Abstract
Sampling is a fundamental technique, and sampling without replacement is often desirable when duplicate samples are not beneficial. Within machine learning, sampling is useful for generating diverse outputs from a trained model. We present an elegant procedure for sampling without replacement from a broad class of randomized programs, including generative neural models that construct outputs sequentially. Our procedure is efficient even for exponentially-large output spaces. Unlike prior work, our approach is incremental, i.e., samples can be drawn one at a time, allowing for increased flexibility. We also present a new estimator for computing expectations from samples drawn without replacement. We show that incremental sampling without replacement is applicable to many domains, e.g., program synthesis and combinatorial optimization.

1. Introduction
Sampling from programmatically-defined distributions is a fundamental technique. Machine learning problems often involve learning distributions over structured objects such as sentences, images, audio, biological sequences, and source code. These distributions usually factorize into a product of conditional distributions, e.g., the probability of a word given the previous words. Such distributions are naturally represented as programs with sampling operations, which we call randomized programs.

To predict a structured output, traditional search methods like beam search compute a set of predictions that approximately maximize the probability according to a learned model. However, recent work has instead favored sampling from the model, with the goal of obtaining diverse and higher-quality predictions (Shi et al., 2019; Fan et al., 2018; Kool et al., 2019b; Holtzman et al., 2018; 2019; Radford et al., 2019). The same considerations suggest that sampling without replacement could be even more desirable by avoiding duplicate samples, especially in machine learning where the training objective often pushes the output distribution to become extremely peaked.

We present a data structure called UniqueRandomizer for sampling without replacement from randomized programs. The main advantage is that it is incremental, allowing for increased flexibility in stopping conditions for the sampling procedure. For instance, one can draw more samples until a solution is found, or until a sample-based estimate has converged, or enough sample diversity is obtained, and so on. A closely related algorithm for sampling without replacement is Stochastic Beam Search (SBS) (Kool et al., 2019b), based on the Gumbel-top-k trick. SBS is mathematically more complex than UniqueRandomizer and does not readily support incrementality, but it has advantages in parallelism and in providing an importance-sampling method for statistical estimation. Fortunately, we are able to present a combined method with both advantages, i.e., the flexibility of incremental sampling and efficiency of batched computations. We also derive an improved estimator that can be used with any method of sampling without replacement.

We experimentally demonstrate that using UniqueRandomizer leads to higher-quality samples in program synthesis and combinatorial optimization. More broadly, a major contribution of this paper is making the case that UniqueRandomizer is a general technique that can be applied to many scenarios.

2. Approach
As a motivating example, consider a program synthesis task: given pseudocode for a short program and some input/output examples, generate the corresponding source code. Suppose that we train a neural sequence-to-sequence model that takes pseudocode as input and generates candidate programs. We would like to sample from the model until we find a program that satisfies the examples.

We do not know upfront how many samples are needed to find a solution, so to minimize computational waste, we want to sample from the model incrementally, one sample at a time until a solution is found. Furthermore, we have no use for duplicate samples because the quality of a sampled
def \( \mathcal{P}(\{h, W_0, W_1\}, C) \):
    tokens = []
    for i in range(0, 100):
        h = softmax(matmul(W0, h))
        probs = softmax(matmul(W1, h))
        tokens.append(C(probs))
    return tokens

Figure 1: A simple example of a randomized program. This program samples from a recurrent neural network with weights \( W_0 \) and \( W_1 \) and initial state \( h \). The random choice operator \( C \) is used to sample tokens from the output distribution \( \text{probs} \) at each step of the RNN.

A program is deterministic. Since we want to trust the trained model’s decisions, we may apply a low temperature when sampling, but this in turn increases the chance of sampling duplicate programs. If we sample without replacement, then we can obtain higher-quality samples while avoiding duplicates altogether. Rejection sampling is a standard approach for this, but can be inefficient for highly skewed distributions, such as what we expect to obtain from a trained model with low sampling temperature. Our approach solves these issues with an augmented trie, described in Section 2.3.

2.1. Problem Formalization

In this section, we formalize the sampling problem that we consider. To unify the broad class of distributions that our method applies to, it is convenient to represent the distribution of interest as a program. Specifically, suppose we have a program \( \mathcal{P} \) that defines a function mapping objects of an arbitrary type \( X \) to those of type \( Y \). We further endow \( \mathcal{P} \) with a second argument that acts as a source of randomness—a random choice operation \( C(\pi) \), which samples from the discrete\(^1\) probability distribution \( \pi \), returning a choice \( c \in \{0, \ldots, \text{len}(\pi) - 1\} \) with probability \( \pi_c \), like \texttt{numpy.random.choice} in Python. Except for choices produced by \( C \), the program \( \mathcal{P} \) is deterministic. Note that \( \mathcal{P} \) may take other input arguments, and it can use control flow including loops, conditionals, and recursion.

We call any program that has this form a discrete randomized program. This class of programs does not seem to have a standard name in the literature, but it is quite broad, including probabilistic grammars, neural sequence models, and graphical models. Randomized programs are essentially the subset of probabilistic programs (van de Meent et al., 2018) without a conditioning operator. An example of a discrete randomized program is shown in Figure 1, which samples from a recurrent neural network. Of course, randomized programs can be more complex than this—for example, the length of the output can be random, such as for programs that sample from a probabilistic context free grammar.

A call to a randomized program defines a distribution \( P(y = \mathcal{P}(x, C)) \) over outputs \( y \). We assume that the function call \( \mathcal{P}(x, C) \) terminates with probability 1, although in general this can be tricky to ensure (Booth and Thompson, 1973). Our goal is to obtain samples \( y_1, \ldots, y_N \) from the distribution \( P(y) \) incrementally, and without replacement (WOR). Sampling without replacement can be formalized as sampling from a sequence of modified distributions

\[
P_{\text{WOR}}(y_i \mid y_{1:i-1}) = \frac{P(y_i = \mathcal{P}(x, C) \mid y_i \notin y_{1:i-1})}{\sum_{y_i} P(y_i = \mathcal{P}(x, C) \mid y_i \notin y_{1:i-1})}. \tag{1}
\]

By sampling incrementally, we mean that samples \( y_i \) are drawn one by one with a minimal amount of computation performed for each sample. Given previous samples \( y_1, \ldots, y_N \) drawn WOR from \( \mathcal{P} \), we may easily obtain a new sample \( y_{N+1} \) drawn from \( P_{\text{WOR}}(y_{N+1} \mid y_1:N) \), without slowing down the sampler as \( N \) increases (as is possible in rejection sampling).

2.2. Sampling WOR with UniqueRandomizer

Our method is able to sample without replacement, even without modifying the program \( \mathcal{P} \). We introduce a data structure called UniqueRandomizer that defines a drop-in replacement for the random choice operator \( C \), which efficiently keeps track of the samples made so far to prevent duplicates. Here, we give an overview of the method, while in the next section we describe the specifics of the UniqueRandomizer data structure and its random choice operator.

To do this, we define two additional concepts. An execution of \( \mathcal{P} \) produces a sequence of calls to \( C \). Each such call takes as input a probability distribution \( \pi \), and outputs a random choice \( c_i \in \{0, \ldots, \text{len}(\pi) - 1\} \). We define a trace as the sequence of all random choices \( t = [c_1, \ldots, c_h] \) produced during a complete execution of \( \mathcal{P} \). Note that \( \mathcal{P} \) defines a distribution over its traces

\[
P(t) = \prod_{i=1}^{h} P(c_i \mid c_1, \ldots, c_{i-1}) = \prod_{i=1}^{h} \pi_{c_i}. \tag{2}
\]

UniqueRandomizer samples traces of \( \mathcal{P} \), incrementally and without replacement, according to \( P(t) \).

Sampling WOR for traces yields a sample WOR of program outputs under a particular condition. We say that \( \mathcal{P} \) is trace-injective if \( \mathcal{P} \) necessarily produces different outputs under different traces. This usually occurs when every call to \( C \) produces a part of the output, such as when sampling from a sequence model. This seems to be the most common situation in machine learning, e.g., the RNN example in Figure 1 is trace-injective. If \( \mathcal{P} \) is trace-injective, then sampling

---

A discrete randomized program \( \mathcal{P} \) cannot draw a random floating-point number, such as \( \text{Uniform}(0, 1) \), because this is not a discrete probability distribution. However, the boolean expression \( \text{Uniform}(0, 1) < 0.3 \) can be rewritten as \( C([0.3, 0.7]) = 0 \), which is allowed in \( \mathcal{P} \).
Algorithm 1 Using UniqueRandomizer to sample outputs of \( P \) without replacement.

1: procedure SAMPLEWOR(\( P, x, k \))
2: \quad samples \leftarrow []
3: \quad INITIALIZE()
4: \quad for \( i \in \{1, 2, \ldots, k\} \) do
5: \quad \quad \quad \quad y \leftarrow P(x, RANDOMCHOICE)
6: \quad \quad \quad \quad samples.append(y)
7: \quad \quad \quad \quad PROCESSTERMINATION()
8: \quad return samples

Traces WOR is equivalent to sampling outputs WOR (which follows from the change-of-variable rules for discrete distributions), so UniqueRandomizer will produce a sample of outputs without replacement.

Trace-injectivity can be characterized more precisely. First, we define a mapping between traces and program outputs. Every execution of \( P \) produces a trace \( t \) and an output \( y \); let \( f(t) = y \) where \( y \) is the output when \( P \) executes with trace \( t \). Trace-injectivity means that the map \( f \) is injective. We can extend this map to trace prefixes by defining \( F(t') = \{ f(t) | t' \text{ is a prefix of } t \} \). Then the following theorem says that \( P \) is trace-injective when every choice contributes to the final output, in a certain sense (proof in Appendix A):

**Theorem 1.** \( P \) is trace-injective \( \iff \) for all trace prefixes \( t' = [c_1, \ldots, c_h] \), the set of all possible outputs \( F(t') \) is partitioned by the next choice \( c_{h+1} \sim C(\pi_{h+1}) \), i.e., the set \( \{ F([c_1, \ldots, c_h, c_{h+1}]) \ | \ c_{h+1} \in \{0, \ldots, \text{len}(\pi_{h+1}) - 1\} \} \) is a partition of \( F(t') \).

UniqueRandomizer can be used to sample without replacement from a trace-injective discrete randomized program \( P \), as shown in Algorithm 1. To obtain \( k \) samples, we simply run \( P \) for \( k \) iterations, providing it with UniqueRandomizer’s RANDOMCHOICE function that remembers the sequence of choices that are made by each invocation of \( P \), and prevents duplicate traces from being generated. The way in which we do this, as well as the implementation of the functions INITIALIZE, RANDOMCHOICE, and PROCESSTERMINATION, are described in the next section.

2.3. The UniqueRandomizer Data Structure

Our main idea is to maintain an augmented trie data structure, which we call the UniqueRandomizer, containing the traces that have been generated while executing the program \( P \) multiple times. Nodes in the trie correspond to trace prefixes seen so far. Each edge represents one element of a trace, i.e., a possible outcome for the next call to \( C \). A trie node is a leaf if its trace prefix is actually a full trace, i.e., \( P \) terminates without further calls to \( C \). After \( P \) terminates, the trie is updated accordingly. Figure 2 shows an example.

Algorithm 2 Random choice operation and trie construction for UniqueRandomizer.

1: procedure INITIALIZE()
2: \quad root \leftarrow TRIENODE(parent = \emptyset, mass = 1)
3: \quad cur \leftarrow root
4: procedure RANDOMCHOICE(\( \pi \))
5: \quad if cur’s children are not initialized yet then
6: \quad \quad for \( 0 \leq i < \text{len}(\pi) \) do
7: \quad \quad \quad \quad cur.children[i] \leftarrow TRIENODE(
8: \quad \quad \quad \quad \quad parent = cur.mass = \pi[i] \cdot \text{cur massa}
9: \quad \quad \quad \quad index \leftarrow \text{randomly sample } i \text{ with probability } \propto \text{cur.children[index]}
10: \quad \quad \quad \quad cur \leftarrow cur.children[index]
11: \quad \quad \quad \quad return index
12: \quad \quad node \leftarrow cur
13: \quad \quad while node \neq \emptyset do
14: \quad \quad \quad node.mass \leftarrow node.mass \cdot cur.mass
15: \quad \quad \quad node \leftarrow node.parent
16: \quad \quad cur \leftarrow root

Each trie node \( n \) stores its total unsampled probability mass, denoted \( \text{mass}(n) \). If \( n \) represents the trace prefix \( t' \), then \( \text{mass}(n) \) equals

\[
\sum_{t \text{ is not sampled}} 1 \cdot 1[t' \text{ is a prefix of } t] \cdot P(t).
\]

We do not compute this sum directly because there will usually be too many traces to enumerate. Instead, we can compute the initial \( \text{mass}(n) \) value for a trie node \( n \) using Equation (2), and then update it incrementally after a trace is sampled (when \( P \) terminates). This is shown in Algorithm 2.

First, INITIALIZE is called exactly once, before the program \( P \) is run. When \( P \) requests a new random choice (RANDOMCHOICE in Algorithm 2), we look up the “current” trie node \( cur \) that corresponds to the state of \( P \)’s execution. If \( cur \) has not been reached before in a previous execution, we initialize its children. Then, we sample a child \( n_i \) of \( cur \) with probability proportional to \( \text{mass}(n_i) \), update \( cur \) to \( n_i \), and return \( i \). When one execution of \( P \) terminates, we have sampled a full trace \( t_s \), and execute PROCESSTERMINATION in Algorithm 2. We mark the corresponding node \( n_i \) as a leaf, and we must now update the \( \text{mass} \) values. The affected nodes are \( n_i \) and all of its ancestors, corresponding to all prefixes of \( t_s \), including \( t_s \) itself. For each affected node \( n_o \), we update \( \text{mass}(n_o) := \text{mass}(n_o) - P(t_s) \), where \( P(t_s) \) equals \( \text{mass}(n_i) \) before it is updated. Appendix B proves that this scheme results in traces sampled exactly from \( P(t) \) without replacement.

Finally, it is useful to detect when all possible traces have been sampled. Mathematically, this simply involves check-
While drawing samples, when the same initial program states are seen many times, UniqueRandomizer even faster than running (Algorithm 1). Trie nodes store their unsampled probability mass. Shaded nodes are those that have been sampled from before, so all of their children are instantiated. Known leaves are outlined in bold. After \( P \) terminates, PROCESSTERMINATION is called: the leaf’s probability mass of 0.27 is subtracted from the leaf and its ancestors. On the next run of \( P \), its first random choice request is again \([0.5, 0.4, 0.1] \). However, UniqueRandomizer’s RANDOMCHOICE function will return an index with probability proportional to the mass values \([0.5, 0.13, 0.1] \), reflecting the fact that the trace \([1, 0, 1] \) was previously sampled.

In practice however, accumulation of floating-point errors makes this unreliable. Appendix C describes a solution.

2.4. Extensions and Optimizations

Skipping Probability Computations Notice that the probability distributions computed by \( P \) and passed to \( C \) are only needed to compute initial mass values for the corresponding trie node’s children. Thus, \( P \) could be modified to only compute probability distributions when needed, depending on whether the current node’s children have already been initialized. This optimization can make sampling with UniqueRandomizer even faster than running \( P \) in a plain loop (i.e., naïve sampling with replacement), especially when the same initial program states are seen many times while drawing samples.

Incremental Batched Sampling One downside of UniqueRandomizer is that runs of \( P \) are difficult to parallelize, since the trie must be updated after each run of \( P \) before the next run can start. Kool et al. (2019b) previously introduced a different approach to sampling without replacement, called Stochastic Beam Search (SBS), which is a modification of beam search using the Gumbel-top-\( k \) trick to sample beam state expansions WOR. Like normal beam search, SBS allows for parallelization when expanding beam states. Section 3 compares UniqueRandomizer and SBS in detail.

It is actually possible to combine the strengths of UniqueRandomizer and SBS, resulting in a method of incremental batched sampling where each batch allows for parallelization and further batches can be sampled without replacement. Intuitively, one can think of UniqueRandomizer as storing the factorized probability distribution of the next sample, conditioned on the fact that previous samples can no longer be chosen. SBS is run with this probability distribution to select the next batch of samples without replacement, where beam states in SBS correspond to UniqueRandomizer’s trie nodes. After SBS returns a batch of samples, the mass values in the UniqueRandomizer trie are updated so that the new samples cannot be chosen by later batches.

Locally Modifying Probabilities By storing more information in the trie, we can enable efficient local updates to the factorized probability distribution, allowing it to change over time in response to new data while still avoiding previously-seen samples. See Appendix D for details.

2.5. Estimating Expectations

Many statistical estimators and learning methods expect i.i.d. samples; is there a way to reweight WOR samples so that they can be used as if they were i.i.d.? Suppose that we have samples \( s_1, \ldots, s_k \) drawn without replacement from an arbitrary distribution \( p(s) \), perhaps using UniqueRandomizer or some other algorithm, and we wish to estimate the expectation \( \mathbb{E}_{s \sim p}[f(s)] \) for some function \( f \). Estimating an expectation from WOR samples is a fundamental problem in survey statistics (Horvitz and Thompson, 1952), but many of these methods do not scale computationally to non-uniform distributions over large outcome spaces.

Kool et al. (2019b) present a clever solution to this problem. When asked to produce \( k \) samples WOR from \( p \), SBS actually produces a sequence of samples \( (s_i, G_i) \), where the set \( \{s_1, \ldots, s_k\} \) is the desired WOR sample, and each \( G_i \) is an auxiliary variable drawn from a Gumbel distribution.
We derive an equivalent estimator for $\kappa$ because we first draw samples and then draw a Gumbel variate from $(\text{Gumbel}(\log(1)))$. From WOR samples $s_1, \ldots, s_k$ and the stochastically-chosen $\kappa$, we can reduce variance in the latter case by repeating the “hindsight Gumbel” process to draw multiple $\kappa$ (using the same samples), producing multiple HGE estimates. Averaging these gives the Repeated HGE estimate with lower variance.

3. Analysis and Comparison to Related Work

SBS Stochastic Beam Search (Kool et al., 2019b) is a prior method of sampling WOR from sequence models. It is similar to UniqueRandomizer but has key differences.

First, one main advantage of UniqueRandomizer is that it draws samples incrementally, while SBS returns a fixed-size batch of samples. Thus, UniqueRandomizer allows for increased flexibility in the number of samples drawn, e.g., drawing samples until a solution is found, until a timeout is reached, after sampling a target fraction of the probability space, until reaching enough diversity in samples, or after the Hindsight Gumbel Estimator (Section 2.5) begins to converge. These kinds of stopping criteria would not be possible with SBS$^4$. Because UniqueRandomizer can be used with any arbitrary stopping criterion, it enables sampling-based approaches to be applicable in more scenarios. For example, given a model that generates candidate solutions to a Traveling Salesman Problem instance, SBS could be used to find the best tour among a fixed number of samples, but it would not be able to efficiently find a tour with cost below some threshold. In contrast, UniqueRandomizer is applicable in the latter case due to its incrementality.

Second, like normal beam search, the input to SBS is a

2Kool et al. (2019b) and Maddison and Tarlow (2017) provide an excellent overview of Gumbels. For more on the Gumbel-top-$k$ trick, see Vieira (2014) and Kool et al. (2019b).

3Sampling from this conditional distribution is related to the single-sample retrospective Gumbel question considered by Maddison and Tarlow (2017) and Dinh (2016).

4Although beam search and SBS produce batches of distinct samples, drawing multiple batches may lead to duplicates between batches. One also cannot reuse computation performed for one batch without an auxiliary data structure, which is the purpose of the trie in the incremental batched version of UniqueRandomizer.
next-state function that enumerates the children of a state. For randomized programs \( \mathcal{P} \), this requires “pausing” and “unpausing” \( \mathcal{P} \)’s execution. Thus, the state must contain all of the program context (e.g., local variables) necessary to resume execution. While this is easy for a sequence model, it can be challenging for more complex programs. In contrast, UniqueRandomizer can be elegantly implemented as a wrapper on libraries for random number generation, and it is agnostic to the program context that \( \mathcal{P} \) might use.

A third difference is in the number of nodes expanded. When sampling from machine learning models, it is usually expensive to compute the probability distribution over a node’s children required to expand a node. In non-degenerate scenarios, UniqueRandomizer requires fewer expansions than SBS, since UniqueRandomizer only expands the nodes necessary to reach the sampled leaves, while SBS also expands states that later fall off the beam. For example, suppose we sample \( k \) sequences of length \( L \) with a vocabulary size \( V \geq k \). SBS expands exactly \( 1 + (L - 1)k \) beam states. For UniqueRandomizer, the worst case is when all \( k \) sequences have a different first element, so UniqueRandomizer expands the same number of nodes as SBS. In the best case, only \( L \) expansions are needed if all \( k \) sequences differ only at the final position. We also note that the combination of UniqueRandomizer and SBS (incremental batched sampling in Section 2.4) provides smooth intermediate behavior: by sampling \( b \) batches of size \( k/b \), we reduce the number of expansions with high \( b \) but obtain better parallelization for low \( b \). Setting \( b = k \) or \( b = 1 \) reduces to the behavior of UniqueRandomizer or SBS, respectively.

Finally, SBS maintains beam nodes for the current and next time step, where each node stores an arbitrarily-complex intermediate state of \( \mathcal{P} \). In contrast, UniqueRandomizer stores more nodes (i.e., the entire trie), but each node only stores one mass value. Using UniqueRandomizer to sample outputs of \( \mathcal{P} \) is at worst a constant factor slower than running \( \mathcal{P} \) in a loop\(^5\), but with the optimization in Section 2.4, the UniqueRandomizer approach may actually be faster.

UniqueRandomizer is mathematically simpler than SBS and was developed independently, but we build upon SBS for the incremental batched sampling extension and the Hindsight Gumbel Estimator.

Algorithms for Random Sampling Matias et al. (2003) describe a dynamic tree data structure to sample from discrete distributions with dynamically changing weights, but this is not naturally adapted to sampling WOR. Various algorithms have been proposed for sampling WOR (Wong and Easton, 1980; Vinterbo, 2010; Duffield et al., 2007; Efraimidis and Spirakis, 2006), but generally these algorithms do not consider the case of factorized distributions like (2), and so are inefficient for programs with potentially long traces. The relationship between these methods and the Gumbel-max trick is described by Vieira (2014). The incremental batched version of UniqueRandomizer is related to the top-down Gumbel heap of Maddison et al. (2014). An alternate way of obtaining diverse samples is to explicitly add penalties to beam search to encourage diversity (Vijayakumar et al., 2018; Li and Jurafsky, 2016). It is likely possible that UniqueRandomizer can be alternatively implemented using a continuation-passing style and other techniques common in probabilistic programming (Goodman and Stuhlmüller, 2014).

4. Experiments

We demonstrate that UniqueRandomizer can lead to improvements in a variety of applications.

4.1. Program Synthesis

We apply UniqueRandomizer to program synthesis as described in the motivating example (Section 2). We use the Search-based Pseudocode to Code (SPoC) dataset (Kulal et al., 2019), which is a collection of 677 problems from programming competitions with 18,356 C++ solution programs written by competitors. Each line of each program has an accompanying human-authored line of pseudocode. Additionally, each problem has corresponding test cases that we can use to verify the correctness of a solution program. The goal is to translate given pseudocode into a code solution for the problem. Compared to natural language translation, the difference is that our “translation” is expected to be syntactically correct code consistent with the test cases, and even a single token error likely means that the translation is incorrect.

We train a Transformer model (Vaswani et al., 2017) on the sequence-to-sequence task of generating a line of code given a line of pseudocode. Details are in Appendix E. To sample full programs, we iteratively condition the model on each line of pseudocode and sample one code line for each. Concatenating the code lines produces the full program.

We evaluate on the 3,386 pseudocode prompts in the two test splits of the SPoC dataset, “testp” and “testw,” drawing up to 100 samples for each pseudocode prompt. We compare sampling with replacement (using Monte Carlo i.i.d. sampling) and sampling without replacement (using the batched version of UniqueRandomizer with batch size 10). Each sampled program is checked for correctness using the provided test cases. We use a temperature \( \tau = 0.8 \) when sampling with replacement and \( \tau = 0.4 \) when sampling
without replacement, which we found to be the best among {0.2, 0.4, 0.6, 0.8, 1.0}. Note that a low temperature leads to samples for which the model is more confident but also causes more duplicates when sampling with replacement.

For both methods of sampling, we report the number of pseudocode prompts for which a sampled program passes all test cases. Figures 3a and 3b show that UniqueRandomizer finds solutions with fewer samples than i.i.d. sampling, and succeeds on more prompts overall.

The success rate of our approach is slightly lower than the method in Kulal et al. (2019) which achieves 32.5% success on testp and 51.0% on testw, with a budget of 100 programs and without using compiler diagnostics for error localization. This may be because Kulal et al. use a more sophisticated model with a copy mechanism and coverage vector, while we simply use a vanilla Transformer. In any case, Kulal et al. find that using compiler diagnostics increases the success rate by 1.7% on testp and 2.7% on testw, while our approach of sampling WOR improves over i.i.d. sampling by a comparable amount, 2.0% on testp and 2.3% on testw.

Our objective in this experiment was not to surpass the state-of-the-art, but rather to show that using UniqueRandomizer to draw samples without replacement leads to significant improvement over sampling with replacement. Note that beam search and SBS are existing methods of drawing unique samples, but they are not incremental and would be difficult to use in this setting where one does not know upfront how many samples are needed. Additionally, sampling a program takes about 3 times as long as compiling and executing it on the test cases, so drawing fewer samples is important.

Efficiency We also examine the efficiency of various sampling methods in Figure 3c, where we draw 100 samples with a GPU using the same medium-sized pseudocode prompt with 12 lines. We observe that UniqueRandomizer is more than twice as fast as Monte Carlo i.i.d. sampling, explained by the optimization of skipping probability computations (Section 2.4). Further note that UniqueRandomizer produces unique samples, while Monte Carlo i.i.d. sampling does not. Rejection sampling is a naïve method of obtaining unique samples but it is slower than Monte Carlo i.i.d. sampling, in fact becoming progressively slower as more samples are drawn. SBS achieves a low time for 100 samples by using batched computations, but all samples are drawn simultaneously, even if a solution is found within the first few samples, and there is no easy way to draw further unique samples if the first 100 samples are insufficient. The batched version of UniqueRandomizer allows for drawing further batches of unique samples, while still taking advantage of batched computation. Thus, if fewer than 100 samples were actually needed, the batched version of UniqueRandomizer would be significantly faster than SBS. We also compare to a naïve way of using SBS “incrementally,” where the batch size doubles on each iteration as \( \min\{2 \times \text{previous batch size}, 100\}\) and duplicate samples between batches are discarded, but this method is much slower than UniqueRandomizer.

4.2. Traveling Salesman Problem

Sampling without replacement can be used for combinatorial optimization problems, such as the Traveling Salesman Problem (TSP). Several deep learning models for TSP have been proposed (Bello et al., 2016; Deudon et al., 2018; Kool et al., 2018), where the model outputs candidate tours, and the best tour is chosen among multiple samples drawn from the model. Most recently, the Attention Model by Kool et al. (2018) was shown to be better than the others on randomly-generated TSP instances with \( n = 20, 50, \) and 100 nodes.

We compare sampling with and without replacement from the Attention Model, using 1280 samples as in Kool et al. (2018), as well as 100 samples. We also record the number
of duplicate tours sampled in each setting, which quantifies the amount of computation wasted by i.i.d. sampling (with replacement). Table 1 shows the results on Kool et al.’s dataset. Observe that i.i.d. sampling (used by Kool et al.) results in many duplicate tours, especially on the smaller graphs. The duplicate rate also increases with the number of samples, e.g., from 63.6% duplicate in 100 samples to 87.6% duplicate in 1280 samples for \( n = 50 \). In contrast, sampling without replacement avoids duplicates altogether, leading to better final costs. For \( n = 20 \), sampling 100 tours without replacement outperforms sampling 1280 i.i.d. tours with replacement (with optimality gaps of 0.011% and 0.063% respectively).

For this application, incremental sampling with \textit{UniqueRandomizer} is 5-25% faster than naïve incremental i.i.d. sampling due to the optimization from Section 2.4. In the batched case, sampling without replacement using SBS is about 45-80% slower than batched i.i.d. sampling, but for the smaller graph sizes, the slowdown of SBS is outweighed by the benefit of avoiding duplicate samples.

\textbf{Farthest Insertion} To demonstrate that \textit{UniqueRandomizer} is applicable to a wide range of randomized programs, we also consider the \textit{farthest insertion} heuristic for TSP, which is the best greedy baseline in Kool et al. (2018)’s results. This heuristic maintains a cycle for a subset of the nodes, and on each iteration, the node that is farthest from the cycle is inserted into the cycle at the cheapest location (such that the new cycle has minimal cost). We transform the greedy heuristic into a discrete randomized program by relaxing the greedy choice for insertion location. Specifically, if inserting a node at location \( i \) causes the cycle’s cost to increase by \( \text{costDelta}(i) \), then we sample an insertion location \( i \) with probability proportional to \( \exp(-\text{costDelta}(i)/\tau) \) where \( \tau \) is a temperature hyperparameter. In the limit, \( \tau = 0 \) corresponds to choosing an insertion heuristic, while \( \tau = \infty \) corresponds to choosing an insertion location uniformly. We set \( \tau = 0.3, 0.2, \) and 0.15 for \( n = 20, 50, \) and 100 nodes, respectively. These choices of \( \tau \) were obtained from a simple search over the range \( 0.05 \leq \tau \leq 0.5 \).

We then use \textit{UniqueRandomizer} to sample candidate tours without replacement from the modified farthest insertion heuristic. The results are in Table 2. We found that the heuristic approach actually outperforms two of the three learned models included in Kool et al. (2018)’s comparison, in fact outperforming all three for \( n = 20 \).

This result is encouraging given the simplicity of this experiment. We started with a well-known greedy heuristic, and by changing only a few lines of code, we turned it into a randomized program via a straightforward relaxation and used \textit{UniqueRandomizer} to sample without replacement. By tuning only one hyperparameter (the temperature \( \tau \)),

### Table 1: Sampling without replacement from the Attention Model (Kool et al., 2018) improves upon i.i.d. sampling. We show results for 100 and 1280 samples per TSP instance. Concorde (Applegate et al., 2003) is an exact solver, and “Gap” is the optimality gap relative to Concorde’s exact solution.

<table>
<thead>
<tr>
<th>Method</th>
<th>( n = 20 )</th>
<th>( n = 50 )</th>
<th>( n = 100 )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cost</td>
<td>Gap</td>
<td>Duplicates</td>
</tr>
<tr>
<td>Concorde (exact)</td>
<td>3.8357</td>
<td>0%</td>
<td>–</td>
</tr>
<tr>
<td>AM, with rep. ( \times 100 )</td>
<td>3.8397</td>
<td>0.105%</td>
<td>96.8</td>
</tr>
<tr>
<td>AM, with rep. ( \times 1280 )</td>
<td>3.8381</td>
<td>0.063%</td>
<td>1274.5</td>
</tr>
<tr>
<td>AM, w/o rep. ( \times 100 )</td>
<td>3.8361</td>
<td>0.011%</td>
<td>0</td>
</tr>
<tr>
<td>AM, w/o rep. ( \times 1280 )</td>
<td>3.8358</td>
<td>0.002%</td>
<td>0</td>
</tr>
</tbody>
</table>

### Table 2: \textit{UniqueRandomizer} applied to the farthest insertion heuristic for TSP outperforms two of three recent deep-learning approaches (Bello et al., 2016; Deudon et al., 2018; Kool et al., 2018). For methods marked with (*), the results are copied from Kool et al. (2018). All of the sampling approaches use 1280 samples per TSP instance.

<table>
<thead>
<tr>
<th>Method</th>
<th>( n = 20 )</th>
<th>( n = 50 )</th>
<th>( n = 100 )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cost</td>
<td>Gap</td>
<td>Cost</td>
</tr>
<tr>
<td>Concorde (exact)</td>
<td>3.8357</td>
<td>0%</td>
<td>5.696</td>
</tr>
<tr>
<td>Bello et al., i.i.d. sampling (*)</td>
<td>–</td>
<td>5.75</td>
<td>0.95%</td>
</tr>
<tr>
<td>EAN, i.i.d. sampling (*)</td>
<td>3.84</td>
<td>0.11%</td>
<td>5.77</td>
</tr>
<tr>
<td>AM, i.i.d. sampling</td>
<td>3.8381</td>
<td>0.063%</td>
<td>5.724</td>
</tr>
<tr>
<td>Farthest Insertion, greedy (1 tour)</td>
<td>3.9262</td>
<td>2.358%</td>
<td>6.011</td>
</tr>
<tr>
<td>Farthest Insertion, \textit{UniqueRandomizer}</td>
<td>3.8372</td>
<td>0.038%</td>
<td>5.746</td>
</tr>
</tbody>
</table>
and without any training, we obtained results that are competitive with those of carefully-constructed deep learning models.

4.3. (Repeated) Hindsight Gumbel Estimator

In Section 2.5 we proposed the Hindsight Gumbel Estimator for \( \mathbb{E}_p[f(s)] \), the expectation of a function of samples drawn without replacement from a distribution \( p \). HGE can be normalized using Equation (5) and/or repeated. Figure 4 shows the performance of the estimators on synthetic data. The Monte Carlo estimate simply averages \( f(s) \) for i.i.d. \( s \) sampled with replacement. We see that HGE, normalized and repeated for 10 iterations, has the lowest variance. We chose a heavily-skewed \( p \) so that sampling with replacement encounters many duplicates, and we enforce a strong correlation between \( p \) and \( f \) (or else incorrect estimators might appear to do well). These properties are common when performing estimation in the context of machine learning.

5. Conclusion

We presented \textit{UniqueRandomizer}, an efficient data structure for incremental sampling without replacement. We also derived the Hindsight Gumbel Estimator, a new estimator for samples drawn without replacement that has lower variance than similar previous estimators. Our experiments show that sampling without replacement leads to significant improvements over i.i.d. sampling in program synthesis and combinatorial optimization. The incremental nature of \textit{UniqueRandomizer} is especially important in domains like program synthesis where the number of required samples is not known upfront. By eliminating redundant probability computations, \textit{UniqueRandomizer} also allows for very efficient sampling, and the batched version is even more efficient.

Because \textit{UniqueRandomizer} is compatible with a broad class of programs, we believe it could be applicable to other problems beyond those explored in this paper. As with our program synthesis and TSP experiments, \textit{UniqueRandomizer} can be used to sample distinct outputs of a neural model to solve search problems (including constraint satisfaction problems) or obtain diversity in natural language generation (Kool et al., 2019b). One can also use \textit{UniqueRandomizer} to perform distinct rollouts in reinforcement learning (Kool et al., 2019a) or Monte Carlo Tree Search (Bronwne et al., 2012) to estimate the value of a state with the Hindsight Gumbel Estimator until that estimate converges sufficiently. \textit{UniqueRandomizer} may also be useful in the context of randomized rounding (Raghavan and Tompson, 1987) and probabilistic programming (van de Meent et al., 2018).


Acknowledgments

The authors would like to thank David Dohan, Andreea Gane, David Belanger, Danny Tarlow, and Percy Liang for helpful conversations and explorations.

References


Incremental Sampling Without Replacement for Sequence Models


A. Proof of Theorem 1

Theorem 1. \( \mathcal{P} \) is trace-injective \( \iff \) for all trace prefixes \( t' = [c_1, \ldots, c_h] \), the set of possible outputs \( F(t') \) is partitioned by the next choice \( c_{h+1} \sim C(\pi_{h+1}) \), i.e., the set \( \{ F([c_1, \ldots, c_h, c_{h+1}]) | c_{h+1} \in \{0, \ldots, \text{len}(\pi_{h+1}) - 1\} \} \) is a partition of \( F(t') \).

Proof. For any trace prefix \( [c_1, \ldots, c_h] \), define the collection
\[
\mathcal{F}([c_1, \ldots, c_h]) = \{ F([c_1, \ldots, c_h, c_{h+1}]) | c_{h+1} \in \{0, \ldots, \text{len}(\pi_{h+1}) - 1\} \}.
\]

First, if \( \mathcal{P} \) is trace-injective, then for any trace prefix \( [c_1, \ldots, c_h] \), and any \( c \neq c' \), let \( y_1 \in F([c_1, \ldots, c_h, c]) \) and \( y_2 \in F([c_1, \ldots, c_h, c']) \). Then there exist traces
\[
t^{(1)}(1) = [c_1, \ldots, c_h, c_{h+1}^{(1)}, \ldots, c_h^{(1)}]
\]
and
\[
t^{(2)}(2) = [c_1, \ldots, c_h, c_{h+1}^{(2)}, \ldots, c_h^{(2)}]
\]
such that
(a) \( c_{h+1}^{(1)} = c \) and \( c_{h+1}^{(2)} = c' \),
(b) \( f(t^{(1)}) = y_1 \) and \( f(t^{(2)}) = y_2 \).

Clearly \( t^{(1)} \neq t^{(2)} \), so because \( f \) is injective, we have that \( y_1 \neq y_2 \). Also \( \cup_c F([c_1, \ldots, c_h, c']) = F([c_1, \ldots, c_h]) \). This means that \( \mathcal{F}([c_1, \ldots, c_h]) \) is a partition of \( F([c_1, \ldots, c_h]) \).

Conversely, assume that \( \mathcal{F}(t') \) is a partition of \( F(t') \) for any trace prefix \( t' \). Let
\[
t^{(1)} = [c_1^{(1)}, \ldots, c_h^{(1)}]
\]
and
\[
t^{(2)} = [c_1^{(2)}, \ldots, c_h^{(2)}]
\]
be distinct traces. Let \( h \) be the length of their longest common prefix, so \( c_i^{(1)} = c_i^{(2)} \) for all \( 1 \leq i \leq h \), and \( c_{h+1}^{(1)} \neq c_{h+1}^{(2)} \). By definition,
\[
f(t^{(1)}) \in F([c_1^{(1)}, \ldots, c_h^{(1)}, c_{h+1}^{(1)}])
\]
and
\[
f(t^{(2)}) \in F([c_1^{(2)}, \ldots, c_h^{(2)}, c_{h+1}^{(2)}])
\]
\[
= F([c_1^{(1)}, \ldots, c_h^{(1)}, c_{h+1}^{(2)}]).
\]
But these two sets are disjoint, because \( \mathcal{F}([c_1^{(1)}, \ldots, c_h^{(1)}]) \) partitions the set \( F([c_1^{(1)}, \ldots, c_h^{(1)}]) \). Therefore, \( f(t^{(1)}) \neq f(t^{(2)}) \), establishing that \( f \) is injective and that \( \mathcal{P} \) is trace-injective.

B. Proof of Correctness

Theorem 2. Let \( \mathcal{P} \) be a discrete randomized program that terminates, and let \( P(t) \) be the probability that \( \mathcal{P} \) runs with trace \( t \). Suppose we have already sampled distinct traces \( t_1, \ldots, t_j \). If, at any UniqueRandomizer trie node \( n \) we move to a child \( c \) with probability proportional to \( \text{mass}(c) \), then upon reaching a leaf node, the resulting trace is drawn from \( P(t | t \notin \{t_1, \ldots, t_j\}) \).

Proof. Let \( n_0, \ldots, n_h \) be any root-to-leaf path, where \( n_0 \) is the root and \( n_h \) is the leaf. Let \( t \) be the trace corresponding to \( n_h \). According to Equation (3),
\[
\text{mass}(n_h) = \begin{cases} 
0 & \text{if } t \in \{t_1, \ldots, t_j\} \\
\text{otherwise} & \end{cases}
\]

We complete the proof by showing that the leaf \( n_h \) is reached with the desired probability:
\[
P(n_h \text{ is reached})
= \prod_{i=1}^{h} P(n_i \text{ is the selected child of } n_{i-1})
= \prod_{i=1}^{h} \frac{\text{mass}(n_i)}{\sum_{c \in \text{children}(n_{i-1})} \text{mass}(c)}
= \prod_{i=1}^{h} \frac{\text{mass}(n_i)}{\frac{\text{mass}(n_h)}{\text{mass}(n_0)} \frac{1}{1 - \sum_{i=1}^{j} P(t_i)}}
= \begin{cases} 
0 & \text{if } t \in \{t_1, \ldots, t_j\} \\
\text{otherwise} & \end{cases}
= P(t | t \notin \{t_1, \ldots, t_j\}).
\]
Equality (7) holds because a non-leaf node’s mass equals the sum of its children’s mass values.

C. Detecting Exhausted Nodes

We say that a trie node is exhausted if it has zero unsampled probability mass, i.e., all of its probability mass is sampled. Due to floating-point errors, a node’s mass might not be set to exactly zero after it should be exhausted. We handle this by carefully propagating the information that a given node has zero unsampled probability mass.

When a node \( n \) is marked as a leaf, we directly assign \( \text{mass}(n) := 0 \). Then, when subtracting mass from one of \( n \)’s ancestors \( a \), we first check if \( \text{mass}(c) = 0 \) for all children \( c \) of \( a \). If so, we directly set \( \text{mass}(a) := 0 \) instead of
using a subtraction operation. With this approach, a node’s mass will be exactly 0 after all of its descendent leaves are sampled. Algorithm 3 includes this process, elaborating on the pseudocode in Algorithm 2.

D. Locally Modifying the Factorized Probability Distribution

A slight modification of UniqueRandomizer’s trie allows for efficient local updates to the factorized probability distribution. Instead of storing unsampled probability masses of nodes, the modified trie nodes now store the unsampled fraction of the node’s total probability mass. Edges in the trie now store the initial probability of following that edge from the source node, as given in the probability distribution provided by $\mathcal{P}$.

Note that the unsampled probability mass of a node $n$ is equal to the product of the edge probabilities from the root to $n$, times the unsampled fraction at $n$. Therefore, by accumulating the product of edge probabilities while walking down the trie, we can compute the unsampled probability mass of nodes, so we can recreate the original UniqueRandomizer behavior with the modified trie.

This decomposition enables local modifications to the factorized probability distribution. More precisely, suppose that a trie node $n$ has $k$ children, denoted $n_1, \ldots, n_k$, and $n$ initially has outward edge probabilities of $p_1, \ldots, p_k$. We wish to change these edge probabilities to $p'_1, \ldots, p'_k$, so that further samples come from the updated probability distribution and previously-seen samples are still avoided. We do this by updating the trie in the following way. First, we directly replace $n$’s outward edge probabilities with the desired $p'_1, \ldots, p'_k$. Then, we compute the new unsampled fraction at $n$ with a weighted average of $n$’s children:

$$\text{unsampledFraction}(n) := \sum_{i=1}^{k} \text{edgeProbability}(n, n_i) \cdot \text{unsampledFraction}(n_i).$$

Finally, we perform a similar update for all of $n$’s ancestors in upward order (with the root being updated last). After these updates, all values in the trie reflect the new probability distribution.

E. Program Synthesis Experiment Details

For the program synthesis task, we train a Transformer model (Vaswani et al., 2017) to translate lines of pseudocode to lines of C++ code. We use the Transformer implementation in the Trax framework\footnote{https://github.com/google/trax}. The Transformer uses 2 attention heads, 3 hidden layers, a filter size of 1024, and a hidden dimension size of 512. We train using ADAM with learning rate 0.05 and batch size 512 for 12,000 steps, which is approximately when the models achieve their lowest evaluation loss. We use linear learning rate warmup for the first 1,000 steps. These hyperparameters were chosen from the search space in Table 3, selecting the run with the lowest evaluation loss at the end of training. As in Kulal et al. (2019), we withhold 10% of the training examples as the validation set.

Some of the shorter lines of code in the SPoC dataset have no pseudocode. In some of these instances, we augment the line with pseudocode ourselves. Specifically, if the line is exactly “}” or “};” we provide pseudocode “end”, if the line is exactly “int main() {” we provide pseudocode “main”, and if the line is exactly “return 0;” we provide pseudocode “return”.

Table 3: The search space used for tuning the Transformer model.

<table>
<thead>
<tr>
<th>Hyperparameter</th>
<th>Search space</th>
<th>Selected value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Learning rate</td>
<td>{0.05, 0.075, 0.1, 0.15}</td>
<td>0.05</td>
</tr>
<tr>
<td>Hidden layers</td>
<td>{1, 2, 3}</td>
<td>3</td>
</tr>
<tr>
<td>Hidden dimension size</td>
<td>{512, 1024}</td>
<td>512</td>
</tr>
<tr>
<td>Attention heads</td>
<td>{2, 4}</td>
<td>2</td>
</tr>
<tr>
<td>Filter size</td>
<td>{512, 1024}</td>
<td>1024</td>
</tr>
</tbody>
</table>

Algorithm 3 \textit{UniqueRandomizer}, with careful detection of exhausted nodes:

\begin{algorithm}
\SetAlgoLined
\caption{UniqueRandomizer, with careful detection of exhausted nodes}
\begin{algorithmic}[1]
\Procedure{Initialize}{} \Comment{Called once to initialize the data structure}
\State $\texttt{root} \leftarrow \text{TrieNode}(\texttt{parent} = \emptyset, \texttt{mass} = 1)$
\State $\texttt{cur} \leftarrow \texttt{root}$
\EndProcedure

\Procedure{Exhausted}{\texttt{node}} \Comment{Whether node is completely sampled}
\If {$\texttt{node}$ is a leaf} \Comment{If node is a leaf then}
\State \Return $\text{True}$
\EndIf
\If {$\texttt{node}$ has never been sampled from before} \Comment{If node has never been sampled from before then}
\State \Return $\text{False}$
\EndIf
\State \Return whether all of $\texttt{node}$'s children have 0 mass
\EndProcedure

\Procedure{RandomChoice}{$\pi$} \Comment{Called when $\mathcal{P}$ requests a random choice}
\If {$\text{Exhausted}(\texttt{cur})$} \Comment{If exhausts $\texttt{cur}$ then}
\State \textbf{raise} Error(“no more unique traces exist”)
\EndIf
\If {$\texttt{cur}$'s children are not initialized yet} \Comment{If $\texttt{cur}$'s children are not initialized yet then}
\For {$0 \leq i < \text{len}(\pi)$} \Comment{For $0 \leq i < \text{len}(\pi)$ do}
\State $\texttt{cur}.\texttt{children}[i] \leftarrow \text{TrieNode}(\texttt{parent} = \texttt{cur}, \texttt{mass} = \pi[i] \cdot \texttt{cur}.\texttt{mass})$
\EndFor
\EndIf
\State $\texttt{index} \leftarrow$ randomly sample $i$ with probability $\propto \texttt{cur}.\texttt{children}[i].\texttt{mass}$
\State $\texttt{cur} \leftarrow \texttt{cur}.\texttt{children}[\texttt{index}]$
\State \Return $\texttt{index}$
\EndProcedure

\Procedure{ProcessTermination}{} \Comment{Called after $\mathcal{P}$ terminates}
\State $\texttt{mark} \texttt{cur} \text{ as a leaf}$
\State $\texttt{node} \leftarrow \texttt{cur}$
\While {$\texttt{node} \neq \emptyset$} \Comment{$\texttt{while}$ $\texttt{node} \neq \emptyset$ do}
\If {$\text{Exhausted}(\texttt{node})$} \Comment{If exhausts $\texttt{node}$ then}
\State $\texttt{node}.\texttt{mass} \leftarrow 0$
\Else
\State $\texttt{node}.\texttt{mass} \leftarrow \max\{\texttt{node}.\texttt{mass} - \texttt{cur}.\texttt{mass}, 0\}$
\EndIf
\State $\texttt{node} \leftarrow \texttt{node}.\texttt{parent}$
\EndWhile
\State $\texttt{cur} \leftarrow \texttt{root}$
\EndProcedure
\end{algorithmic}
\end{algorithm}