Scaling Neural Program Synthesis with Distribution-Based Search

Nathanaël Fijalkow[1, 2], **Guillaume Lagarde**[1], **Théo Matricon**[1], **Kevin Ellis**[3], **Pierre Ohlmann**[4], **and Akarsh Potta**[5]

[1] CNRS, LaBRI and Université de Bordeaux, France
[2] The Alan Turing Institute of data science, United Kingdom
[3] Cornell University, United States
[4] University of Paris, France
[5] Indian Institute of Technology Bombay, India

Abstract

We consider the problem of automatically constructing computer programs from input-output examples. We investigate how to augment probabilistic and neural program synthesis methods with new search algorithms, proposing a framework called distribution-based search. Within this framework, we introduce two new search algorithms: HEAP SEARCH, an enumerative method, and SQRT SAMPLING, a probabilistic method. We prove certain optimality guarantees for both methods, show how they integrate with probabilistic and neural techniques, and demonstrate how they can operate at scale across parallel compute environments. Collectively these findings offer theoretical and applied studies of search algorithms for program synthesis that integrate with recent developments in machine-learned program synthesizers.

Introduction

Writing software is tedious, error-prone, and accessible only to a small share of the population – yet coding grows increasingly important as the digital world plays larger and larger roles in peoples' lives. Program synthesis seeks to make coding more reliable and accessible by developing methods for automatically constructing programs (Gulwani, Polozov, and Singh 2017). For example, the FlashFill system (Gulwani, Polozov, and Singh 2017) in Microsoft Excel makes coding more accessible by allowing nontechnical users to synthesize spreadsheet programs by giving input-output examples, while the TF-coder system (Shi, Bieber, and Singh 2020) seeks to make coding neural networks more reliable by synthesizing TensorFlow code from input-outputs. Where these systems have been most successful is when they pair a specialized *domain-specific language* (DSL) to a domain-specific search algorithm for synthesizing programs in that DSL. A recent trend – both in industry (Kalyan et al. 2018) and academia (Devlin et al. 2017) – is to employ machine learning methods to learn to quickly search for a program in the DSL (Balog et al. 2017; Devlin et al. 2017; Lee et al. 2018; Zhang et al. 2018; Polosukhin and Skidanov 2018; Kalyan et al. 2018; Zohar and Wolf 2018; Chen, Liu, and Song 2018). Many such recent works have explored engineering better neural networks for guiding program search, effectively by training the network to act as a language model over source code that conditions on input-outputs (Polosukhin and Skidanov 2018). Here, we 'pop up' a level and instead ask: given a neural net that probabilistically generates source code, how can we most efficiently deploy that model in order to find a program consistent with some input-outputs? This concern arises because program synthesis requires solving a hard combinatorial search problem (exploring a possibly infinite space of programs), so taming this combinatorial explosion makes the difference between a practically useful system, and a system which cannot scale to anything but the most trivial of programs.

At a high-level the approaches we develop in this work follow a 2-stage pipeline: in the first stage a learned model predicts probabilistic weights, and in the second stage a symbolic search algorithm uses those weights to explore the space of source code. Our contributions target the second stage of this pipeline, and we focus on theoretical analysis of sampling-based search algorithms, new search algorithms based on neurally-informed enumeration, and empirical evaluations showing that recent neural program synthesizers can compose well with our methods.

This 2-stage pipelined approach has several benefits. The first is that the cost of querying the neural network is usually very small compared to the cost of combinatorial search, yet in practice the neural model learns to provide rough-and-ready probabilistic predictions to guide the search. A second benefit is that even if the probabilistic predictions are inaccurate, our methods maintain soundness and completeness (but may take longer to run). Another appeal is that it can be naturally combined with other classical approaches for program synthesis.

Our contributions:

- A theoretical framework called distribution-based search for evaluating and comparing search algorithms in the context of machine-learned predictions.
- Two new search algorithms: HEAP SEARCH, an enumerative method, and SQRT SAMPLING, a probabilistic method. We prove a number of theoretical results about them, in particular that they are both loss optimal.
- A generic method for parallelizing any search algorithm that explores a space of (syntax) trees

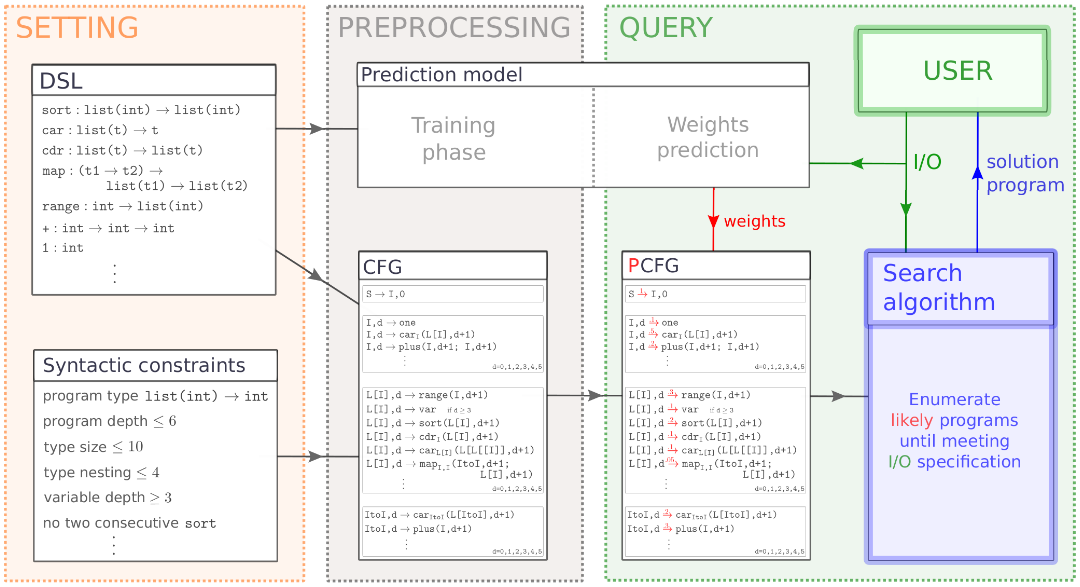


Figure 1: Pipeline for neural predictions for syntax guided program synthesis.

Distribution-Based Search

We work within the syntax guided program synthesis (SyGuS) framework introduced by (Alur et al. 2013), see also (Alur et al. 2018). In this setting, the DSL is given by a set of primitives together with their (possibly polymorphic) types and semantics.

We describe the machine learning pipeline for program synthesis, illustrated in Figure 1 on a toy DSL describing integer list manipulating programs.

The compilation phase constructs a context-free grammar (CFG) from the DSL together with a set of syntactic constraints. The CFG may incorporate important information about the program being generated, such as the n last primitives (encompassing n-gram models) or semantic information (*e.g.* non-zero integer, sorted list).

A prediction model (typically a neural network) takes as inputs a set of I/O and outputs a probabilistic labelling of the CFG, inducing a probabilistic context-free grammar (PCFG). The network is trained so that most likely programs (with respect to the PCFG) are the most likely to be solutions, meaning map the inputs to corresponding outputs.

We refer to the long version (Fijalkow et al. 2021) for an in-depth technical discussion on program representations and on the compilation phase. In this work we focus on the search phase and start with defining a theoretical framework for analysing search algorithms.

The PCFG obtained through the predictions of the neural network defines a probabilistic distribution $\mathcal{D}$ over programs. We make the theoretical assumption that the program we are looking for is actually sampled from $\mathcal{D}$, and construct algorithms searching through programs which find programs sampled from $\mathcal{D}$ as quickly as possible. Formally, the goal is to minimise the expected number of programs the algorithm outputs before finding the right program.

We write $A(n)$ for the n^{th} program chosen by the algorithm A; since A may be a randomised algorithm $A(n)$ is a random variable. The performance $\mathcal{L}(A,\mathcal{D})$ of the algorithm A, which we call its loss, is the expected number of tries it makes before finding x:

$$\mathcal{L}(A,\mathcal{D}) = \mathbb{E}_{x\sim\mathcal{D}}\left[\inf\{n\in\mathbb{N} : A(n)=x\}\right].$$

An algorithm A^* is 'loss optimal' if $\mathcal{L}(A^*,\mathcal{D}) = \inf_A \mathcal{L}(A,\mathcal{D})$. Let us state a simple fact: an algorithm is loss optimal if it generates each program once and in non-increasing order of probabilities. Depending on $\mathcal{D}$ constructing an efficient loss optimal algorithm may be challenging, pointing to a trade off between quantity and quality: is it worth outputting a lot of possibly unlikely programs quickly, or rather invest more resources into outputting fewer but more likely programs?

An example. To illustrate the definitions let us consider the distribution $\mathcal{D}$ over the natural numbers such that $\mathcal{D}(n) = \frac{1}{2^{n+1}}$; it is generated by the following PCFG:

$$S \to^{.5} f(S) \quad ; \quad S \to^{.5} x,$$

when identifying n with the program $f^n(x)$. Let us analyse a few algorithms.

- The algorithm A_1 enumerates in a deterministic fashion the natural numbers starting from 0: $A_1(n) = n$. Then $\mathcal{L}(A_1,\mathcal{D}) = \sum_{n\geq 0}\frac{n+1}{2^{n+1}} = 2$. This enumeration algorithm A_1 is loss optimal.
- The algorithm A_2 samples the natural numbers using the distribution $\mathcal{D}$. For $n \geq 0$, the value of $\mathbb{E}\left[\inf\{n' : A_2(n') = n\}\right]$ is 2^{n+1}: this is the expectation of the geometric distribution with parameter $\frac{1}{2^{n+1}}$. Then $\mathcal{L}(A_2,\mathcal{D}) = \sum_{n\geq 0}\frac{2^{n+1}}{2^{n+1}} = \infty$. Hence the naive sampling algorithm using $\mathcal{D}$ has infinite loss.

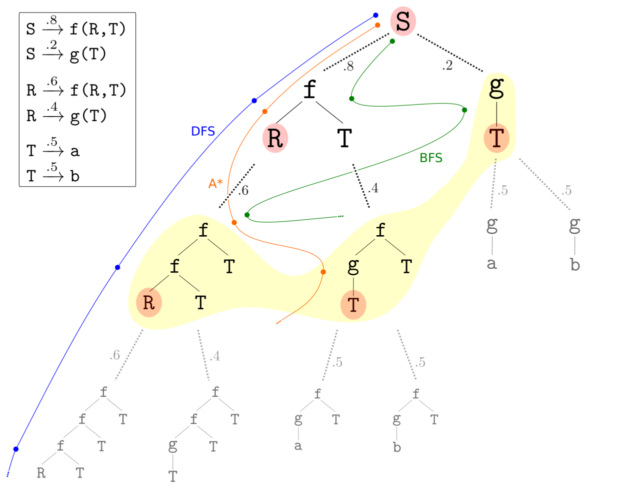

S $\xrightarrow{.8}$ f(R,T)
S $\xrightarrow{.2}$ g(T)
R $\xrightarrow{.6}$ f(R,T)
R $\xrightarrow{.4}$ g(T)
T $\xrightarrow{.5}$ a
T $\xrightarrow{.5}$ b
DFS
A*
BFS

Figure 2: Illustration of the tree of leftmost derivations.

- The algorithm A_3 samples the natural numbers using a distribution that we call $\sqrt{\mathcal{D}}$ defined[1] by $\sqrt{\mathcal{D}}(n) = \frac{1}{1+\sqrt{2}} \frac{1}{2^{\frac{n+1}{2}}}$.

  For $n \geq 0$, the value of $\mathbb{E}\left[\inf\{n' : A_3(n') = n\}\right]$ is $(1+\sqrt{2})2^{\frac{n+1}{2}}$: this is the expectation of the geometric distribution with parameter $\frac{1}{1+\sqrt{2}} \frac{1}{2^{\frac{n+1}{2}}}$. Then

$$\begin{aligned}\mathcal{L}(A_3, \mathcal{D}) &= \sum_{n \geq 0} \frac{(1+\sqrt{2})2^{\frac{n+1}{2}}}{2^{n+1}} \\ &= (1+\sqrt{2}) \sum_{n \geq 0} \frac{1}{2^{\frac{n+1}{2}}} \\ &= (1+\sqrt{2})^2 \approx 5.83.\end{aligned}$$

  As we will prove in a more general statement (Theorem 2), the algorithm A_3 is loss optimal among sampling algorithms. Suprisingly it is not much worse than the loss optimal algorithm, yet offers many advantages: it is much easier to implement, and requires $O(1)$ memory.

The next two sections are devoted to the two natural classes of algorithms for distribution-based search: *enumeration* and *sampling*. We then study how they can run at scale accross parallel compute environments.

Enumerative Methods and the HEAP SEARCH Algorithm

A number of enumerative methods have been investigated in previous works (Menon et al. 2013; Balog et al. 2017; Feng et al. 2018; Zohar and Wolf 2018). They proceed in a top-down fashion, and can be understood as ways of exploring the tree of leftmost derivations of the grammar as illustrated in Figure 2.

We present a new efficient and loss optimal algorithm called HEAP SEARCH and following a bottom-up approach.

[1]For the normalisation factor, note that $\sum_{n \geq 0} \frac{1}{2^{\frac{n+1}{2}}} = 1+\sqrt{2}$.

It uses a data structure based on heaps to efficiently enumerate all programs in non-increasing order of the probabilities in the PCFG.

Let us write $\mathcal{D}$ for the distribution induced by a PCFG. For a program x, we say that x' is the 'successor of x' if it is the most likely program after x, meaning $\mathcal{D}(x) > \mathcal{D}(x')$ and there are no programs x'' such that $\mathcal{D}(x) > \mathcal{D}(x'') > \mathcal{D}(x')$. For a non-terminal T in the grammar, the 'successor of x from T' is the most likely program after x among those generated from T. We define 'predecessor of x' and 'predecessor of x from T' in a similar way.

We create a procedure **Query**(T, x) which for any program x generated from a non-terminal T outputs the successor of x from T. Note that once this is defined, the algorithm performs successive calls to **Query**(S, x) with S the initial non-terminal and x the latest generated program, yielding all programs in non-increasing order of the probabilities.

To explain how **Query**(T, x) works, we first define the data structure. For each non-terminal T we have a hash table $\textsc{Succ}_T$ which stores the successors of all previously seen programs generated from T, and a heap $\textsc{Heap}_T$ which contains candidate programs, ordered by non-increasing probability. The key invariant is the following: the successor of T from x has either already been computed, hence is in $\textsc{Succ}_T$, or is the maximum program in $\textsc{Heap}_T$. This means that implementing **Query**(T, x) is very simple: it checks whether the successor has already been computed and returns it in that case, and if not it pops the heap. The difficulty is in maintaining the invariant; for this we need to add a number of candidate programs to the heaps. They are obtained by substituting one argument from the returned successor, and propagate this recursively to the corresponding non-terminals.

Theorem 1. *The* HEAP SEARCH *algorithm is loss optimal: it enumerates every program exactly once and in non-increasing order of probabilities.*

We refer to the long version (Fijalkow et al. 2021) for a complete description of the algorithm with a pseudocode, a proof of Theorem 1, and a computational complexity analysis.

HEAP SEARCH is related to A^* from (Feng et al. 2018): they are both loss optimal algorithms, meaning that they both enumerate programs in non-increasing order of probabilities. As we will see in the experiments HEAP SEARCH is better in two aspects: it is faster, and it is bottom-up, implying that program evaluations can be computed along with the programs and avoiding evaluating twice the same (sub)program.

Sampling Methods and the SQRT SAMPLING Algorithm

A *sampling algorithm* takes random samples from a distribution $\mathcal{D}'$; what is both a strength and a weakness is that a sampling algorithm is memoryless: a weakness because the algorithm does not remember the previous draws, which means that it may draw them again, but also a strength because it uses very little space and can be very easily implemented.

In the case of sampling algorithms, we identify algorithms with distributions. The following theorem shows a dichotomy: either there exists a loss optimal sampling algorithm among sampling algorithms, and then it is characterised as the 'square root' of the distribution, or all sampling algorithms have infinite loss.

Theorem 2. *Let $\mathcal{D}$ a distribution over a set X. If $\sum_{x \in X} \sqrt{\mathcal{D}(x)} < \infty$, the distribution $\sqrt{\mathcal{D}}$ defined by*

$$\sqrt{\mathcal{D}}(x) = \frac{\sqrt{\mathcal{D}(x)}}{\sum_{y \in X} \sqrt{\mathcal{D}(y)}}$$

is loss optimal among all sampling algorithms. If $\sum_{x \in X} \sqrt{\mathcal{D}(x)} = \infty$, for all sampling algorithms $\mathcal{D}'$ we have $\mathcal{L}(\mathcal{D}', \mathcal{D}) = \infty$.

Proof. Let $\mathcal{D}'$ be a distribution. For an element x, the expectation of the number of tries for $\mathcal{D}'$ to draw x is $\frac{1}{\mathcal{D}'(x)}$: this is the expectation of success for the geometric distribution with parameter $\mathcal{D}'(x)$. It follows that

$$\mathcal{L}(\mathcal{D}', \mathcal{D}) = \mathbb{E}_{x \sim D}\left[\frac{1}{\mathcal{D}'(x)}\right] = \sum_{x \in X} \frac{\mathcal{D}(x)}{\mathcal{D}'(x)}.$$

Let us assume that $\sum_{x \in X} \sqrt{\mathcal{D}(x)} < \infty$. Thanks to Cauchy-Schwarz inequality we have:

$$\begin{aligned} \left(\sum_{x \in X} \sqrt{\mathcal{D}(x)}\right)^2 &= \left(\sum_{x \in X} \sqrt{\frac{\mathcal{D}(x)}{\mathcal{D}'(x)}} \sqrt{\mathcal{D}'(x)}\right)^2 \\ &\leq \left(\sum_{x \in X} \frac{\mathcal{D}(x)}{\mathcal{D}'(x)}\right) \cdot \underbrace{\left(\sum_{x \in X} \mathcal{D}'(x)\right)}_{=1} \\ &= \sum_{x \in X} \frac{\mathcal{D}(x)}{\mathcal{D}'(x)}. \end{aligned}$$

We note that $\mathcal{L}(\sqrt{\mathcal{D}}, \mathcal{D}) = \left(\sum_{x \in X} \sqrt{\mathcal{D}(x)}\right)^2$, so the previous inequality reads $\mathcal{L}(\mathcal{D}', \mathcal{D}) \geq \mathcal{L}(\sqrt{\mathcal{D}}, \mathcal{D})$. Thus $\sqrt{\mathcal{D}}$ is loss optimal among sampling algorithms, and if it is not defined, then for any $\mathcal{D}'$ we have $\mathcal{L}(\mathcal{D}', \mathcal{D}) = \infty$. $\square$

Theorem 2 characterises the loss optimal sampling algorithm, but does not explain how to implement it. The following result answers that question.

Theorem 3. *If $\mathcal{D}$ is defined by a PCFG and $\sqrt{\mathcal{D}}$ is well defined, then we can effectively construct a PCFG defining $\sqrt{\mathcal{D}}$.*

The PCFG for $\sqrt{\mathcal{D}}$ is obtained from the PCFG for $\mathcal{D}$ by taking the square root of each transition probability, and then globally renormalising. Details of this procedure can be found in the long version (Fijalkow et al. 2021).

Parallel Implementations

Harnessing parallel compute environments is necessary for scalable, future-proof search algorithms, because combinatorial search bottlenecks on compute, and both the present

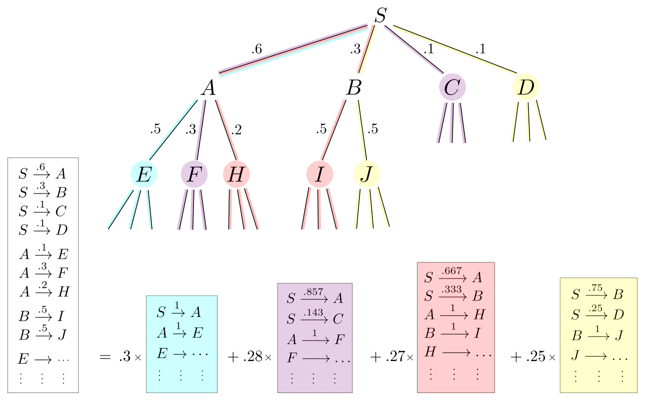


Figure 3: The grammar splitter: a balanced partition with imbalance $\alpha = \frac{.3}{.25} = 1.2$.

and likely future of massive compute is a parallel one. Accordingly, we have taken care to design and evaluate extensions of our algorithms which can metabolize these compute resources through multiprocessing.

Our key idea is to see the grammar as a tree and that to split the work we have to split the tree. We introduce a new algorithm called the *grammar splitter*, which partitions a PCFG into a balanced family of k sub-PCFGs. Each of the k threads is assigned a sub-PCFG and simply runs a search algorithm on it. Two key advantages of our approach are that any search algorithm can be used in this very simple parallel architecture, and that the theoretical gain of using k threads is linear in k. The output of the grammar splitter is illustrated in Figure 3: the white PCFG is split into 4 sub-PCFGs.

The two crucial properties of the grammar splitter are:

- the space of programs is partitioned into k subspaces. This implies that the threads do not carry out redundant work and that all programs are generated,
- the k program subspaces are balanced, meaning that their mass probabilities are (approximately) equal. This implies that all threads contribute equally to the search effort.

A split is a collection of partial programs, for instance `map (* 2) HOLE` and `fold + HOLE HOLE` are two such partial programs where `HOLE` is to be replaced by a program. A partial program induces a sub-PCFG. A split simply combines this collection of sub-PCFG according to the probabilities of the partial programs.

A set of k disjoint splits yields a partition of the PCFG. The probability mass of a split is the sum of probabilities of all programs that can be generated from a split. Let us write α for the imbalance of a partition, defined as the ratio between the maximum and the minimum probability mass of a split. We are looking for a balanced partition, *i.e.* one for which the imbalance α is close to 1.

Our algorithm finds a balanced partition through a hill climbing process: at each point the algorithm either looks for an improving swap or a refinement. In the first case, the action of an improving swap is to transfer a partial program from one split to another, and its goal is to lower the imbalance coefficient. In the second case, we consider the partial program with maximal probability in a split and refine it: for example `map (* 2) HOLE` could be replaced by `map (* 2) var0` and `map (* 2) (filter HOLE HOLE)`, thus obtaining more partial programs with smaller probability mass enabling new improving swaps.

Experiments

We study a range of search algorithms – both our new ones and prior work – across list processing and string manipulation domains, with the goal of answering the following questions:

- HEAP SEARCH and A^* are both loss optimal enumerative algorithms; beyond these theoretical guarantees, how do the two algorithms compare in practice?
- How effective are our search algorithms for solving complex program synthesis benchmarks using neural guidance?
- How do our algorithms scale with parallel compute?

We use a generic program synthesizer written from scratch in Python (see the long version (Fijalkow et al. 2021)), studying random PCFGs (more controlled) and machine-learned PCFGs (more naturalistic).

We report results on DSLs from DeepCoder (Balog et al. 2017) and DreamCoder (Ellis et al. 2021). Both target the classic program synthesis challenge of integer list processing programs, but with different properties. DeepCoder's DSL is larger and more specialized, with around 40 high-level primitives, and does not use polymorphic types, while DreamCoder's is smaller and more generic, with basic functional programming primitives such as map, fold, unfold, car, cons, and cdr, etc., for a total of around 20 primitives. Both DSLs are compiled into a CFG with minimal syntactic constraints generating programs of depth 6.

The search algorithms under consideration are:

- THRESHOLD from (Menon et al. 2013): iterative-deepening-search, where the threshold that is iteratively deepened is a bound on program description length (i.e. negative log probability),
- SORT AND ADD from (Balog et al. 2017): an inner loop of depth-first-search, with an outer loop that sorts productions by probability and runs depth-first-search with the top k productions for increasing values of k,
- A^* from (Feng et al. 2018): best-first-search on the graph of (log probabilities of) tree derivations,
- BEAM SEARCH from (Zhang et al. 2018): breadth-first-search with bounded width that is iteratively increased.

As well as our new algorithms: HEAP SEARCH and SQRT SAMPLING. We refer to the long version (Fijalkow et al. 2021) for a description of the algorithms and their implementations.

Our implementation of SQRT SAMPLING uses the Alias method (Walker 1977), which is an efficient data structure for sampling from a categorical distribution. We associate to each non-terminal an Alias table, reducing the task of sampling a derivation rule with n choices to sampling uniformly in $[1, n]$ and in $[0, 1]$.

All algorithms have been reimplemented and optimised in the codebase to provide a fair and uniform comparison. We also report on parallel implementations using our grammar splitter.

Random PCFGs

In this first set of experiments we run all search algorithms on random PCFGs until a timeout, and compare the number of programs they output and the cumulative probability of all programs output.

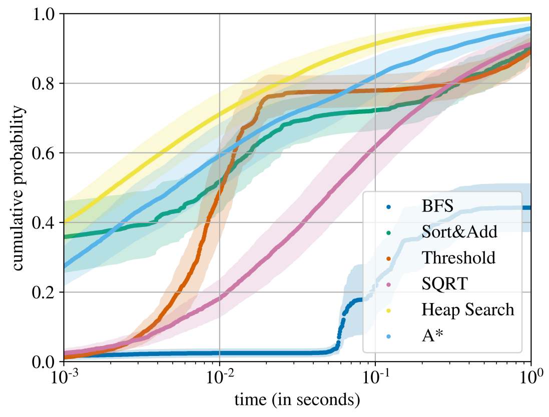
To obtain random PCFGs from the CFGs we sample a probabilistic labeling with an exponential decrease (this is justified by the fact that machine-learned PCFGs feature exponential decrease in transition probabilities). In this experiment the initialization cost of each algorithm is ignored. The results presented here are averaged over 50 samples of random PCFGs, the solid lines represent the average and a lighter color indicates the standard deviation. Details on the sampling procedure can be found in the long version (Fijalkow et al. 2021).

Figure 4 shows the results for all algorithms in a non-parallel implementation. On the lhs we see that HEAP SEARCH (almost always) has the highest cumulative probability against both time and number of distinct programs. Note that since A^* and HEAP SEARCH enumerate the same programs in the same order they produce the same curve in the rhs of Figure 4 so we did not include A^*.

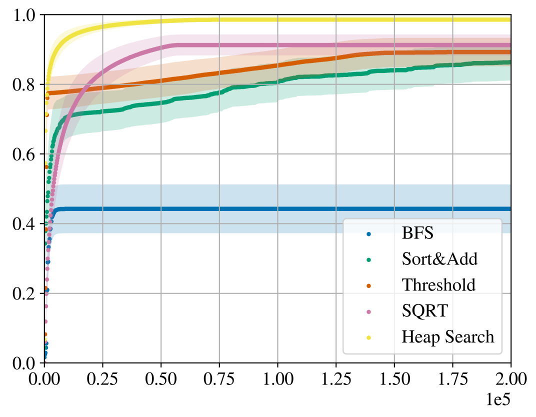
To compare A^* and HEAP SEARCH we refer to Figure 5, showing that HEAP SEARCH generates 2.35 times more programs than A^*, consistently over time. The larger variations for A^* are due to the manipulation of a single heap of growing size, requiring frequent memory reallocations. The difference in performance can be explained by the fact that A^* uses a single heap for storing past computations, while HEAP SEARCH distributes this information in a family of connected heaps and hash tables.

We then turn to parallel implementation and perform the same experiments using a variable number of CPUs for HEAP SEARCH and SQRT SAMPLING using the grammar splitter. We do not report on a baseline parallel implementation of SQRT SAMPLING which would simply sample using the same PCFG on multiple CPUs. Indeed this naive approach performs poorly in comparison, since thanks to the grammar splitter two CPUs cannot generate the same program.

The results are shown in Figure 6, where we count programs with repetitions. We see that for SQRT SAMPLING the scale-up is linear, and it is mostly linear for HEAP SEARCH with an acceleration from the 0.2s mark. This acceleration can be explained in two ways: first, each sub-PCFG is shallower since it is split thus it is faster to enumerate program from it, second, once all the successors have been computed HEAP SEARCH is a simple lookup table. At the end of the experiment, SQRT SAMPLING has generated 2.8 times more programs with 6 CPUs than with 2 CPUs,



(a) Cumulative probability against time in log-scale



(b) Cumulative probability against number of programs output

Figure 4: Comparing all search algorithms on random PCFGs

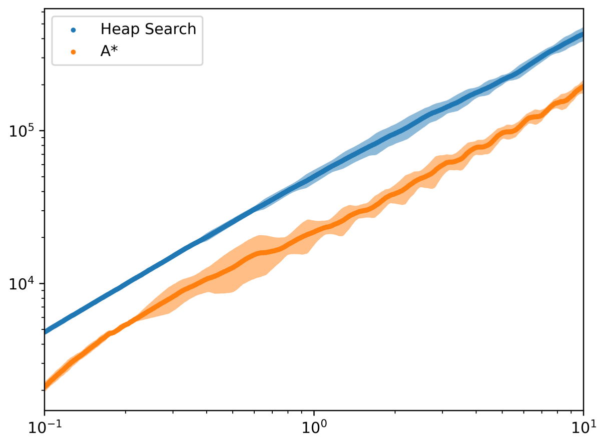


Figure 5: Comparing Heap Search and A^*

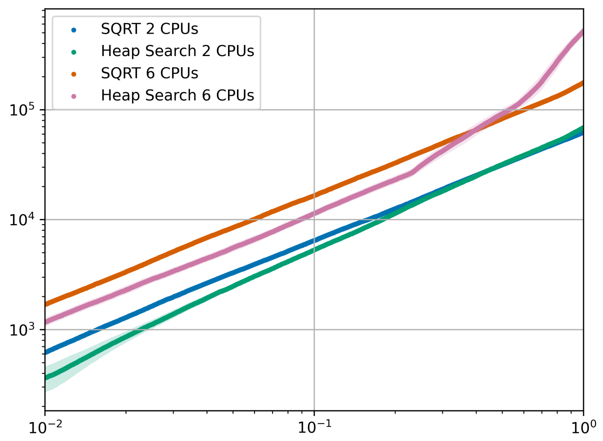


Figure 6: Parallel implementations of Heap Search and SQRT Sampling using the grammar splitter

whereas Heap Search has generated 7.6 times more programs with 6 CPUs than with 2 CPUs.

This experiment suggests that the grammar splitter enables us to scale our search on multiple CPUs with a linear speed up in the number of CPUs.

Machine-learned PCFGs

In this second set of experiments we consider the benchmark suites of hand-picked problems and sets of I/O. We extracted 218 problems from DreamCoder's dataset (Ellis et al. 2021). (The experiments can be easily replicated on DeepCoder's dataset (Balog et al. 2017) but we do not report on the results here.)

We train a neural network to make predictions from a set of I/O. Our neural network is composed of a one layer GRU (Cho et al. 2014) and a 3-layer MLP with sigmoid activation functions, and trained on synthetic data generated from the DSL. The details of the architecture and the training can be found in the long version (Fijalkow et al. 2021). Our network architecture induces some restrictions, for instance the types of the programs must be `int list -> int list`; we removed tasks that did not fit our restrictions and obtained a filtered dataset of 137 tasks. For each task we run every search algorithm on the PCFG induced by the neural predictions with a timeout of 100s and a maximum of $1M$ programs. Unlike in the previous experiments the initialization costs of algorithms are not ignored.

Figure 7 shows the number of tasks solved within a time budget. Heap Search solves the largest number of tasks for any time budget, and in particular 97 tasks out of 137 before timeout. The comparison between Threshold and A^* is insightful: A^* solves a bit more tasks than Threshold (85 vs 83) but in twice the time. SQRT Sampling performs just a bit worse than A^* despite being a sampling algorithm.

Table 1 shows for each algorithm how many programs were generated per second. Recall that Heap Search and

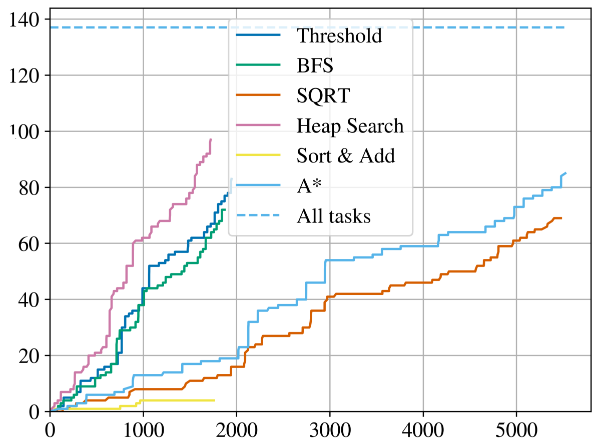


Figure 7: Comparing all search algorithms on the DreamCoder reduced dataset with machine-learned PCFGs

Algorithm	Number of programs generated
Sort & Add	75031 prog/s
Heap Search	38735 prog/s
Threshold	25381 prog/s
BFS	20980 prog/s
SQRT Sampling	14020 prog/s
A^*	6071 prog/s

Table 1: Number of programs generated

A^* generate the same programs in the same order. Overall in these experiments Heap Search is 6 times faster than A^*.

Since Heap Search follows a bottom-up approach we save on program evaluations in two ways: partial programs are evaluated along the search, and the results are cached. On the other hand A^* is a top-down enumeration method so every new program has to be evaluated from scratch.

It is interesting to compare the rates of SQRT Sampling and A^*: although SQRT Sampling generates over two times more programs, their overall performances are similar. This can be explained in two ways: SQRT Sampling may sample the same programs many times, while A^* enumerates each program once and starts with the most promising ones according to the predictions.

Sort & Add despite being the fastest only solves 4 tasks, showing that it does not generate relevant programs for the task at hand.

Discussion

Related Work

The idea of guiding program search via probabilities is an old one (Solomonoff 1989) but which recently has fast become a standard practice in the AI and program synthesis communities. To the best of our knowledge, practical program synthesizers that learn to use probabilistic predictions originated in (Menon et al. 2013), which was first extended with deep learning in (Balog et al. 2017), and such methods are now winning program synthesis competitions (Lee et al. 2018). To first approximation, such recent progress has drawn on advances in neural network architecture: e.g., early learned FlashFill-like systems (Parisotto et al. 2017) benefited from sophisticated attention mechanisms (Devlin et al. 2017), and procedural planning programs (Bunel et al. 2018) benefited from feeding execution traces into the neural net (Chen, Liu, and Song 2018). While prior works have explored novel test-time strategies, from Sequential Monte Carlo (Ellis et al. 2019) to ensembling methods (Chen, Liu, and Song 2018), here we sought a systematic empirical/theoretical study of two different families of inference strategies, which we intend to mesh well with the larger body of work on neural program synthesis. While the algorithms introduced here do not straightforwardly extend to neural autoregressive models (e.g. RobustFill (Devlin et al. 2017)), methods such as SQRT Sampling in principle apply to this setting too. We hope that our work here spurs the development of the right tricks to get the theoretical benefits of SQRT Sampling for these more flexible model classes, just as DeepCoder paved the way for RobustFill.

Shi, Bieber, and Sutton (2020) introduced Unique Randomizer in order to sample without replacement: it is a general technique effectively turning a sampling method into an enumerative one by updating the probabilistic weights along the search. It is further improved through batching via Stochastic Beam Search (Kool, Van Hoof, and Welling 2019). It is possible to combine the SQRT Sampling algorithm with the Unique Randomizer and Stochastic Beam Search. Our experiments did not yield interesting results in that direction, possibly because of memory allocation issues. We leave for future work to optimise this approach.

Contributions and Outlook

Learning to synthesize programs is a canonical neural-symbolic learning problem: training high capacity statistical models to guide the construction of richly structured combinatorial objects, such as programs. Yet while the neural side of this problem has received much deserved attention, the symbolic component is sometimes taken for granted–after all, symbolic program synthesis has received decades of attention. But the entrance of powerful neural networks for synthesizing programs forces us to reconsider how we deploy symbolic methods for program synthesis. We have considered both systematic and stochastic methods, from both theoretical angles (obtaining guarantees) and also engineering perspectives (such as how to parallelize our new techniques). We hope this work helps contribute to thinking through the symbolic search back-end in this more modern context.

References

Alur, R.; Bodík, R.; Juniwal, G.; Martin, M. M. K.; Raghothaman, M.; Seshia, S. A.; Singh, R.; Solar-Lezama, A.; Torlak, E.; and Udupa, A. 2013. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD*.

Alur, R.; Singh, R.; Fisman, D.; and Solar-Lezama, A. 2018. Search-based program synthesis. *Communications of the ACM*, 61(12).

Balog, M.; Gaunt, A. L.; Brockschmidt, M.; Nowozin, S.; and Tarlow, D. 2017. DeepCoder: Learning to Write Programs. In *International Conference on Learning Representations, ICLR*.

Bunel, R.; Hausknecht, M. J.; Devlin, J.; Singh, R.; and Kohli, P. 2018. Leveraging Grammar and Reinforcement Learning for Neural Program Synthesis. In *International Conference on Learning Representations, ICLR*.

Chen, X.; Liu, C.; and Song, D. 2018. Execution-guided neural program synthesis. In *International Conference on Learning Representations, ICLR*.

Cho, K.; van Merriënboer, B.; Gulcehre, C.; Bahdanau, D.; Bougares, F.; Schwenk, H.; and Bengio, Y. 2014. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. In *Conference on Empirical Methods in Natural Language Processing, EMNLP*.

Clymo, J.; Gascón, A.; Paige, B.; Fijalkow, N.; and Manukian, H. 2020. Data Generation for Neural Programming by Example. In *International Conference on Artificial Intelligence and Statistics, AI&STATS*.

Devlin, J.; Uesato, J.; Bhupatiraju, S.; Singh, R.; Mohamed, A.; and Kohli, P. 2017. RobustFill: Neural Program Learning under Noisy I/O. In *International Conference on Machine Learning, ICML*, volume 70 of *Proceedings of Machine Learning Research*.

Ellis, K.; Nye, M.; Pu, Y.; Sosa, F.; Tenenbaum, J.; and Solar-Lezama, A. 2019. Write, Execute, Assess: Program Synthesis with a REPL. In *Neural Information Processing Systems, NeurIPS*.

Ellis, K.; Wong, C.; Nye, M. I.; Sablé-Meyer, M.; Morales, L.; Hewitt, L. B.; Cary, L.; Solar-Lezama, A.; and Tenenbaum, J. B. 2021. DreamCoder: bootstrapping inductive program synthesis with wake-sleep library learning. In *International Conference on Programming Language Design and Implementation, PLDI*.

Feng, Y.; Martins, R.; Bastani, O.; and Dillig, I. 2018. Program synthesis using conflict-driven learning. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*.

Fijalkow, N.; Lagarde, G.; Matricon, T.; Ellis, K.; Ohlmann, P.; and Potta, A. 2021. Scaling Neural Program Synthesis with Distribution-based Search. *CoRR*, abs/2110.12485.

Gulwani, S.; Polozov, O.; and Singh, R. 2017. Program Synthesis. *Foundations and Trends in Programming Languages*, 4(1-2).

Kalyan, A.; Mohta, A.; Polozov, O.; Batra, D.; Jain, P.; and Gulwani, S. 2018. Neural-Guided Deductive Search for Real-Time Program Synthesis from Examples. In *International Conference on Learning Representations, ICLR*.

Kool, W.; Van Hoof, H.; and Welling, M. 2019. Stochastic beams and where to find them: The gumbel-top-k trick for sampling sequences without replacement. In *International Conference on Machine Learning, ICML*.

Lee, W.; Heo, K.; Alur, R.; and Naik, M. 2018. Accelerating search-based program synthesis using learned probabilistic models. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*.

Menon, A. K.; Tamuz, O.; Gulwani, S.; Lampson, B. W.; and Kalai, A. 2013. A Machine Learning Framework for Programming by Example. In *International Conference on Machine Learning, ICML*.

Parisotto, E.; Mohamed, A.; Singh, R.; Li, L.; Zhou, D.; and Kohli, P. 2017. Neuro-Symbolic Program Synthesis. In *International Conference on Learning Representations, ICLR*.

Polosukhin, I.; and Skidanov, A. 2018. Neural Program Search: Solving Programming Tasks from Description and Examples. In *International Conference on Learning Representations, ICLR*.

Shi, K.; Bieber, D.; and Singh, R. 2020. TF-Coder: Program Synthesis for Tensor Manipulations. In *Workshop on Computer-Assisted Programming, CAP*.

Shi, K.; Bieber, D.; and Sutton, C. 2020. Incremental Sampling Without Replacement for Sequence Models. In *International Conference on Machine Learning, ICML*.

Solomonoff, R. 1989. A system for incremental learning based on algorithmic probability. In *Israeli Conference on Artificial Intelligence, Computer Vision and Pattern Recognition*.

Walker, A. J. 1977. An Efficient Method for Generating Discrete Random Variables with General Distributions. *ACM Transactions on Mathematical Software*, 3(3).

Zhang, L.; Rosenblatt, G.; Fetaya, E.; Liao, R.; Byrd, W. E.; Urtasun, R.; and Zemel, R. S. 2018. Leveraging Constraint Logic Programming for Neural Guided Program Synthesis. In *International Conference on Learning Representations, ICLR*.

Zohar, A.; and Wolf, L. 2018. Automatic Program Synthesis of Long Programs with a Learned Garbage Collector. In *Neural Information Processing Systems, NeurIPS*.