Efficient Grounding with Bounds using Bit Vectors

Lucas Van Laer[0009−0001−3941−1697]1,2,3,4, Bart Bogaerts[0000−0003−3460−4251]1,2,4, and Joost Vennekens[0000−0002−0791−0176]4

[1] KU Leuven, Dept. Of Computer Science,
[2] Leuven.AI – KU Leuven institute for AI, B-3000 Leuven, Belgium
[3] Flanders Make – DTAI-FET
[4] Vrije Universiteit Brussel, Brussel, Belgium
{lucas.vanlaer, bart.bogaerts}@kuleuven.be, Joost.Vennekens@vub.be

Abstract. Systems for declarative problem-solving often include a "grounding" step to remove first-order variables in order to produce a purely propositional representation. Because a naive implementation of this grounding step can be highly inefficient, algorithms were created to derive new information from the problem specification which can be used to reduce the size of the grounding. In this paper, we show how to use this technique with bit vectors, thereby taking advantage of the fact that, on modern computers, logical operations on bit vectors can be executed at extreme speeds, and investigate some places where bit vectors may perform badly. We conduct an experimental analysis, which shows that bit vectors perform well on certain problems, but have limitations.

1 Introduction

Declarative problem-solving systems may handle a "high-level" problem specification by first translating it to a more "low-level" specification. This is a popular approach in areas such as Constraint Programming or Answer Set Programming, where the high-level languages use first-order variables to represent domain knowledge in a way which is independent of the size of the problem domain. In the translation to a low-level language, these first-order variables are removed by *grounding*, i.e., replacing them with domain elements in all possible ways, thereby introducing a dependency on the domain size. The resulting ground specification is typically many times larger than the high-level specification, which may make grounding challenging for large domain sizes, due to time and/or memory constraints.

Consider, for instance, the following formula in classical first-order logic (FO):

$$\forall x, y, z : edge(x, y) \wedge edge(y, z) \wedge edge(x, z) \Rightarrow p(x, y, z) \tag{1}$$

This formula expresses that p holds for all triangles in a graph. Its grounding is of size $O(n^3)$, with n the number of nodes in the graph. However, suppose the graph is already known, i.e., we are given a structure S with a domain D and

interpretation $edge^S \subseteq D^2$ and are only interested in models S' that extend this given S with an interpretation for the predicate p. In this case, the grounding can be reduced to a conjunction $\bigwedge_{(d,e,f)\in\Delta_S} p(d,e,f)$ over the set Δ_S of all tuples $(d,e,f) \in D^3$ for which $\{(d,e),(e,f),(f,d)\} \subseteq edge^S$. Previous work introduced an efficient method for computing such reduced groundings, using bit vectors to implement the required operations [21].

Suppose now that we are interested in the task of computing a subgraph $edge/2$ of a given supergraph $super/2$, such that all triangles in this subgraph belong to p. We can reuse (1), together with the additional constraint $\forall x, y : edge(x,y) \Rightarrow super(x,y)$. However, since $edge/2$ is now no longer given, we cannot compute Δ_S, which causes the method of [21] to ground (1) over all triples in D^3.

Earlier work [26] proposed a way to avoid this. The idea of this method is to maintain for each subformula $\phi(\boldsymbol{x})$ that occurs in the theory, an approximation (L,U) of the set S of all tuples $\boldsymbol{d} \in D^n$ for which $\phi(\boldsymbol{d})$ holds, i.e., $L \subseteq S \subseteq U$. Before the grounding is actually computed, there is a propagation phase in which bounds of one (sub-)formula are used to refine bounds of other (sub-)formulas. In the case of our example, the second constraint would introduce an upper bound $edge^S \subseteq super^S$ on the interpretation of $edge$, which would then be propagated to the first constraint (1), where it would then be used to reduce the grounding to $\bigwedge_{(d,e,f)\in\Delta'_S} edge(d,e) \wedge edge(e,f) \wedge edge(f,d) \Rightarrow p(d,e,f)$, with Δ'_S the set of triangles in $super^S$. Indeed, for all other tuples, no grounding is needed since the antecedent of the implication is already guaranteed to be false.

In the practical implementation of this method, each of the bounds L and U is represented by a binary decision diagram (BDD) for FO [13], which with every operation can double in size. As such approximation methods were needed to limit the memory usage. Moreover, since equivalence checking of two (first-order)- BDDs corresponds to equivalence checking of two FO formulas, approximations and heuristics must be used to find when the propagation method has reached a fixpoint.

In this paper, we examine whether the novel bit vector method of [21] can be used to replace the BDD operations in the earlier propagation methods of [26]. On the one hand, we might expect efficiency gains from the fact that the relatively complex BDD operations are replaced by simpler bit vector operations, that can benefit from the single instruction multiple data (SIMD) parallelism of modern hardware to efficiently execute large operations. In addition, also the difficulty of checking equivalence is avoided. On the other hand, however, bit vectors excel at executing a small number of large operations, whereas an iterative propagation method may instead lead to a large number of small operations. To examine this, we also propose a new variant of the original propagation method, which imposes additional structure on the order in which propagations can be executed.

In particular, we will proceed to define an approach in which top-down and bottom-up propagations are `combined`, as done by Wittocx et al. [26], but using bit vectors instead of BDDs. We then also consider a variant in which top-

down and bottom-up propagations are `separated` to impose additional structure. While this second approach greatly simplifies the propagation process, the imposed order may not always be optimal: in particular, for certain examples, this algorithm may end up essentially performing a step-by-step grounding during the propagation. We compare both of these approaches to the original methods in [26] and [21], as well as to the Answer Set Programming solver clingo [12].

2 Preliminaries

We consider classical first-order logic. A *vocabulary* Σ is a set of predicate and function symbols σ, each with an associated arity $n \geq 0$, denoted as σ/n. The symbols $\top$ and $\bot$ are used to represent true and false respectively. A *partial structure* S for a vocabulary Σ has a set of elements called its *domain* $D = domain(S)$, and assigns to each symbol $\sigma \in \Sigma$ an *interpretation* σ^S. If σ is a predicate symbol P/n, then P^S is a mapping from all tuples in D^n to $\{\mathbf{t}, \mathbf{f}, \mathbf{u}\}$ (where $\mathbf{u}$ stands for "unknown"). For a function symbol f/n, f^S is a mapping from D^{n+1} to $\{\mathbf{t}, \mathbf{f}, \mathbf{u}\}$ such that for each $\boldsymbol{d} \in D^n$, there exists at most one $e \in D$ for which $f^S(\boldsymbol{d}, e) = \mathbf{t}$. In this paper, we restrict attention to structures with a *finite* domain. Moreover, we assume that for each element $d \in domain(S)$ there exists a constant $c_d/0 \in \Sigma$ such that c_d^S is fixed to d. For convenience, we will often not distinguish between the domain element d and the constant c_d.

We call a partial structure S' an *extension* of another partial structure S if for all $\sigma \in \Sigma$ and all $\boldsymbol{d}$, if $\sigma^S(\boldsymbol{d}) \neq \mathbf{u}$, then $\sigma^{S'}(\boldsymbol{d}) = \sigma^S(\boldsymbol{d})$. A *(total) structure* is a partial structure in which all $\sigma^S(\boldsymbol{d}) \neq \mathbf{u}$ (and therefore it has no extensions except itself). In such a total structure S with domain D, the interpretation P^S of a predicate symbol P/n can be identified with a subset of D^n and the interpretation f^S of a function symbol f/n with a function $D^n \to D$, as usual.

A *term* is either a variable or a function symbol f/n applied to n terms. An *atom* is a predicate symbol P/n applied to n terms. For Φ a formula, $\boldsymbol{x}$ a tuple of variables and $\boldsymbol{t}$ a tuple of terms, $\Phi[\boldsymbol{t}/\boldsymbol{x}]$ denotes the result of replacing in Φ the free occurrences of each $x_i \in \boldsymbol{x}$ by the corresponding $t_i \in \boldsymbol{t}$. Formulas are constructed as usual by combining atoms with the logical operators $\forall, \exists, \neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$. A *sentence* is a formula without free variable and a *theory* is a set of sentences.

For Φ a formula and S a partial structure, we denote by Φ^S the truth value of Φ in S under the three-valued Kleene evaluation. Note that if S is total, then $\Phi^S \neq \mathbf{u}$; in this case, we also write $S \models \Phi$ to say that $\Phi^S = \mathbf{t}$. A *model* of a theory T is a total interpretation S such that $S \models \phi$ for all $\phi \in T$. Given a partial structure S and theory T, the *model expansion problem* $MX(T, S)$ is the task of producing all models of T that extend S [18]. With a slight abuse of notation, we also write $MX(T, S)$ to denote the set of these models. We write $(T, S) \models \Phi$ to say that $S' \models \Phi$ for all $S' \in MX(T, S)$

A *grounding* of a theory T w.r.t. a partial structure S is a theory T' such that $MX(T, S) = MX(T', S)$ and T' does not contain any variables. The *naive* grounding is obtained by replacing each $\forall x : \Phi$ by $\bigwedge_{d \in domain(S)} \Phi[c_d/x]$ and

each $\exists x$ by the corresponding disjunction. It is clear that the naive grounding is indeed a grounding, but in general, also other, smaller groundings exists.

3 Propagation using a combined approach

In this section, we present the first of two approaches we explore in this paper. We first give the theoretical foundations and a high-level algorithm, before discussing implementation issues.

3.1 Algorithm

When grounding T w.r.t. a partial structure S, the grounding of a formula $\Phi(\boldsymbol{x})$ for a tuple of domain elements $\boldsymbol{d}$ can be simplified to $\top$ (or $\bot$) if $\Phi(\boldsymbol{d})^S = \mathbf{t}$ (or $\mathbf{f}$, respectively). Only when $\Phi(\boldsymbol{d})^S = \mathbf{u}$, do we actually need to produce the instantiation $\Phi(\boldsymbol{d})$. To capture this idea, we introduce the following concepts.

Definition 1. *For a model expansion problem (T, S) and a formula $\Phi(\boldsymbol{x})$ with n free variables, the* certainly true set (ct-set) *of $\Phi(\boldsymbol{x})$, denoted as $\lceil \Phi(\boldsymbol{x}) \rceil_S^T$, consists of all tuples $\boldsymbol{d} \in domain(S)^n$ such that $(T, S) \models \Phi(\boldsymbol{d})$. Similarly, the* certainly false set (cf-set) *of Φ, denoted $\lfloor \Phi(\boldsymbol{x}) \rfloor_S^T$, consists of all $\boldsymbol{d}$ for which $(T, S) \models \neg\Phi(\boldsymbol{d})$. When (T, S) is clear from the context, we omit them from the notation.*

In the worst case, computing ct- and cf-sets is as difficult as solving the model expansion problem, so we will instead approximate them.

Definition 2. *In the context of a given model expansion problem (T, S), a pair (t, f) of sets of n-tuples of domain elements is called an* approximating pair *for a formula $\Phi(\boldsymbol{x})$ if $t \subseteq \lceil \Phi \rceil$ and $f \subseteq \lfloor \Phi \rfloor$. We call such a pair* total *if $t \cup f = domain(S)^n$.*

The central idea behind our approach is to associate to each subformula of T an approximating pair (such as $(\{\}, \{\})$ if no other information is available), and then iteratively refine the approximation pair associated to a formula $\Phi(\boldsymbol{x})$ by doing both bottom-up propagation of information about the approximating pairs of Φ's subformulas, and top-down propagations of information about the approximation pair of the superformula in which Φ appears. What exactly these propagations look like depends on the form of Φ. We start by considering the case of a conjunction.

Proposition 1. *For a conjunction, $\lceil \Phi_1 \wedge \Phi_2 \rceil = \lceil \Phi_1 \rceil \cap \lceil \Phi_2 \rceil$ and $\lfloor \Phi_1 \wedge \Phi_2 \rfloor = \lfloor \Phi_1 \rfloor \cup \lfloor \Phi_2 \rfloor$. If (t_1, f_1) and (t_2, f_2) are approximating pairs of Φ_1 and Φ_2, then $(t_1 \cap t_2, f_1 \cup f_2)$ is an approximating pair of $\Phi_1 \wedge \Phi_2$. If (t, f) and (t_1, f_1) are approximating pairs of $\Phi_1 \wedge \Phi_2$ and Φ_1, then $(t, f \cap t_1)$ is an approximating pair of Φ_2.*

Similarly, the case for negation is as follows:

Proposition 2. *For a negation,* $\lceil \neg \Phi \rceil = \lfloor \Phi \rfloor$ *and* $\lfloor \neg \Phi \rfloor = \lceil \Phi \rceil$*. If* (t, f) *is an approximating pair of* $\neg \Phi$*, then* (f, t) *is an approximating pair of* Φ*, and vice versa.*

Together, the propositions for conjunction and negation obviously imply a similar proposition for disjunction.

The above two propositions can be used to compute, e.g., the ct/cf-sets of $p(x, y) \wedge \neg q(x, y)$ from those of $p(x, y)$ and $q(x, y)$. However, computing the ct/cf-sets of formulas such as $p(x, y) \wedge r(x)$ or $p(x, y) \wedge q(y, x)$ is not immediately possible due to the mismatch between the number and order of the variables in the subformulas. In [21], a number of transformation were defined that can be used to eliminate such mismatches by changing the number of variables over which a ct/cf-set ranges, or the order of these variables. From now on, we will just assume that such operations are applied whenever necessary to ensure a match between the variables of different subformula.

Finally, we also need to consider the case of quantifications.

Proposition 3. *For a universal quantifier,* $\lceil \forall x : \Phi \rceil = \{(d_1, \ldots, d_n) \in domain(S)^n \mid$ *for all* $d \in domain(S)$*,* $(d, d_1, \ldots, d_n) \in \lceil \Phi \rceil\}$ *and* $\lfloor \forall x : \Phi \rfloor = \{(d_1, \ldots, d_n) \in domain(S)^n \mid$ *there exists* $d \in domain(S)$*,* $(d, d_1, \ldots, d_n) \in \lfloor \Phi \rfloor\}$*. If* (t, f) *is an approximating pair of* Φ*, then the following* (t', f') *is an approximating pair for* $\forall x : \Phi$*:*

- $t' = \{(d_1, \ldots, d_n) \in domain(S)^n \mid$ *for all* $d \in domain(S)$*,* $(d, d_1, \ldots, d_n) \in t\}$
- $f' = \{(d_1, \ldots, d_n) \in domain(S)^n \mid$ *there is a* $d \in domain(S)$*,* $(d, d_1, \ldots, d_n) \in f\}$

Conversely, if (t', f') *is an approximating pair of* $\forall x : \Phi$*, then the following* (t, f) *is an approximating pair for* Φ*:*

- $t = \{(d, \boldsymbol{d}) \mid d \in domain(S)$ *and* $\boldsymbol{d} \in t'\}$
- $f = \{\}$

The case for existential quantification can again be derived from this proposition and the one for negation.

We can use the above propositions to add additional elements to the sets t, f of an approximating pair (t, f) for a formula Φ by looking at the approximating pairs of the subformulas of Φ and of the superformula in which Φ appears.

Algorithm 1 shows how to build a concrete propagation algorithm based on this idea. As its central datastructure, the algorithm uses a mapping α of each subformula that appears anywhere in T to an approximating pair (t, f). Most of the $\alpha(\Phi)$ are initialised to $(\{\}, \{\})$, with two exceptions: for all sentences $\Phi \in T$, $\alpha(\Phi)$ that Φ must be true in all models of (T, S); for all atoms $P(\boldsymbol{t})$, $\alpha(\Phi)$ indicates that this atom must be true/false for all tuples $\boldsymbol{d}$ such that $P^S(\boldsymbol{d}) = \mathbf{t}/\mathbf{f}$.

To keep track of the propagation steps, a queue is used, to which a formula Φ is added whenever $\alpha(\Phi)$ changes. In each iteration of the algorithm, a formula Φ whose approximating pair has recently changed is taken from the queue, and the algorithm tries to propagate the recent changes to the subformulas and

superformula of Φ. If any of those propagations actually changes some $\alpha(\Psi)$, then Ψ is requeued. A special case occurs when a propagation is made that changes $\alpha(P(\boldsymbol{t}))$ for some atom $P(\boldsymbol{t})$. In this case, the case is also recorded in the interpretation P^s of P in the structure S, and all atoms of the form $P(\boldsymbol{t}')$ must be enqueued.

Algorithm 1 Propagation algorithm

Require: A model expansion problem T, S
  $\alpha \leftarrow \{\phi \mapsto (\{\}, \{\}) \mid \phi$ is subformula appearing in $T\}$
  queue $\leftarrow []$
  for each sentence $\Phi \in T$ **do**
    $\alpha(\Phi) = (\mathbf{t}, \{\})$ and *queue.add*(Φ)
  end for
  for each atom $P(\boldsymbol{t}) \in T$ **do**
    $\alpha(P(\boldsymbol{t})) = (\{\boldsymbol{d} \mid P^S(\boldsymbol{d}) = \mathbf{t}\}, \{\boldsymbol{d} \mid P^S(\boldsymbol{d}) = \mathbf{f}\})$ and *queue.add*$(P(\boldsymbol{t}))$
  end for
  while *queue* is not empty **do**
    $\Phi \leftarrow$ *queue.pop*()
    if Φ is an atom $P(\boldsymbol{t})$ **then**
      Extend P^S with $\alpha(P(\boldsymbol{t})) = (t, f)$, setting $P^S(\boldsymbol{d}) = \mathbf{t}/\mathbf{f}$ for each $\boldsymbol{d} \in t/f$
      Add all other atoms that contain predicate P to *queue* and change their approximating pair in α to use P^S
    end if
    for each sub-/superformula Ψ of Φ **do**
      Propagate $\alpha(\Phi)$ to $\alpha(\Psi)$ and do *queue.add*(Ψ) if changes made
    end for
  end while

Example 1. Consider the partial structure S with domain $\{1, 2, 3, 4, 5\}$ and p^S and q^S both mapping all elements to $\mathbf{u}$, together with the theory T consisting of the following sentences:

$$\forall x : p(x) \vee q(x). \tag{2}$$
$$\neg p(4) \Rightarrow \neg p(5). \tag{3}$$
$$p(1) \Rightarrow p(2). \tag{4}$$
$$p(1). \tag{5}$$
$$\neg p(4). \tag{6}$$

If sentence (2) is popped from the queue first, its approximating pair $(\{()\}, \emptyset)$ will be propagated to $\alpha(p(x) \vee q(x)) = (\{1, 2, 3, 4, 5\}, \emptyset)$. For (3) and (4) nothing can be propagated yet. For (5), the approximating pair $\alpha(p)$ will be changed to $(\{1\}, \emptyset)$, which triggers a change to the partial structure, i.e. $p^S(1) = \mathbf{t}$, and also causes all other atoms for p to be added to the queue. When the occurrence of $p(1)$ in (4) is selected, the approximating $\alpha(p)$ is extended to $(\{(1, 2)\}, \emptyset)$.

From (6), it will be propagated that $\alpha(p) = (\{1,2\},\{4\})$, which will trigger a propagation in (3) that results in $\alpha(p) = (\{1,2\},\{4,5\})$. In assertion 2, the changes made to $\alpha(p)$ will trigger a propagation $\alpha(q) = (\{4,5\},\emptyset)$. After the remainder of the queue has been popped without additional propagations, the algorithm ends with a partial structure S in which:

- $p^S = \{1 \mapsto \mathbf{t}, 2 \mapsto \mathbf{t}, 3 \mapsto \mathbf{u}, 4 \mapsto \mathbf{f}, 5 \mapsto \mathbf{f}\}$
- $q^S = \{1 \mapsto \mathbf{u}, 2 \mapsto \mathbf{u}, 3 \mapsto \mathbf{u}, 4 \mapsto \mathbf{t}, 5 \mapsto \mathbf{t}\}$

The idea is now that, for any $\alpha(\Phi(\boldsymbol{x})) = (t, f)$, we can ground Φ to $\top$ for all $\boldsymbol{d} \in t$ and to $\bot$ for all $\boldsymbol{d} \in f$. For the remaining tuples $\boldsymbol{d} \notin t \cup f$, we still need to produce $\boldsymbol{d}$. To formally prove that this is correct, we first introduce the following concept.

Definition 3. *For a model expansion problem (T, S) in vocabulary Σ, let Σ^T be the vocabulary that extends T with two fresh n-ary predicate symbols P^t_Φ and P^f_Φ for each subformula Φ with n free variables that appears in T. We define the* approximative theory *T' of T as the result of recursively replacing each subformula $\Phi(\boldsymbol{x})$ in T by $\neg P^f_\Phi(\boldsymbol{x}) \wedge (\Phi(\boldsymbol{x}) \vee P^t_\Phi(\boldsymbol{x}))$.*

The idea is now just to fill in the truth values obtained in each $\alpha(\Phi) = (t, f)$ by using t as an interpretation of P^t_Φ and f as an interpretation for P^f_Φ, so that, for instance, $\neg P^f_\Phi(\boldsymbol{x}) \wedge (\Phi(\boldsymbol{x}) \vee P^t_\Phi(\boldsymbol{x})$ reduces to $\mathbf{t}$ for each $\boldsymbol{d} \in t$. However, we must be careful here: if we just fill in $\alpha(\Phi)$ everywhere, then because for each top-level sentence $\Phi \in T$, $\alpha(\Phi) = (\{()\},\emptyset)$, we would actually reduce the entire theory to $\top$. Clearly, the resulting model expansion problem $(\top, S)$ will no longer be equivalent to the original (T, S).

The issue here is that during propagation we make the assumption that all sentences in T will be true in order to perform top-down propagations. When doing the actual grounding, however, we can no longer make this assumption because then we will have nothing to ground. Instead, we can only rely on the information that could be propagated to the interpretation of predicates/functions in S' and whatever propagations can be derived from this in a purely bottom-up manner (i.e., without assuming that all sentences will be true). The following definition formalises this idea.

Definition 4. *For a model expansion problem (T, S) in vocabulary Σ, let α be a mapping from each of the subformulas of T to an approximating pair (t, f). A function α' is called a* safe approximation mapping *of α if it maps each subformula of T to an approximating pair for the model expansion problem $(\{\}, S')$ where each $P^{S'}$ is the result of extending P^S by setting $P^{S'}(\boldsymbol{d}) = \mathbf{t}/\mathbf{f}$ for each $\boldsymbol{d} \in t/f$ with $\alpha(P) = (t, f)$.*

Now, we can prove the desired result.

Proposition 4. *Consider a model expansion problem (T, S) and a mapping α of the subformulas of T to pairs (t, f) of sets of atoms. Let α' be the safe approximation mapping of α, T' the approximative theory of T and let S' be the*

result of extending S with a total interpretation for each of the new predicates introduced in T' such that $(P^t_\Phi)^S = t$ and $(P^f_\Phi)^S = f$ with $\alpha(\Phi) = (t, f)$. If each $\alpha(\Phi)$ is an approximating pair of the model expansion problem $(\{\}, S)$, then $MX(T, S) = MX(T', S')$ (modulo projection of each structure onto the original vocabulary Σ).

Putting all of this together, we can now produce a reduced grounding by recursively replacing each formula Φ by $\neg P^f_\Phi \wedge (P^t_\Phi \vee \Phi)$ and filling in the propagated truth values by assigned $(P^t_\Phi, P^f_\Phi) = \alpha(\Phi)$ for a safe approximation mapping α.

Example 1 (continued). From the partial structure S that we computed previously, we can obtain a safe approximation mapping by looking at the model expansion problem $(\{\}, S)$. Here, we find that all the sentences (3), (4), (5) and (6) have $(\mathbf{t}, \emptyset)$ as an approximation pair w.r.t. $(\{\}, S)$ and therefore do not need to be grounded. Sentence (2) only has $(\emptyset, \emptyset)$ as approximating pair, but for its immediate subformula $p(x) \vee q(x)$ we can find $(\{1, 2, 4, 5\}, \emptyset)$. We can then rewrite the theory into $\forall x : x \in \{1, 2, 4, 5\} \vee (p(x) \vee q(x))$, and eventually produce the grounding $p(3) \vee q(3)$.

We call this the `combined` approach, because whenever a formula is selected, its approximating pair is used to do both top-down and bottom-up propagation. This is essentially the same general approach as proposed by [26], but we propose a different implementation technique.

3.2 Implementation

While [26] uses Binary Decision Diagrams to represent ct-/cf sets in a syntactic way, we examine here an approach in which these sets are represented by bit vectors, similar to [21]. Essentially, the idea is that a ct/cf-set for a formula with n free variables in a structure S can be represented by a vector of $domain(S)^n$ bits, where each bit represents whether one particular n-tuple $\boldsymbol{d}$ of domain elements belongs to the set or not. Obviously, this requires some careful bookkeeping to keep track of which bit positions correspond to which tuples. Propagations between sub-/superformulas which have the same number of free variables in the same order can then be done by bitwise boolean operations, which are executed very efficiently on modern hardware.

As mentioned above, when the free variables of different subformulas do not match precisely, reordering of the bits or a change in the size of the bit vectors may be required (the detailed operations are described in [21]). However, these operations are much less efficient. As an optimisation, when computing the conjunction of two formulas whose free variables do not match precisely, we use the *sort-merge* join algorithm, to avoid the costly resizing/reordering of bit vectors.

The bit vector representation can be inefficient for sets which are either very sparse or very dense. To avoid this, we do not use normal bit vectors, but instead make use of *roaring bitmaps* [15]. These provide a compressed representation for

bit vectors in which sparse segments are represented using an array of numbers and dense segments using an array of intervals. To further optimise the representation, we add to each roaring bitmap a single boolean which indicates whether the bitmap represents the actual set itself (i.e., a 1 in the bitmap indicates membership) or the complement of the set (i.e., a 0 in the bitmap indicates membership of the actual set). In this way, when propagating across negation, we avoid having to change all 1s into 0s and vice versa, and instead just flip this single bit. Lastly, we use only a single set for representing total approximating pairs, where a 1 would indicate membership in the certainly true part and 0 in the certainly false part.

4 Separating bottom-up and top-down propagations

The approach presented in the previous section mixes top-down and bottom-up propagations: whenever the approximating pair of a formula is updated, this change can be propagated to both its subformulas and to its superformula. However, one of the key strengths of the bitmap representation is that it is good for executing a small number of large operations. Therefore, it might make sense not to propagate information as soon as it becomes available, but to instead delay propagation in order to first gather multiple updates to the same approximating pair and then propagate all of these in a single big propagation step.

As we recall from the discussion of Proposition 4, information that is propagated in a top-down down way only becomes relevant once it reaches the atom level. For instance, for a theory $T = \{\forall x : \neg(p(x) \wedge q(x))\}$ and a structure S in which p and q are entirely unknown, a top-down propagation can tell us that $p(x) \wedge q(x)$ must be universally false, but this information does not help to reduce the grounding in any way. Indeed, a safe approximation mapping will have to discard this information, which will cause a full grounding to be produced any way, so the effort of doing the top-down propagation is wasted and we would like to avoid doing it.

To this end, we introduce the following concept of a *ct-target* and a *cf-target*. Intuitively, if a formula is a ct/cf-target, this means that its ct/cf-set can be updated in a top-down propagation.

Definition 5. *Let Φ be a formula and Ψ its superformula. We define ct- and cf-target by simultaneous top-down induction:*

- *If Ψ does not exist (i.e., Φ is a top-level sentence), then Φ is a ct-target*
- *If Ψ is a universal quantification or a conjunction, then if Ψ is a ct-target, so is Φ*
- *If Ψ is an existential quantification or a disjunction, then if Ψ is a cf-target, so is Φ*
- *If Ψ is a negation, then if Ψ is a ct-target (or cf-target), then Φ is a cf-target (ct-target, resp.).*

Whether a formula is ct-/cf-target depends entirely on the structure of the top-level sentence it appears in, and is easy to check. The idea is now that we only

want to do top-down propagations for formulas that contain at least one atom that is a ct-/cf-target. As discussed above, none of the atoms in $\forall x : \neg(p(x) \wedge q(x))$ are ct-/cf-targets, so we will not do top-down propagation here. By contrast, for instance in $\forall x : \neg(p(x) \vee q(x))$, both atoms are cf-targets, so here we will propagate in a top-down way that both p and q must be universally false.

There are also cases in which top-down propagation to the atom-level is possible, even when the atom is not a ct-/cf-target. For instance, if we consider again $\forall x : \neg(p(x) \wedge q(x))$ but now with a structure S with domain $\{0, 1\}$ in which p is $\mathbf{t}$ for 0 and $\mathbf{f}$ for 1. In this case, top-down propagation would be able to reach the atom $q(x)$, producing $\alpha(q) = (\emptyset, \{0\})$, and the resulting safe approximation mapping would tell us that $\alpha(\neg(p(x) \wedge q(x))) = (\{0, 1\}, \emptyset)$, so the empty grounding can be produced. In general, if instead of $p(x)$ there would be a more complex formula $\Phi(x)$, bottom-up propagation might be required to figure out that this $\Phi(x)$ has a non-empty ct-set, which would allow for top-down propagation to $q(x)$.

To do this bottom-up propagation in a way that enables easy detection of ct-/cf-target, we make use of the idea of *quantification splitting* from [21]. Quantification splitting transforms a given formula Φ into a set of new formulas based on a partial structure S, or more precisely, based on which symbols have a total interpretation in S. If Φ is of the form $\forall x : \Psi$, the transformation works as follows.

Let $\gamma_1, \ldots, \gamma_m$ be the maximal subformulas of Φ that contain only symbols that have a total interpretation in S. The original formula Φ is transformed into 2^m new formulas of the form $\forall \boldsymbol{x} : (\bigwedge_{i=1}^{m} (\neg)\gamma_i) \Rightarrow \Psi'$, in which each γ_i appears either negated or not. Here, Ψ' is the result of replacing each subformula γ_i with $\top$ or $\bot$ depending on whether γ_i appears negated or not. The conjunction $\bigwedge_{i=1}^{m} (\neg)\gamma_i)$ is called the quantifier's *guard*.

The idea behind this transformation is that the ct-/cf-sets of the guards can be completely computed using bottom-up propagation, so we can equivalently view each formula $\forall \boldsymbol{x} : (\bigwedge_{i=1}^{m} (\neg)\gamma_i) \Rightarrow \Psi'$ as a $\forall \boldsymbol{x} \in G : \Psi'$ where G is the ct-set $\lceil \bigwedge_{i=1}^{m} (\neg)\gamma_i \rceil$ of the guard. In other words, the result of quantification splitting a formula Φ is essentially to remove all symbols that are fully known in S from Φ. To handle atoms $P(\boldsymbol{t})$ where the P^S is not fully known, we replace $P(\boldsymbol{t})$ with $\neg P^f(\boldsymbol{t}) \wedge (\Phi(\boldsymbol{t}) \vee P^t(\boldsymbol{t}))$ where P^t and P^f have the interpretations $\{\boldsymbol{d} \mid P^S(\boldsymbol{d}) = \mathbf{t}\}$ and $\{\boldsymbol{d} \mid P^S(\boldsymbol{d}) = \mathbf{f}\}$ respectively.

If the formula Φ is an existentially quantified $\exists \Psi$, the result of quantification splitting is a disjunction of formulas of the form $\exists \boldsymbol{x} : (\bigwedge_{i=1}^{m} (\neg)\gamma_i) \wedge \Psi'$. Details can be found in [21]. Lastly, to produce ct-/cf-targets for non-quantifier formulas without free variables it suffices to simplify the formulas using the interpretations in S, i.e. the formula $p() \vee t()$ can be simplified to $t()$ if $S^p = \{() \mapsto \mathbf{f}\}$.

We can combine top-down propagation to ct-/cf-targets with bottom-up quantification splitting as shown in Algorithm 2, which we refer to as the `separated` approach, because top-down and bottom-up propagation are now split into two different phases of the algorithm.

Algorithm 2 Algorithm of `separated` implementation

Require: S a structure over Σ, T a theory over Σ
  while fixpoint not reached **do**
    $T \leftarrow$ *quantification_splitting*(S, T)
    for sentence $\Phi \in T$ **do**
      for atom $p(\boldsymbol{t})$ in Φ which is a ct-target or cf-target **do**
        Update S with top-down propagation from Φ to p
      end for
    end for
  end while

Example 2. Consider the following theory T:

$$\forall x : p(x) \vee q(x). \tag{7}$$
$$p(1) \Rightarrow q(3). \tag{8}$$

and the structure S with domain $\{1, 2, 3\}$ in which q is universally unknown and $p^S = \{1 \mapsto \mathbf{t}, 2 \mapsto \mathbf{f}, 3 \mapsto \mathbf{t}\}$.

Quantification splitting will split the first formula into two:

$$\forall x \in \{1, 3\} : \top \tag{9}$$
$$\forall x \in \{2\} : q(x). \tag{10}$$
$$q(3). \tag{11}$$

Here, the first formula is trivially satisfied, so it can be removed. In the second formula, $q(x)$ is a ct-target, so it can be propagated that $q^{S'}(2) = \mathbf{t}$. Similarly, $q(3)$ is also a ct-target in the third formula, so also $q^{S'}(3) = \mathbf{t}$ is propagated.

5 The SLI system

Both the `combined` and `separated` approaches were implemented in the SLI[5] reasoning engine for the FO(·) language. We compared these two implementations with SLI, IDP3 [7] only using BDDs, and clingo [12]. The IDP3 system contains different implementations and integrations with different systems for finding solutions, in order to fairly compare the BDD approach and bit vector approach we configured IDP3 to only use the BDD-based propagations.

While our presentation in this paper so far has only considered first-order logic FO, the language FO(·) contains also a number of language extension, the most important being a construct for representing inductive definitions in a rule-based manner [10]. The grounding methods implemented in SLI handle this extension as well, by performing propagation on Clark's completion [5] of the rules in an inductive definition.

The SLI system uses the Z3 solver [9] as a back-end, grounding its FO(·) input to the SMT-LIB format.

[5] `https://gitlab.com/sli-lib/SLI`

6 Evaluation

We evaluated these implementations using the DIRT [22] benchmark suite, together with a new benchmark, the full benchmark results and steps to reproduce them are available online.[6] The new benchmark is called `ImplicationLadder`, it consists of two unary predicates p and q and the sentences $\forall x : p(x) \Rightarrow q(x)$, $\forall x : p(x) \Rightarrow p(x+1)$ and $p(1)$. The goal of this benchmark is to measure the performance of the worst case scenario of each propagation step only propagating a single domain element. Preliminary experiments showed that due to the treatment of nested terms in SLI, the second sentence could not be grounded efficiently and hence the effect of iterative small propagations was not tested with this benchmark. For this reason, we switched to a version where the second sentence is grounded manually.

We ran the benchmarks on a computer with an AMD EPYC 9334 and 16 GB of RAM available with a timeout of 10 minutes for each problem instance. We denote SLI_s as SLI using `separated` implementation and SLI_c using the `combined` implementation. Table 1 shows the average grounding time of all benchmarks with interesting results, for the calculation of the average grounding time a failed instance counted as 15 minutes.

For many of the `TriangleGraph` benchmarks both algorithms are competitive with both IDP3 and clingo, and occasionally outperform them. In `TriangleGraphFindAll`, the `separated` approach performs much worse than the `combined` approach. This seems to be caused by the `separated` approach extending the structure twice, once for the ct-target and once for the cf-target, while the `combined` approach is able to extend the structure using a total approximating pair represented by a single set. Both `separated` and `combined` perform better than SLI and IDP3 in `TriangleGraphSubset`, but are still substantially outperformed by clingo and dlv. The poor results of `IDP3` on `TriangleGraphCheck` occurred on satisfiable instances, where early cutoff was not possible.

The results of `ImplicationLadder` show that bit vectors are not suitable for performing many small propagations.

Both clingo and dlv performed the best on `NonPartitionRemovalColoring`, while both `separated` and `combined` performed much better than SLI, and slightly better than IDP3. In `HamiltonianPath` and `Reachability` all SLI variants perform much worse than IDP3. Tthis is likely caused by the fact that SLI adds level mappings to its groundings, while IDP3 uses a solver that can handle recursive rules.

7 Related work

The main goal of this paper is to make the grounding process more efficient by exploiting information that is already present in the theory and structure. We achieved this building on previously developed techniques of grounding with

[6] `https://rdr.kuleuven.be/previewurl.xhtml?token=99d551ba-631d-483b-8608-8793f7d965b9`

	SLI	**SLI$_s$**	**SLI$_c$**	**IDP3**	**clingo**	**dlv**
ImplicationLadder (101)	2.21	497	498	31.6	2.41	**1.51**
TGCheck (14)	0.247	0.249	0.255	44.0	0.221	**0.148**
TGConstructAtleast (14)	0.415	0.418	**0.400**	0.885	473	709
TGConstructGraph (14)	515	**456**	**456**	475	476	710
TGContainsAll (14)	0.631	0.650	0.641	0.366	0.396	**0.261**
TGFindAll (14)	515	327	0.408	485	0.269	**0.190**
TGSubset (14)	515	389	366	489	0.329	**0.283**
NPRC (110)	104	7.09	7.23	28.6	**2.67**	2.83
CommonItem (100)	0.956	0.971	0.969	1.298	1.70	**0.936**
CompleteSets (100)	1.14	1.14	**1.13**	1.78	1.93	1.16
GraphColouring (50)	**1.26**	1.27	1.27	3.91	66.6	2.11
HamiltonianPath (20)	179	175	176	16.7	**0.170**	0.189
PPM (200)	**0.117**	0.122	0.124	242	95.7	47.0
RamseyNumbers (15)	92.1	730	594	88.8	1.70	**0.936**
Reachability (50)	526	505	506	276	0.0647	**0.0165**

Table 1: Average results of all benchmarks in seconds. The number of instances for each benchmark is shown in brackets. `NonPartitionRemovalColoring` has been shortened to `NPRC` and `PermutationPatternMatching` has been shortened to `PPM`. `TriangleGraph` has been shortened for `TG`.

bounds [26] and with bitvectors [21]. There are classes of approaches that tackle the problem of large grounding time and size (also known as the *grounding bottleneck* [1]) in different ways. The first is *lazy grounding* [4, 6, 8, 14, 19, 24], a technique that, unlike the classical ground-and-solve approach, defers grounding by only grounding parts of the input problem when the solver requires it. Such lazy grounding systems typically require changes to both the solver and the grounder, and as such are more complicated to get to work than techniques such as grounding with bounds. Second, there are solvers that solve (typically in a top-down fashion), without grounding the input problem at all [16,17]. Another approach to tackle large grounding time and size is by compiling rules to propagators and making the solver use these propagators during the search process [11], allowing compiled rules to not be grounded. A different approach named body-decoupled grounding [3], decouples non-ground atoms such that the number of variables in the rule does not contribute to the overall grounding size, only the maximum predicate arity used does. All the classes of methods above could potentially be combined with the techniques presented in this paper. Indeed, lazy-grounding, solving-without-grounding and body-decoupled grounding approaches could use the bounds computed by our method for clever instantiation of variables, while compilation-based approaches could use our method to achieve more efficient propagators; investigating this possibility is a topic for future work.

8 Conclusion

In this paper, we described a theoretical method to reduce groundings using information from a theory and structure. We used this method in two concrete implementations, an implementation that followed the abstract algorithm, and one designed to propagate everything all at once using bit vectors. Experimental results of both implementations showed that they perform similarly to each other, and both have bad performance on problems where many small propagations are possible, implying that a stop criterion is likely worthwhile. Both implementations are competitive with both IDP3 using BDDs, and clingo, often outperforming IDP3 using BDDs.

This research opens up several opportunities for future work. First, it would be interesting to investigate how to deal with the issue of many small propagations and to develop methods that perform well on this kind of instances and still propagate well on more complicated problems. Second, investigating how to improve propagation for formulas with nested terms. Third, to improve trust in combinatorial solving algorithms, there is a trend to use *certifying* algorithms in several fields, e.g., SAT solving [25], MaxSAT solving [23], and SMT solving [2]. Recently, this idea was also explored for the *grounding* phase [20]. It would be interesting to investigate how to produce certificates for the grounding methods described in this paper. An important challenge would then be how to efficiently translate the parallel reasoning steps performed using bit vectors to inherently sequential proofs that can be checked by a proof checker.

Acknowledgements This research received funding from the Flemish Government under the "Onderzoeksprogramma Artificiële Intelligentie (AI) Vlaanderen" programme and by Flanders Innovation & Entrepreneurship (HBC.2022.0071, Tetra-project) and the European Union (ERC, CertiFOX, 101122653). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

References

1. Balduccini, M., Lierler, Y., Schüller, P.: Prolog and ASP inference under one roof. In: LPNMR. LNCS, vol. 8148, pp. 148–160. Springer (2013)
2. Barbosa, H., Reynolds, A., Kremer, G., Lachnitt, H., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Viswanathan, A., Viteri, S., Zohar, Y., Tinelli, C., Barrett, C.W.: Flexible proof production in an industrial-strength SMT solver. In: IJCAR. LNCS, vol. 13385, pp. 15–35. Springer (2022)
3. Besin, V., Hecher, M., Woltran, S.: Body-decoupled grounding via solving: A novel approach on the ASP bottleneck. In: Raedt, L.D. (ed.) Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022, Vienna, Austria, 23-29 July 2022. pp. 2546–2552. ijcai.org (2022)
4. Bogaerts, B., Weinzierl, A.: Exploiting justifications for lazy grounding of answer set programs. In: IJCAI. pp. 1737–1745. ijcai.org (2018)

5. Clark, K.L.: Negation as failure. In: Logic and Data Bases. pp. 293–322. Advances in Data Base Theory, Plemum Press, New York (1977)
6. Dao-Tran, M., Eiter, T., Fink, M., Weidinger, G., Weinzierl, A.: Omiga : An open minded grounding on-the-fly answer set solver. In: JELIA. LNCS, vol. 7519, pp. 480–483. Springer (2012)
7. De Cat, B., Bogaerts, B., Bruynooghe, M., Janssens, G., Denecker, M.: Predicate logic as a modeling language: the IDP system. In: Declarative Logic Programming, pp. 279–323. ACM / Morgan & Claypool (2018)
8. De Cat, B., Denecker, M., Stuckey, P.J.: Lazy model expansion by incremental grounding. In: ICLP (Technical Communications). LIPIcs, vol. 17, pp. 201–211. Schloss Dagstuhl (2012)
9. de Moura, L.M., Bjørner, N.S.: Z3: an efficient SMT solver. In: TACAS. LNCS, vol. 4963, pp. 337–340. Springer (2008)
10. Denecker, M.: Extending classical logic with inductive definitions. In: Computational Logic. LNCS, vol. 1861, pp. 703–717. Springer (2000)
11. Dodaro, C., Mazzotta, G., Ricca, F.: Blending grounding and compilation for efficient ASP solving. In: KR (2024)
12. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Multi-shot ASP solving with clingo. CoRR **abs/1705.09811** (2017)
13. Goubault, J.: A BDD-based simplification and skolemization procedure. Log. J. IGPL **3**(6), 827–855 (1995)
14. Lefèvre, C., Nicolas, P.: The first version of a new ASP solver : ASPerix. In: LPNMR. LNCS, vol. 5753, pp. 522–527. Springer (2009)
15. Lemire, D., Kai, G.S.Y., Kaser, O.: Consistently faster and smaller compressed bitmaps with roaring. Softw. Pract. Exp. **46**(11), 1547–1569 (2016)
16. Marple, K., Gupta, G.: Galliwasp: A goal-directed answer set solver. In: LOPSTR. LNCS, vol. 7844, pp. 122–136. Springer (2012)
17. Marple, K., Salazar, E., Gupta, G.: Computing stable models of normal logic programs without grounding. CoRR **abs/1709.00501** (2017)
18. Mitchell, D.G., Ternovska, E.: A framework for representing and solving NP search problems. In: AAAI. pp. 430–435. AAAI Press / The MIT Press (2005)
19. Palù, A.D., Dovier, A., Pontelli, E., Rossi, G.: GASP: answer set programming with lazy grounding. Fundam. Informaticae **96**(3), 297–322 (2009)
20. Van Caudenberg, D., Ek, A., Cantero, C., Bogaerts, B.: Towards a certifying grounder. In: Proceedings of 42nd International Conference on Logic Programming (ICLP) (2026), accepted
21. Van Laer, L., Vandevelde, S., Vennekens, J.: Efficiently grounding FOL using bit vectors. In: LPNMR. LNCS, vol. 15245, pp. 167–173. Springer (2024)
22. Van Laer, L., Vandevelde, S., Vennekens, J.: DIRT: a literature-based benchmark suite for grounders. In: JELIA (1). pp. 343–356. LNCS, Springer (2025)
23. Vandesande, D., Coll, J., Bogaerts, B.: Certified branch-and-bound MaxSAT solving. In: AAAI. pp. 14342–14351. AAAI Press (2026)
24. Weinzierl, A., Taupe, R., Friedrich, G.: Advancing lazy-grounding ASP solving techniques - restarts, phase saving, heuristics, and more. Theory Pract. Log. Program. **20**(5), 609–624 (2020)
25. Wetzler, N., Heule, M., Hunt, Jr., W.A.: DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In: SAT. LNCS, vol. 8561, pp. 422–429. Springer (2014)
26. Wittocx, J., Mariën, M., Denecker, M.: Grounding FO and FO(ID) with bounds. J. Artif. Intell. Res. **38**, 223–269 (2010)