Differentiable SAT/ASP

Matthias Nickles

National University of Ireland, Galway
School of Engineering and Informatics
matthias.nickles@nuigalway.ie

Abstract. We propose Differentiable SAT and Differentiable Answer Set Programming for multi-model optimization through gradient-controlled answer set or satisfying assignment computation. As a use case, we also show how our approach can be used for expressive probabilistic inference constrained by logical background knowledge. In addition to presenting an enhancement of the CDNL/CDCL algorithm as primary implementation approach, we introduce alternative algorithms which use an unmodified ASP solver and map the optimization task to conventional answer set optimization or use so-called propagators.

Keywords: SAT · ASP · Probabilistic Programming · Gradient Descent · Approximate Probabilistic Inference · Relational Artificial Intelligence

1 Introduction

Modern SAT and Answer Set solvers are, like their closely related cousins constraint processing and Satisfiability Modulo Theories (SMT), powerful and fast tools for logical reasoning. We present an approach which utilizes and enhances current SAT/ASP solving algorithms for multi-model optimization, with probabilistic inference as the focused - but not the only - use case. We build on previous work [12] and add new algorithms, in particular methods which require only an existing, unmodified ASP solver or map the task to a regular answer set optimization problem.
With *Multi-model optimization* we mean the search for a multi-set of models (satisfying Boolean assignments or answer sets) which indirectly represents an (approximate) minimum of a user-provided cost function. We consider cost functions defined over certain statistical properties of the model multi-set, namely frequencies of atoms (with cost functions over fact, rule, clause or model weights as straightforward instances), and we incrementally sample models until a cost minimum is reached, with partial derivatives of the cost function guiding the assignment of decision literals in the SAT or ASP solving process.
In the use case of probabilistic logic programming (a form of declarative probabilistic programming), the resulting multi-model optimum approximates a probability distribution over possible worlds (models) induced by probabilistic constraints (encoded as the cost function) and non-probabilistic rules and clauses (a regular ASP program or Boolean formula in CNF of which all sampled models are answer sets respectively satisfying assignments). By sampling only as many models as required for cost minimization, we reduce the number of expensive conventional deductive inference steps

CEUR-WS.org/Vol-2219/paper7.pdf

and avoid the combinatorial explosion of materialized possible worlds with increasing number of nondeterministic atoms which typically precludes the use of straightforward optimization techniques (such as linear programming) in probabilistic logic programming.
In principle, arbitrary differentiable cost function can be used (although obviously not all cost functions lead to convergence of the optimization process) and there are no restrictions on the background knowledge rules or clauses, or the random variables (such as independence assumptions), except consistency. The expressiveness of the framework is thus quite high, and, despite the use of sampling, computation times remain a challenge (in particular in comparison with approaches which deliberately put restrictions into place, such as frameworks based on the Distribution Semantics). We tackle this concern with our first concrete algorithm (a variant of the algorithm presented in [12]) by integrating the cost minimization steps directly into a state-of-the-art ASP/SAT solving approach [4], and we compare its performance experimentally with that of several alternative methods introduced in this paper.
The remainder of this paper is organized as follows: The following section introduces basic concepts and notation. Sect. 3 presents our general approach and proposes several concrete computation methods. Sect. 4 presents results from preliminary experiments, and Sect. 5 discusses related approaches. Sect. 6 concludes.

2 Preliminaries

We consider ground normal logic programs under stable model semantics and SAT problems in form of propositional formulas in Conjunctive Normal Form (CNF). Recall that a normal logic program is a finite set of rules of the form
$h :- b_1, ..., b_m,\ not\ b_{m+1}, ..., not\ b_n$ (with $0 \leq m \leq n$). h and the b_i are atoms without variables. not represents default negation. Rules without body literals are called facts. Most other syntactic constructs supported by contemporary ASP solvers (like integrity constraints, choice rules or classical negation) can be translated into (sets of) normal rules. We consider only ground programs in this work. The *answer sets* (*stable models*) of a normal logic program are as defined in [5]. Throughout the paper, we use the term "answer set program" to mean a ground normal logic program and "model" in the sense of answer set or, in the SAT case, a complete truth assignment such that the formula evaluates to true, however, to use the same model notation with both ASP and SAT, we generally do not show false atoms in models, i.e., a model is represented as a set of atoms which hold in the model. As common in probabilistic ASP, we identify *possible worlds* with models. Ψ_Υ denotes the set of all answer sets or satisfying assignments of answer set program or propositional formula Υ. Sometimes we use only logic programming terminology where the translation to SAT terminology is obvious (e.g., "program" instead of set of clauses). The set of all atoms respectively propositional variables in a program or formula Υ is denoted as $atoms(\Upsilon)$. We write $\overline{S}$ to denote a set of negative literals $\{\overline{s_i} : s_i \in S\}$.
A *partial assignment* denotes an incomplete "model under construction": a sequence of literals which have been iteratively added (assigned) to the assignment (and sometimes retracted from the assignment in backtracking steps) by the SAT or ASP solver until the

assignment is complete or the procedure aborts in case of unsatisfiability.
We use a unified approach to SAT and ASP solving based on *nogoods* [4] (corresponding to clauses in CNF, but with all literals negated; a concept originally introduced for constraint solving). Clauses and rules are translated into nogoods in a preprocessing step (covering Clark's completion in the ASP case). Additional nogoods are learned from conflicts and loops (in non-tight ASP programs).

2.1 Cost Functions and Parameter Atoms

The cost functions considered in this work are user-provided functions of several variables. Each variable corresponds to a so-called *parameter atom*. When evaluating the cost function, we instantiate each variable with the normalized count (*frequency*) of the respective parameter atom in a possibly incomplete *sample*. The *sample* is a multi-set of models sampled with replacement from the complete set Ψ_Υ of answers sets of the given program, respectively the set of all satisfying assignments (SAT case). Where a parameter atom or its negation can occur we speak of *parameter literals*. Parameter atoms can occur in rules and clauses of the given answer set program or Boolean formula without limitations, and may even be conditioned on the truth values of other atoms, including other parameter atoms. The set of parameter atoms is denoted as θ or $\{\theta_i\}$, and θ_i is the i-th parameter atom under some arbitrary but fixed order. We denote the individual cost function variables corresponding to the parameter atoms as θ_i^v or a^v (where a is a parameter atom). The parameter atoms need to be chosen by the user from the overall set of atoms in the program or set of clauses (theoretically, all atoms could be declared parameter atoms, but this might be inefficient).
With each newly sampled model, the frequencies of the parameter atoms are updated as follows. $\beta(sample) = \langle \beta^{sample}(\theta_1), \beta^{sample}(\theta_2), ... \rangle$ is defined as the (parameter) frequencies vector $\langle \frac{|[m_j : m_j \in sample, \theta_1 \in m_j]|}{|sample|}, ..., \frac{|[m_j : m_j \in sample, \theta_n \in m_j]|}{|sample|} \rangle$ of parameter atom frequencies in the model multi-set $sample$. $\beta^{sample}(\theta_i)$ denotes the frequency of parameter atom θ_i in $sample$. We omit $sample$ and simply write $\beta(\theta_i)$ where it is clear from the context what the sample is. $cost(\beta(\theta_1), \beta(\theta_2), ...)$ evaluates the cost function $cost$ over parameter frequencies vector $\beta(sample)$. We sometimes write $cost(sample)$ in place of $cost(\beta(\theta_1), \beta(\theta_2), ...)$.
It makes sense to allow only parameter atoms whose truth values are not fully fixed by the input program or formula. To ensure this in the ASP case, we can give the logic program the shape of a so-called *spanning program* [13, 9] where uncertain atoms a are defined by *spanning formulas*: choice rules or analogous constructs amounting to nondeterministic facts $0\{a\}1$ or (informally) $a \vee not\ a$. However, our framework does not require any particular form of the input program or formula.
Informal examples for cost functions are "In 30% of all models, atom a should hold and in 40% of all models, atom b should hold" or "Atom a should hold more often than the square root of the frequency of atom b minus 10%". In principle, other types of cost functions which refer to other properties of the current sample multi-set are conceivable too, provided the model sampler is able to minimize such a cost.

2.2 Cost Functions and Parameter Atoms for Probabilistic Logic Programming

For the application case of deductive probabilistic inference, cost functions specify probabilistic constraints, whereas the plain ASP rules or SAT clauses serve as hard constraints. More concretely, we consider the case that the probabilistic constraints are provided by the user in form of *weights* associated with individual parameter atoms θ_i. Weights directly represent probabilities. Weights can also be attached to arbitrary rules, by introducing weighted fresh auxiliary atoms as "shortcuts" for these rules, using the following scheme: $h :- b_1, ..., b_n,\ not\ aux$ and $aux :- b_1, ..., b_n,\ not\ h$ [9].
As one (but not the only) suitable cost function which can be derived directly from a set of weighted parameter atoms, we propose the use of the *Mean Squared Error* (MSE) $cost(\theta_1^v, \theta_2^v, ...) := \frac{1}{n}\sum_{i=1}^{n}(\beta(\theta_i) - \phi_i)^2$ (squared Euclidean distance normalized with the number n of parameter atoms). The ϕ_i are the user-defined weights of the parameter atoms θ_i.
With this cost function, appending models to the sample until the cost reaches zero (i.e., maximum accuracy) corresponds to finding a solution of a linear equation system with additional conditions to ensure that the solutions form a probability distribution, with the probabilities of *all* possible worlds Ψ_Υ as the unknowns and the actual model frequencies in $sample$ as approximate solution vector (details are provided in [12]). Queries can then be performed as usual, by adding up the approximated probabilities of those possible worlds where the query holds (that is, the frequencies of those models in $sample$ which positively contain the query atoms).
As an example for how to assign probabilities to atoms using cost functions, consider function $cost(a^v, b^v) = ((0.2 - a^v)^2 + (0.6 - b^v)^2)/2)$ which specifies that the weight of parameter atom a should be 0.2 and the weight of parameter atoms b should be 0.6.
As another example, this time for a cost function different from MSE (but also solvable using Differentiable ASP), consider $cost(aux^v, q^v) = (0.4 - aux^v/q^v)^2$ which specifies that the conditional probability $Pr(p|q)$ is 0.4. The accompanying answer set program needs to include rules $aux :- p, q$ and $p :- aux$ and $q :- aux$.

3 Differentiable SAT/ASP

To find a sample of models which minimizes the cost function, our approach iteratively appends models to multi-set $sample$ until the cost falls below a specified threshold (allowing the user to trade speed against accuracy). All models are answer sets or satisfying truth assignment of the given answer set program or Boolean formula, ensuring that they adhere to the given "hard" logical constraints. Partial cost function derivatives guide, during the generation of each individual model, the SAT/ASP solving process on the level of branching decisions, namely truth value assignments to parameter atoms.

Fig. 1(a) shows a high level view of this approach, named *Differentiable SAT/ASP* or ∂SAT/ASP for short. An outer loop (left side of Fig. 1(a)) samples models and adds them to an initially empty multi-set $sample$ until the termination criterion is reached (there are several specific possibility for checking termination, depending on the nature of the cost function and the use case: if the cost expression is not too large, we can check if it is equal or below a given threshold ψ (accuracy), and/or we could perform a stagnation check on the cost or the parameter atom frequencies. In some applications

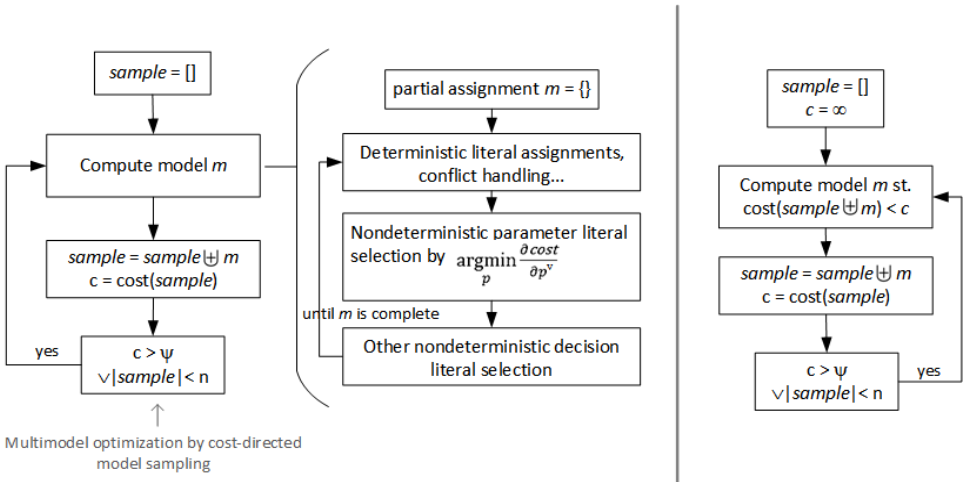


Fig. 1: Differentiable SAT/ASP (outline)

it might also be sensible to demand a minimum sample size n in addition to reaching threshold ψ, e.g., to increase the sample entropy). In our experiments, we simply stopped sampling when the cost reached or fell below ψ.

The models are sampled with the aim of reducing the multi-model cost, using a form of discretized gradient descent. We approach this with a special branching rule for selecting decision literals (literals not enforced by propagation) for inclusion in the partial assignment: Each time the solver nondeterministically (i.e., not forced by rules or clauses) extends the partial assignment with a not yet assigned literal, the literal is selected from all unassigned parameter literals (if any) if the value (or its negation in case of negative literals) of the cost function's partial derivative with respect to this literal is minimal (compared with the values obtained for the other unassigned parameter literals). Since the parameter search space is discrete, we could theoretically measure the cost impacts of hypothetically assignments of candidate literals directly. But taking the partial derivatives with respect to the parameter atoms splits the overall cost calculation (which might be complex) into typically simpler calculations whose results can even be pre-computed and ranked after each new model according to their current values, after updating the frequencies vector β with the new model. Finally, for branching decisions on non-parameter decision literals, some conventional branching heuristics (e.g., VSIDS) can be used.

Fig. 1(b) outlines a variant where each new model is explicitly required to lower the cost, without specifying how to achieve this. A specific approach to this variant, proposed in [12], uses a so-called *cost backtracking* mechanism which in case a model candidate fails to improve the cost, jumps back to the most recent literal decision point with untried parameter literals and tries a new parameter literal. [12] also indicates how cost backtracking and a split of the set of parameter atoms into *measured atoms* and actual parameter atoms can be utilized for inductive weight learning and abductive reasoning (which are not possible using the approach in Fig. 1(a)).

In the following subsections, we propose concrete approaches to put the general approach in Fig. 1(a) into practice.

3.1 Implementing Differentiable SAT/ASP based on CDNL

A concrete approach to the rather general scheme described above is to enhance the current state-of-the-art approach to Answer Set Programming (Conflict-Driven Nogood Learning (CDNL) [4]) or a similar approach (CDCL (Conflict-Driven Clause Learning)- or DPLL-style solving) with a new branching rule which selects free parameter literals for inclusion into the current partial assignment according to their negative "impact" on the cost, determined using partial derivatives wrt. parameter atoms. This approach, which we call *Diff-CDNL-ASP/SAT* (as it covers both SAT and Answer Set solving), is shown as Algo. 1. The SAT solving path through the algorithm, enabled by parameter $SATmode$, is largely identical to the more complex ASP path, with stable model checks and loop handling omitted. The algorithm is a variant of the approach presented in [12], with a somewhat more general branching approach using partial derivatives but omitting the optional cost backtracking.

Algo. 1 iteratively adds literals to a partial assignment until all literals are covered and no *nogood* is violated. Important steps are unit propagation, i.e., the deterministic expansion of the partial assignment (procedure PROPAGATE) with literals "fired" by nogoods, and conflict analysis (conflict means at least one nogood is subset of the partial assignment) and handling, including the deriving of further nogoods from the analysis of conflicts and backjumping (undoing recent literal assignments) in line 21 (details follow [4] and are omitted here for lack of space).

The procedure for generating a single model (the while-loop from line 6) is thus guided by the following factors: 1) the initial set of nogoods obtained from the CNF clauses or Clark's completion of the answer set program, 2) further nogoods added to this set by conflict handling or due to the presence of loops (ASP mode), 3) the given cost function and set of parameter atoms, and 4) our new branching approach for assigning parameter literals (lines 9 to 13). The inner while loop ends once all atoms are covered as positive or negative literals (or UNSAT). Afterwards (line 25), the stable model check takes place (unless in SAT mode), and the new model is appended to the multi-set $sample$. The outer loop (from line 3) ends when a convergence criterion is met (e.g., when the cost falls below the given accuracy threshold ψ (line 32) and we have obtained the requested number of models, or if there is no more progress).

The decision branching rule is the main different to regular CDNL: In lines 11ff., we select the next parameter literal according to the previously described approach using partial derivatives. At this, it was in all our initial tests sufficient for reaching convergence to fix a new ranking of parameter literals by the values of the respective partial derivatives only after a new model was generated (ignoring the current partial assignment in the computation of the parameter atom frequencies) and to use this ranking to determine the next decision literal in line 11.

For further details on the non-probabilistic aspects of the algorithm, we need to refer to [4] for lack of space. Note that for loop handling, Algo. 1 uses the older and simpler ASSAT approach [10], just to simplify our initial implementation.

3.2 Differentiable ASP using propagators (*Diff-ASP-ThProp*)

While the approach from Sect. 3.1 is quite fast (see Sect. 4), it has the shortcoming that it cannot be realized using an unmodified existing ASP or SAT solver. As an alternative

Algorithm 1 Diff-CDNL-ASP/SAT

Arguments: Υ (program or formula), $nogoods(\Upsilon)$, ψ (accuracy), θ, $cost$, $SATmode$, n (minimum number of models sampled with specified accuracy, 0 in our experiments)
$sample \leftarrow [\,]$
repeat ▷ Outer loop (enhances multi-set $sample$)
  $as \leftarrow$ PROPAGATE$(\{\}, nogoods(\Upsilon))$ ▷ Unit propagation (enhancing as with unit-resulting literals until fixpoint).
  $dl \leftarrow 0$ ▷ Decision level dl initially 0 (no nondeterministic decisions made yet)
  while $incomplete(as) \vee conflicting(as)$ **do** ▷ Inner loop (computes a single model)
    if $\neg conflicting(as)$ **then** ▷ Branching...
      $dl \leftarrow dl + 1$
      $ua\theta \leftarrow (\theta \cup \overline{\theta}) \backslash as$ ▷ unassigned parameter literals
      if $ua\theta \neq \{\}$ **then**
        $decLit \leftarrow \text{argmin}_{p \in ua\theta}\ sg\ \frac{\partial cost}{\partial pa^v}(\beta^{sample}(\theta_1), \beta^{sample}(\theta_2), ...)$,
        with $(pa, sg) = \begin{cases} (p, 1)\ if\ p \in atoms(\Upsilon) \\ (\overline{p}, -1)\ otherwise \end{cases}$
        $decLitPr \leftarrow 1 - noise$ ▷ $noise \ll 1$
      else
        $(decLit, decLitPr) \leftarrow ...$ (using some conventional branching heuristics)
      end if
      if $rand_0^1 < decLitPr$ **then** $as \leftarrow as \cup decLit$ **else** $as \leftarrow as \cup \overline{decLit}$
      $as \leftarrow$ PROPAGATE$(as, nogoods(\Upsilon))$
    end if
    if $conflicting(as)$ **then**
      ConflictHandling, back jumping (or stop with UNSAT if irrecoverable ($dl = 0$))
    end if
  end while
  $modelCand \leftarrow$ positive literals in as ▷ (see Sect. 2)
  if $SATmode \vee tight(\Upsilon) \vee stable(modelCand)$ **then**
    $sample \leftarrow sample \uplus \{modelCand\}$
  else
    if $\neg SATmode$ **then**
      add subset of loop nogoods ▷ (or some other approach to non-tight programs)
    end if
  end if
until $cost(\beta^{sample}(\theta_1), \beta^{sample}(\theta_2), ...) \leq \psi \wedge |sample| \geq n$ (or until cost stagnates)

approach to ∂SAT/ASP, we can make use of Clingo's[1] *propagators*. We show how to do this using preliminary code[2] (file `propdiff_1.py.lp`), instantiated with an MSE-shaped example cost function and two parameter atoms a (with given probability 0.6) and b (probability 0.2). It requires only Clingo (tested with version 5.2) with Clingo's Python scripting interface and Python 2.7.
While custom propagators cannot directly implement a branching heuristics, they can be used to add (parameter) literals in form of singleton clauses to the ongoing solving

[1] `https://github.com/potassco/clingo`
[2] `https://github.com/MatthiasNickles/Diff-ASP-Propagators`

process, intercepting propagation. We compute the parameter literal we would like to assign next (again using the approach in Sect. 3) and then pass this literal (`branch_param_lit`) on to the propagator (lines 56ff. in the code) to add it to the partial assignment.
The rest of the Python code is straightforward: The loop from line 155 corresponds to the outer loop in Algo. 1: it iteratively calls the solver to compute new models, adds each newly sampled model to the model list `sample` (in callback `on_model`), updates frequencies and evaluates the cost (method `update_cost`), and checks for convergence against threshold `psi`.
The code supports both numerical and automatic differentiation (the latter using the *ad* package[3]). In both cases, the search for the parameter literal which gives the minimum partial derivative (i.e., steepest descent) is performed in lines 124ff. Automatic differentiation wrt. parameter atoms takes place in method `__cost_ad`. For numerical differentiation, we use a simple approximation which just adds a small value `h` to the frequency of each respective parameter atom to estimate the slope (method`__cost_upd`).
The actual cost function (including the given weights of the two parameter atoms) is in line 84 (we use the expression format of the ad-package for Python to represent the cost expression, also with numerical differentiation).
(Remark: We have also experimented with domain heuristics using Clingo's designated predicate `_heuristic/3`, but found no way yet to make this reliably working for our use case. Also, this attempt appears to be much slower than all other approaches presented in this paper.)
An example for a simple associated background theory (file `propdiff_bgk_1.lp`) is

```
0{a}1. % spanning rule for parameter atom a
0{b}1.
:- a, b. % an example for a hard rule
```

The overall program is called with `clingo-python propdiff_1.py.lp propdiff_bgk_1.lp`

3.3 Direct cost minimization using model reification (*Diff-ASP-Reification*)

As the final approach proposed in this paper, we use Clingo with reified predicates and models to solve the cost function directly (without derivatives involved), alternatively by mapping it to a conventional single answer set optimization task or by mapping the problem to ASP-encoded equation solving.
Here, each predicate in the original answer set program (not only the parameter atoms) whose truth value is not fully fixed is enhanced with a model number as extra argument. This way, we can let a single actual model (returned by Clingo) represent multiple reified models where each reified model consists exactly of those atoms in the actual model which share the same model number as their extra argument.
We distinguish two flavors of this approach: 1) Computation of one or more individually optimal answer set(s) using optimization statements or weak constraints (...:∼...), as supported by several ASP solvers, including smodels, DLV and Clingo/clasp, and 2) using ASP directly for constrained linear equations solving.
We consider variant 1) first, shown in the code below for the same example as before (two uncertain atoms a and b with weights 0.2 and 0.6), and MSE as cost function format (adaptations to some other types of cost function should be straightforward). The number of reified models `nmodels` does not need to be known precisely but should be at least 10^n where n is the number of decimal places which should be accurate

[3] `https://pypi.org/project/ad/`

when using the overall result to query $Pr(a)$ and $Pr(b)$. We found this approach very slow in our preliminary experiments with default settings (more efficient encodings or optimization strategies might exist), but in any case it is useful to exemplify how our multi-model optimization task can be mapped to conventional answer set optimization using reification.

```
#const nmodels = 10.
model(1..nmodels).
mcount(0..nmodels).
{a(M)} :- model(M).  % spanning formulas
{b(M)} :- model(M).
:- a(M), b(M), model(M). % an example for a background knowledge rule (hard constraint)

wa(nmodels * 2 / 10).   % weight a = 0.2
wb(nmodels * 6 / 10).  % weight b = 0.6
fa(F) :- F { a(M): model(M) } F, mcount(F).
fb(F) :- F { b(M): model(M) } F, mcount(F).

diffa(D) :- D = (W - F)**2, wa(W), fa(F). % alternatively: D = |F - W|
diffb(D) :- D = (W - F)**2, wb(W), fb(F).
#minimize { DA : diffa(DA) }.  %  minimize the distances betw. weights and frequencies
#minimize { DB : diffb(DB) }.

#show a/1.
#show b/1.
```

In variant 2) of our reification-based approach, we map the problem to a set of linear equations (or inequalities, if error tolerance bounds are considered) as outlined in Sect. 2.2, and encode it as a plain answer set program. It is immediately clear that here the number of reified models introduces a bottleneck: Every predicate whose truth value is not fully fixed across all reified models needs to be reified, which multiplies the number of its instances with the overall number of reified models (`nmodels` in the code below). Nevertheless, as detailed in the next section, this simple approach fares surprisingly well for relatively small problem sizes, and significantly better than the approach using propagation (Sect. 3.2). The following plain ASP program shows how to implement this for the example problem above.

```
#const tol = 3. % NB: tol has a different semantics than \psi
#const multiplier = 100.  % to map float numbers to integers; limits  precision
#const nmodels = 400.
model(1..nmodels).

wa(nmodels * 2 * multiplier / (10 * multiplier)).  % 2 represents given weight 0.2
W-tol < { a(M): model(M) } < W+tol :- wa(W).
wb(nmodels * 6 * multiplier / (10 * multiplier)).  % 6 represents given weight 0.6
W-tol < { b(M): model(M) } < W+tol :- wb(W).

1{__aux_1(M);a(M)}1 :- model(M).  % spanning formulas
1{__aux_2(M);b(M)}1 :- model(M).

:- a(M), b(M).  % example for a hard background knowledge rule

#show a/1.
#show b/1.
```

4 Preliminary Experiments

To provide initial insight into the performance characteristics of the presented approaches, we have performed two preliminary experiments. Approaches considered were Diff-CDNL-ASP/SAT (Sect. 3.1), Diff-ASP-ThProp (Sect. 3.2) and Diff-ASP-Reification

(Sect. 3.3), the latter in the equation solving variant (the version using `#minimize` proved too slow with these experiments to be considered).

We performed two synthetic experiments (Figs. 2a and 2b): a coin tossing game with background rules and a variant of the well-known "Friends & Smokers" scenario (which exists in several variants in the literature). Times have been averaged over three trials per experiment on a i7-4810MQ machine (4 cores, 2.8GHz) with 16GB RAM. The Diff-CDNL variant of the ∂SAT/ASP approach (Sect. 3.1) has been experimentally implemented in Scala 2.12 (i.e., running in a Java VM). Experiments have been performed with different accuracy thresholds ψ. All times are end-to-end times, so they include some overhead for file operations, parsing, etc. For the Clingo-based tasks we have used Clingo 5.2.2 with Python 2.7.10 for scripting. The tolerance 20 specified with the reification-based tasks with 400 models corresponds roughly to accuracy $\psi \approx .001$ for coins and $\psi \approx 0.05$ for smokers.

In the coin game, a number of coins are tossed and the game is won if a certain subset of all coins comes up with "heads". The inference task is the approximation of the winning probability, calculated by counting the models with the winning coin combinations within the resulting sample and normalizing this count with the size of the sample. In addition, another random subset of coins are magically dependent from each other and one of the coins is biased (probability of "heads" is 0.6). This scenario contains probabilistically independent as well as mutually dependent uncertain facts. Also, inference difficulty clearly scales with the number of coins. In pseudo-syntax, such a randomly generated program looks, e.g., as follows:

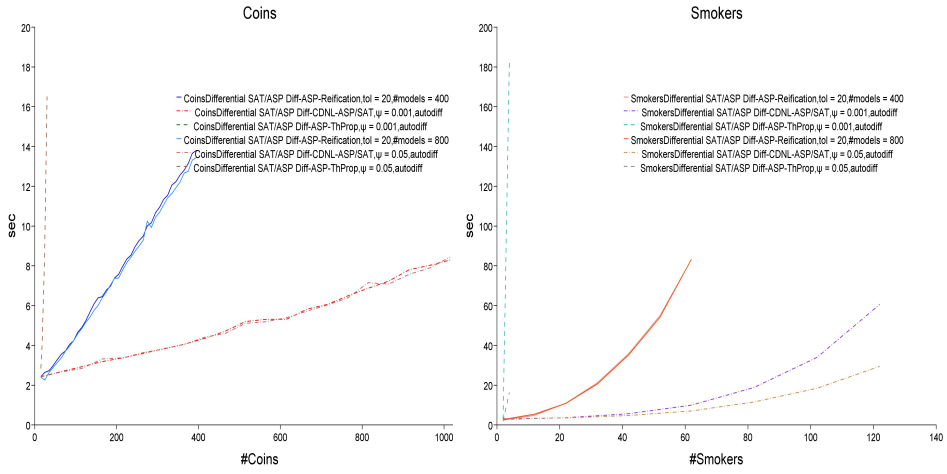
```
coin(1..8).
0.6: coin_out(1,heads).
0.5: coin_out(N,heads) :- coin(N), N != 1.
1{coin_out(N,heads), coin_out(N,tails)}1 :- coin(N).
win :- 2{coin_out(3,heads),coin_out(4,heads)}2.
coin_out(4,heads) :- coin_out(6,heads).
```

The proposed methods cannot directly work with non-ground rules, so weighted non-ground rules anywhere have been translated into sets of ground rules, each (respectively, their corresponding auxiliary "shortcut" parameter atoms) annotated with the respective weights.

In "Friends & Smokers", a randomly chosen number of persons are friends, a randomly chosen subset of all people smoke, there is a certain probability for being stressed (`0.3: stress(X)`), it is assumed that stress leads to smoking (`smokes(X) :- stress(X)`), and that friends influence each other with a certain probability (`0.2: influences(X,Y)`), in particular with regard to smoking: `smokes(X) :- friend(X,Y), influences(Y,X), smokes(Y).` Smoking may leads to asthma (`0.4: h(X). asthma(X) :- smokes(X), h(X)`). The query atoms are `asthma(X)` per each person X.

For both experiments, it should be kept in mind that ∂SAT/ASP does not include any special treatment of probabilistic independence.

While the solution using Clingo with reified models cannot compete with the native approach Diff-CDNL-ASP/SAT, it is still surprisingly fast, which, together with the fact that there is virtually no difference between the 400 and 800 model variants (the curves almost cover each other), exemplifies the strong solving performance of Clingo. However, these results also seem to indicate (since both Clingo/clasp and Diff-CDNL-ASP/SAT are based on CDNL) that the prototypical Diff-CDNL-ASP/SAT implementation could be made significantly faster by further optimizing its code or by using, e.g., C/C++ or Rust. The alternative approach using propagators is clearly usable only for tiny problems, at least with the current implementation (the graph for Diff-ASP-ThProp with $\psi = 0.001$ is not visible in Fig. 2a because its task immediately timed out).



(a) Coin Game

(b) Friends & Smokers

Fig. 2: Preliminary performance results

5 Related Work

[2] proposes distribution-aware sampling for SAT with weight functions over entire models (which might be seen as an instance of our MSE-style variant of cost optimization, albeit technically approached very differently). Also related is the weighted satisfiability problem which aims at maximizing the sum of the given weights of satisfied clauses (e.g., using MaxWalkSAT [8]), which can be used for inference in Bayesian networks. PSAT (Probabilistic Boolean Satisfiability) [16] and SSAT problem [14] tackle related but different problems compared to ours. The envisaged main application case for our multi-model sampling approach is probabilistic logic (programming) and probabilistic deductive databases, in particular Probabilistic ASP (of which existing approaches include, e.g., [1, 11, 9, 13]), but we expect other uses cases too, such as working with combinatorial and search problems with uncertain facts and rules. In the context of nonmonotonic Probabilistic Inductive Logic Programming, gradient descent has been used for weight estimation. E.g., [3] uses gradient descent to estimate the probabilities of abducibles. In contrast to the common MC-SAT sampling approach to inference with Markov Logic Networks (MLNs) [15] (a form of slice sampling), our task has a different semantics and our sampler is not wrapped around a uniform sampler, and we allow to specify rule or clause probabilities directly. An interesting approach with combines MLN with logic programming under the stable model semantics and compiles programs directly into ASP (using weak constraints) is [9]. Other than our approach, existing approaches to machine learning-based SAT solving (such as Learning Rate Branching Heuristic (LRB)), or the combination of gradient descent / neural network-based techniques or numerical optimization with SAT (such as [17]) aim at an improvement of SAT solving itself (which is not our concern in this work), or enable single model optimization.

6 Conclusion

We have presented differentiation-based approaches to SAT and ASP multi-model computation which sample models in order to minimizes a custom cost function. Using customized cost functions and parameter atoms, our overall approach can be instantiated in order to provide a tool for probabilistic logic programming. Building on existing work [12], we have presented an approach using a steepest descent method, an algorithm which utilizes Clingo's propagators, and finally an approach (not differentiation-based) which maps the problem to plain answer set optimization. Planned work comprises further experiments and the determination of formal convergence criteria.

References

1. Baral, C., Gelfond, M., Rushton, N.: Probabilistic reasoning with answer sets. Theory Pract. Log. Program. **9**(1), 57–144 (2009)
2. Chakraborty, S., Fremont, D.J., Meel, K.S., Seshia, S.A., Vardi, M.Y.: Distribution-aware sampling and weighted model counting for SAT. In: Procs. 28th AAAI Conference on Artificial Intelligence (AAAI). pp. 1722–1730 (July 2014)
3. Corapi, D., Sykes, D., Inoue, K., Russo, A.: Probabilistic rule learning in nonmonotonic domains. In: Computational Logic in Multi-Agent Systems. pp. 243–258. Springer (2011)
4. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In: Procs. of the 20th Int'l Joint Conf. on Artificial Intelligence (IJCAI'07) (2007)
5. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Proc. of the 5th Int'l Conference on Logic Programming. vol. 161 (1988)
6. Gomes, C.P., Sabharwal, A., Selman, B.: Near-uniform sampling of combinatorial spaces using xor constraints. In: NIPS. pp. 481–488 (2006)
7. Gressler, A., Oetsch, J., Tompits, H.: Harvey: A system for random testing in asp. In: Logic Programming and Nonmonotonic Reasoning. Springer (2017)
8. Kautz, H., Selman, B., Jiang, Y.: A general stochastic approach to solving problems with hard and soft constraints. In: The Satisfiability Problem: Theory and Applications. pp. 573–586. American Mathematical Society (1996)
9. Lee, J., Talsania, S., Wang, Y.: Computing lp using asp and mln solvers. Theory and Practice of Logic Programming **17**(5-6), 942–960 (2017)
10. Lin, F., Zhao, Y.: Assat: computing answer sets of a logic program by sat solvers. Artificial Intelligence **157**(1), 115 – 137 (2004), nonmonotonic Reasoning
11. Ng, R.T., Subrahmanian, V.S.: Stable semantics for probabilistic deductive databases. Inf. Comput. **110**(1), 42–83 (1994)
12. Nickles, M.: Sampling-based sat/asp multi-model optimization as a framework for probabilistic inference. In: Procs. 28th International Conference on Inductive Logic Programming (ILP'18). Springer (2018 (to appear))
13. Nickles, M., Mileo, A.: A hybrid approach to inference in probabilistic non-monotonic logic programming. In: 2nd Int'l Workshop on Probabilistic Logic Programming (PLP'15) (2015)
14. Papadimitriou, C.H.: Games against nature. J. Comput. Syst. Sci. **31**(2), 288–301 (1985)
15. Poon, H., Domingos, P.: Sound and efficient inference with probabilistic and deterministic dependencies. In: Procs. of AAAI'06. pp. 458–463. AAAI Press (2006)
16. Pretolani, D.: Probability logic and optimization sat: The psat and cpa models. Annals of Mathematics and Artificial Intelligence **43**(1), 211–221 (Jan 2005)
17. Selsam, D., Lamm, M., Bünz, B., Liang, P., de Moura, L., Dill, D.L.: Learning a SAT solver from single-bit supervision. CoRR **abs/1802.03685** (2018)