Computing Diverse Optimal Stable Models*

Javier Romero[1], Torsten Schaub[1,2], and Philipp Wanko[1]

1 **University of Potsdam, Germany**
2 **INRIA, Rennes**

Abstract

We introduce a comprehensive framework for computing diverse (or similar) solutions to logic programs with preferences. Our framework provides a wide spectrum of complete and incomplete methods for solving this task. Apart from proposing several new methods, it also accommodates existing ones and generalizes them to programs with preferences. Interestingly, this is accomplished by integrating and automating several basic ASP techniques — being of general interest even beyond diversification. The enabling factor of this lies in the recent advance of multi-shot ASP solving that provides us with fine-grained control over reasoning processes and abolishes the need for solver modifications and wrappers that were indispensable in previous approaches. Our framework is implemented as an extension to the ASP-based preference handling system *asprin*. We use the resulting system *asprin* 2 for an exhausting empirical evaluation of the whole spectrum of diversification methods comprised in our framework.

1998 ACM Subject Classification D.1.6 Logic Programming, F.4.1 Mathematical Logic, I.2.3 Deduction and Theorem Proving

Keywords and phrases Answer Set Programming, Diversity, Similarity, Preferences, Maxmin optimization

Digital Object Identifier 10.4230/OASIcs.ICLP.2016.first-page-number

1 Introduction

Answer Set Programming (ASP; [4]) has become a prime paradigm for solving combinatorial problems in Knowledge Representation and Reasoning. As a matter of fact, such problems have an exponential number of solutions in the worst-case. A first means to counterbalance this is to impose preference relations among solutions to filter out optimal ones. Often enough, this still leaves us with a large number of optimal models. A typical example is the computation of Pareto frontiers for multi-objective optimization problems [19], as we encounter in design space exploration [1] or timetabling [3]. Other examples include configuration, planning, and phylogeny, as discussed in [9]. This calls for computational support that allows for identifying small subsets of diverse solutions. The computation of diverse stable models was first considered in ASP by [9]. The analogous problem regarding optimal stable models is addressed in [25] in the case of answer set optimization [7]. Beyond ASP, the computation of diverse solutions is also studied in CP [17] and SAT [18].

In this paper, we introduce a comprehensive framework for computing diverse (or similar) solutions to logic programs with preferences. One of its distinguishing factors is that it allows for dealing with aggregated (or plain) qualitative and quantitative preferences among stable models of logic programs. This is accomplished by building on the preference handling capacities of *asprin* [5]. The other appealing factor of our framework is that it covers a wide spectrum of methods for diversification. Apart from new techniques, it also accommodates and generalizes existing approaches by lifting them to programs with preferences. Interestingly, this is done by taking advantage of several

* This work was partially supported by DFG-SCHA-550/9

Technical Communications of the 32nd Int'l Conference on Logic Programming (ICLP'16).
Editors: Manuel Carro, Andy King, Marina De Vos, and Neda Saeedloei; pp. 1–14
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

existing basic ASP techniques that we automate and integrate in our framework. The enabling factor of this is the recent advance of multi-shot ASP solving that allows for an easy yet fine-grained control of ASP-based reasoning processes (cf. [13]). In particular, this abolishes the need for internal solver modifications or singular solver wrappers that were often unavoidable in previous approaches. We have implemented our approach as an extension to the preference handling framework *asprin*. The resulting system *asprin* 2 is then used for an empirical evaluation contrasting several alternative approaches to computing diverse solutions. Last but not least, note that although we concentrate on diversity, our approach applies just as well to the dual concept of *similarity*. This is also reflected by its implementation supporting both settings.

2 Background

In ASP, problems are described as (disjunctive) *logic programs*, being sets of *rules* of the form

$$\mathtt{a}_1\mathtt{;}\ldots\mathtt{;}\mathtt{a}_m \mathtt{ :- } \mathtt{a}_{m+1}\mathtt{,}\ldots\mathtt{,}\mathtt{a}_n\mathtt{,not }\mathtt{a}_{n+1}\mathtt{,}\ldots\mathtt{,not }\mathtt{a}_o$$

where each $\mathtt{a}_i$ is a propositional atom and `not` stands for *default negation*. We call a rule a *fact* if $m = o = 1$, *normal* if $m = 1$, and an *integrity constraint* if $m = 0$. We may reify a rule r with the set of facts $R(r) = \{\mathtt{rule(}r\mathtt{)}\} \cup \{\mathtt{head(}r\mathtt{,}\mathtt{a}_i\mathtt{)} \mid 1 \leq i \leq m\} \cup \{\mathtt{body(}r\mathtt{,pos,}\mathtt{a}_i\mathtt{)} \mid m+1 \leq i \leq n\} \cup \{\mathtt{body(}r\mathtt{,neg,}\mathtt{a}_i\mathtt{)} \mid n+1 \leq i \leq o\}$, and we reify a program by joining its reified rules. Semantically, a logic program induces a collection of *stable models*, which are distinguished models of the program determined by stable models semantics; see [16] for details.

To ease the use of ASP in practice, several extensions have been developed. First of all, rules with variables are viewed as shorthands for the set of their ground instances. Further language constructs include *conditional literals* and *cardinality constraints* [24]. The former are of the form $\mathtt{a:}\mathtt{b}_1\mathtt{,}\ldots\mathtt{,}\mathtt{b}_m$, the latter can be written as $\mathtt{s\{}\mathtt{c}_1\mathtt{,}\ldots\mathtt{,}\mathtt{c}_n\mathtt{\}t}$, where `a` and $\mathtt{b}_i$ are possibly default-negated literals and each $\mathtt{c}_j$ is a conditional literal; `s` and `t` provide lower and upper bounds on the number of satisfied literals in the cardinality constraint. We refer to $\mathtt{b}_1\mathtt{,}\ldots\mathtt{,}\mathtt{b}_m$ as a *condition*. The practical value of both constructs becomes apparent when used with variables. For instance, a conditional literal like `a(X):b(X)` in a rule's antecedent expands to the conjunction of all instances of `a(X)` for which the corresponding instance of `b(X)` holds. Similarly, `2{a(X):b(X)}4` is true whenever at least two and at most four instances of `a(X)` (subject to `b(X)`) are true. Specifically, we rely in the sequel on the input language of the ASP system *clingo* [13]; further language constructs are explained on the fly.

In what follows, we go beyond plain ASP and deal with *logic programs with preferences*. More precisely, we consider programs P over some set $\mathcal{A}$ of atoms along with a strict partial order $\succ \subseteq \mathcal{A} \times \mathcal{A}$ among their stable models. Given two stable models X, Y of P, $X \succ Y$ means that X is preferred to Y. Then, a stable model X of P is *optimal* wrt $\succ$, if there is no other stable model Y such that $Y \succ X$. In what follows, we often leave the concrete order implicit and simply refer to a program with preferences and its optimal stable models. We restrict ourselves to partial orders and distance measures (among pairs of stable models) that can be computed in polynomial time. For simplicity, we focus on the Hamming distance, defined for two stable models X, Y of a program P over $\mathcal{A}$ as $d(X,Y) = |\mathcal{A} - X - Y| + |X \cap Y|$. Given a logic program P with preferences and a positive integer n, we define a set $\mathcal{X}$ of optimal stable models of P as *most diverse*, if $\min\{\, d(X,Y) \mid X,Y \in \mathcal{X}, X \neq Y\} > \min\{\, d(X,Y) \mid X,Y \in \mathcal{X}', X \neq Y\}$ for every other set $\mathcal{X}'$ of optimal stable models of P. We are thus interested, following [9], in the problem *n Most Diverse Optimal Models*: Given a logic program P with preferences and a positive integer n, find n most diverse optimal stable models of P.

For representing logic programs with complex preferences and computing their optimal models, we built upon the preference framework of *asprin* [5], a system for dealing with aggregated qualitative

and quantitative preferences. In *asprin*, the above mentioned *preference relations* are represented by declarations of the form `#preference(p,t){t1:b1,...,tn:bn}` where `p` and `t` are the name and type of the preference relation, and $\texttt{t}_i$ and $\texttt{b}_i$ are tuples of terms and conditions, respectively,[1] serving as arguments of `p`. The directive `#optimize(p)` instructs *asprin* to search for stable models that are optimal wrt the strict partial order $\succ_{\texttt{p}}$ associated with `p`. While *asprin* already comes with a library of predefined primitive and aggregate preference types, like `subset` or `pareto`, respectively, it also allows for adding customized preferences. We illustrate how this works by implementing preference type `maxmin` in Section 4.

Finally, we investigate whether the heuristic capacities of *clingo* allow for boosting our approach. In fact, *clingo* 5 features heuristic directives of the form '`#heuristic c. [k,m]`' where c is a conditional atom, k is a term evaluating to an integer, and m is a heuristic modifier among `init`, `factor`, `level`, or `sign`. The effect of the heuristic modifiers is to bias the score of *clasp*'s heuristic by initially adding or multiplying the score, prioritizing variables, or preferably assigning a truth value, respectively. The value of k serves as argument to the respective modification. A more detailed description can be found in [15].

3 Our diversification framework at a glance

We begin with an overview over the various techniques integrated in our framework.

Basic solving techniques.e first summarize several basic solving techniques that provide essential pillars of our framework and that are also of interest for other application areas.

Maxmin optimization is a popular strategy in game theory and beyond that is not supported by existing ASP systems. We address this issue and consider *maxmin* (and *minmax*) optimization that, given a set of sums, aims at maximizing the value of the minimum sum. We have implemented both preference types and made them available via *asprin* 2's library.

Guess and Check automation. [11] defined a framework for representing and solving Σ_2^P problems in ASP. Given two normal logic programs P and Q capturing a guess-and-check (G&C) problem, X is a solution to $\langle P,Q\rangle$ if X is a stable model of P and $Q \cup X$ is unsatisfiable. We *automatize* this by using reification along with the meta-encoding methodology of *metasp* [14]. In this way, the two normal programs P and Q are transformed into a single disjunctive logic program. The resulting mini-system *metagnc* is implemented in Python and available at [2]. We build upon this approach for dealing with logic programs with preferences. To this end, *asprin* translates a logic program with preferences into a G&C problem, which is then translated by *metagnc* into a disjunctive logic program and solved by an ASP system.

Querying programs with preferences consists of deciding whether there is an optimal stable model of a program P with preferences that contains a given query atom q. To this end, we elaborate upon four alternatives:

Q-1. Enumerate *models* of $P \cup \{\bot \leftarrow \mathit{not}\ q\}$ until one is an optimal model of P.
Q-2. Enumerate *optimal models* of P until one contains q.
Q-3. Enumerate *optimal models* of $P \cup \{\bot \leftarrow \mathit{not}\ q\}$ until one is an optimal model of P.
Q-4. Enumerate *optimal models* of P until one contains q
while alternately adding $\{\bot \leftarrow \mathit{not}\ q\}$ or $\{\bot \leftarrow q\}$ during model-driven optimization.

The first two methods were implemented by [25] in the case of programs with *aso* preferences [7]. We generalize both to arbitrary preferences, propose two novel ones, and provide all four methods in

[1] See [5] for more general preference elements.

asprin 2. Applications of querying programs with preferences are clearly of greater interest and go well beyond diversification.

Preferences over optimal models allow for further narrowing down the stable models of interest by imposing a selection criterion among the optimal models of a logic program with preferences. For one thing, this is different from a lexicographic preference, since the secondary preference takes into account all optimal models wrt the first preference, no matter whether they are equal or incomparable. For another, it aims at preference combinations whose complexity goes beyond the expressiveness of ASP and thus cannot be addressed via an encoding in *asprin*. Rather we conceived a nested variant of *asprin*'s optimization algorithm that computes the preferred optimal models. Interestingly, this makes use of our querying capacities in posing the "improvement constraint" as a query.

Advanced diversification techniques.e elaborate upon three ways of diversification, viz. enumeration, replication, and approximation, for solving the *n Most Diverse Optimal Models* problem. While the two former are complete the latter is not.

Enumeration consists of two steps:

1. Enumerate all optimal models of the logic program P with preferences.
2. Find among all computed optimal models, the n most diverse ones.

While we carry out the first step by means of *asprin*'s enumeration mode, we cast the second one as an optimization problem and express it as a logic program with preferences. This method was first used by [9] for addressing diversity in the context of logic programs without preferences; we lift it here to programs with preferences.

Replication consists of three steps:

1. Translate a normal logic program P with preferences into a disjunctive logic program D by applying the aforementioned guess-and-check method.
2. Reify D into $\mathcal{R}(D)$, and add a meta-encoding M replicating D such that each stable model of $M \cup \mathcal{R}(D)$ corresponds to n optimal models of the original logic program P.
3. Turn the disjunctive logic program $M \cup \mathcal{R}(D)$ into a *maxmin* optimization problem by applying the aforementioned method such that its optimal stable models correspond to n most diverse optimal stable models of the original program P with preferences.

This method was outlined for logic programs without preferences in [9] but not automated. We generalize this approach to normal programs with preferences and provide a fully automated approach.

Approximation. Our approximation techniques can be understood as instances of the following algorithm, whose input is a logic program with preferences P:

1. Find an optimal model of P. If P is unsatisfiable then return $\{\bot\}$, else add the solution to $\mathcal{X}$.
2. While $test(\mathcal{X})$ is true, call $solve(P,\mathcal{X})$ and add the solution to $\mathcal{X}$.
3. Return $solution(\mathcal{X})$.

In the basic case, $test(\mathcal{X})$ returns *true* until there are n solutions in $\mathcal{X}$, $solution(\mathcal{X})$ returns the set $\mathcal{X}$, and the algorithm simply computes n solutions by successively calling $solve(P,\mathcal{X})$. More elaborate approaches are obtained, for example, computing $n + k$ solutions and returning the n most distant among them in $solution(\mathcal{X})$.

The implementation of $solve(P,\mathcal{X})$ leads to different approaches:

A-1. $solve(P,\mathcal{X})$ returns an optimal model of P most diverse to those in $\mathcal{X}$. We accomplish this by defining a *maxmin* preference maximizing the minimal distance to any of the solutions in $\mathcal{X}$, and impose this on top of the optimal models of P by applying the two aforementioned approaches to *maxmin* optimization and preferences over optimal models. This method was first used by [9] for addressing diversity in the context of logic programs without preferences; we lift it here to programs with preferences.

A-2. *solve*$(P,\mathcal{X})$ first computes a partial interpretation I distant to $\mathcal{X}$, and then returns an optimal model of P closest to I:

- **a.** Select a partial interpretation I in one of the following ways: (i) a random one, (ii) a heuristically chosen one, (iii) one most diverse wrt the solutions in $\mathcal{X}$, or (iv) one complementary to the last computed optimal model.
- **b.** Use a cardinality-based preference minimizing the distance to I, and apply the aforementioned approach to preferences over optimal models to enforce this preference among the optimal models of P.

A-3. *solve*$(P,\mathcal{X})$ approximates *A-2* using heuristics. To this end, we select a partial interpretation I as in *A-2*, and then guide the computation of the optimal model fixing the sign of the atoms to their value in I. The approach is further developed prioritizing the variables in I. A similar method was used by [18] for SAT.

4 Basic solving techniques

We first show how the *Most Distant (Optimal) Model* problem can be represented in *asprin* using the new preference type *maxmin*: Given a logic program P (with preferences) over $\mathcal{A}$, and a set $\mathcal{X} = \{X_1, \ldots, X_m\}$ of (optimal) stable models of P, find an (optimal) stable model of P that maximizes the minimal distance to the (optimal) stable models in $\mathcal{X}$. The *Most Distant Optimal Model* is used by our approximation algorithms in Section 5.

Maxmin optimization in *asprin*. Let $H_\mathcal{X}$ be the set of facts $\{\texttt{holds(}a\texttt{,}i\texttt{).} \mid a \in X_i, X_i \in \mathcal{X}\}$ reifying the stable models in $\mathcal{X}$, and let `distance` be the following preference statement:

```
#preference(distance,maxmin){
  I,1,X : holds(X,0), not holds(X,I), I=1..m;
  I,1,X : not holds(X,0), holds(X,I), I=1..m }.
```

Then, the *Distant Model* problem is solved by the following program with preferences: $P \cup \{\texttt{holds(}a\texttt{,0) :- }a\texttt{.} \mid a \in \mathcal{A}\} \cup H_\mathcal{X} \cup \{\texttt{distance}\} \cup \{\texttt{\#optimize(distance).}\}$. P generates stable models that are reified with `holds(`a`,0)` for $a \in \mathcal{A}$. The preference statement `distance` represents a `maxmin` preference over `I` sums, with `I=1..`m, and the value of each sum amounts to the distance between the generated stable model and $X_\texttt{I}$. Finally, the optimize statement selects the optimal stable models wrt $\succ_\texttt{distance}$.

Formally, the preference elements of preference type *maxmin* have the restricted form '`s,w,t:B`' where `s`, `w`, `t` are terms, and `B` is a condition. Term `s` names different sums, whose value is specified by the rest of the element '`w,t:B`' (similar to aggregate elements). For defining the semantics of *maxmin*, preference elements stand for their ground instantiations, and we consider a set E of such ground preference elements. We say that `s` is a sum of E if it is the first term of some preference element. Given a stable model X and a sum `s` of E, the value of `s` in X is:

$$v(\texttt{s},X) = \textstyle\sum_{(\texttt{w},\texttt{t}) \in \{\texttt{w},\texttt{t} \mid \texttt{s},\texttt{w},\texttt{t:B} \in E, X \models \texttt{B}\}} \texttt{w}$$

For a set E of ground preference elements for preference statement `p`, *maxmin* defines the following preference relation:[2]

$$X \succ_\texttt{p} Y \text{ if } \min\{v(\texttt{s},X) \mid \texttt{s} \text{ is a sum of } E\} > \min\{v(\texttt{s},Y) \mid \texttt{s} \text{ is a sum of } E\}$$

Applying this definition to the preference statement `distance` gives the partial order $\succ_\texttt{distance}$.

[2] For defining *minmax*, we simply switch min by max, and $>$ by $<$.

In *asprin*, partial orders $\succ$ are implemented by so-called *preference programs*. For our example, we say that Q is a *preference program* for $\succ_{\texttt{distance}}$ if it holds that $X \succ_{\texttt{distance}} Y$ iff $Q \cup H_X \cup H'_Y$ is satisfiable, where $H_X = \{\texttt{holds(}a\texttt{).} \mid a \in X\}$ and $H'_Y = \{\texttt{holds'(}a\texttt{).} \mid a \in Y\}$. In practice, the preference program Q consists of three parts.

First, each preference statement is translated into a set of facts, and added to Q. Our example preference statement `distance` results in `preference(distance,maxmin)` and the instantiations of `preference(distance,1,(I,X),for(t1),(I,1,X))` and `preference(distance,2,(I,X),for(t2),(I,1,X))` where `t1` and `t2` are terms standing for the conditions of the two non-ground preference elements.

Second, Q contains the implementation of the preference type `p`, consisting of rules defining an atom `better(p)` that indicates whether $X \succ_{\texttt{p}} Y$ holds for two stable models X, Y. The sets X and Y are provided by *asprin* in reified form via unary predicates `holds` and `holds'`.[3] Further rules are added by *asprin* to define `holds` and `holds'` for the conditions appearing in the preference statement (`t1` and `t2` in our example). The definition of `better(p)` then draws upon the instances of both predicates for deciding $X \succ_{\texttt{p}} Y$. For the new preference type `maxmin` (being now part of *asprin* 2's library), we get the following rules:

```
#program preference(maxmin).
sum(P,S) :- preference(P,maxmin), preference(P,_,_,_,(S,_,_)).

value(P,S,V) :- preference(P,maxmin), sum(P,S),
  V = #sum { W,T : holds'(X), preference(P,_,_,for(X),(S,W,T)).

minvalue(P,V) :- preference(P,maxmin), V = #min { W : value(P,S,W) }.

better(P,S) :- preference(P,maxmin), sum(P,S), minvalue(P,V),
  V < #sum { W,T : holds(X), preference(P,_,_,for(X),(S,W,T)).

better(P) :- preference(P,maxmin), better(P,S) : sum(P,S).
```

Predicate `sum/2` stores the sums `S` of the preference statement `P`, while `value/3` collects the value `V` of every sum for the stable model Y, and `minvalue/2` stores the minimum of them. In the end, `better(P)` is obtained if `better(P,S)` holds for all sums `S`, and this is the case whenever the value of the sums for the stable model X is greater than the minimum value for the stable model Y.

Third, the constraint '`:- not better(distance).`' is added to Q, enforcing that the set of rules is satisfiable iff `better(p)` is obtained, which is the case whenever $X \succ_{\texttt{distance}} Y$.

We can show that for any preference statement `p` of type `maxmin`, the union of the above three sets of rules constitutes a preference program for $\succ_{\texttt{p}}$.

Automatic Guess and Check in *clingo*. Given a logic program P over $\mathcal{A}$, and a preference statement s with preference program Q_s, the optimal models of P wrt $\succ_s$ correspond to the solutions of the G&C problem $\langle P \cup R'_{\mathcal{A}}, P \cup R_{\mathcal{A}} \cup Q_s \rangle$, where (given a set X) R'_X stands for $\{\texttt{holds'(}a\texttt{) :- } a\texttt{.} \mid a \in X\}$, R_X for $\{\texttt{holds(}a\texttt{) :- } a\texttt{.} \mid a \in X\}$, and only the atoms of predicate `holds'/1` are passed to the checker.[4] The guess program generates stable models X of P reified with `holds'/1`, while the check program looks for models better than X wrt s reified with `holds/1`, so that X is optimal whenever the checker along with the `holds'/1` atoms of X becomes unsatisfiable. This correspondence is the basis of a method for computing optimal models in *asprin*, where the logic program with preferences is translated into a G&C problem, that *metagnc* translates into a disjunctive logic program, which is

[3] That is, `holds(a)` (or `holds'(a)`) is true iff $\texttt{a} \in X$ (or $\texttt{a} \in Y$).
[4] In the system *metagnc* this is declared via directive '`#guess holds'/1.`'

then solved by *clingo*. This allows, for example, for solving the *Distant Model* problem using the logic program $P \cup \{\texttt{holds(}a\texttt{,0) :- } a\texttt{.} \mid a \in \mathcal{A}\} \cup H_\chi$ and the preference statement `distance`, with the corresponding preference program comprising the three sets of rules described before.

In general, the G&C framework [11] allows for representing Σ_2^p problems in ASP, and solving them using the *saturation* technique by Eiter and Gottlob in [10]. The idea is to re-express the problem as a positive disjunctive logic program, containing a special-purpose atom `bot`. Whenever `bot` is obtained, saturation derives all atoms (belonging to a "guessed" model). Intuitively, this is a way to materialize unsatisfiability. We automatize this process in *metagnc* by building on the meta-interpretation-based approach of [14]. For a G&C problem $\langle G, C\rangle$ over $\langle \mathcal{A}_G, \mathcal{A}_C\rangle$, the idea is to reify the program $C \cup \{\{a\texttt{.}\} \mid a \in \mathcal{A}_G\}$ into the set of facts $\mathcal{R}(C \cup \{\{a\texttt{.}\} \mid a \in \mathcal{A}_G\})$. The latter are combined with the meta-encoding $\mathcal{M}$ from [14] implementing saturation. This leads to the positive disjunctive logic program:

$$\mathcal{R}(C \cup \{\{a\texttt{.}\} \mid a \in \mathcal{A}_G\}) \cup \mathcal{M}$$

This program has a stable model (excluding `bot`) for each $X \subseteq \mathcal{A}_G$ such that $C \cup X$ is satisfiable, and it has a saturated stable model (including `bot`) if there is no such X. Next, we just have to add the generator program G, map the true and false atoms of G to their counterparts in the positive disjunctive logic program (represented by predicates `true/1` and `false/1`, respectively), and enforce the atom `bot` to hold:

$$\begin{aligned}&\mathcal{R}(C \cup \{\{a\texttt{.}\} \mid a \in \mathcal{A}_G\}) \cup \mathcal{M} \cup \\ &\quad G \cup \{\texttt{true(}a\texttt{) :- } a\texttt{.} \mid a \in \mathcal{A}_G\} \cup \{\texttt{false(}a\texttt{) :- not } a\texttt{.} \mid a \in \mathcal{A}_G\} \cup \{\texttt{:- not bot.}\}\end{aligned}$$

The stable models of the resulting program correspond to the solutions of the G&C problem.

Solving queries in *asprin*. Given a logic program with preferences P and a query atom `q`, the query problem is to decide whether there is an optimal stable model of P that contains `q`. From the point of view of complexity theory, the problem is Σ_2^p-complete when P is normal. Membership holds because for solving this problem, we can use the G&C method by translating the logic program with preferences into a disjunctive logic program and adding the query as a constraint '`:- not q.`'. Hardness can be proved by a reduction of the problem of deciding the existence of a stable model of a disjunctive logic program P (see [20]).

Alternatively to the G&C approach, we propose four enumeration-based algorithms for solving this problem. All of them search for an optimal model containing the query, and their worst case occurs when there is none and they have to enumerate all solutions.

Algorithm **Q-1** enumerates stable models of $P \cup \{\texttt{:- not q.}\}$ and tests them for optimality, until one test succeeds. In the worst case, **Q-1** enumerates all stable models of the program, but still it runs in polynomial space given that it enumerates normal stable models.

Algorithm **Q-2** enumerates optimal models of P, until one contains q. In the worst case, **Q-2** enumerates all optimal models of P, and this enumeration may need exponential space (see [5]). Note that this exponential blow-up may also occur with the other algorithms **Q-3** and **Q-4**. In addition, even when **Q-2** succeeds in finding an optimal model containing the query, it may have to enumerate many optimal models without the query before reaching the right one.

Trying to alleviate this problem, algorithm **Q-3** enumerates optimal models of $P \cup \{\texttt{:- not q.}\}$, and tests whether they are also optimal for P, until one test succeeds. However, **Q-3** may have to enumerate many non optimal models of P containing the query before finding a truly optimal one.

Algorithm **Q-4** follows a different approach, enumerating optimal models of P (as **Q-2**) but modifying the algorithm of *asprin* for computing optimal models. The input of *asprin*'s algorithm is a logic program P and a preference statement s with preference program Q_s. It follows these steps:

1. Solve program P and assign the result to Y. Return Y if it is $\bot$.

2. Assign Y to X, and solve program $P \cup Q_s \cup R_{\mathcal{A}} \cup H'_X$ assigning the result to Y.
   If Y is $\bot$, return X, else repeat this step.

Step 2 searches iteratively for better models of P wrt s. With Algorithm **Q-3**, it may be the case that many iterations are performed only to find out afterwards that there was a model better than all of them but without the query. Then, the idea of **Q-4** is to alternatively add `{:- not q.}` and `{:- q.}` at Step 2 trying to avoid those cases. When a solve call at Step 2 with `{:- q.}` fails, we know that the query is true in an optimal model, and we keep iterating with `{:- not q.}` until we find such a model.[5]

Preferences over optimal models in *asprin*. Formally, this extension of *asprin* is defined as follows. Let P be a logic program over $\mathcal{A}$, and let s and t be two preference statements. A stable model X of P is optimal wrt s *and then* t if it is optimal wrt s, and there is no optimal model Y of P wrt s such that $Y \succ_t X$. From the point of view of complexity theory, when P is normal, finding a stable model optimal wrt s and then t is $F\Delta_3^p$-hard. We prove this by reducing the problem of finding an optimal stable model of a disjunctive logic program with weight minimization (see [20]). We note that finding a stable model of a normal logic program P with preferences is in $F\Sigma_2^p$, given the translation to disjunctive logic programs using the G&C method. Therefore, it is believed that we cannot find a polynomial translation to a normal program with preferences.

It turns out that the *Most Distant Optimal Model* problem can be easily formulated within this approach. Given a logic program P with a preference statement s, and a set $\mathcal{X} = \{X_1, \ldots, X_m\}$ of optimal stable models of P, the most distant optimal models for this problem correspond to the stable models of the logic program $P \cup \{$ `holds(`a`,0) :-` a`.` $\mid a \in \mathcal{A}\} \cup H_{\mathcal{X}}$ that are optimal wrt s and then `distance`. In *asprin*, this is represented simply by adding to the resulting logic program the preference statements s and `distance`, along with the declarations '`#optimize(`s`).`' and '`#reoptimize(distance).`'.

For computing optimal models of a logic program P over $\mathcal{A}$ wrt preference statements s and then t, we propose a variant of *asprin*'s iterative algorithm. Let $solveOpt(P, s)$ be the *asprin* procedure for computing one optimal model of P wrt s, and let $solveQuery(P, s, q)$ be any of our algorithms for solving the query problem given a logic program P with preference statement s and query atom q. The algorithm follows these steps:

1. Call $solveOpt(P, s)$ and assign the result to Y. Return Y if it is $\bot$.
2. Assign Y to X, and call $solveQuery(P \cup Q_t^* \cup R_{\mathcal{A}} \cup H'_X, s, \texttt{better}(t))$ assigning the result to Y.
   If Y is $\bot$, return X, else repeat this step.

where Q_t^* is the result of deleting the constraint '`:- not better(`t`).`' from a preference program Q_t for t. The first step of the algorithm computes an optimal model of P wrt s. Then Step 2, like in *asprin*'s basic algorithm, searches iteratively for better models. Specifically, it searches for optimal models of P wrt s that are better than X wrt t. Note that by construction of Q_t^*, the stable models Y of $P \cup Q_t^* \cup R_{\mathcal{A}} \cup H'_X$ are better than X wrt t iff `better(`t`)` $\in Y$. Then if *solveQuery* returns a model Y, it contains `better(`t`)`, and therefore it is better than X wrt t. On the other hand, if *solveQuery* returns $\bot$, there is no optimal model of P wrt s that is better than X wrt t, and this implies that X is an optimal model wrt s and then t.

[5] If we are only interested in *deciding* the query problem, we can stop without iterating any more and return true.

5 Advanced diversification techniques

Enumeration. With this technique, we first enumerate all optimal stable models of P with *asprin* and afterwards we find, among all those stable models, the n most diverse. For the initial step, we use *asprin*'s enumeration algorithm (see [5]). For the second, let $\mathcal{X} = \{X_1, \ldots, X_m\}$ be the set of m optimal stable models of P. Then, the following encoding along with the facts $H_\mathcal{X}$ reifying $\mathcal{X}$ provides a correct and complete solution to the *n Most Diverse Optimal Models* problem:

```
n { sol(1..m) } n.
#preference(enumeration,maxmin) {
  (I,J),1,X : holds(X,I), not holds(X,J), sol(I), sol(J), I < J;
  (I,J),1,X : not holds(X,I), holds(X,J), sol(I), sol(J), I < J;
  (I,J),#sup,0 : sol(I), not sol(J), I < J ;
  (I,J),#sup,0 : not sol(I), sol(J), I < J }.
#optimize(enumeration).
```

The choice rule guesses `n` solutions among `m` in $\mathcal{X}$, and the `enumeration` preference statement selects the optimal ones. In `enumeration`, there is a sum for every pair `(I,J)` with `I < J`. If both `I` and `J` are chosen (first two preference elements) then the sum represents their actual distance. In the other case (last two elements) the sum has the maximum possible value in *asprin* (represented by `#sup`). This allows for comparing only sums of pairs `(I,J)` of selected solutions.

Replication. With this technique *asprin* begins translating a normal logic program with preferences P into a disjunctive logic program D applying the G&C method. Next, D is reified onto $\mathcal{R}(D)$ and combined with a meta-encoding $\mathcal{M}_n$ replicating D:[6]

```
sol(1..n).
holds(A,S) : head(R,A) :- rule(R); sol(S); holds(A,S) : body(R,pos,A);
                                    not holds(A,S) : body(R,neg,A).
```

The stable models of $\mathcal{M}_n \cup \mathcal{R}(D)$ correspond one to one to the elements of $Opt(P)^n$, where $Opt(P)$ stands for the set of all optimal models of P. Further rules are added for having exactly one stable model for every set of n optimal stable models, but we do not detail them here for space reasons. Finally, adding the following preference and optimize statements results in a correct and complete solution to the *n Most Diverse Optimal Models* problem:

```
#preference(replication,maxmin) {
  (I,J),1,X : hold(A,I), not hold(A,J), sol(I), sol(J), I < J ;
  (I,J),1,X : not hold(A,I), hold(A,J), sol(I), sol(J), I < J }.
#optimize(replication).
```

The preference statement is similar to the one for Enumeration, but now the n solutions are generated by the meta-encoding, and all of them are taken into account for calculating the sums.

Approximation. We describe the different implementations of the procedure $solve(P,\mathcal{X})$ outlined in Section 3.

In Algorithm A-1, $solve(P,\mathcal{X})$ solves the *Most Distant Optimal Model* problem given the optimal stable models in $\mathcal{X}$ using the method described in Section 4.

In Algorithm A-2, $solve(P,\mathcal{X})$ first computes a partial interpretation I distant to $\mathcal{X}$ in one of the following ways:

1. A random one (named *rd*).

[6] The actual encoding handles the whole *clingo* language [13] and is more involved.

2. A heuristically chosen one, following the *pguide* heuristic from [18] (*pg*): for an atom a, a is added to I if it is true in $\mathcal{X}$ more often than false, $\neg a$ is added in the opposite case, and nothing happens if there is a tie.
3. One most distant to the solutions in $\mathcal{X}$ (*dist*), computed applying the solution to the *Most Distant Model* problem described in Section 4, where the program P is '{holds(a)}. | $a \in \mathcal{A}$}'.
4. One complementary to the last computed optimal model L taking into account either true ($\{\neg a \mid a \in L\}$), false ($\{a \mid a \notin L\}$), or both types of atoms ($\{\neg a \mid a \in L\} \cup \{a \mid a \notin L\}$). They are named *true*, *false* and *all*, respectively.

For selecting an optimal model closest to I, the technique is similar to the one for the *Most Distant Optimal Model* problem: we add to P the rules { holds(a,0) :-a. | $a \in \mathcal{A}$} reifying the atoms of P, the facts { holds(a,1). | $a \in I \cap \mathcal{A}$} ∪ { nholds(a,1). | $\neg a \in I, a \in \mathcal{A}$} reifying I, and the following preference statement:

```
#preference(partial,less(cardinality)) {
  holds(X,0), nholds(X,1); not holds(X,0), holds(X,1) }.
```

Finally, we compute the optimal models of this program wrt s and `partial` using the method for preferences over optimal models described in Section 4.

In Algorithm **A-3**, we select a distant solution I as we do for **A-2**, we add the same reifying rules, and we add the following heuristic rules for approximating an optimal model of P close to I:

```
#heuristic hold(X,0) :  holds(X,1). [  1, sign ]
#heuristic hold(X,0) : nholds(X,1). [ -1, sign ]
```

For prioritizing the variables in I, we add another two heuristic rules like the previous ones, but replace both `[ 1, sign ]` and `[ -1, sign ]` by `[ 1, level ]`, respectively.

6 Experiments

In this section, we present experiments focusing on the *approximation* techniques of the *asprin* system for obtaining most dissimilar optimal solutions. While *enumeration* and *replication* provide exact results, they need to calculate and store a possibly exponential number of optimal models or deal with a large search space, respectively. Those techniques are therefore not tractable for most practical applications. For Algorithm **A-2**, we considered the variations rd, pg, dist, dist-to, true, false, and all. In dist, we issued no timeout for the computation of the partial interpretation, while in dist-to, we set a timeout for this computation of half the total possible runtime. For Algorithm **A-3**, we consider the variations that include no extra ASP computation, namely, rd, pg, dist, dist-to, true, false, and all. We also evaluated a version without any heuristic modification (named simply **A-3**). Furthermore, following [18], we considered a variation of pg, viz. pg-l, where the atoms of the selected partial interpretation are given a higher priority, and pg-l-rd, extending pg-l by fixing initially a random sign to all atoms not appearing in the partial interpretation.

We gathered 186 instances from six different classes: *Design Space exploration (DSE)* from [1], *Timetabling (CTT)* from [3], *Crossing minimization* from ASP competition 2013, *Metabolic network expansion* from [21], *Biological network repair* from [12] and *Circuit Diagnosis* from [23]. Since we required instances with multiple optimal solutions, we exclusively focus on Pareto optimality. DSE and CTT are inherently multi-objective and therefore we could naturally define a Pareto preference for them. For the other classes, we turned single-objective into multi-objective optimization problems by distributing their optimization statements. First, we split the atoms in the optimization statements into four or eight groups evenly. We chose for each group the same preference type, either cardinality or subset minimization, and defined a Pareto preference that incorporates all cardinality or the subset

Class	T	TO
A-3	**165**	**70**
A-3-true	200	113
A-3-all	202	118
A-3-rd	277	280
A-3-pg	317	351
A-3-pg-l-rd	354	442
A-3-false	351	443
A-3-pg-l	351	443
A-2-true	482	618
A-2-rd	474	648
A-1	482	672
A-2-dist-to	528	689
A-2-all	515	696
A-2-false	532	696
A-2-pg	542	708
A-2-dist	572	773

Class	S	avg
A-1	**15**	0.13
A-2-dist-to	14	0.14
A-2-pg	13	**0.18**
A-3-pg-l	11	0.17
A-3-pg-l-rd	10	0.16
A-2-all	10	0.15
A-2-dist	8	0.07
A-2-false	8	0.15
A-2-true	7	0.12
A-3-false	6	0.16
A-2-rd	5	0.12
A-3-all	5	0.08
A-3-true	4	0.08
A-3-rd	2	0.09
A-3-pg	1	0.09
A-3	0	0.06

Class	S	avg
A-1	**15**	12.25
A-2-dist-to	13	10.38
A-3-pg-l-rd	13	11.82
A-2-dist	12	5.31
A-3-pg-l	12	11.10
A-2-pg	10	**12.86**
A-2-rd	9	8.77
A-3-all	7	3.99
A-3-true	6	4.00
A-3-false	6	7.07
A-2-false	6	6.80
A-2-all	4	6.98
A-2-true	3	5.31
A-3-rd	2	6.43
A-3	2	4.28
A-3-pg	0	2.79

Table 1 Comparison of approximation techniques by (a) runtime and timeouts, (b) diversification quality, and (c) minimal distance

preferences respectively. We calculated optimal solutions of that regarding those Pareto preference statements. The same was done for CTT and DSE. An instance was selected if for some Pareto preference ten optimal solutions could be obtained within 600 seconds by *asprin*. This method generated 816 instances in total. We ran the benchmarks on a cluster of Linux machines with dual Xeon E5520 quad-core 2.26 GHz processors and 48 GB RAM. We restricted the runtime to 600 seconds and the memory usage to 20 GB RAM.

Since algorithms A-1 and A-2 involve querying programs over preferences, we started by evaluating the different query techniques. For that, we executed A-1 with query methods Q-1 to Q-4 on all selected instances, stopping after the first *solveQuery* call was finished. The performance of query techniques Q-2, Q-3, and Q-4 was similar regarding runtime and only Q-1 was clearly worse. We selected Q-4 for the remaining experiments due to its slightly lower runtime. For more detailed tables, we refer to [20].

Next, we approximated four most diverse optimal models with methods A-1 to A-3. We measured runtime and two quality measures. The first, called diversification quality [18], gives the sum of the Hamming distances among all pairs of solutions normalized to values between zero and one. The second is the minimal distance among all pairs of solutions of a set in percent. The solution set size of four was chosen because [22] claims that three solutions is the optimal amount for a user, and considering one additional solution provides further insight into the different quality measures. For all algorithms that do not use heuristics for diversification, we instead enabled heuristics preferring a negative sign for the atoms appearing in preference statements, since it was observed in [6] that this improved the performance of the system.

Table 1(a) provides in column T the average runtime and in column TO the sum of timeouts. The different methods are ordered by the number of timeouts. The best results in a column are shown in bold. We see that A-3 is by far the fastest with 70 timeouts, solving 91% of the instances. Heuristic variations of A-3 perform the best after that. Less invasive heuristics achieve similar runtimes with 113-118 timeouts. More sophisticated heuristics perform worse at 349-443 timeouts. In a range from 618 to 773 timeouts, non-heuristic methods solve the least instances by a significant margin.

The results are in tune with the nature of the methods. Heuristics modifying the solving process for diversity decrease the performance in comparison with solving heuristics aimed at performance, but not as much as more complex methods involving preferences over optimal models.

In particular, non-heuristic methods show many timeouts. If we tried to analyze the quality of the solutions by assuming worst possible values for the instances that timed out, the results would be dominated by these instances. To avoid that, we calculated a score independent of the runtime. We considered all possible parings of the different methods. For each pair, we compared only instances where both found a solution set. The method with better quality value for the majority of instances receives a point. Finally, we ordered the subsequent tables according to that score.

In Table 1(b), for each method we see the score in column S, and the average of the diversification quality (over the instances solved by the method) in column avg. This way, we can examine the quality a method has achieved compared to other methods, and also the individual average quality. **A-1** has the best quality with a score of 15, followed by **A-2**-dist-to, **A-2**-pg, **A-3**-pg-l and **A-3**-pg-l-rd. All of those techniques regard the whole previous solution set to calculate the next solution and guide the solving strictly to diversity. **A-2**-pg, **A-3**-pg-l and **A-3**-pg-l-rd are also the first, second and third place, respectively, for average diversification quality. Next, with scores ranging from 10-7, we see **A-2** methods that do not take into account the whole previous set, or that were simply unable to find many solutions at all, as in the case of **A-2**-dist. Finally, we observe that **A-3** variations only regarding the last solution or no previous information perform worst in score and average. In these cases, the heuristic does not seem to be strong enough to steer the solving to high quality solution sets, and **A-3** uses no heuristic or optimization techniques to ensure diverse solutions.

Table 1(c) provides information in analogy to Table 1(b) for the minimal distance among the solutions. The overall grouping of the methods is similar to Table 1(b). The best methods considering score and average minimal distance, viz. **A-1**, **A-2**-dist-to, **A-3**-pg-l-rd, **A-3**-pg-l, **A-2**-pg, utilize information from the whole previous solution set and have strict diversification techniques.

Overall, plain heuristic methods perform better as regards runtime while more complex methods, depending on all previous solutions, lead to better quality. Furthermore, **A-3**-pg-l-rd and **A-3**-pg-l provide the best trade-off between performance and quality. While **A-1**, **A-2**-dist-to and **A-2**-pg achieve higher quality, they could solve only 18%, 16% and 13% of the instances. On the other hand, **A-3**-pg-l-rd and **A-3**-pg-l provide good diversification quality and minimal distance while solving 46% of the instances.

7 Discussion

We presented a comprehensive framework for computing diverse (or similar) solutions to logic programs with generic preferences and implemented it in *asprin* 2 available at [2]. To this end, we introduced a spectrum of different methods, among them, generalizations of existing work to the case of programs with general preferences. Hence, certain fragments of our framework provide implementations of the proposals in [9, 25]. While the latter had to resort to solver wrappers or even internal solver modifications, *asprin* heavily relies upon multi-shot solving that allows for an easy yet fine-grained control of reasoning processes. Moreover, we provided several generic building blocks, such as *maxmin* (and *minmax*) preferences, query-answering for programs with preferences, preferences among optimal models, and an automated approach for the generate and test methodology of [11], all of which are also of interest beyond diversification. Finally, we took advantage of the uniform setting offered by *asprin* 2 to conduct a comparative empirical analysis of the various methods for diversification. Generally speaking, there is a clear trade-off between performance and diversification quality, which allows for selecting the most appropriate method depending on the hardness of the application at hand.

References

1 B. Andres, M. Gebser, M. Glaß, C. Haubelt, F. Reimann, and T. Schaub. Symbolic system synthesis using answer set programming. In Cabalar and Son [8], pages 79–91.

2

3 M. Banbara, T. Soh, N. Tamura, K. Inoue, and T. Schaub. Answer set programming as a modeling language for course timetabling. *Theory and Practice of Logic Programming*, 13(4-5):783–798, 2013.

4 C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

5 G. Brewka, J. Delgrande, J. Romero, and T. Schaub. asprin: Customizing answer set preferences without a headache. In B. Bonet and S. Koenig, editors, *Proceedings of the Twenty-Ninth National Conference on Artificial Intelligence (AAAI'15)*, pages 1467–1474. AAAI Press, 2015.

6 G. Brewka, J. Delgrande, J. Romero, and T. Schaub. Implementing preferences with asprin. In F. Calimeri, G. Ianni, and M. Truszczyński, editors, *Proceedings of the Thirteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'15)*, volume 9345 of *Lecture Notes in Artificial Intelligence*, pages 158–172. Springer-Verlag, 2015.

7 G. Brewka, I. Niemelä, and M. Truszczyński. Answer set optimization. In G. Gottlob and T. Walsh, editors, *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI'03)*, pages 867–872. Morgan Kaufmann Publishers, 2003.

8 P. Cabalar and T. Son, editors. *Proceedings of the Twelfth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'13)*, volume 8148 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2013.

9 T. Eiter, E. Erdem, H. Erdogan, and M. Fink. Finding similar/diverse solutions in answer set programming. *Theory and Practice of Logic Programming*, 13(3):303–359, 2013.

10 T. Eiter and G. Gottlob. On the computational cost of disjunctive logic programming: Propositional case. *Annals of Mathematics and Artificial Intelligence*, 15(3-4):289–323, 1995.

11 T. Eiter and A. Polleres. Towards automated integration of guess and check programs in answer set programming: a meta-interpreter and applications. *Theory and Practice of Logic Programming*, 6(1-2):23–60, 2006.

12 M. Gebser, C. Guziolowski, M. Ivanchev, T. Schaub, A. Siegel, S. Thiele, and P. Veber. Repair and prediction (under inconsistency) in large biological networks with answer set programming. In F. Lin and U. Sattler, editors, *Proceedings of the Twelfth International Conference on Principles of Knowledge Representation and Reasoning (KR'10)*, pages 497–507. AAAI Press, 2010.

13 M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. *Clingo* = ASP + control: Preliminary report. In M. Leuschel and T. Schrijvers, editors, *Technical Communications of the Thirtieth International Conference on Logic Programming (ICLP'14)*, volume arXiv:1405.3694v1 of *Theory and Practice of Logic Programming, Online Supplement*, 2014. Available at `http://arxiv.org/abs/1405.3694v1`.

14 M. Gebser, R. Kaminski, and T. Schaub. Complex optimization in answer set programming. *Theory and Practice of Logic Programming*, 11(4-5):821–839, 2011.

15 M. Gebser, B. Kaufmann, R. Otero, J. Romero, T. Schaub, and P. Wanko. Domain-specific heuristics in answer set programming. In M. desJardins and M. Littman, editors, *Proceedings of the Twenty-Seventh National Conference on Artificial Intelligence (AAAI'13)*, pages 350–356. AAAI Press, 2013.

16 M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.

17 E. Hebrard, B. Hnich, B. O'Sullivan, and T. Walsh. Finding diverse and similar solutions in constraint programming. In M. Veloso and S. Kambhampati, editors, *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI'05)*, pages 372–377. AAAI Press, 2005.

18 A. Nadel. Generating diverse solutions in SAT. In K. Sakallah and L. Simon, editors, *Proceedings of the Fourteenth International Conference on Theory and Applications of Satisfiability Testing (SAT'11)*, volume 6695 of *Lecture Notes in Computer Science*, pages 287–301. Springer-Verlag, 2011.

19 V. Pareto. *Cours d'economie politique*. Librairie Droz, 1964.

20 J. Romero, T. Schaub, and P. Wanko. Computing diverse optimal stable models. In Manuel Carro and Andy King, editors, *Technical Communications of the Thirty-second International Conference on Logic Programming (ICLP'16)*. Leibniz International Proceedings in Informatics (LIPIcs), 2016. To appear.

21 T. Schaub and S. Thiele. Metabolic network expansion with ASP. In P. Hill and D. Warren, editors, *Proceedings of the Twenty-fifth International Conference on Logic Programming (ICLP'09)*, volume 5649 of *Lecture Notes in Computer Science*, pages 312–326. Springer-Verlag, 2009.

22 H. Shimazu. Expertclerk: Navigating shoppers' buying process with the combination of asking and proposing. In B. Nebel, editor, *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI'01)*, pages 1443–1448. Morgan Kaufmann Publishers, 2001.

23 S. Siddiqi. Computing minimum-cardinality diagnoses by model relaxation. In T. Walsh, editor, *Proceedings of the Twenty-second International Joint Conference on Artificial Intelligence (IJCAI'11)*, pages 1087–1092. IJCAI/AAAI Press, 2011.

24 P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2):181–234, 2002.

25 Y. Zhu and M. Truszczyński. On optimal solutions of answer set optimization problems. In Cabalar and Son [8], pages 556–568.