EXECUTION-GUIDED WITHIN-PROMPT SEARCH FOR PROGRAMMING-BY-EXAMPLE WITH LLMS

Anonymous authors
Paper under double-blind review

ABSTRACT

Soundness is an important property in programming-by-example (PBE) as any learned program is expected to be correct for at least the examples that were part of the problem statement. This allows synthesizers to perform a search over a domain-specific language (DSL) that terminates when any sound program is found. Large language models (LLMs) can generate code from examples without being limited to a DSL, but they lack soundness guarantees (generated code is not even guaranteed to execute) and the concept of search (samples are independent). In this paper, we use an LLM as a policy that generates lines of code and then join these lines of code to let the LLM implicitly estimate the value of each of these lines in its next iteration. We further guide the policy and value estimation by executing each line and annotating it with its results on the given examples. This allows us to search for programs within a prompt until a sound program is found by letting the policy reason in both the syntactic (code) and semantic (execution) space. We evaluate this approach on five benchmarks across different domains, such as string transformations, list transformations, and arbitrary Python programming problems, showing that within-prompt search and execution allows us to sample better programs more consistently. Additionally, our experiments indicate that the model does behave like a policy and value.

1 INTRODUCTION

Automatically synthesising code from any form of intent or specification is considered as a grail of computer science (Gulwani et al., 2017). One particularly challenging specification are a set of inputs and their expected outputs, as the synthesizer has to not only detect a pattern between the input and the output, but also write code for it. Large language models (LLMs) (Brown et al., 2020; OpenAI, 2024) are trained to recognize patterns in text to predict the most likely next token. Their training data contains significant amounts of code (Xu et al., 2022) and examples of how to use that code, both through tutorials and unit tests.

Consider, for example, a prompt that consists of two `assert` statements and a function signature:

```
assert f(a = 'Program-Synthesis') == 'PS'
assert f(a = 'Large-Language-Model') == 'LLM'

def f(a):
```

Different versions of the `gpt` model series by OpenAI (`4o`, `4-turbo`, `35-turbo`) complete it with a variation on `return ''.join(i[0] for i in a.split('-'))` to make `f` satisfy the assertions.[1]

When the patterns become more complicated, however, the LLM often fails. A first reason is the model overfitting on a dominant (more common) pattern in the assertions. If we change the second input to `'Large--Language--Models'` we notice that `gpt-4o` does not change its answer, not realizing that splitting on `'-'` in a string with `'--'` results in empty strings. The model then only considers the syntactic space, rather than reasoning about the semantics of the program. A second reason is that iteratively sampling tokens does not allow the model to reconsider earlier code like a human would do. Once the model has written `return ''.join(i[0] for i in a.split())` —even if it would

[1] At temperature 0, which we use for all demonstrations.

now realize that `i` can be empty—it cannot easily[2] recover. In other words, the model is not able to *search* for a correct program, whereas search is indispensable in all symbolic synthesizers (Gulwani et al., 2017).

We tackle the limitation on search by sampling multiple lines of code, combining them into a single program, and implicitly letting the model choose which of these lines to continue from in a next iteration. We call this *within-prompt search* as one prompt contains all states explored so far. We tackle the limitation on reasoning about semantics by executing generated code and providing these executions as comments to the model. The model then has access to both the syntax and semantics of the program, and the within-prompt search is thus *execution-guided*.

Considering the previous example and appropriately prompting the model to generate options for the next line of code, the model generates three unique statements:

`v1 = a.split('-')` `v1 = a.split('-')[0][0]` `v1 = a[0]`

These statements are de-duplicated and executed to yield a new prompt for the next iteration:

```
assert f(a = 'Program-Synthesis') == 'PS'  # e1
assert f(a = 'Large--Language--Model') == 'LLM'  # e2

def f(a):
    v1 = a.split('-')  # {'e1': ['Program', 'Synthesis'],
                       #  'e2': ['Large', '', 'Language', '', 'Model']}
    v2 = a[0]  # {'e1': 'P', 'e2': 'L'}
    v
```

The model now completes it with `v3 = ''.join((word[0] for word in v1 if word))`.

Relating this to existing neural program synthesizers, we observe that the model acts as both a policy—by generating candidate lines—and a value—by choosing which previous lines to consider the result of—by looking at both syntax and semantics of generated code. Core differences are that it is not limited to a domain-specific language or even a specific domain—the same method can be used on string transformations (Gulwani, 2011) and lists (Rule et al., 2024) without any intervention—and does not require any training.

We evaluate the effect of within-prompt search and execution-guidance on straight-line program synthesis on five benchmarks spanning three domains (string transformations, list functions, and generic Python programming problems). We provide insights in how the model performs as a policy and value, evaluate the effect of more diversity in sampling lines of code, and compare with a baseline of unrestricted synthesis.

In summary, we make the following contributions:

- We leverage pre-trained language models to perform programming-by-example without restrictions on the domain (specific language).
- We propose execution-guided within-prompt search, which allows the model to reason about the semantics while exploring different candidate programs.
- We evaluate our approach on five datasets across various domains to highlight generality of our approach, and show that execution-guidance and search both improve pass@1, and that combining them further augments these results.

2 Background

Given n input-output examples $E = \{(\mathbf{x}_i, y_i)\}_{i=1}^{n}$ the goal of programming-by-example (PBE) is to find a program P such that $P(\mathbf{x}_i) = y_i$ for all i. Typically, only a small subset of examples $E_g \subset E$ is given. We say that P is sound if $P(\mathbf{x}_i) = y_i$ for all $(\mathbf{x}_i, y_i) \in E_g$ and we write $P \models E_g$.

[2]It could still add a `if '--'not in a else ...` to the end of the line, but we never witnessed this.

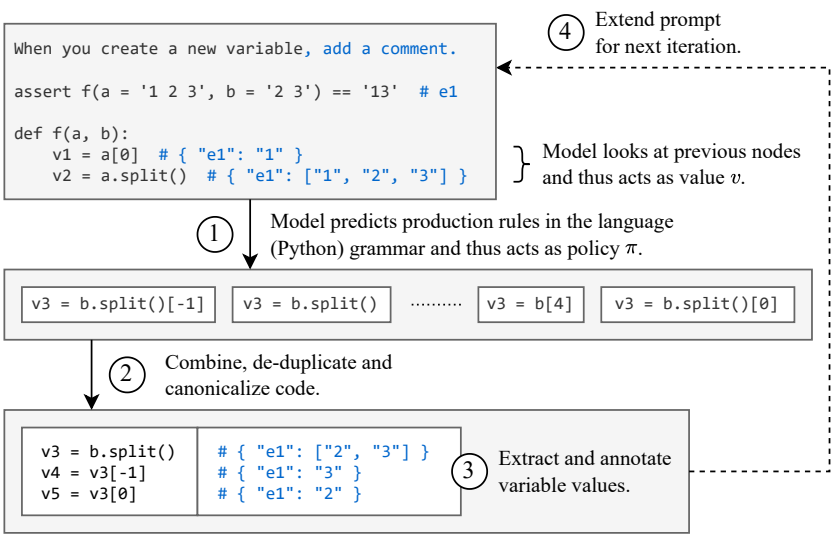


Figure 1: Overview of our approach for PBE with large language models. ① The model acts as a policy by predicting next lines of code. ② We combine, de-duplicate and canonicalize that code in order for the model to act as a value predictor. ③ We execute code, extract values and add them as comments. ④ We extend the prompt with the newly generated code.

PBE with search Constraining P to a domain-specific language (DSL) $\mathcal{L}$ defined by a context-free grammar allows us to easily—and sometimes even efficiently—search for programs. FlashFill, likely the most popular application of PBE, traverses its grammar top-down by breaking down the original problem into smaller sub-problems and recursively solving them (Gulwani, 2011). Recent neural approaches perform a bottom-up search by using a policy network to predict one or more production rules to follow, which either initialize a new sub-program (terminal) or combine two subprograms (non-terminal) (Parisotto et al., 2017; Balog et al., 2017; Ellis et al., 2018; 2021; Odena et al., 2021; Chen et al., 2018). Optionally, a separate value network can be used to prioritize or filter candidates in subsequent iterations Ellis et al. (2019).

PBE with execution Since each node in a bottom-up search is a program without free variables, it can be executed, and those execution results are a powerful signal for predictions in subsequent search steps. One approach is to add the current execution state as an additional input to the model (Ellis et al., 2019). Another approach is to use the execution of a candidate program $P'(\mathbf{x}_i)$ to define a new problem specification $\{(P'(\mathbf{x}_i), y_i) \mid (\mathbf{x}_i, y_i) \in E_g\}$ for the next iteration (Chen et al., 2018).

PBE with large language models Instead of searching over a grammar, some neural approaches generate programs by having the model directly predict tokens from the alphabet of the DSL (Devlin et al., 2017; Bunel et al., 2018). Pre-trained language models also auto-regressively generate tokens and typically see a lot of code during training (OpenAI, 2024). They can be prompted or fine-tuned (Li & Ellis, 2024) to generate programs by example, without being limited to a DSL.

3 Execution-guided semantic decoding and search

We propose to use large language networks as both the (explicit) policy and (implicit) value network to search for code in a bottom-up way, and inject the execution results of that code to guide the model towards better predictions. An overview of this approach is shown in Figure 1. The following sections describe (1) how the model acts as a **policy**, (2) how to prepare that code for the model to implicitly consider the **value** of each line, (3) how to **guide** these results by executing the generated code, and (4) how this constitutes an in-prompt **search**.

3.1 POLICY: PREDICTING LINES OF CODE

Consider a function $P(\mathbf{x}) = [o_1, \ldots, o_n]$ where each operation $o_i(o_{<i}, \mathbf{x})$ is a function of the outputs of the previous operations $o_{<i}$ and the input $\mathbf{x}$ that returns a single value (denoted as o_i). The value of o_n is the return value. An operation can be drawn from the grammar of any programming language.

Encoding the specification given by E_g and a partial program $P_j = [o_1, \ldots, o_j]$ in a prompt, we can sample a new operation $o_{j+1} \sim p_{\text{LLM}}(\cdot \mid E_g, P_j, t)$ from the LLM with t the sampling temperature (Ackley et al., 1985). If the temperature is set to 0, the model acts as a deterministic policy that maps each state (E_g, P_j) to its most likely next operation o_{j+1}. For higher temperatures, it acts as a stochastic policy that maps (E_g, P_j) to a distribution of operations.

Instead of a learned policy π, the model is prompted to find $\arg\max_{o_j \sim \pi} [Q^\pi(o_j, P_{j-1})]$ where the value $Q^\pi(o_j, P_{j-1})$ of considering an operation o_j in program P_{j-1} is the expectation that it can be completed into a valid program, or $Q(o_j, P_{j-1}) = \mathrm{E}_{P_\tau \sim \pi} [P_{j-1} \circ o_j \circ P_\tau \models E_g]$ where P_τ is a sequence of operations sampled from the (prompted) policy.

3.2 VALUE: COMBINING LINES OF CODE

Suppose we sample k operations $O_{j+1} = \{o_{j+1}^1, \ldots, o_{j+1}^k\}$ from the language model with $t > 0$. Instead of heuristically picking one operation $o_{j+1}^* \in O_{j+1}$ to extend P_j before sampling the next operation—writing such heuristic is hard even without considering the different types of values that each operation can return—we set $P_{j+1} = f_C(P_j \circ O_{j+1})$ and let the model act as that heuristic in the next iteration (where $\circ$ denotes operation concatenation). Here, f_C is a combination function that de-duplicates and (optionally) canonicalizes the given operations.

Instead of a single program, P_j then represents all programs explored in all iterations $\leq j$, where each operation can be considered as its own root node. When writing an operation o_{j+1}, the model needs to select some nodes $o_i \in P_j$ to continue from. Ideally, it selects operations that are expected to lead to a correct program by implicitly considering $V^\pi(o_i) = \mathrm{E}_{o \sim \pi} [Q^\pi(o, P_i)]$ to estimate the expected value of each operation $o_i \in P_j$.

3.3 GUIDANCE: EXECUTING CODE

Understanding the execution semantics of P_j is crucial when deciding on a next operation o_{j+1}—especially since the specification E_g is in this semantic space. Previous work (Ellis et al., 2019) has shown that executing code $[\![P_j]\!] = \{P(\mathbf{x}) \mid \mathbf{x} \in E_g\}$ on the given examples E_g and conditioning the policy and value on $([\![P_j]\!], E_g)$ instead of (P_j, E_g) allows them to make better decisions in this semantic (*execution*) space. The intuition is to mimic a human developer using code printing statements to evaluate their progress so far, and then make decisions based on that progress.

We extend this idea to our prompted policy and, as LLMs have shown strong code understanding capabilities, additionally allow the model to make decisions based on both the semantic and syntactic representation of P_j. Instead of only the final output $[\![P_j]\!]$, we obtain the value of each operation $o \in P_j$ on the given examples and add $[\![O_j]\!] = \{\{o(\mathbf{x}) \mid \mathbf{x} \in E_g\} \mid o \in O_j\}$ to the condition of our policy—to the prompt.

3.4 SEARCH

Sampling multiple operations O_j and combining them into a new state $(E_g, P_{j+1}, [\![O_{j+1}]\!])$ allows us to perform a within-prompt search. Each state (program) in the search is combined into a single *super*state, which the model can expand. We stop the search when $o_l \models E_g$ for any $o_l \in P_j$ and remove all operations that are not used by o_l. In theory, we can keep expanding the state until the context window of the LLM is reached and rely on *implicit* backtracking where the (implicit) value function simply ignores operations that cannot lead to a valid program. In practice, we can *explicitly* backtrack, for example, by pruning operations $o \in O_j$ that are not used in any of the k most recent iterations $[O_{j-k}, \ldots, O_j]$.

4 EXPERIMENTS

We evaluate the effect of execution and search, show insights in how the model behaves as a policy and value, and describe other properties of the search.

4.1 IMPLEMENTATION DETAILS

We use Python as the programming language in which we synthesize programs.

Prompt The prompt consists of four parts: (1) an instruction, (2) a single static example, (3) a set of assertions on a function `f`, and (4) the function `f` itself. The instruction instructs to generate code without control flow statements, that any function from the standard library can be used, that it can use variables as a scratchpad, and to create a comment each time a new variable is initialized. The example contains one line that is not needed. We initialize each line with "`v`" to ensure that the model generates a straight-line program with variables (`v1`, `v2`, ...) as the model tokenizes digits separately—`v1` is always broken down into `v` and `1`. A full example is shown in the Appendix.[3]

Sampling operations As the model is instructed to generate each statement on a single line, we can sample a single operation by using the newline character "`\n`" as a stopping token. Operations that do not parse or execute are skipped.

Canonicalization After ensuring that each generated line of code assigns its value to a unique variable, we concatenate these lines and perform global canonicalization steps. All comparisons are made on the abstract syntax tree (AST) level.

1. For every assignment `v = r`, we replace any occurrence of `r` in another operation with `v` and ensure that the lines remain topologically sorted.
2. For every assignment `v = r` where `r` is a `generator`, which does not have a pretty printing implementation due to lazy execution semantics, we wrap it in a `list` call as `v = list(r)`.
3. We remove any duplicate lines of code based on their execution results.
4. A return statement `return r` is converted to `v = r` with `v` a unique variable name.

Execution and comments We assign assertion `assert f(x1) == y1` a unique identifier `e1` as a comment "`# e1`". This allows us to easily anchor the execution result of each operation `v = r(x)` on to that identifier in a dictionary by mapping `{ "e1": r(x1) }`. Ensuring that each value can be printed is done in the canonicalization step to not have a disconnect between the operation and the commented values.

Search The search stops when any of the variable results matches the assertions or when a limit on the number of iterations is reached.

4.2 EXPERIMENTAL SETUP

Datasets We use five popular datasets that span different domains. No changes are required to apply our approach to these different datasets.

- **PROSE** (Microsoft) is a set of 354 benchmarks originally used to evaluate FlashFill Gulwani (2011)—the first widely adapted application of PBE. We filter it down to 332 non-trivial problems (no empty values, three or more examples).
- **SyGuS** (Alur et al., 2019) is a set of 205 benchmarks from the string transformation track of the SyGuS Alur et al. (2013) competition.
- **Playgol** (Cropper, 2019) is a set of 327 benchmarks from inductive logic programming.
- **Lists** (Rule et al., 2024) is a dataset of 251 list understanding problems, where each task transforms a list of integers.

[3] See the supplementary material.

- **MBPP** (Austin et al., 2021) is a dataset of basic Python problems, where each problem consists of both a description in natural language and a set of assertions. These assertions are given as part of the input, mostly to provide any synthesizer with information about the expected signature of the function to be synthesized. We only use the assertions, remove any that are not simply evaluating `f(x) == y` with `x` $\in$ `x` and `y` constants that can be evaluated with `eval` without additional imports and that are `json` serializable (no `set`, no `datetime`). Across the train and test set, this leaves us with 382 problems.

In order to save on cost of prompting for experiments, we filter *trivial* benchmarks by using a simple prompt conditioned on the first example $(\mathbf{x}_1, y_1) \in E$ to sample five programs P^i at $t = 0.6$ and do not include a benchmark if $P^i \models E$ for all of them. This leaves 238 benchmarks from PROSE, 94 benchmarks for SyGus, 170 benchmarks for Playgol, 211 for Lists, and 268 for MBPP.

Hyperparameters In all experiments, we sample $k = 4$ operations at each line at a temperature of 0.6, which is a common value that achieves a nice balance between exploration and exploitation. The model is `gpt-4o`. We set the maximal number of iterations to 8.

Baselines We consider two baselines to evaluate the effect of execution and search: an unrestricted baseline (*Free*) where all instructions related to straight-line programs and execution are removed, and a baseline where we do instruct the model to generate straight-line programs and sample a whole function from the model (*Straight*). Both prompts are shown in the Appendix.

Metrics We use the *pass@1* rate—the probability of one sample from the model being the correct one—computed over 8 completions for baselines without search and execution, and over 3 iterations for baselines with search or execution (where each iteration counts as a sample from the model in streaming mode).

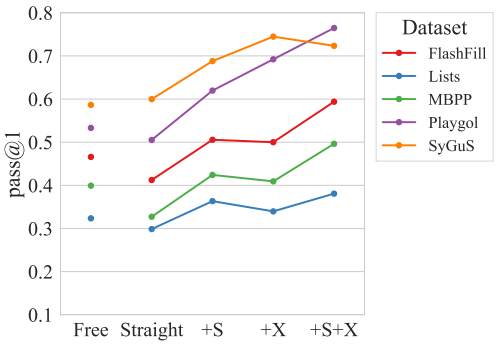
4.3 Results

Search and execution sample better programs Figure 2a shows the pass@1 of unrestricted and straight-line baselines, as well as adding search (+ S), execution (+ X) and both (+S+X) to straight-line program generation. Restricting the model to straight-line programs hurts performance, as it restricts the ability to rely on known patterns and involves a better understanding of the generated code. Adding search and execution both undo this effect. Together, their effect is amplified, with performance improvements going from +4% on Lists (the hardest benchmark) to +23% on Playgol with respect to unrestricted generation.

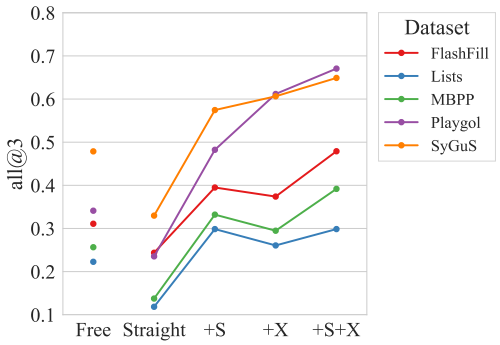
Search and execution improves consistency Figure 2b shows the probability of 3 answers on a problem being all correct as $\binom{c}{3}/\binom{t}{3}$ with t number of answers (3 or 8) and c number of correct answers (all@3). Search and execution ensure that the generations for any given problem are more consistent: the proportion of cases where all generations are correct improves even more, from +7.7% on Lists to +33% on Playgol. The model is less susceptible to poor sampling when having access to execution results, and can recover from a poor sample because of search.

Within-prompt search works in parallel Figure 3 shows the number of iterations required to find a successful program of a given length. Most of the weight is above the identity line, where the final program uses two or more nodes from the same iteration at least once. The stochastic policy thus allows within-prompt search to consider different leaf nodes in parallel.

Within-prompt search can backtrack Cases below the line in Figure 3 has iterations without any effect: either the policy failed to generate a (valid) new line or the value backtracked and completely ignored the results from one iteration. Figure 4 compares the number backtracks to the inputs in each iteration—when an operation o_i is sampled that only refers to the inputs $\mathbf{x}$ and no other operations $o_{<i}$—for within-prompt search with (WPS + X) and without (WPS) execution. Without execution, the model backtracks less, as it does not realize that the current operations $o_{<i}$ are dead ends, and as such performs worse (see Figure 2).



(a) Execution (X) and search (S) both improve the performance of programming by example with LLMs. Together, the improvements are even more significant.



(b) Execution (X) and search (S) make the model more consistent, as the number of cases where all 3 generations are correct increases significantly.

Figure 2: Results of unrestricted (Base) and straight-line code generation (Line) with search (+S), execution (+X) and both (+S+X). Forcing straight-line programs hurts performance, as it restricts the different ways of solving the problem. Both search and execution undo this effect and help the model in generating more correct code. Combining search and execution further significantly improves results—from +4% on Lists to +23% on Playgol in pass@1 (overall performance) and +7% on Lists to +33% on Playgol in the proportion of pass@1 rates that are one (consistency).

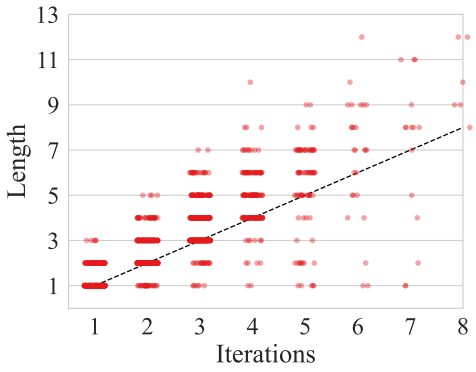


Figure 3: Length of successful programs versus the iteration in which they were found. Above the line, the policy returned multiple operations that were used in the final program. Below the line, the policy either failed to generate any new results or the value backtracked and completely ignored the results from one iteration.

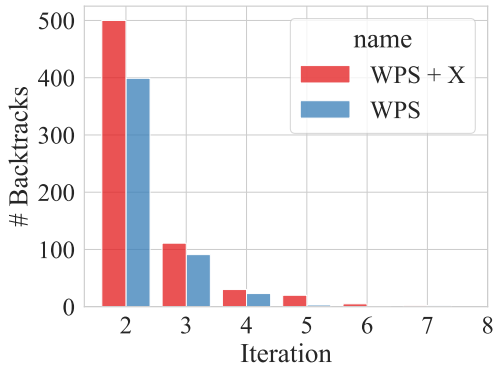
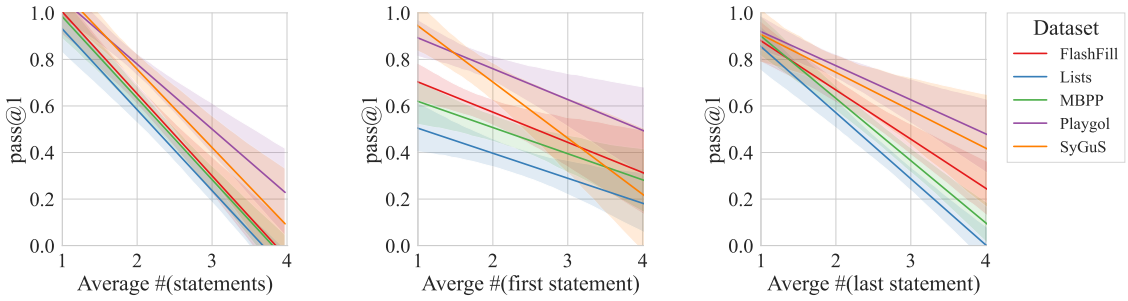


Figure 4: Number of backtracks to the inputs—an operation is sampled that does not reference any other operation—over different iterations without execution (WPS) and with execution (WPS + X) for successful programs. With execution, the value backtracks more often which leads to more successful solutions.

Model as policy Figure 5 shows the pass@1 rate in function of the average number of statements (all, first and last) sampled by the policy. There is a clear negative correlation: if the policy is certain and samples with less diversity, the model is more likely to solve a problem in the end. Conversely, if the policy samples many diverse lines, it is less likely to solve the problem in the end. This indicates that the model does behave like a policy.

Model as value Figure 6a compares the default policy (sample $k = 4$ operations) with a policy that keeps on sampling until we have $k = 4$ syntactically unique operations. Having more possibilities requires the model to act as a better value function. For 4 out of 5 datasets, the performance slightly increases, reinforcing our intuition that the model correctly picks which option to continue from. Only on Playgol, the easiest dataset, the added diversity is not required. Figure 6b shows the number of operations in the final program that are not required by the operation $o_l \models E_g$ and were not added



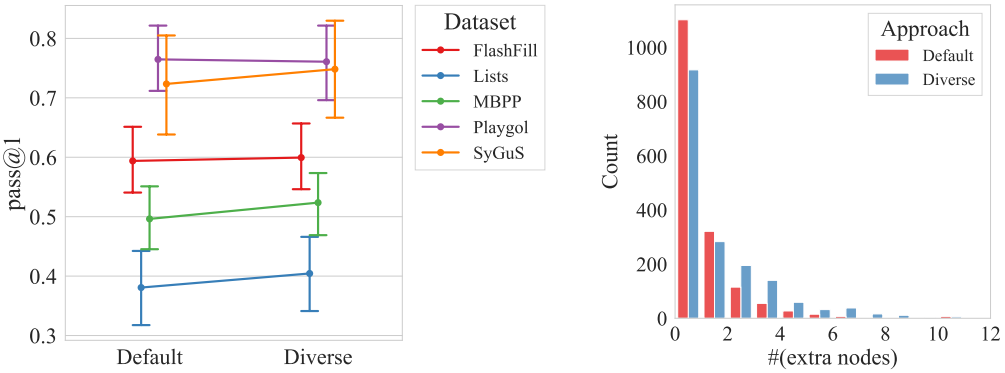
(a) pass@1 for average # of operations sampled over all iterations.

(b) pass@1 for average # of operations sampled in first iteration.

(c) pass@1 for average # of operations sampled in last iteration.

Figure 5: The pass@1 rate is inversely correlated with the number of unique operations sampled in all, first and last iterations. The first operation (b) is less important, as there is still room to recover.

in the last iteration, for both the default and diverse policy. As expected, the diverse policy expands the search (which explains the higher pass@1 rate) at the cost of creating more unused nodes. When the diverse policy does not create unused nodes, the syntactically unique nodes either do not execute, or they are semantically equivalent.



(a) Comparing the performance between the default policy and the diverse policy. Getting more diverse operations can increase the performance if the model correctly acts as a value—which it does for 4 out of 5 datasets. The Playgol dataset is considered easy, which means that this diversity is not required.

(b) Number of operations not used in the final program (without the final iteration) using the default or diverse policy for successful programs. If 0, the model only generated useful operations, which means that syntactically unique nodes were either invalid or semantically equivalent. Getting more diverse operations means that more nodes are not needed and should thus be ignored by the value.

Figure 6: Comparing the default policy with a diverse policy that keeps sampling until $k = 4$ syntactically unique operations are found.

5 Related Work

Programming-by-example with search Two popular search-based strategies for programming-by-example approaches are *bottom-up* synthesis and *top-down* synthesis (Gulwani et al., 2017). In forward synthesis, the search space is traversed by combining subprograms into more complex programs through enumeration (Alur et al., 2017) or enumeration with priority defined by a neural network (Odena et al., 2021; Ellis et al., 2021) or through an incomplete search powered by neural network heuristics (Ellis et al., 2019; Shi et al., 2022; 2023).

Programming-by-example with language models Without search, auto-regressive neural networks can also be trained to output the program token by token (Devlin et al., 2017; Bunel et al.,

2018). This is particularly interesting with large, pre-trained language models, which can be fine-tuned on synthetic data (Li & Ellis, 2024). Models that are large enough to exhibit in-context learning abilities, such as the GPT family (Brown et al., 2020; OpenAI, 2024), can program by example through prompts. One can use chain-of-thought prompting too by first asking the model to describe the transformation problem in natural language (Wang et al., 2024).

6 Conclusion

We introduce execution-guided within-prompt search for programming-by-example with LLMs, where the premise is to sample multiple atomic operations, combine them into a single program, and repeat the process from that combined program. Sampling multiple operations, where the model acts as a policy that explores different next states, allows us to perform a search. Combining operations into a single program enables the search to happen within one expanding prompt. This allows the model to act as a value function that evaluates which of these operations are promising candidates to extend in next iterations. Our results show that both search and execution help the model in generating straight-line programs, an effect that is further reinforced when they are combined.

References

David H Ackley, Geoffrey E Hinton, and Terrence J Sejnowski. A learning algorithm for boltzmann machines. *Cognitive science*, 9(1):147–169, 1985.

Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. *Syntax-guided synthesis*. IEEE, 2013.

Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. Scaling enumerative program synthesis via divide and conquer. In *International conference on tools and algorithms for the construction and analysis of systems*, pp. 319–336. Springer, 2017.

Rajeev Alur, Dana Fisman, Saswat Padhi, Rishabh Singh, and Abhishek Udupa. Syntax-guided synthesis competition. `https://github.com/SyGuS-Org/benchmarks/tree/master/comp/2019/PBE_SLIA_Track`, 2019.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. In *International Conference on Learning Representations*, 2017.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 1877–1901, 2020.

Rudy Bunel, Matthew Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. Leveraging grammar and reinforcement learning for neural program synthesis. In *International Conference on Learning Representations*, 2018.

Xinyun Chen, Chang Liu, and Dawn Song. Execution-guided neural program synthesis. In *International Conference on Learning Representations*, 2018.

Andrew Cropper. Playgol: learning programs through play. In *International Joint Conference on Artificial Intelligence*, 2019.

Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o. In *International conference on machine learning*, pp. 990–998, 2017.

Kevin Ellis, Lucas Morales, Mathias Sablé-Meyer, Armando Solar-Lezama, and Josh Tenenbaum. Learning libraries of subroutines for neurally–guided bayesian program induction. *Advances in Neural Information Processing Systems*, 31, 2018.

Kevin Ellis, Maxwell Nye, Yewen Pu, Felix Sosa, Josh Tenenbaum, and Armando Solar-Lezama. Write, execute, assess: Program synthesis with a repl. *Advances in Neural Information Processing Systems*, 32, 2019.

Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B Tenenbaum. Dreamcoder: Bootstrapping inductive program synthesis with wake-sleep library learning. In *International Conference on Programming Language Design and Implementation*, pp. 835–850, 2021.

Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Principles of Programming Languages*, pp. 317–330, 2011.

Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.

Wen-Ding Li and Kevin Ellis. Is programming by example solved by llms? *arXiv preprint arXiv:2406.08316*, 2024.

Microsoft. PROSE public benchmark suite. `https://github.com/microsoft/prose-benchmarks/tree/main/Transformation.Text`. Accessed: 2010-09-30.

Augustus Odena, Kensen Shi, David Bieber, Rishabh Singh, Charles Sutton, and Hanjun Dai. Bustle: Bottom-up program synthesis through learning-guided exploration. In *International Conference on Learning Representations*, 2021.

OpenAI. Gpt-4 technical report, 2024.

Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis. In *International Conference on Learning Representations*, 2017.

Joshua S Rule, Steven T Piantadosi, Andrew Cropper, Kevin Ellis, Maxwell Nye, and Joshua B Tenenbaum. Symbolic metaprogram search improves learning efficiency and explains rule learning in humans. *Nature Communications*, 15(1):6847, 2024.

Kensen Shi, Hanjun Dai, Kevin Ellis, and Charles Sutton. Crossbeam: Learning to search in bottom-up program synthesis. In *International Conference on Learning Representations*, 2022.

Kensen Shi, Hanjun Dai, Wen-Ding Li, Kevin Ellis, and Charles Sutton. LambdaBeam: Neural program search with higher-order functions and lambdas. In *Conference on Neural Information Processing Systems*, 2023.

Ruocheng Wang, Eric Zelikman, Gabriel Poesia, Yewen Pu, Nick Haber, and Noah Goodman. Hypothesis search: Inductive reasoning with language models. In *International Conference on Learning Representations*, 2024.

Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pp. 1–10, 2022.