# Clustering-Guided SMT($\mathcal{LRA}$) Learning[⋆]

Tim Meywerk$^{1[0000-0002-5960-5456]}$, Marcel Walter$^{1[0000-0001-5660-9518]}$, Daniel Große$^{2,3[0000-0002-1490-6175]}$, and Rolf Drechsler$^{1,3[0000-0002-9872-1740]}$

[1] Research Group of Computer Architecture, University of Bremen, Germany
[2] Chair of Complex Systems, Johannes Kepler University Linz, Austria
[3] Cyber Physical Systems, DFKI GmbH, Bremen, Germany
`{tmeywerk,m_walter,drechsler}@uni-bremen.de`
`daniel.grosse@jku.at`

**Abstract.** In the SMT($\mathcal{LRA}$) learning problem, the goal is to learn SMT($\mathcal{LRA}$) constraints from real-world data. To improve the scalability of SMT($\mathcal{LRA}$) learning, we present a novel approach called *SHREC* which uses hierarchical clustering to guide the search, thus reducing runtime. A designer can choose between higher quality (*SHREC1*) and lower runtime (*SHREC2*) according to their needs. Our experiments show a significant scalability improvement and only a negligible loss of accuracy compared to the current state-of-the-art.

**Keywords:** Satisfiability Modulo Theories · Clustering · Machine Learning

## 1 Introduction

Since the invention of effective solving procedures for the *Boolean Satisfiability* (SAT) problem [20], many formalisms for problem modeling have been introduced over the decades, including but not limited to *Linear Programming* (LP), *Quantified Boolean Formulas* (QBF), and *Satisfiability Modulo Theories* (SMT) (cf. [5] for an overview). With the progressive development of highly specialized solving engines for these domains [8,7], it has become possible to tackle critical problems like verification [11]. Also, exact logic synthesis [10], optimal planning, and other optimization problems [9] could be approached in a more effective manner (again, cf. [5] for an overview).

A trade-off between SAT's efficient solvers and SMT's expressive power is *Satisfiability Modulo Linear Real Arithmetic* (SMT($\mathcal{LRA}$)) [5]. It combines propositional logic over Boolean variables and linear arithmetic over real-valued variables. SMT($\mathcal{LRA}$) has a wide variety of applications, including formal verification [6,2], AI planning and scheduling [21], and computational biology [24].

[⋆] The research reported in this paper has been supported by the German Research Foundation DFG, as part of Collaborative Research Center (Sonderforschungsbereich) 1320 *EASE – Everyday Activity Science and Engineering*, University of Bremen (`http://www.ease-crc.org/`). The research was conducted in sub-project P04.

Generating SMT($\mathcal{LRA}$) models[4] by hand is both time-consuming and error-prone and requires detailed domain-specific knowledge. Nevertheless, in many cases, both satisfying and unsatisfying examples of model configurations can be extracted from measurements of the modeling domain. In these cases, the actual modeling task can be automated by an approach called *concept learning*. Concept learning has a long history in artificial intelligence, with *Probably Approximately Correct* (PAC) learning [23], *inductive logic programming* [16], and *constraint programming* [4]. These approaches usually focus on pure Boolean descriptions, i. e. SAT formulae. More recently, [13] introduced SMT($\mathcal{LRA}$) learning, which is the task of learning an SMT($\mathcal{LRA}$) formula from a set of satisfying and unsatisfying examples.

Alternatively, SMT($\mathcal{LRA}$) learning can also be formulated as a variation on the *programming by example* problem known from the *Syntax-Guided Synthesis* (SyGuS) framework. Most solvers in this area (e. g. [22,1,18,3]) are based on enumeration of possible solutions to be able to tackle a wide variety of syntactic constraints. This does, however, lead to overly complicated and inconvenient reasoning on continuous search spaces such as SMT($\mathcal{LRA}$). Apart from that, the problems have further subtle differences, e. g. accuracy of the solutions has higher significance in the concept learning setting.

On top of defining the SMT($\mathcal{LRA}$) learning problem, [13] also introduced an exact algorithm called *INCAL*. As the first of its kind, INCAL naturally comes with certain drawbacks in terms of runtime and is therefore not applicable to learn large models, which are required by most real-world concept learning applications (e. g. [15,12]).

Our contribution in this work is a novel approach for SMT($\mathcal{LRA}$) learning which uses *Hierarchical Clustering* on the examples to guide the search and thus speed up the model generation process. We call our general approach SHREC (S̲MT($\mathcal{LRA}$) learner with h̲ierar̲chical e̲xample c̲lustering) and introduce two algorithms SHREC1 and SHREC2 based on this idea. SHREC1 aims at a higher accuracy of the solution and therefore requires more runtime than SHREC2. SHREC2 instead follows a very fast and scalable method with minor losses of accuracy. Therefore, we provide the users, i. e. the model designers, with the possibility to choose between maximizing the accuracy of the learned model or improving runtime of the generation process so that also larger models can be learned in a reasonable time frame.

The remainder of this paper is structured as follows: To keep this paper self-contained, Section 2 gives an overview of related work and preliminaries in the area of SMT($\mathcal{LRA}$) learning as well as hierarchical clustering. Section 3 and Section 4 propose our main ideas, i. e. novel approaches for SMT($\mathcal{LRA}$) learning to tackle larger and more complex models using methods from machine learning. In Section 5 we conduct an experimental evaluation where we compare our results to the state-of-the-art. Section 6 concludes the paper.

[4] The term *model* is often used to refer to a satisfying assignment to some logical formula. In the context of this paper however, *model* refers to a logical formula that describes a system in the real world.

## 2 Related Work & Preliminaries

In this section, we give an overview of relevant related work and introduce concepts that we utilize in the remainder of this work.

### 2.1 SMT($\mathcal{LRA}$) Learning

The problem of SMT($\mathcal{LRA}$) learning has first been introduced in [13]. The goal is to find an SMT($\mathcal{LRA}$) formula which describes some system in the real world. However, no formal representation of the system is available. Instead, a set of measurements is given. In the following, these measurements are called examples. It is further assumed, that there exists an SMT($\mathcal{LRA}$) formula $\phi^*$ that accurately describes the system. The problem of SMT($\mathcal{LRA}$) learning is now defined as follows:

**Definition 1.** *Given a finite set of Boolean variables $B \coloneqq \{b_1, \dots, b_n\}$ and a finite set of real-valued variables $R \coloneqq \{r_1, \dots, r_m\}$ together with a finite set of examples $E$. Each example $e \in E = (a_e, \phi^*(a_e))$ is a pair of an assignment and a label. An assignment $a_e : B \cup R \mapsto \{\top, \bot\} \cup \mathbb{R}$ maps Boolean variables to* true *($\top$) or* false *($\bot$), and real-valued variables to real-valued numbers. The label $\phi^*(a_e)$ is the truth value obtained by applying $a_e$ to $\phi^*$. We call an example* positive *if $\phi^*(a_e) = \top$ and* negative *otherwise. We denote the sets of positive and negative examples by $E^\top$ and $E^\bot$, respectively.*

*The task of SMT($\mathcal{LRA}$) learning is to find an SMT($\mathcal{LRA}$) formula $\phi$ which satisfies all elements in $E^\top$, but does not satisfy any element in $E^\bot$, which can be written as $\forall e \in E : \phi(a_e) = \phi^*(a_e)$.*

*Example 1.* Consider the SMT($\mathcal{LRA}$) formula

$$\phi^*(b_1, r_1) = (\neg b_1 \vee (-0.5 \cdot r_1 \leq -1)) \wedge (b_1 \vee (1 \cdot r_1 \leq 0))$$

A possible set of examples would be

$$\begin{aligned} E = \{&(\{b_1 \mapsto \top, r_1 \mapsto 0\}, \bot), (\{b_1 \mapsto \top, r_1 \mapsto 2.5\}, \top), \\ &(\{b_1 \mapsto \bot, r_1 \mapsto 2\}, \bot), (\{b_1 \mapsto \bot, r_1 \mapsto -0.6\}, \top)\} \end{aligned}$$

We call an algorithm that tackles the task of finding an unknown SMT($\mathcal{LRA}$) formula to a given set of examples, i. e. finding a solution to an instance of the aforementioned problem, *learner*.

Each learner must operate on a given set of possible target formulae, called the *hypothesis space $\Phi$*. Similar to [13], we focus on *CNF* formulae as our hypothesis space.

**Definition 2.** *A CNF formula over a set of variables $B \cup R$ is a conjunction of* clauses*, a clause is a disjunction of* literals *and a literal can be a Boolean variable $b \in B$, its negation, or a linear constraint over the real variables. Linear constraints (also called* halfspaces*) have the form $a_1 \cdot r_1 + \dots + a_m \cdot r_m \leq d$ with real constants $a_i$ and $d$ and real variables $r_i \in R$.*

*Additionally, we define the* cost $c$ *of a CNF formula with a given number of clauses $k$ and (not necessarily unique) halfspaces $h$ as $c = w_k \cdot k + w_h \cdot h$, where $w_k$ and $w_h$ are weights associated with clauses and halfspaces, respectively. The cost is a measure for the size and complexity of a formula and can be tuned to focus more on clauses or halfspaces.*

A learner tries to find an SMT($\mathcal{LRA}$) formula $\phi \in \Phi$. We say that an example $e$ satisfies a formula $\phi$ iff $\phi(a_e) = \top$ and is consistent with $\phi$ iff $\phi(a_e) = \phi^*(a_e)$. Using these definitions, the goal of SMT($\mathcal{LRA}$) learning is to find a formula $\phi$ that is consistent with all examples, i. e. as mentioned before, one that is satisfied by all positive examples and unsatisfied by all negative ones.

*Example 2.* Consider the example set $E$ from Example 1 again. A possible CNF solution to those examples would be

$$\phi = (b_1 \vee (0.5 \cdot r_1 \leq -0.25)) \wedge (\neg b_1 \vee (-1 \cdot r_1 \leq -2.1))$$

Obviously, $\phi^*$ is also a feasible solution, but might not be found by the learner which only knows about the example set $E$.

Since $\phi^*$ is not known to the learner and the example set $E$ is usually non-exhaustive, it can not be expected that the learner finds a model equivalent to $\phi^*$. It should, however, be as close as possible. This leads to the measure of *accuracy.*

**Definition 3.** *Given two example sets $E_{train}$ and $E_{test}$ which were independently sampled from the (unknown) SMT($\mathcal{LRA}$) formula $\phi^*$, the accuracy of a formula $\phi$, which was learned from $E_{train}$, is the ratio of correctly classified examples in $E_{test}$.*

Generally, finding any formula for a given example set is not a hard problem. One could construct a simple CNF that explicitly forbids one negative example in each clause and allows all other possible assignments. However, such a formula would have numerous clauses and would not generalize well to new examples, yielding a low accuracy. To avoid such cases of overfitting, a smaller target formula, i. e. one with lower cost, should generally be preferred over a larger i. e. more expensive one.

### 2.2 INCAL

In addition to introducing the problem of SMT($\mathcal{LRA}$) learning, [13] also presented the first algorithm to tackle it, called *INCAL*. INCAL addresses the SMT($\mathcal{LRA}$) learning problem by fixing the number of clauses $k$ and the number of halfspaces $h$ and then encodes the existence of a feasible CNF with those parameters in SMT($\mathcal{LRA}$). If no such formula exists, different values for $k$ and $h$ need to be used. The order in which to try values for $k$ and $h$ can be guided by the cost function $w_k \cdot k + w_h \cdot h$.

INCAL's SMT encoding uses Boolean variables to encode which clauses contain which literals, real variables for the coefficients and offset of all halfspaces, and Boolean auxiliary variables encoding which halfspace and clause are satisfied by which example. It consists of the definition of those auxiliary variables and a constraint enforcing the consistency of examples with the learned formula. To cope with a high number of examples, INCAL uses an iterative approach and starts the encoding with only a small fraction of all variables. After a solution consistent with this subset has been found, additional conflicting examples are added.

The complexity of the learning problem does however not exclusively stem from the size of the input. Another, arguably even more influential factor is the complexity of the learned formula. If an example set requires numerous clauses or halfspaces, it will be much harder for INCAL to solve.

So far, we have discussed the state-of-the-art related work in SMT($\mathcal{LRA}$) learning. In this work, we present a new SMT($\mathcal{LRA}$) learner which incorporates a *hierarchical clustering* technique. To keep this paper self-contained, we give some preliminaries about clustering in the following section.

### 2.3 Hierarchical Clustering

In machine learning, the problem of clustering is to group a set of objects into several clusters, such that all objects inside the same cluster are closely related, while all objects from different clusters are as diverse as possible (cf. [14] for an overview). To describe the similarity between objects, a *distance metric* is needed.

Often, objects are described by the means of a vector $(v_1, \ldots, v_n)$ of real values. Typical distance metrics of two vectors $v, w$ are (1) the *Manhattan distance* ($L_1$ norm) $dist(v,w) = \sum_{i=1}^{n} |v_i - w_i|$ , (2) the *Euclidean distance* ($L_2$ norm) $dist(v,w) = \sqrt{\sum_{i=1}^{n} (v_i - w_i)^2}$ , or (3) the $L_\infty$ norm $dist(v,w) = max(|v_i - w_i|)$.

A common approach to clustering is *hierarchical clustering* [19]. The main idea of hierarchical clustering is to build a hierarchical structure of clusters called a *dendrogram*. A dendrogram is a binary tree annotated with distance information. Each node in the dendrogram represents a cluster. Each inner node thereby refers to the union of clusters of its two children; with leaf nodes representing clusters that contain exactly one vector. This way, the number of contained vectors per node increases in root direction with the root node itself containing all vectors given to the clustering algorithm. Each inner node is also annotated with the distance between its two children. In graphical representations of dendrograms, this is usually visualized by the height of these nodes.

*Example 3.* An example dendrogram can be seen in Figure 1. The dashed and dotted lines may be ignored for now. The dendrogram shows a clustering over six input vectors, labeled $A$ to $F$. The distance between nodes can be seen on the y-axis. For instance, the distance between vectors $\{B\}$ and $\{C\}$ is 1, while the distance between their combined cluster $\{B, C\}$ and vector $\{A\}$ is 2.

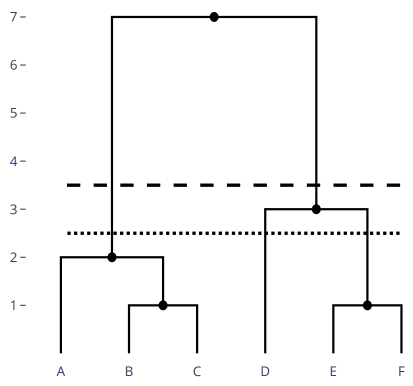


Fig. 1: A simple dendrogram

In this paper, we will focus on *agglomerative hierarchical clustering* [19], which builds the dendrogram by assigning each vector to its own cluster and then combines the two closest clusters until a full dendrogram has been built.

To combine the two closest clusters, it is necessary to not only measure the distance between two vectors but also between larger clusters. To this end, a *linkage criterion* is needed. Given two clusters $c$ and $d$, some established linkage criteria are (1) the *single linkage* criterion, which picks the minimum distance between two vectors from $c$ and $d$, (2) the *complete linkage* criterion, which picks the maximum distance between two vectors from $c$ and $d$, or (3) the *average linkage* criterion, which takes the average of all distances between vectors from $c$ and $d$.

Most combinations of distance measure and linkage criterion can be applied to a given hierarchical clustering problem. The results may, however, vary heavily depending on the application.

To obtain a concrete clustering from a dendrogram, one fixes a *distance threshold*. The final clustering is then made up of the nodes whose distances lie just below the distance threshold and whose parent nodes are already above it. In graphical representations, the distance threshold can be indicated by a horizontal line, making the clusters easily visible.

*Example 4.* The dashed line in Figure 1 represents a distance threshold of 3.5. Following this threshold, the dendrogram would be split into the two clusters $\{A, B, C\}$ and $\{D, E, F\}$. Using a smaller distance threshold of 2.5, indicated by the dotted line, would result in the three clusters $\{A, B, C\}$, $\{D\}$, and $\{E, F\}$.

The following section shows how we utilize hierarchical clustering in our novel SMT($\mathcal{LRA}$) learner.

## 3 Using Dendrograms for SMT($\mathcal{LRA}$) Learning

In this section, we introduce our novel SMT($\mathcal{LRA}$) learner. We describe how the hierarchical clustering is used to guide its search and discuss the resulting algorithm which we call SHREC1. We start with the general idea in Section 3.1, followed by the algorithm in Section 3.2 and finally optimizations in Section 3.3. In Section 4 we present the algorithm SHREC2 to trade-off some accuracy for a further increase of scalability.

### 3.1 Main Idea

The main scalability problem of exact approaches for SMT($\mathcal{LRA}$) learning lies in the large combined encoding that is needed to describe a full CNF. This encoding quickly becomes hard to solve for SMT solvers when the number of clauses and halfspaces is increased. We, therefore, propose to not learn the target CNF as a whole, but rather to learn single clauses and then combine them into the target formula.

When looking at the structure of CNF formulas, it becomes apparent that positive examples need to satisfy all individual clauses, while negative ones only need to unsatisfy a single one. If one had a perfect prediction, which negative examples belong to which clause, one could simply learn each clause on its own, using a simpler encoding, and still obtain an exact solution. But even an imperfect prediction, which needs some additional clauses, would yield a correct and relatively small solution.

We propose a novel heuristic that produces such a prediction using agglomerative hierarchical clustering. The clustering algorithm partitions the negative examples into groups of closely related examples given their values in the assignment $a_e$. This is due to the intuition that it is easier to find a single clause for a set of closely related examples as opposed to an arbitrary one. The reason to use hierarchical clustering as opposed to other clustering algorithms is the ability to seamlessly adjust the number of clusters and thus the number of clauses in the target formula.

To obtain a suitable clustering vector, we normalize the examples. For Boolean variables, the values of $\top$ and $\bot$ are replaced with 1 and 0, respectively. The values $a_e(r)$ of real variables $r$ are translated into the form $\frac{a_e(r)-r_{min}}{r_{max}-r_{min}}$, where $r_{min}$ and $r_{max}$ are the smallest and highest possible values for variable $r$, respectively. If those values are not known beforehand, they can simply be estimated from the existing data. This normalization ensures that all feature values lie in the interval $[0, 1]$, which results in each variable having a similar influence on the clustering outcome.

### 3.2 Algorithm SHREC1

The full algorithm SHREC1 is described in Algorithm 1. The algorithm receives as input a set of examples $E$ and returns a formula $\phi$ consistent with $E$. The first step of the algorithm is the function BUILD-DENDROGRAM, which uses agglomerative hierarchical clustering to build a dendrogram from the negative examples. The function uses the normalization procedure described in the previous section. Please note that BUILD-DENDROGRAM is agnostic to specific distance metrics and linkage criteria.

**Algorithm 1** Algorithm SHREC1

**Input:** Example set $E$
**Output:** SMT($\mathcal{LRA}$) formula $\phi$

```
function LEARN-MODEL($E$)
    $N_0 \leftarrow$ BUILD-DENDROGRAM($E^{\perp}$)
    $cost \leftarrow w_k$
    loop
        $k \leftarrow 1$
        while $w_k \cdot k \leq cost$ do
            $\phi \leftarrow \top$
            $nodes \leftarrow$ SELECT-NODES($N_0$, $k$)
            $h \leftarrow 0$
            $valid \leftarrow \top$
            for all $N_i \in nodes$ do
                $cost\text{-}bound \leftarrow cost - w_k \cdot k - w_h \cdot h$
                $h', \psi \leftarrow$ SEARCH-CLAUSE($N_i$, $cost\text{-}bound$)
                if $\psi = \emptyset$ then
                    $valid \leftarrow \perp$
                    break
                else
                    $\phi \leftarrow \phi \wedge \psi$
                    $h \leftarrow h + h'$
            if $valid$ then
                return $\phi$
            else
                $k \leftarrow k + 1$
        $cost \leftarrow$ NEXT-COST($cost$)

function SEARCH-CLAUSE($N_i$, $cost\text{-}bound$)
    $h \leftarrow 0$
    while $w_h \cdot h \leq cost\text{-}bound$ do
        $\omega \leftarrow$ ENCODE-CLAUSE($E^{\top} \cup N_i$, $h$)
        $\psi \leftarrow$ SOLVE($\omega$)
        if $\psi \neq \emptyset$ then
            return $h, \psi$
        $h \leftarrow h + 1$
    return $h, \emptyset$
```

The resulting dendrogram is referred to by its root node $N_0$. Each subsequent node $N_i$ has a unique, positive index $i$. As we do not need to distinguish between a node and the set of examples covered by it, we use $N_i$ to refer to both the node $N_i$ and its example set.

The algorithm is composed of several nested loops. The outermost loop (Lines 4-24) searches for a solution with increasing cost. Similar to INCAL, the cost is determined using a linear cost function $w_k \cdot k + w_h \cdot h$. The algorithm starts with the cost value set to $w_k$ in the first iteration, allowing a solution with exactly one clause and no halfspaces. After each iteration, the cost is incremented, increasing the search space.

Since for each cost value multiple combinations of $k$ and $h$ are possible, the next loop (Lines 6-23) starts with $k = 1$ and keeps increasing the number of clauses $k$ in each iteration. This, in turn, decreases the number of possible halfspaces. In each iteration, $k$ nodes are selected from the dendrogram through an appropriate distance threshold and stored in the variable *nodes*. The algorithm then tries to find a clause consistent with each node $N_i$ using as few halfspaces as possible. This is done in the function SEARCH-CLAUSE. If clauses for all nodes could be found within the cost bound, they are combined (Line 18) and the resulting CNF formula is returned. Since each clause satisfies all positive examples and each negative example is unsatisfied by at least one clause, this trivial combination yields a consistent CNF.

The function SEARCH-CLAUSE constitutes the innermost loop of the algorithm. Given a node $N_i$ and the remaining cost left for halfspaces, the function tries to find a clause that is consistent with all positive examples and the negative examples in $N_i$. To keep the cost as low as possible, an incremental approach is used again, starting the search with 0 halfspaces and increasing the number of possible halfspaces $h$ with each iteration. To find a clause for a fixed set of examples and a fixed number of halfspaces, an SMT encoding is used in Line 28. This encoding is a simplified version of the encoding from INCAL and uses the following variables: $l_b$ and $\hat{l}_b$ with $b \in B$ encode whether the clause contains $b$ or its negation, respectively; $a_{jr}$ and $d_j$ with $r \in R$ and $1 \leq j \leq h$ describe the coefficients and offset of halfspace $j$, respectively; $s_{ej}$ with $e \in E$ and $1 \leq j \leq h$ is an auxiliary variable encoding whether example $e$ satisfies halfspace $j$.

The overall encoding for a single example $e$ can now be formulated with only two parts, i. e., (1) the definition of $s_{ej}$, which is identical to INCAL's

$$\bigwedge_{j=1}^{h} s_{ej} \iff \sum_{r \in R} a_{jr} \cdot a_e(r) \leq d_j,$$

and (2) the constraint which enforces consistency of $e$ with the learned clause

$$\begin{aligned} \bigvee_{j=1}^{h} s_{ej} \vee \bigvee_{b \in B} \Big( (l_b \wedge a_e(b)) \vee \Big(\hat{l}_b \wedge \neg a_e(b)\Big)\Big), \quad & \text{if } \phi^*(a_e) \\ \bigwedge_{j=1}^{h} \neg s_{ej} \vee \bigwedge_{b \in B} \Big( (\neg l_b \vee \neg a_e(b)) \wedge \Big(\neg\hat{l}_b \vee a_e(b)\Big)\Big), \quad & \text{otherwise.} \end{aligned}$$

The full encoding is the conjunction of the encodings for all examples in $E^{\top} \cup N_i$. Like INCAL, SHREC1 also uses an incremental approach. First, we only generate the above encoding for a few examples and then iteratively add more conflicting examples.

The function SOLVE in Line 29 takes an encoding, passes it to an SMT solver, and if a solution to the encoding is found, it is translated back into an SMT($\mathcal{LRA}$) clause. Otherwise, SOLVE returns $\emptyset$.

If a clause could be found within the cost bound (Line 30), it is returned together with the number of halfspaces used. Otherwise, $\emptyset$ is returned together with the highest attempted number of halfspaces.

This basic algorithm can be further improved in terms of runtime and cost by two optimizations described in the next section.

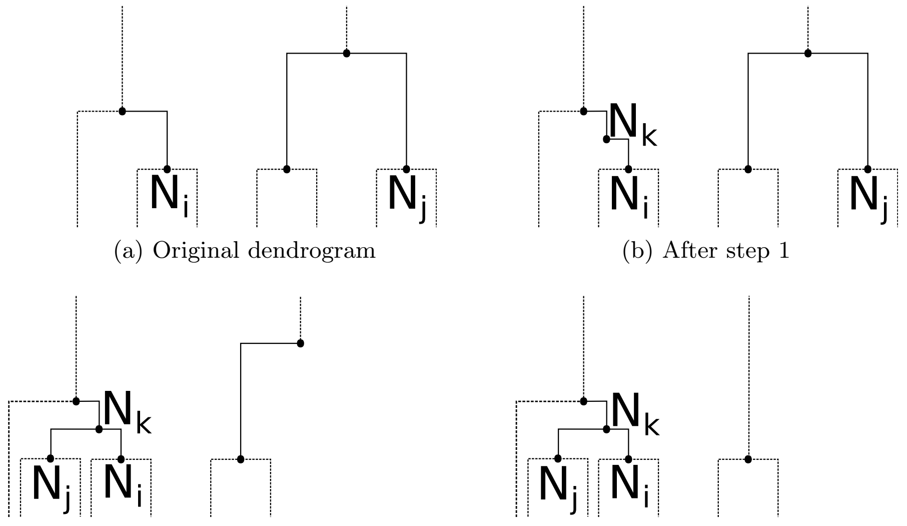
## 3.3 Result Caching and Dendrogram Reordering

The algorithm SHREC1 as described above suffers from two problems, namely (A) repeated computations and (B) an inflexible search, which we will both discuss and fix in this section.

First, we address issue (A), that SHREC1 re-computes certain results multiple times. When a node is passed to the function SEARCH-CLAUSE together with some *cost-bound*, a consistent clause is searched using up to $\frac{cost\text{-}bound}{w_k}$ halfspaces. In later iterations of the algorithm's main loop, SEARCH-CLAUSE is called again with the same node and higher *cost-bound*. This leads to the same SMT encoding being built and solved again. To avoid these repeated computations, each node caches the results of its computations and uses them to avoid unnecessary re-computation in the future.

Second, SHREC1 never modifies the initial dendrogram during the search, making the approach inflexible. We address this issue (B) in the following. If the initial clustering assigns only a single data point to an unfavorable cluster, this might lead to a much larger number of clauses needed to find a consistent formula. This, in turn, leads to a lower accuracy on new examples as well as a higher runtime. To counteract this problem, we apply a novel technique, which we call *dendrogram reordering*: whenever a clause $\psi$ has been found for a given node $N_i$ and some number of halfspaces $h$, it might be that $\psi$ is also consistent with additional examples, which are not part of $N_i$, but instead of some other node $N_j$. To find such nodes $N_j$, a breadth-first search is conducted on the dendrogram. If some node $N_j$ has been found such that $\forall e \in N_j : \psi(a_e) = \phi^*(a_e)$, the dendrogram is reordered to add $N_j$ to the sub-tree under $N_i$. This does not increase the cost of $N_i$, because the new examples are already consistent with $\psi$, but might reduce the cost of $N_j$'s (transitive) parent node(s).

Figure 2 illustrates the reordering procedure, which consists of the following steps: (1) Generate a new node $N_k$ and insert it between $N_i$ and its parent. Consequently, $N_k$'s first child node is $N_i$ and its parent node is $N_i$'s former parent node. Set $N_k$'s cached clause to $\psi$. (2) Remove $N_j$ and its whole sub-tree from its original place in the dendrogram and move it under $N_k$ as $N_k$'s second child node. (3) To preserve the binary structure of the dendrogram, $N_j$'s former parent node must now be removed. The former sibling node of $N_j$ takes its place in the dendrogram.



(a) Original dendrogram (b) After step 1

(c) After step 2 (d) Final result

Fig. 2: Dendrogram reordering

This way, additional examples can be assigned to an already computed clause, reducing the complexity in other parts of the dendrogram, too, inherently. Consequently, the reordering can only decrease the overall cost of the dendrogram and never increase it. Therefore, dendrogram reordering can handle imperfect initial clusterings by dynamically improving them.

## 4 Improving Runtime using Nested Dendrograms

In the previous section, we introduced a novel SMT($\mathcal{LRA}$) learner with improved runtime compared to INCAL (as we will demonstrate by an experimental evaluation in Section 5) without a significant impact on the quality, i. e. the accuracy of the resulting formulae. In real-world applications, however, an even faster and more scalable algorithm might be preferred, even with minor losses of accuracy. In this section, we propose a technique for *nested hierarchical clustering* to realize this trade-off. We call this algorithm SHREC2.

### 4.1 Main Idea

While SHREC1 is already expected to reduce the runtime of the SMT($\mathcal{LRA}$) learner, it still has to solve relatively complex SMT constraints to find a consistent clause. To further improve runtime, we again reduce the complexity of these SMT solver calls. The algorithm SHREC2 starts just like SHREC1 by clustering the negative examples and then searching for clauses consistent with the different clusters. However, instead of searching for consistent clauses through an

SMT encoding, SHREC2 also clusters the positive examples, ultimately leaving only the learning of single halfspaces to the solver. This is realized through a simpler encoding, shifting the algorithm's overall complexity from exponential to polynomial runtime.

When searching for a single clause, negative examples must not satisfy any literal of the clause, while positive examples only have to satisfy a single literal each. This fact can now be used to learn literals one by one. To this end, *nested dendrograms* are introduced.

We, therefore, extend our definition of dendrograms from the previous sections. A dendrogram that clusters negative examples like the one used in SHREC1 is called a *negative dendrogram* from now on. Its nodes are called *negative nodes* denoted as $N_i^\perp$. In SHREC2, we also use *positive dendrograms*, which analogously cluster the positive examples. Each node $N_i^\perp$ of the negative dendrogram is assigned a new positive dendrogram $N_{i,0}^\top$. Each positive node $N_{i,j}^\top$ holds a set of positive examples from $E^\top$ which again are being clustered just like their negative counterparts.

Given a negative node $N_i^\perp$ and some halfspaces $h$, SHREC2 first finds all Boolean literals that are consistent with the examples in $N_i^\perp$. Because the cost function is only dependent on the number of clauses and halfspaces, these Boolean literals can be part of the clause without increasing the cost. Then, all positive examples that are inconsistent with any of the Boolean literals are determined. These examples constitute $N_{i,0}^\top$. The positive dendrogram under $N_{i,0}^\top$ is built in the same manner as the negative dendrogram, using the same normalization scheme. Values of Boolean variables are however left out of the clustering.

To find a set of halfspaces that are consistent with the remaining positive examples as well as the negative examples in $N_i^\perp$, an encoding is generated for each of the top $h$ nodes from $N_{i,0}^\top$ matching them with individual halfspaces.

## 4.2 Algorithm SHREC2

Algorithm 2 describes the algorithm SHREC2. The main function (LEARN-MODEL) is identical to the one in Algorithm 1. The difference here can be found in the function SEARCH-CLAUSE, which tries to learn a clause given a set of negative examples and a cost bound.

The function starts by computing the set $L$ of all literals that are consistent with all negative examples in $N_i^\perp$ (Line 4). It then computes the subset $E'$ of all positive examples not consistent with any literal in $L$ (Line 7). These remaining examples need further literals to be consistent with the clause. Consequently, if $E'$ is already empty at this point, the disjunction of the literals in $L$ is already a consistent clause and can be returned.

Otherwise, additional literals are needed. Because any further Boolean literals would be inconsistent with the negative examples, halfspaces are needed. To find a reasonable assignment of examples in $E'$ to halfspaces, hierarchical clustering is used again. Instead of clustering the negative examples, the algorithm clusters the positive ones in $E'$. Since Boolean values have no influence on the halfspaces, they are not used in this clustering.

**Algorithm 2** Algorithm SHREC2

**Input:** Example set $E$
**Output:** SMT($\mathcal{LRA}$) formula $\phi$

```
1:  function LEARN-MODEL($E$)
2:      ...                                                ▷ identical to SHREC1

3:  function SEARCH-CLAUSE($N_i^\perp$, $cost\text{-}bound$)
4:      $L \leftarrow \{b \in B \mid \forall e \in N_i^\perp : a_e(b) = \perp\} \cup$
            $\{\neg b \mid b \in B, \forall e \in N_i^\perp : a_e(b) = \top\}$
5:      $\psi \leftarrow \bigvee_{l \in L} l$
6:      $\psi' \leftarrow \psi$
7:      $E' \leftarrow \{e \in E^\top \mid \psi(a_e) = \perp\}$
8:      if $E' = \emptyset$ then
9:          return $0, \psi$
10:     $N_{i,0}^\top \leftarrow$ BUILD-DENDROGRAM($E'$)
11:     $h \leftarrow 1$
12:     while $w_h \cdot h \leq cost\text{-}bound$ do
13:         $\psi \leftarrow \psi'$
14:         $nodes \leftarrow$ SELECT-NODES($N_{i,0}^\top, h$)
15:         $valid \leftarrow true$
16:         for all $N_{i,j}^\top \in nodes$ do
17:             $\omega \leftarrow$ ENCODE-HALFSPACE($N_i^\perp \cup N_{i,j}^\top$)
18:             $\theta \leftarrow$ SOLVE($\omega$)
19:             if $\theta = \emptyset$ then
20:                 $valid \leftarrow false$
21:                 break
22:             else
23:                 $\psi \leftarrow \psi \vee \theta$
24:         if $valid$ then
25:             return $h, \psi$
26:         else
27:             $h \leftarrow h + 1$
28:     return $h, \emptyset$
```

The remainder of the algorithm is now very similar to the process in the main function. The algorithm increases the number of halfspaces in each iteration, starting at 1, until a solution has been found or the cost bound has been reached. In each iteration, the top $h$ nodes from the positive dendrogram are selected. For each node $N_{i,j}^\top$, the algorithm tries to find a halfspace for the examples in $N_i^\perp$ and $N_{i,j}^\top$ via an encoding. If no such halfspace exists, the algorithm retries with an increased $h$. If halfspaces could be found for all nodes, a disjunction of those halfspaces and the literals in $L$ is returned as a consistent clause.

The encoding for a single example $e \in E$ is a simplified version of the one used in SHREC1, which uses variables $a_r$ and $d$, describing the coefficients and offset of the halfspace, respectively. The encoding now only consists of a single constraint per example:

$$\sum_{r \in R} a_r \cdot a_e(r) \bowtie d$$

where $\bowtie$ is $\leq$ if $\phi^*(a_e) = \top$ and $>$ otherwise. The full encoding is again the conjunction of the encodings for all examples. Like in INCAL and SHREC1, examples are also added iteratively. Please note that the encoding of SHREC2 is only a linear program instead of a more complex SMT($\mathcal{LRA}$) encoding, making it solvable in polynomial time.

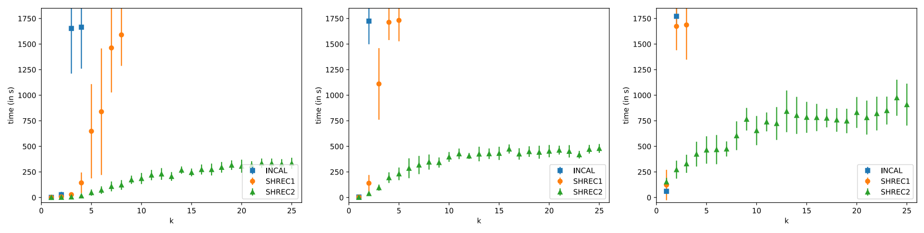
SHREC2 also uses result caching and dendrogram reordering in both levels of dendrograms. Besides, the positive dendrograms are computed only once for each negative node $N_i^{\perp}$ and are immediately cached for faster access.

## 5 Experiments

In this section, we evaluate the capabilities and applicability of the proposed algorithms SHREC1 and SHREC2. We have implemented them in Python using the SMT solver *Z3* [7] version 4.8.6 64 Bit and the *scikit-learn* package [17] version 1.3.1 for the hierarchical clustering. To this end, we conducted case studies and compared the results in terms of accuracy and runtime to INCAL. We ran all evaluations on an Intel Xeon E3-1240 v2 machine with 3.40 GHz (up to 3.80 GHz boost) and 32 GB of main memory running Fedora 26. In the following, we give detailed insight into the experimental setup in Section 5.1. We present the comparison of our approaches to INCAL in Section 5.2.

### 5.1 Experimental Setup

Due to the poor scalability of current approaches, no suitable real-world benchmarks for SMT($\mathcal{LRA}$) learning exist yet. In addition, benchmarks for SMT solving like the SMT-LIB collection are usually either unsatisfiable or only satisfied by few assignments, meaning they do not produce adequately balanced example sets. Therefore, experiments have to be conducted on randomly generated benchmarks. To this end, we use an approach similar to [13]: Given a set of parameters consisting of the number of clauses ($k$) and halfspaces per clause ($h$), we generate a CNF formula fitting these parameters. The generation procedure is also given a set of 1000 randomly generated assignments from variables to their respective values. The formula is then generated in such a way, that at least 30% and at most 70% of those assignments satisfy it. To ensure that the formula does not become trivial, it is also required that each clause is satisfied by at least $\frac{30}{k}$% of assignments that did not satisfy any previous clause. This ensures, that each clause has a significant influence on the formula and cannot be trivially simplified.



(a) $h = 1$ (b) $h = 2$ (c) $h = 3$

Fig. 3: Runtime comparison for different values of $h$

Since the main focus of SHREC is the improved scalability on larger formulae, we (similar to [13]) generated benchmarks with increasing $k$ and $h$ and fixed all other parameters to constant values. All generated formulae have 4 Boolean variables, 4 real variables, and 3 literals per clause. The benchmarks have between 1 and 25 clauses and between 1 and 3 halfspaces per clause, resulting in 75 different parameter configurations. We expect a higher number of clauses or halfspaces per clause to generally result in a harder benchmark. Since we cannot precisely control the difficulty, however, some smaller formulae might turn out to be more difficult than other larger ones. To mitigate these random fluctuations, we generated 10 formulae for each configuration, resulting in a total of 750 benchmarks.

For each benchmark, 1000 examples were randomly drawn. Boolean variables had an equal probability to be assigned to $\top$ or $\bot$. Real values were uniformly distributed in the interval $[0, 1)$.[5]

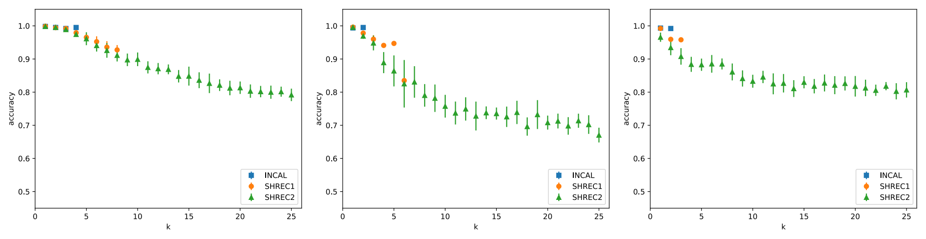
We used INCAL, SHREC1, and SHREC2 to find a CNF formula consistent with all examples. All three algorithms used a cost function with equal weights for clauses and halfspaces ($w_k = w_h = 1$). For each run, we measured the runtime and the accuracy on another independent set of 1000 examples. We set a timeout of 30 minutes for each run. This timeout is substantially longer than the one used in [13] and allows us to adequately observe the effect of the different configurations.

In the following section, the results are presented and discussed.

## 5.2 Comparison to INCAL

As mentioned in Section 3, SHREC1 and SHREC2 are able to use various distance metrics and linkage criteria in their clustering routine. To determine the most effective combination, we ran some preliminary experiments on a subset of the generated benchmarks. We evaluated the Manhattan distance, Euclidean distance, and the $L_\infty$ norm as possible distance metrics and the single, complete and average linkage criteria. Out of the nine possible combinations, the

[5] Please note that the choice of the interval does not influence the hardness of the learning problem because smaller values do not make the SMT solving process easier.



(a) $h = 1$ (b) $h = 2$ (c) $h = 3$

Fig. 4: Accuracy comparison for different values of $h$

Manhattan distance together with the average linkage criterion performed best. Therefore, this combination is used in the following comparison with INCAL.

Figure 3 shows the runtime for 1, 2 and 3 halfspaces per clause, respectively. On the x-axis, the number of clauses from $k = 1$ to $k = 25$ is shown. The y-axis shows the runtime in seconds. Each data point covers the runs on the 10 different benchmarks for the respective configuration. The squares, circles, and triangles mark the mean of all 10 runtimes, while the vertical error bars show the standard deviation. Runs that timed out were included in the calculation of mean and standard deviation as if they needed exactly 1800 seconds. If all runs of one configuration timed out, no data point is shown.

As expected, the number of clauses and halfspaces increases the runtime of all three algorithms. However, we can observe that the increase in runtime becomes smaller at a higher number of clauses. INCAL already times out at $k \geq 5$ for benchmarks with a single halfspace per clause and even at $k \geq 3$ for benchmarks with 2 or 3 halfspaces per clause. SHREC1 is able to handle larger benchmarks better, but still times out at $k \geq 8$, $k \geq 6$ and $k \geq 4$ for $h = 1$, $h = 2$, and $h = 3$, respectively. On instances where neither INCAL nor SHREC1 time out, SHREC1 is consistently considerably faster. SHREC2 is a lot more robust for increasing $k$ and $h$ and does not time out for any benchmark. SHREC2's runtime stays far below that of INCAL and SHREC1 for almost all of the benchmarks. This indicates SHREC2's superior scalability in terms of runtime, outperforming INCAL and SHREC1 by a large margin.

Naturally, we expect this success to come with a trade-off in the form of lower accuracy. Figure 4 shows the accuracy for 1, 2 and 3 halfspaces per clause, respectively. As before, each data point shows the mean and standard deviation of 10 benchmarks. Timeouts were not considered in the calculation this time. Configurations with 10 timeouts again have no data point displayed. As expected, the accuracy of all three algorithms is lower for larger problems. This is because a more complicated CNF needs to be found with the same number of examples. One can also observe, that SHREC1 and especially SHREC2 suffer more from this decrease in accuracy than INCAL. However, as Figure 4a shows, SHREC1's accuracy still stays above 95% even for benchmarks with up to 7 clauses.

The decrease of accuracy is only crucial for larger values of $k$ and $h$, which were not solved by INCAL at all. If given enough time, we can also expect INCAL to show a lower accuracy for these harder benchmarks. For the benchmarks which were solved by INCAL, SHREC1 and SHREC2 stay very close to 100% accuracy, as well. If one wants to compensate for the lower accuracy in other ways, the improved scalability of SHREC1 and SHREC2 could also be utilized to simply incorporate more training examples that can be handled due to better scalability.

Overall, the experimental results clearly show that SHREC is superior to the state-of-the-art exact approach INCAL in terms of scalability. SHREC1 needs considerably less runtime to learn formulae with only a slight loss of accuracy, while SHREC2 was several magnitudes faster and still kept the accuracy at a reasonable level.

## 6 Conclusion

In this work, we proposed a novel approach for SMT($\mathcal{LRA}$) learning. Our approach, SHREC, incorporates hierarchical clustering to speed up the learning process. Additionally, we presented two specific algorithms exploiting our findings with different objectives: SHREC1 aims for high accuracy of the learned model while SHREC2 trades-off accuracy for runtime, yielding a scalable approach to tackle even harder problems.

Our conducted experimental evaluation supports these claims. When compared to the state-of-the-art algorithm INCAL, our results clearly show that SHREC1 outperforms INCAL in terms of runtime with almost no loss of accuracy. SHREC2 on the other hand can handle benchmarks for which INCAL and SHREC1 timeout.

The better scalability permits our approach to handling interesting real-world problems on a larger scale. This opens up new possibilities on a variety of applications and enables future research in the domain.

## References

1. Alur, R., Radhakrishna, A., Udupa, A.: Scaling enumerative program synthesis via divide and conquer. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 319–336. Springer Berlin Heidelberg (2017)
2. Baldoni, R., Coppa, E., D'Elia, D.C., Demetrescu, C., Finocchi, I.: A Survey of Symbolic Execution Techniques. ACM Comput. Surv. **51**(3) (2018)
3. Barbosa, H., Reynolds, A., Larraz, D., Tinelli, C.: Extending enumerative function synthesis via smt-driven classification. In: Formal Methods in Computer Aided Design (FMCAD). pp. 212–220 (2019)
4. Bessiere, C., Coletta, R., Koriche, F., O'Sullivan, B.: A SAT-Based Version Space Algorithm for Acquiring Constraint Satisfaction Problems. In: Machine Learning: ECML 2005. pp. 23–34. Springer Berlin Heidelberg (2005)
5. Biere, A., Heule, M., van Maaren, H.: Handbook of Satisfiability, vol. 185. IOS press (2009)
6. Cordeiro, L., Fischer, B.: Verifying Multi-threaded Software using SMT-based Context-Bounded Model Checking. In: 2011 33rd International Conference on Software Engineering (ICSE). IEEE (2011)

7. De Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: TACAS/ETAPS. Berlin, Heidelberg (2008)
8. Eén, N., Sörensson, N.: An Extensible SAT-Solver. In: International conference on theory and applications of satisfiability testing. pp. 502–518. Springer (2003)
9. Green, C.: Application of Theorem Proving to Problem Solving. In: Readings in Artificial Intelligence, pp. 202–222. Elsevier (1981)
10. Haaswijk, W., Mishchenko, A., Soeken, M., De Micheli, G.: SAT Based Exact Synthesis using DAG Topology Families. In: Proceedings of the 55th Annual Design Automation Conference. p. 53. ACM (2018)
11. King, J.C.: Symbolic Execution and Program Testing. Communications of the ACM **19**(7), 385–394 (1976)
12. Kolb, S., Paramonov, S., Guns, T., De Raedt, L.: Learning constraints in spreadsheets and tabular data. Machine Learning **106**(9), 1441–1468 (2017)
13. Kolb, S., Teso, S., Passerini, A., De Raedt, L.: Learning SMT(LRA) Constraints Using SMT Solvers. In: Proceedings of the 27th International Joint Conference on Artificial Intelligence. pp. 2333–2340. IJCAI'18 (2018)
14. Maimon, O., Rokach, L.: Data Mining and Knowledge Discovery Handbook, 2nd ed. Springer (2010)
15. Michalowski, M., Knoblock, C.A., Bayer, K., Choueiry, B.Y.: Exploiting Automatically Inferred Constraint-Models for Building Identification in Satellite Imagery. In: Proceedings of the 15th International Symposium on Advances in Geographic Information Systems (2007)
16. Muggleton, S., de Raedt, L.: Inductive Logic Programming: Theory and Methods. The Journal of Logic Programming **19-20**, 629 – 679 (1994)
17. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: Machine Learning in Python. Journal of Machine Learning Research **12**, 2825–2830 (2011)
18. Reynolds, A., Barbosa, H., Nötzli, A., Barrett, C., Tinelli, C.: cvc4sy: Smart and fast term enumeration for syntax-guided synthesis. In: Computer Aided Verification. pp. 74–83 (2019)
19. Sibson, R.: SLINK: An Optimally Efficient Algorithm for the Single-Link Cluster Method. The computer journal **16**(1), 30–34 (1973)
20. Silva, J.P.M., Sakallah, K.A.: GRASP-A New Search Algorithm for Satisfiability. In: Proceedings of the International Conference on Computer-Aided Design. pp. 220–227 (1996)
21. Steiner, W.: An Evaluation of SMT-Based Schedule Synthesis for Time-Triggered Multi-Hop Networks. In: Proceedings - Real-Time Systems Symposium. pp. 375–384 (2010)
22. Udupa, A., Raghavan, A., Deshmukh, J., Mador-Haim, S., Martin, M., Alur, R.: Transit: specifying protocols with concolic snippets. ACM SIGPLAN Notices **48**, 287 (2013)
23. Valiant, L.G.: A Theory of the Learnable. Commun. ACM **27**(11), 1134–1142 (1984)
24. Yordanov, B., Wintersteiger, C.M., Hamadi, Y., Kugler, H.: SMT-Based Analysis of Biological Computation. In: NASA Formal Methods. pp. 78–92. Springer Berlin Heidelberg (2013)