop(\Pi)} \setminus \{\emptyset\}\}\ .$$

In what follows, we give some schematic examples with programs Π for which $loop(\Pi) = \{\bigcup_{L \in A} L \mid A \in 2^{aloop(\Pi)} \setminus \{\emptyset\}\}$. Our first program sketches the worst case, i.e. $eloop(\Pi) = \{\bigcup_{L \in A} L \mid A \in 2^{aloop(\Pi)} \setminus \{\emptyset\}\}$, whereas the second program reflects the best case that $eloop(\Pi) = mloop(\Pi)$. The programs show that the set of elementary loops can vary significantly between the given lower and the upper bound.

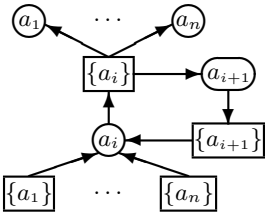
For illustrating the worst case, consider Program Π_3 in Figure 3(a). First observe that $|body^+(r)| = 1$ for every rule $r \in \Pi_3$. Thus by Proposition 10, each loop in Π_3 is elementary. The atom dependency graph of Π_3 is a complete graph because there is a rule $a_i \leftarrow a_j$ for every pair of distinct atoms $a_i \in atom(\Pi_3), a_j \in atom(\Pi_3)$. As a consequence, any combination of distinct elementary loops gives a new elementary loop, and we have $eloop(\Pi_3) = \{\bigcup_{L \in A} L \mid A \in 2^{aloop(\Pi_3)} \setminus \{\emptyset\}\}$.

Program Π_4 in Figure 3(c) is complementary to Π_3. Here $|body^+(r)| = |atom(\Pi_4)| - 1$ for every rule $r \in \Pi_4$. However, the atom dependency graph of Π_4 is identical to that of Π_3. As observed with Π_1 and Π_2 (cf. Figures 1(b) and 2(b)), Π_3 and Π_4 are thus indistinguishable from their atom dependency graphs. Again the body-head dependency graphs reveal the different natures of Π_3 and Π_4. We have that, similar to Π_3, every two-elementary subset of $atom(\Pi_4)$ forms a $\subseteq$-minimal and, thus, elementary loop. However, looking at the body-head dependency graph of Π_4 in Figure 3(d), we see that each atom has a single body as predecessor (i.e. there is a single supporting rule) such that distinct elementary loops can only be "glued" at bodies. In the resulting induced subgraph, bodies have several predecessors. Such an induced subgraph does not satisfy property *1.* from Proposition 11, and the obtained loop is non-elementary. Thus, we have $eloop(\Pi_4) = mloop(\Pi_4)$ and can omit loop formulas for all loops in $\{\bigcup_{L \in A} L \mid A \in 2^{aloop(\Pi_4)} \setminus \{\emptyset\}\} \setminus mloop(\Pi_4)$.

The achievements obtainable through using elementary instead of general loops can be underpinned by looking at the approaches of `assat` [4] and `smodels` [1] for

$$\Pi_3 = \left\{ \begin{array}{llll} a_1 \leftarrow a_2 & \ldots & a_1 \leftarrow a_n \\ \vdots & & \\ a_i \leftarrow a_1 & \ldots & a_i \leftarrow a_n \\ a_{i+1} \leftarrow a_1 & \ldots & a_{i+1} \leftarrow a_n \\ \vdots & & \\ a_n \leftarrow a_1 & \ldots & a_n \leftarrow a_{n-1} \end{array} \right\}$$

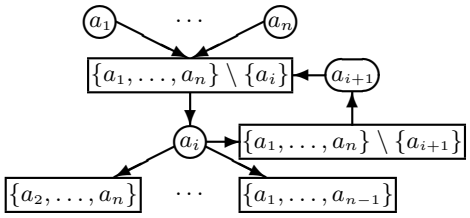
(a) Schematic program Π_3.



(b) Schematic body-head dependency graph of Π_3.

$$\Pi_4 = \left\{ \begin{array}{l} a_1 \leftarrow a_2, \ldots, a_n \\ \vdots \\ a_i \leftarrow a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n \\ a_{i+1} \leftarrow a_1, \ldots, a_i, a_{i+2}, \ldots, a_n \\ \vdots \\ a_n \leftarrow a_1, \ldots, a_{n-1} \end{array} \right\}$$

(c) Schematic program Π_4.



(d) Schematic body-head dependency graph of Π_4.

Fig. 3. Logic programs Π_3 and Π_4, where $loop(\Pi_i) = \{\bigcup_{L \in A} L \mid A \in 2^{aloop(\Pi_i)} \setminus \{\emptyset\}\}$, $eloop(\Pi_3) = \{\bigcup_{L \in A} L \mid A \in 2^{aloop(\Pi_3)} \setminus \{\emptyset\}\}$, and $eloop(\Pi_4) = mloop(\Pi_4)$.

dealing with unfounded sets. The `assat` system is based on a program's completion and identifies loops, whose loop formulas are violated, on demand, that is, whenever a supported model not representing an answer set has been found. The circular support of such loops is in future prohibited by loop formulas such that an unwanted supported model cannot be recomputed. Now assume that the supported model $\{a, b, c, \bar{d}, \bar{e}\}$ is found first for Program Π_2 in Figure 2(a). Then `assat` identifies $\{a, b, c\}$ as a so-called *terminating loop* [4] and adds loop formula

$$LF(\Pi_2, \{a, b, c\}) \;=\; \neg(\neg\bar{d} \vee \neg\bar{e}) \rightarrow \neg a \wedge \neg b \wedge \neg c \;\equiv\; \bar{d} \wedge \bar{e} \rightarrow \neg a \wedge \neg b \wedge \neg c$$

to $comp(\Pi_2)$ before searching for another supported model. The problem is that loop $\{a, b, c\}$ is non-elementary and that circular support within elementary loops $\{a, c\}$ and $\{b, c\}$ is not prohibited by $LF(\Pi_2, \{a, b, c\})$. Consequently, `assat` may find supported models $\{a, b, c, \bar{d}, e\}$ and $\{a, b, c, d, \bar{e}\}$ next, necessitating additional loop formulas

$$\begin{array}{rcl} LF(\Pi_2, \{a, c\}) & = & \neg(\neg\bar{d}) \rightarrow \neg a \wedge \neg c \;\equiv\; \bar{d} \rightarrow \neg a \wedge \neg c \quad \text{and} \\ LF(\Pi_2, \{b, c\}) & = & \neg(\neg\bar{e}) \rightarrow \neg b \wedge \neg c \;\equiv\; \bar{e} \rightarrow \neg b \wedge \neg c\,, \end{array}$$

before finding the first answer set. The possibility of computing the supported models $\{a, b, c, \bar{d}, e\}$ and $\{a, b, c, d, \bar{e}\}$ can be avoided by splitting the non-elementary loop $\{a, b, c\}$ into its elementary sub-loops $\{a, c\}$ and $\{b, c\}$. Besides $\{a, c\}$ and $\{b, c\}$, $LF(\Pi_2, \{a, c\})$ and $LF(\Pi_2, \{b, c\})$ prohibit circular support within loop $\{a, b, c\}$, and

`assat` may treat three loops using only two loop formulas. In general, the elementary closure, as given in Definition 8, can be used for checking whether a terminating loop is elementary. If not, an elementary sub-loop, whose loop formula also prohibits circular support within the genuine terminating loop, can be determined.[6]

The `smodels` answer set solver falsifies greatest unfounded sets in its function *atmost*. At the implementation level, *atmost* is restricted to strongly connected components of a program's positive atom dependency graph (but may spread over different components if an unfounded set is detected) [1]. When *atmost* is applied to Program Π_4 in Figure 3(c) (or a program having a comparable body-head dependency graph), it has to take the whole strongly connected component induced by $atom(\Pi_4)$ into consideration, since the atom dependency graph of Π_4 is complete. The efforts of *atmost* can be restricted by concentrating on elementary loops, which are pairs of atoms in case of Π_4. That is, any pair of unfounded atoms is sufficient for falsifying the bodies of all rules that contribute to the strongly connected component induced by $atom(\Pi_4)$.

Finally, it is noteworthy to mention that [15] describes how the computation of a program's well-founded model [16] simplifies based on certain properties of the program's full atom dependency graph (i.e. both positive and negative edges are included). The simplifications can be applied if a strongly connected component contains either only positive edges or if no atom depends positively on itself. The first case reflects that the contained atoms are involved in a loop and the second that circular support is impossible. An interesting topic for future investigation is whether the above conditions can be refined using the richer structure of body-head dependency graphs, which, for instance, allows for distinguishing between elementary and non-elementary loops.

6 Conclusion

The purpose of loop formulas is to falsify unfounded sets whose atoms circularly depend upon each other in a given program. The detection of unfounded sets traces back to well-founded semantics [16]. Basically, the well-founded semantics infers atoms that are consequences of a program's rules and falsifies unfounded sets. In accord with the well-founded semantics, all atoms in an answer set are consequences and no atom is unfounded. Complementary to [16] concentrating on *greatest* unfounded sets, this paper investigates indispensable unfounded sets whose falsification is essential for answer set computation. To this end, we have introduced the notion of an elementary loop and have described it using body-head dependency graphs. Although we cannot avoid the theoretical barrier of exponentially many loops in the worst case, we have shown that elementary loops provide necessary and sufficient criteria for characterizing answer sets. Apart from their theoretical importance, our results have furthermore a practical impact since they allow to focus the computation in ASP solvers to ultimately necessary parts. An interesting topic for future research will be generalizing our new concept of elementary loops to disjunctive programs, as has been done for general loops in [17].

[6] A non-elementary loop may yield exponentially many elementary sub-loops. Thus, identifying *all* elementary sub-loops might be intractable. However, the number of elementary loops needed to cover the atoms in a (general) terminating loop of size n is bound by n.

Acknowledgments We are grateful to Enno Schultz and the anonymous referees for their helpful comments, leading to a new presentation of our contribution. This work was supported by DFG under grant SCHA 550/6-4 as well as the EC through IST-2001-37004 WASP project.

References

1. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence **138** (2002) 181–234
2. Leone, N., Faber, W., Pfeifer, G., Eiter, T., Gottlob, G., Koch, C., Mateis, C., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Transactions on Computational Logic (2005) To appear.
3. Babovich, Y., Lifschitz, V.: Computing answer sets using program completion. Unpublished draft. (2003)
4. Lin, F., Zhao, Y.: Assat: computing answer sets of a logic program by sat solvers. Artificial Intelligence **157** (2004) 115–137
5. Lierler, Y., Maratea, M.: Cmodels-2: Sat-based answer sets solver enhanced to non-tight programs. In Lifschitz, V., Niemelä, I., eds.: Proceedings of the Seventh International Conference on Logic Programming and Nonmonotonic Reasoning. Springer (2004) 346–350
6. Clark, K.: Negation as failure. In Gallaire, H., Minker, J., eds.: Logic and Data Bases. Plenum Press (1978) 293–322
7. Fages, F.: Consistency of clark's completion and the existence of stable models. Journal of Methods of Logic in Computer Science **1** (1994) 51–60
8. Lifschitz, V., Razborov, A.: Why are there so many loop formulas? ACM Transactions on Computational Logic (To appear.)
9. Linke, T.: Suitable graphs for answer set programming. In Vos, M.D., Provetti, A., eds.: Proceedings of the Second International Workshop on Answer Set Programming. CEUR Workshop Proceedings (2003) 15–28
10. Apt, K., Blair, H., Walker, A.: Towards a theory of declarative knowledge. In Minker, J., ed.: Foundations of Deductive Databases and Logic Programming. Morgan Kaufmann Publishers (1987) 89–148
11. Przymusinski, T.: On the declarative semantics of deductive databases and logic programs. In Minker, J., ed.: Foundations of Deductive Databases and Logic Programming. Morgan Kaufmann Publishers (1988) 193–216
12. (`http://www.cs.uni-potsdam.de/~linke/nomore`)
13. Papadimitriou, C., Sideri, M.: Default theories that always have extensions. Artificial Intelligence **69** (1994) 347–357
14. Janhunen, T.: Comparing the expressive powers of some syntactically restricted classes of logic programs. In Lloyd, J., Dahl, V., Furbach, U., Kerber, M., Lau, K., Palamidessi, C., Pereira, L., Sagiv, Y., Stuckey, P., eds.: Proceedings of the First International Conference on Computational Logic. Springer (2000) 852–866
15. Dix, J., Furbach, U., Niemelä, I.: Nonmonotonic reasoning: Towards efficient calculi and implementations. In Robinson, J., Voronkov, A., eds.: Handbook of Automated Reasoning. Elsevier and MIT Press (2001) 1241–1354
16. van Gelder, A., Ross, K., Schlipf, J.: The well-founded semantics for general logic programs. Journal of the ACM **38** (1991) 620–650
17. Lee, J., Lifschitz, V.: Loop formulas for disjunctive logic programs. In Palamidessi, C., ed.: Proceedings of 19th International Conference on Logic Programming. Springer (2003) 451–465