Computing Diverse Optimal Stable Models

Javier Romero, Torsten Schaub, and Philipp Wanko

No Institute Given

Abstract. We

1 Introduction

- Answer Set Programming (ASP; [1]) 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 a preference relation 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 [2] or timetabling [3]. 1
- Other examples include product configuration, planning, and phylogeny, as discussed in [4].
- This calls for computational support that allows for identifying small subsets of diverse solutions.
- The computation of diverse answer sets was first considered in [4].
- The analogous problem regarding optimal answer sets is addressed in [5] in the context of answer set optimization [6]
- Beyond ASP, the computation of diverse solution is also studied in SAT [7] and CP [8].
- Contributions
  -
    2: TO BE FILLED
  - Last but not least, our framework is easily customizable thanks to its implementation via multi-shot solving techniques. In particular, this abolishes the need for internal solver modifications that were partly necessary in previous approaches. We have implemented our approach as an extension to the preference handling framework *asprin*.
    asprin 2
- Although we concentrate on diversity, our approach applies just as well to to its dual concept of *similarity*. (This is also reflected by its implementation supporting both settings.)
- *asprin* [9]

1 T: Here we could need specifics about the application areas

2 Background

- In ASP, problems are described as *logic programs*, which are sets of *rules* of the form

  $\mathtt{a}_0$ `:-` $\mathtt{a}_1$`,...,`$\mathtt{a}_m$`,not` $\mathtt{a}_{m+1}$`,...,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 $n = 0$, and an *integrity constraint* if $\mathtt{a}_0$ is omitted. Semantically, a logic program induces a collection of *stable models*, which are distinguished models of the program determined by stable models semantics; see [10] 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* [11]. The former are of the form $\mathtt{a}$`:`$\mathtt{b}_1$`,...,`$\mathtt{b}_m$, the latter can be written as `s{`$\mathtt{c}_1$`,...,`$\mathtt{c}_n$`}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 `#minimize{`w_1`:`c_1`,...,`w_n`:`c_n`}`.
- Specifically, we rely in the sequel on the input language of the ASP system *clingo* [12]; 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. [3] T: Needed...?
- [3]Note that an empty order yields all stable models of a program.
  Hence, our contributions also apply to this base case without further mention. [4] T: Neededd...?: other choices are possible just as well.
- [4]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 [4] 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* [9], 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){c_1,...,c_n}` where `p` and `t` are the name and type of the preference relation, respectively, and each $\mathtt{c_j}$ 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 $\succ_{\mathtt{p}}$ 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 $X \succ_{\mathtt{p}} Y$ 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 $X \succ_{\mathtt{p}} Y$.
- 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`, `sign`, `true`, or `false`, 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. Modifiers `true` and `false` combine `level` with a positive and negative `sign` selection, respectively.
  The value of k serves as argument to the respective modification.
  A more detailed description can be found in [13].
- 5

5 JR: a running example would be nice

3 Our diversification framework at a glance

- We summarize the methods developed, and the contributions.

3.1 Basic solving techniques

1. Maxmin Optimization in *asprin*:
   - Definition of preference type $maxmin$: Given a set of #sum aggregates, maxmize the value of the aggregate with the minimum value among all in the set.

[1] See [9] for more general preference elements.
[2] That is, `holds(a)` (or `holds'(a)`) is true iff $\mathtt{a} \in X$ (or $\mathtt{a} \in Y$).

   - Implementation of the preference type in *asprin*.
   - Related Work: Nothing special.
   - Contributions: Definition of preference type, and implementation in *asprin*.
2. Guess and Check in *clingo*:
   - Framework defined by [14] for representing and solving Σ_2^p problems.
   - Problem: Given two logic programs P and Q, P guesses an stable model X, and X is a solution if $Q \cup X$ is unsatisfiable.
   - Method: Tanslation to disjunctive logic programming, using the reification techniques and the metaencoding of metasp [15]
   - Implemented by a Python script.
   - Application in *asprin* for translating a normal logic program with preferences into a disjunctive logic program: *asprin* translates a logic program with preferences into a guess and check problem, which is then translated into a disjunctive logic program by the Python script.
   - Related Work: [14] devised the framework, and [**?**] the implementation techniques for *clingo*.
   - Contributions: Implementation of the framework [14] for *clingo*, using the techniques of metasp [15]. Application to *asprin*.
3. Solving queries in *asprin*:
   - Problem: Find an optimal model of a program P with preferences, that satisfies a query atom q
   - Methods:
     (a) Enumerate all *optimal models* of P until one satisfies q
     (b) Enumerate all *models* of $P \cup \{\bot \leftarrow not\ q\}$ and test each for optimality
     (c) Enumerate all *optimal models* of $P \cup \{\bot \leftarrow not\ q\}$ and test each for optimality
     (d) Modify *asprin* improving algorithm, adding alternatively $\{\bot \leftarrow not\ q\}$ or $\{\bot \leftarrow q\}$ at each step, and enumerate solutions until one satisfies q.
   - Implemented in *asprin*
   - Related Work: Truczcynski et al. defined and implemented the first two methods with aso preferences, for computing an optimal model that is at a distance less or equal than k of another stable model.
   - Contributions: Solve queries for *asprin* preferences in general, and propose two more methods.
4. Preferences over optimal models in *asprin*:
   - Problem: Given a logic program with preferences P, and a preference specification s, find, among the optimal models of P, one that is optimal wrt s.
   - Method: First, compute an optimal model of P. Then, compute iteratively optimal models of P that are better than the last one wrt s, until no one exists, in which case the last one is a solution.
   - Implementation: Iterative algorithm around *asprin*. The condition of being better than the last optimal model is posed as a query, and at every step *asprin* tries to find an optimal model that satisfies the query.
   - Related Work: iterative method is well known.
   - Contributions: Define problem, methods and implement.

3.2 Advanced diversification techniques

- Enumeration:
  - Step 1: Enumerate all optimal models of the logic program P with preferences.
  - Step 2: Find among all optimal models already computed, those n which are most diverse.
  - Implementation: Step 1 is implemented via *asprin* enumeration mode, step 2 is implemented by a logic program with preferences.
  - Related work: The method appears in (Eiter et al., ICLP 2009) for logic programs without preferences.
  - Contributions: Extension of the method of (Eiter et al., ICLP 2009) to logic programs *with* preferences (this is trivial), and implementation in *asprin*.
- Replication:
  - Step 1: Translate the logic program with preferences P into a disjunctive logic program D applying the guess and check method in *asprin*.
  - Step 2: Reify D into R_D, and add a metaencoding M replicating P, such that every stable model of the metaencoding along with the reified program ($R_D \cup M$), corresponds to n optimal models of the original logic program P.
  - Step 3: Add a maxmin preference statement s such that the optimal stable models of $R_D \cup M \cup s$ correspond to n most diverse optimal stable models of the original program P.
  - Related Work: The method appears in (Eiter et al., ICLP 2009) for logic programs without preferences, but it is not automated, i.e., the user must modify himself the program P for having n solutions per stable model.
  - Contributions: Automation of the method of (Eiter et al., ICLP 2009), and extension to logic programs *with* preferences. Method for replicating a logic program (Step 2), and maxmin preference statement for selecting most diverse optimal models (Step 3).
- Approximation:
  We propose different techniques, which are variations of Algorithm 1.

Algorithm 1: $\mathit{iterative}(P, n)$

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

$\mathcal{X} = \{\mathit{solve}(P, \emptyset)\}$;
while $\mathit{test}(\mathcal{X})$ **do**
  $\mathcal{X} = \mathcal{X} \cup \mathit{solve}(P, \mathcal{X})$;
return $\mathit{solution}(\mathcal{X})$;

In the basic case, $\mathit{test}(\mathcal{X})$ returns true until there are n solutions in $\mathcal{X}$, $\mathit{solution}(\mathcal{X})$ returns the set $\mathcal{X}$, and the algorithm simply computes n solutions by calling $\mathit{solve}(P, \mathcal{X})$. This can be further elaborated.
The techniques differ in the procedure $\mathit{solve}(P, \mathcal{X})$:

1. $solve(P, \mathcal{X})$ returns an optimal model of P most dissimilar to those in $\mathcal{X}$.
   - Method: Define a maxmin preference statement s maximizing the minimal distance to any of the solutions in $\mathcal{X}$. Add s to P for computing among the optimal models of P, one that is optimal wrt s. [6]
   - Related Work: The method appears in (Eiter et al., ICLP 2009) for logic programs without preferences.
   - Contributions: Extension of the method of (Eiter et al., ICLP 2009) to logic programs with preferences, applying the method for preferences over optimal models of *asprin*, that uses the method for queries on *asprin*.

[6] JR: I didn't manage to write this nicer.

2. $solve(P, \mathcal{X})$ computes a partial interpretation I distant to $\mathcal{X}$, and returns an optimal model of P most similar to I.
   - Step 1: Select a partial interpretation I in one of the following ways:
     * A random one
     * The best according to $pguide$ heuristic from (A. Nadel, SAT 2011).
     * The most dissimilar to the solutions in $\mathcal{X}$ (using ASP for the computation).
     * Different to the last optimal model computed, taking into account either true atoms, or false atoms, or both.
   - Step 2: Define a $less(cardinality)$ preference statement s minimizing the distance to I. Add s to P for computing among the optimal models of P, one that is optimal wrt s.
   - Related Work: Truczcynski et al. defined and implemented two methods with aso preferences, for computing an optimal model that is at a distance less or equal than k of another stable model.
   - Contributions: The method seems novel.
3. $solve(P, \mathcal{X})$ returns any optimal model (not in $\mathcal{X}$).
4. Combining heuristics with the previous methods.
   - For technique 2, fix the sign of the atoms to their value in the selected partial interpretation I.
   - For technique 3, select a partial interpretation I as for technique 2, and fix the sign of the atoms to their value in I.
   - For techniques 1 to 3, apply a dynamic heuristic. This heuristic, when the current assignment is very close to a previous solution, modifies the signs to get away from it.
   - Furthermore, different priorities may be given to the atoms.

– Note: enumeration and replication are complete, while approximation is not.

4 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 $less(weight)$, swithcing 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,W,T)).

%%% 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, in the longer version of this papers we investigate other possible encodings, and compare them with the *asprin* implementation.

5 Guess and Check in *clingo*

[7]

Definition 1 (Guess and Check [14]). *Let P and Q be two logic programs, and X an interpretation of P. X is a guess and check solution for $\langle P,Q\rangle$ if X is a stable model of P and $\{holds'(a) \mid a \in X\} \cup Q$ is unsatisfiable.*

- Guess and Check (GT) is a useful setting for representing problems at the second level of the polynomial hierarchy. [8]
- Example (quantified boolean cnf). Let $\exists X \forall Y \phi$ be a quantified boolean CNF formula, where ϕ is a CNF formula over atoms $X \cup Y$ such that $X \cap Y = \emptyset$. [9]This can be represented in ASP via facts:

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

[8] 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.

[9] JR: Eiter and Polleres to DNF, instead of CNF

- `clause(C)`: for every clause `C` in ϕ
- `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 P be the program:

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

and Q be the program:

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

The guess and check solutions of $\langle P, Q\rangle$ 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 ϕ. Informally, P guesses a solution S, then if $\{holds'(a) \mid a \in S\} \cup Q$ is unsatisfiable, there is no interpretation of the atoms in Y that makes ϕ false, which means that for all interpretations of the atoms in Y, ϕ is true, and the boolean formula holds.

– Example (conformant planning). [10]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 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 the 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:

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

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

and Q be the program:

```
:- not holds(F,0), initial(F).
:- holds(goal,n).
:- not occurs(A,T), holds'(occurs(A,T)).
```

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 $\langle P, P \cup Q \cup \{holds(a) \leftarrow a \mid a \in \mathcal{A}\}\rangle$ correspond to the $\succ_s$-preferred stable models of P.

– Implementation. [11]

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

- 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 [15]. 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 [15] implementing saturation.
- In our case, we consider for a GT problem $\langle P, Q\rangle$ the positive disjunctive logic program
$$\mathcal{R}\big(Q \cup \{\{holds'(a)\} \mid a \in \mathcal{A}_{\mathcal{P}}\}\big) \cup \mathcal{M}\ .$$
- This program has a stable model (excluding bot) for each $X \subseteq \mathcal{A}_{\mathcal{P}}$ such that $\{holds(a) \mid a \in X\} \cup Q$ is satisfiable, and it has a saturated stable model (including bot) if there is no such X.
- For computing a solution to the GT problem, one just has to add the generator program P, map the atoms of P to their names in the positive disjunctive logic program, and inforce the atom bot
$$P \cup \mathcal{R}\big(Q \cup \{\{holds'(a)\} \mid a \in \mathcal{A}_{\mathcal{P}}\}\big) \cup \mathcal{M} \cup$$
$$\{holds(a) \leftarrow a \mid a \in X\} \cup \{not\ holds(a) \leftarrow not\ a \mid a \in X\} \cup \{\leftarrow\ not\ bot\}\ .$$

[12]

– Deciding whether there is a solution to a GC problem is Σ_2^p-complete. Membership comes from the translation to disjunctive logic programming, and hardness comes from the translation from quantified boolean CNF formulas.

[12] JR: The rules generating $holds(a)$ and $not\ holds(a)$ are not exactly like that, I have to go again through it.

– Differences with [14]: [13]
  - 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. i
  - 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

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

In this section, we implement Eiter and Polleres framework with the metaencoding and reification of metasp. [14]

[14] 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. . .

6 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.*

[15]

[15] JR: Posed as a model finding problem

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.*

Methods:

- (From Y. Zhu and M. Truszczyinski, LPNMR 2013) Enumerate optimal models until one contains q.
- (From Y. Zhu and M. Truszczyinski, LPNMR 2013) Enumerate possibly nonoptimal models containing q, and test each one for optimality.
- Enumerate optimal stable models of $P \cup \{\bot \leftarrow \; not\ q\}$, testing each for optimality on P.
  TO BE ADDED: Justification of the algorithm.
  1. Find an optimal model X of $P \cup \{\bot \leftarrow \; not\ q\}$. If none exists, return $false$, else goto 2.
  2. Find a stable model Y of $P \cup \{\bot \leftarrow \; q\}$ better than X. If none exists, return $true$. If one exists, optionally Y can be further improved until an optimal stable model of P is produced. Add to P rules deleting the best stable model generated, and all stable models worse than it. Goto 1.
- Find a stable model with query, then another better without query, then another better with query…
  TO BE ADDED: Justification of the algorithm.
  1. Find an stable model X of $P \cup \{\bot \leftarrow \; not\ q\}$. If none exists, return $false$, else goto 2.
  2. Find a stable model Y of $P \cup \{\bot \leftarrow \; q\}$ that is better than X. If none exists, return $true$, else goto 3. Optionally, if none exists, X can be improved until an optimal model of P is obtained.
  3. Find an stable model X of $P \cup \{\bot \leftarrow \; not\ q\}$ that is better than Y. If one exists, goto 2. If none exists, optionally, Y can be improved until an optimal model of P is obtained. Add to P rules deleting the best stable model generated and all stable models worse than it. Goto 1.

7 Preferences over optimal models in *asprin*

[16] JR: Best title so far…

[16]

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 `s` is a preference statement. [17]

[17] JR: This is not implemented yet! And `reoptimize` is just a first try as a name ;)

[18]Given a program P, define $q(P)$ as the program

$$(P \setminus \{r \in P \mid head(r) = \emptyset\}) \cup \{u \leftarrow body(r) \mid r \in P, head(r) = \emptyset\} \cup \{\, q \leftarrow \; not\ u\}$$

[18] JR: Copy, paste and modify from Draft on Preferences

where u and q are new atoms.

Proposition 1. *If program P is stratified, P is satisfiable iff $q \in X$, where X is the stable model of $q(P)$.*

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 ← solveOpt(P, s);
if Y = ⊥ then return ⊥;
repeat
    X ← Y;
    Y ← solveOptq(P ∪ q(E_{t_t} ∪ F_t ∪ R_A ∪ holds'(X)), q) ∩ A;
until Y = ⊥;
return X
```

8 Complete methods

8.1 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: [19]

[19] 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.

8.2 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
}.
```

[20]

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

– This method does not work if P is disjunctive.

9 Approximation

[21] 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.

9.1 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).

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

- [4]

[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.

- 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 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).

9.3 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.

9.4 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.

10 Experiments

11 Discussion

References

1. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)

2. Andres, B., Gebser, M., Glaß, M., Haubelt, C., Reimann, F., Schaub, T.: Symbolic system synthesis using answer set programming. [16] 79–91
3. Banbara, M., Soh, T., Tamura, N., Inoue, K., Schaub, T.: Answer set programming as a modeling language for course timetabling. Theory and Practice of Logic Programming **13**(4-5) (2013) 783–798
4. Eiter, T., Erdem, E., Erdogan, H., Fink, M.: Finding similar/diverse solutions in answer set programming. Theory and Practice of Logic Programming **13**(3) (2013) 303–359
5. Zhu, Y., Truszczyński, M.: On optimal solutions of answer set optimization problems. [16] 556–568
6. Brewka, G., Niemelä, I., Truszczyński, M.: Answer set optimization. In Gottlob, G., Walsh, T., eds.: Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI'03), Morgan Kaufmann Publishers (2003) 867–872
7. Nadel, A.: Generating diverse solutions in SAT. In Sakallah, K., Simon, L., eds.: Proceedings of the Fourteenth International Conference on Theory and Applications of Satisfiability Testing (SAT'11). Volume 6695 of Lecture Notes in Computer Science., Springer-Verlag (2011) 287–301
8. Hebrard, E., Hnich, B., O'Sullivan, B., Walsh, T.: Finding diverse and similar solutions in constraint programming. In Veloso, M., Kambhampati, S., eds.: Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI'05), AAAI Press (2005) 372–377
9. Brewka, G., Delgrande, J., Romero, J., Schaub, T.: asprin: Customizing answer set preferences without a headache. In Bonet, B., Koenig, S., eds.: Proceedings of the Twenty-Ninth National Conference on Artificial Intelligence (AAAI'15), AAAI Press (2015) 1467–1474
10. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing **9** (1991) 365–385
11. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence **138**(1-2) (2002) 181–234
12. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: *Clingo* = ASP + control: Preliminary report. In Leuschel, M., Schrijvers, T., eds.: 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.
13. Gebser, M., Kaufmann, B., Otero, R., Romero, J., Schaub, T., Wanko, P.: Domain-specific heuristics in answer set programming. In desJardins, M., Littman, M., eds.: Proceedings of the Twenty-Seventh National Conference on Artificial Intelligence (AAAI'13), AAAI Press (2013) 350–356
14. Eiter, T., Polleres, A.: 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) (2006) 23–60
15. Gebser, M., Kaminski, R., Schaub, T.: Complex optimization in answer set programming. Theory and Practice of Logic Programming **11**(4-5) (2011) 821–839
16. Cabalar, P., Son, T., eds.: Proceedings of the Twelfth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'13). In Cabalar, P., Son, T., eds.: 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)

This article was processed using the comments style on April 6, 2016.
There remain 21 comments to be processed.