Efficient Synthesis with Probabilistic Constraints

Samuel Drews(✉), Aws Albarghouthi, and Loris D'Antoni

University of Wisconsin-Madison, Madison, USA
sedrews@wisc.edu

Abstract. We consider the problem of synthesizing a program given a probabilistic specification of its desired behavior. Specifically, we study the recent paradigm of *distribution-guided inductive synthesis* (DIGITS), which iteratively calls a synthesizer on finite sample sets from a given distribution. We make theoretical and algorithmic contributions: (*i*) We prove the surprising result that DIGITS only requires a polynomial number of synthesizer calls in the size of the sample set, despite its ostensibly exponential behavior. (*ii*) We present a property-directed version of DIGITS that further reduces the number of synthesizer calls, drastically improving synthesis performance on a range of benchmarks.

1 Introduction

Over the past few years, progress in automatic program synthesis has touched many application domains, including automating data wrangling and data extraction tasks [2,13,15,21,22,30], generating network configurations that meet user intents [10,29], optimizing low-level code [25,28], and more [4,14].

The majority of the current work has focused on synthesis under Boolean constraints. However, often times we require the program to adhere to a probabilistic specification, e.g., a controller that succeeds with a high probability, a decision-making model operating over a probabilistic population model, a randomized algorithm ensuring privacy, etc. In this work, we are interested in (1) investigating probabilistic synthesis from a theoretical perspective and (2) developing efficient algorithmic techniques to tackle this problem.

Our starting point is our recent framework for probabilistic synthesis called *distribution-guided inductive synthesis* (DIGITS) [1]. The DIGITS framework is analogous in nature to the *guess-and-check* loop popularized by counterexample-guided approaches to synthesis and verification (CEGIS and CEGAR). The key idea of the algorithm is reducing the probabilistic synthesis problem to a non-probabilistic one that can be solved using existing techniques, e.g., SAT solvers. This is performed using the following loop: (1) approximating the input probability distribution with a finite sample set; (2) synthesizing a program for various possible output assignments of the finite sample set; and (3) invoking a probabilistic verifier to check if one of the synthesized programs indeed adheres to the given specification.

© The Author(s) 2019
I. Dillig and S. Tasiran (Eds.): CAV 2019, LNCS 11561, pp. 278–296, 2019.
https://doi.org/10.1007/978-3-030-25540-4_15

DIGITS has been shown to theoretically converge to correct programs when they exist—thanks to learning-theory guarantees. The primary bottleneck of DIGITS is the number of expensive calls to the synthesizer, which is ostensibly exponential in the size of the sample set. Motivated by this observation, this paper makes theoretical, algorithmic, and practical contributions:

- On the theoretical side, we present a detailed analysis of DIGITS and prove that it only requires a polynomial number of invocations of the synthesizer, explaining that the strong empirical performance of the algorithm is not merely due to the heuristics presented in [1] (Sect. 3).
- On the algorithmic side, we develop an improved version of DIGITS that is property-directed, in that it only invokes the synthesizer on instances that have a chance of resulting in a correct program, without sacrificing convergence. We call the new approach τ-DIGITS (Sect. 4).
- On the practical side, we implement τ-DIGITS for sketch-based synthesis and demonstrate its ability to converge significantly faster than DIGITS. We apply our technique to a range of benchmarks, including illustrative examples that elucidate our theoretical analysis, probabilistic repair problems of unfair programs, and probabilistic synthesis of controllers (Sect. 5).

2 An Overview of DIGITS

In this section, we present the synthesis problem, the DIGITS [1] algorithm, and fundamental background on learning theory.

2.1 Probabilistic Synthesis Problem

Program Model. As discussed in [1], DIGITS searches through some (infinite) set of programs, but it requires that the set of programs has *finite VC dimension* (we restate this condition in Sect. 2.3). Here we describe one constructive way of obtaining such sets of programs with finite VC dimension: we will consider sets of programs defined as *program sketches* [27] in the simple grammar from [1], where a program is written in a loop-free language, and "holes" defining the sketch replace some constant terminals in expressions.[1] The syntax of the language is defined below:

$$P \coloneqq V \leftarrow E \mid \texttt{if}\ B\ \texttt{then}\ P\ \texttt{else}\ P \mid P\ P \mid \texttt{return}\ V$$

Here, P is a program, V is the set of variables appearing in P, E (resp. B) is the set of linear arithmetic (resp. Boolean) expressions over V (where, again, constants in E and B can be replaced with holes), and $V \leftarrow E$ is an assignment. We assume a vector $\boldsymbol{v}_I$ of variables in V that are inputs to the program. We

[1] In the case of loop-free program sketches as considered in our program model, we can convert the input-output relation into a real arithmetic formula that guaranteedly has finite VC dimension [12].

also assume there is a single Boolean variable $v_r \in V$ that is returned by the program.[2] All variables are real-valued or Boolean. Given a vector of constant values $\boldsymbol{c}$, where $|\boldsymbol{c}| = |\boldsymbol{v}_I|$, we use $P(\boldsymbol{c})$ to denote the result of executing P on the input $\boldsymbol{c}$.

In our setting, the inputs to a program are distributed according to some *joint probability distribution* $\mathbb{D}$ over the variables $\boldsymbol{v}_I$. Semantically, a program P is denoted by a *distribution transformer* $[\![P]\!]$, whose input is a distribution over values of $\boldsymbol{v}_I$ and whose output is a distribution over $\boldsymbol{v}_I$ and v_r.

A program also has a *probabilistic postcondition*, *post*, defined as an inequality over terms of the form $\Pr[B]$, where B is a Boolean expression over $\boldsymbol{v}_I$ and v_r. Specifically, a probabilistic postcondition consists of Boolean combinations of the form $e > c$, where $c \in \mathbb{R}$ and e is an arithmetic expression over terms of the form $\Pr[B]$, e.g., $\Pr[B_1]/\Pr[B_2] > 0.75$.

Given a triple $(P, \mathbb{D}, \mathit{post})$, we say that P is *correct* with respect to $\mathbb{D}$ and *post*, denoted $[\![P]\!](\mathbb{D}) \models \mathit{post}$, *iff* *post* is true on the distribution $[\![P]\!](\mathbb{D})$.

Example 1. Consider the set of intervals of the form $[0, a] \subseteq [0, 1]$ and inputs x uniformly distributed over $[0, 1]$ (i.e. $\mathbb{D} = \text{Uniform}[0, 1]$). We can write inclusion in the interval as a (C-style) program (left) and consider a postcondition stating that the interval must include at least half the input probability mass (right):

```
if(0 <= x && x <= a) {
    return 1;
}
return 0;
```

$$\Pr_{x \sim \mathbb{D}}[P(x) = 1] \geqslant 0.5$$

Let P_c denote the interval program where a is replaced by a constant $c \in [0, 1]$. Observe that $[\![P_c]\!](\mathbb{D})$ describes a joint distribution over (x, v_r) pairs, where $[0, c] \times \{1\}$ is assigned probability measure c and $(c, 1] \times \{0\}$ is assigned probability measure $1 - c$. Therefore, $[\![P_c]\!](\mathbb{D}) \models \mathit{post}$ if and only if $c \in [0.5, 1]$.

Synthesis Problem. DIGITS outputs a program that is approximately "similar" to a given functional specification and that meets a postcondition. This functional specification is some input-output relation which we quantitatively want to match as closely as possible: specifically, we want to minimize the *error* of the output program P from the functional specification $\hat{P}$, defined as $\text{Er}(P) \coloneqq \Pr_{x \sim \mathbb{D}}[P(x) \neq \hat{P}(x)]$. (Note that we represent the functional specification as a program.) The postcondition is Boolean, and therefore we always want it to be true. DIGITS is guaranteed to converge whenever the space of solutions satisfying the postcondition is *robust* under small perturbations. The following definition captures this notion of robustness:

Definition 1 (α-Robust Programs). *Fix an input distribution $\mathbb{D}$, a postcondition post, and a set of programs $\mathcal{P}$. For any $P \in \mathcal{P}$ and any $\alpha > 0$, denote the*

[2] Restricting the output to Boolean is required by the algorithm; other output types can be turned into Boolean by rewriting. See, e.g., thermostat example in Sect. 5.

open α-ball centered at P *as* $B_\alpha(P) = \{P' \in \mathcal{P} \mid \Pr_{x\sim\mathbb{D}}[P(x) \neq P'(x)] < \alpha\}$. *We say a program* P *is* α-robust *if* $\forall P' \in B_\alpha(P).\, [\![P']\!](\mathbb{D}) \models post$.

We can now state the synthesis problem solved by DIGITS:

Definition 2 (Synthesis Problem). *Given an input distribution* $\mathbb{D}$*, a set of programs* $\mathcal{P}$*, a postcondition post, a functional specification* $\hat{P} \in \mathcal{P}$*, and parameters* $\alpha > 0$ *and* $0 < \varepsilon \leqslant \alpha$*, the synthesis problem is to find a program* $P \in \mathcal{P}$ *such that* $[\![P]\!](\mathbb{D}) \models post$ *and such that any other* α*-robust* P' *has* $Er(P) \leqslant Er(P') + \varepsilon$.

2.2 A Naive DIGITS Algorithm

Algorithm 1 shows a simplified, naive version of DIGITS, which employs a *synthesize-then-verify* approach. The idea of DIGITS is to utilize non-probabilistic synthesis techniques to synthesize a set of programs, and then apply a probabilistic verification step to check if any of the synthesized programs is a solution.

Specifically, this "Naive DIGITS" begins by sampling an appropriate number of inputs from the input distribution and stores them in the set S. Second, it iteratively explores each possible function f that maps the input samples to a Boolean and invokes a synthesis oracle to synthesize a program P that implements f, i.e. that satisfies the set of input–output examples in which each input $x \in S$ is mapped to the output $f(x)$. Naive DIGITS then finds which of the synthesized programs satisfy the postcondition (the set *res*); we assume that we have access to a probabilistic verifier $\mathcal{O}_{\text{ver}}$ to perform these computations. Finally, the algorithm outputs the program in the set *res* that has the lowest error with respect to the functional specification, once again assuming access to another oracle $\mathcal{O}_{\text{err}}$ that can measure the error.

```
1 Procedure DIGITS (P̂, 𝔻, post, m)
2     S ← {x ∼ 𝔻 | i ∈ [1, ..., m]}
3     progs ← ∅
4     foreach f : S → {0, 1} do
5         P ← O_syn({(x, f(x)) | x ∈ S})
6         if P ≠ ⊥ then
7             progs ← progs ∪ {P}
8     res ← {P ∈ progs |
      O_ver(P, 𝔻, post)}
9     return argmin_{P∈res} {O_err(P)}
```

Algorithm 1: Naive DIGITS

Note that the number of such functions $f : S \to \{0, 1\}$ is exponential in the size of $|S|$. As a "heuristic" to improve performance, the actual DIGITS algorithm as presented in [1] employs an incremental trie-based search, which we describe (alongside our new algorithm, τ-DIGITS) and analyze in Sect. 3. The naive version described here is, however, sufficient to discuss the convergence properties of the full algorithm.

2.3 Convergence Guarantees

DIGITS is only guaranteed to converge when the program model $\mathcal{P}$ has *finite VC dimension*.[3] Intuitively, the VC dimension captures the expressiveness of the set

[3] Recall that this is largely a "free" assumption since, again, sketches in our loop-free grammar guaranteedly have finite VC dimension.

of ($\{0,1\}$-valued) programs $\mathcal{P}$. Given a set of inputs S, we say that $\mathcal{P}$ *shatters* S iff, for every partition of S into sets $S_0 \sqcup S_1$, there exists a program $P \in \mathcal{P}$ such that (i) for every $x \in S_0$, $P(x) = 0$, and (ii) for every $x \in S_1$, $P(x) = 1$.

Definition 3 (VC Dimension). *The* VC dimension *of a set of programs $\mathcal{P}$ is the largest integer d such that there exists a set of inputs S with cardinality d that is shattered by $\mathcal{P}$.*

We define the function $\text{VCCOST}(\varepsilon, \delta, d) = \frac{1}{\varepsilon}(4\log_2(\frac{2}{\delta}) + 8d\log_2(\frac{13}{\varepsilon}))$ [5], which is used in the following theorem:

Theorem 1 (Convergence). *Assume that there exist an $\alpha > 0$ and program P^* that is α-robust w.r.t. $\mathbb{D}$ and post. Let d be the VC dimension of the set of programs $\mathcal{P}$. For all bounds $0 < \varepsilon \leqslant \alpha$ and $\delta > 0$, for every function $\mathcal{O}_{\text{syn}}$, and for any $m \geqslant \text{VCCOST}(\varepsilon, \delta, k)$, with probability $\geqslant 1 - \delta$ we have that* DIGITS *enumerates a program P with $\Pr_{x\sim\mathbb{D}}[P^*(x) \neq P(x)] \leqslant \varepsilon$ and $[\![P]\!](\mathbb{D}) \models post$.*

To reiterate, suppose P^* is a correct program with small error $\text{Er}(P^*) = k$; the convergence result follows two main points: (i) P^* must be α-*robust*, meaning every P with $\Pr_{x\sim\mathbb{D}}[P(x) \neq P^*(x)] < \alpha$ must also be correct, and therefore (ii) by synthesizing *any* P such that $\Pr_{x\sim\mathbb{D}}[P(x) \neq P^*(x)] \leqslant \varepsilon$ where $\varepsilon < \alpha$, then P is a correct program with error $\text{Er}(P)$ within $k \pm \varepsilon$.

2.4 Understanding Convergence

The importance of finite VC dimension is due to the fact that the convergence statement borrows directly from *probably approximately correct (PAC) learning*. We will briefly discuss a core detail of efficient PAC learning that is relevant to understanding the convergence of DIGITS (and, in turn, our analysis of τ-DIGITS in Sect. 4), and refer the interested reader to Kearns and Vazirani's book [16] for a complete overview. Specifically, we consider the notion of an ε-*net*, which establishes the approximate-definability of a target program in terms of points in its input space.

Definition 4 (ε-net). *Suppose $P \in \mathcal{P}$ is a target program, and points in its input domain $\mathcal{X}$ are distributed $x \sim \mathbb{D}$. For a fixed $\varepsilon \in [0,1]$, we say a set of points $S \subset \mathcal{X}$ is an* ε-net *for P (with respect to $\mathcal{P}$ and $\mathbb{D}$) if for every $P' \in \mathcal{P}$ with $\Pr_{x\sim\mathbb{D}}[P(x) \neq P'(x)] > \varepsilon$ there exists a witness $x \in S$ such that $P(x) \neq P'(x)$.*

In other words, if S is an ε-net for P, and if P' "agrees" with P on all of S, then P and P' can only differ by at most ε probability mass.

Observe the relevance of ε-nets to the convergence of DIGITS: the synthesis oracle is guaranteed not to "fail" by producing only programs ε-far from some ε-robust P^* if the sample set happens to be an ε-net for P^*. In fact, this observation is exactly the core of the PAC learning argument: having an ε-net exactly guarantees the approximate learnability.

A remarkable result of computational learning theory is that whenever $\mathcal{P}$ has finite VC dimension, the probability that m random samples fail to yield an ε-net

becomes diminishingly small as m increases. Indeed, the given VCCOST function used in Theorem 1 is a dual form of this latter result—that polynomially many samples are sufficient to form an ε-net with high probability.

3 The Efficiency of Trie-Based Search

After providing details on the search strategy employed by DIGITS, we present our theoretical result on the polynomial bound on the number of synthesis queries that DIGITS requires.

3.1 The Trie-Based Search Strategy of DIGITS

Naive DIGITS, as presented in Algorithm 1, performs a very unstructured, exponential search over the output labelings of the sampled inputs—i.e., the possible Boolean functions f in Algorithm 1. In our original paper [1] we present a "heuristic" implementation strategy that incrementally explores the set of possible output labelings using a trie data structure. In this section, we study the complexity of this technique through the lens of computational learning theory and discover the surprising result that DIGITS requires a polynomial number of calls to the synthesizer in the size of the sample set! Our improved search algorithm (Sect. 4) inherits these results.

For the remainder of this paper, we use DIGITS to refer to this incremental version. A full description is necessary for our analysis: Fig. 1 (non-framed rules only) consists of a collection of guarded rules describing the construction of the trie used by DIGITS to incrementally explore the set of possible output labelings. Our improved version, τ-DIGITS (presented in Sect. 4), corresponds to the addition of the framed parts, but without them, the rules describe DIGITS.

Nodes in the trie represent partial output labelings—i.e., functions f assigning Boolean values to only some of the samples in $S = \{x_1, \ldots, x_m\}$. Each node is identified by a binary string $\sigma = b_1 \cdots b_k$ (k can be smaller than m) denoting the path to the node from the root. The string σ also describes the partial output-labeling function f corresponding to the node—i.e., if the i-th bit b_i is set to 1, then $f(x_i) = \mathit{true}$. The set *explored* represents the nodes in the trie built thus far; for each new node, the algorithm synthesizes a program consistent with the corresponding partial output function ("Explore" rules). The variable *depth* controls the incremental aspect of the search and represents the maximum length of any σ in *explored*; it is incremented whenever all nodes up to that depth have been explored (the "Deepen" rule). The crucial part of the algorithm is that, if no program can be synthesized for the partial output function of a node identified by σ, the algorithm does not need to issue further synthesis queries for the descendants of σ.

Figure 2 shows how DIGITS builds a trie for an example run on the interval programs from Example 1, where we suppose we begin with an incorrect program describing the interval $[0, 0.3]$. Initially, we set the root program to $[0, 0.3]$ (left

$$\frac{}{explored \leftarrow \{\epsilon\} \quad P_\epsilon \leftarrow \hat{P} \quad depth \leftarrow 0 \quad best \leftarrow \bot} \text{ Initialize}$$

$$\frac{\forall \sigma \in explored. \forall b \in \{0,1\}. \; (P_\sigma \neq \bot \wedge |\sigma b| \leqslant depth \boxed{\wedge unblocked(\sigma b)}) \Rightarrow \sigma b \in explored}{sample_{depth+1} \sim \mathbb{D} \quad depth \leftarrow depth + 1} \text{ Deepen}$$

$$\frac{\sigma \in explored \quad P_\sigma \neq \bot \quad b \in \{0,1\} \quad \sigma b \notin explored \quad |\sigma b| \leqslant depth \quad \boxed{unblocked(\sigma b)}}{P_{\sigma b} \leftarrow \mathcal{O}_{\text{syn}}(\{(sample_{i+1}, \sigma b(i)) : 0 \leqslant i < |\sigma b|\}) \quad explored \leftarrow explored \cup \{\sigma b\}} \text{ Explore (Synthesis Query)}$$

$$\frac{\sigma \in explored \quad P_\sigma \neq \bot \quad b \in \{0,1\} \quad \sigma b \notin explored \quad |\sigma b| \leqslant depth \quad \boxed{unblocked(\sigma b)} \quad P_\sigma(sample_{|\sigma b|}) = b}{P_{\sigma b} \leftarrow P_\sigma \quad explored \leftarrow explored \cup \{\sigma b\}} \text{ Explore (Solution Propagation)}$$

$$\frac{\sigma^* = \text{argmin}_\sigma\{\mathcal{O}_{\text{err}}(P_\sigma) \mid \sigma \in explored \wedge P_\sigma \neq \bot \wedge \mathcal{O}_{\text{ver}}(P_\sigma) = \text{true}\}}{best \leftarrow P_{\sigma^*}} \text{ Best}$$

$$\boxed{\text{where } unblocked(\sigma) \coloneqq |\{i : 0 \leqslant i < |\sigma| \wedge \sigma(i) \neq \hat{P}(sample_{i+1})\}| \leqslant \tau \cdot depth}$$

Fig. 1. Full DIGITS description and our new extension, τ-DIGITS, shown in boxes.

figure). The "Deepen" rule applies, so a sample is added to the set of samples—suppose it's 0.4. "Explore" rules are then applied twice to build the children of the root: the child following the 0 branch needs to map $0.4 \mapsto 0$, which $[0, 0.3]$ already does, thus it is propagated to that child without asking $\mathcal{O}_{\text{syn}}$ to perform a synthesis query. For the child following 1, we instead make a synthesis query, using the oracle $\mathcal{O}_{\text{syn}}$, for any value of a such that $[0, a]$ maps $0.4 \mapsto 1$—suppose it returns the solution $a = 1$, and we associate $[0, 1]$ with this node. At this point we have exhausted depth 1 (middle figure), so "Deepen" once again applies, perhaps adding 0.6 to the sample set. At this depth (right figure), only two calls to $\mathcal{O}_{\text{syn}}$ are made: in the case of the call at $\sigma = 01$, there is no value of a that causes both $0.4 \mapsto 0$ and $0.6 \mapsto 1$, so $\mathcal{O}_{\text{syn}}$ returns $\bot$, and we do not try to explore any children of this node in the future. The algorithm continues in this manner until a stopping condition is reached—e.g., enough samples are enumerated.

3.2 Polynomial Bound on the Number of Synthesis Queries

We observed in [1] that the trie-based exploration seems to be efficient in practice, despite potential exponential growth of the number of explored nodes in the trie as the depth of the search increases. The convergence analysis of DIGITS relies on the finite VC dimension of the program model, but VC dimension itself is just a summary of the *growth function*, a function that describes a notion

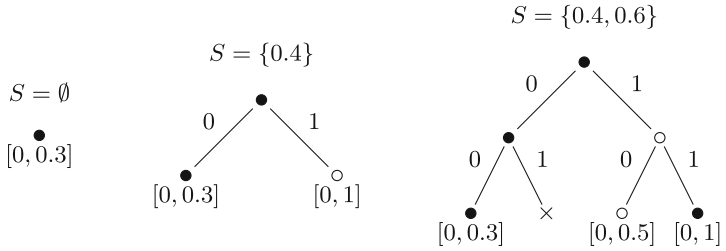

Fig. 2. Example execution of incremental DIGITS on interval programs, starting from $[0, 0.3]$. Hollow circles denote calls to $\mathcal{O}_{\text{syn}}$ that yield new programs; the cross denotes a call to $\mathcal{O}_{\text{syn}}$ that returns $\bot$.

of complexity of the set of programs in question. We will see that the growth function much more precisely describes the behavior of the trie-based search; we will then use a classic result from computational learning theory to derive better bounds on the performance of the search. We define the growth function below, adapting the presentation from [16].

Definition 5 (Realizable Dichotomies). *We are given a set $\mathcal{P}$ of programs representing functions from $\mathcal{X} \to \{0, 1\}$ and a (finite) set of inputs $S \subset \mathcal{X}$. We call any $f : S \to \{0, 1\}$ a* dichotomy *of S; if there exists a program $P \in \mathcal{P}$ that extends f to its full domain $\mathcal{X}$, we call f a* realizable dichotomy *in $\mathcal{P}$. We denote the set of realizable dichotomies as*

$$\Pi_{\mathcal{P}}(S) \coloneqq \{f : S \to \{0, 1\} \mid \exists P \in \mathcal{P}. \forall x \in S. P(x) = f(x)\}.$$

Observe that for any (infinite) set $\mathcal{P}$ and any finite set S that $1 \leqslant |\Pi_{\mathcal{P}}(S)| \leqslant 2^{|S|}$. We define the growth function in terms of the realizable dichotomies:

Definition 6 (Growth Function). *The* growth function *is the maximal number of realizable dichotomies as a function of the number of samples, denoted*

$$\hat{\Pi}_{\mathcal{P}}(m) \coloneqq \max_{\substack{S \subset \mathcal{X}: \\ |S| = m}} \{|\Pi_{\mathcal{P}}(S)|\}.$$

Observe that $\mathcal{P}$ has VC dimension d if and only if d is the largest integer satisfying $\hat{\Pi}_{\mathcal{P}}(d) = 2^d$ (and infinite VC dimension when $\hat{\Pi}_{\mathcal{P}}(m)$ is identically 2^m)— in fact, VC dimension is often defined using this characterization.

Example 2. Consider the set of intervals of the form $[0, a]$ as in Examples 1 and Fig. 2. For the set of two points $S = \{0.4, 0.6\}$, we have that $|\Pi_{[0,a]}(S)| = 3$, since, by example: $a = 0.5$ accepts 0.4 but not 0.6, $a = 0.3$ accepts neither, and $a = 1$ accepts both, thus these three dichotomies are realizable; however, no interval with 0 as a left endpoint can accept 0.6 and not 0.4, thus this dichotomy is not realizable. In fact, for any (finite) set $S \subset [0, 1]$, we have that $|\Pi_{[0,a]}(S)| = |S|+1$; we then have that $\hat{\Pi}_{[0,a]}(m) = m + 1$.

When DIGITS terminates having used a sample set S, it has considered all the dichotomies of S: the programs it has enumerated exactly correspond to extensions of the realizable dichotomies $\Pi_{\mathcal{P}}(S)$. The trie-based exploration is effectively trying to minimize the number of $\mathcal{O}_{\text{syn}}$ queries performed on non-realizable ones, but doing so without explicit knowledge of the full functional behavior of programs in $\mathcal{P}$. In fact, it manages to stay relatively close to performing queries only on the realizable dichotomies:

Lemma 1. DIGITS *performs at most* $|S||\Pi_{\mathcal{P}}(S)|$ *synthesis oracle queries. More precisely, let* $S = \{x_1, \ldots, x_m\}$ *be indexed by the depth at which each sample was added: the exact number of synthesis queries is* $\sum_{\ell=1}^{m} |\Pi_{\mathcal{P}}(\{x_1, \ldots, x_{\ell-1}\})|$.

Proof. Let T_d denote the total number of queries performed once depth d is completed. We perform no queries for the root,[4] thus $T_0 = 0$. Upon completing depth $d-1$, the realizable dichotomies of $\{x_1, \ldots, x_{d-1}\}$ exactly specify the nodes whose children will be explored at depth d. For each such node, one child is skipped due to solution propagation, while an oracle query is performed on the other, thus $T_d = T_{d-1} + |\Pi_{\mathcal{P}}(\{x_1, \ldots, x_{d-1}\})|$. Lastly, $|\Pi_{\mathcal{P}}(S)|$ cannot decrease by adding elements to S, so we have that $T_m = \sum_{\ell=1}^{m} |\Pi_{\mathcal{P}}(\{x_1, \ldots, x_{\ell-1}\})| \leqslant \sum_{\ell=1}^{m} |\Pi_{\mathcal{P}}(S)| \leqslant |S||\Pi_{\mathcal{P}}(S)|$. □

Connecting DIGITS to the realizable dichotomies and, in turn, the growth function allows us to employ a remarkable result from computational learning theory, stating that the growth function for any set exhibits one of two asymptotic behaviors: it is either *identically* 2^m (infinite VC dimension) or dominated by a polynomial! This is commonly called the Sauer-Shelah Lemma [24,26]:

Lemma 2 (Sauer-Shelah). *If* $\mathcal{P}$ *has finite VC dimension* d*, then for all* $m \geqslant d$, $\hat{\Pi}_{\mathcal{P}}(m) \leqslant \left(\frac{em}{d}\right)^d$*; i.e.* $\hat{\Pi}_{\mathcal{P}}(m) = O(m^d)$.

Combining our lemma with this famous one yields a surprising result—that for a fixed set of programs $\mathcal{P}$ with finite VC dimension, the number of oracle queries performed by DIGITS *is guaranteedly polynomial* in the depth of the search, where the degree of the polynomial is determined by the VC dimension:

Theorem 2. *If* $\mathcal{P}$ *has VC dimension* d*, then* DIGITS *performs* $O(m^{d+1})$ *synthesis-oracle queries.*

In short, the reason an execution of DIGITS *seems* to enumerate a sub-exponential number of programs (as a function of the depth of the search) is because it literally must be polynomial. Furthermore, the algorithm performs oracle queries on *nearly* only those polynomially-many realizable dichotomies.

Example 3. A DIGITS run on the $[0, a]$ programs as in Fig. 2 using a sample set of size m will perform $O(m^2)$ oracle queries, since the VC dimension of these intervals is 1. (In fact, every run of the algorithm on these programs will perform exactly $\frac{1}{2}m(m+1)$ many queries.)

[4] We assume the functional specification itself is some $\hat{P} \in \mathcal{P}$ and thus can be used—the alternative is a trivial synthesis query on an empty set of constraints.

4 Property-Directed τ-DIGITS

DIGITS has better convergence guarantees when it operates on larger sets of sampled inputs. In this section, we describe a new optimization of DIGITS that reduces the number of synthesis queries performed by the algorithm so that it more quickly reaches higher depths in the trie, and thus allows to scale to larger samples sets. This optimized DIGITS, called τ-DIGITS, is shown in Fig. 1 as the set of all the rules of DIGITS plus the framed elements. The high-level idea is to skip synthesis queries that are (quantifiably) unlikely to result in optimal solutions. For example, if the functional specification $\hat{P}$ maps every sampled input in S to 0, then the synthesis query on the mapping of every element of S to 1 becomes increasingly likely to result in programs that have maximal distance from $\hat{P}$ as the size of S increases; hence the algorithm could probably avoid performing that query.In the following, we make use of the concept of *Hamming distance* between pairs of programs:

Definition 7 (Hamming Distance). *For any finite set of inputs S and any two programs P_1, P_2, we denote* $\text{Hamming}_S(P_1, P_2) \coloneqq |\{x \in S \mid P_1(x) \neq P_2(x)\}|$ *(we will also allow any $\{0,1\}$-valued string to be an argument of* Hamming_S*).*

4.1 Algorithm Description

Fix the given functional specification $\hat{P}$ and suppose that there exists an ε-robust solution P^* with (nearly) minimal error $k = \text{Er}(P^*) \coloneqq \Pr_{x\sim\mathbb{D}}[\hat{P}(x) \neq P^*(x)]$; we would be happy to find *any* program P in P^*'s ε-ball. Suppose we angelically know k a priori, and we thus restrict our search (for each depth m) only to constraint strings (i.e. σ in Fig. 1) that have Hamming distance not much larger than km.

To be specific, we first fix some threshold $\tau \in (k, 1]$. Intuitively, the optimization corresponds to modifying DIGITS to consider only paths σ through the trie such that $\text{Hamming}_S(\hat{P}, \sigma) \leqslant \tau|S|$. This is performed using the *unblocked* function in Fig. 1. Since we are ignoring certain paths through the trie, we need to ask: *How much does this decrease the probability of the algorithm succeeding?*— It depends on the tightness of the threshold, which we address in Sect. 4.2. In Sect. 4.3, we discuss how to adaptively modify the threshold τ as τ-DIGITS is executing, which is useful when a good τ is unknown a priori.

4.2 Analyzing Failure Probability with Thresholding

Using τ-DIGITS, the choice of τ will affect both (i) how many synthesis queries are performed, and (ii) the likelihood that we *miss* optimal solutions; in this section we explore the latter point.[5] Interestingly, we will see that all of the analysis is dependent only on parameters directly related to the threshold; notably, none of this analysis is dependent on the complexity of $\mathcal{P}$ (i.e. its VC dimension).

[5] The former point is a difficult combinatorial question that to our knowledge has no precedent in the computational learning literature, and so we leave it as future work.

If we really want to learn (something close to) a program P^*, then we should use a value of the threshold τ such that $\Pr_{S\sim\mathbb{D}^m}[\text{Hamming}_S(\hat{P}, P^*) \leqslant \tau m]$ is large—to do so requires knowledge of the distribution of $\text{Hamming}_S(\hat{P}, P^*)$. Recall the *binomial distribution*: for parameters (n, p), it describes the number of successes in n-many trials of an experiment that has success probability p.

Claim. Fix P and let $k = \Pr_{x\sim\mathbb{D}}[\hat{P}(x) \neq P(x)]$. If S is sampled from $\mathbb{D}^m$, then $\text{Hamming}_S(\hat{P}, P)$ is binomially distributed with parameters (m, k).

Next, we will use our knowledge of this distribution to reason about the *failure probability*, i.e. that τ-DIGITS does not preserve the convergence result of DIGITS.

The simplest argument we can make is a union-bound style argument: the thresholded algorithm can "fail" by (i) failing to sample an ε-net, or otherwise (ii) sampling a set on which the optimal solution has a Hamming distance that is not representative of its actual distance. We provide the quantification of this failure probability in the following theorem:

Theorem 3. *Let P^* be a target ε-robust program with $k = \Pr_{x\sim\mathbb{D}}[\hat{P}(x) \neq P^*(x)]$, and let δ be the probability that m samples do not form an ε-net for P^*. If we run the τ-DIGITS with $\tau \in (k, 1]$, then the failure probability is at most $\delta + \Pr[X > \tau m]$ where $X \sim \text{Binomial}(m, k)$.*

In other words, we can use tail probabilities of the binomial distribution to bound the probability that the threshold causes us to "miss" a desirable program we otherwise would have enumerated. Explicitly, we have the following corollary:

Corollary 1. *τ-DIGITS increases failure probability (relative to DIGITS) by at most $\Pr[X > \tau m] = \sum_{i=\lfloor \tau m \rfloor + 1}^{m} \binom{m}{i} k^i (1-k)^{m-i}$.*

Informally, when m is *not too small*, k is *not too large*, and τ is *reasonably forgiving*, these tail probabilities can be quite small. We can even analyze the asymptotic behavior by using any existing upper bounds on the binomial distribution's tail probabilities—importantly, the additional error diminishes exponentially as m increases, dependent on the size of τ relative to k.

Corollary 2. *τ-DIGITS increases failure probability by at most $e^{-2m(\tau-k)^2}$.*[6]

Example 4. Suppose $m = 100$, $k = 0.1$, and $\tau = 0.2$. Then the extra failure probability term in Theorem 3 is less than 0.001.

As stated at the beginning of this subsection, the balancing act is to choose τ (i) small enough so that the algorithm is still fast for large m, yet (ii) large enough so that the algorithm is still likely to learn the desired programs. The further challenge is to relax our initial strong assumption that we know the optimal k a priori when determining τ, which we address in the following subsection.

[6] A more precise (though less convenient) bound is $e^{-m(\tau \ln \frac{\tau}{k} + (1-\tau) \ln \frac{1-\tau}{1-k})}$.

4.3 Adaptive Threshold

Of course, we do not have the angelic knowledge that lets us pick an ideal threshold τ; the only absolutely sound choice we can make is the trivial $\tau = 1$. Fortunately, we can begin with this choice of τ and *adaptively* refine it as the search progresses. Specifically, every time we encounter a correct program P such that $k = \mathrm{Er}(P)$, we can refine τ to reflect our newfound knowledge that "the best solution has distance of at most k."

We refer to this refinement as *adaptive* τ-DIGITS. The modification involves the addition of the following rule to Fig. 1:

$$\frac{\mathit{best} \neq \bot}{\tau \leftarrow g(\mathcal{O}_{\mathrm{err}}(\mathit{best}))}\ \text{Refine Threshold (for some } g : [0,1] \to [0,1])$$

We can use any (non-decreasing) function g to update the threshold $\tau \leftarrow g(k)$. The simplest choice would be the identity function (which we use in our experiments), although one could use a looser function so as not to over-prune the search. If we choose functions of the form $g(k) = k + b$, then Corollary 2 allows us to make (slightly weak) claims of the following form:

Claim. Suppose the adaptive algorithm completes a search of up to depth m yielding a best solution with error k (so we have the final threshold value $\tau = k + b$). Suppose also that P^* is an optimal ε-robust program at distance $k - \eta$. The optimization-added failure probability (as in Corollary 1) for a run of (non-adaptive) τ-DIGITS completing depth m and using this τ is at most $e^{-2m(b+\eta)^2}$.

5 Evaluation

Implementation. In this section, we evaluate our new algorithm τ-DIGITS (Fig. 1) and its adaptive variant (Sect. 4.3) against DIGITS (i.e., τ-DIGITS with $\tau = 1$). Both algorithms are implemented in Python and use the SMT solver Z3 [8] to implement a sketch-based synthesizer $\mathcal{O}_{\mathrm{syn}}$. We employ statistical verification for $\mathcal{O}_{\mathrm{ver}}$ and $\mathcal{O}_{\mathrm{err}}$: we use Hoeffding's inequality for estimating probabilities in *post* and Er. Probabilities are computed with 95% confidence, leaving our oracles potentially unsound.

Research Questions. Our evaluation aims to answer the following questions:

RQ1 Is adaptive τ-DIGITS more effective/precise than τ-DIGITS?
RQ2 Is τ-DIGITS more effective/precise than DIGITS?
RQ3 Can τ-DIGITS solve challenging synthesis problems?

We experiment on three sets of benchmarks: (*i*) synthetic examples for which the optimal solutions can be computed analytically (Sect. 5.1), (*ii*) the set of benchmarks considered in the original DIGITS paper (Sect. 5.2), (*iii*) a variant of the thermostat-controller synthesis problem presented in [7] (Sect. 5.3).

5.1 Synthetic Benchmarks

We consider a class of synthetic programs for which we can compute the optimal solution exactly; this lets us compare the results of our implementation to an ideal baseline. Here, the program model $\mathcal{P}$ is defined as the set of axis-aligned hyperrectangles within $[-1,1]^d$ ($d \in \{1,2,3\}$ and the VC dimension is $2d$), and the input distribution $\mathbb{D}$ is such that inputs are distributed uniformly over $[-1,1]^d$. We fix some probability mass $b \in \{0.05, 0.1, 0.2\}$ and define the benchmarks so that the best error for a correct solution is exactly b (for details, see [9]).

We run our implementation using thresholds $\tau \in \{0.07, 0.15, 0.3, 0.5, 1\}$, omitting those values for which $\tau < b$; additionally, we also consider an adaptive run where τ is initialized as the value 1, and whenever a new best solution is enumerated with error k, we update $\tau \leftarrow k$. Each combination of parameters was run for a period of 2 min. Figure 3 fixates on $d = 1$, $b = 0.1$ and shows each of the following as a function of time: (*i*) the depth completed by the search (i.e. the current size of the sample set), and (*ii*) the best solution found by the search. (See our full version of the paper [9] for other configurations of (d, b).)

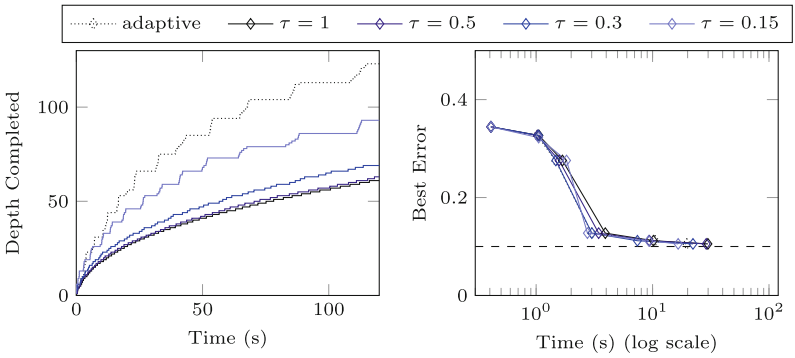

Fig. 3. Synthetic hyperrectangle problem instance with parameters $d = 1$, $b = 0.1$.

By studying Fig. 3 we see that the adaptive threshold search performs at least as well as the tight thresholds fixed a priori because reasonable solutions are found early. In fact, all search configurations find solutions very close to the optimal error (indicated by the horizontal dashed line). Regardless, they reach different depths, and *the main advantage of reaching large depths concerns the strength of the optimality guarantee.* Note, also, that small τ values are necessary to see improvements in the completed depth of the search. Indeed, the discrepancy between the depth-versus-time functions diminishes drastically for the problem instances with larger values of b (See our full version of the paper [9]); the gains of the optimization are contingent on the existence of correct solutions *close* to the functional specification.

Findings (RQ1): τ-DIGITS *does* tend to find *reasonable* solutions at early depths and near-optimal solutions at later depths, thus adaptive τ-DIGITS is more effective than τ-DIGITS, and we use it throughout our remaining experiments.

5.2 Original DIGITS Benchmarks

The original DIGITS paper [1] evaluates on a set of 18 repair problems of varying complexity. The functional specifications are machine-learned decision trees and support vector machines, and each search space $\mathcal{P}$ involves the set of programs formed by replacing some number of real-valued constants in the program with holes. The postcondition is a form of *algorithmic fairness*—e.g., the program should output true on inputs of type A as often as it does on inputs of type B [11]. For each such repair problem, we run both DIGITS and adaptive τ-DIGITS (again, with initial $\tau = 1$ and the identity refinement function). Each benchmark is run for 10 min, where the same sample set is used for both algorithms.

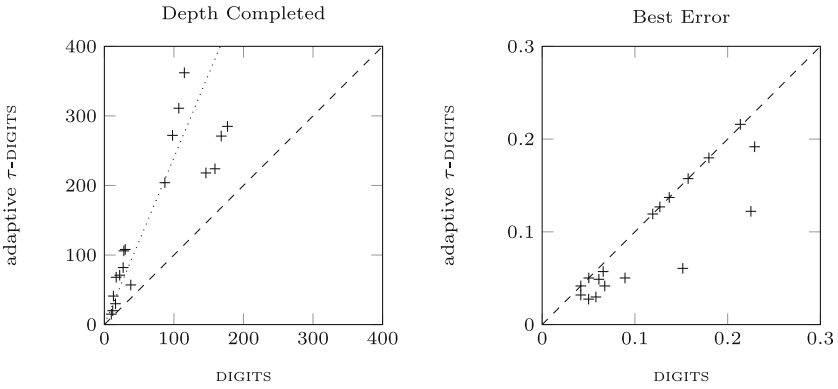

Fig. 4. Improvement of using adaptive τ-DIGITS on the original DIGITS benchmarks. Left: the dotted line marks the 2.4× average increase in depth.

Figure 4 shows, for each benchmark, (*i*) the largest sample set size completed by adaptive τ-DIGITS versus DIGITS (left—above the diagonal line indicates adaptive τ-DIGITS reaches further depths), and (*ii*) the error of the best solution found by adaptive τ-DIGITS versus DIGITS (right—below the diagonal line indicates adaptive τ-DIGITS finds better solutions). We see that adaptive τ-DIGITS reaches further depths on every problem instance, many of which are substantial improvements, and that it finds better solutions on 10 of the 18 problems. For those which did not improve, either the search was already deep enough that DIGITS was able to find near-optimal solutions, or the complexity of the synthesis queries is such that the search is still constrained to small depths.

Findings (RQ2): Adaptive τ-DIGITS can find better solutions than those found by DIGITS and can reach greater search depths.

5.3 Thermostat Controller

We challenge adaptive τ-DIGITS with the task of synthesizing a thermostat controller, borrowing the benchmark from [7]. The input to the controller is the initial temperature of the environment; since the world is uncertain, there is a specified probability distribution over the temperatures. The controller itself is a program sketch consisting primarily of a single main loop: iterations of the loop correspond to timesteps, during which the synthesized parameters dictate an incremental update made by the thermostat based on the current temperature. The loop runs for 40 iterations, then terminates, returning the absolute value of the difference between its final actual temperature and the target temperature.

The postcondition is a Boolean probabilistic correctness property intuitively corresponding to controller safety, e.g. with high probability, the temperature should never exceed certain thresholds. In [7], there is a quantitative objective in the form of minimizing the expected value $\mathrm{E}[|\mathit{actual} - \mathit{target}|]$—our setting does not admit optimizing with respect to expectations, so we must modify the problem. Instead, we fix some value N ($N \in \{2, 4, 8\}$) and have the program return 0 when $|\mathit{actual} - \mathit{target}| < N$ and 1 otherwise. Our quantitative objective is to minimize the error from the constant-zero functional specification $\hat{P}(x) := 0$ (i.e. the actual temperature always gets close enough to the target). The full specification of the controller is provided in the full version of our paper [9].

We consider variants of the program where the thermostat runs for fewer timesteps and try loop unrollings of size $\{5, 10, 20, 40\}$. We run each benchmark for 10 min: the final completed search depths and best error of solutions are shown in Fig. 5. For this particular experiment, we use the SMT solver CVC4 [3] because it performs better than Z3 on the occurring SMT instances.

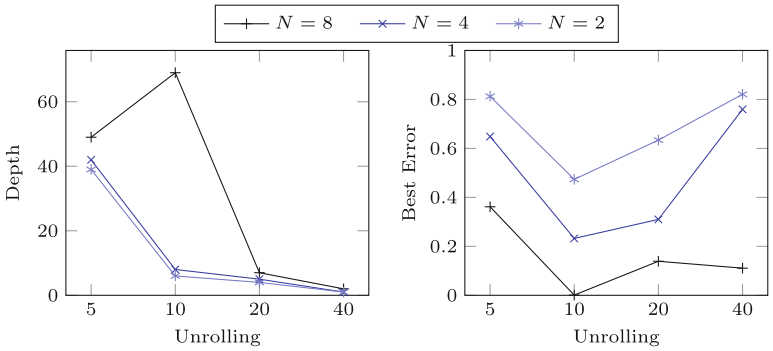

Fig. 5. Thermostat controller results.

As we would expect, for larger values of N it is "easier" for the thermostat to reach the target temperature threshold and thus the quality of the best solution increases in N. However, with small unrollings (i.e. 5) the synthesized controllers do not have enough iterations (time) to modify the temperature enough for the

probability mass of extremal temperatures to reach the target: as we increase the number of unrollings to 10, we see that better solutions can be found since the set of programs are capable of stronger behavior.

On the other hand, the completed depth of the search plummets as the unrolling increases due to the complexity of the $\mathcal{O}_{\text{syn}}$ queries. Consequently, for 20 and 40 unrollings, adaptive τ-DIGITS synthesizes worse solutions because it cannot reach the necessary depths to obtain better guarantees.

One final point of note is that for $N = 8$ and 10 unrollings, it seems that there is a sharp spike in the completed depth. However, this is somewhat artificial: because $N = 8$ creates a very lenient quantitative objective, an early $\mathcal{O}_{\text{syn}}$ query happens to yield a program with an error less than 10^{-3}. Adaptive τ-DIGITS then updates $\tau \leftarrow\approx 10^{-3}$ and skips most synthesis queries.

Findings (RQ3): Adaptive τ-DIGITS can synthesize small variants of a complex thermostat controller, but cannot solve variants with many loop iterations.

6 Related Work

Synthesis and Probability. Program synthesis is a mature area with many powerful techniques. The primary focus is on synthesis under Boolean constraints, and probabilistic specifications have received less attention [1,7,17,19]. We discuss the works that are most related to ours.

DIGITS [1] is the most relevant work. First, we show for the first time that DIGITS only requires a number of synthesis queries polynomial in the number of samples. Second, our adaptive τ-DIGITS further reduces the number of synthesis queries required to solve a synthesis problem without sacrificing correctness.

The technique of *smoothed proof search* [7] approximates a combination of functional correctness and maximization of an expected value as a smooth, continuous function. It then uses numerical methods to find a local optimum of this function, which translates to a synthesized program that is likely to be correct and locally maximal. The benchmarks described in Sect. 5.3 are variants of benchmarks from [7]. Smoothed proof search can minimize expectation; τ-DIGITS minimizes probability only. However, unlike τ-DIGITS, smoothed proof search lacks formal convergence guarantees and cannot support the rich probabilistic postconditions we support, e.g., as in the fairness benchmarks.

Works on synthesis of probabilistic programs are aimed at a different problem [6,19,23]: that of synthesizing a generative model of data. For example, Nori et al. [19] use sketches of probabilistic programs and complete them with a stochastic search. Recently, Saad et al. [23] synthesize an ensemble of probabilistic programs for learning Gaussian processes and other models.

Kŭcera et al. [17] present a technique for automatically synthesizing program transformations that introduce uncertainty into a given program with the goal of satisfying given privacy policies—e.g., preventing information leaks. They leverage the specific structure of their problem to reduce it to an SMT constraint solving problem. The problem tackled in [17] is orthogonal to the one targeted in this paper and the techniques are therefore very different.

Stochastic Satisfiability. Our problem is closely related to E-MAJSAT [18], a special case of *stochastic satisfiability* (SSAT) [20] and a means for formalizing probabilistic planning problems. E-MAJSAT is of NP^{PP} complexity. An E-MAJSAT formula has deterministic and probabilistic variables. The goal is to find an assignment of deterministic variables such that the probability that the formula is satisfied is above a given threshold. Our setting is similar, but we operate over complex program statements and have an additional optimization objective (i.e., the program should be close to the functional specification). The deterministic variables in our setting are the holes defining the search space; the probabilistic variables are program inputs.

Acknowledgements. We thank Shuchi Chawla, Yingyu Liang, Jerry Zhu, the entire fairness reading group at UW-Madison, and Nika Haghtalab for all of the detailed discussions. This material is based upon work supported by the National Science Foundation under grant numbers 1566015, 1704117, and 1750965.

References

1. Albarghouthi, A., D'Antoni, L., Drews, S.: Repairing decision-making programs under uncertainty. In: Majumdar, R., Kunčak, V. (eds.) Computer Aided Verification, pp. 181–200. Springer International Publishing, Cham (2017)
2. Barowy, D.W., Gulwani, S., Hart, T., Zorn, B.G.: Flashrelate: extracting relational data from semi-structured spreadsheets using examples. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, 15–17 June 2015, pp. 218–228 (2015). https://doi.org/10.1145/2737924.2737952
3. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_14
4. Bastani, O., Sharma, R., Aiken, A., Liang, P.: Synthesizing program input grammars. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, 18–23 June 2017, pp. 95–110 (2017). https://doi.org/10.1145/3062341.3062349
5. Blumer, A., Ehrenfeucht, A., Haussler, D., Warmuth, M.K.: Learnability and the vapnik-chervonenkis dimension. J. ACM (JACM) **36**(4), 929–965 (1989)
6. Chasins, S., Phothilimthana, P.M.: Data-driven synthesis of full probabilistic programs. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 279–304. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_14
7. Chaudhuri, S., Clochard, M., Solar-Lezama, A.: Bridging boolean and quantitative synthesis using smoothed proof search. In: POPL, vol. 49, pp. 207–220. ACM (2014)
8. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
9. Drews, S., Albarghouthi, A., D'Antoni, L.: Efficient synthesis with probabilistic constraints (2019). http://arxiv.org/abs/1905.08364
10. El-Hassany, A., Tsankov, P., Vanbever, L., Vechev, M.: Network-wide configuration synthesis. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10427, pp. 261–281. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63390-9_14

11. Feldman, M., Friedler, S.A., Moeller, J., Scheidegger, C., Venkatasubramanian, S.: Certifying and removing disparate impact. In: Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 259–268. ACM (2015)
12. Goldberg, P.W., Jerrum, M.: Bounding the vapnik-chervonenkis dimension of concept classes parameterized by real numbers. Mach. Learn. **18**(2–3), 131–148 (1995). https://doi.org/10.1007/BF00993408
13. Gulwani, S.: Automating string processing in spreadsheets using input-output examples. In: Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, 26–28 January 2011, pp. 317–330 (2011). https://doi.org/10.1145/1926385.1926423
14. Gulwani, S.: Program synthesis. In: Software Systems Safety, pp. 43–75 (2014). https://doi.org/10.3233/978-1-61499-385-8-43
15. Gulwani, S.: Programming by examples - and its applications in data wrangling. In: Dependable Software Systems Engineering, pp. 137–158 (2016). https://doi.org/10.3233/978-1-61499-627-9-137
16. Kearns, M.J., Vazirani, U.V.: An Introduction to Computational Learning Theory. MIT Press, Cambridge (1994)
17. Kučera, M., Tsankov, P., Gehr, T., Guarnieri, M., Vechev, M.: Synthesis of probabilistic privacy enforcement. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, pp. 391–408. ACM, New York (2017). https://doi.org/10.1145/3133956.3134079
18. Littman, M.L., Goldsmith, J., Mundhenk, M.: The computational complexity of probabilistic planning. J. Artif. Intell. Res. **9**, 1–36 (1998)
19. Nori, A.V., Ozair, S., Rajamani, S.K., Vijaykeerthy, D.: Efficient synthesis ofprobabilistic programs. SIGPLAN Not. **50**(6), 208–217 (2015). https://doi.org/10.1145/2813885.2737982
20. Papadimitriou, C.H.: Games against nature. J. Comput. Syst. Sci. **31**(2), 288–301 (1985)
21. Polozov, O., Gulwani, S.: Flashmeta: a framework for inductive program synthesis. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, 25–30 October 2015, pp. 107–126 (2015). https://doi.org/10.1145/2814270.2814310
22. Raza, M., Gulwani, S.: Automated data extraction using predictive program synthesis. In: Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, 4–9 February 2017, San Francisco, California, USA, pp. 882–890 (2017). http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/15034
23. Saad, F.A., Cusumano-Towner, M.F., Schaechtle, U., Rinard, M.C., Mansinghka, V.K.: Bayesian synthesis of probabilistic programs for automatic data modeling. Proc. ACM Program. Lang. **3**(POPL), 37 (2019)
24. Sauer, N.: On the density of families of sets. J. Comb. Theory, Seri. A **13**(1), 145–147 (1972)
25. Schkufza, E., Sharma, R., Aiken, A.: Stochastic program optimization. Commun. ACM **59**(2), 114–122 (2016). https://doi.org/10.1145/2863701
26. Shelah, S.: A combinatorial problem; stability and order for models and theories in infinitary languages. Pac. J. Math. **41**(1), 247–261 (1972)
27. Solar-Lezama, A.: Program Synthesis by Sketching. Ph.D. thesis, Berkeley, CA, USA (2008), aAI3353225

28. Srinivasan, V., Reps, T.W.: Synthesis of machine code from semantics. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, 15–17 June 2015, pp. 596–607 (2015). https://doi.org/10.1145/2737924.2737960
29. Subramanian, K., D'Antoni, L., Akella, A.: Genesis: synthesizing forwarding tables in multi-tenant networks. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, 18–20 January 2017, pp. 572–585 (2017). http://dl.acm.org/citation.cfm?id=3009845
30. Wang, X., Gulwani, S., Singh, R.: FIDEX: filtering spreadsheet data using examples. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, 30 October - 4 November 2016, pp. 195–213 (2016). https://doi.org/10.1145/2983990.2984030

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.