Solving Weighted Constraints with Applications to Program Analysis

Ravi Mangal[1], Xin Zhang[1], Mayur Naik[1], and Aditya Nori[2]

[1] Georgia Institute of Technology
[2] Microsoft Research

Abstract. Systems of weighted constraints are a natural formalism for many emerging tasks in program analysis and verification. Such systems include both hard and soft constraints: the desired solution must satisfy the hard constraints while optimizing the objectives expressed by the soft constraints. Existing techniques for solving such constraint systems sacrifice scalability or soundness by grounding the constraints eagerly, rendering them unfit for program analysis applications. We present a lazy grounding algorithm that generalizes and extends these techniques in a unified framework. We also identify an instance of this framework that, in the common case of computing the least solution of Horn constraints, strikes a balance between the eager and lazy extremes by guiding the grounding based on the logical structure of constraints. We show that our algorithm achieves significant speedup over existing approaches without sacrificing soundness for several real-world program analysis applications.

1 Introduction

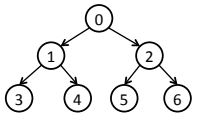
Many emerging tasks in program analysis and verification require optimizing certain objectives in addition to satisfying soundness conditions. These objectives concern aspects such as tradeoffs between precision and cost [14, 15, 32], assumptions about unknown information [1, 12, 13, 18], and treatment of user knowledge [6]. Motivated by such tasks, even existing verifiers have begun to address optimization extensions of conventional logical problems [2, 7, 17].

A natural formalism for such tasks is a system of constraints in the form of first-order inference rules with optional weights. Such a system allows both hard (inviolable) constraints and soft (violable) constraints. The desired solution satisfies all hard constraints while maximizing the sum of the weights of satisfied soft constraints. Hard constraints thus enable to express soundness conditions while soft constraints allow to express the objectives to be optimized.

Figure 1 shows an example of such constraint system that formulates a graph reachability problem. Input predicate $e(n_1, n_2)$ is true if the graph has an edge from node n_1 to n_2. Derived predicate $p(n_1, n_2)$ is true if the graph has a path from n_1 to n_2. Hard constraints (1) and (2) enforce reflexitivity and transitivity conditions respectively, while soft constraint (3) ensures that the desired solution derives the least possible number of paths. The solution thus corresponds to the least fixed-point solution to the graph reachability problem. An example input and solution are shown in Figure 2.

$$
\begin{array}{lll}
 & \forall n_1, & p(n_1, n_1) \qquad (1)\\
\bigwedge & \forall n_1, n_2, n_3, & p(n_1, n_2) \wedge e(n_2, n_3) \Rightarrow p(n_1, n_3) \qquad (2)\\
\bigwedge & 1.5 : \forall n_1, n_2, & \neg p(n_1, n_2) \qquad (3)
\end{array}
$$

Fig. 1. A graph reachability problem formulated using weighted constraints.



Input facts:		**Output facts:**	
$e(0,1)$	$e(0,2)$	$p(0,0)$	$p(0,1)$
$e(1,3)$	$e(1,4)$	$p(0,2)$	$p(0,3)$
$e(2,5)$	$e(2,6)$	$p(0,4)$	$p(0,5)$
		...	

Fig. 2. Example input and solution to the graph reachability problem.

The standard technique for solving such a constraint system involves two phases. In the first phase, the constraints (which are expressed in first-order logic) are *grounded* by instantiating the predicates or relations over all constants in the corresponding input domains. In the second phase, the grounded constraints are solved to produce a solution that satisfies all hard constraints and maximizes the sum of the weights of satisfied soft constraints. This problem, called Weighted Partial Maximum Satisfiability or MaxSAT, is computationally hard [24].

Producing tractable MaxSAT instances motivates the need to avoid grounding constraints needlessly. Naive grounding can cause an exponential blow-up, with respect to input size, in the number of constraints. For the example in Figure 1, naively grounding constraints (1)–(3) instantiates the quantifiers over all nodes n in the graph. For instance, naively grounding constraint (2) results in $|N|^3$ grounded constraints where N is the domain of nodes in the graph.

Several techniques have been proposed to lazily ground constraints [5, 11, 22, 23]. Despite significant advances, however, these techniques produce instances that at best take orders-of-magnitude more time to solve than afforded by tasks in program analysis and verification, and at worst are well beyond the reach of state-of-the-art MaxSAT solvers. For instance, a recent technique [5] grounds hard constraints lazily and soft constraints naively. For the above example, this generates $|N|^2$ constraints from grounding constraint (3). In comparison, a fully lazy approach would only ground as many soft constraints as the number of paths in the least fixed-point solution, as we show in Section 3.

More significantly, all prior techniques achieve scalability by sacrificing soundness, that is, the solution they produce is not even guaranteed to satisfy all the hard constraints. This is tolerable for applications in information retrieval and machine learning that are the primary focus of existing techniques. Soundness, however, is indispensable for our target application domain of program analysis and verification, which renders existing techniques inapplicable.

We propose an iterative lazy grounding algorithm `LGS` that generalizes and extends previous techniques in a unified framework. Starting with an initial set of grounded constraints, `LGS` alternates between lazily grounding and invoking an off-the-shelf MaxSAT solver to solve these grounded constraints. At the end of each iteration, the solution produced by the solver is checked for violations

of any hard or soft constraint. Any violated constraint is instantiated with the values that led to the violation and added to the set of grounded constraints to be fed to the solver. This continues until no hard constraints are violated and the weight of the produced solution stops changing.

LGS is parameterized by an initial grounding strategy that provides the set of constraints to be grounded at the start of the iterative process. This set can range from an empty set to the set of all constraints fully grounded, thereby enabling the complete spectrum of choices from fully lazy to fully eager. We prove that, for any set in this range, LGS produces an optimal and sound solution. Moreover, in the case where all constraints are Horn clauses whose least solution is desired, which is common for applications in program analysis and verification, we identify a strategy that upfront grounds all constraints that will necessarily be grounded during the iterative process. For instance, all the constraints in the example in Figure 1 are Horn clauses, and our proposed strategy upfront grounds constraints (1) and (2) into a set of constraints that would produce the least solution as their answer. This strategy effectively strikes a balance between the eager and lazy extremes by guiding the grounding based on the logical structure of the constraints.

We evaluate LGS on 21 benchmarks resulting from applying three static analyses (datarace detection, monomorphic call site inference, and downcast safety checking) to seven large Java programs. We compare four instances of LGS:

- Eager: exhaustive initial grounding, that is, a fully eager approach [22];[1]
- SoftCegar (lazy for hard constraints only): a semi-lazy approach [5] that fully grounds only soft constraints upfront;
- Lazy: no initial grounding, that is, our fully lazy approach; and
- Guided: our guided initial grounding.

Our Guided approach, on average, is 16× faster than our Lazy approach while the other approaches run out of memory.

In summary, our work makes the following contributions:

1. We present a lazy grounding algorithm that generalizes and extends previous techniques for solving weighted first-order constraints in a unified framework. We prove the optimality and soundness of our algorithm.
2. We identify an instance of our framework that strikes a balance between the eager and lazy extremes in the case of Horn constraints by guiding the grounding based on the logical structure of the constraints.
3. We evaluate the performance of our algorithm on realistic program analysis applications. Our guided approach significantly speeds up the solving process for these applications without sacrificing soundness.

2 Preliminaries

Our goal is to take weighted constraints such as those in the graph reachability example in Figure 1, conjoin them with extensional predicates (EDB) such as

[1] Although fully eager, this approach is not equivalent to naive grounding and employs optimizations. See Section 5 for details.

Grounded constraints:		MaxSAT formula:
Hard clauses:	$e(0,1) \wedge$	$b0 \wedge$
	$e(0,2) \wedge$	$b1 \wedge$
	$p(0,0) \wedge$	$b2 \wedge$
	$p(1,1) \wedge$	$b3 \wedge$
	$(\neg p(0,0) \vee \neg e(0,1) \vee p(0,1)) \wedge$	$(\neg b2 \vee \neg b0 \vee b4) \wedge$
	$(\neg p(0,1) \vee \neg e(1,3) \vee p(0,3)) \wedge ...$	$(\neg b4 \vee \neg b5 \vee b6) \wedge ...$
Soft clauses:	$1.5 : (\neg p(0,1)) \wedge$	$1.5 : (\neg b4) \wedge$
	$1.5 : (\neg p(1,1)) \wedge ...$	$1.5 : (\neg b3) \wedge ...$

Fig. 3. Example grounded constraints and corresponding MaxSAT formula.

the edge relation e in Figure 2, and compute intensional predicates (IDB) such as the path relation p in Figure 2. We seek to develop an iterative *ground-and-solve* approach that consists of two phases in each iteration: grounding followed by solving. The *grounding* phase grounds a subset of the given constraints on the EDB. A sample result of this phase for our graph reachability example is the grounded constraints shown on the left in Figure 3. The *solving* phase converts these grounded constraints into a MaxSAT formula, solves it using an off-the-shelf MaxSAT solver, and maps the solution to the IDB. The MaxSAT formula produced from the grounded constraints in Figure 3 is shown in the same figure on the right. The remainder of this section introduces the setting for these two phases and our overall approach.

We begin by defining the abstract syntax of weighted constraints, shown in Figure 4. A system of weighted constraints C consists of a set of hard constraints H and a set of soft constraints S.

A hard constraint $h \in H$ is an inference rule $A \Rightarrow B$, where A is a conjunction of facts and B is a disjunction of facts. A *fact* t comprises a relation name and a tuple of arguments, which include variables and constants; a fact is a *ground fact* g when all arguments are constants.

A soft constraint $s \in S$ is a hard constraint along with a positive real weight w. A weight has a natural probabilistic interpretation, where the confidence associated with a soft constraint increases with the weight. For more details on the precise semantics of weights, the reader is referred to [8].

For convenience in formulating program analysis and verification tasks, we augment the constraint system with an *input* P which is a set of ground facts that comprise the EDB. The solution to the constraints, *output* Q, represents the set of ground facts that comprise the IDB. For instance, the input can encode the program being analyzed, while the output can represent the analysis result.

Example. The graph reachability problem applied to the graph in Figure 2 can be formulated as a system of weighted constraints (H, S) where H comprises hard constraints (1) and (2) in Figure 1, and S comprises soft constraint (3) with an arbitrary weight of 1.5. Further, the input P comprises all ground facts in relation $e(n_1, n_2)$, denoting all edges in the graph, while output Q comprises all ground facts in relation $p(n_1, n_2)$, denoting all paths in the graph. □

$$
\begin{array}{rlrl}
\text{(relation)} & r \in \mathbf{R} & \text{(argument)} & a \in \mathbf{A} = \mathbf{V} \cup \mathbf{C} \\
\text{(constant)} & c \in \mathbf{C} & \text{(fact)} & t \in \mathbf{T} = \mathbf{R} \times \mathbf{A}^* \\
\text{(variable)} & v \in \mathbf{V} & \text{(ground fact)} & g \in \mathbf{G} = \mathbf{R} \times \mathbf{C}^* \\
\text{(valuation)} & \sigma \in \mathbf{V} \to \mathbf{C} & \text{(weight)} & w \in \mathbb{R}+ = (0, \infty]
\end{array}
$$

$$
\begin{array}{rl}
\text{(hard constraints)} & H ::= \{h_1, ..., h_n\},\ h ::= \bigwedge_{i=1}^{n} t_i \Rightarrow \bigvee_{i=1}^{m} t'_i \\
\text{(soft constraints)} & S ::= \{s_1, ..., s_n\},\ \ s ::= (h, w)
\end{array}
$$

$$
\begin{array}{rlcl}
\text{(weighted constraints)} & C & ::= & (H, S) \\
\text{(input, output)} & P, Q & \subseteq & \mathbf{G}
\end{array}
$$

Fig. 4. Abstract syntax of weighted constraints.

$$
\begin{array}{rl}
[\![(H, S)]\!] & = ([\![H]\!], [\![S]\!]) \\
[\![\{h_1, ..., h_n\}]\!] & = \bigwedge_{i=1}^{n} [\![h_i]\!] \\
[\![\{s_1, ..., s_n\}]\!] & = \bigwedge_{i=1}^{n} [\![s_i]\!] \\
[\![h]\!] & = \bigwedge_{\sigma} [\![h]\!]_\sigma \\
[\![(h, w)]\!] & = \bigwedge_{\sigma} ([\![h]\!]_\sigma, w) \\
[\![\bigwedge_{i=1}^{n} t_i \Rightarrow \bigvee_{i=1}^{m} t'_i]\!]_\sigma & = (\bigvee_{i=1}^{n} \neg [\![t_i]\!]_\sigma \vee \bigvee_{i=1}^{m} [\![t'_i]\!]_\sigma) \\
[\![r(a_1, ..., a_n)]\!]_\sigma & = r([\![a_1]\!]_\sigma, ..., [\![a_n]\!]_\sigma) \\
[\![v]\!]_\sigma & = \sigma(v) \\
[\![c]\!]_\sigma & = c
\end{array}
$$

$$
\begin{array}{rl}
\text{(ground clause)} & \rho ::= \bigvee_{i=1}^{n} \neg g_i \vee \bigvee_{i=1}^{m} g'_i \\
\text{(hard clauses)} & \phi ::= \bigwedge_{i=1}^{n} \rho_i \\
\text{(soft clauses)} & \psi ::= \bigwedge_{i=1}^{n} (\rho_i, w_i)
\end{array}
$$

Fig. 5. From weighted constraints to ground clauses.

We now describe the grounding phase. Weighted constraints are grounded by instantiating the predicates or relations over all constants in the corresponding input domains. The grounding procedure is shown in Figure 5. The procedure grounds each weighted constraint into a set of corresponding clauses. In particular, the conversion $[\![h]\!] = \bigwedge_{\sigma} [\![h]\!]_\sigma$ grounds hard constraint h by enumerating all possible groundings σ of variables to constants, yielding a different clause for each unique valuation to the variables in h. Enumerating all possible valuations, called *full grounding*, does not scale to real-world programs and analyses.

Example. Figure 3 shows a subset of the ground clauses constructed for the constraints from Figure 1 applied to the graph in Figure 2. Along with the input ground facts from relation $e(n_1, n_2)$, hard constraints (1) and (2) in the graph reachability problem are grounded to construct the shown set of hard clauses, while soft constraint (3) is grounded to produce the shown set of soft clauses. □

We next describe the solving phase. The ground clauses are solved to produce a solution that satisfies all hard clauses and maximizes the sum of the weights of satisfied soft clauses. This problem is called Weighted Partial Maximum Satisfiability or MaxSAT [24]. In the solving process, ground clauses are first converted into a boolean MaxSAT formula by replacing each unique ground fact with a separate boolean variable. The `MaxSAT` procedure in Figure 6 then takes this

$$\texttt{MaxSAT}(\phi, \textstyle\bigwedge_{i=1}^{n}(\rho_i, w_i)) = \begin{cases} \text{UNSAT} & \text{if } \nexists Q:\ Q \models \phi \\ Q \text{ such that } \begin{bmatrix} Q \models \phi \text{ and} \\ \Sigma_{i=1}^{n}\{w_i \mid Q \models \rho_i\} \text{ is maximized} \end{bmatrix} & \text{otherwise} \end{cases}$$

$$\begin{array}{rcl} Q \models \bigwedge_{i=1}^{n} \rho_i & \text{iff} & \forall i : Q \models \rho_i \\ Q \models \bigvee_{i=1}^{n} \neg g_i \vee \bigvee_{i=1}^{m} g'_i & \text{iff} & \exists i : g_i \notin Q \text{ or } \exists i : g'_i \in Q \end{array}$$

$$\texttt{Weight}(Q, \textstyle\bigwedge_{i=1}^{n}(\rho_i, w_i)) \ = \ \Sigma_{i=1}^{n}\{\ w_i \mid Q \models \rho_i\ \}$$

$$\texttt{Violations}(h, Q) \ = \ \{\ [\![h]\!]_\sigma \mid Q \not\models [\![h]\!]_\sigma\ \}$$

Fig. 6. Specification of solving MaxSAT formulae.

formula, comprising a set of hard clauses ϕ and a set of soft clauses ψ, as input. The procedure returns either: (1) UNSAT, if no assignment of truth values to the boolean variables satisfies the set of hard clauses ϕ, or (2) a solution Q, denoting the assignment "$\lambda g.(g \in Q)\,?\,\mathsf{true}:\mathsf{false}$" that sets variables corresponding to ground facts contained in Q to true, and the rest to false. Solution Q not only satisfies all hard clauses in ϕ (it is *sound*) but it also maximizes the sum of the weights of satisfied soft clauses in ψ (it is *optimal*). Q is not necessarily unique; two solutions Q_1 and Q_2 are *equivalent* if $\texttt{Weight}(Q_1, \psi) = \texttt{Weight}(Q_2, \psi)$. Many off-the-shelf MaxSAT solvers implement the above procedure.

Example. Figure 3 shows a snippet of the MaxSAT formula constructed for the system of weighted constraints from Figure 1 applied to the graph in Figure 2. The formula is constructed from the grounded hard and soft clauses, shown in the same figure, by replacing each unique ground fact with a separate boolean variable. It is then fed to a MaxSAT solver to generate the final solution Q. □

Recall that full grounding is infeasible in practice. In the following section, we propose an iterative lazy grounding technique that grounds and solves only a subset of constraints in each iteration. The `Violations` procedure in Figure 5 is invoked by our technique to find any constraints violated by the solution in each iteration. This procedure take as input a hard constraint h and a MaxSAT solution Q, and returns all grounded instances of h that are violated by Q.

Example. Let h denote the constraint $p(n_1, n_2) \wedge e(n_2, n_3) \Rightarrow p(n_1, n_3)$ from our example in Figure 1. If the MaxSAT solution is $Q = \{p(0, 0)\}$, then we have:

$$\texttt{Violations}(h, Q) \ = \ \{(\neg p(0,0) \vee \neg e(0,1) \vee p(0,1))\}$$

In this case, all grounded instances of the constraint are satisfied by the solution Q except the instance shown above. □

There are many ways to implement the above procedure, including via SMT and Datalog solvers; we follow existing techniques [22,23] and implement it using SQL queries that can be executed by an off-the-shelf RDBMS.

Example. Consider the constraint $\neg p(n_1, n_2) \vee \neg e(n_2, n_3) \vee p(n_1, n_3)$ from the graph reachability example in Figure 1. The SQL query to find violations of this constraint by a solution $p(n_1, n_2)$ is:

```
SELECT DISTINCT P.c1, P.c2, E.c2 FROM p AS P, e AS E
WHERE (P.c2 = E.c1 AND (NOT EXISTS (
                SELECT * FROM p
                WHERE p.c1 = P.c1 AND p.c2 = E.c2)))
```

This SQL query assumes that the database has a table for each relation specified in the constraints, and the table columns are named c1 through cM where M is the number of domains in a relation. □

3 LGS: A Ground-and-Solve Framework

We propose a framework LGS for solving systems of weighted constraints. The framework, described in Algorithm 1, has the following key features:

1. LGS generalizes and extends existing techniques for solving weighted constraints in a lazy, iterative manner. In addition to a fully lazy approach that starts from an empty set of grounded constraints, LGS allows starting from any non-empty set of initial grounded constraints to accelerate convergence while guaranteeing to produce a sound, optimal result equivalent to the full grounding solution (Theorem 1). In contrast, all existing techniques forgo this guarantee (see related work in Section 5).
2. In the case where all constraints are Horn clauses whose least solution is desired, LGS exploits the logical structure of the constraints to produce an initial grounding that is optimal (Theorem 2).
3. LGS provides maximum flexibility, lazily grounding both soft and hard constraints. On the other hand, existing techniques typically only allow hard constraints to be ground lazily (see Section 5).

LGS takes a weighted constraint system (H, S) as input and produces a boolean assignment Q. [1] LGS also takes as input a grounding strategy θ, which can be Eager, Lazy, Guided, or Intermediate. The choice of this strategy only affects scalability and does not affect the soundness or optimality of Q; in particular, in Theorem 1, we prove that all these strategies yield a final output that is equivalent to Q. Strategy Eager corresponds to full grounding, a baseline which does not use a lazy approach. Strategy Intermediate corresponds to using a subset of the full grounded set of constraints as the initial set. The other two strategies, Lazy and Guided, are instances of our lazy iterative technique. These two strategies differ only with respect to how they construct the initial grounding, which is presented in Algorithm 2.

In line 4, LGS invokes the `InitialGrounding` procedure (described in Algorithm 2) with an input grounding strategy θ to compute an initial set of hard clauses ϕ and soft clauses ψ. For the case when $\theta =$ Eager, the MaxSAT solution to a full grounding of ϕ and ψ is returned (line 5). Otherwise, LGS enters the loop

[1] We assume that any input P is encoded as part of the hard constraints H. For clarity, we also assume that the hard constraints H are satisfiable, allowing us to elide showing UNSAT as a possible alternative to output Q.

Algorithm 1 LGS: the ground-and-solve framework.

1: **PARAM** $\theta \in \{$ Eager, Lazy, Guided, Intermediate $\}$: Grounding strategy.
2: **INPUT** (H, S): Weighted constraints.
3: **OUTPUT** Q: Solution (assumes $[\![H]\!]$ is satisfiable).
4: $(\phi, \psi) :=$ InitialGrounding(θ, H, S)
5: **if** $(\theta =$ Eager$)$ **then return** MaxSAT(ϕ, ψ)
6: $Q := \emptyset$; $w := 0$
7: **while true do**
8: $\quad \phi' := \bigwedge_{h \in H} \bigwedge$ Violations(h, Q)
9: $\quad \psi' := \bigwedge_{(h,w) \in S} \bigwedge \{ (\rho, w) \mid \rho \in$ Violations$(h, Q) \}$
10: $\quad (\phi, \psi) := (\phi \wedge \phi', \psi \wedge \psi')$
11: $\quad Q' :=$ MaxSAT(ϕ, ψ)
12: $\quad w' :=$ Weight(Q', ψ)
13: $\quad$ **if** $(w' = w \wedge \phi' =$ true$)$ **then return** Q
14: $\quad Q := Q'$; $w := w'$
15: **end while**

defined in lines 7–15. In each iteration of the loop, the algorithm keeps track of the previous solution Q, and the weight w of the solution Q by calling the Weight procedure specified in Figure 6. Initially, the solution is empty with weight zero (line 6). In line 8, LGS computes all the violations of the hard constraints for the previous solution Q. Similarly, in line 9, the set of soft clauses ψ' violated by the previous solution Q is computed. In line 10, both sets of violations ϕ' and ψ' are added to the corresponding sets of grounded hard clauses ϕ and grounded soft clauses ψ respectively. The intuition for adding violated hard clauses ϕ' to the set ϕ is straightforward—the set of hard clauses ϕ is not sufficient to prevent the MaxSAT procedure from producing a solution Q that violates the set of hard constraints H. The intuition for soft clauses is similar—since the goal of MaxSAT is to maximize the sum of the weights of satisfied soft constraints in S, and all weights in our weighted constraint system are positive, any violation of a soft clause possibly leads to a sub-optimal solution which could have been avoided if the violated clause was present in the set of soft clauses ψ.

In line 11, this updated set ϕ of hard clauses and set ψ of soft clauses are fed to the MaxSAT procedure to produce a new solution Q' and its corresponding weight w'. At this point, in line 13, the algorithm checks if the terminating condition is satisfied by the solution Q'. Theorem 1 states that our LGS algorithm always terminates with a sound and optimum solution. The termination condition ensures that the current solution satisfies all the hard clauses ($\phi' =$ true) while ensuring that there is no benefit in adding any new soft clauses ($w = w'$). The proof of the theorem is provided in Appendix A.

Theorem 1. (Soundness and Optimality of LGS) *For any weighted constraint system (H, S) where H is satisfiable, $\text{LGS}_\theta(H, S)$ produces equivalent results under $\theta =$ Eager, $\theta =$ Lazy, $\theta =$ Guided, and $\theta =$ Intermediate.*

The initial grounding used by LGS is computed by Algorithm 2. It takes a weighted constraint system (H, S) and initial grounding strategy θ as inputs.

Algorithm 2 `InitialGrounding`

INPUT $\theta \in \{$ Eager, Lazy, Guided $\}$: Grounding strategy.
INPUT (H, S): Weighted constraints.
OUTPUT (ϕ, ψ): Initial grounding of weighted constraints.
switch (θ)
case Eager**: return** $[\![(H, S)]\!]$
case Lazy**: return** (true, true)
case Guided**:**
  $\phi :=$ any $\bigwedge_{i=1}^{n} \rho_i$ such that $\forall i : \exists h \in H : \exists \sigma : \rho_i = [\![h]\!]_\sigma$
  return $(\phi,$ true$)$
case Intermediate**:**
  $\phi :=$ any $\bigwedge_{i=1}^{n} \rho_i$ such that $\forall i : \exists h \in H : \exists \sigma : \rho_i = [\![h]\!]_\sigma$
  $\psi :=$ any $\bigwedge_{i=1}^{n} \rho_i$ such that $\forall i : \exists s \in S : \exists \sigma : \rho_i = [\![s]\!]_\sigma$
  return (ϕ, ψ)
end switch

If the Eager grounding strategy is used, the algorithm enumerates all possible groundings for the constraints, yielding a MaxSAT clause for every possible valuation of the variables in a constraint. In practice, this produces an intractably large number of clauses. On the other hand, a Lazy grounding strategy results in no clauses being grounded initially. The algorithm simply returns an empty set for both hard clauses ϕ and soft clauses ψ. The Intermediate strategy operates between the two extremes of Eager and Lazy grounding: it picks a valid subset of grounded hard clauses as the initial set ϕ and a valid subset of grounded soft clauses as the initial set ψ. The Guided strategy is a special case of the Intermediate strategy: it picks a valid subset of grounded hard clauses as the initial set ϕ while continuing to have an empty initial set of soft clauses. In particular, when the hard constraints in the weighted constraint system are Horn rules, the Guided strategy prescribes a recipe for generating an optimal initial set of grounded constraints.

In Theorem 2, we show that for Horn constraints, the Lazy strategy grounds at least as many hard clauses as the number of true ground facts in the least solution of such Horn rules. Also, and more importantly, we show there exists a Guided strategy that can upfront discover the set of all these necessary hard clauses that are grounded by the Lazy strategy and guarantee that no more clauses, besides those in the initial set, will be grounded. In practice, for constraints with Horn hard rules, using this Guided strategy for initial grounding allows the iterative process to terminate in far fewer iterations while ensuring that each iteration does approximately as much work as when using the Lazy strategy. The proof of the theorem is provided in Appendix B.

Theorem 2. (Optimal Initial Grounding for Horn Rules) *If a weighted constraint system is constituted of a set of hard constraints H, each of which is a Horn rule $\bigwedge_{i=1}^{n} t_i \Rightarrow t_0$, whose least solution is desired:*

$$G = \mathsf{lfp}\ \lambda G'.\ G' \cup \{\ [\![t_0]\!]_\sigma \mid (\textstyle\bigwedge_{i=1}^{n} t_i \Rightarrow t_0) \in H \text{ and } \forall i \in [1..n] : [\![t_i]\!]_\sigma \in G'\ \},$$

Iteration 1	Iteration 2	Iteration 3	Iteration 4
$p(0,0)$ $p(1,1)$ $p(2,2)$ $p(3,3)$ $p(4,4)$ $p(5,5)$ $p(6,6)$	$\neg p(0,0) \vee \neg e(0,1) \vee p(0,1)$ $\neg p(0,0) \vee \neg e(0,2) \vee p(0,2)$ $\neg p(1,1) \vee \neg e(1,3) \vee p(1,3)$ $\neg p(1,1) \vee \neg e(1,4) \vee p(1,4)$ $\neg p(2,2) \vee \neg e(2,5) \vee p(2,5)$ $\neg p(2,2) \vee \neg e(2,6) \vee p(2,6)$ $1.5 : \neg p(0,0)$ $1.5 : \neg p(1,1)$ $1.5 : \neg p(2,2)$ $1.5 : \neg p(3,3)$ $1.5 : \neg p(4,4)$ $1.5 : \neg p(5,5)$ $1.5 : \neg p(6,6)$	$\neg p(0,1) \vee \neg e(1,3) \vee p(0,3)$ $\neg p(0,1) \vee \neg e(1,4) \vee p(0,4)$ $\neg p(0,2) \vee \neg e(2,5) \vee p(0,5)$ $\neg p(0,2) \vee \neg e(2,6) \vee p(0,6)$ $1.5 : \neg p(0,1)$ $1.5 : \neg p(0,2)$ $1.5 : \neg p(1,3)$ $1.5 : \neg p(1,4)$ $1.5 : \neg p(2,5)$ $1.5 : \neg p(2,6)$	$1.5 : \neg p(0,3)$ $1.5 : \neg p(0,4)$ $1.5 : \neg p(0,5)$ $1.5 : \neg p(0,6)$

Table 1. Additional clauses grounded in each iteration for graph reachability example.

then for such a system, (a) $\texttt{LGS}_{\theta=\mathsf{Lazy}}(H, \emptyset)$ *grounds at least* $|G|$ *clauses, and (b)* $\texttt{LGS}_{\theta=\mathsf{Guided}}(H, \emptyset)$ *with the initial grounding* ϕ *does not ground any more clauses:*

$$\phi = \bigwedge\{\ \textstyle\bigvee_{i=1}^{n} \neg[\![t_i]\!]_\sigma \vee [\![t_0]\!]_\sigma \mid (\bigwedge_{i=1}^{n} t_i \Rightarrow t_0) \in H \text{ and } \forall i \in [0..n] \colon [\![t_i]\!]_\sigma \in G\ \}.$$

We illustrate the differences between the Eager, Lazy and Guided strategies via our graph reachability example described in Figures 1 and 2. The Eager strategy converts the weighted constraints into MaxSAT clauses by naively enumerating all possible valuations of the variables in a constraint. This results in many clauses that are trivially satisfiable and thus, play no effective role in influencing the final solution. For our graph example, Eager grounding generates clauses such as $\neg p(0,1) \vee \neg e(1,5) \vee p(1,5)$ and $\neg p(1,4) \vee \neg e(4,2) \vee p(1,2)$, that are trivially satisfied given the input edge relation.

The Lazy strategy, on the other hand, uses the lazy, iterative process and, in each iteration, grounds only those clauses that are violated by the current MaxSAT solution. Table 1 progressively shows the MaxSAT clauses that are grounded in each iteration for our graph example. Initially, the set of grounded clauses is empty. The process ends when all the necessary clauses are grounded and the optimal solution is produced.

Finally, the Guided strategy accelerates the Lazy grounding process by grounding a subset of hard clauses upfront. As shown in Theorem 2, for Horn constraints, the Guided strategy grounds only those hard clauses that appear in the least fixed point solution of these constraints. For our graph example, this implies that the Guided strategy upfront grounds all the hard clauses in Table 1.

4 Empirical Evaluation

We evaluate LGS on 21 benchmarks resulting from applying three program analyses to seven large Java programs. All our experiments were done using Oracle HotSpot JVM 1.6.0 on a Linux server with 64GB RAM and 3.0GHz processors.

Table 2 shows statistics of three Java programs (`antlr`, `avrora`, `lusearch`) from the DaCapo suite [3], each comprising 131–198 thousand lines of code.

	brief description	# classes		# methods		bytecode (KB)		source (KLOC)	
		app	total	app	total	app	total	app	total
microbench	stack data-structure implementation	2	5	9	15	0.3	0.4	0.1	1.4
antlr	parser/translator generator	111	350	1,150	2,370	128	186	29	131
avrora	microcontroller simulator/analyzer	1,158	1,544	4,234	6,247	222	325	64	193
lusearch	text indexing and search tool	219	640	1,399	3,923	94	250	40	198

Table 2. Characteristics of our benchmark programs. Columns "total" and "app" report numbers with and without counting Java standard library code, respectively.

	total constraints	hard constraints	input relations	output relations
polysite	76	12	50	42
downcast	77	12	51	43
datarace	30	5	18	18

Table 3. Statistics of our program analyses.

For the sake of brevity, we report the statistics and evaluation results for the additional four programs in Appendix C.

Table 3 describes our three analyses: monomorphic call site inference (*polysite*), downcast safety checking (*downcast*), and datarace detection (*datarace*). The *polysite* and *downcast* analyses involve computing the program call-graph and points-to information, and have been used in previous works [16, 30, 32] to evaluate pointer analyses. The *datarace* analysis is from [21] and includes thread-escape and may-happen-in-parallel analyses. All constraints in these analyses are Horn rules. Each analysis is sound but incomplete. The hard constraints express the soundness conditions of the analysis. The soft constraints influence the analysis precision and scalability by encoding tradeoffs between different abstractions and incorporating user knowledge about analysis outcomes.

Our evaluation considers four different initial grounding strategies: (1) Eager: exhaustive initial grounding, that is, a fully eager approach as implemented by TUFFY [22]; (2) SoftCegar: a semi-lazy approach [5] that fully grounds only soft constraints upfront; (3) Lazy: no initial grounding, that is, our fully lazy approach; and (4) Guided: our guided initial grounding. All these approaches can be viewed as instances of our LGS algorithm. To implement the Eager approach, we replace the call to the MaxSAT solver on line 5 of Algorithm 1 with a call to TUFFY, a non-iterative solver for weighted constraints. TUFFY employs the Eager approach but uses optimizations to accelerate the solving process. To implement the SoftCegar approach, we replace the call to the MaxSAT solver on line 11 of Algorithm 1 with a call to TUFFY, with grounded hard constraints but ungrounded soft constraints. Essentially, the hard constraints are grounded lazily by our iterative algorithm while soft constraints are grounded by TUFFY in an eager manner. Note that the Eager and SoftCegar approaches sacrifice soundness for performance, and are unsuitable for program analysis applications. To compute the initial grounded hard constraints in the Guided strategy, we use bddbddb [31], a Datalog solver. For the MaxSAT procedure invoked by the Lazy and Guided approaches, we use MCSls [19], a MaxSAT solver that guarantees soundness.

analysis	program	$\|EDB\|$ ($\times 10^6$)	total ground clauses	# iterations		total time (hh:mm)		# ground clauses($\times 10^6$)		$\|IDB\|$ ($\times 10^6$)
				Lazy	Guided	Lazy	Guided	Lazy	Guided	
polysite	`antlr`	8.2	$2.9\text{x}10^{29}$	167	7	15:34	1:12	12	13	8.7
	`avrora`	18	$1.8\text{x}10^{31}$	157	8	29:02	2:04	15	17	12
	`lusearch`	10	$1.2\text{x}10^{30}$	173	8	13:03	1:12	10	11	7.8
downcast	`antlr`	8.2	$2.9\text{x}10^{29}$	159	9	18:36	1:41	13	15	9.4
	`avrora`	18	$1.8\text{x}10^{31}$	191	9	41:07	2:41	16	18	11
	`lusearch`	10	$1.2\text{x}10^{30}$	192	9	22:12	1:32	11	12	8.1
datarace	`antlr`	1.6	$2.4\text{x}10^{24}$	751	4	3:02	0:05	0.2	0.3	0.2
	`avrora`	2.6	$1.8\text{x}10^{26}$	492	12	6:31	0:25	0.8	1.6	0.7
	`lusearch`	1.6	$1.7\text{x}10^{25}$	429	6	2:38	0:14	0.6	1.0	0.5

Table 4. Evaluation of LGS on program analysis applications.

Table 4 summarizes the results of running LGS with the Lazy and Guided strategies on our benchmarks. The Eager and SoftCegar approaches run out of memory for all the problem instances above. Basically both these approaches are too aggressive in their grounding strategy despite the optimizations used by tools like TUFFY. Though these approaches have been successfully used for information retrieval applications, the nature of those applications allows approximations that are not acceptable for program analysis applications.

The '$|EDB|$' column in the table shows the number of input ground facts for the given system of weighted constraints. The input facts encode the analyzed program in terms of relations that are relevant to the program analysis being applied. The large size of the EDB (average of 8.6×10^6 facts) reflects the fact that real-world programs are being analyzed using sophisticated analyses.

The 'total ground clauses' column reports the theoretical upper bound for the number of ground clauses if all the constraints were grounded naively. It is clear from the numbers that any approach attempting to tackle problems of this scale needs to employ lazy techniques for solving the constraints.

The next two columns show the number of iterations, and the total running time of our Lazy and Guided approaches for solving the weighted constraints. The reported running time is the end-to-end time including all the phases involved in solving these constraints. For the Guided approach, this includes the time needed for generating the initial set of ground clauses. For all benchmarks, the Guided strategy clearly outperforms the Lazy strategy with an average speedup of 16×. This is consistent with the fact that, on average, the Guided approach needs 47× fewer iterations than the Lazy approach. The higher drop in the number of iterations compared to the drop in running time is attributable to the fact that, on average, the Guided approach needs more time per iteration (in Table 4 this is 'total time' divided by '# of iterations'). This is primarily because the Lazy approach starts with an empty initial grounded set and thereby, the average size of the MaxSAT problem solved by this approach is smaller than for the Guided approach. Note that the *datarace* analysis benefits more from using the Guided approach compared to the other analyses. This is primarily due to the

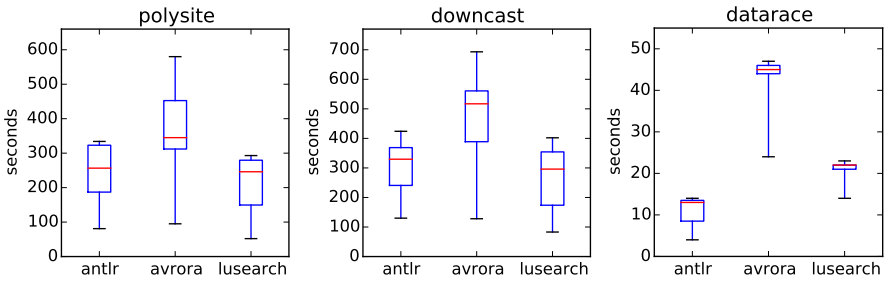


Fig. 7. MaxSAT solver time per iteration for the Guided approach. The boxes extend from the lower to upper quartile values of the iteration times, with a line at the median value. The solid lines extend from the boxes to show the range of the iteration times.

fact that the *datarace* analysis has fewer constraints that the others (Table 4). As a result, the Guided approach is able to generate most of the required ground clauses upfront for this analysis.

The '# ground clauses' column indicates the distinct number of clauses grounded by the Lazy and Guided approaches in the process of solving the weighted constraints. In other words, it indicates the size of the problem fed to the MaxSAT solver in the last iteration of the `LGS` algorithm. The higher number of ground clauses for the Guided strategy in comparison to the Lazy one is along expected lines. Theorem 2 states that the initial grounded set used by the Guided is inclusive of the ground clauses generated by the Lazy strategy.

Finally, the last column shows the size of the IDB, that is, the number of ground facts in the solution produced by `LGS`. The Lazy and the Guided strategies produce the same number of IDB facts, consistent with Theorem 1.

Figure 7 plots the time consumed by the MaxSAT solver in each iteration using the Guided approach. The maximum MaxSAT time of 693 seconds arises for one of the iterations of *downcast* on `avrora`. More generally, the average running time per iteration, across the nine benchmarks shown here, is 227 seconds. This successfully demonstrates that even for large, real-world programs our technique, in conjunction with the Guided strategy, generates MaxSAT instances that are small enough to be handled by existing solvers. The running times look very similar for the Lazy approach with a maximum MaxSAT time of 1,089 seconds for a particular iteration of *downcast* on `avrora`, and an average of 160 seconds. We omit the corresponding graph for the sake of brevity.

Due to the inability of the Eager and SoftCegar approaches to scale to real-world programs, we evaluate these approaches by applying the *polysite* analysis on `microbench`, a micro-benchmark with only about 1.4K lines of code (Table 2). Table 5 shows the results of this evaluation. The input EDB for this benchmark only contains 1,274 ground facts. Even so, total running time of the Eager approach is almost 14× of the running times for our Lazy and Guided approaches, while also grounding more clauses. More importantly, the solution produced by the Eager approach is much worse than that produced by our pro-

strategy	# iterations	total time	# ground clauses	solution cost	$\|IDB\|$
Eager	1	6m 56s	604	77	193
SoftCegar	-	-	-	-	-
Lazy	23	29s	578	0	528
Guided	6	28s	599	0	528

Table 5. Evaluation of `LGS` for *polysite* applied to `microbench` where $|EDB| = 1,274$. The SoftCegar approach did not terminate even after running for 24 hours.

posed approaches. The 'solution cost' column shows the sum of the weights of the constraints violated by the final solution. The solution generated by the Eager approach has a cost of 77 while our proposed approaches produce a solution not violating any constraint. This is also reflected in the fact that the size of the IDB reported by the Eager approach is much smaller. The SoftCegar approach fails to terminate on this benchmark even after running for 24 hours, highlighting a limitation of the approach proposed in [5]. This approach uses a lazy, iterative strategy only for hard constraints while the soft constraints are grounded upfront. To ensure scalability, this strategy gives up on soundness, and thus the solution produced in each iteration might even violate the already grounded hard constraints. However, the terminating condition for this approach requires that no hard constraints be violated by the current solution. It is easy to see that this combination of factors can lead to a condition where the hard constraints are violated in every iteration due to the unsoundness, and consequently, the specific terminating condition prevents the solver from terminating.

5 Related Work

Our work on solving weighted constraints is related to the inference problem in Markov Logic Networks (MLN) [27]. A MLN is a specification for various tasks in statistical relational learning [9], and essentially is a mixed system of hard and soft constraints. These constraints are used to express various information retrieval tasks like link prediction, social network modeling, collective classification, and others. A large body of work exists to solve these MLN constraints efficiently. Lazy inference techniques [26, 29] rely on the observation that most ground facts in the final solution to a MLN problem have a false value. If the constraints are in the form of Horn rules, this implies that most ground clauses are trivially true and do not need to be generated. These techniques start by assuming a default false value for most ground facts, gradually refining and grounding clauses as facts are determined to be true. In practice, these techniques tend towards being too eager in their grounding. Lazy, iterative ground-and-solve approaches [5, 28] are in the same class of techniques as our approach. They solve constraints lazily by iteratively grounding only those constraints that are violated by the current solution. However, the approach in [5] only solves hard constraints in a lazy manner while the soft constraints are effectively grounded upfront. Further, as described in Section 4, the terminating condition for these

approaches can lead to scenarios where the iterative algorithm exits the loop with an unsound solution or does not exit the loop at all. Lifted inference techniques [4, 20, 25] use approaches from first-order logic, like variable elimination, to simplify the system of weighted constraints, and can be used in conjunction with various standard ground-and-solve techniques for solving such constraints.

All of these previous techniques are approximate, and the final solution is neither guaranteed to be sound nor optimal. Though these approaches have been successfully used for statistical relational learning tasks like information retrieval, the nature of those applications allows approximations that are not acceptable for program analysis applications. In contrast, `LGS` generalizes and extends these works for solving weighted constraints, guaranteeing a sound and optimal final solution. `LGS` provides maximum flexibility; it allows the solving process to start with any set of initial grounded constraints, including an empty set, and uses lazy grounding for both, hard and soft constraints.

Additionally, existing MLN solvers do not effectively exploit the logical structure of Horn constraints. However, constraints in the form of Horn rules, whose least solution is desired, are common in program analysis and verification. The Guided strategy in `LGS` provides a powerful mechanism for exploiting the structure in such constraints to significantly accelerate the process of solving them. Given the generality of Horn rules, it is plausible that many information retrieval tasks could be expressed in this form to benefit from our approach.

An alternative to grounding followed by MaxSAT solving is to directly use a MaxSMT solver such as Z3 [2]. While Z3 currently does not scale to our program analysis applications, our techniques could be incorporated into it as Z3 already includes solvers for MaxSAT [2] and Datalog [10] that are needed by our approach. Conversely, it would be interesting to extend our approach to the richer theories that are supported by Z3.

6 Conclusion

We presented a technique for solving weighted constraints with applications to program analysis and verification. Unlike existing approaches, which ground the constraints eagerly and produce intractably large propositional instances to MaxSAT solvers, our technique grounds them lazily and iteratively, producing instances that can be handled by existing off-the-shelf MaxSAT solvers. We formalized our technique in a framework that generalizes and extends existing approaches, and we proved the optimality and soundness of the framework. We also identified an instance of the framework that accelerates the iterative process in the case where the constraints are in the form of Horn clauses, which is common for applications in program analysis and verification. This instance strikes a balance between the eager and lazy extremes by upfront grounding constraints that will necessarily be grounded during the iterative process. We showed that our technique scales significantly better than other approaches without sacrificing soundness for a suite of several real-world program analysis applications.

References

1. Beckman, N., Nori, A.: Probabilistic, modular and scalable inference of typestate specifications. In: PLDI (2011)
2. Bjørner, N., Phan, A.D.: νz: Maximal satisfaction with Z3. In: Proceedings of International Symposium on Symbolic Computation in Software Science (SCSS) (2014)
3. Blackburn, S.M., Garner, R., Hoffman, C., Khan, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J.E.B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The DaCapo benchmarks: Java benchmarking development and analysis. In: OOPSLA (2006)
4. Braz, R.D.S., Amir, E., Roth, D.: Lifted first-order probabilistic inference. In: IJCAI. pp. 1319–1325 (2005)
5. Chaganty, A., Lal, A., Nori, A., Rajamani, S.: Combining relational learning with SMT solvers using CEGAR. In: CAV (2013)
6. Dillig, I., Dillig, T., Aiken, A.: Automated error diagnosis using abductive inference. In: PLDI (2012)
7. Dillig, I., Dillig, T., McMillan, K., Aiken, A.: Minimum satisfying assignments for SMT. In: CAV (2012)
8. Domingos, P., Lowd, D.: Markov Logic: An Interface Layer for Artificial Intelligence. Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan & Claypool Publishers (2009)
9. Getoor, L., Taskar, B.: Introduction to Statistical Relational Learning (Adaptive Computation and Machine Learning). The MIT Press (2007)
10. Hoder, K., Bjørner, N., De Moura, L.: μz: An efficient engine for fixed points with constraints. In: CAV (2011)
11. Kok, S., Sumner, M., Richardson, M., Singla, P., Poon, H., Lowd, D., Domingos, P.: The alchemy system for statistical relational AI. Tech. rep., Department of Computer Science and Engineering, University of Washington, Seattle, WA (2007), `http://alchemy.cs.washington.edu`
12. Kremenek, T., Ng, A., Engler, D.: A factor graph model for software bug finding. In: IJCAI (2007)
13. Kremenek, T., Twohey, P., Back, G., Ng, A., Engler, D.: From uncertainty to belief: Inferring the specification within. In: OSDI (2006)
14. Larraz, D., Nimkar, K., Oliveras, A., Rodríguez-Carbonell, E., Rubio, A.: Proving non-termination using max-smt. In: CAV (2014)
15. Larraz, D., Oliveras, A., Rodríguez-Carbonell, E., Rubio, A.: Proving termination of imperative programs using Max-SMT. In: FMCAD (2013)
16. Lhoták, O., Hendren, L.: Context-sensitive points-to analysis: is it worth it? In: CC (2006)
17. Li, Y., Albarghouthi, A., Kincaid, Z., Gurfinkel, A., Chechik, M.: Symbolic optimization with SMT solvers. In: POPL (2014)
18. Livshits, B., Nori, A., Rajamani, S., Banerjee, A.: Merlin: specification inference for explicit information flow problems. In: PLDI (2009)
19. Marques-Silva, J., Heras, F., Janota, M., Previti, A., Belov, A.: On computing minimal correction subsets. In: IJCAI (2013)
20. Milch, B., Zettlemoyer, L.S., Kersting, K., Haimes, M., Kaelbling, L.P.: Lifted probabilistic inference with counting formulas. In: AAAi (2008)

21. Naik, M., Aiken, A., Whaley, J.: Effective static race detection for Java. In: PLDI (2006)
22. Niu, F., Ré, C., Doan, A., Shavlik, J.W.: Tuffy: Scaling up statistical inference in markov logic networks using an RDBMS. In: VLDB (2011)
23. Noessner, J., Niepert, M., Stuckenschmidt, H.: RockIt: Exploiting parallelism and symmetry for MAP inference in statistical relational models. In: AAAI (2013)
24. Papadimitriou, C.H.: Computational complexity. Addison-Wesley (1994)
25. Poole, D.: First-order probabilistic inference. In: IJCAI (2003)
26. Poon, H., Domingos, P., Sumner, M.: A general method for reducing the complexity of relational inference and its application to MCMC. In: AAAI (2008)
27. Richardson, M., Domingos, P.: Markov logic networks. Machine Learning 62(1-2) (2006)
28. Riedel, S.: Improving the accuracy and efficiency of MAP inference for Markov Logic. In: UAI (2008)
29. Singla, P., Domingos, P.: Memory-efficient inference in relational domains. In: AAAI (2006)
30. Sridharan, M., Bodík, R.: Refinement-based context-sensitive points-to analysis for Java. In: PLDI (2006)
31. Whaley, J., Lam, M.: Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In: PLDI (2004)
32. Zhang, X., Mangal, R., Grigore, R., Naik, M., Yang, H.: On abstraction refinement for program analyses in Datalog. In: PLDI (2014)

A Proof of Theorem 1

Theorem 3. (Soundness and Optimality of LGS) *For any weighted constraint system (H, S) where H is satisfiable, $\texttt{LGS}_\theta(H, S)$ produces equivalent results under $\theta = \mathsf{Eager}$, $\theta = \mathsf{Lazy}$, $\theta = \mathsf{Guided}$, and $\theta = \mathsf{Intermediate}$.*

Proof. It suffices to show that $\texttt{LGS}_{\theta=\mathsf{Intermediate}}(H, S) = \texttt{LGS}_{\theta=\mathsf{Eager}}(H, S)$, since $\theta = \mathsf{Lazy}$ and $\theta = \mathsf{Guided}$ are special cases of $\theta = \mathsf{Intermediate}$ that use different subsets of the fully ground set as the initial grounding.

First, observe that $\texttt{LGS}_{\theta=\mathsf{Intermediate}}(H, S)$ terminates: in each iteration of the loop on line 7 of Algorithm 1, it must be the case that at least one new hard clause is added to ϕ or at least one new soft clause is added to ψ, because otherwise the condition on line 13 will hold and the loop will be exited.

Now, suppose that in last iteration of the loop on line 7 for computing $\texttt{LGS}_{\theta=\mathsf{Intermediate}}(H, S)$, we have:
(1) $gc_1 = hgc_1 \cup sgc_1$ is the set of hard and soft clauses accumulated in (ϕ, ψ) so far (line 10);
(2) $Q_1 = \emptyset$ and $w_1 = 0$ (line 6), or $Q_1 = \texttt{MaxSAT}(hgc_1, sgc_1)$ and its weight is w_1 (lines 11 and 12);
(3) $gc_2 = hgc_2 \cup sgc_2$ is the set of all hard and soft clauses that are violated by Q_1 (lines 8 and 9);
(4) $Q_2 = \texttt{MaxSAT}(hgc_1 \cup hgc_2, sgc_1 \cup sgc_2)$ and its weight is w_2 (lines 11 and 12); and the condition on line 13 holds as this is the last iteration:
(5) $w_1 = w_2$ and $hgc_2 = \emptyset$.
Then, the result of $\texttt{LGS}_{\theta=\mathsf{Intermediate}}(H, S)$ is Q_1. On the other hand, the result of $\texttt{LGS}_{\theta=\mathsf{Eager}}(H, S)$ is:
(6) $Q_f = \texttt{MaxSAT}(hgc_f, sgc_f)$ (line 5 of Alg. 1) where:
(7) $gc_f = hgc_f \cup sgc_f$ is the set of fully grounded hard and soft clauses (line 5 of Alg. 2).
Thus, it suffices to show that Q_1 and Q_f are equivalent. We first extend the function `Weight` (Figure 6) to hard clauses, yielding $-\infty$ if any such clause is violated:

$$\texttt{W} \;=\; \lambda(Q, hgc \;\cup\; sgc).\; \textit{if}\; (\exists \rho \in hgc : Q \not\models \rho)\; \textit{then}\; -\infty \;\; \textit{else}\; \texttt{Weight}(Q, sgc)$$

Define $gc_m = gc_2 \setminus gc_1$.
(8) For any Q, we have:

$$\begin{array}{rcll} \texttt{W}(Q, gc_1 \cup gc_2) & = & \texttt{W}(Q, gc_1) + \texttt{W}(Q, gc_m) & \\ & = & \texttt{W}(Q, gc_1) + \texttt{W}(Q, hgc_2 \setminus hgc_1) + \texttt{W}(Q, sgc_2 \setminus sgc_1) & \\ & = & \texttt{W}(Q, gc_1) + \texttt{W}(Q, sgc_2 \setminus sgc_1) & [a] \\ & \geq & \texttt{W}(Q, gc_1) & [b] \end{array}$$

where [a] follows from (5), and [b] from $\texttt{W}(Q, sgc_2) >= 0$ (i.e., soft clauses do not have negative weights). Instantiating (8) with Q_1, we have: (9): $\texttt{W}(Q_1, gc_1 \cup gc_2) \geq \texttt{W}(Q_1, gc_1)$. Combining (2), (4), and (5), we have: (10): $\texttt{W}(Q_1, gc_1) = \texttt{W}(Q_2, gc_1 \cup$

gc_2). Combining (9) and (10), we have: (11)$\mathtt{W}(Q_1, gc_1 \cup gc_2) \geq \mathtt{W}(Q_2, gc_1 \cup gc_2)$. This means Q_1 is a better solution than Q_2 on $gc_1 \cup gc_2$. But from (4), we have that Q_2 is an optimum solution to $gc_1 \cup gc_2$, so we have: (12): Q_1 is also an optimum solution to $gc_1 \cup gc_2$.

It remains to show that Q_1 is also an optimum solution to the set of fully grounded hard and soft clauses gc_f, from which it will follow that Q_1 and Q_f are equivalent. Define $gc_r = gc_f \setminus (gc_1 \cup gc_2)$. For any Q, we have:

$$\begin{array}{rcl} \mathtt{W}(Q, gc_f) & = & \mathtt{W}(Q, gc_1 \cup gc_2 \cup gc_r) \\ & = & \mathtt{W}(Q, gc_1 \cup gc_2) + \mathtt{W}(Q, gc_r) \\ & \leq & \mathtt{W}(Q_1, gc_1 \cup gc_2) + \mathtt{W}(Q, gc_r) \quad [c] \\ & \leq & \mathtt{W}(Q_1, gc_1 \cup gc_2) + \mathtt{W}(Q_1, gc_r) \; [d] \\ & = & \mathtt{W}(Q_1, gc_1 \cup gc_2 \cup gc_r) \\ & = & \mathtt{W}(Q_1, gc_f) \end{array}$$

i.e. $\forall Q, \mathtt{W}(Q, gc_f) \leq \mathtt{W}(Q_1, gc_f)$, proving that Q_1 is an optimum solution to gc_f. Inequality [c] follows from (11), that is, Q_1 is an optimum solution to $gc_1 \cup gc_2$. Inequality [d] holds because from (3), all clauses that Q_1 possibly violates are in gc_2, whence Q_1 satisfies all clauses in gc_r, whence $\mathtt{W}(Q, gc_r) \leq \mathtt{W}(Q_1, gc_r)$. □

B Proof of Theorem 2

Theorem 4. (Optimal Initial Grounding for Horn Rules) *If a weighted constraint system is constituted of a set of hard constraints H, each of which is a Horn rule $\bigwedge_{i=1}^{n} t_i \Rightarrow t_0$, whose least solution is desired:*

$$G = \mathsf{lfp}\ \lambda G'.\ G' \cup \{\ [\![t_0]\!]_\sigma \mid (\textstyle\bigwedge_{i=1}^{n} t_i \Rightarrow t_0) \in H \text{ and } \forall i \in [1..n] \colon [\![t_i]\!]_\sigma \in G'\ \},$$

then for such a system, (a) $\mathtt{LGS}_{\theta=\mathsf{Lazy}}(H, \emptyset)$ grounds at least $|G|$ clauses, and (b) $\mathtt{LGS}_{\theta=\mathsf{Guided}}(H, \emptyset)$ with the initial grounding ϕ does not ground any more clauses:

$$\phi = \textstyle\bigwedge\{\ \bigvee_{i=1}^{n} \neg [\![t_i]\!]_\sigma \vee [\![t_0]\!]_\sigma \mid (\bigwedge_{i=1}^{n} t_i \Rightarrow t_0) \in H \text{ and } \forall i \in [0..n] \colon [\![t_i]\!]_\sigma \in G\ \}.$$

Proof. To prove (a), we will show that for each $g \in G$, $\mathtt{LGS}_{\theta=\mathsf{Lazy}}(H, \emptyset)$ must ground some clause with g on the r.h.s. Let the sequence of sets of clauses grounded in the iterations of this procedure be $C_1, ..., C_n$. Then, we have:
Proposition (1): each clause $\bigwedge_{i=1}^{m} g_i \Rightarrow g'$ in any Q_j was added because the previous solution set all g_i to true and g' to false. This follows from the assumption that all rules in H are Horn rules. Let $x \in [1..n]$ be the earliest iteration in whose solution g was set to true. Then, we claim that g must be on the r.h.s. of some clause ρ in Q_x. Suppose for the sake of contradiction that no clause in Q_x has g on the r.h.s. Then, it must be the case that there is some clause ρ' in Q_x where g is on the l.h.s. (the `MaxSAT` procedure will not set variables to true that do not even appear in any clause in Q_x). Suppose clause ρ' was added in some iteration $y < x$. Applying proposition (1) above to clause ρ' and $j = x$, it must be that g was true in the solution to iteration y, contradicting the assumption above that x was the earliest iteration in whose solution g was set to true.

To prove (b), suppose $\texttt{LGS}_{\theta=\mathsf{Guided}}(H, \emptyset)$ grounds an additional clause, that is, there exists a $(\bigwedge_{i=1}^{n} t_i \Rightarrow t_0) \in H$ and a σ such that $G \not\models \bigvee_{i=1}^{n} \neg[\![t_i]\!]_\sigma \vee [\![t_0]\!]_\sigma$. The only way by which this can hold is if $\forall i \in [1..n] : [\![t_i]\!]_\sigma \in G$ and $[\![t_0]\!]_\sigma \notin G$, but this contradicts the definition of G. □

C Additional Experiments

	brief description	# classes		# methods		bytecode (KB)		source (KLOC)	
		app	total	app	total	app	total	app	total
`ftp`	Apache FTP server	93	414	471	2,206	29	118	13	130
`hedc`	web crawler from ETH	44	353	230	2,134	16	140	6	153
`weblech`	website download/mirror tool	11	576	78	3,326	6	208	12	194
`luindex`	document indexing and search tool	206	619	1,390	3,732	102	235	39	190

Table 6. Characteristics of our benchmark programs. Columns "total" and "app" report numbers with and without counting Java standard library code, respectively.

analysis	program	$\lvert EDB\rvert$ ($\times 10^6$)	total ground clauses	# iterations		total time (hh:mm)		# ground clauses($\times 10^6$)		$\lvert IDB\rvert$ ($\times 10^6$)
				Lazy	Guided	Lazy	Guided	Lazy	Guided	
polysite	**hedc**	6.6	$6.2\text{x}10^{28}$	145	7	4:48	0:30	4.7	4.8	3.8
	ftp	3.9	$1.7\text{x}10^{28}$	168	7	3:31	0:20	2.8	3.0	2.3
	weblech	8.7	$3.0\text{x}19^{29}$	237	7	13:11	0:54	7.7	8.6	6.0
	luindex	11	$1.2\text{x}10^{30}$	216	8	18:41	1:18	11	12	8.4
downcast	**hedc**	6.6	$6.2\text{x}10^{28}$	140	8	5:01	0:34	4.9	5.1	4.0
	ftp	3.9	$1.6\text{x}10^{28}$	171	8	3:55	0:23	3.0	3.2	2.4
	weblech	8.7	$3.0\text{x}10^{29}$	254	9	16:18	1:07	8.5	9.6	6.5
	luindex	11	$1.2\text{x}10^{30}$	186	9	24:43	1:54	11	13	9.2
datarace	**hedc**	0.5	$1.9\text{x}10^{24}$	354	6	1:55	0:06	0.8	0.9	0.7
	ftp	0.4	$3.7\text{x}10^{23}$	463	5	7:53	0:08	1.2	1.4	1.0
	weblech	1.2	$4.4\text{x}10^{24}$	416	6	1:59	0:07	0.6	0.9	0.5
	luindex	1.9	1.610^{25}	481	7	4:07	0:12	0.6	1.1	0.5

Table 7. Evaluation of LGS on program analysis applications.