Bayesian Program Learning via Sampling

Abstract

We introduce an algorithm for learning programs from data in a probabilistic framework. Our algorithm can sample from posterior distributions over programs conditioned on data, where there are theoretical guarantees on how well the samples approximate the true posterior. The algorithm uses a symbolic solver to search for programs, making it efficient in practice, allowing us to evaluate our approach on 22 program learning problems across the domains of text editing and computer-aided programming.

Input	Output
“1/21/2001”	“01”

↓

`substr(pos('0','',-1),-1)`	“last 0 til end”
`const('01')`	“output 01”
`substr(-2,-1)`	“take last two”

Figure 1. Learning string manipulation programs by example (top input/output pair). Our system receives data like that shown above and then sampled the different programs shown below (among others) from a description-length prior.

1. Introduction

Learning programs from examples is a central problem in artificial intelligence, and many recent approaches draw on techniques from machine learning. Connectionist approaches, like the Neural Turing Machine (Graves et al., 2014) and symbolic approahes, like Hierarchical Bayesian Program Learning (Lake et al., 2015; Liang et al., 2010; Menon et al., 2013), couple a probabilistic learning framework with either gradient- or sampling-based search procedures. In this work, we consider the problem of Bayesian inference over program spaces. We combine solver-based program synthesis (Solar Lezama, 2008) and sampling via random projections (Ermon et al., 2013), showing how to sample from posterior distributions over programs where the samples provably come from a distribution arbitrarily close to the true posterior. The new approach is implemented in a system called PROGRAMSAMPLE and evaluated on a suite of synthesis problems that include list and string manipulation routines.

1.1. Motivation and problem statement

Consider the problem of learning string edit programs, a well studied domain for programming by example. Often end users provide these examples and are unwilling to give more than one instance, which leaves the target program highly ambiguous. One way of handling this ambiguity is to *sample* string edit programs, allowing us to learn from very few examples (Fig. 1) and offer different plausible solutions.

Another program learning domain comes from *computer-aided programming*, where the goal is to synthesize algorithms from either examples or formal specifications. This problem can be ill posed because many programs may satisfy the specification or examples. When this ambiguity arises, PROGRAMSAMPLE can propose multiple implementations with a bias towards, eg, shorter or simpler ones. The samples can also be used to form a posterior predictive distribution, effectively integrating out the program. We show our system learning counting routines and recursive sorting/reversing algorithms while modeling the uncertainty over the correct implementation of the algorithm.

Because any model can be represented as a (probabilistic or deterministic) program, we need to carefully delimit the scope of this work. The programs we learn are a subset of those handled by constraint-based program synthesis tools. This means that the program is *finite* (bounded size, bounded runtime, bounded memory consumption), can be modeled in a constraint solver (like a SAT or SMT solver), and that the program's high-level structure is already given as a *sketch*, (Solar Lezama, 2008) which can take the form of a recursive grammar over expressions. The sketch constrains the search space and imparts prior knowledge. For example, we use one sketch when learning string edit programs and a different sketch when learning recursive list manipulation programs.

More formally, our sketch specifies a finite set of programs, X. We want to sample from the distribution $p(x) = \frac{2^{-|x|}}{Z}$ where $|x|$ is the description length of $x \in X$ (ie, $|x|$ is a nat-

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

ural number) and $Z = \sum_{x \in X} 2^{-|x|}$. The set X will in general be very large and subject to constraints upon the program's behavior, for example, consistency with input/output examples. We can invoke a *solver*, which enumerates members of X, possibly subject to extra constraints, but without any guarantees on the order of enumeration. We use a SAT solver, and encode $x \in X$ in the values of n SAT variables. With a slight abuse of notation we will use x to refer to both a member of X and an assignment to those n SAT variables. An assignment to j^{th} variable we write as x_j for $1 \leq j \leq n$.

1.2. Algorithmic contribution

Over the past decade there has been the concurrent development of solver-basedtechniques for (1) sampling of combinatorial spaces (Gomes et al., 2006b; Ermon et al., 2013; 2012; Chakraborty et al., 2014b) and (2) program synthesis (Solar Lezama, 2008; Gulwani et al., 2011). This work merges these two lines of research to attack the problem of program learning in a probabilistic setting. We use program synthesis tools to convert a program learning problem into a boolean satisfiability formula (SAT). Then, rather than search for one program (formula solution), we augment the formula with random constraints that cause it to (approximately) sample the space of programs, effectively "upgrading" our SAT solver from a program synthesizer to a program sampler.

While one could use a tool like Sketch to reduce a program learning problem to SAT and then use an algorithm like PAWS, XORSample, or Unigen (Ermon et al., 2013; Gomes et al., 2006b; Chakraborty et al., 2014b) to sample programs from a description length prior, doing so can be extremely inefficient[1]. Intuitively, these solver-based samplers work by creating "duplicate" SAT solutions for high probability programs such that uniform sampling of SAT solutions gives nonuniform sampling of programs. When there are a few very likely (short) programs and many extremely unlikely (long) programs, these techniques scale poorly, because they have to "duplicate" the short programs many times. This phenomenon has been observed more generally when using these techniques to sample from large combinatorial spaces, and is related to a quantity called the distribution's *tilt* (Chakraborty et al., 2014a). Previous work has relied on upper bounding the tilt. Our main technical contribution is showing how to extend these techniques to distributions with high tilt, such as those encountered in program synthesis problems, where there are often a few short, high-probability programs and many long, low-probability programs.

[1]In many cases, slower than rejection sampling or enumerating all of the programs

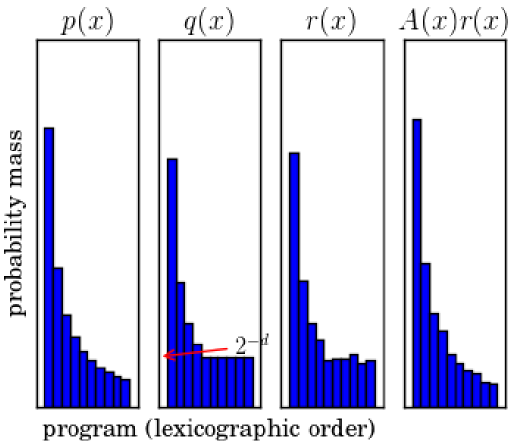


Figure 2. PROGRAMSAMPLE twice distorts the distribution $p(\cdot)$. First it introduces a parameter d that bounds the tilt; we correct for this by accepting samples w.p. $A(x)$. Second it samples from $q(\cdot)$ by drawing instead from $r(\cdot)$, where $KL(q||r)$ can be made arbitrarily small by appropriately setting another parameter, K. $A(x)r(x)$ is distribution of samples.

2. The sampling algorithm

The key idea is to approximate the distribution $p(\cdot)$ with another distribution, $q(\cdot)$, whose *tilt*, defined in (Chakraborty et al., 2014a) as $\frac{\max_x q(x)}{\min_x q(x)}$, is guaranteed to be small and whose KL divergence from $p(\cdot)$ is also guaranteed to be small. We then embed X in a higher dimensional space, E, such that sampling uniformly from E corresponds to sampling from another distribution $r(\cdot)$, whose fluctuations around $q(\cdot)$ can be made arbitrarily small. We can sample uniformly from E using a variety of techniques based on adding XOR constraints (random projections mod 2) to the set X, which are extensively studied in, for example, (Gomes et al., 2006b; Valiant & Vazirani, 1985; Chakraborty et al., 2014b; Gomes et al., 2006a). A final step of rejection sampling corrects for the low-tilt approximation. Fig. 2 illustrates this process, which we call PROGRAMSAMPLE .

2.1. Getting high-quality samples

Low-tilt approximation. We introduce a parameter into the sampling algorithm, d, that parameterizes $q(\cdot)$. The parameter d acts as a threshold, or cut-off, for the description length of a program; the distribution $q(\cdot)$ acts as though any program with description length exceeding d can be encoded using d bits. Concretely,

$$q(x) \propto \begin{cases} 2^{-|x|}, & \text{if } |x| \leq d \\ 2^{-d}, & \text{otherwise} \end{cases} \tag{1}$$

If we could sample exactly from $q(\cdot)$, we could reject a sample x with probability $1 - A(x)$ where A is

$$A(x) \propto \begin{cases} 1, & \text{if } |x| \le d \\ 2^{-|x|+d}, & \text{otherwise} \end{cases} \tag{2}$$

and get exact samples from $p(\cdot)$, where the acceptance rate would approach 1 exponentially quickly in d.

Proposition 1. *Let $x \in X$ be a sample from $q(\cdot)$. The probability of accepting x is at least $\frac{1}{1+|X|2^{|x_*|-d}}$ where $x_* = \arg\min_x |x|$.*

Proof. The probability of acceptance is

$$\sum_x q(x)A(x) = \sum_x \frac{2^{-|x|}}{\sum_{x'} q(x')} \tag{3}$$

$$= \frac{Z}{Z + \sum_{|x|>d}(2^{-d} - 2^{-|x|}]} \tag{4}$$

$$> \frac{1}{1+|X|2^{-d}/Z} > \frac{1}{1+|X|2^{|x_*|-d}}. \tag{5}$$

□

The distribution $q(\cdot)$ is useful because we can guarantee that it has tilt bounded by $d - |x_*|$. Recent work has shown that the performance of nonuniform sampling via random projections degrades as the tilt increases (Chakraborty et al., 2014a); introducing the proposal $q(\cdot)$ effectively reifies the tilt, making it a parameter of the program induction algorithm, not the distribution over programs.

We now show how to approximately sample from $q(\cdot)$ using a variant of the Embed and Project framework (Ermon et al., 2013).

The embedding. The idea is to define a new set of programs, which we call E, such that short programs are included in the set much more often than long programs. Each program x will be represented in E by an amount proportional to $2^{-\min(|x|,d)}$, thus proportional to $q(x)$, such that sampling elements uniformly from E samples according to $q(\cdot)$.

We embed X within the larger set E by introducing d *auxiliary variables*, written $(y_1, \cdots, y_d)$, such that every element of E is a tuple of an element of $x = (x_1, \cdots, x_n)$ and an assignment to $y = (y_1, \cdots, y_d)$:

$$E = \{(x,y) \,:\, x \in X, \bigwedge_{1 \le j \le d} |x| \ge j \implies y_j = 1\} \tag{6}$$

Suppose we sample (x, y) uniformly from E. Then the probability of getting a particular $x \in X$ is proportional to

$$|\{(x', y) \in E : x' = x\}| =$$

$$|\{y : |x| \ge j \implies y_j = 1\}| = 2^{\min(0,d-|x|)} \tag{7}$$

which is proportional to $q(x)$. Notice that $|E|$ grows exponentially with d, and thus with the tilt of the $q(\cdot)$. This is the crux of the inefficiency of sampling from high-tilt distributions in these frameworks: empirically, generating uniform samples using random projections, described below, becomes inefficient when the size of the support of the distribution is very high.

The random projections. We could sample exactly from E by invoking the solver $|E|+1$ times to get every element of E, but in general it will have $O(|X|2^d)$ elements, which could be very large. Instead, we ask the solver for all the elements of E consistent with K random constraints such that (1) few elements of E are likely to satisfy ("survive") the constraints, and (2) any element of E is approximately equally likely to satisfy the constraints. We can then sample a survivor uniformly to get an approximate sample from E, an idea introduced in the XORSample' algorithm (Gomes et al., 2006b).

Our random constraints take the form of XOR, or parity constraints, which are random projections mod 2. Each constraint fixes the parity of a random subset of SAT variables in x to either 1 or 0; thus any x survives a constraint with probability $\frac{1}{2}$. A surprising feature of random parity constraints is that whether an assignment to the SAT variables survives is independent of whether another, different assignment survives. These events are even 3-wise independent, and these facts have been exploited to create a variety of approximate sampling algorithms (Gomes et al., 2006b; Valiant & Vazirani, 1985; Chakraborty et al., 2014b; Gomes et al., 2006a).

Then the K constraints are of the form $h \left(\begin{smallmatrix} x \\ y \end{smallmatrix}\right) \overset{2}{\equiv} b$ where h is a $K \times (d+n)$ binary matrix and b is a K-dimensional binary vector. If no solutions satisfy the K constraints than the sampling attempt is rejected.

These samples are close to uniform in the following sense:

Proposition 2. *The probability of sampling (x, y) is at least $\frac{1}{|E|} \times \frac{1}{1+2^K/|E|}$ and the probability of getting any sample at all is at least $1 - 2^K/|E|$.*

Proof. The probability sampling (x, y), given that (x, y) survives the K constraints, is $\frac{1}{\text{mc}}$, where mc is the model count (# of survivors). The probability of (x, y) surviving the K constraints is 2^{-K} and is independent of whether any other element of E survives the constraints (Gomes et al., 2006b). So the probability of sampling (x, y) is

$$2^{-K} \sum_{i=1}^{|E|} P(\text{mc} = i | (x,y) \text{ survives}) \frac{1}{i} \tag{8}$$

$$= 2^{-K} E[\frac{1}{mc} | (x,y) \text{ survives}] \tag{9}$$

$$> 2^{-K}\frac{1}{E[mc|(x,y)\text{ survives}]}, \text{Jensen's inequality} \quad (10)$$

$$= 2^{-K}\frac{1}{1+(|E|-1)2^{-K}}, \text{pairwise independence} \quad (11)$$

$$> \frac{1}{|E|}\times\frac{1}{1+2^K/|E|}. \quad (12)$$

We fail to get a sample if mc $= 0$. We bound the probability of this event using Chebyshev's inequality: $E[\text{mc}] = |E|2^{-K} > \text{Var}(\text{mc})$, so

$$P(\text{mc}=0) \leq P(|\text{mc}-E[\text{mc}]| \geq E[\text{mc}]) \quad (13)$$

$$\leq \frac{\text{Var}(mc)}{E[\text{mc}]^2} < 1/E[\text{mc}] = 2^K/|E|. \quad (14)$$

□

So we get approximate samples from E as long as $|E|2^{-E}$ is not small. In reference to Fig. 2, we call the distribution of these samples $r(x) = \sum_y r(x,y)$. Schemes more sophisticated than XORSample', like (Ermon et al., 2013), also guarantee upper bounds on sampling probability, but we found that these were unnecessary for our main result, which is that the KL between $p(\cdot)$ and $A(x)r(x)$ goes to zero exponentially quickly in a new quantity we call Δ:

Proposition 3. *Write $Ar(x)$ to mean the distribution proportional to $A(x)r(x)$. Then $D(p||Ar) < \log\left(1+\frac{1+2^{-\gamma}}{1+2^{\Delta}}\right)$ where $\Delta = \log|E| - K$ and $\gamma = d - \log|X| - |x_*|$.*

Proof. Define $c = \frac{1}{1+2^K/|E|}$. As $p(x) \propto A(x)q(x)$,

$$D(p||Ar) = \sum_x p(x)\log\frac{p(x)\sum_y A(y)r(y)}{A(x)r(x)} \quad (15)$$

$$= \sum_x p(x)\log\frac{A(x)q(x)}{\sum_y A(y)q(y)}\frac{\sum_y A(y)r(y)}{A(x)r(x)} \quad (16)$$

$$= \log\frac{\sum_x A(x)r(x)}{\sum_x A(x)q(x)} + \sum_x p(x)\log\frac{q(x)}{r(x)} \quad (17)$$

$$< \log\frac{\sum_x A(x)r(x)}{\sum_x A(x)q(x)} - \log c \quad (18)$$

where 18 comes from Proposition 2. We know that $A(x) \leq 1 = A(x_*)$, that $r(x) \geq cq(x)$, and $\sum_x r(x) = P(\text{mc} > 0)$. Optimizing subject to these constraints,

$$\sum_x A(x)r(x) < P(\text{mc}>0) - \sum_{x\neq x_*} cq(x) + \sum_{x\neq x_*} cq(x)A(x)$$

$$= P(\text{mc}>0) + c\sum_x A(x)q(x) - c. \quad (19)$$

So the KL divergence is bounded above by

$$D(p||Ar) < \log\left(c + \frac{P(\text{mc}>0)-c}{\sum_x A(x)q(x)}\right) - \log c \quad (20)$$

The quantity $\sum_x A(x)q(x)$ is the probability of accepting a perfect sample from $q(\cdot)$, which Proposition 1 lower bounds:

$$D(p||Ar) < \log\left(c + (1-c)(1+2^{-\gamma})\right) - \log c \quad (21)$$

$$= \log\left(\frac{1}{1+2^{-\Delta}} + \frac{1+2^{-\gamma}}{1+2^{\Delta}}\right) + \log(1+2^{-\Delta}) \quad (22)$$

which for the sake of clarity we can weaken to

$$D(p||Ar) < \log\left(1+\frac{1+2^{-\gamma}}{1+2^{\Delta}}\right). \quad (23)$$

□

So we can approximate the true distribution $p(\cdot)$ arbitrarily well. Here we depart from the Unigen and Uniwit families of approaches (Chakraborty et al., 2014a;b) which only guarantee that the distribution of samples is with in a factor of 6.84 of the target distribution. Other approaches, like PAWS (Ermon et al., 2013), can prove analogously strong bounds.

Application of Proposition 3 requires knowledge $\min_x |x|$, which we compute using the iterative minimization routine in (Singh et al., 2013); in practice this is very efficient for finite program spaces. We also need to calculate $|X|$ and $|E|$, which are model counts that are in general difficult to compute exactly. However, many approximate model counting schemes exist, which provide upper and lower bounds that hold with arbitrarily high probability. We use Hybrid-MBound (Gomes et al., 2006a) to upper bound $|X|$ and lower bound $|E|$ that each individually hold with probability at least $1-\delta/2$ (see Alg. 1), thus giving lower bounds on both the γ and Δ parameters of Proposition 3 with probability at least $1-\delta$. and thus an upper bound on the KL divergence.

Efficient enumeration. The embedding E introduces a symmetry into the solution space of the SAT formula, where one program (an x) corresponds to many points in the embedding (pairs (x,y)). We can more efficiently enumerate surviving members of E by only enumerating unique surviving programs, and then counting the corresponding members of E implicitly through the following result:

Proposition 4. *Let $x \in X$ and $(x,y) \in E$ satisfy $h(a,y) \overset{2}{\equiv} b$. If $|x| \geq d$ then (x,y) is the only surviving member of E corresponding to x. Otherwise there are $2^{d-|x|-rank(g)}$ survivors where g is the rightmost $d-|x|$ columns of h.*

Algorithm 1 ProgramSample

```
Input: Program space $X$, failure probability $\delta$, natural numbers $\Delta$, $\gamma$, number of samples $N$
Output: $N$ samples
Set $|x_*| = \min_{x \in X} |x|$
Set $B_X$ = ApproximateUpperBoundModelCount($X$,$\delta/2$)
Set $d = \gamma + \log B_X + |x_*|$
Define $E = \{(x, y) : x \in X, \bigwedge_{1 \leq j \leq d} |x| \geq j \implies A_j = 1\}$
Set $B_E$ = ApproximateLowerBoundModelCount($E$,$\delta/2$)
Set $K = \log B_E - \Delta$
Initialize samples = $\varnothing$
repeat
  Sample $h$ uniformly from $\{0, 1\}^{(d+n)\times K}$
  Sample $b$ uniformly from $\{0, 1\}^K$
  Enumerate $S = \{(x, y)$ where $h(x, y) = b \wedge x \in X\}$
  if $|S| > 0$ then
    Sample $(x, y)$ uniformly from $S$
    if Uniform(0, 1) $< 2^{d-|x|}$ then
      samples = samples $\cup \{x\}$
    end if
  end if
until |samples| = $N$
return samples
```

Proof. If $|x| \geq d$ then there is only one element of E corresponding to x. Otherwise, any assignment to y satisfying

$$b \overset{2}{\equiv} \begin{pmatrix} h_x & \vdots & h_y & \vdots & g \end{pmatrix} \begin{pmatrix} x \\ y_{\leq|x|} \\ y_{>|x|} \end{pmatrix} \quad (24)$$

satisfies the random hashing constraints, where we have partitioned the columns of h into those multiplied into x, $y_{\leq|x|}$, and $y_{>|x|}$. Because $(x, y) \in E$ the values of x and $y_{\leq|x|}$ are fixed, so we can define a new vector $c \overset{2}{\equiv} b + h_x x + h_y y_{\leq|x|}$ and rewrite Eq. 24 as $c = g y_{>|x|}$. Let $r = \text{rank}(g)$. Then there is a coordinate system where Eq. 24 reads

$$c \overset{2}{\equiv} \begin{pmatrix} 1 & & & \\ & \ddots & & \\ & & 0 & \\ & & & \ddots \end{pmatrix} \begin{pmatrix} y_{1+|x|} \\ \vdots \\ y_{1+|x|+r} \\ \vdots \end{pmatrix} \quad (25)$$

Eq. 25 is satisfied iff, for all $1 \leq j \leq r$, $y_{j+|x|} = c_j$. For $j > r$ the entries of $y_{j+|x|}$ are unconstrained, and so $2^{d-|x|-r}$ satisfying values for y exist. □

This enumeration strategy helps when sampling from sharply peaked posteriors, where there are few surviving programs; it also bounds the number of solver invocation to $|X|$.

2.2. Accuracy/Performance trade-off

We now analyze the runtime of Alg. 1, using number of solver calls as a proxy for runtime. Our main results are that (1) the amortized time/sample decreases as we increase the dimension of the embedding, and (2) decreases as we increase the number of random constraints, which introduces a trade-off between accuracy of the samples and speed of the sampling.

Proposition 5. *The expected number of calls to the solver per sample is bounded above by* $\frac{1+2^{\Delta}}{(1+2^{-\gamma})^{-1}(1+2^{-\Delta})^{-1}-2^{-\Delta}}$.

Proof. First upper bound the probability of failing, $P(\text{fail})$, to get a sample, which could happen if S is empty or if the sample from S is rejected, which is distributed according to $r(\cdot)$:

$$\begin{aligned} P(\text{fail}) &< P(\text{reject}) + P(\text{mc} = 0), \text{ union bound} \quad (26)\\ &< 1 - \frac{\sum_x A(x)q(x)}{1 + 2^K/|E|} + 2^K/|E|, \text{ Prop. 2} \quad (27)\\ &< 1 - \frac{1}{(1 + 2^{-\Delta})(1 + 2^{-\gamma})} + 2^{-\Delta}, \text{ Prop. 1} \\ &\qquad (28) \end{aligned}$$

The expected number of solver invocations per iteration is $< 1 + E[\text{mc}] = 1 + |E|2^{-K} = 1 + 2^{\Delta}$ and the expected number of iterations is $1/P(\neg\text{failure})$. Because the iterations are independent the expected number of solver invocations is just their product, which is the desired result. □

Proposition 5 shows that the number of invocations to the solver grows exponentially in Δ, while Proposition 3 shows that the KL divergence from $p(\cdot)$ decays exponentially in Δ. Algorithm 1 balances this trade-off through its preliminary model counting steps; see Fig. 3.

3. Experimental results

We evaluated ProgramSample on program learning problems in a text editing domain and a list manipulation domain. For each domain, we wrote down a sketch using the tool in (Solar-Lezama et al., 2006), specifying a large but finite set of possible programs. This implicitly defined a description-length prior, where $|x|$ is the number of bits required to specify x in the SAT encoding. We used CryptoMiniSAT (cry), which can efficiently handle parity constraints.

3.1. Learning Text Edit Scripts

We applied our program sampling algorithm to a suite of programming by demonstration problems within a text editing domain. Here, the challenge is to learn a small

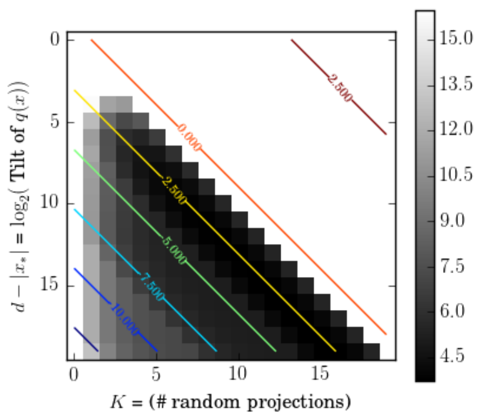


Figure 3. Accuracy (colored contours) vs Performance (monochrome cells) trade-off for a program synthesis problem; upper bounds plotted. Performance measured in expected solver invocations; accuracy measured in log KL divergence. Prop. 1 lower bounds the tilt of performant samplers, while Prop. 2 upper bounds K to $O(d)$, forcing our sampler into the darker (faster) regions. KL divergence falls off exponentially fast in $\Delta = O(d - K)$, (Prop. 3) while solver invocations grows exponentially in Δ (Prop. 5) but is bounded by $|X|$ (Prop. 4), shown in white.

text editing program from very few examples and apply that program to held out inputs. This problem is timely, given the widespread use of the Flashfill program synthesis tool, which now ships by default in Microsoft Excel (Gulwani et al., 2015) and can learn sophisticated edit operations in real time from examples. We modeled a subset of the Flashfill (Gulwani, 2011a) language; our goal here is not to compete with Flashfill, which is heavily engineered for its specific domain, but to study the behavior of our more general-purpose program learner in a real-world task. To impart domain knowledge, we used a sketch equivalent to the following grammar:

```
Program ::= Term | Program + Term
Term    ::= String | substr(Pos,Pos)
Pos     ::= Number | pos(String,
                         String,Number)
Number  ::= 0 | 1 | 2 | ...
          | -1 | -2 | ...
String  ::= Character
          |  Character + String
Character ::= a | b | c | ...
```

Because FlashFill's training set is not yet public, we drew text editing problems from (Lin et al., 2014) and adapted them to our subset of FlashFill, giving 19 problems, each with 5 training examples.

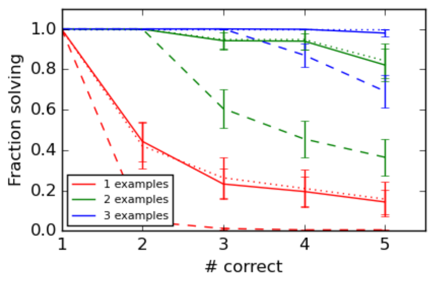


Figure 4. Generalization when learning text edit operations by example. Results averaged across 19 problems. Solid: 100 samples from PROGRAMSAMPLE . Dashed: enumerating 100 programs. Dotted: MDL learner. Test cases past 1 (resp. 2,3) examples are held out when trained on 1 (resp. 2,3) examples.

We are interested both in the ability of the learner to generalize and in PROGRAMSAMPLE's ability to generate samples quickly. Tbl. 1 shows the average time per sampling attempt using PROGRAMSAMPLE, which is on the order of a minute. These text edit problems come from distributions with extremely high tilt: often the smallest program is only tens of bits long, but the program space contains (implausible) solutions with over 100 bits. By putting d to $|x_*| - n$ we eliminate the tilt correction and recover a variant of the approaches in (Ermon et al., 2013). This baseline does not produce any samples for any of our text edit problems in under an hour.[2]

The learner generalizes to unseen examples, as Fig. 4 shows. We evaluated the performance of the learner on held out test examples as a function of training set size, and compares with baselines that either (1) enumerate programs in the arbitrary order provided by the underlying solver, or (2) takes the most likely program under $p(x)$ (MDL learner). The posterior is sharply peaked, with most samples being from the MAP estimate, and so our learner does about as well as the MDL learner. However, sampling offers an (approximate) predictive posterior over solutions to the held out examples; in a real world scenario, one would offer the top C solutions to the user and let them choose, much like how spelling correction works. Performing this procedure allows us to offer the correct solutions more often than the MDL learner (Fig. 5), because we are able to correctly handle ambiguous problems like in Fig. 1.

3.2. Learning list manipulation algorithms

One goal of program synthesis is *computer-aided programming* (Solar Lezama, 2008), where a program induction

[2]Approximate model counting of E was also intractable in this regime, so we used the lower bound $|E| \geq 2^{d-|x_*|} + |X| - 1$

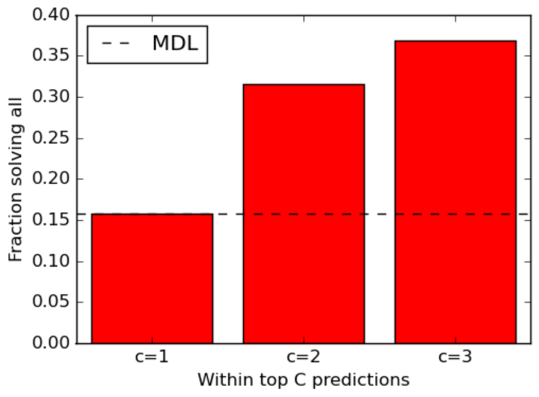


Figure 5. Comparing the MDL learner (dashed black line) to program sampling when doing one-shot learning. We count a problem as "solved" if the correct joint prediction to the test cases is in the top C most frequent samples.

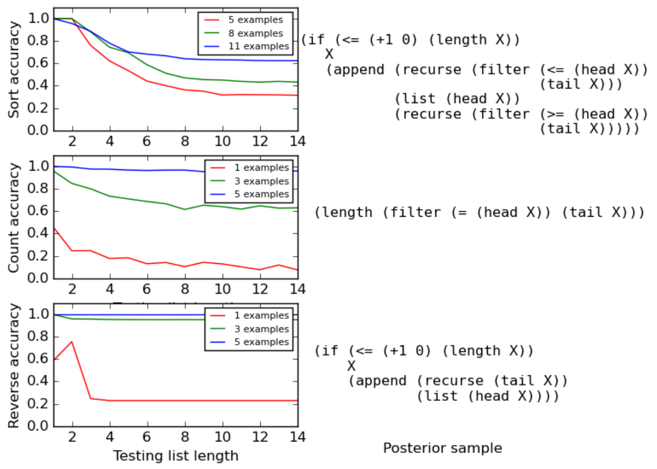
Table 1. Average solver time to generate a sample measured in seconds. See Fig. 6 and 4 for training set sizes.

	Large set	Medium set	Small set
text edit	49±3	21 ±1	84 ±3
sort	1549±155	905 ±58	463 ±65
reverse	326±42	141 ±18	39 ±3
count	≤ 1	≤ 1	≤ 1

system automatically generates executable code from either declarative specifications or examples of desired behavior. Systems with this goal has been successfully applied to, for example, synthesizing intricate bitvector routines from specifications (Gulwani, 2011b). However, when learning from examples, there is often uncertainty over the correct program. While past approaches have handled this uncertainty within an optimization framework (see (Raychev et al., 2016; Ellis et al., 2015; Singh et al., 2013)), we show that ProgramSample can *sample* algorithms. We take as our goal to learn recursive routines for sorting, reversing, and counting list elements from input/output examples, particularly in the ambiguous, unconstrained regime of few examples.

We wrote down a sketch with a set of basis primitives capable of representing a range of list manipulation routines equivalent to the following grammar:

```
Program   ::= (if Predicate List
                  (append RecursiveList
                          RecursiveList
                          RecursiveList))
Predicate ::= (<= Number) | (>= Number)
Number    ::= 0 | (1+ Number) | (1- Number)
              | (length List) | (head List)
List      ::= a | nil | (list Number)
```



```
(if (<= (+1 0) (length X))
  X
  (append (recurse (filter (<= (head X))
                           (tail X)))
          (list (head X))
          (recurse (filter (>= (head X))
                           (tail X)))))
```

```
(length (filter (= (head X)) (tail X)))
```

```
(if (<= (+1 0) (length X))
    X
    (append (recurse (tail X))
            (list (head X))))
```

Figure 6. Generalization performance of the program learner on list manipulation tasks. Trained on random lists of length ≤ 3.

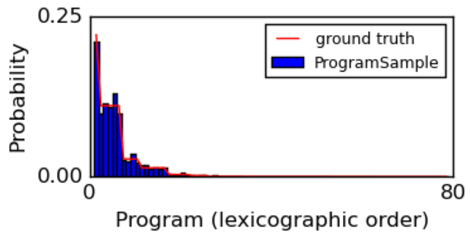


Figure 7. Sampling frequency vs. ground truth probability for 1000 runs of ProgramSample on a counting (see Fig. 6) task.

```
              | (tail List)
              | (filter Predicate List)
RecursiveList ::= List | (recurse List)
```

A description-length prior that penalizes longer programs allowed learning of recursive list manipulation routines (from production `Program`) and a non-recursive count routine (from production `Number`); see Fig. 6, which shows average accuracy on held out test data when trained on variable numbers of short randomly generated lists. With the large training set (5–11 examples) ProgramSample recovers a correct implementation, and with less data it recovers a distribution over programs that functions effectively as a "noisy algorithm".

For some of these program learning tasks the number of consistent programs is small enough that we can enumerate all of them, allowing us to compare our sampler with the ground-truth probabilities. Fig. 7 shows this comparison for a problem with 80 consistent programs, showing empirically that the tilt correction and random constraints do not significantly perturb the distribution.

Tbl. 1 shows the average solver time per sample; we see that generating recursive routines like sorting and reversing is much more costly than generating the nonrecursive counting routine. Our constraint-based approach propositionalizes higher-order constructs like recursion, and so reasoning about them is much more costly.

4. Discussion

4.1. Related work

There is a vast literature on program learning within the AI and machine learning communities. Many employ a (possibly stochastic) heuristic search over structures using genetic programming (Koza, 1993) or MCMC (Schkufza et al., 2013). These approaches often find good programs and can discover more high-level structure than our approach. However, they are prone to getting trapped in local minima and, when used as a sampler, determining the mixing time is in general not possible. Recently researchers have turned to the problem of learning good priors over programs in a multitask setting (Liang et al., 2010; Menon et al., 2013; Dechter et al., 2013). We see our work here as particularly complementary to these methods: while they focus on learning the structure of the hypothesis space, we focus on efficiently sampling an already given hypothesis space (the sketch). There are several recent proposals for recurrent deep networks that learn algorithms (Reed & de Freitas, 2015; Graves et al., 2014). We see our system working in a different regime, where we want to quickly learn an algorithm from a small number of examples or an ambiguous specification.

The program synthesis community has several recently proposed learners that work in an optimization framework (Raychev et al., 2016; Ellis et al., 2015; Singh et al., 2013), and these motivate our description length prior. By computing a posterior over programs, we can more effectively represent uncertainty, particularly in the small data limit, but at the cost of more computation: our algorithm requires an optimization routine to calibrate the sampler.

Our sampling algorithm borrows heavily from a line of work started in (Gomes et al., 2006b;a) on sampling of combinatorial spaces using random XOR constraints, which is motivated by results in complexity theory (Valiant & Vazirani, 1985).

4.2. Limitations of the approach

The constraint-based synthesis methods on which our technique is built tend to excel in domains where the structure of the expected solutions can be restricted by a "sketch" (Solar Lezama, 2008) and where much of the program's description length can be easily computed from the program text. For example, PROGRAMSAMPLE can synthesize text editing programs taking almost 60 bits of description length in a couple seconds, but spends 10 minutes synthesizing a recursive sorting routine that takes fewer bits of description length but where the program structure is less restricted. Constraint-based methods also require the entire problem to be represented symbolically, so they have trouble when the function to be synthesized involves complex building blocks such as numerical routines that are difficult to analyze. For such problems, methods based on stochastic search (Nori et al., 2015; Schkufza et al., 2013; Koza, 1993) can be more effective because they only need to be able to run the functions under consideration. Finally, past work has also shown empirically that constraint-based approaches scale poorly with data set size, although this can be mitigated with clever approaches to consider data incrementally (Ellis et al., 2015; Raychev et al., 2016).

The requirement of producing representative samples imposes some additional overheads on our approach, so scalability can more limited than for standard symbolic techniques on some problems. For example, our method requires 1 MAP inference query, and 2 queries to an approximate model counter for every new problem. The approximate model counter serves to "calibrate" the sampler, and its cost can be amortized because it only has to be invoked twice in order to generate an arbitrary number of iid samples. Approximate model counters like MBound (Gomes et al., 2006a) have complexity comparable with that of generating a sample, but the complexity is strongly dependent on the number of solutions to the system. Thus, for good performance, PROGRAMSAMPLE requires that there not be too many programs consistent with the data—the largest spaces considered in our experiments had $\leq 10^7$ programs. This limitation, together with the general performance characteristics of symbolic techniques, means that the approach will work best for "needle in a haystack" problems, where the space of possible programs is large but restricted in its structure, and where only a small fraction of those programs satisfy the semantics constraints.

4.3. Future work

This work could naturally extend to other domains that involve inducing latent symbolic structure from small amounts of data, such as semantic parsing to logical forms (Liang et al., 2011), synthesizing motor programs (Lake et al., 2015), or learning relational theories (Katz et al., 2008). These applications have some component of transfer learning, and building efficient program learners that can transfer inductive biases across tasks is a prime target for future research.

References

Cryptominisat. http://www.msoos.org/documentation/cryptominisat/.

Chakraborty, Supratik, Fremont, Daniel J, Meel, Kuldeep S, Seshia, Sanjit A, and Vardi, Moshe Y. Distribution-aware sampling and weighted model counting for sat. *arXiv preprint arXiv:1404.2984*, 2014a.

Chakraborty, Supratik, Meel, Kuldeep S, and Vardi, Moshe Y. Balancing scalability and uniformity in sat witness generator. In *Proceedings of the 51st Annual Design Automation Conference*, pp. 1–6. ACM, 2014b.

Dechter, Eyal, Malmaud, Jon, Adams, Ryan P., and Tenenbaum, Joshua B. Bootstrap learning via modular concept discovery. In *IJCAI*, pp. 1302–1309. AAAI Press, 2013. ISBN 978-1-57735-633-2. URL http://dl.acm.org/citation.cfm?id=2540128.2540316.

Ellis, Kevin, Solar-Lezama, Armando, and Tenenbaum, Josh. Unsupervised learning by program synthesis. In *Advances in Neural Information Processing Systems*, pp. 973–981, 2015.

Ermon, Stefano, Gomes, Carla P, and Selman, Bart. Uniform solution sampling using a constraint solver as an oracle. *UAI*, 2012.

Ermon, Stefano, Gomes, Carla P, Sabharwal, Ashish, and Selman, Bart. Embed and project: Discrete sampling with universal hashing. In *Advances in Neural Information Processing Systems*, pp. 2085–2093, 2013.

Gomes, Carla P, Sabharwal, Ashish, and Selman, Bart. Model counting: A new strategy for obtaining good bounds. 2006a.

Gomes, Carla P, Sabharwal, Ashish, and Selman, Bart. Near-uniform sampling of combinatorial spaces using xor constraints. In *Advances In Neural Information Processing Systems*, pp. 481–488, 2006b.

Graves, Alex, Wayne, Greg, and Danihelka, Ivo. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

Gulwani, Sumit. Automating string processing in spreadsheets using input-output examples. In *POPL*, pp. 317–330, New York, NY, USA, 2011a. ACM. ISBN 978-1-4503-0490-0. doi: 10.1145/1926385.1926423. URL http://doi.acm.org/10.1145/1926385.1926423.

Gulwani, Sumit. Automating string processing in spreadsheets using input-output examples. In *ACM SIGPLAN Notices*, volume 46, pp. 317–330. ACM, 2011b.

Gulwani, Sumit, Jha, Susmit, Tiwari, Ashish, and Venkatesan, Ramarathnam. Synthesis of loop-free programs. In *PLDI*, pp. 62–73, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0663-8. doi: 10.1145/1993498.1993506. URL http://doi.acm.org/10.1145/1993498.1993506.

Gulwani, Sumit, Hernandez-Orallo, Jose, Kitzelmann, Emanuel, Muggleton, Stephen, Schmid, Ute, and Zorn, Ben. Inductive programming meets the real world. *Commun. ACM*, 2015.

Katz, Yarden, Goodman, Noah D., Kersting, Kristian, Kemp, Charles, and Tenenbaum, Joshua B. Modeling semantic cognition as logical dimensionality reduction. In *CogSci*, pp. 71–76, 2008.

Koza, John R. *Genetic programming - on the programming of computers by means of natural selection*. Complex adaptive systems. MIT Press, 1993. ISBN 978-0-262-11170-6.

Lake, Brenden M, Salakhutdinov, Ruslan, and Tenenbaum, Joshua B. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, 2015.

Liang, P., Jordan, M. I., and Klein, D. Learning dependency-based compositional semantics. In *Association for Computational Linguistics (ACL)*, pp. 590–599, 2011.

Liang, Percy, Jordan, Michael I., and Klein, Dan. Learning programs: A hierarchical bayesian approach. In Fürnkranz, Johannes and Joachims, Thorsten (eds.), *ICML*, pp. 639–646. Omnipress, 2010. ISBN 978-1-60558-907-7.

Lin, Dianhuan, Dechter, Eyal, Ellis, Kevin, Tenenbaum, Joshua B., and Muggleton, Stephen. Bias reformulation for one-shot function induction. In *ECAI 2014 - 21st European Conference on Artificial Intelligence, 18-22 August 2014, Prague, Czech Republic - Including Prestigious Applications of Intelligent Systems (PAIS 2014)*, pp. 525–530, 2014. doi: 10.3233/978-1-61499-419-0-525. URL http://dx.doi.org/10.3233/978-1-61499-419-0-525.

Menon, Aditya, Tamuz, Omer, Gulwani, Sumit, Lampson, Butler, and Kalai, Adam. A machine learning framework for programming by example. In *Proceedings of The 30th International Conference on Machine Learning*, pp. 187–195, 2013.

Nori, Aditya V, Ozair, Sherjil, Rajamani, Sriram K, and Vijaykeerthy, Deepak. Efficient synthesis of probabilistic programs. In *Proceedings of the 36th ACM SIGPLAN*

Conference on Programming Language Design and Implementation, pp. 208–217. ACM, 2015.

Raychev, Veselin, Bielik, Pavol, Vechev, Martin, and Krause, Andreas. Learning programs from noisy data. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 761–774. ACM, 2016.

Reed, Scott and de Freitas, Nando. Neural programmer-interpreters. *CoRR*, abs/1511.06279, 2015. URL `http://arxiv.org/abs/1511.06279`.

Schkufza, Eric, Sharma, Rahul, and Aiken, Alex. Stochastic superoptimization. In *ACM SIGARCH Computer Architecture News*, volume 41, pp. 305–316. ACM, 2013.

Singh, Rishabh, Gulwani, Sumit, and Solar-Lezama, Armando. Automated feedback generation for introductory programming assignments. In *ACM SIGPLAN Notices*, volume 48, pp. 15–26. ACM, 2013.

Solar Lezama, Armando. *Program Synthesis By Sketching*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2008. URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-177.html`.

Solar-Lezama, Armando, Tancau, Liviu, Bodik, Rastislav, Seshia, Sanjit, and Saraswat, Vijay. Combinatorial sketching for finite programs. In *ACM Sigplan Notices*, volume 41, pp. 404–415. ACM, 2006.

Valiant, Leslie G and Vazirani, Vijay V. Np is as easy as detecting unique solutions. In *Proceedings of the seventeenth annual ACM symposium on Theory of computing*, pp. 458–463. ACM, 1985.