Sampling for Bayesian Program Learning

Kevin Ellis, Armando Solar-Lezama, and Joshua B. Tenenbaum
Massachusetts Institute of Technology

Problem statement

Bayesian Program Learning: Learn programs from input/output examples, framed as Bayesian inference. Given a description-length prior over programs, condition on examples. Approximate the posterior by a set of samples.

- **Efficient in practice**: synthesizes non-trivial programs in minutes
- **Theoretical guarantees**: our algorithm, ProgramSample, generates iid samples from a distribution that provably approximates the true posterior

Sampling text edit programs:

Input	Output
1/21/2001	01

A sampled program	English interpretation
`substr(pos('0',-1),-1)`	"last 0 til end"
`const('01')`	"output 01"
`substr(-2,-1)`	"take last two"

Two Key Ingredients

How can we get practical performance and theoretical guarantees for a problem that feels so obviously intractable?

- **Sketching:** Constrain the program structure by a *sketch*, like a recursive grammar over expressions. Consider *finite* programs (bounded size/runtime), modeled in a SAT solver, which does the heavy lifting of searching for programs.
- **Sampling via random XOR constraints:** Sample SAT solutions (programs) by adding random XOR constraints to the SAT formula, an idea first introduced in Gomes et al. 2006.

Bayesian Program Learning & Other Approaches

- Assumes **strong prior knowledge**, allowing learning from **one or a few examples**; large data sets possible but difficult in practice. Contrast with eg Neural Turing Machines
- Explicitly models **uncertainty**, contrast with eg program synthesis
- **Theoretical guarantees**, like much program synthesis work, contrast with eg genetic programming

Example: Learning to Sort

Examples	MDL	Sampled correct?	Posterior is...
(7 4 3)→(3 4 7)	reverse	✗	diffuse
(7 4 3)→(3 4 7) (5 2 3)→(2 3 5)	buggy code	✓	diffuse
(7 4 3)→(3 4 7) (5 2 3)→(2 3 5) (1 6 4)→(1 4 6)	sort	✓	peaky
(3 2 4 1)→(1 2 3 4)	count up to list length	✗	peaky
(3 2 4 1)→(1 2 3 4) (1 6 2 0)→(0 1 2 6)	sort	✓	peaky

Background: Program Synthesis by SAT Solving

```
Program ::= i
| nand(Program,Program)
```

(a) Sketch

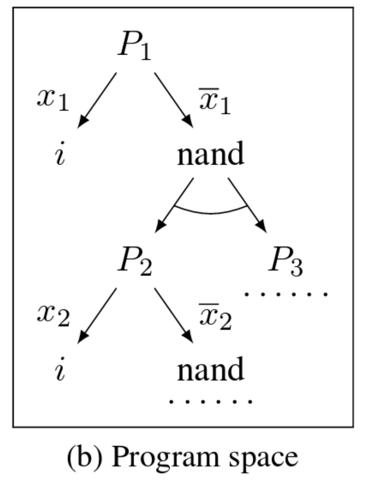
```
Program(i = 0) = 1
```

(d) Specification

$x_1 = 0,\ x_2 = 1,\ x_3 = 1$

```
Program = nand(i, i)
```

(e) A constraint solution; $|x| = 3$ bits



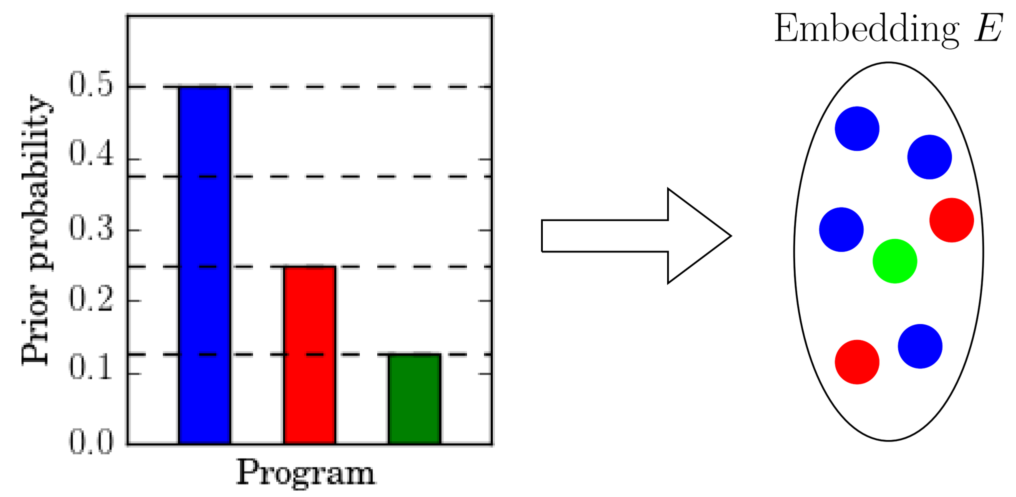
(b) Program space

$i \Leftrightarrow 0 \wedge P_1 \Leftrightarrow 1$
$x_1 \Rightarrow (P_1 \Leftrightarrow i)$
$\overline{x}_1 \Rightarrow (P_1 \Leftrightarrow \overline{P_2 \wedge P_3})$
$x_2 \Rightarrow (P_2 \Leftrightarrow i)$
$\overline{x}_2 \Rightarrow (P_2 \Leftrightarrow \overline{P_4 \wedge P_5})$
$x_3 \Rightarrow (P_3 \Leftrightarrow i)$
......

(c) Constraints for SAT solver

Figure 2: Synthesizing a program via sketching and constraint solving. `Typewriter font` refers to pieces of programs or sketches, while math font refers to pieces of a constraint satisfaction problem. The variable `i` is the program input.

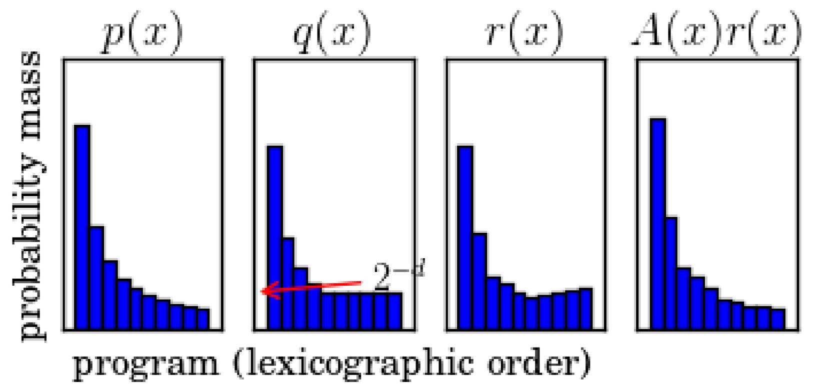
Sampling by Random XOR Constraints



Sample approximately uniformly from E by fixing the parity of random sums (XOR's) of SAT variables. Main idea in Ermon et al. 2013.

Definition: Tilt. The *tilt* of $p(\cdot)$ is $\frac{\max_x p(x)}{\min_x p(x)}$.

A problem. Very inefficient if the posterior $p(x)$ is highly tilted. ProgramSample approximates $p(x)$ by a low tilt distribution $q(x)$. Rejection sampling corrects this distortion: $A(x)$ is acceptance probability.



Summary: Approximate $p(x)$ by $q(x)$, embed in E, approximately uniform samples of E give $r(x)$, distribution of samples is $\propto A(x)r(x)$

Theoretical guarantees

ProgramSample takes two real-valued parameters, γ and Δ.

Proposition 1: Distribution of samples is close in KL to the true posterior. Write $Ar(x)$ to mean the distribution of the samples. Then $\mathrm{KL}(p||Ar) < \log\left(1 + \frac{1+2^{-\gamma}}{1+2^{\Delta}}\right)$.

Proposition 2: Not too much work is needed to get a sample. The expected number of calls to the solver per sample is bounded above by $\frac{1+2^{\Delta}}{(1+2^{-\gamma})^{-1}(1+2^{-\Delta})^{-1}-2^{-\Delta}}$.

Domain: Learning text edit scripts

Programming By Example: Learn a string editing program from a few examples. Systems like this ship in Microsoft Excel.

Examples — **A program**

Input	Output
don steve g.	dsg
Kevin Jason Mat	KJM
Jose Larry S	JLS
Arthur Joe Juan	AJJ
Raymond F. Timothy	RFT

```
SubStr(0,1)+
SubStr(Pos(' ',",0),Pos(' ',",0))+
SubStr(Pos(' ',",-1),Pos(' ',",1))
```

Input	Output
12/31/13	12.31
1/23/2009	1.23
4/12/2023	4.12
6/23/15	6.23
7/15/2015	7.15

```
SubStr(0,Pos(",'/',0))+
Constant('.')+
SubStr(Pos('/',",0),Pos(",'/',-1))
```

Key problem: PBE systems target end-users who are unwilling to provide more than one or a few examples, leaving the intended behavior highly ambiguous. Model this ambiguity using sampling.

Sketching a Program Space

```
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 | ...
```

Learning from very few examples

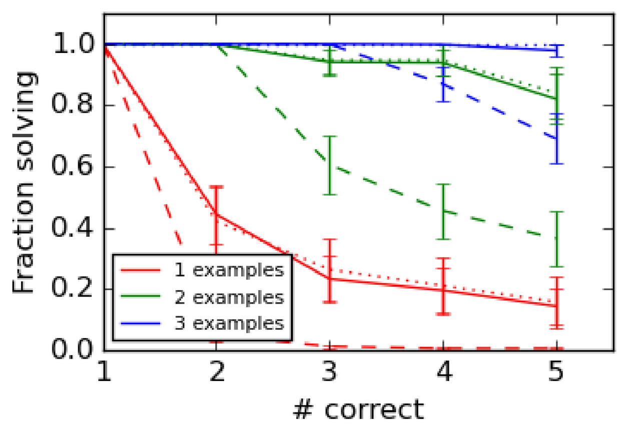


Figure 1: Results averaged across 19 problems. Solid: ProgramSample . Dashed: enumerating 100 programs. Dotted: MDL.

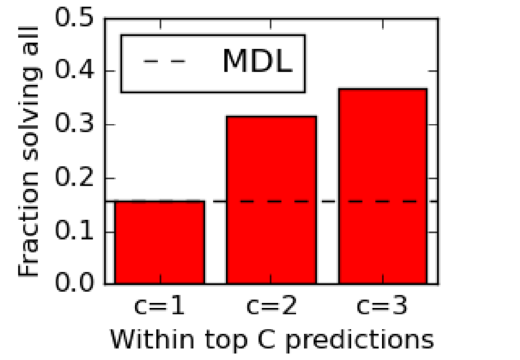


Figure 2: MDL learner vs ProgramSample on one-shot learning. Predictions marginalize out the program.

Good performance needs tilt correction. w/o low-tilt approximation, get no samples for any of these problems after an hour, vs $\approx$ 1 minute w/ ProgramSample

Domain: List Manipulation

Computer Aided Programming: Synthesize source code consistent with a specification. We tackle these problems:

- **Sort**: synthesize quicksort
- **Reverse**: recursively reverse a list
- **Count**: count occurrences of list head in list tail

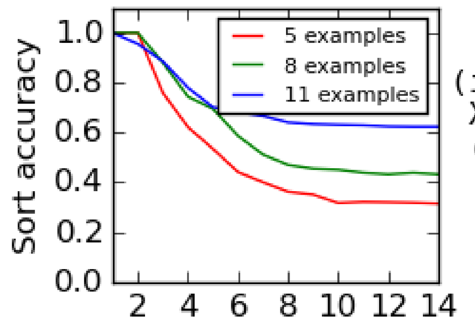
Multiple implementations might satisfy the spec, especially if the specs are examples. Model the ambiguity by sampling plausible implementations.

Sketching a Program Space

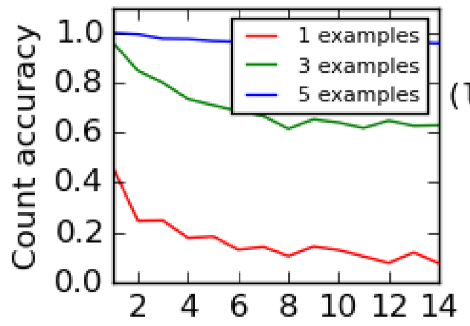
```
Program ::=
  (if Bool List
    (append RecursiveList
            RecursiveList
            RecursiveList))
Bool ::= (<= Int) | (>= Int)
Int  ::= 0 | (1+ Int) | (1- Int)
       | (length List) | (head List)
List ::= nil | (filter Bool List)
       | X | (tail List) | (list Int)
RecursiveList ::= List
               | (recurse List)
```

Learner Generalizes to Longer Sequences

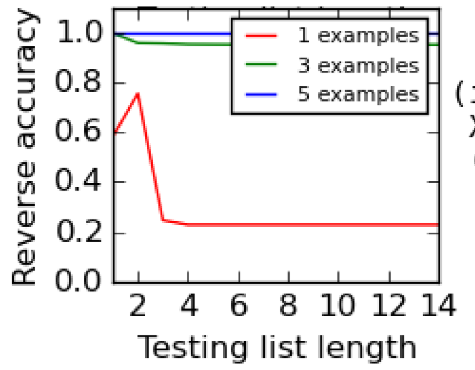
Trained to sort, reverse, or count on lists of length ≤ 3; tested on lists of length ≤ 14.




```
(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))))
```

Posterior sample

Sampling aids generalization. Even if the most likely program has a bug, the posterior puts enough mass on correct programs that the ensemble of samples acts as a probabilistic algorithm with a high probability of succeeding.