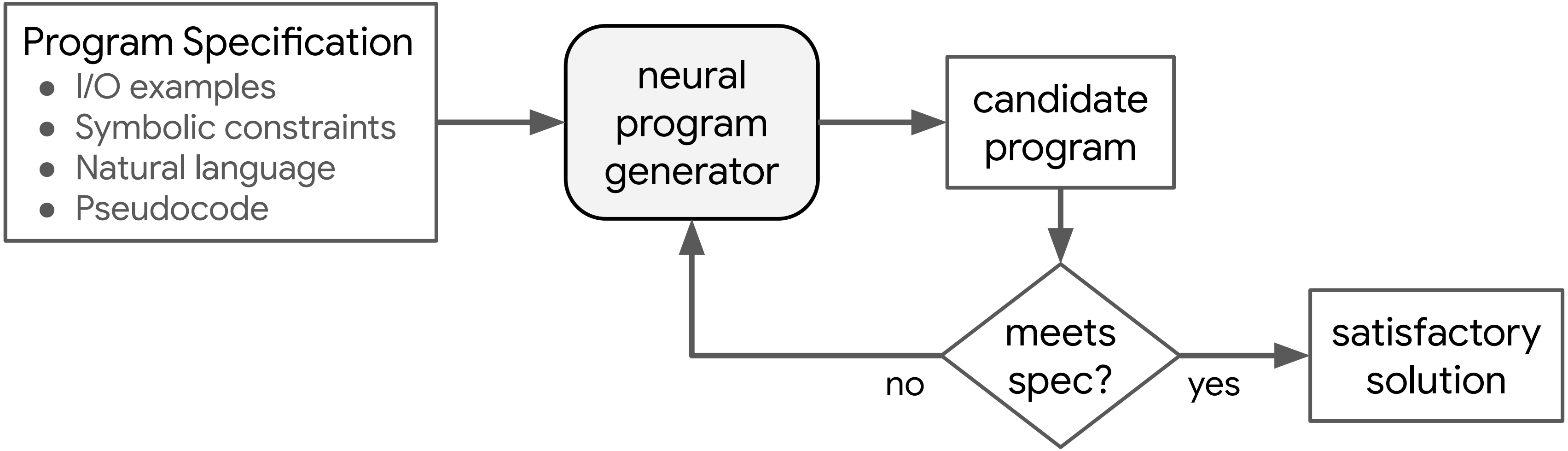
Incremental Sampling Without Replacement for Sequence Models

Kensen Shi, David Bieber, Charles Sutton
(Google Research)

Example Motivation

Program synthesis: generate a program that satisfies a given specification



Sample candidate programs from the neural generator conditioned on the spec

- Incrementally: stopping as soon as a satisfactory program is found
- Without replacement: duplicate candidate programs are not useful

Motivation, More Generally

Neural search in a discrete output space for a solution that satisfies constraints

Sample candidate solutions from the neural generator conditioned on the spec

- Incrementally: stopping as soon as a satisfactory solution is found
- Without replacement: duplicate candidate solutions are not useful

Examples of search problems:

- Program synthesis
- Traveling Salesman Problem: find a tour with cost at most X
- Other combinatorial optimization problems
- SAT and SMT: find assignments to variables to satisfy all constraints

Benefits of Incremental Sampling

Incremental sampling enables more flexibility in stopping conditions.

With incremental sampling, one can draw distinct samples until...

- ... a satisfactory solution is found
- ... a time limit has passed
- ... enough variety is obtained
- ... an estimate has converged
- ... a target fraction of the search space is explored
- ... any arbitrary stopping criterion is met

Contrast with beam search...

Existing methods of drawing samples

Beam search and variants

- Produces a batch of distinct outputs
- Not incremental
  - One does not know upfront how large a batch should be
  - If one batch is insufficient, the next batch may have duplicates

Naive Monte Carlo I.I.D. sampling

- This is sampling with replacement since samples are independent

Rejection sampling

- Like Monte Carlo I.I.D. sampling, but duplicate samples are discarded
- Potentially inefficient if the output distribution is very peaked, as one would expect from a well trained neural model

Our Contributions

- Approaching the sampling problem by manipulating the random choices made by the program that generates the samples
- UniqueRandomizer, a data structure for sampling distinct outputs of a randomized program
  - Incremental
  - Samples without replacement
  - Time and memory efficient
  - Can be extended to support batching
- Describing *discrete randomized programs*, the broad class of programs that UniqueRandomizer can sample from
- A statistical estimator that applies to samples drawn without replacement
  - See paper for details

What can we sample from?

Discrete randomized programs:

- All randomness comes from a *choice function* that chooses a random index given a discrete probability distribution
- Cannot draw random floats
  - But, `Uniform(0, 1) < 0.3` can be written as `choice_fn([0.3, 0.7]) == 0`
- Can accept inputs, e.g., a trained model and problem instance
- Can use control flow including conditionals, loops, and recursion
- This broad class of programs includes sequence models!

```
def draw_sample(model, h,
                choice_fn):
  tokens = []
  token = BOS
  for i in range(MAX_LEN):
    probs, h = model(token, h)
    token = choice_fn(probs)
    tokens.append(token)
    if token == EOS:
      break
  return tokens
```

A simple randomized program that draws a sample from a recurrent sequence model. It uses `choice_fn` to make random decisions.

UniqueRandomizer: Overview

UniqueRandomizer is our solution to incremental sampling without replacement

- Maintains a trie of unsampled probability masses corresponding to states in the randomized program

Provides 3 functions:

- Initialization: creates the data structure
- `choice_fn`: provides choices while accounting for previous samples
- `process_termination`: updates the trie to reflect the most recent sample

```
def sample_wor(draw_sample,
               model, h, k):
  samples = []
  ur = UniqueRandomizer()
  for i in range(k):
    s = draw_sample(model, h,
                    ur.choice_fn)
    samples.append(s)
    ur.process_termination()
  return samples
```

Using UniqueRandomizer to draw samples without replacement from the draw_sample function.

UniqueRandomizer: Algorithm Summary

Trie structure:

- Each node represents a state of the randomized program, between random choices.
- Each node stores the *unsampled probability mass* at that state.
- Each edge represents one possible result of one random choice.

While sampling, maintain a current node that walks down the trie as random choices are made.

- In `choice_fn`, use the probability distribution induced by the current node's children to choose a random index to return. Update the current node to the corresponding child.
- In `process_termination`, subtract the current node's probability mass from all of its ancestors. Reset the current node back to the trie root.

UniqueRandomizer: Example

```
def draw_sample(choice_fn):
  sequence = []
  length = choice_fn([0.5, 0.4, 0.1])
  for i in range(length):
    sequence.append(choice_fn([0.75, 0.25]))
  return sequence
```

A randomized program that produces binary sequences of length 0 to 2.

Note: probability distributions are hardcoded for the sake of example, but in practice they could be computed by a model.

UniqueRandomizer: Example

```
def draw_sample(choice_fn):
  sequence = []
  length = choice_fn([0.5, 0.4, 0.1])
  for i in range(length):
    sequence.append(choice_fn([0.75, 0.25]))
  return sequence
```

A randomized program that produces binary sequences of length 0 to 2.

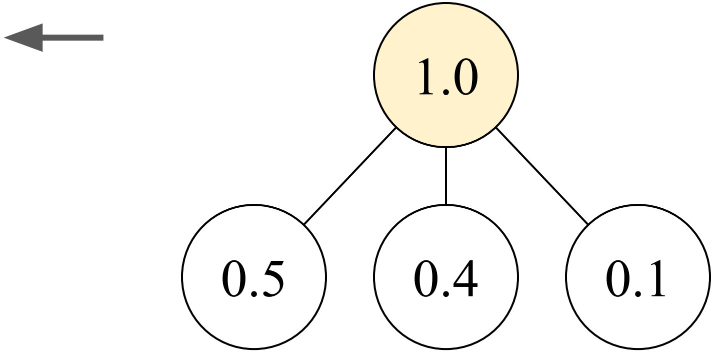
```
sequence: []
length: ?
i: ?
```

UniqueRandomizer: Example

```
def draw_sample(choice_fn):
  sequence = []
  length = choice_fn([0.5, 0.4, 0.1])
  for i in range(length):
    sequence.append(choice_fn([0.75, 0.25]))
  return sequence
```

A randomized program that produces binary sequences of length 0 to 2.

```
sequence: []
length: ?
i: ?
```

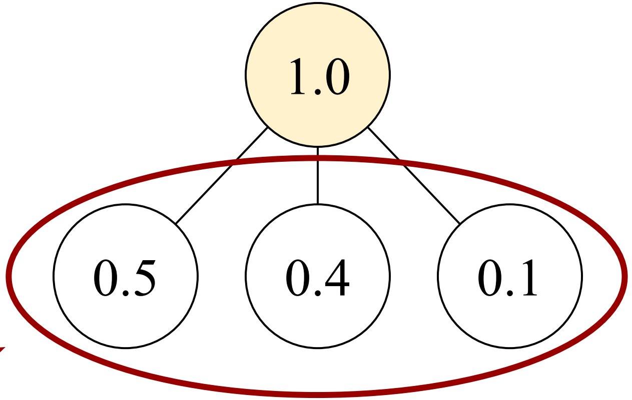


UniqueRandomizer: Example

```
def draw_sample(choice_fn):
  sequence = []
  length = choice_fn([0.5, 0.4, 0.1])
  for i in range(length):
    sequence.append(choice_fn([0.75, 0.25]))
  return sequence
```

A randomized program that produces binary sequences of length 0 to 2.

```
sequence: []
length: ?
i: ?
```



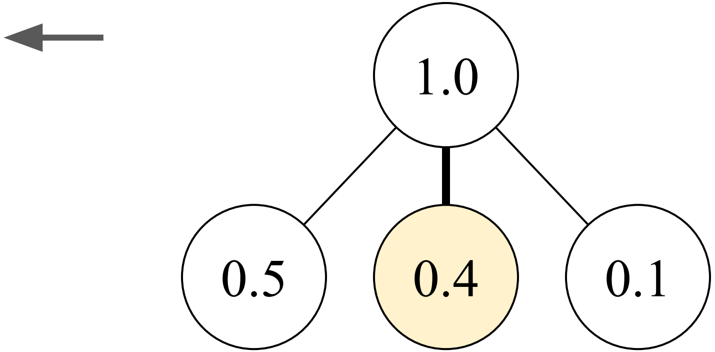
Choose `length` using the distribution [0.5, 0.4, 0.1]. Suppose we choose `length = 1` (with probability 0.4).

UniqueRandomizer: Example

```
def draw_sample(choice_fn):
  sequence = []
  length = choice_fn([0.5, 0.4, 0.1])
  for i in range(length):
    sequence.append(choice_fn([0.75, 0.25]))
  return sequence
```

A randomized program that produces binary sequences of length 0 to 2.

```
sequence: []
length: 1
i: ?
```

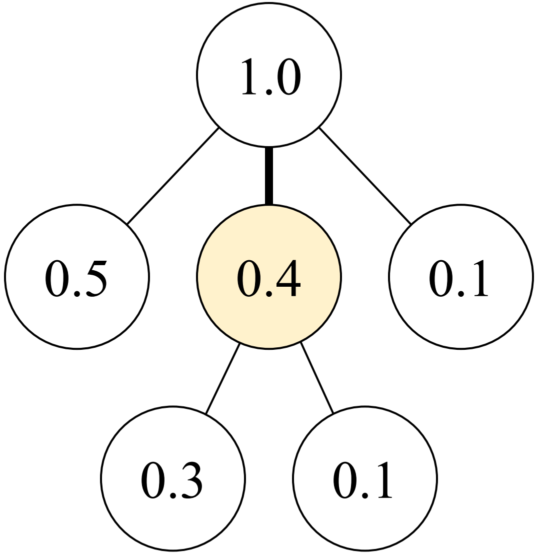


UniqueRandomizer: Example

```
def draw_sample(choice_fn):
  sequence = []
  length = choice_fn([0.5, 0.4, 0.1])
  for i in range(length):
    sequence.append(choice_fn([0.75, 0.25]))
  return sequence
```

A randomized program that produces binary sequences of length 0 to 2.

```
sequence: []
length: 1
i: 0
```

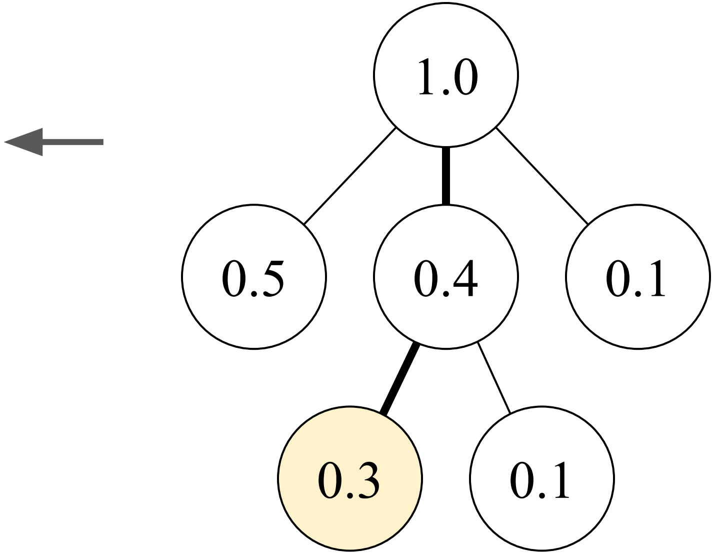


UniqueRandomizer: Example

```
def draw_sample(choice_fn):
  sequence = []
  length = choice_fn([0.5, 0.4, 0.1])
  for i in range(length):
    sequence.append(choice_fn([0.75, 0.25]))
  return sequence
```

A randomized program that produces binary sequences of length 0 to 2.

```
sequence: [0]
length: 1
i: 0
```

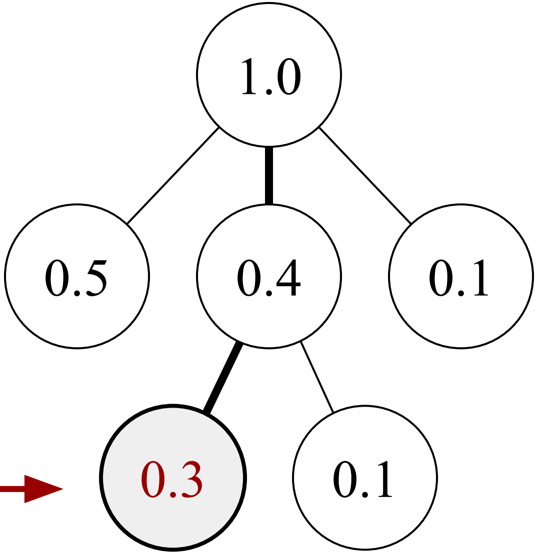


UniqueRandomizer: Example

```
def draw_sample(choice_fn):
  sequence = []
  length = choice_fn([0.5, 0.4, 0.1])
  for i in range(length):
    sequence.append(choice_fn([0.75, 0.25]))
  return sequence
```

A randomized program that produces binary sequences of length 0 to 2.

```
sequence: [0]
length: 1
i: 1
```



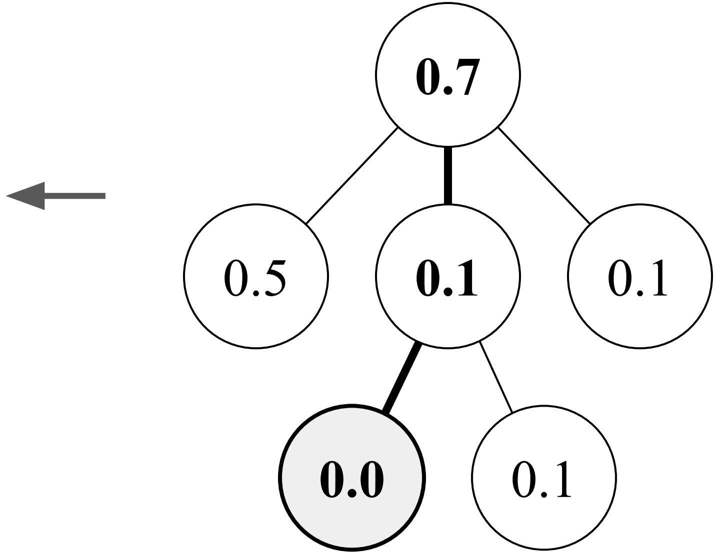
The randomized program terminated. In `process_termination`, we subtract the leaf's probability mass (0.3) from all of its ancestors, since the path has been sampled.

UniqueRandomizer: Example

```
def draw_sample(choice_fn):
  sequence = []
  length = choice_fn([0.5, 0.4, 0.1])
  for i in range(length):
    sequence.append(choice_fn([0.75, 0.25]))
  return sequence
```

A randomized program that produces binary sequences of length 0 to 2.

```
sequence: [0]
length: 1
i: 1
```

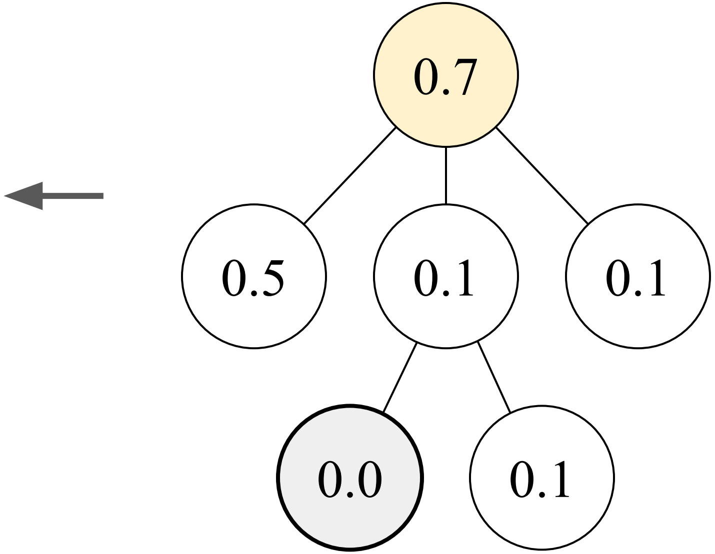


UniqueRandomizer: Example

```
def draw_sample(choice_fn):
  sequence = []
  length = choice_fn([0.5, 0.4, 0.1])
  for i in range(length):
    sequence.append(choice_fn([0.75, 0.25]))
  return sequence
```

A randomized program that produces binary sequences of length 0 to 2.

```
sequence: [0]
length: 1
i: 1
```



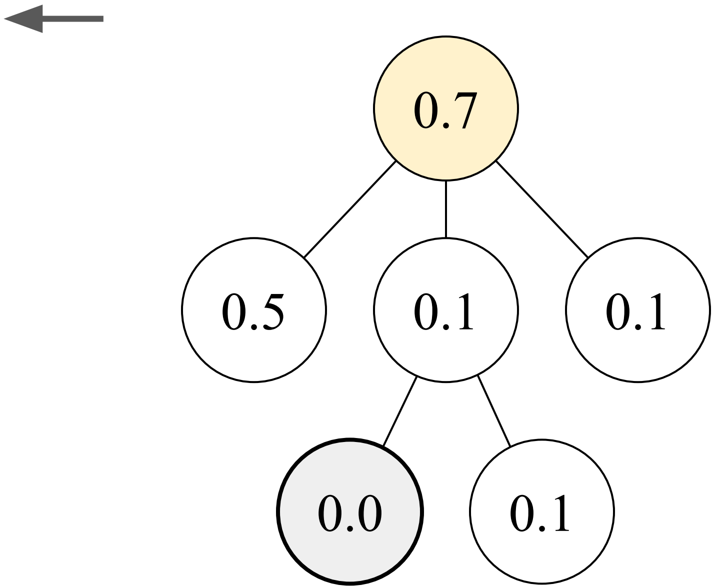
UniqueRandomizer: Example

```
def draw_sample(choice_fn):
  sequence = []
  length = choice_fn([0.5, 0.4, 0.1])
  for i in range(length):
    sequence.append(choice_fn([0.75, 0.25]))
  return sequence
```

A randomized program that produces binary sequences of length 0 to 2.

Run `draw_sample` again to draw the next sample, without replacement. The trie is preserved from the previous run.

```
sequence: []
length: ?
i: ?
```

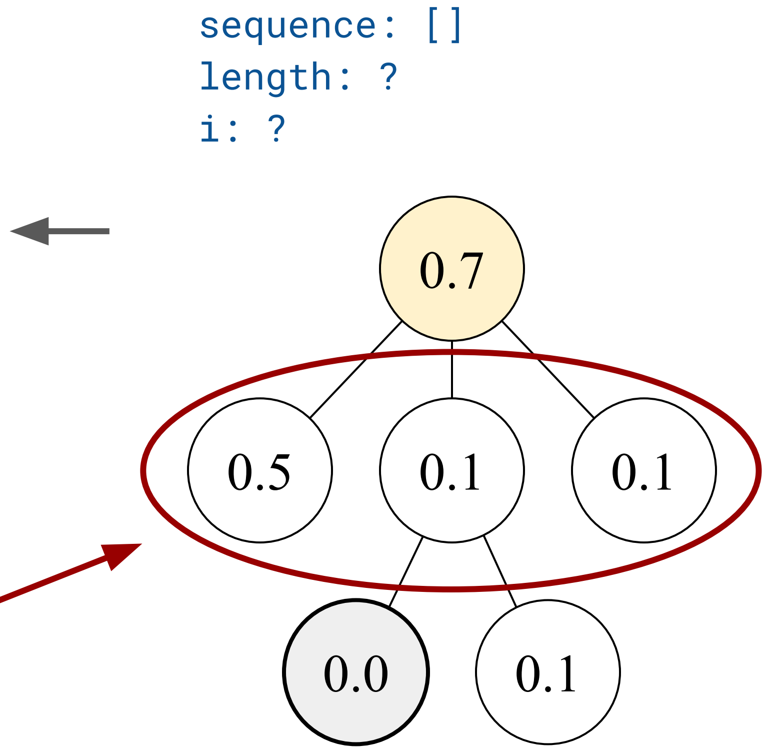


UniqueRandomizer: Example

```
def draw_sample(choice_fn):
  sequence = []
  length = choice_fn([0.5, 0.4, 0.1])
  for i in range(length):
    sequence.append(choice_fn([0.75, 0.25]))
  return sequence
```

A randomized program that produces binary sequences of length 0 to 2.

Choose `length` using the *unnormalized* distribution [0.5, 0.1, 0.1], which normalizes to approximately [0.71, 0.14, 0.14].



Unique Choices vs. Unique Outputs

UniqueRandomizer actually guarantees that there are no duplicate *sequences of random choices*. When does this lead to *unique outputs*?

Theorem (informal):

UniqueRandomizer samples unique *outputs* of a randomized program $\mathcal{P}$

if and only if

every random choice in the execution of $\mathcal{P}$ partitions the set of outputs that were possible at the time.

See the paper for a formal statement and proof.

Importantly, this condition is satisfied by sequence models!

Distribution of Samples

A randomized program $\mathcal{P}$ run on the input x induces a probability distribution over its outputs $y_i \sim P(y = \mathcal{P}(x))$.

Theorem: When using UniqueRandomizer to sample unique outputs, the outputs are drawn from the sequence of distributions

$$P_{\mathrm{WOR}}(y_i \mid y_{1:i-1}) = P(y_i = \mathcal{P}(x) \mid y_i \notin y_{1:i-1}).$$

This is the same distribution as produced by rejection sampling, without any potential inefficiency!

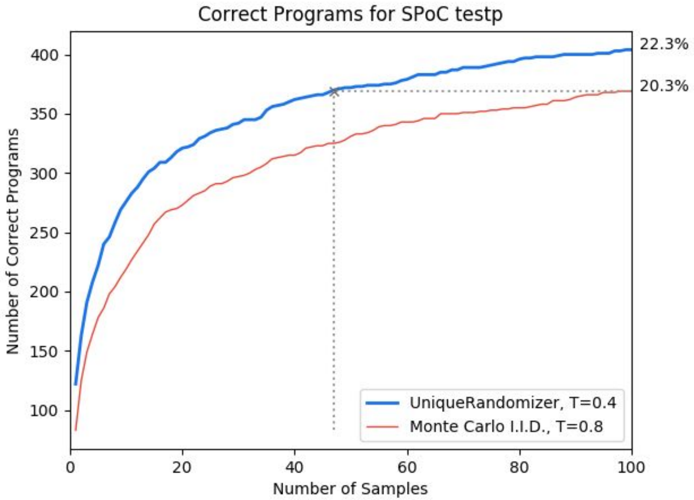
Extensions (see paper)

- Skipping probability computations when trie values will be used instead
  - Avoid expensive model computations when revisiting a trie node
- Incremental batched sampling by combining UniqueRandomizer with Stochastic Beam Search[1] to enable parallelism
  - Use SBS to sample a batch using the probability distribution in the trie, and then update the trie to prevent those samples from appearing in subsequent batches
- Detecting when all outputs have been sampled
- Locally modifying probabilities in the trie
  - Could be useful to shift the distribution in response to new data
- A novel estimator for the expectation $E_{y \sim \mathcal{P}}[f(y)]$, where $f(y)$ is an arbitrary function of the samples y drawn from the randomized program $\mathcal{P}$

[1] Wouter Kool, Herke van Hoof, and Max Welling. *Stochastic Beams and Where To Find Them: The Gumbel-Top-k Trick for Sampling Sequences Without Replacement*. ICML 2019.

Experiments: Program Synthesis

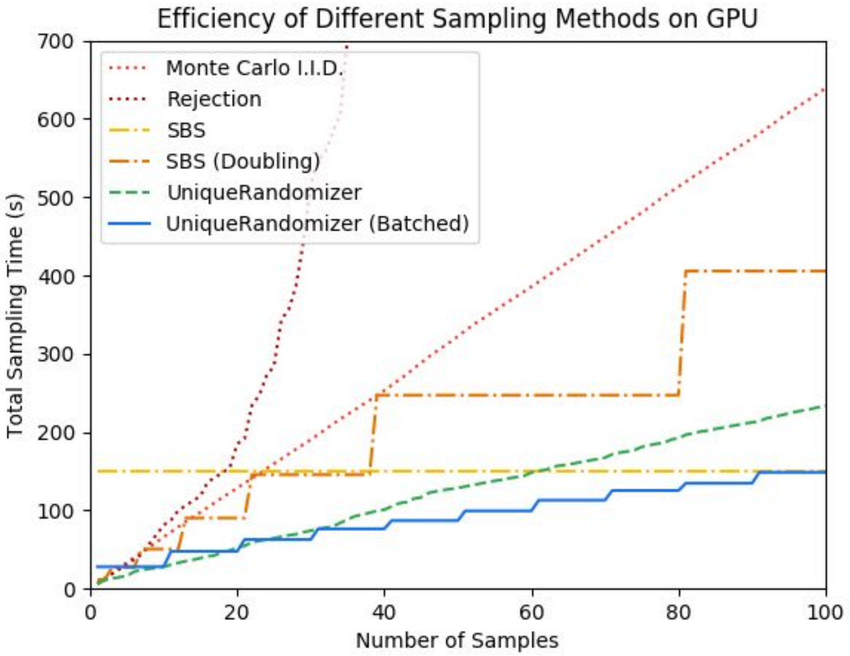
- SPoC[2] dataset: C++ programs with pseudocode and I/O test cases
- Train a Transformer to generate code given pseudocode
- UniqueRandomizer gives +2.0% success rate over I.I.D. sampling
- SPoC's use of compiler diagnostics led to +1.7% success rate



[2] Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy Liang. *SPoC: Search-based Pseudocode to Code*. NeurIPS 2019.

Experiments: Efficiency

- UniqueRandomizer is faster than naive Monte Carlo I.I.D. sampling
- Batched UniqueRandomizer is as fast as SBS for a fixed number of samples, but is incremental



Experiments: TSP Heuristic + UniqueRandomizer

- *Farthest Insertion* heuristic for TSP: maintain a cycle, iteratively choose the node that is farthest from the cycle and insert it at the cheapest location
- Relaxation: sample an insertion location i with probability $\propto \text{costDelta}(i)^{-1/\tau}$
- UniqueRandomizer applied to this heuristic outperforms 2 of 3 recent neural approaches, and is competitive with the SOTA neural approach

	$n = 20$		$n = 50$		$n = 100$	
Method	Cost	Gap	Cost	Gap	Cost	Gap
Concorde (exact)	3.8357	0%	5.696	0%	7.765	0%
Bello et al., i.i.d. sampling (*)	–		5.75	0.95%	8.00	3.03%
EAN, i.i.d. sampling (*)	3.84	0.11%	5.77	1.28%	8.75	12.70%
AM, i.i.d. sampling	3.8381	0.063%	**5.724**	**0.49%**	**7.944**	**2.31%**
Far. Ins., greedy	3.9262	2.358%	6.011	5.53%	8.354	7.59%
Far. Ins., *UniqueRandomizer*	**3.8372**	**0.038%**	5.746	0.88%	7.981	2.79%

Conclusion

- UniqueRandomizer is a novel data structure for incremental sampling without replacement from a wide class of randomized programs
- Incremental sampling offers increased flexibility in stopping criteria, in contrast to beam search where the number of samples is decided upfront
- UniqueRandomizer is efficient and supports incremental batched sampling
- Potentially useful in many domains:
  - Program synthesis
  - Combinatorial optimization
  - Constraint satisfaction problems
  - Neural approaches to search problems
  - Natural language generation
  - Rollouts in reinforcement learning
  - Randomized rounding
  - Probabilistic programming