Learning Invariants using Decision Trees and Implication Counterexamples

Pranav Garg Daniel Neider P. Madhusudan Dan Roth
University of Illinois at Urbana-Champaign, USA
{garg11, neider2, madhu, danr}@illinois.edu

Abstract

Inductive invariants can be robustly synthesized using a learning model where the teacher is a program verifier who instructs the learner through concrete program configurations, classified as positive, negative, and implications. We propose the first learning algorithms in this model with implication counter-examples that are based on machine learning techniques. In particular, we extend classical decision-tree learning algorithms in machine learning to handle implication samples, building new scalable ways to construct small decision trees using statistical measures. We also develop a decision-tree learning algorithm in this model that is guaranteed to converge to the right concept (invariant) if one exists. We implement the learners and an appropriate teacher, and show that the resulting invariant synthesis is efficient and convergent for a large suite of programs.

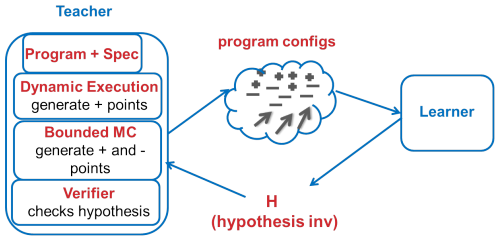
Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs: Invariants; D.2.4 [*Software Engineering*]: Software/Program Verification: Correctness proofs

Keywords Invariant synthesis, machine learning, decision trees, ICE learning

1. Introduction

Automatically synthesizing invariants, in the form of inductive pre / post conditions and loop invariants, is a challenging problem that lies at the heart of automated program verification. If an adequate inductive invariant is found or given by the user, the problem of checking whether the program satisfies the specification can be reduced to logical validity of verification conditions, which is increasingly tractable with the advances in automated logic solvers.

In recent years, the *black-box* or *learning* approach to finding invariants, which contrast white-box approaches such as interpolants, methods using Farkas' lemma, IC3, etc. [14, 18, 28, 29, 31, 38], have gained popularity [24, 25, 49, 51, 52]. In this data-driven approach, we split the synthesizer of invariants into two parts (see figure to the right). One component is a *teacher*, which is essentially a program verifier that can verify the program using a conjectured



invariant and generates counter-examples; it may also have other ways of generating configurations that must or must not be in the invariant (e.g., dynamic execution engines, bounded model-checking engines, etc.). The other component is a *learner*, which learns from counter-examples given by the teacher to synthesize the invariant. In each round, the learner proposes an invariant hypothesis H, and the teacher checks if the hypothesis is adequate to verify the program against the specification; if not, it returns concrete program configurations that are used in the next round by the learner to refine the conjecture. The most important feature of this framework is that the learner is completely agnostic of the program and the specification (and hence the semantics of the programming language, its memory model, etc.). The learner is simply constrained to learn some predicate that is consistent with the sample configurations given by the teacher.

ICE Learning Model: The simplest way for the teacher to refute an invariant is to give positive and negative program configurations, S^+ and S^-, constraining the learner to find a predicate that includes S^+ and excludes S^-. However, this is not always possible. In a recent paper, Garg et al. [25] note that if the learner gives a hypothesis that covers all states known to be positive by the teacher and excludes all states known to be negative by the teacher, but yet is not *inductive*, then the teacher is stuck and cannot give any positive or negative counter-example to refute the hypothesis.

Garg et al. [25] define a new learning model, which they call *ICE* (for implication counter-examples) that allows the teacher to give counter-examples of the form (x, y), where both x and y are program configurations, with the constraint that the learner must propose a predicate such that if the predicate includes x, then it includes y as well. These implication counter-examples can be used to refute non-inductive invariants: if H is not inductive, then the teacher can find a configuration x satisfying H such that x evolves to y in the program but y is not satisfied by H. This learning model forms a robust paradigm for learning invariants, including loop invariants, multiple loop invariants, and nested loop invariants in programs [25]—the teacher can be both *honest* (never give an example classification that precludes an invariant) and make *progress* (always be able to refute an invariant that is not inductive or adequate). This is in sharp contrast to learning only from positive

and negative examples, where the teacher is forced to be dishonest to make progress.

Machine Learning for Finding Invariants: One of the biggest advantages of the black-box learning paradigm is the possible usage of *machine learning techniques* to synthesize invariants. The learner, being completely agnostic to the program (its programming language, semantics, etc.), can be seen as a machine learning algorithm that learns a Boolean classifier of configurations [39]. Machine learning algorithms can be used to learn Boolean functions that belong to various classes, such as k-CNF/k-DNF, Linear Threshold Functions, Decisions Trees, etc., and some algorithms have already been used in invariant generation [50]. However standard machine learning algorithms for classification are trained on given sets of positive and negative examples, but do not handle implications and hence do not help in building robust learning platforms for invariant generation.

The goal of this paper is to adapt the standard decision tree learning algorithms to the ICE learning model in order to synthesize invariants. The key differences between current machine learning algorithms and the algorithms that we need are (a) to ensure that the algorithms build consistent classifiers, without errors, however small, (b) to ensure that they can learn from a sample that includes implication counter-examples and be consistent with them, (c) to adapt the statistical measures within the learning algorithm to take into account the existence of implication counterexamples, and (d) to ensure that the learning converges eventually to an invariant, if one exists.

Decision trees over a set of numerical and Boolean attributes provide a universal representation of Boolean functions composed of atomic formulas that are either inequalities bounding the numerical attributes by constants or Boolean attributes. Our starting points are the well-known decision tree learning algorithms of Quinlan [39, 44, 45] that work by constructing the tree top-down from positive and negative samples. These are efficient algorithms as they choose heuristically the best attribute that classifies the sample at each stage of the tree based on statistical measures, and do not backtrack nor look ahead. One of the well-known ways to pick these attributes is based on a notion called *information gain*, which is in turn based on a statistical measure using Shannon's entropy function [39, 44, 48]. The inductive bias in these learning algorithms is roughly to compute the *smallest* decision tree that is consistent with the sample—a bias that again suits our setting well, as we would like to construct the smallest invariant formula amongst the large number of invariants that may exist.

Machine learning algorithms, including the decision tree learning algorithm, often do not produce concepts that are fully consistent with the given sample—this is done on purpose to avoid over-fitting to the training set, under the assumption that, once learned, the hypothesis will be evaluated on new, previously unseen data. We first remove these aspects from the decision tree algorithm (which includes, for example, pruning to produce smaller trees at the cost of inconsistency) as we aim at identifying a hypothesis that is *correct* rather than one that only *approximates* the target hypothesis.

Our first technical contribution is a generic top-down decision tree algorithm that works on samples with implications. This algorithm constructs a tree top-down, with no backtracking, classifying end-points of implications as it builds the tree. Whenever the algorithm decides to classify a set of end-points, it immediately propagates the decisions across implication chains. By maintaining an invariant on the properties of the partial classification, it ensures that it will never end up in an inconsistent classification nor end up in a state where it cannot make progress. This ensures that we never need to backtrack to find new valuations to end-points. We prove that our algorithm, independent of what methods are used to pick attributes to split nodes, always results in a decision tree that is consistent with the sample (Section 3).

Our second technical contribution is a study of various natural measures for learning decision trees in the presence of positive, negative, and implication examples. We do this by developing several novel "information gain" measures that are used to determine the best attribute to split on the current collection of examples and implications. A naïve metric is to simply ignore implications when choosing attributes—however, ignoring implications entirely seems to result in non-robust invariant generation, where we found that even on simple examples the learning diverges. We propose two new metrics, one that imposes an additional penalty to the information gain measure that quantifies the number of implication pairs that are *cut* by a predicate, and the other which is a natural extension of the entropy measure to account for implications.

Our third contribution is to build a decision tree learning algorithm for the ICE model so that it is *convergent* in an iterative learning framework. In invariant synthesis, the passive learner for decision trees operates in conjunction with a verification oracle that takes hypothesized invariants produced by the learner and returns samples that exhibit why the hypothesized invariant is incorrect. We would like this iterative learning to terminate, but unfortunately, as the concept class is infinite, decision tree learning (both the classical one as well as the one we develop above for ICE) need not terminate. Hence, we build a new algorithm that uses the Occam principle to gain convergence—it strives to produce decision trees that use the *minimum* thresholds for any sample. We do this by setting a maximum threshold m, and learning decision trees where thresholds stay below m, incrementing m in an outer loop when such a tree does not exist. Decision tree building is extremely random (due to the statistical measures used to judge which attributes are best used for a split), and hence figuring out that *no tree with maximum threshold* m exists, without back-tracking, is challenging. We show how to build a polynomial time learning algorithm that either constructs the decision tree for a maximum threshold m or declares that no such tree exists. This gives us an invariant generation scheme that is guaranteed to converge to an invariant, if one exists that is expressible using the given attributes.

Finally, we implement our ICE decision tree algorithms (i.e., our generic ICE learning algorithm with the various statistical measures for choosing attributes) and build teachers to work with these learners to guide them towards learning invariants. We perform extensive experiments on a set of programs, with small and intricate loop invariants, taken from software verification competitions and the literature. We compare the tool with both invariant generation tools that use interpolation, as well as black-box ICE learning algorithms that use random search and constraint solvers. Despite being a fledgling algorithm, the first to use machine-learning algorithms adapted to the ICE setting, we show that the techniques using both statistical measures performs extremely well. On the class of more than 50 benchmarks, our tool is in fact the only one that finds adequate invariants for all of them.

The black-box approach to generating invariants is especially attractive in situations where a programmer has partially annotated programs (which can be complex, involving quantifiication, arrays, etc.) that need to be strengthened to prove a program correct. White-box techniques are not ready for this as they cannot typically handle such quantified invariants and strengthen them. However, black-box techniques follow a "guess-and-check" approach, which is readily adaptable to this setting. We report a few experiments where our tool is able to help strengthen partially annotated programs.

We believe that this work breaks new ground in adapting machine learning techniques to invariant synthesis, giving the first efficient robust ICE machine-learning algorithms for synthesizing invariants. The experiments give hope that this technique, which works well

on small intricate loops, holds promise in tackling larger problems, especially in helping lessen the annotation burden on programmers proving their code correct.

Related Work Algorithms for invariant synthesis can be broadly categorized as white-box techniques and black-box techniques. Prominent examples for white-box techniques include abstract interpretation [20], interpolation [31, 38], and IC3 [14]. Template-based approaches for synthesizing invariants using constraint solvers have been also explored in a white-box setting before [18, 28, 29]. On the other hand, a prominent early example of black-box techniques for synthesizing invariants is Daikon [22], which uses conjunctive Boolean learning to find *likely* invariants from program configurations recorded along dynamic test runs. A similar line of work has been explored more recently in [24, 41] for synthesizing likely polynomial and quantified list and array invariants, respectively. However, it is important to note that the invariants synthesized in this manner might not be *true* inductive invariants of the program. This shortcoming is gotten rid of in Houdini [23], which, like Daikon also uses conjunctive Boolean learning to learn conjunctive invariants over templates of atomic formulas but, has a constraint-solver based teacher to iteratively guide the conjunctive learner by providing counterexamples till the concept learned is inductive and adequate. Work on liquid types [32] uses a similar algorithm for inferring refinement types for program expressions.

Following these early works, learning was explicitly introduced in the context of verification by Cobleight et al. [16], followed by applications of Angluin's L* [8] to verification problems such as finding rely-guarantee contracts [5], stateful interface specifications for programs [4], verifying CTL properties [55], and Regular Model Checking [40].

Recently, learning has gained renewed interest in the context of program verification, particularly for synthesizing loop invariants [15, 24, 25, 34, 35, 49–52]. However, Garg et al. [25] argue that merely learning from positive and negative examples for synthesizing invariants is inherently non-robust and introduce ICE-learning, which extends the classical learning setting with implications. Implication counter-examples were also identified by Sharma et al [52], but the learners proposed did not handle them in any way. Examples of algorithms using ICE-learning have been subsequently proposed for learning invariants over octogonal domains and universally quantified invariants over linear data structures [24], Regular Model Checking [40], and a general framework for generating invariants based on randomized search [49]. Some generalizations of Houdini [26, 54] can also be seen as ICE-learning algorithms.

In *program* or *expression* synthesis, a popular approach to synthesis is using counter-example guided inductive synthesis (CEGIS), which is also a black-box learning paradigm [6, 53] like ICE, and is gaining traction aided by explicit enumeration, symbolic constraint-solving and stochastic search algorithms.

Machine learning algorithms (see [39] for an overview) are often used in practical learning scenarios due to their high scalability. Amongst the most well-known machine learning algorithms for learning linear classifiers are the winnow algorithm [36], perceptron [47], and support vector machines [19]. Since invariants in our current work are arbitrary Boolean functions, our learner is build on decision tree algorithms such as ID3 [45], C4.5 [44] and C5.0, and not linear classifiers. Apart from these classification algorithms, algorithms for learning more complicated structured objects using structured prediction [9] have also become popular recently.

Turning to general learning theory, the field of algorithmic learning theory is a well-established field [33, 39]. Several learning models like PAC [33] and exact learning by Angluin [7] are well known in machine learning literature. However, the learning setting closest to ICE-learning is perhaps the mistake bound learning model [36], in which a learner iteratively learns from incorrectly classified data and needs to converge to a correct classifier within a bounded number of wrong conjectures. In a passive setting, there has been some prior work on learning concepts in a semi-supervised setting with equivalence constraints (where unlabeled counterexample pairs need to be labeled with the same classification) [1, 13]. This setting is arguably easier to handle than handling directional implication constraints. We are not aware of any machine learning algorithm designed to learn from labeled (positive or negative) data as well as implications constraints.

2. Background: Learning Decision Trees from Positive and Negative Examples

Our algorithm for learning invariants builds on the classical recursive algorithms to build *decision trees*. (We refer the interested reader to standard texts on learning for more information on decision tree learning [39].) In particular, our learning algorithms are based on learning decision trees using the algorithms of Quinlan, where the tree is built top-down, choosing the best attribute at each stage using an information theoretic measure that computes the information gain in applying the attribute (quantified as the decrease in entropy on the resulting divided samples), and as implemented by the ID3, C4.0, and C5.0 algorithms [39, 44, 45].

Before we delve into the decision tree learning algorithm, let us first give the context in which these algorithms will be used for invariant synthesis. The reader is encouraged to think of decision trees as Boolean combinations of formulae of the form $a_i \leq c$, where a_i is drawn from a fixed set of *numerical attributes* A (which assign a numerical value to each sample) and c is a constant, or of the form b_i, where b_i is drawn from a fixed set of *Boolean attributes* (which assign a truth value to each sample). When performing invariant learning, we will fix a set of attributes typically as certain arithmetic combinations of integer variables (for example, octagonal combinations of variables of the form $\pm x \pm y$ for all pair of program variables x, y or certain linear combinations of variables with bounded co-efficients). Boolean attributes are useful for other non-numerical constraints (such as whether x and y aliased, does x point to *nil*, etc.). Consequently, the learner would learn the thresholds (the values for c in $a_i \leq c$) and the Boolean combination of the resulting predicates, including arbitrary conjunctions of disjunctions as a proposal for the invariant.

The Basic ID3 Algorithm Given a set of positive and negative samples, the classical approach to building a decision tree works by constructing the tree top-down (without backtracking), and is sketched in pseudo code as Algorithm 1. In each step, the algorithm chooses an attribute a from the given set of attributes (and possibly a threshold c if the attribute is numeric), and then applies the predicate defined by the attribute to the sample; this splits the sample into two sets, distinguished by the predicate— those satisfying the predicate and those that do not satisfy it. The algorithm then recurses on these two sets of samples, building decision trees for them, and then returns a tree where the root is labeled with the predicate and the left and right subtrees are those returned by the recursive calls. The base cases are when the sample is entirely positive or entirely negative—in this case the algorithm returns a tree with a single node that assigns the classification of these points appropriately as true or false, respectively.

The crucial aspect of the extremely scalable decision tree learning algorithms is that they choose the attribute for the current sample in some heuristic manner, and never back-track (or look forward) to optimize the size of the decision tree. The prominent technique for choosing attributes is based on a statistical property, called *information gain*, to measure how well each attribute classifies the examples at any stage of the algorithm. This measure is typically defined using a notion called *Shannon entropy* [48], which, intuitively, captures

input : A sample $S = \langle S^+, S^- \rangle$ and $Attributes$

1 Return `ID3` $(\langle S^+, S^- \rangle, Attributes)$.

Proc `ID3` *(Examples* $= \langle Pos, Neg \rangle$*, Attributes)*

2 Create a root node of the tree.

3 **if** *all examples are positive or all are negative* **then**

4 Return the single node tree Root with label $+$ or $-$, respectively.

else

5 Select an attribute a (and a threshold c for a if a is numerical) that (heuristically) *best* classifies *Examples*.

6 Label the root of the tree with this attribute (and threshold).

7 Divide *Examples* into two sets: $Examples_a$ that satisfy the predicate defined by attribute (and threshold), and $Examples_{\neg a}$ that do not.

8 Return tree with root and left tree `ID3` $(Examples_a, Attributes)$ and right subtree `ID3` $(Examples_{\neg a}, Attributes)$;

end

Algorithm 1: The basic inductive decision tree construction algorithm underlying ID3, C4.0, and C5.0

the impurity of a sample. The entropy of a sample S with p positive samples and n negative samples is defined to be

$$Entropy(S) = -\frac{p}{p+n}\log_2\frac{p}{p+n} - \frac{n}{p+n}\log_2\frac{n}{p+n}.$$

Intuitively, if the sample contains only positive (or only negative) points (i.e., if $p = 0$ or $n = 0$), then its entropy is 0, while if the sample had roughly an equal number of positive and negative points, then its entropy is high. Samples of lower entropy indicate that the classification is more complete, and hence are preferred.

When evaluating an attribute a (and threshold) on a sample S, splitting it into S_a and $S_{\neg a}$ (points satisfying the attribute and points that do not), one computes the information gain of that attribute (w.r.t. the chosen threshold): the information gain is the difference between the entropy of S and the sum of the entropies of S_a and $S_{\neg a}$ weighted by the number of points in the respective samples. For numerical attributes, the thresholds also need to be synthesized; in the case of information gain, however, it turns out that the best threshold is always at some value occurring in the points in the sample [44]. The algorithm chooses the attribute that results in the largest information gain. So, for example, an attribute that results in a split that causes two samples each with roughly half positive and half negative points will be less preferred than an attribute that results in a split that causes skewed samples (more positive than negative or vice versa).

The above heuristic for greedily picking the attribute that works best at each level has been shown to work very well in a wide variety of large machine learning applications [39, 44]. When decision tree learning is used in machine learning contexts, there are other important aspects: (a) the learning is achieved using a small portion of the available sample, so that the tree learned can be evaluated for accuracy against the rest of the sample, and (b) there is a *pruning* procedure that tries to generalize and reduce the size of the tree so that the tree does not overfit the sample. When using decision tree learning for synthesizing invariants, we prefer to use all the samples as we anyway place the passive learning algorithm in a larger context by building a teacher which is a verification oracle. Also, we completely avoid the pruning phase since pruning often produces trees that are (mildly) inconsistent with the sample; since we cannot tolerate any inconsistent trees, we prefer to avoid this (incorporating pruning in a meaningful and useful way in our setting is an interesting future direction).

In the context of invariant synthesis, we assume that all integer variables mentioned in the program occur explicitly as numerical attributes; hence, it turns out that *any* sample of mixed positive and negative points can be split (potentially using the same attribute repeatedly with different thresholds) and eventually separated into purely positive and purely negative points (in the worst case, each point is separated into its own leaf). Consequently, we are guaranteed to always obtain some decision tree that is consistent with any input sample.

3. A Generic Decision Tree Learning Algorithm in the Presence of Implications

In this section, we present the skeleton of our new decision tree learning algorithm for samples containing implication examples in addition to positive and negative examples. We present this algorithm at the same level of detail as we presented Algorithm 1, excluding the details of *how* the best attribute at each stage is chosen. In Section 4, we articulate several different natural ways of choosing the best attribute, and evaluate them in experiments.

Our goal in this section is to build a top-down construction of a decision tree for an *ICE sample*, such that the tree is guaranteed to be consistent with respect to the sample; an ICE sample is a tuple $S = (S^+, S^-, S^{\Rightarrow})$ consisting of a finite set S^+ of positive points, a finite set S^- of negative points, and a finite set $S^{\Rightarrow}$ of pairs of points corresponding to implication counterexamples. The algorithm is an extension of the classical decision tree algorithm presented in Section 2, which preserves the property to be consistent with positive and negative samples. The main hurdle we need to cross is to construct a tree consistent with the implication counterexamples. Note that the pairs of points corresponding to implications do not have a classification, and it is the learner's task to come up with a classification in a manner consistent with the implication constraints. As part of the design, we would like the learner to not classify the points a priori in any way, but classify these points in a way that leads to a smaller concept (or tree).

Our algorithm, shown in pseudo code as Algorithm 2, works as follows. First, given an ICE sample $\langle S^+, S^-, S^{\Rightarrow} \rangle$ and a set of attributes, we store $S^{\Rightarrow}$ in a global variable $Impl$ and create a set *Unclass* of unclassified points as the end-points of the implication samples. We also create a global table G that holds the partial classification of all the unclassified points (initially empty). We then call our recursive decision tree constructor with the sample $\langle S^+, S^-, Unclass \rangle$.

Receiving a sample $\langle Pos, Neg, Unclass \rangle$ of positive, negative, and unclassified examples, our algorithm chooses the best attribute that divides this sample, say a, and recurses on the two resulting samples $Examples_a$ and $Examples_{\neg a}$. Unlike the classical learning algorithm, we do not recurse *independently* on the two sets—rather we recurse first on $Examples_a$, which will, while constructing the left subtree, make classification decisions on some of the unclassified points, which in turn will affect the construction of the right subtree for $Examples_{\neg a}$ (see the `else` clause in Algorithm 2). The new classifications that are decided by the algorithm are stored and communicated using the global variable G.

Whenever Algorithm 2 reaches a node where the current sample has only positive points and implication end-points that are either classified positively or unclassified yet, the algorithm will, naturally, decide to mark *all* remaining unclassified points positive, and declare the current node to be a leaf of the tree (see first conditional in the algorithm). Moreover, it marks in G all the unclassified end-points of implications in *Unclass* as positive and propagates this constraint across implications (taking the implication closure of G with respect to the global set $Impl$ of implications). For instance, if (x, y) is an implication pair, both x and y are yet unclassified, and the algorithm decides to classify x as positive, it propagates this constraint by making y also positive in G; similarly, if the algorithm decided

input : An ICE sample $S = \langle S^+, S^-, S^{\Rightarrow} \rangle$ and *Attributes*

1 Initialize global *Impl* to $S^{\Rightarrow}$.
2 Initialize G, a partial valuation of end-points of implications in *Impl* , to be empty.
3 Let *Unclass* be the set of all end-points of implications in *Impl* .
4 Set G to be the implication closure of the positive and negative classifications in S^+ and S^- with respect to *Impl*.
5 Return `DecTreeICE` $(\langle S^+, S^-, \mathit{Unclass} \rangle, \mathit{Attributes})$.

Proc `DecTreeICE` *(Examples* $= \langle \mathit{Pos}, \mathit{Neg}, \mathit{Unclass} \rangle$*, Attributes)*
6 Move all points from *Unclass* that are positively, respectively negatively, classified in G to *Pos*, respectively *Neg*.
7 Create a root node of the tree.
8 **if** $\mathit{Neg} = \emptyset$ **then**
9 Mark all elements in *Unclass* as positive in G.
10 Take the implication closure of G w.r.t. *Impl*.
11 Return the single node tree Root, with label $+$.
12 **else if** $\mathit{Pos} = \emptyset$ **then**
13 Mark all elements in *Unclass* as negative in G.
14 Take the implication closure of G w.r.t. *Impl*.
15 Return the single node tree Root, with label $-$.
16 **else**
17 Select an attribute a (and a threshold c for a if a is numerical) that (heuristically) *best* classifies *Examples* and *Impl*.
18 Label the root of the tree with this attribute a (and threshold c).
19 Divide *Examples* into two sets: $\mathit{Examples}_a$ that satisfy the predicate defined by the attribute (and threshold) and $\mathit{Examples}_{\neg a}$ that do not.
20 $T_1 = \mathit{DecTreeICE}(\mathit{Examples}_a, \mathit{Attributes})$ (may update G).
21 $T_2 = \mathit{DecTreeICE}(\mathit{Examples}_{\neg a}, \mathit{Attributes})$ (may update G).
22 Return tree with root, left tree T_1, and right tree T_2.
end

Algorithm 2: The basic decision tree construction algorithm for ICE samples

to classify y as negative, then it would mark x also as negative in G. Deciding to classify x as negative or y as positive places no restrictions on the other point, of course.

We need to argue why Algorithm 2 always results in a terminating procedure that constructs a decision tree consistent with the sample. As a preparation, we introduce a property of the sample and the partial valuation for the implication end-points, called a *valid sample*.

In the following description, let us fix an ICE sample $S = \langle S^+, S^-, S^{\Rightarrow} \rangle$, and let G be a partial valuation of end-points of in $S^{\Rightarrow}$. By abuse of notation, we use $S \cup G$ to refer to the sample one obtains from classifying the end-points of implications in $S^{\Rightarrow}$ with the (partial) valuation of G.

Definition 1 (Valid sample)**.** *A sample $S = \langle S^+, S^-, S^{\Rightarrow} \rangle$ is* valid *if for every implication $(x, y) \in S^{\Rightarrow}$*

1. *it is not the case that x is classified positively and y negatively in $S \cup G$;*
2. *it is not the case that x is classified positively and y is unclassified in $S \cup G$; and*
3. *it is not the case that y is classified negatively and x is unclassified in $S \cup G$.*

A valid sample has the following property.

Lemma 1. *For any valid sample (with partially classified end-points of implications), extending it by classifying all unclassified points as positive will result in a consistent classification, and extending it by classifying all unclassified points as negative will also result in a consistent classification.*

Proof of Lemma 1. First, consider the extension of a valid sample by classifying all unclassified points as positive. Assume, for the sake of contradiction, that this valuation is inconsistent. Then, there exists an implication pair (x, y) such that x is classified as positive and y is classified as negative. Since such an implication pair could not have already existed in the valid sample (by definition), it must have been caused by the extension. Since we introduced only positive classifications, it must have been that x (and not y) is the only new classification. Hence the valid sample must have had the implication pair (x, y) with y classified as negative and x being unclassified, which contradicts Condition 3 of Definition 1. The proof of the extension with negative classifications follows from similar arguments. □

However, Algorithm 2 does not classify *all* implication end-points completely positive, or completely negative; recall that Algorithm 2 only changes the classification (from unknown to positive or negative, respectively) of unclassified implication end-points in the current leaf and those that need to be updated during the implication closure. It is not hard to verify that even such a partial assignment preserves consistency of an ICE sample.

Corollary 1. *Lemma 1 also holds if a subset of unclassified points are classified (completely positively or completely negatively) and the implication closure is taken.*

It is now straightforward to prove the correctness of the decision tree ICE learner.

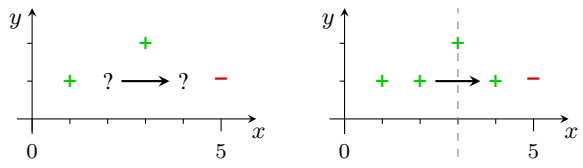
Theorem 1. *Algorithm 2, independent of how the attributes are chosen to split a sample, always terminates and produces a decision tree that is consistent with the input ICE sample.*

Proof of Theorem 1. Theorem 1 follows from the fact that Algorithm 2 always maintains a valid sample:

1. Algorithm 2 receives an ICE sample and applies the implication closure, which results in a valid sample (or an inconsistency is detected and the learning stops as there does not exist a decision tree that classifies the sample correctly while satisfying the implication constraints).
2. When Algorithm 2 arrives at a leaf that has only *positive and unclassified* points, it classifies all these unclassified points to be positive and takes the implication closure. Assuming that the ICE sample was valid, the new sample is also valid due to Corollary 1. In the case that Algorithm 2 arrives at a leaf that has only *negative and unclassified* points, validity of the resulting sample follows using similar arguments.

The above argument shows that Algorithm 2 never ends up in an inconsistent sample, which proves its correctness. Moreover, since we assume that every integer variable is also included as a numerical attribute by itself (i.e., an attribute x exists for each integer variable x), there always will be a numerical attribute that can split any sample with a positive and a negative example, in Line 17 of procedure ProcDecTree, which proves termination as well. □

The running time of Algorithm 2 is $O(nlogn \times m \times |T|)$ for a sample of size n, when the total number of attributes is m, and when the learner learns a tree of size $|T|$. Decision tree algorithms, in general, scale very well with respect to number of attributes and the sample. In Section 6, we report on scalability of our decision-tree based learning algorithm with respect to the size of the sample. In the next section, we explore several different ways to choose the best attribute, at each stage of the decision tree construction, all of which are linear or sub-quadratic on the sample.



(a) Sample before splitting (b) Samples after splitting

Figure 1: The samples discussed in Example 1.

4. Choosing Attributes in the Presence of Implications

Algorithm 2 always constructs a decision tree that is consistent with the given sample, irrespective of the exact mechanism used to determine the attributes to split and their thresholds. As a consequence, the original split heuristic based on information gain (see Section 2), which is unaware of implications, can simply be employed. However, since our algorithm propagates the classification of data points once a leaf node is reached, just ignoring implications can easily lead to splits that are good at the time of the split but later turn into bad ones since the classification of points has changed. The following example illustrates such a situation.

Example 1. *Suppose that Algorithm 2 processes the sample shown in Figure 1a, which also depicts the (only) implication in the global set Impl.*

When using the original split procedure (i.e., using information gain while ignoring the implication and its corresponding unclassified data points), the learner splits the sample with respect to attribute x at threshold $c = 3$ since this split yields the highest information gain—the information gain is 1 since the entropy of the resulting two subsamples is 0. Using this split, the learner partitions the sample into $Examples_a$ and $Examples_{\neg a}$ and recurses on $Examples_a$. Since $Examples_a$ contains only unclassified and positively classified points, it turns all unclassified points in this sample positive and propagates this information along the implication. This results in the situation depicted in Figure 1b. Note that the learner now needs to split $Examples_{\neg a}$ since the unclassified data points in it are now classified positively.

On the other hand, if we consider the implications and their corresponding data points while deciding the split, it allows us to split the sample such that the antecedent and the consequent of the implication both belong to either $Examples_a$ or $Examples_{\neg a}$ (e.g., on splitting with respect to x with threshold $c = 4$). Such a split has the advantage that no further splits are required and, often, results in a smaller tree. ⌟

In fact, some experiments we did showed that a learner that ignores implications when choosing an attribute often learns relatively large decision trees, and even diverges in the iterative setting. Hence, we next propose two methods that take implications into account while choosing the attribute to split.

Penalizing Cutting Implications In order to better understand how to deal with implications, we analyzed classifiers learned by other ICE-learning algorithms for invariant synthesis, such as the constraint solver-based ICE learner of [25]. This analysis showed that the classifiers finally learned (and also those conjectured during the learning) almost always classify the both endpoints of implications the same way (either both positive or both negative).

The fact that successful ICE learners almost always endpoints of implications the same way suggests that our decision tree learner should avoid to "cut" implications. This is a heuristic which assumes that, since the endpoints of the implication counterexamples will eventually be likely labeled with the same classification, they would often reach the same leaf node in the learned tree. Formally, we say that an implication $(p, q) \in Impl$ is *cut* by the samples S_a and $S_{\neg a}$ if $p \in S_a$ and $q \in S_{\neg a}$, or $p \in S_{\neg a}$ and $q \in S_a$;[1] in this case, we also say that the split of S into S_a and $S_{\neg a}$ cuts the implication.

A straightforward approach to discourage cutting implications builds on top of the original information gain and imposes a penalty for every implication that is cut. This idea gives rise to the *penalized information gain* that we define by

$$Gain_{pen}(S, S_a, S_{\neg a}) = Gain(S, S_a, S_{\neg a}) - Penalty(S_a, S_{\neg a}, Impl) \quad (1)$$

where $S_a, S_{\neg a}$ is the split of the sample S, $Gain(S, S_a, S_{\neg a})$ is the original information gain based on Shannon's entropy, and $Penalty(S_a, S_{\neg a}, Impl)$ is a total penalty function that we assume to be monotonically increasing with the number of implications cut (we make this precise shortly). Note that this new information gain does not prevent the cutting of implications (if required) but favors not to cut them.

However, not every cut implication poses a problem: implications whose antecedents are classified negatively and whose consequents are classified positively are safe to cut (as this helps creating more pure samples), and we do not want to penalize cutting those. Since we do not know the classifications of unclassified points when choosing an attribute, we penalize an implication depending on how "likely" it is an implication of this type (i.e., we assign no penalty if the sample containing the antecedent is predominantly negative and the one containing the consequent is predominantly positive). More precisely, given the samples S_a and $S_{\neg a}$, we define the penalty function $Penalty(S_a, S_{\neg a}, Impl)$ by

$$\Big(\sum_{\substack{(x,y)\in Impl \\ x\in S_a, y\in S_{\neg a}}} 1 - f(S_a, S_{\neg a})\Big) + \Big(\sum_{\substack{(x,y)\in Impl \\ x\in S_{\neg a}, y\in S_a}} 1 - f(S_{\neg a}, S_a)\Big),$$

where for two samples $S_1 = \langle Pos_1, Neg_1, Unclass_1\rangle$, $S_2 = \langle Pos_2, Neg_2, Unclass_2\rangle$,

$$f(S_1, S_2) = \frac{|Neg_1|}{|Pos_1| + |Neg_1|} \cdot \frac{|Pos_2|}{|Pos_2| + |Neg_2|}$$

is the relative frequency of the negative points in S_1 and the positive points in S_2 (which can be interpreted as likelihood of an implication from S_1 to S_2 being safe).

Extending Entropy to ICE Samples In the second variant of information gain that we develop for deciding the *best* attribute to split a given ICE sample, we do not change the definition of information gain (as a function of the entropy of the sample) but we extend Shannon's entropy to deal with implications in the sample using conditional probabilities. The entropy of a set of examples is a function of the discrete probability distribution of the classification of a point drawn randomly from the examples. In a classical sample that only has points labeled positive or negative, one could count the fraction of positive (or negative) points in the set to compute these probabilities. However, an estimation of these probabilities becomes non-trivial in the presence of unclassified points that can be classified as either positive or negative. Moreover, in an ICE sample, the classification of these points is not independent anymore as the classification for the points need to satisfy the implication constraints. Given a set of examples with implications and unclassified points, we will first estimate the probability distribution of the classification of a random point drawn from these examples, taking into account the implication constraints, and then use it for computing the entropy. We will use

[1] Given a sample $S = \langle Pos, Neg, Unclass\rangle$, we write $x \in S$ as a shorthand notation for $x \in Pos \cup Neg \cup Unclass$.

this new entropy to compute the information gain while choosing the attribute for the split.

Given $S = \langle Pos, Neg, Unclass \rangle$, and a set of implications *Impl*, let $Impl_S$ be the set of implications projected onto S such that both the antecedent and consequent end-points in the implication are unclassified (i.e., $Impl_S = \{(x_1, x_2) \in Impl \mid x_1, x_2 \in Unclass\}$). For the purpose of entropy computation, we will assume that there is no point in the examples that is common to more than one implication. This is a simplifying assumption, which also holds statistically if the space enclosing all the points is much larger than the number of points. Let $Unclass' \subseteq Unclass$ be the set of unclassified points in the sample that are not part of any implication in $Impl_S$ (for example, $x_1 \in Unclass'$ if $(x_1, x_2) \in Impl$ *and* $x_2 \in Pos$). Note that points in $Unclass'$ can be classified as either positive or negative by the learner, completely independent of the classification of any other point. This is, for instance, not true for points that are end-points of implications in $Impl_S$.

Let $Pr(x = c)$ be the probability of the event that a point $x \in S$ is classified as c, where c is either positive/$+$ or negative/$-$. Note that $Pr(x = +)$ is 1 when $x \in Pos$ and is 0 when $x \in Neg$. Let us define $Pr(S, c)$ to be the probability of the event that a point which is drawn randomly from S is classified as c. Then,

$$\begin{aligned} Pr(S,+) &= \frac{1}{|S|} \sum_{x \in S} Pr(x = +) \\ &= \frac{1}{|S|} \Big(\sum_{x \in Pos \cup Neg \cup Unclass'} Pr(x = +) \quad + \\ &\qquad \sum_{(x_1,x_2) \in Impl_S} Pr(x_1 = +) \quad + \quad Pr(x_2 = +) \Big) \end{aligned} \tag{2}$$

Recall that unclassified points $x_u \in Unclass'$ are statistically classified by the learner completely independent of other points in the sample; so, we assume that the probability that a point x_u is classified as positive (or negative) is in accordance with the distribution of points in the sample set S. In other words, we recursively assign $Pr(x_u = +) = Pr(S, +)$.

For points x_1 and x_2 that are involved in an implication $Impl_S$, we assume that the antecedents x_1 are classified independently and the classification of consquents x_2 is conditionally dependent on the classification of antecedents, such that the implication constraint is satisfied. As a result, we assign $Pr(x_1 = +) = Pr(S, +)$, for the same reason as described for x_u above. And for consequents x_2, using conditional probabilities we obtain, $Pr(x_2 = +) = Pr(x_2 = + \mid x_1 = +) \cdot Pr(x_1 = +) + Pr(x_2 = + \mid x_1 = -) \cdot Pr(x_1 = -)$. From the implication constraint between x_1 and x_2, we know that x_2 is guaranteed to be positive if x_1 is classified positive, i.e., $Pr(x_2 = + \mid x_1 = +) = 1$. However, when x_1 is classified negative, the consequent x_2 is allowed to be classified as either positive or negative completely independently, and hence we assign $Pr(x_2 = + \mid x_1 = -) = Pr(S, +)$.

Plugging in these values for probabilities in Equation 2 and using $p = |Pos|, n = |Neg|, i = |Impl|, u' = |Unclass' \mid$ *and* $|S| = p + n + 2i + u'$, $Pr(S, +)$ is the positive solution of the following quadratic equation:

$$ix^2 + (p + n - i)x - p = 0$$

As a sanity check, note that $Pr(S, +) = \frac{p}{p+n}$, if there are no implications in the sample set (i.e., $i = 0$). Also, $Pr(S, +) = 1$ if $n = 0$ and $Pr(S, +) = 0$ if $p = 0$ (i.e., when the set S has no negative or positive points). Once we have computed $Pr(S, +)$, the entropy of S can be computed in the standard way as

$$\begin{aligned} Entropy(S) = &-Pr(S,+) \cdot \log_2 Pr(S,+) \\ &- Pr(S,-) \cdot \log_2 Pr(S,-) \end{aligned}$$

where $Pr(S, -) = \big(1 - Pr(S, +)\big)$. Now, we plug this new entropy in the information gain and obtain a gain measure that explicitly takes implications into account.

5. Convergent Learning of Decision Trees

In this section, we build a decision-tree learner that is guaranteed to terminate and produce an invariant for any program, provided the invariant is expressible as a formula involving the numerical and Boolean attributes the learner has been instantiated with. Note that decision-tree learning for a sample always terminates (see Theorem 1); we are concerned here with the termination of the iterative learning algorithm that works with the teacher to learn an invariant. The set of formulas involving numerical and Boolean attributes is *infinite* and hence the iterative learning algorithm presented thus far need not converge.

If we bound the maximum thresholds that occur in the inequalities associated with numerical attributes, then, clearly, the set of all semantically different Boolean formulas that respect this restriction is bounded as well. Our strategy for convergence is to *iteratively* bound the (absolute value) of the maximum threshold, thereby effectively searching for decision trees with minimal thresholds, and growing the threshold only if we can *prove* that the current threshold is insufficient to build a classifier consistent with the given sample. Note that this search biases the learner towards finding concepts with smaller thresholds, which is an Occam razor that we are happy with, as we would like to infer invariants that have smaller constants before exploring those with larger constants. Clearly, if there is some invariant expressible using the numerical and Boolean attributes, the learner will never increment the maximum threshold beyond the maximum absolute constant used in this invariant. Moreover, since there are only a finitely many (semantically different) formulae with a bounded maximum threshold and since the learner can never propose a conjecture twice (due to the fact that it always constructs conjectures that are consistent with the current sample, which contains a counterexample to any previous conjecture), the learner must find a valid invariant.

Let us assume a set of numerical attributes $A = \{a_1, \ldots, a_n\}$ and let us assume (for simplicity) that there are no Boolean attributes. Then the set of all Boolean formulae with absolute maximum threshold m are defined to be the set of all Boolean combinations of atomic predicates of the form $a_i \leq c$, where $|c| \leq m$. For any $m \in \mathbb{N}$, the set of (semantically different) formulae with absolute maximum threshold m is bounded.

In the setting when we have only positively/negatively labeled samples (as opposed to ICE samples), such an algorithm is actually easy to build, since we can easily know when no formula with maximum threshold m can be consistent with the sample. We simply take the standard C5 algorithm, and modify the learner so that when processing any node n of the tree with a sample (S^+, S^-) (line 5 of Algorithm 1), we only look at predicates involving an attribute and a threshold c, where $|c|$ is at most m. If we succeed in building a tree, then we would have learned a formula with absolute maximum threshold m. However, if we get stuck, then it must be the case that we are at a node n with sample (S^+, S^-), and *no* predicate with threshold at most m is able to separate a positive point $s^+ \in S^+$ and a negative point $s^- \in S^-$. This of course means that these two points are not separable by any Boolean formula with maximum threshold m, and hence we can safely increment m, and iterate.

The above, however, fails in the ICE setting! The problem is that the learner may have made some choices regarding the valuations of end-points of implication counterexamples, and *this choice* may lead it to get to a node where there are two points s^+ and s^- that are not separable (where these points were not marked to be positive and negative in the original sample, but by the learner). Backtracking from such a scenario to consider some other valuation is dangerous as the learner may no longer run in polynomial time. The goal of

this section is to show how to build a passive decision-tree learner that takes an ICE sample S and an absolute maximum threshold m, and either produces a decision-tree consistent with S or declares that none exists, and runs in polynomial time in both $|A|$ and $|S|$ and is *independent* of m.

Before we describe the precise algorithm, let us explain the underlying idea that makes it work. Consider two points s, s' in the sample (i.e., s and s' occur as classified points or as some end-point of an implication). Now, assume that there is *no* atomic predicate of the form $a_i \leq c$ with $|c| \leq m$ that separates these points (i.e., which evaluates to true on one and false on the other). Then, no matter what final classification we come up with using a decision tree, the classification of s and s' will be necessarily identical: if s was classified already to be positive (or negative), then s' must have the same classification; if s and s' are both unclassified, then we need to classify them both to be positive or both to be negative.

The key idea now is to identify *all* such pairs of inseparable points s and s', and add the *implications* (s, s') and (s', s) to the ICE sample, which captures the condition that they need to be labeled with the same classification. We can then simply run our ICE decision tree learner on this augmented ICE sample. In fact, we can do even better— when it is not possible to construct any decision-tree, consistent with the ICE sample, with absolute threshold values at most m, we also detect this using a pre-processing check.

Let us formally define the way we add new implications for a maximum threshold $m \in \mathbb{N}$. Assume that we have an ICE sample $S = (S^+, S^-, S^{\Rightarrow})$, and let T be the set of points occurring in S (classified or otherwise). Given T, we now define the following equivalence class $\equiv_m$ on T:

$$s \equiv_m s' \text{ iff there is no predicate of the form } a_i \leq c \text{ with } |c| \leq m \text{ that separates } s \text{ and } s'.$$

Based on this equivalence relation, we now augment the original sample S to obtain what we call the *m-augmented ICE sample of S*, denoted by $S \oplus m$. We proceed in two steps. First, we construct a new sample $S' = (S^+, S^-, S^{\Rightarrow} \cup E)$ by taking S and adding implications $E = \{(s, s') \mid s \equiv_m s'\}$ to $S^{\Rightarrow}$ (i.e., we add implications between all $\equiv_m$ points). Then, we take the implication closure of S', which produces the m-augmented ICE sample $S \oplus m$. Recall that the implication closure repeats the following steps until a fixed point is reached: it takes implications (s, s') where s is positively classified and adds s' as positive to the sample; similarly, it takes implications (s, s') where s' is negatively classified, and adds s as negative to the sample.

The notion of valid samples (see Definition 1), which intuitively states that a sample is implication-closed and has no inconsistent implications, now gives us a handle to check whether a Boolean formula with absolute maximum threshold m that is consistent with S exists or not. More precisely, we can show that:

a) If $S \oplus m$ is not valid, then there is *no Boolean formula with absolute maximum threshold* m that is consistent with S. In the overall learning loop, we would increment m in this case and restart learning from S.

b) If $S \oplus m$ is valid, then calling our ICE decision tree learner (i.e., Algorithm 2) on $S \oplus m$ while restricting it to predicates that use thresholds with absolute values at most m is *guaranteed to terminate and return a tree that is consistent with S*.

Before we prove these claims, let us describe in more detail the modification of Algorithm 2 mentioned in Part (b). To force Algorithm 2 to only split a sample with a threshold whose absolute value is less or equal to some bound $m \in \mathbb{N}$, it is indeed enough to modify Line 17 such that the algorithm only considers thresholds c that satisfy $|c| \leq m$. We call the learning algorithm resulting from this simple modification the *bounded ICE decision tree learner*, or *bounded ICE-DT learner* for short.

With this defined, we can now prove both claims.

Proof of Part (a). Let S be an ICE sample and $m \in \mathbb{N}$. Assume, for the sake of a contradiction, that $S \oplus m = (S^+, S^-, S^{\Rightarrow})$ is not valid but there is a formula f with absolute maximum threshold m that is consistent with S (where consistency is defined in the obvious way as classifying points correctly and respecting implications). Since $S \oplus m$ is implication-closed (by definition), the only property of validity that can fail is that there exist s, s' such that $s \in S^+$, $s' \in S^-$, and $(s, s') \in S^{\Rightarrow}$. Since f is consistent with S and has maximum threshold m, it satisfies all the new implication constraints that were added between $\equiv_m$-equivalent points. Hence f is also consistent with its implication closure, which is $S \oplus m$. This, however, is a contradiction as f must classify s as positive and s' as negative, and hence does not satisfy the implication (s, s'). □

Proof of Part (b). Again, let S be an ICE sample and $m \in \mathbb{N}$. Moreover, assume that $S \oplus m$ is a valid sample. Analogous to the correctness proof of Algorithm 2 (i.e., Theorem 1), we know that when given a valid initial sample, the bounded ICE-DT learner maintains validity and produce a decision tree that is consistent with the sample. However, it is left to show that the bounded ICE-DT learner is guaranteed to terminate. In particular, we need to show that in any node that is not a leaf, it will be able to find some predicate (using a threshold $|c| \leq m$) according to which it can split the current (sub-)sample. This was guaranteed in Algorithm 2 due to the fact that two points can always be separated (using an attribute that is different for both points and a suitable threshold), which does not hold now because we have restricted thresholds to those whose absolute values is at most m. We show now, however, that a split with threshold $|c| \leq m$ is always possible due to the validity of $S \oplus m$.

Assume that the bounded ICE-DT learner processes a node with the (sub-)sample $S' = (S^+, S^-, S^{\Rightarrow})$ of $S \oplus m$ containing a positive point $p \in S^+$ and a negative point $n \in S^-$. Moreover, assume that it cannot find an attribute and threshold $|c| \leq m$ that separates p and n. Hence, it follows that $p \equiv_m n$, and S' contains the implication pair (p, n) as this pair is added to the initial sample and the set of implications $S^{\Rightarrow}$ does not change during the recursive construction of the tree. This means that S' is not valid since $p \in S^+$, $n \in S^-$, and $(p, n) \in S^{\Rightarrow}$. However, this is a contradiction since the bounded ICE-DT learner starts with a valid sample $S \oplus m$, and maintains valid samples throughout, which implies that S' must be valid. □

The algorithm sketched in pseudo code as Algorithm 3 produces a consistent decision tree with minimal threshold $m \in \mathbb{N}$, which can be seen as follows. Since any valid sample allows for a consistent decision tree, there exists a threshold m^* for which $S \oplus m^*$ is valid and, therefore, Algorithm 3 terminates. Moreover, Algorithm 3 starts with $m = 0$ and increases m by one only if the sample $S \oplus m$ is not valid (i.e., there is no Boolean formula with absolute threshold m that is consistent with S), hence, it will produce a tree with minimal threshold.

Building $S \oplus m$ and checking whether it is valid can be done in poynomial-time. Computing $\equiv_m$ is a partition refinement algorithm: in each refinement step, we consider an attribute a and *only those values for thresholds that occur in the sample* plus the thresholds $-m$ and $+m$, and refine the equivalence class based on the elements of the sample it can separate. Consequently, computing $S \oplus m$ can be done in polynomial time. Furthermore, the bounded DT-ICE learner also considers only those values for thresholds that occur in the sample plus the thresholds $-m$ and $+m$. Consequently, it too will work polynomial in the input sample.

input : An valid ICE sample $S = \langle S^+, S^-, S^{\Rightarrow} \rangle$ and *Attributes*

$m \leftarrow 0$.
$T \leftarrow$ set of all points occurring in the sample.
while *(true)* **do**
  Compute $\equiv_m$ on T.
  Construct $S \oplus m$.
  if *($S \oplus m$ is valid)* **then**
    Run the bounded ICE-DT learner with bound m and return the decision tree it constructs.
  else
    $m \leftarrow m + 1$.
  end
end

Algorithm 3: Convergent ICE decision-tree learning algorithm.

The following theorem states the main result of this section, namely that when paired with a honest teacher, Algorithm 3 always terminates and returns an invariant given that one exists.

Theorem 2. *Fix a set of numerical attributes and Boolean attributes over a set of program configurations. Consider a program P which has an inductive invariant that proves it satisfies a safety specification, where the invariant is expressible as a Boolean combination of the attributes with inequalities with respect to integers. Then Algorithm 3 will converge on an inductive invariant of the program P when paired with any honest teacher.*

6. Experiments and Evaluation

To assess the performance of our decision tree ICE learner, we implemented a prototype of Algorithm 2, with the two information gain measures, described in Section 4, as an invariant synthesis tool for Boogie [10] programs and compare it to other invariant generation algorithms. We do not implement the convergent learner, and in the experiments we report here, our learners had no converging on an inductive invariant. We conducted all of the following experiments on a Core i5 CPU with 6 GB of RAM running Windows 7 with a 10 minute timeout limit.

Learner: We implemented the learning algorithm on top of the freely available version of the C5.0 algorithm (Release 2.10) [44]. Since we rely on learning without classification errors, we disable all of C5.0's optimizations, such as pruning, boosting, etc.

Furthermore, though our learning algorithm can work with any class of predicates (including non-linear predicates), we parameterize it, by default, with the class of octagonal predicates (of the form $\pm x \pm y \leq c$). More precisely, we add all numerical attributes of the form $\pm x \pm y$ for all combinations of variables x, y in the program (note that the learner learns the thresholds c as well as the Boolean combination of these predicates). This class of predicates is sufficient to express most invariants in our benchmark and, furthermore, all black-box learners that we compare with are also instantiated, by default, with this class of predicates.

Teacher: We implemented a teacher in Boogie [10], which generates verification conditions for a given input program, and comes with interfaces for invariant synthesis tools like Houdini [23]. We build over this to give a richer interface that returns numerical predicates in addition to Boolean predicates. The teacher prioritizes returning positive/negative counterexamples to implication counterexamples. Since loop invariants usually do not involve large constants, the teacher biases the learner towards trees with smaller thresholds by producing counterexamples that have small values. When searching for counterexamples, we iteratively bound the absolute values of the variables to 2, 5, 10, and ∞ till we find a counterexample.

Experimental Results for Invariant Generation: We evaluate the two configurations of the decision-tree based learner discussed in Section 4 in the context of invariant synthesis and compare them to various other invariant synthesis algorithms. The experimental results are tabulated in Table 1, and the scatter-plots comparing the time taken by solvers is depicted in Figure 2.

Solvers: We first tabulate times of CPAchecker [12], which is a white-box state-of-the-art verifier; we use the configuration that corresponds to the predicate abstraction and the interpolation [37] based refinement. Note that CPAchecker has a disadvantage as it does not restrict itself to finding Boolean combinations of octagonal constraints, while all the black-box learners do. Next we tabulate results for the black-box learners. The first is a randomized search based invariant synthesis tool [49]; this uses its own in-built teacher. The second is the constraint-solver based learner from [25] (called ICE-CS); this uses a teacher that does not bound counterexamples (as its in-built search bounds the constants in predicates iteratively anyway). The last two columns depict the new decision tree learners in this paper, from Section 4, which includes the learner that computes the information gain which accounts for implications (called ICE-DT-entropy) and the learner that penalizes splits that cut implications (called ICE-DT-penalty).

Benchmarks: We report results for a suite of programs[2], which includes *all* programs from the SV-COMP (2014 and 2015) benchmark "loop" suite [2] (which involves small programs but with intricate loop invariants) and programs from the literature [3, 25, 27, 28, 30]. The SV-COMP examples which were buggy programs are excluded, of course, as our focus is on proving correct programs correct[3] Furthermore, we exclude SV-COMP benchmarks where the loop invariant involved arrays[4] as well as programs where the invariant is extremely simple (like *true* or $x = 0$), since all tools took negligible time. Some programs in Table 1 are the natural unbounded versions of programs that were artificially bounded in SV-COMP to simplify them. We additionally have a few benchmarks that require invariants over non-linear integer arithmetic, and also some partially annotated programs where the invariant needs to be strengthened to prove them correct (we explain these benchmarks later in this section). The programs in our suite have up to 100 lines of C code, involve up to 15 program variables, and often need complex invariants for their static verification.

Since the performance of the randomized search based invariant synthesis depends on the randomly chosen seed, we run it 10 times for every program and report the minimum and maximum time, the average of the times (when it doesn't timeout) and the number of runs where it times out ($\geq$ 10 min.). For the constraint solver based learner and our decision tree learners we provide details about the composition of the final sample, in terms of the number of positive, negative and implication counter-examples that were required to learn an adequate invariant and the number of rounds to converge.

Observations: The hypothesis that we want to test is whether the learning-based invariant synthesis tools developed in this paper are competitive with (not necessarily better than) state-of-the-art tools in verification.

CPAchecker is a mature tool (it won the software verification competition (SV-COMP) including the competition for the "loop" category in it). From Table 1 and Figure 2(a), we conclude that our learning algorithms are competitive to CPAchecker for synthesizing

[2] Programs available at `http://madhu.cs.illinois.edu/popl16`

[3] When a program is incorrect, we could find this bug when we find an inconsistent sample; however, we do not explore this aspect in this paper.

[4] We handle only those array programs in which the invariant itself is scalar, and not a quantified array invariant.

Table 1: Results comparing different invariant synthesis tools. $\chi_{\mathbf{TO}}$ indicates that the tool times out (> 10 minutes); χ indicates that the tool incorrectly concludes that the program is buggy; $\chi_{\mathbf{MO}}$ indicates that the tool runs out of memory; P, N, I are the number of positive, negative examples and implications in the final sample of the resp. learner; *#R* is the number of rounds, and T is the time in seconds.

Program	White-box	Black-box											
	CPAchecker [12] (s)	Randomized Search [49]			ICE-CS [25]			ICE-DT-entropy			ICE-DT-penalty		
		Min.(s)	Max.(s)	Avg.(s) + TO	P,N,I	#R	T(s)	P,N,I	#R	T(s)	P,N,I	#R	T(s)
SV-COMP programs and variants [2]													
array	2	0	123	18.5 + 3/10 TO	4,7,11	14	0.5	6,7,22	34	1.47	5,11,32	48	2.2
array2	2.4	0.1	384.5	105.7 + 4/10 TO	4,7,5	7	0.3	2,3,1	5	0.22	2,4,1	6	0.39
afnp	$\chi_{\mathbf{TO}}$	0.1	0.7	0.3 + 0/10 TO	1,19,15	29	3.6	1,3,7	11	0.48	1,2,7	10	0.47
cggmp	2	—	—	— + **10/10 TO**	1,36,50	71	51.1	1,18,45	64	3.48	1,17,42	60	3.01
countud	χ	—	—	— + **10/10 TO**	3,12,7	13	1	3,10,5	17	0.69	2,9,3	13	0.51
dtuc	$\chi_{\mathbf{TO}}$	4.9	190.4	62.8 + 2/10 TO	3,9,14	12	0.7	2,5,11	12	0.51	4,11,14	21	0.83
ex14	2.4	0	0.1	0.0 + 0/10 TO	2,5,1	7	0	1,1,0	2	0.12	1,1,0	2	0.11
ex14c	1.8	0.2	31.6	3.4 + 0/10 TO	2,2,1	4	0	2,2,0	3	0.12	2,2,0	3	0.14
ex23	5.4	0.1	127.5	21.8 + 1/10 TO	5,32,40	69	17.5	6,23,12	36	1.59	8,9,1	15	0.56
ex7	5.7	0	160.2	22.0 + 0/10 TO	1,2,1	2	0	1,1,0	2	0.12	1,1,0	2	0.09
matrixl1	3.3	—	—	— + **10/10 TO**	2,9,3	8	0.3	6,8,2	9	0.61	6,9,2	10	0.58
matrixl1c	3	—	—	— + **10/10 TO**	4,12,4	8	0.9	7,13,2	10	0.59	7,13,1	9	0.5
matrixl2	3.4	0.7	0.7	0.7 + 9/10 TO	8,19,13	27	22.9	8,11,8	23	1.25	9,11,6	22	1.06
matrixl2c	3.1	308	308	308.0 + 9/10 TO	$\chi_{\mathbf{TO}}$			15,26,10	44	2.61	20,35,22	66	3.95
nc11	2.1	0	0.1	0.1 + 0/10 TO	5,15,7	18	0.7	3,6,5	13	0.58	2,4,4	9	0.39
nc11c	2.1	0.1	46.1	6.3 + 2/10 TO	4,6,3	10	0.4	3,3,3	8	0.36	3,3,3	8	0.27
sum1	1.9	270.2	270.2	270.2 + 9/10 TO	2,15,10	17	2.3	3,11,2	14	0.58	3,11,2	14	0.56
sum3	2	0	0.1	0.1 + 0/10 TO	1,3,1	4	0.1	1,4,1	6	0.31	1,4,1	6	0.31
sum4	2.2	4.7	26.8	11.4 + 0/10 TO	1,23,31	44	3.5	1,9,41	51	2.42	1,8,41	50	2.46
sum4c	2	3.1	420.2	171.2 + 6/10 TO	6,29,21	34	11.6	4,14,7	22	1.05	4,13,4	18	0.86
tacas	1.8	0	0.1	0.0 + 0/10 TO	7,8,5	14	1.7	14,10,17	38	1.65	11,8,7	23	0.81
trex1	1.9	0	90.6	9.1 + 0/10 TO	2,3,0	3	0	2,3,0	5	0.19	2,3,0	5	0.19
trex3	χ	—	—	— + **10/10 TO**	6,19,6	19	2.7	3,7,4	12	0.55	2,6,3	10	0.42
vsend	1.8	0	0.1	0.0 + 0/10 TO	1,1,0	2	0	1,1,0	2	0.14	1,1,0	2	0.11
Other programs													
arrayinv1	3.8	—	—	— + **10/10 TO**	$\chi_{\mathbf{MO}}$			4,48,222	271	30.87	5,45,121	168	13.17
arrayinv2	4.5	0.1	56.4	16.7 + 0/10 TO	4,22,33	43	20.9	5,24,50	78	4.65	4,16,14	33	1.26
dec	15.4	0	0	0.0 + 0/10 TO	1,1,1	3	0	1,2,0	3	0.12	1,2,0	3	0.14
formula22	2	1.7	347.9	172.8 + 6/10 TO	1,18,11	22	1.8	1,16,32	49	2	1,7,20	28	1.09
formula25	2.3	9.1	163.5	56.6 + 2/10 TO	1,46,30	49	14	1,53,3	57	2.26	1,53,3	57	2
formula27	2.2	—	—	— + **10/10 TO**	$\chi_{\mathbf{MO}}$			1,183,18	202	9.56	1,119,11	131	5.55
inc	15.4	0	0	0.0 + 0/10 TO	3,12,101	112	1.7	3,1,102	106	4.31	3,1,100	104	3.92
inc2	1.8	0	8	0.8 + 0/10 TO	3,4,3	8	0.1	2,3,1	6	0.22	2,3,1	6	0.23
loops	$\chi_{\mathbf{TO}}$	96.1	284.1	159.2 + 4/10 TO	4,3,10	7	0.2	2,6,11	16	0.66	2,5,10	14	0.58
Programs from [3, 27, 28, 30]													
add	χ	—	—	— + **10/10 TO**	1,11,0	12	0.1	2,12,1	15	0.59	2,12,1	15	0.56
cegar1	1.9	0	0.1	0.1 + 0/10 TO	1,1,1	3	0	3,1,1	5	0.17	3,1,1	5	0.22
cegar2	2.2	1.2	305.6	82.1 + 3/10 TO	4,20,14	28	9.5	4,7,8	17	0.61	5,9,14	26	0.94
dillig01	1.9	4.8	56.4	16.7 + 0/10 TO	5,15,10	17	0.7	2,4,1	6	0.27	2,4,1	6	0.23
dillig03	$\chi_{\mathbf{TO}}$	0.4	6.3	4.0 + 4/10 TO	2,12,9	15	1	1,3,2	6	0.22	1,4,2	7	0.37
dillig05	$\chi_{\mathbf{TO}}$	6.4	172.3	87.5 + 4/10 TO	3,21,25	29	4.9	2,26,3	30	1.2	2,26,3	30	1.22
dillig07	1.9	0.2	16.6	4.1 + 0/10 TO	2,6,8	13	0.3	2,4,6	12	0.47	3,4,6	13	0.41
dillig12	$\chi_{\mathbf{TO}}$	—	—	— + **10/10 TO**	$\chi_{\mathbf{MO}}$			1,5,136	109	7.91	1,5,98	68	3.46
dillig15	1.9	—	—	— + **10/10 TO**	3,8,16	22	2.9	2,3,6	10	0.37	2,3,10	14	0.5
dillig17	$\chi_{\mathbf{TO}}$	—	—	— + **10/10 TO**	3,15,53	34	12.7	2,6,23	21	0.87	2,6,21	21	0.95
dillig19	2.3	62.7	455.7	269.0 + 0/10 TO	4,12,18	20	8.6	5,4,17	22	0.94	3,3,7	12	0.45
dillig24	1.9	—	—	— + **10/10 TO**	6,7,28	17	1.4	0,11,6	15	0.62	0,11,6	15	0.7
dillig25	2	—	—	— + **10/10 TO**	1,41,96	51	14.9	1,7,276	112	11.15	1,6,130	62	3.45
dillig28	$\chi_{\mathbf{TO}}$	115.3	228.5	193.6 + 2/10 TO	1,5,14	11	0.2	1,4,26	19	0.75	1,3,17	14	0.59
fig1	1.9	0.5	51.2	11.1 + 4/10 TO	2,5,1	6	0.1	2,4,1	6	0.22	2,4,1	6	0.22
fig3	1.9	0.3	5.2	2.7 + 8/10 TO	2,4,2	6	0.1	4,3,0	5	0.22	4,3,0	5	0.27
fig9	1.9	0	0.1	0.0 + 0/10 TO	0,2,0	2	0	1,1,0	2	0.12	1,1,0	2	0.09
w1	1.8	0	0.2	0.1 + 0/10 TO	1,3,3	5	0	2,1,1	4	0.22	2,1,1	4	0.16
w2	1.9	0.1	223.9	27.5 + 1/10 TO	2,4,1	4	0	1,1,1	3	0.14	1,1,1	3	0.12
Programs with invariants over non-linear integer arithmetic													
multiply	χ	—	—	— + **10/10 TO**	$\chi_{\mathbf{MO}}$			2,28,12	42	24.18	6,47,19	71	59.76
sqrt	χ	—	—	— + **10/10 TO**	3,26,26	32	9.2	3,15,14	31	1.42	4,28,14	43	1.97
square	χ	—	—	— + **10/10 TO**	$\chi_{\mathbf{MO}}$			1,8,2	11	0.41	1,8,2	11	0.42
Invariant synthesis in a deductive-verification setting													
array_diff	—	—			2,2,2	4	0.07	2,2,0	3	0.14	2,2,0	3	0.14
cpm1	—	—			—			2,4,0	5	2.78	2,4,0	5	2.81
cpm2	—	—			—			1,8,11	20	7.37	1,8,11	20	7.08
Aggregate	**41 / 58 programs**	**39 / 58 programs**			**50 / 58 programs**			**58 / 58 programs**			**58 / 58 programs**		

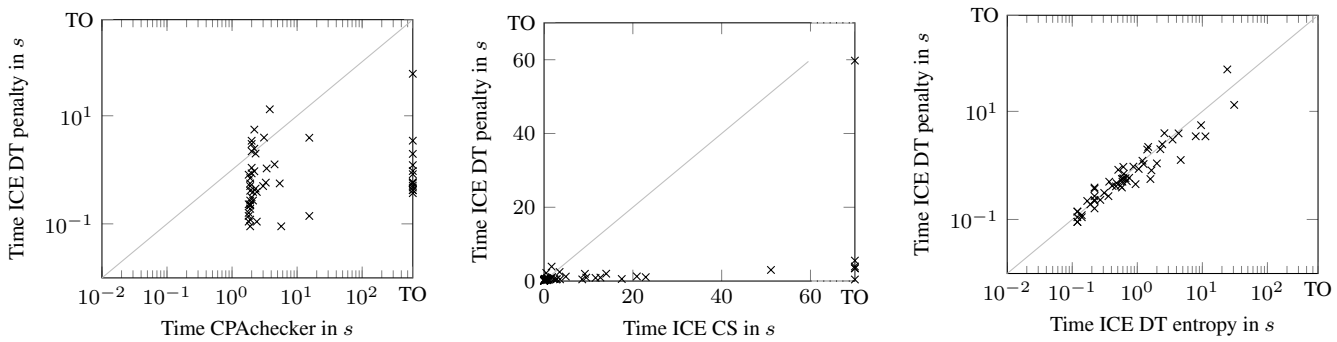


Figure 2: Runtime comparison between invariant synthesis tools. TO denotes timeout after 600 s

invariants for the benchmarks. Out of the 58 programs, CPAchecker synthesizes adequate invariants for 41 programs; however, recall that it is at a disadvantage here as since it does not know that the programs have octagonal constraints, while the black-box learners do. Moreover, when we run CPAchecker in its competition configuration, it was successful on 6 additional programs (but couldn't find invariants on the other 11).

To showcase the advantages of black-box learning, 3 out of these 58 programs, namely *multiply*, *sqrt*, and *square*, require invariants over non-linear integer arithmetic which CPAchecker is unable to synthesize. The black-box learning algorithms are completely agnostic of the semantics of the program as long as the teacher can generate counterexamples for the learner. Learning invariants for these programs was not a problem for our decision-tree based learners and also, for a couple of these programs, for the constraint-solver based learner. Note that for these three programs, we manually ask the learning algorithms for all black-box learners to learn an invariant over attributes which include octagonal attributes, as before, and, additionally, attributes over non-linear terms. Even though solving constraints over non-linear integer arithmetic is undecidable, our Boogie-based teacher (in ICE-DT) had no problems in providing counterexamples to refute inadequate invariant hypotheses.

Invariant synthesis using randomized search [49], in our experience, is very volatile, as shown by the large variation in run times (see column showing minimum and maximum times), performing fast as well as timing out on the same program. It times out on 16 programs on all runs. Further, there are around 22 programs for which randomized search fails more than half of the time. However, note that there are certain random walks where it finds the invariant very fast.

Our decision tree based learners are also faster than the constraint solver based learner ICE-CS, which times out or runs out of memory on 6 programs. More importantly, we think that ICE-CS will inherently hit a wall for larger programs. The number of formulas that we are learning from grows with the number of variables, the size of the invariant formula, and hence a learning based algorithm may have to collect a large set of samples, each of which refutes some of these formulae. The size of the constraints in ICE-CS grows unduly, and running a constraint solver suffers greatly when programs get slightly larger. We refer the reader to the microbenchmarks described below that show how poorly ICE-CS performs when compared to ICE-DT as the sample sizes increase. In fact, in the examples where it failed to find invariants, we found that the sample sizes grew to be hundreds and involved up to 5 variables with a template that would cause the constraints it generates to be fairly large, which caused it run out of memory.

The two decision tree learners we build, ICE-DT-Entropy and ICE-DT-Penalty are both equally efficient and effective in synthesizing invariants (Figure 2(c)). Note that since our teacher returns implications only if it cannot find positive/negative examples, all programs that report a non-zero number of implications in the final sample ($\sim 80\%$ of programs) *required* implication counterexamples to make progress.

Finding Invariants in Partially Annotated Programs Black-box learning of invariants is particularly advantageous in the context of deductive verification, where the programmer has manually partially annotated the program with the specification, pre/post-conditions for methods and complicated invariants that form the crux of the proof of the correctness of the program, and we wish to reduce the annotation burden of the programmer by automatically strengthening the specified invariants by inferring additional *simpler* invariants over scalar variables to prove the specification. Inferring such a simple scalar invariant is hard for static analysis tools or white-box invariant synthesis engines such as CPAchecker, as they cannot work with the complex quantified invariants already written. A "guess-and-check" approach that black-box invariant synthesis entails (including methods like Houdini [23]), on the other hand, can learn such simple invariants as long as the teacher can generate counterexamples for the learner (by even using sound but incomplete ways to prove validity of VCs, such as E-matching [21] or natural proofs [42, 43]), and incomplete ways for finding finite models that prove the VC to be invalid [46]). The table lists 3 examples where we tried our tool, and the tool was able to strengthen invariants. These include *array_diff* (a program that computes differences between successive values of a sorted array) and two verified modules of C programs that model a distributed key-value store protocol. The latter two had been deductively verified in VCC [17] with complex invariants, and we removed some of the scalar invariants from some methods, converted them to Boogie code and asked the invariant synthesizers to infer them.

Robustness to Small Changes: To see how the learners would perform if there were a few other variables which were not involved in the invariant, we generated variations for 18 programs so that they have (*three*) extra variables that are havoc-ed inside the loop. This increases the search-space of the invariant synthesis problem for all the black-box learners. ICE-CS times out on *two* of these additional programs (*sum1, matrix2c*); it finishes but takes more time for some programs. Randomized search fails (no successful run) on *eight* additional programs. However, our decision tree learners continued to perform equally well for programs with these extra variables. The learning algorithms underlying our approach are particularly good in weeding out irrelevant attributes, and we believe this to be the reason for their superior performance.

In practical examples such as GPUVerify [11], an invariant typically involves only a small set of variables (two or three), but

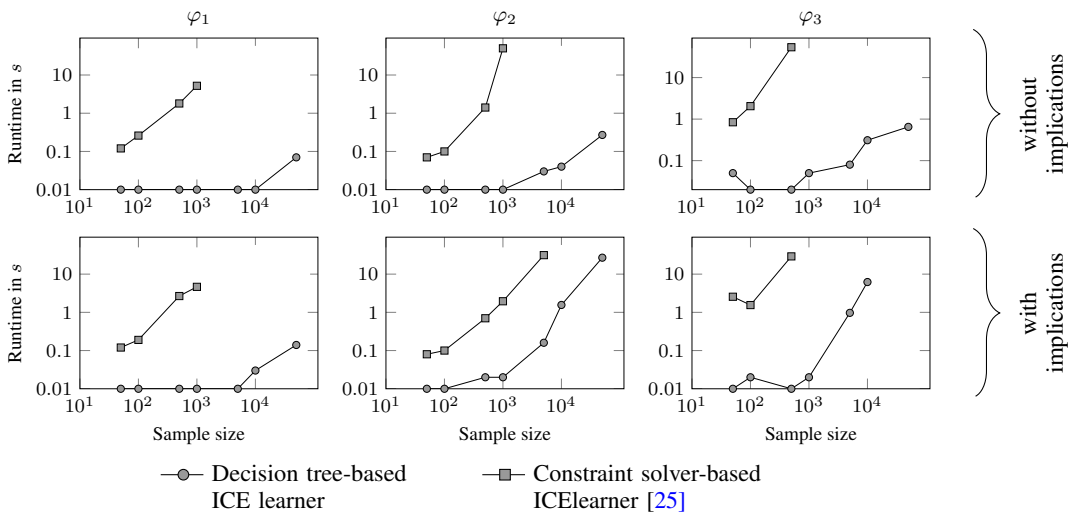


Figure 3: Results of the scalability micro-benchmarks

candidates range over a large set of variables (up to hundreds). We believe it is important for black-box invariant generation algorithms to handle such scenarios. We have also successfully applied our decision tree learner to learn invariants for selected large GPUVerify programs (for instance *prefixSum* and *binomialOption*), and though these are conjunctive invariants, our learning algorithm worked well. A more careful study and a full-blown invariant generation based on our techniques for race-checking GPU programs is ongoing research.

6.1 Scalability Micro-benchmarks:

We finally report our tool's performance on some micro-benchmarks demonstrating the scalability of the decision tree ICE learner compared to other learning techniques, namely the constraint solver-based ICE learner of [25]. This benchmark consists of samples of increasing size (containing between 50 and 50,000 data points) that have been randomly drawn and classified with respect to three formulas $\varphi_1 = x_1 \leq -1 \vee y \geq 1$, $\varphi_2 = x_1 - x_2 \geq 2$, and $\varphi_3 = 0 \leq x_0 \wedge 0 \leq x_1 \wedge 0 \leq x_2 \wedge x_3 \neq 1 \wedge x_4 \neq 1 \wedge x_5 \neq 1$) such that half of the data points are positive and half are negative. For each of the three formulas, the benchmark consist of a sample with no implications and a sample with $\frac{1}{4}$ of the data points unclassified and part of an implication.

The results of this benchmark are shown in Figure 3. The upper row shows the results on samples without implications, wheres the lower row shows results on samples with implications. Except for one sample suite, the decision-tree based learner can handle samples up to 50 000 data points, whereas the constraint-solver based learner times out or runs out of memory on samples with 1000 data points (with one exception). Note that the formulas ϕ_1, ϕ_2, ϕ_3 involve $1-6$ atomic formulas and range over $2-6$ variables; for invariant templates that have more atomic formulas to be determined and for programs with more variables, the memory/time-out limit for the constraint solver based learner might be reached even below samples with 1000 data-points (as seen in some of the invariant synthesis benchmarks). As Figure 3 shows, the decision tree-learner is on average one order of magnitude faster than ICE-CS, though it learns formulas that are roughly of the same "size" (not depicted in the figure). In summary, the micro-benchmark shows that the decision-tree based ICE learner scales much better than the constraint-solver based ICE learner, which motivates the use of machine-learning based tools for invariant synthesis. Note, however, that these are simply *scalability* micro-benchmarks to compare the learners, and do not measure their efficacy in actually learning inductive program invariants; efficicacy of learning invariants was discussed earlier.

7. Conclusions

We have presented a promising machine learning technique of learning decision trees from positive, negative, and implication counter-examples that can be used to efficiently synthesize invariants expressed as Boolean combinations of predicates over numerical and Boolean attributes. We have also adapted it so that it is provably convergent, assuring that it will learn an invariant when an expressible invariant exists. It would be interesting to see if we can use the learning algorithms in this paper to synthesize invariants over more complex logics, such as data-structure invariants expressed in separation logic. We believe that building custom invariant-generation tools for particular domains, using domain knowledge to identify interesting Boolean and numerical predicates using which invariants can be synthesized, would bring our techniques to bear on larger programs. GPUVerify [11], a race-checker for GPU programs does precisely this and uses Houdini to generate *conjunctive* invariants; our technique will extend invariant inference to arbitrary Boolean combinations and allow omitting thresholds in numerical attributes. Finding more domains where such black-box invariant synthesis will scale and be effective is the most promising direction for future work.

Acknowledgments

This work was partially supported by NSF Expeditions in Computing ExCAPE Award #1138994. Dan Roth is partly supported by DARPA under agreement number FA8750-13-2-0008.

References

[1] Learning bayesian network parameters under equivalence constraints. *Artificial Intelligence*, (0):–, 2015.

[2] Competition on Software Verification (SV-COMP) benchmarks. https://svn.sosy-lab.org/software/sv-benchmarks/tags/svcomp14/loops/.

[3] A. Albarghouthi and K. L. McMillan. Beautiful interpolants. In *Proceedings of the 25th International Conference on Computer Aided Verification*, CAV'13, pages 313–329, 2013. ISBN 978-3-642-39798-1.

[4] R. Alur, P. Cerný, P. Madhusudan, and W. Nam. Synthesis of interface specifications for java classes. In *POPL 2005*, pages 98–109. ACM, 2005.

[5] R. Alur, P. Madhusudan, and W. Nam. Symbolic compositional verification by learning assumptions. In *CAV 2005*, volume 3576 of *LNCS*, pages 548–562. Springer, 2005.

[6] R. Alur, R. Bodík, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *FMCAD 2013*, pages 1–17. IEEE, 2013.

[7] D. Angluin. Queries and concept learning. *Mach. Learn.*, 2(4):319–342, Apr. 1988. ISSN 0885-6125. . URL http://dx.doi.org/10.1023/A:1022821128753.

[8] D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

[9] G. BakIr, T. Hofmann, B. Scholkopf, A. J. Smola, B. Taskar, and S. Vishwanathan. *Predicting Structured Data*. MIT Press, Cambridge, MA, USA, 2007.

[10] M. Barnett, B. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO 2005*, volume 4111 of *LNCS*, pages 364–387. Springer, 2005.

[11] A. Betts, N. Chong, A. Donaldson, S. Qadeer, and P. Thomson. Gpuverify: A verifier for gpu kernels. *SIGPLAN Not.*, 47(10):113–132, Oct. 2012.

[12] D. Beyer and M. E. Keremoglu. Cpachecker: A tool for configurable software verification. In *CAV 2011*, volume 6806 of *LNCS*, pages 184–190. Springer, 2011.

[13] A. Blum and T. Mitchell. Combining labeled and unlabeled data with co-training. In *Proceedings of the Eleventh Annual Conference on Computational Learning Theory*, COLT' 98, pages 92–100, 1998. ISBN 1-58113-057-0.

[14] A. R. Bradley. Sat-based model checking without unrolling. In *VMCAI 2011*, volume 6538 of *LNCS*, pages 70–87. Springer, 2011.

[15] Y.-F. Chen and B.-Y. Wang. Learning boolean functions incrementally. In *CAV*, volume 7358 of *LNCS*, pages 55–70. Springer, 2012.

[16] J. M. Cobleigh, D. Giannakopoulou, and C. S. Pasareanu. Learning assumptions for compositional verification. In *TACAS 2003*, volume 2619 of *LNCS*, pages 331–346. Springer, 2003.

[17] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. Vcc: A practical system for verifying concurrent c. In *TPHOLs*, pages 23–42, 2009.

[18] M. Colón, S. Sankaranarayanan, and H. Sipma. Linear invariant generation using non-linear constraint solving. In *CAV 2003*, volume 2725 of *LNCS*, pages 420–432. Springer, 2003.

[19] C. Cortes and V. Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.

[20] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL 1977*, pages 238–252. ACM Press, 1977.

[21] L. M. de Moura and N. Bjørner. Efficient e-matching for smt solvers. In *CADE*, pages 183–198, 2007.

[22] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *ICSE 2000*, pages 449–458. ACM Press, 2000.

[23] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *FME*, volume 2021 of *LNCS*, pages 500–517. Springer, 2001.

[24] P. Garg, C. Löding, P. Madhusudan, and D. Neider. Learning universally quantified invariants of linear data structures. In *CAV 2013*, volume 8044 of *LNCS*, pages 813–829. Springer, 2013.

[25] P. Garg, C. Löding, P. Madhusudan, and D. Neider. ICE: A robust framework for learning invariants. In *CAV 2014*, volume 8559 of *LNCS*, pages 69–87. Springer, 2014.

[26] P. Garoche, T. Kahsai, and C. Tinelli. Incremental invariant generation using logic-based automatic abstract transformers. In *NFM 2013*, volume 7871 of *LNCS*, pages 139–154. Springer, 2013.

[27] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. Synergy: a new algorithm for property checking. In *SIGSOFT FSE*, pages 117–127. ACM, 2006.

[28] S. Gulwani, S. Srivastava, and R. Venkatesan. Program analysis as constraint solving. In *PLDI*, pages 281–292. ACM, 2008.

[29] A. Gupta and A. Rybalchenko. Invgen: An efficient invariant generator. In *CAV 2009*, volume 5643 of *LNCS*, pages 634–640. Springer, 2009.

[30] F. Ivancic and S. Sankaranarayanan. NECLA Benchmarks. http://www.nec-labs.com/research/system/systems_SAV-website/small_static_bench-v1.1.tar.gz.

[31] R. Jhala and K. L. McMillan. A practical and complete approach to predicate refinement. In *TACAS*, volume 3920 of *LNCS*, pages 459–473. Springer, 2006.

[32] M. Kawaguchi, P. M. Rondon, and R. Jhala. Type-based data structure verification. In *PLDI 2009*, pages 304–315. ACM, 2009.

[33] M. J. Kearns and U. V. Vazirani. *An introduction to computational learning theory*. MIT Press, Cambridge, MA, USA, 1994. ISBN 0-262-11193-4.

[34] S. Kong, Y. Jung, C. David, B.-Y. Wang, and K. Yi. Automatically inferring quantified loop invariants by algorithmic learning from simple templates. In *APLAS*. Springer, 2010.

[35] S. Krishna, C. Puhrsch, and T. Wies. Learning invariants using decision trees. *CoRR*, abs/1501.04725, 2015.

[36] N. Littlestone. Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm. *Machine Learning*, 2(4):285–318, 1987.

[37] K. L. McMillan. Lazy abstraction with interpolants. In *CAV 2006*, volume 4144 of *LNCS*, pages 123–136. Springer, 2006.

[38] K. L. McMillan. Interpolation and SAT-Based model checking. In *CAV*, volume 2725 of *LNCS*, pages 1–13. Springer, 2003.

[39] T. M. Mitchell. *Machine learning*. McGraw-Hill, 1997. ISBN 978-0-07-042807-2.

[40] D. Neider. *Applications of Automata Learning in Versification and Synthesis*. PhD thesis, RWTH Aachen University, April 2014.

[41] T. Nguyen, D. Kapur, W. Weimer, and S. Forrest. Using dynamic analysis to discover polynomial and array invariants. In *ICSE*, pages 683–693. IEEE, 2012.

[42] E. Pek, X. Qiu, and P. Madhusudan. Natural proofs for data structure manipulation in c using separation logic. In *PLDI*, page 46, 2014.

[43] X. Qiu, P. Garg, A. Stefanescu, and P. Madhusudan. Natural proofs for structure, data, and separation. In *PLDI*, pages 231–242, 2013.

[44] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993. ISBN 1-55860-238-0.

[45] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1): 81–106, 1986.

[46] A. Reynolds, C. Tinelli, A. Goel, S. Krstic, M. Deters, and C. Barrett. Quantifier instantiation techniques for finite model finding in SMT. In *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, pages 377–391, 2013. . URL http://dx.doi.org/10.1007/978-3-642-38574-2_26.

[47] F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6): 386–408, 1958.

[48] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27:379–423, 623–656, 1948.

[49] R. Sharma and A. Aiken. From invariant checking to invariant inference using randomized search. In *CAV 2014*, volume 8559 of *LNCS*, pages 88–105. Springer, 2014.

[50] R. Sharma, A. V. Nori, and A. Aiken. Interpolants as classifiers. In *CAV*, volume 7358 of *LNCS*, pages 71–87. Springer, 2012.

[51] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, P. Liang, and A. V. Nori. A data driven approach for algebraic loop invariants. In *ESOP*, volume 7792 of *LNCS*, pages 574–592. Springer, 2013.

[52] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, and A. V. Nori. Verification as learning geometric concepts. In *SAS*, volume 7935 of *LNCS*, pages 388–411. Springer, 2013.

[53] A. Solar Lezama. *Program Synthesis By Sketching*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2008.

[54] A. Thakur, A. Lal, J. Lim, and T. Reps. Posthat and all that: Attaining most-precise inductive invariants. Technical Report TR1790, University of Wisconsin, Madison, WI, Apr 2013.

[55] A. Vardhan and M. Viswanathan. Learning to verify branching time properties. *Formal Methods in System Design*, 31(1):35–61, 2007.