

Differentiable SAT/ASP

MATTHIAS NICKLES

SCHOOL OF ENGINEERING & INFORMATICS

NATIONAL UNIVERSITY OF IRELAND, GALWAY

Overview

- Introduction
- SAT and Answer Set Programming
- Approach outline
- Cost functions and parameter atoms for PLP
- Native algorithm
- Propagator-based approach
- Mapping to conventional Answer Set optimization
- Preliminary results
- Conclusion

Introduction (1)

- Modern SAT and ASP solvers are mature and fast inference tools
- Geared towards complex search, combinatorial and optimization problems (ASP & SAT)
- Strong foothold in industry (SAT)
- Rich, Prolog-like syntax but fully declarative (ASP)
- Non-monotonic reasoning (ASP)
- Similar solving techniques, ASP solving \approx SAT solving + loop handling
- Closely related to Satisfiability Modulo Theories (SMT) and Constraint Programming
- First-Order logic syntax, action logics, event calculus, ... can be translated to ASP

Introduction (2)

- ASP/SAT solving is a *multi-model* approach to inference
- Solving can produce some or all models as witnesses (if input is satisfiable)
- Stable models (a.k.a. answer sets) or satisfying Boolean assignments (SAT)
- Multiple alternative models as a natural way to express non-determinism
- Models as a natural way to represent possible worlds (as for PLP)

Introduction (3)

- How can we utilize SAT/ASP solving to compute not just models but also probability distributions over models?
- We could then use these distributions directly for probabilistic inference tasks
- Idea: distribution finding as a multi-model optimization task, using a suitable cost function over multiple models
- Generalized to (in principle) arbitrary differentiable multi-model cost functions
- Various solving techniques, outlined in this talk (further methods certainly exist)
- Focus on *gradient*-based approaches
- Also, we use *sampling*, for higher efficiency: a *sample* (a multi-set of models in our case) represents an approximate solution of the cost function

Input logic language

- SAT: we assume DIMACS-CNF input (set of clauses)
- ASP: Ground *Answer Set program* consisting of a finite set of *normal* rules:
 $a \text{ :- } b_1, \dots, b_k, \text{ not } b_{k+1}, \dots, \text{ not } b_m.$
- Example for an Answer Set program (before grounding, i.e., instantiating $X=\text{dilbert}$):

```
man(dilbert).
```

```
single(X) :- man(X), not husband(X).
```

```
husband(X) :- man(X), not single(X).
```

This program has two so-called *stable models* ("answer sets") – the *possible worlds*:

```
Sm1 = { man(dilbert), single(dilbert) }
```

```
Sm2 = { man(dilbert), husband(dilbert) }
```

Approach outline (1)

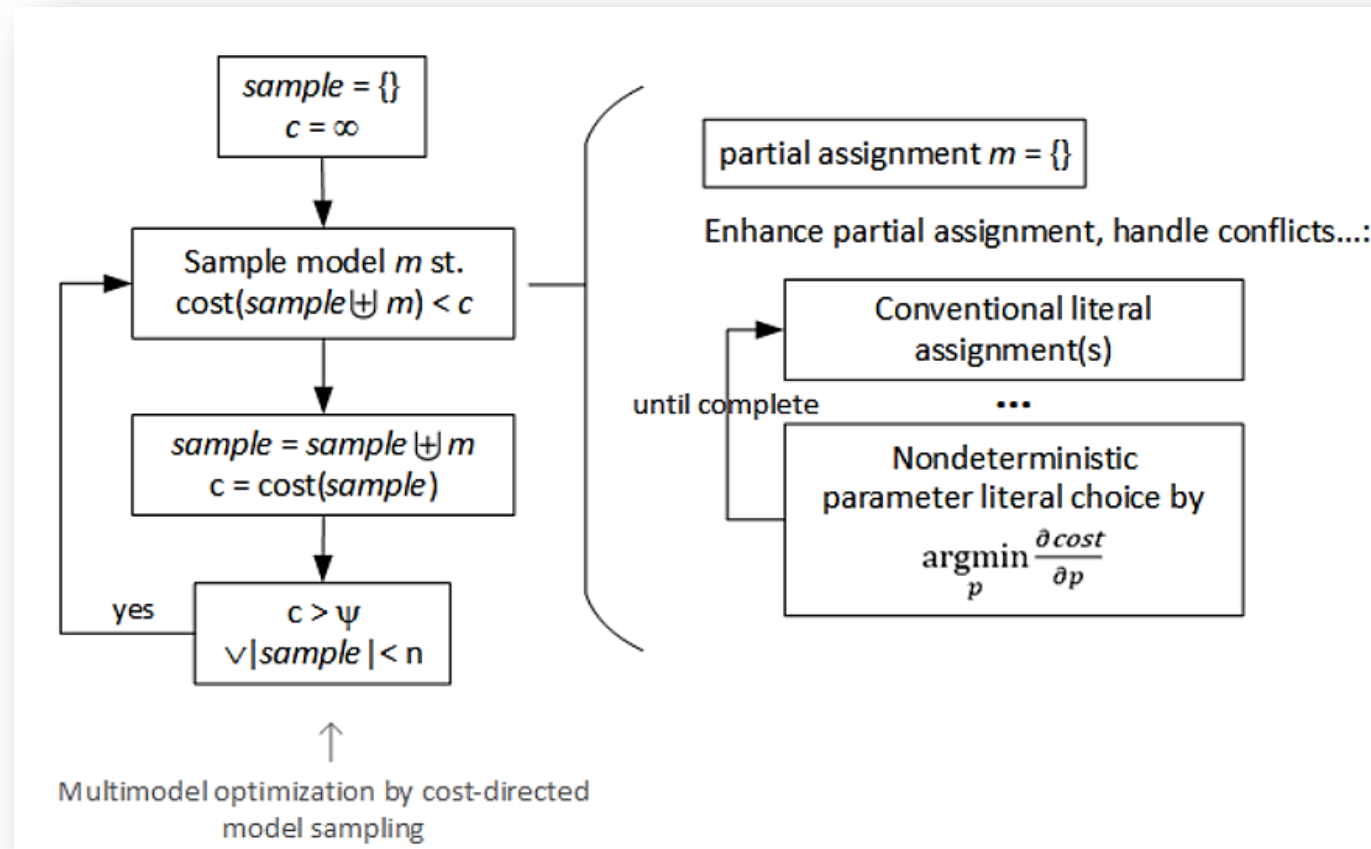
- Besides CNF-clauses or an Answer Set program, we require
 - a user-specified *cost function*
 - a user-specified set of *parameter atoms*
- Parameter atoms: subset of all atoms/variables which serve as random variables
- Parameter atoms carry *frequencies*: normalized atom counts within the sample
- Cost function: arbitrary differentiable function parameterized with a vector of parameter atom frequencies
- Idea: incrementally add models to sample until cost function value \leq threshold Ψ
- If process guided by (partial) derivatives of the cost function wrt. parameter atoms:
Differentiable SAT/ASP

Approach outline (2)

- Each time we decide about which parameter atom to add to partial assignment (the current incomplete model “under construction”):
compute how this decision would influence the overall cost function

=> partial derivatives of cost function wrt. parameter atoms (as variables representing their frequencies in the incomplete sample)
- Select parameter atom and its truth value (signed literal) which minimizes derivative (steepest descent)
- In that sense, we make the iterated (multi-model) SAT/ASP solving process *differentiable*

Differentiable SAT/ASP



Cost functions and parameter atoms for PLP (1)

- Various possibilities for cost function. For deductive probabilistic inference, we can use *Mean Squared Error* (MSE):

$$\text{cost}(\theta_1^v, \theta_2^v, \dots) := \frac{1}{n} \sum_{i=1}^n (\beta(\theta_i) - \phi_i)^2$$

- Parameter atoms θ_i : atoms which carry given (user-defined) probabilities (*weights*) ϕ_i .
- Parameter atom frequencies $\beta(\theta_i)$ updated with each sampled model
- Weighted rules and weighted models can be rewritten as instances of this approach
- Arbitrary MSE cost, parameter atoms, ASP/SAT rules/clauses...; no required independence assumptions
- But of course not all cost functions and logic programs/formulas have a solution (cost=0)

Cost functions and parameter atoms for PLP (2)

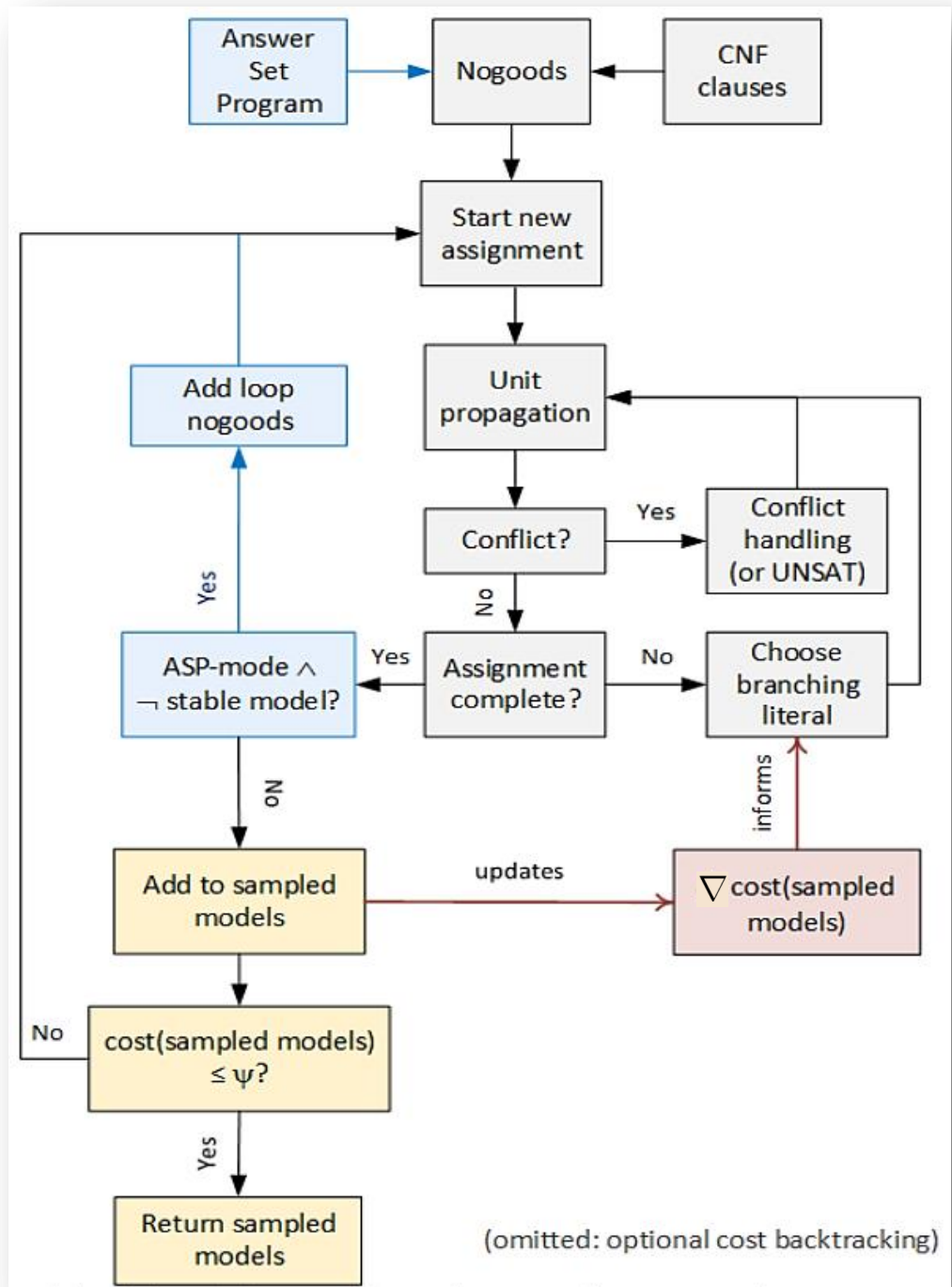
- The Answer Set input program (or analogously SAT formula) can contain arbitrary rules and facts.
- For parameter atoms, it is sensible to add so-called *spanning rules* which make the parameter atoms nondeterministic (although this is not a requirement for our algorithms)

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

- MSE cost function, e.g.,: $\frac{1}{2} ((\beta(a) - 0.2)^2 + (\beta(b) - 0.6)^2)$
(assigns atom a weight 0.2 and atom b weight 0.6)

Native algorithm: Diff-CDNL-ASP/SAT

- Fastest (currently known) implementation of described approach by directly enhancing SAT/ASP solving algorithm
- Current state of the art solving algorithm: CDCL/CDNL
- CDCL (*Conflict-Driven Clause Learning*) based on older DPLL algorithm but with clause learning capability and non-chronological backtracking
- CDNL-ASP (*Conflict-Driven Nogood Learning*): variant of CDNL with *nogoods* (think of clauses with negated literals) as basic representative concept
- Also comprises loop handling (required for non-tight Answer Set programs)
- CDNL used by Clingo/Clasp (but we've created an independent implementation in Scala). Suitable for SAT as well as ASP solving
- We enhanced CDNL-ASP with a new decision literal selection (branching) policy



Propagator-based approach (1)

- Previous implementation approach fast but cannot use existing ASP or SAT solver "out of the box"
- Idea: tweak a regular ASP solver's branching heuristics to decide on parameter atoms' truth values using differentiation
- Various ways to implement this, e.g., *domain heuristics*, external atoms, HEX?, ...
- We use Clingo's *propagators*; cannot directly implement branching heuristics, but can be customized to enforce dynamically created singleton clauses (representing our parameter atom truth assignments)
- Requires outer sampling loop (e.g., Python script using Clingo's Python API) - calls ASP solver multiple times until cost goal reached
- Very slow (at least with current prototypical code)

Propagator-based approach (2)

```
ps = [ (adnumber(freqs['a']), 'a'), (adnumber(freqs['b']), 'b') ]
```

```
c = ( (1*(0.2-ps[0][0])**2) + (1*(0.6-ps[1][0])**2) ) / 2 # example MSE-shaped cost function  
# 0.6 = target probability of b)
```

```
if atom_x == "":
```


```
    return c
```

```
else:
```

```
    pxi = next(i for i,v in enumerate(ps) if v[1] == atom_x)
```

```
    return c.d(ps[pxi][0])
```

partial derivative (using automatic differentiation): if we change the frequency of atom pxi (keeping all other parameter atoms fixed), how does this influence the cost function?



Propagator-based approach (3)

```
for atomlit, atom in param_atoms.iteritems(): # we search for the minimum partial derivative
```

```
    diff_p = __cost_ad(freqs, atom) ← call automatic differentiation (see prev slide)
```

```
    if diff_p < min_diff_lit[1]:
```

```
        min_diff_lit = (atomlit, diff_p)
```

```
    diff_n = -diff_p
```

```
    if diff_n < min_diff_lit[1]:
```

```
        min_diff_lit = (-atomlit, diff_n)
```

← to simplify diff. for negated literals
(represented as negative numbers)

→ `branch_param_lit = min_diff_lit[0]`


Propagator-based approach (4)

```
def propagate(self, control, changes):
    global branch_param_lit
    global param_atoms

    if branch_param_lit != sys.maxint:
        if branch_param_lit > 0:
            control.add_clause([Propagator.solver_lits[branch_param_lit]], True)
        else:
            control.add_clause([-Propagator.solver_lits[abs(branch_param_lit)]], True)
```

negation

previously computed parameter
literal which moves the cost
function into the desired
direction



Mapping to conventional ASP optimization (1)

- Further approach to solve the multi-model cost function: map task to regular Answer Set optimization task using reification
- Reify all sample models and (non-constant) atom predicates using model indices
- In MSE case: directly solve for model probability distribution, using linear equation solving encoded in ASP
- Does not use derivatives
- Slow. But exemplifies how to translate problem to regular (single or top-k model) ASP/SAT optimization

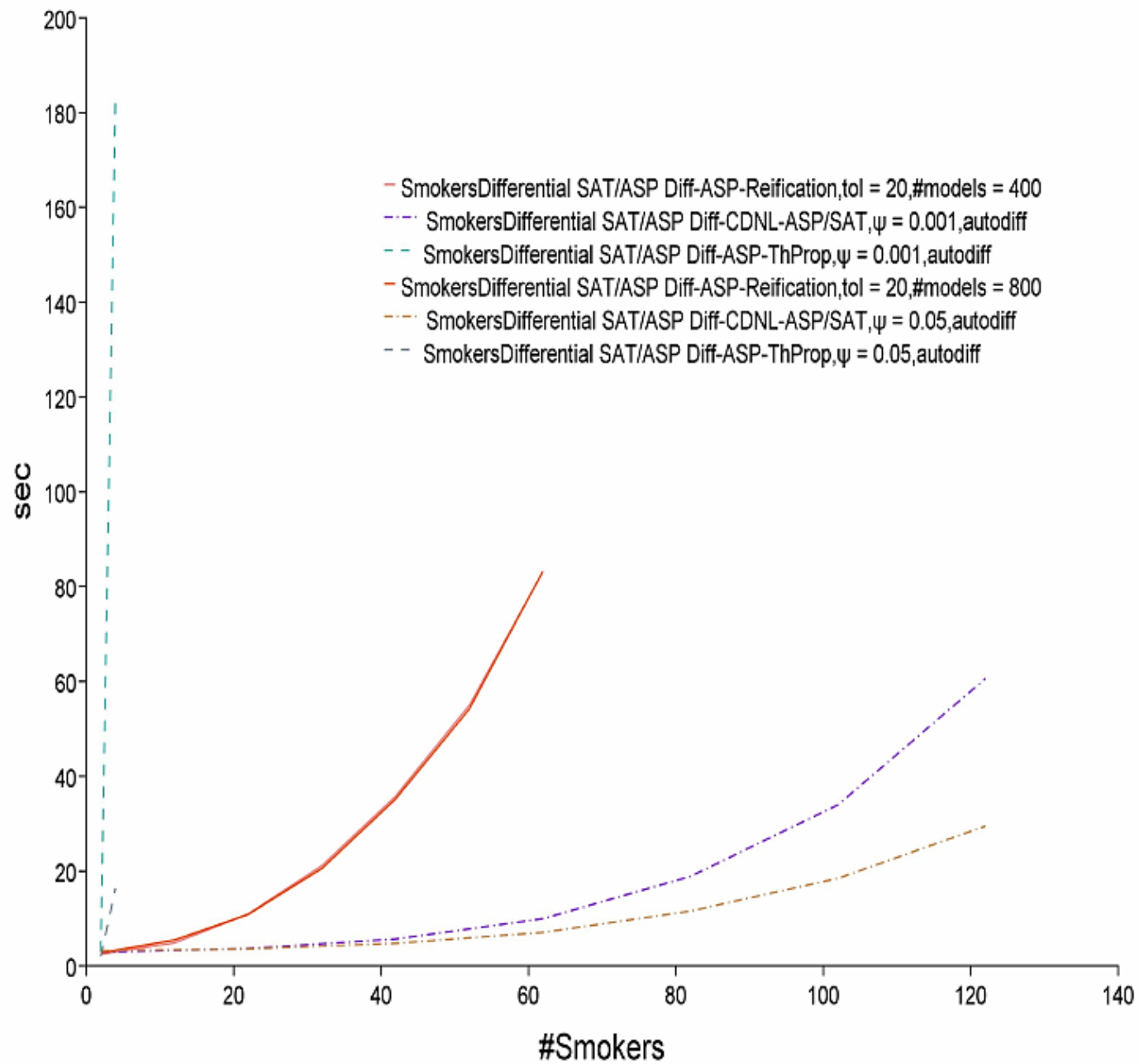
Mapping to conventional ASP optimization (2)

```
#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) }.
```

Smokers



Conclusion

- First approach (to our best knowledge) to differentiation of ASP and SAT solving
- General use case: multi-model optimization with custom cost functions and user-specified accuracy
- Probabilistic Logic Programming as primarily targeted (but not only) use case
- Simple and relatively fast (for a probabilistic logic without dependence restrictions) =>SUM'18
- Uses iterative Boolean assignment / answer set sampling for scalability
- Various implementation approaches, including direct (native) approach based on CDCL/CDNL or custom branching heuristics
- Alternatively, translation to plain Answer Set optimization possible (but quite slow)
- Planned work: further experiments, theoretical criteria for termination (beyond convex cost functions), further optimization of prototype implementations

Any questions?