# *Computing Diverse Optimal Stable Models*

Javier Romero[1] Torsten Schaub[1,2] Philipp Wanko[1]
[1] *Universität Potsdam* [2] *INRIA Rennes*

*submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003*

**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 ~~even mere~~ general interest beyond diversification. The enabling factor of this lies in the recent advance of ASP multi-shot capacities that provide us with fine-grained control over ASP reasoning processes and notably abolish 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.

## 1 Introduction

Answer Set Programming (ASP; (Baral 2003)) has become a prime paradigm for solving combinatorial problems in the area of 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 in order 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, as we encounter in design space exploration (Andres et al. 2013) or timetabling (Banbara et al. 2013). Other examples include product configuration, planning, and phylogeny, as discussed in (Eiter et al. 2013). This calls for computational support that allows for identifying small subsets of diverse solutions. The computation of diverse answer sets was first considered in (Eiter et al. 2013). The analogous problem regarding optimal answer sets is addressed in (Zhu and Truszczyński 2013) in the context of answer set optimization (Brewka et al. 2003). Beyond ASP, the computation of diverse solution is also studied in SAT (Nadel 2011) and CP (Hebrard et al. 2005).

We introduce a comprehensive framework for computing diverse (or similar) solutions to logic programs with preferences. One of its distinguishing factor's 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* (Brewka et al. 2015). 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 existing basic ASP techniques that we automate and integrate in our framework. The enabling factor of this is the recent advance of ASP multi-shot solving capacities that allow for an easy yet fine-grained control of ASP-based reasoning processes (cf. (Gebser

et al. 2014)). 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 exhaustive empirical evaluation contrasting a whole spectrum of 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

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

where each $\mathtt{a}_i$ is a propositional atom and `not` stands for *default negation*. We call a rule a *fact* if $m = n = 1$, *normal* if $m = 1$, and an *integrity constraint* if $m = 0$. Semantically, a logic program induces a collection of *stable models*, which are distinguished models of the program determined by stable models semantics; see (Gelfond and Lifschitz 1991) for details.

To facilitate 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* (Simons et al. 2002). The former are of the form $\mathtt{a}\mathtt{:}\mathtt{b}_1\mathtt{,}\ldots\mathtt{,}\mathtt{b}_m$, 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. 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. Finally, objective functions minimizing the sum of weights $w_j$ of conditional literals $c_j$ are expressed as $\mathtt{\#minimize\{}w_1\mathtt{:}c_1\mathtt{,}\ldots\mathtt{,}w_n\mathtt{:}c_n\mathtt{\}}$. Specifically, we rely in the sequel on the input language of the ASP system *clingo* (Gebser et al. 2014); 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. For simplicity, we consider a Hamming distance between two stable models $X, Y$ of a program $P$ over $\mathcal{A}$, defined as $d(X,Y) = |\mathcal{A} - X - Y| + |X \cap Y|$. Given a logic program $P$ with preferences and a positive integer $n$, we follow (Eiter et al. 2013) in defining 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 in the following problem: 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* (Brewka et al. 2015), 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){c1,...,cn}` where `p` and `t` are the name and type of the preference relation, respectively, and each $\texttt{c}_\texttt{j}$ is a conditional literal[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. To this end, users provide 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'`.[2] The definition of `better(p)` then draws upon the instances of both predicates for deciding $X \succ_\texttt{p} Y$. Note that this delineates the expressiveness of preference handling in *asprin*.

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`, respectively. 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. The value of $k$ serves as argument to the respective modification. Modifiers `true` and `false` combine `level` with `sign` selection, respectively. A more detailed description can be found in (Gebser et al. 2013).

## 3 Our diversification framework at a glance

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

**Basic solving techniques.** We first summarize several basic solving techniques that provide essential pilars 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*. (Eiter and Polleres 2006) defined a framework for representing and solving $\Sigma_2^p$ problems in ASP. Given two normal logic programs $P$ and $Q$ capturing a guess-and-check problem, the role of $P$ is to guess a stable model $X$, such that $X$ is a solution to $P, Q$, if $Q \cup X$ is unsatisfiable. We automatize this by using reification along with the meta-encoding methodology of *metasp* (Gebser et al. 2011). In this way, the two normal programs $P$ and $Q$ are translated into a single disjunctive logic program. The resulting mini-system *metasp-gnt* [1]is implemented in Python and available at (asprin ). [2]We build upon this approach when dealing with logic programs with preferences. To this end, *asprin* translates a logic program with preferences into a guess-and-check problem which is then translated by *metasp-gnt* into a disjunctive logic program and solved by an ASP system.

[1] ?
[2] T: (metasp )?!

*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 $q$. To this end, we elaborate upon four alternatives and empirically evaluate them in Section 6.

[1] See (Brewka et al. 2015) and Section 4.1 for more general preference elements.
[2] That is, `holds(a)` (or `holds'(a)`) is true iff $\texttt{a} \in X$ (or $\texttt{a} \in Y$).

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

The first two methods were implemented by (Zhu and Truszczyński 2013) in the case of programs with *aso* preferences (Brewka et al. 2003). We generalize both to arbitrary preferences, propose two novel ones, and provide all four methods in *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.** We elaborate upon three ways of diversification, viz. enumeration, replication, and approximation, to determine the $n$ most diverse optimal stable models of a logic programs with preferences. 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 (Eiter et al. 2013) 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 logic program $P$ with preferences into a disjunctive logic program $D$ by applying the aforementioned guess-and-check method.
2. Reify $D$ into a set $R_D$ of facts and add a meta-encoding $M$ replicating ~~$P$~~ $D$ such that every stable model of $M \cup R_D$ corresponds to $n$ optimal models of the original logic program $P$.
3. Turn the disjunctive logic program $M \cup R_D$ 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 (Eiter et al. 2013) but not automated. We generalize this approach to programs with preferences and provide a fully automated approach.

*Approximation*. Our approximation techniques can be understood as instances of Algorithm 1. 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 by enhancing procedure $solve(P, \mathcal{X})$:

**Algorithm 1:** $iterative(P, n)$

**Input :** A logic program $P$ with preferences and a positive integer $n$
**Output :** A set of optimal stable model of $P$, or $\{\bot\}$

```
1 X = {solve(P, ∅)};
2 while test(X) do
3     X = X ∪ solve(P, X);
4 return solution(X);
```

1. $solve(P, \mathcal{X})$ returns an optimal model of $P$ most dissimilar 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 (Eiter et al. 2013) for addressing diversity in the context of logic programs without preferences; we lift it here to programs with preferences.
2. $solve(P, \mathcal{X})$ first computes a partial interpretation $I$ distant to $\mathcal{X}$, and returns an optimal model of $P$ most similar to $I$.
   (a) Select a partial interpretation $I$ in one of the following ways: [4]
      i a random one
      ii a heuristically chosen one
      iii one most dissimilar to the solutions in $\mathcal{X}$ [5]
      iv one complementary to the last computed optimal model, taking into account either true, false, or both types of atoms.
   (b) Use a cardinality-based preference minimizing the distance to $I$. Apply the aforementioned approach to preferences over optimal models to enforce this preference among the optimal models of $P$.
3. $solve(P, \mathcal{X})$ returns any optimal model (not in $\mathcal{X}$).

[4] T: possibly put (i) to (iv) inline in case we need space!

[5] (using ASP for the computation).

We can refine the previous methods by combining them with heuristics. [6]

[6] T: Perhaps modify enumeration tags for better reference!?

1. For 2, prefer assigning atoms the same sign as in the selected partial interpretation $I$.
2. For 3, select a partial interpretation $I$ as in 2, and guide the computation of the optimal model by fixing the sign of the atoms to their value in $I$.
3. For 1 to 3, apply dynamic heuristics. That is, when the current assignment is somewhat close to a previous solution, heuristically prefer opposite signs to get away from it.
4. Furthermore, different priorities may be given to the atoms depending upon [7].

[7] what?

## 4 Basic solving techniques

### 4.1 *Maxmin optimization in asprin*

- All methods apply $maxmin$ optimization via *asprin* preference type $maxmin$.
- *asprin* preference type *maxmin* is defined as:
  — $dom(maxmin)$ is $\mathcal{P}(\{g, w, t : F\})$, where $g$ and $w$ are integers, and $t$ is a term tuple, $F$ is a boolean formula, and $\mathcal{P}$ stands for the power set.

We say that $g$ appears in $E$ if there is some preference element with $g$ as the first term. Given a set of preference elements of that form, $maxmin$ maps these elements to the preference relation defined as follows. Given an stable model $X$, a set of preference elements $E$, and an integer $g$ standing for a group, let $w(X, E, g)$ be

$$\sum_{(w,\boldsymbol{t})\in\{w,\boldsymbol{t}|g,w,\boldsymbol{t}:F\in E,X\models F\}} w$$

Then

$$X > Y \text{if } \min\{w(X,E,g) \mid g \text{ appears in } E\} > \min\{w(Y,E,g) \mid g \text{ appears in } E\}$$

- Switching the signs of the weights in the preference statements, we get $minmax$ preference, and with only one group, it reduces to $more(weight)$ (or $less(weight)$, switching the signs).
- The preference type is implemented by the following preference program:

```
#program preference(maxmin).
%%% gather groups
group(P,G) :- preference(P,maxmin), preference(P,_,_,_,(G,_,_)).

%%% holds must be better
better(P) :- preference(P,maxmin),
             better(P,G) : group(P,G).

better(P,G) :- preference(P,maxmin), group(P,G), valueh'(P,V),
               V < #sum { W,T : holds(X), preference(P,_,_,for(X),(G,W,T)).

%%% get min value for holds'
valueh'(P,G,V) :- preference(P,maxmin), group(P,G),
                  V = #sum { W,T : holds'(X), preference(P,_,_,for(X),(G,W,T
valueh'(P,V) :- preference(P,maxmin), V = #min { VV : valueh'(P,G,VV) }.
```

- The naive implementation of this preference in *clingo* via $\#minimize$ statements leads to large groundings:

```
group(G) :- preference(P,maxmin), preference(P,_,_,_,(G,_,_)).
dist(X)  :- group(G), X = #sum { W,T : holds(X), preference(P,_,_,for(X),(G,
 min(X)  :- X = #min { Y: dist(Y) }.
#maximize { X : min(X) }.
```

Let us assume that the $maxmin$ preference is specified directly via facts of predicates `preference/2` and `preference/5`, and that `holds/1` is defined as in *asprin*. Then the first line defines the groups of the $maxmin$ preference (with predicate `group/1`), the second line stablishes when the distance to any group is `X` (with predicate `dist/1`), and this is used in the third line to determine the minimal distance of any of the groups (with predicate `min/1`), which is finally maximized in the last line. The large grounding is due to the second rule, leading to a ground rule for every group and for every possible value of the distance to that group. In the longer version of this paper we investigate other possible encodings, and compare them with the *asprin* implementation.

### *4.2 Guess and Check in clingo*

[8]

[8] JR: This is exactly Eiter and Polleres paper :(, I changed 'guess and check' to 'guess and check', the name they use

*Definition 1* (*Guess and Check (Eiter and Polleres 2006)*)
Let $G$ and $C$ be two logic programs, and $X$ an interpretation of $G$. $X$ is a guess and check solution for $\langle G, C\rangle$ if $X$ is a stable model of $G$, and $X \cup C$ is unsatisfiable.

- Guess and Check (GC) is a useful setting for representing problems at the second level of the polynomial hierarchy. [9]
- We allow selecting only some predicates of the guess program to be passed to the checker program. This is expressed via a guess directive

  ```
  #guess p/n.
  ```

  where `p` is a predicate of arity `n` appearing in $G$. [10]Given a logic program $G$, we say that a set of guess directives $D$ selects the atoms $A_D$ appearing in $G$ whose signature coincide with that specified by any of the guess directives. Then, $X$ is a guess and check solution for $\langle G, C\rangle$ constrained by a set of guess directives $D$ if $X$ is a stable model of $G$, and $(X \cap A_D) \cup C$ is unsatisfiable, where $A_D$ are the atoms selected by $D$ in $G$.
- Example (quantified boolean cnf). Let $\exists X \forall Y \phi$ be a quantified boolean CNF formula, where $\phi$ is a CNF formula over atoms $X \cup Y$ such that $X \cap Y = \emptyset$. [11]This can be represented in ASP via facts:
  - `clause(C)`: for every clause `C` in $\phi$
  - `exists(V)`: for every variable `V` $\in X$
  - `forall(V)`: for every variable `V` $\in Y$
  - `pos(C,V)`: for every positive literal `V` in clause `C`.
  - `neg(C,V)`: for every negative literal `V` in clause `C`.

  Let $G$ be the program:

  ```
  { holds(X) : exists(X) }.
  ```

  and $C$ be the program:

  ```
  { holds(X) : forall(X) }.
  bot :- clause(C); not holds(X) : pos(C,X);
                       holds(X) : neg(C,X).
  :- not bot.
  ```

  Let $F$ be the set of facts representing a formula $\phi$, then the guess and check solutions of $\langle G \cup F, C\rangle$ correspond one to one to the models of $\exists X \forall Y \phi$. The atom *bot* holds if the interpretation of the variables in $X \cup Y$ is not a model of $\phi$. Informally, $G$ guesses a solution $X$, then if $X \cup C$ is unsatisfiable, there is no interpretation of the atoms in $Y$ that makes $\phi$ false, which means that for all interpretations of the atoms in $Y$, $\phi$ is true, and the boolean formula holds.
- Example (conformant planning). [12]Let $C = \langle F, A, T, I, G, n\rangle$ be a conformant planning problem with fluents $F$, actions $A$, transition function $T : F \times A \to F$, initial fluents $I \subseteq F$, goal fluent $G \in F$, and a positive integer $n$ representing the plan length. The transition function $T$ induces a transition diagram $D_T = \langle S, E\rangle$ with states $S = \{s \mid s \subseteq F\}$ and arcs from $s_1$ to $s_2$ labelled by $a$ if $T(s_1, a) = s_2$). A solution to $C$ is a sequence of

[9] JR: I put three examples here, but I don't know whether the first two should go. The first (2QCNF) is good for proving the hardness of the problem, the second (conformant planning) shows how to represent easily an interesting problem, and the third is asprin.
JR: Eiter and Polleres have 2QDNF, conformant planning and strategic companies.
[10] JR: Should we add this? It is nice for the user, and nice for the *asprin* definition. Should we specify it formally? I'd say no, but just in case, I add the formalities next.
[11] JR: Eiter and Polleres to DNF, instead of CNF

[12] JR: If we want this to stay, I can make it much cleaner

actions $a_1, a_2, \ldots, a_{n-1}, a_n$ such that for all possible states $I' \in S$, if $I \subseteq I'$ then there is a path of length $n$ in $D_T$ from $I'$ to a state $s_f$ such that $g \in s_f$. Let $P_T$ be a logic program representing all paths of length $n$ in $D_T$. Predicate `holds(F,T)` stands for fluent `F` being true at state `T` of the path, and `occurs(A,T)` stands for action `A` connecting states `T-1` and `T` of the path. Let $P$ be the program:

```
{ occurs(A,T) : action(A) } :- T=1..n.
```

and $Q$ be the program:

```
:- not holds(F,0), initial(F).
:- holds(goal,n).
```

The guess and check solutions of $\langle P, Q \cup P_T \rangle$ correspond one to one to the conformant plans of the problem.

- Example. Preferences in *asprin*. Let $P$ be a logic program with signature $\mathcal{A}$, let $s$ be a preference statement defining preference relation $\succ_s$ over $\mathcal{A} \times \mathcal{A}$, and $Q$ a preference program for $s$. The guess and check solutions of $\langle P \cup \{holds'(a) \leftarrow a \mid a \in \mathcal{A}\}, P \cup Q \cup \{holds(a) \leftarrow a \mid a \in \mathcal{A}\}\rangle$ constrained by $\{$`#guess` $holds'$`/1.`$\}$ correspond to the $\succ_s$-preferred stable models of $P$.
- Implementation. [13]
  - — Eiter and Gottlob invented the *saturation* technique. 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. For automatizing this process, we build upon the meta-interpretation-based approach in (Gebser et al. 2011). The idea is to map a program $R$ onto a set $\mathcal{R}(R)$ of facts via reification.The set $\mathcal{R}(R)$ of facts is then combined with a meta-encoding $\mathcal{M}$ from (Gebser et al. 2011) implementing saturation.
  - — In our case, we consider for a GC problem $\langle G, C \rangle$ the positive disjunctive logic program

$$\mathcal{R}\big(C \cup \{\{a\} \mid a \in \mathcal{A}_\mathcal{G}\}\big) \cup \mathcal{M} \,.$$

  - — This program has a stable model (excluding $bot$) for each $X \subseteq \mathcal{A}_\mathcal{G}$ such that $X \cup C$ is satisfiable, and it has a saturated stable model (including $bot$) if there is no such $X$.
  - — For computing a solution to the $GC$ problem, one just has to add the generator program $G$, map the true and false atoms of $G$ to their counterparts in the positive disjunctive logic program, and inforce the atom $bot$ to hold

$$G \cup \mathcal{R}\big(C \cup \{\{a\} \mid a \in \mathcal{A}_\mathcal{G}\}\big) \cup \mathcal{M} \cup$$

$$\{true(atom(a)) \leftarrow a \mid a \in X\} \cup \{false(atom(a)) \leftarrow not\ a \mid a \in X\} \cup \{\leftarrow\ not\ bot\} \,.$$

- Deciding whether there is a solution to a GC problem is $\Sigma_2^p$-complete. Membership comes from the translation to disjunctive logic programming, and hardness comes from the translation from quantified boolean CNF formulas.
- Differences with (Eiter and Polleres 2006): [14]
  - — Our encoding avoids "guessing" a level mapping to describe the formation of a counterexample, but directly denies models for which there is no such construction.

[13] JR: I copy the explanation from the Draft of Preferences

[14] JR: Copied from the differences stated in metasp paper

— Notably, our meta-programs apply to (reified) extended logic programs (Simons et al. 2002), possibly including choice rules and #sum constraints, and we are unaware of any existing meta-encoding of their answer sets, neither as candidates nor as counterexamples refuting optimality

In this section, we implement Eiter and Polleres framework with the metaencoding and reification of metasp. [15]

[15] JR: Not much... If we wanted, one way to go would be giving another implementation (maybe for the long paper, I dont now?) An easy one is using Tomi's tools to translate logic programs $P$ and $Q$ to CNF, and then calling a QBF solver. Another, which I'd really like to do, is doing it right inside clasp, with two interleaved solvers (maybe with SMT?) But I guess that becomes another paper.

### *4.3 Solving queries in asprin*

*Definition 2* (*Query Problem*)
Let $P$ be a logic program over $\mathcal{A}$, let $s$ be a preference statement, and $q$ an atom of $\mathcal{A}$, decide if any $\succ_s$-preferred stable model of $P$ contains $q$.

[16]

*Definition 3* (*Query Problem*)
Let $P$ be a logic program over $\mathcal{A}$, let $s$ be a preference statement, and $q$ an atom of $\mathcal{A}$, find a $\succ_s$-preferred stable model of $P$ containing $q$.

[16] JR: Posed as a model finding problem

Methods:

1. Enumerate all *models* of $P \cup \{\bot \leftarrow \sim q\}$ and test each for optimality. In the worst case, this method may enumerate first all models satisfying the query which are not optimal.
2. Enumerate all *optimal models* of $P$ until one satisfies $q$. In the worst case, this method may enumerate first all optimal models not satisfying the query.
3. Enumerate all *optimal models* of $P \cup \{\bot \leftarrow \sim q\}$ and test each for optimality. In the worst case, this method may enumerate first all optimal models of $P \cup \{\bot \leftarrow \sim q\}$ which are not optimal models of $P$. Steps:
   (a) Find an optimal model $X$ of $P \cup \{\bot \leftarrow \sim q\}$. If none exists, return $\bot$, else goto 3b.
   (b) Find a stable model $Y$ of $P \cup \{\bot \leftarrow \ q\}$ better than $X$. If none exists, return $X$, else add to $P$ rules deleting the best stable model $Y$ generated, and all stable models worse than it. Goto 3a.

   In step 3b, the stable model $Y$ may be not optimal. Optionally, before adding rules to $P$, $Y$ can be further improved until an optimal stable model of $P$ is produced.
4. Modify *asprin* improving algorithm, adding alternatively $\{\bot \leftarrow \sim q\}$ or $\{\bot \leftarrow q\}$ at each step, starting with $\{\bot \leftarrow \sim q\}$, and enumerating solutions until one satisfies $q$. In the worst case, this method may enumerate first all optimal models of $P \cup \{\bot \leftarrow q\}$.
   (a) Find an optimal model $X$ of $P \cup \{\bot \leftarrow \sim q\}$. If none exists, return $\bot$, else goto 4b.
   (b) Find a stable model $Y$ of $P \cup \{\bot \leftarrow \ q\}$ that is better than $X$. If none exists, improve $X$ until an optimal stable model $X'$ of $P$ is generated and return $X'$, else goto 4c.
   (c) Find an stable model $X$ of $P \cup \{\bot \leftarrow \sim q\}$ that is better than $Y$. If one exists, goto 4b, else add to $P$ rules deleting the best stable model $Y$ generated, and all stable models worse than it, and goto 4b.

   For simply deciding whether the query holds or not (without necessarily finding a model where the query holds) in step 4b, if no stable model exists, $X$ does not have to be improved. In step 4c, if no stable model exists, the previously computed stable model $Y$ may be not optimal. Optionally, before adding rules to $P$, $Y$ can be further improved until an optimal stable model of $P$ is produced.

[17]

[17] JR: We could analyze more in detail the relationships between the algorithms

### *4.4 Preferences over optimal models in asprin*

[18] JR: Best title so far...

[18]

*Definition 4* (*Preferences over optimal models*)
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 $\succ_{s,t}$-preferred if it is $\succ_s$-preferred, and there is no $\succ_s$-preferred stable model $Y$ of $P$ such that $Y \succ_t X$.

- In *asprin*, simply add

  `#reoptimize(t).`

  where `t` is a preference statement.
- Like with `#optimize` statements, this is translated into the fact `optimize(t)`, which is added to the preference program for `t`.
- Internally, some modifications are needed to separate the preference programs for `s` and `t` (with facts `optimize(s)` and `optimize(t)`, respectively) but from a formal and a user perspective, the original definitions suffice.
  [19]
- Given a program $P$, define $P^q$ as the program

  $$(P \setminus \{r \in P \mid head(r) = \emptyset\}) \cup \{bot \leftarrow body(r) \mid r \in P, head(r) = \emptyset\} \cup \{\, q \leftarrow \sim bot\}$$

  where $bot$ and $q$ are new fresh atoms.

  *Proposition 1*
  If program $P$ is stratified, $P^q$ has a unique stable model $X$, and $P$ is satisfiable iff $q \in X$.
- Let $P$ be a program over $\mathcal{A}$, and $s$ be a preference statement, then $solveOpt(P, s)$ is a procedure that returns a $\succ_s$-preferred stable model of $P$, if $P$ is satisfiable, and $\bot$ otherwise.
  [20]
- Let $P$ be a program over $\mathcal{A}$, $s$ be a preference statement, and $q \in \mathcal{A}$ be an atom, then $solveQueryOpt(P, s, q)$ is a procedure that returns a $\succ_s$-preferred stable model of $P$ containing $q$ if it exists, and $\bot$ otherwise. [21]
-

[19] JR: This is not implemented yet!
And `reoptimize` is just a first try as a name ;)
JR: The term reoptimization is used in the optimization community with the following meaning: given an optimization problem, one instance with a solution, and another similar instance, (re-)optimize the second instance taking advantage of the first instance with its solution.

[20] JR: Should we add the specification of the algorithm?

[21] JR: If we add this, then it should go in the queries section

*Proposition 2*
Let $P$ be a program over $\mathcal{A}$, let $s$ and $t$ be preference statements, and let $O_s$ be a preference program for $s$.

1. If $X$ is a $\succ_s$-preferred stable model of $P$, then $X$ is $\succ_{s,t}$-preferred iff $\big(P \cup (O_t \cup R_{\mathcal{A}} \cup H'_X)^q\big)$ has no $\succ_s$-preferred stable model containing $q$.
2. If $Y$ is a $\succ_s$-preferred stable model of $\big(P \cup (O_t \cup R_{\mathcal{A}} \cup H'_X)^q\big)$ containing $q$ for some $X \subseteq \mathcal{A}$, then $Y \cap \mathcal{A}$ is a $\succ_s$-preferred stable model of $P$ such that $(Y \cap \mathcal{A}) \succ_t X$.

**Algorithm 2:** $solveOpt(P, s, t)$

**Input :** A program $P$ over $\mathcal{A}$ and preference statements $s$ and $t$.
**Output :** A $\succ_{s,t}$-preferred stable model of $P$, if $P$ is satisfiable, and $\bot$ otherwise.

$Y \leftarrow solveOpt(P, s)$;
**if** $Y = \bot$ **then return** $\bot$;
**repeat**
  $X \leftarrow Y$;
  $Y \leftarrow solveQueryOpt\big((P \cup (O_t \cup R_{\mathcal{A}} \cup H'_X)^q), s, q\big) \cap \mathcal{A}$;
**until** $Y = \bot$;
**return** $X$

## 5 Advanced diversification techniques

### 5.1 Enumeration

- Enumerate all optimal stable models of $P$ with *asprin*, and afterwards find, among all those stable models, the $n$ most diverse (with *asprin* again).
- This method may be exponential in space, given that we may have to compute and store an exponential number of solutions.
- For the first step, we simply enumerate all optimal stable models of $P$ with *asprin*.
- For the second step, let $\mathcal{X} = \{X_1, \ldots, X_m\}$ be the set of $m$ optimal stable models of $P$. This set may be represented in ASP via the set of atoms $A_{\mathcal{X}} = \{holds(a, i) \mid a \in X_i\}$. Consider the *asprin* encoding $E$: [22]

[22] JR: I put two encodings, the first one for asprin 1.0, the second (nicer) for asprin 2.0

```
n { select(I) : model(I) } n.
#preference(p,maxmin) {
  (I,J),1,X :: select(I) & select(J) :
holds(A,I), not holds(A,J), model(I), model(J), I < J;
  (I,J),1,X :: select(I) & select(J) : not holds(A,I),
holds(A,J), model(I), model(J), I < J
}.
```

Consider the *asprin* encoding $E$:

```
n { select(I) : model(I) } n.
#preference(p,maxmin) {
  (I,J),1,X :     holds(A,I), not holds(A,J), select(I), select(J), I < J;
  (I,J),1,X : not holds(A,I),     holds(A,J), select(I), select(J), I < J
}.
```

Then the optimal stable models of $A_{\mathcal{X}} \cup E$, computed by *asprin*, correspond to most diverse solutions of $P$.

### 5.2 Replication

- First, translate the *normal* input logic program with preferences $P$ into a disjunctive logic program without preferences $D_P$ using `asprin`. This is done applying a general framework for generate and test in ASP.

- Second, reify the resulting logic program with $reify$ tool into a set of facts $F_{D_P}$.
- Consider a metaencoding $meta$ such that the stable models of $F_{D_P} \cup meta$ correspond one to one to the stable models of $D_P$.
- For the case where $D_P$ contains no choice rules or weight constraints, $meta$ is:

```
lits(B) :- normal(A,B).
lits(B) :- disjunction(A,B).
body(B) :- lits(B),
               hold(L) : lits(B, L), L > 0;
           not hold(L) : lits(B,-L), L > 0.
hold(A)              :- normal(A,B), body(B).
hold(A) : atoms(H,A) :- disjunction(H,B), body(B).
```

- Consider metaencoding $meta(n)$ such that given a positive integer $n$, from every stable model of $F_{D_P} \cup meta(n)$, $n$ stable models of $P$ may be extracted.
- More technically, the stable models of $F_{D_P} \cup meta(n)$ correspond one to one to the elements of the set $\underbrace{SM(D_P) \times \ldots \times SM(D_P)}_{n}$, where $SM(D_P)$ stands for the set of stable models of $D_P$.
- For the case where $D_P$ contains no choice rules or weight constraints, $meta(n)$ is:

```
model(1..n).
lits(B) :- normal(A,B).
lits(B) :- disjunction(A,B).
body(B,M) :- lits(B), model(M)
                 hold(L,M) : lits(B, L), L > 0;
             not hold(L,M) : lits(B,-L), L > 0.
hold(A,M)              :- normal(A,B), body(B,M).
hold(A,M) : atoms(H,A) :- disjunction(H,B), body(B,M).
```

- Note that with this basic encoding every set of $n$ models will appear in $n!$ stable models. For having one stable model for every set of $n$ models, we add the following set of rules:

```
TO BE ADDED
```

- For computing most diverse solutions, we add the following preference specification:

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

[23]

- This method does not work if $P$ is disjunctive.

[23] JR: If we decide to keep the encodings, I can choose better predicates or print them nicer.

### *5.3 Approximation*

[24]

[24] JR: I made no changes after this point.

The following methods approximate $n$ most dissimilar solutions. They are variations of Algorithm 3.

**Algorithm 3:** $iterative(P, n)$

**Input :** $P$ is a logic program possibly with preferences, $n$ is a positive integer
**Output :** A set of solutions of $P$, or $\bot$

```
1 X = {solve(P, ∅)};
2 while test(X) do
3     X = X ∪ solve(P, X);
4 return solution(X);
```

In the basic case, $test(X)$ returns $true$ while there are less than $n$ solutions in $X$, $solution(X)$ returns the set $X$, and the algorithm simply computes $n$ solutions by calling $solve$. This can be further elaborated. For example, $test(X)$ may return $true$ until $k$ ($k \geq n$) solutions are in $X$, and $solution(X)$ returns the $n$ most dissimilar solutions among those in $X$. The algorithm is complete if $test(X)$ returns $true$ until all solutions have been computed (in which case the algorithm reduces to **enumerate all** above).

The methods differ in the implementation of the $solve(P, n)$ call. Below, every method is more imprecise than the previous ones, i.e. the solutions given are more similar than with the previous methods.

### *5.4 Find a solution most dissimilar to those in $\mathcal{X}$.*

- [3]
- Add maxmin optimization to $P$ to compute a solution that maximizes the minimal distance to any of the solutions in $\mathcal{X}$.
- Implementation: Without preferences, using Maxmin Optimization (see next subsection). With preferences, using the method for preferences over `asprin`, that uses the method for queries (see next subsection).

### *5.5 Consider a partial interpretation $I$ distant to $\mathcal{X}$, and find a solution close to $I$.*

- [4]
- Select a partial interpretation $I$:
  1. A Random one.
  2. According to $pguide$ heuristic from (A. Nadel, SAT 2011). An atom is true if among the solutions in $\mathcal{X}$ it is $false$ more times than $true$, and it is false in the opposite case. In case of a tie, it does not appear in $I$.
  3. The most dissimilar to the solutions in $\mathcal{X}$ (computed using maxmin optimization in ASP).
  4. Different to the last added element $L$ of $\mathcal{X}$ (for this, $\mathcal{X}$ should be a list). $I$ may be the result of changing all signs of $L$ ($\{\neg a \mid a \in L\} \cup \{a \mid \neg a \in L\}$), or taking only

[3] For future work, when $test(X)$ allows computing more than $n$ solutions, we could find a solution along with at most $n-1$ solutions in $X$, such that they altogether are most dissimilar. In this way, we make choices on the solution we look for, and on which of the previous solutions are also selected.

[4] For future work, one could consider looking for a solution close to $I$ for a number of conflicts, and if no solution is found, pick another partial interpretation $I'$ and continue from there.

the positive atoms of $L$ and changing the signs ($\{\neg a \mid a \in L\}$), or similarly with the negative atoms of $L$ ($\{a \mid \neg a \in L\}$).

- Apply minimization to compute a solution as close to $I$ as possible.
- Implementation: Without preferences, using normal optimization. With preferences, using the method for preferences over `asprin`, that uses the method for queries (see next subsection).

### *5.6 Find any solution of $P$.*

- No optimization here, but we expect that heuristics alone give a good approximation.
- Implementation: Without preferences, add a rule to delete the last model. Alternatively, we can simply enumerate models. With preferences, use `asprin` option `--input-optimal` to delete the last computed optimal models, and all models worse than them. Alternatively, we can simply enumerate optimal models.

### *5.7 Heuristics*

They may be combined with any of the previous three methods:

- Fix the sign of the atoms to their value in a partial interpretation $I$ selected by any of the methods above (1–4).
- Adding to modifying the signs, give priority 1 to the atoms relevant for dissimilarity, or to the atoms in the partial interpretation $I$. Furthermore, different priorities may be given depending on the $pguide$ heuristic value (i.e., the priority of atom $a$ is $abs(|\{Y \in \mathcal{X} | a \in Y\}| - |\{Y \in \mathcal{X} | \neg a \in Y\}|)$).
- Adding to modifying the signs, apply the dynamic heuristic. This heuristic, when the current assignment is very close to a previous solution, modifies the signs to get away from it.
- Different default sign heuristics could also be tried. For example, it would be interesting to try a random sign heuristic.

## 6 Experiments

## 7 Discussion

## References

ANDRES, B., GEBSER, M., GLASS, M., HAUBELT, C., REIMANN, F., AND SCHAUB, T. 2013. Symbolic system synthesis using answer set programming. See Cabalar and Son (2013), 79–91.

asprin.

BANBARA, M., SOH, T., TAMURA, N., INOUE, K., AND SCHAUB, T. 2013. Answer set programming as a modeling language for course timetabling. *Theory and Practice of Logic Programming 13,* 4-5, 783–798.

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

BREWKA, G., DELGRANDE, J., ROMERO, J., AND SCHAUB, T. 2015. asprin: Customizing answer set preferences without a headache. In *Proceedings of the Twenty-Ninth National Conference on Artificial Intelligence (AAAI'15)*, B. Bonet and S. Koenig, Eds. AAAI Press, 1467–1474.

BREWKA, G., NIEMELÄ, I., AND TRUSZCZYŃSKI, M. 2003. Answer set optimization. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI'03)*, G. Gottlob and T. Walsh, Eds. Morgan Kaufmann Publishers, 867–872.

CABALAR, P. AND SON, T., Eds. 2013. *Proceedings of the Twelfth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'13)*. Lecture Notes in Artificial Intelligence, vol. 8148. Springer-Verlag.

EITER, T., ERDEM, E., ERDOGAN, H., AND FINK, M. 2013. Finding similar/diverse solutions in answer set programming. *Theory and Practice of Logic Programming 13,* 3, 303–359.

EITER, T. AND POLLERES, A. 2006. 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.

GEBSER, M., KAMINSKI, R., KAUFMANN, B., AND SCHAUB, T. 2014. *Clingo* = ASP + control: Preliminary report. In *Technical Communications of the Thirtieth International Conference on Logic Programming (ICLP'14)*, M. Leuschel and T. Schrijvers, Eds. Theory and Practice of Logic Programming, Online Supplement, vol. arXiv:1405.3694v1. Available at `http://arxiv.org/abs/1405.3694v1`.

GEBSER, M., KAMINSKI, R., AND SCHAUB, T. 2011. Complex optimization in answer set programming. *Theory and Practice of Logic Programming 11,* 4-5, 821–839.

GEBSER, M., KAUFMANN, B., OTERO, R., ROMERO, J., SCHAUB, T., AND WANKO, P. 2013. Domain-specific heuristics in answer set programming. In *Proceedings of the Twenty-Seventh National Conference on Artificial Intelligence (AAAI'13)*, M. desJardins and M. Littman, Eds. AAAI Press, 350–356.

GELFOND, M. AND LIFSCHITZ, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Computing 9*, 365–385.

HEBRARD, E., HNICH, B., O'SULLIVAN, B., AND WALSH, T. 2005. Finding diverse and similar solutions in constraint programming. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI'05)*, M. Veloso and S. Kambhampati, Eds. AAAI Press, 372–377.

metasp.

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

SIMONS, P., NIEMELÄ, I., AND SOININEN, T. 2002. Extending and implementing the stable model semantics. *Artificial Intelligence 138,* 1-2, 181–234.

ZHU, Y. AND TRUSZCZYŃSKI, M. 2013. On optimal solutions of answer set optimization problems. See Cabalar and Son (2013), 556–568.

This article was processed using the comments style on 17 April 2016.
There remain 24 comments to be processed.