Exploring Probe-Style Grammar Updates in Budgeted Search Program Synthesis

Chris Preda[1]

Supervisor: Dr. Sebastijan Dumančić[1]

[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
January 27, 2026

Name of the student: Chris Preda
Final project course: CSE3000 Research Project
Thesis committee: Dr. Sebastijan Dumančić, Jesper Cockx

An electronic version of this thesis is available at http://repository.tudelft.nl/.

Abstract

A key problem in program synthesis is that the search space is too big to traverse without a tactic. Budgeted search, specifically the Probe algorithm, is a promising one. Unfortunately Probe is treated as a monolith, instead of a collection of design choices. We identify these choices in the Herb-Probe algorithm and implement it in a budgeted search framework. Through its evaluation we learn more about Probe's design and create a framework for easy budgeted search comparison and hybridization.

1 Introduction

Program synthesis is the task of automatically constructing programs that satisfy a specification given by a user, often expressed using I/O examples[2]. Using a grammar, building block sized code pieces, the program tries to search through all combinations until it finds one that succeeds. As you can imagine the time it takes to find them increases as problems get more complex.

A promising strategy to address this challenge is budgeted search. Instead of tackling the problem as one single search, we perform multiple smaller ones which we give a limited budget. Between attempts we modify the search in some way as we try to learn something from our smaller attempt.

One influential approach in this space is the Probe algorithm[1]. Probe introduces a search technique called Guided Bottom Up search, where we assign probability values to each grammar rule and iterate through the possible programs in order of most to least likely to be the solution. We learn these probabilities between attempts using a technique called just in time learning.

While the algorithm is successful, Probe is often treated as a monolith instead of a collection of design choices. Think of selection strategies for promising programs, grammar update rules, budget allocations for each attempt, etc. It therefor is unclear which aspects of Probe are fundamental to its design, and which are implementation specific choices. Moreover, there exists little systematic evaluation of how these components behave when transferred to different synthesis frameworks or search architectures.

This work addresses this gap by reinterpreting Probe as a set of modular design principles rather than a fixed algorithm. We implement a Probe-style synthesis system, Herb-Probe, within a shared budgeted search framework built in Herb.jl[3]. This framework provides a common controller structure that enables consistent comparison between different budgeted synthesis strategies. Within this setting, we systematically vary key components of the Probe methodology, such as selection rules, grammar update rules, and budget allocation strategies to study their individual and combined effects on synthesis performance.

The research question that was formulated for this paper is the following:

- How do we effectively leverage Probe-Style grammar updates, to improve program synthesis performance in Budgeted Search?

We try to answer this question using the following subquestions:

- **RQ1:** *Does increasing the total budget of a Herb-Probe run without any cycles lead to an increasing number of solved benchmarks?*
- **RQ2:** *At a fixed total budget, do increased Herb-Probe cycles solve more benchmarks than a single search?*
- **RQ3:** *Does the specific selector policy of Herb-Probe notably affect success?*
- **RQ4:** *Does Probe's original grammar update rule remain competitive for Herb-Probe?*
- **RQ5:** *Can the shared budgeted search framework successfully compare multiple differing implementations?*

By framing Probe as a collection of transferable design principles rather than a fixed algorithm, this work aims to clarify which components are essential for effective grammar learning in synthesis and which are implementation-dependent. In addition, the proposed shared framework establishes a foundation for future comparative research on budgeted program synthesis, enabling systematic evaluation and hybridization of these search strategies.

2 Background

This section introduces the foundational concepts and prior work necessary to understand the remainder of this paper. In particular, we discuss program synthesis, grammar-based search spaces, budgeted search, and the Probe algorithm. All methods described here are taken from prior work; modifications and extensions introduced in this thesis are described later in Section 5.

2.1 Program Synthesis

Program synthesis is the task of automatically constructing a program that satisfies a given specification. In many practical settings, this specification is provided as a finite set of input–output examples, also referred to as programming-by-example (PBE)[**?**]. A synthesized program is considered correct if it produces the desired output for all provided inputs.

Formally, given a specification

$$\mathcal{E} = \{(x_1, y_1), \ldots, (x_n, y_n)\},$$

the goal of synthesis is to find a program p such that

$$p(x_i) = y_i \quad \forall (x_i, y_i) \in \mathcal{E}.$$

The search space of candidate programs is typically defined by a **Context-Free Grammar (CFG)**. A CFG consists of a set of terminal symbols, non-terminal symbols, and production rules that describe how programs can be composed. The grammar implicitly defines an (often infinite) space of programs, which must be searched efficiently.

Search Strategies

Given a grammar, program synthesis can be viewed as a structured search problem. Common strategies for traversing the search space include **bottom-up** and **top-down** enumeration[**?**].

In **bottom-up enumeration**, the search begins with the simplest expressions—typically terminals or constant expressions—and gradually constructs larger programs by applying grammar rules. Programs are often enumerated in order of increasing size or cost. This approach has the advantage that smaller programs are explored first, but it can suffer from combinatorial explosion.

In contrast, **top-down enumeration** starts from the start symbol of the grammar and incrementally expands non-terminals until complete programs are formed. The order of expansion can be guided by heuristics, priorities, or probabilities associated with grammar rules.

Enumerative approaches are conceptually simple and complete, but without guidance they scale poorly as the grammar or target programs become more complex.

2.2 Probabilistic Grammars

To guide the search more effectively, grammars can be augmented with probabilities, resulting in a **Probabilistic Context-Free Grammar (PCFG)**[1]. In a PCFG, each production rule r_i is assigned a probability $P(r_i)$ such that for each non-terminal symbol, the probabilities of its outgoing rules sum to one.

Given a program p composed of a sequence of grammar rules $\{r_1, \ldots, r_k\}$, its probability is defined as

$$p(p) = \prod_{i=1}^{k} P(r_i).$$

2.3 Budgeted Search

A key challenge in program synthesis is that exhaustive search is rarely feasible within realistic time constraints. **Budgeted search** addresses this issue by dividing the search process into multiple bounded attempts rather than performing a single unbounded search.

Each attempt is constrained by a predefined budget, such as a maximum number of enumerated programs or a time limit. After each attempt, information gathered during the search can be used to modify the search strategy before restarting. Budgeted search thus provides a natural framework for iterative learning and refinement.

2.4 The Probe Algorithm

Probe is a program synthesis algorithm that combines probabilistic grammars with budgeted search to efficiently explore large program spaces [1]. Its central idea is to iteratively learn a probability distribution over grammar rules that increases the likelihood of enumerating correct programs early.

Probe consists of two main components:

1. Guided bottom-up search[1]
2. Just-in-time learning of grammar probabilities[1]

Guided Bottom-Up Search

Probe performs a variant of bottom-up enumeration known as **size-based bottom-up search**. Rather than enumerating programs strictly by derivation depth, programs are generated in increasing order of size, where size typically corresponds to the number of nodes in the abstract syntax tree.

The search is guided by a probabilistic grammar: programs are prioritized according to their probability under the current PCFG. Intuitively, programs that are more likely according to the grammar are explored earlier.

During search, Probe maintains several data structures, including:

- A program bank containing generated expressions
- An evaluation cache to avoid redundant executions
- A set of partial solutions that satisfy some, but not all, examples

Programs are evaluated against the specifications as they are generated, and partial correctness is recorded. One cycle of Guided Bottom-Up Search ends when the cost limit is reached (calculated by 6 C where C the maximal production cost in the initial PCFG)[1].

Just-in-Time Learning

The defining feature of Probe is its **just-in-time learning** mechanism. After each budgeted search attempt, Probe updates the grammar probabilities based on the partial solutions discovered during that attempt.

Each partial solution is assigned a fitness score Fit corresponding to the fraction of examples it satisfies. For each grammar rule r_i, Probe computes the maximum fitness among all partial solutions that use that rule. The probability of rule r_i is then updated according to the following rule:

$$p(R) = \frac{p_u(R)^{(1-\mathrm{Fit})}}{Z} \tag{1}$$

where Z is the normalization factor and pu is the uniform distribution of the grammar[1]. This update mechanism biases the grammar toward rules that appear in higher-quality partial solutions, increasing the likelihood that future search attempts will combine these rules into a fully correct program.

Restart and Iteration

After updating the grammar, Probe restarts the search from scratch using the new probability distribution. This process is repeated for multiple budgeted attempts until either a correct program is found or the overall resource budget is exhausted.

By alternating between bounded search and targeted learning, Probe effectively balances exploration and exploitation, enabling it to solve synthesis problems that are infeasible for unguided enumeration.

3 Related Work

This work builds on two groups of related work.

Firstly, several recent studies by members of our research group explore budgeted search strategies for program synthesis within a shared experimental framework just like in this paper. However the key difference is the type of strategies we explore. They investigate techniques relating to subprograms[7][6] and structural sketches[5], but do not consider probabilistic grammar updates or grammar learning. In contrast, this work focuses specifically on Probe-style grammar adaptation and compares its effectiveness within the same framework to their approaches.

Secondly, the Probe algorithm [1] introduces guided bottom-up synthesis with just-in-time learning of grammar probabilities. While Probe demonstrates the effectiveness of grammar learning, it evaluates the approach largely as a fixed algorithm and does not systematically analyze the impact of individual design choices. Only doing individual tests for its selector functions, where we learn selecting only one promising program is very beneficial. This work reinterprets Probe as a set of modular design principles and evaluates their robustness when transferred to a different synthesis architecture. Thus bringing extra clarity to it.

4 Problem Description

In this section we formally define the underlying problem that is being tackled in this research paper: program synthesis from input-output examples given a limited budget.

Definition (Budgeted Input–Output Program Synthesis). Let $\mathcal{P}$ be a context-free grammar defining a space of candidate programs, and let $\mathcal{S} = \{(x_i, y_i)\}_{i=1}^{n}$ be a finite set of input–output examples specifying desired program behavior. Given a computational budget B, the goal is to synthesize a program $p \in \mathcal{P}$ such that $p(x_i) = y_i$ for all $(x_i, y_i) \in \mathcal{S}$ before the budget B is exhausted.

The budget B may be expressed in terms of wall-clock time, number of program enumerations, or an equivalent cost measure. If no program satisfying all examples is found within the budget, the synthesis procedure returns failure.

5 Methods

5.1 Overview and Motivation

The goal of this work is to investigate how closely a grammar-guided program synthesis algorithm must follow its original formulation in order to remain effective when embedded in a different synthesis framework. To this end, we implemented a Probe-inspired synthesis algorithm within the HERB.JL ecosystem, which we refer to as *Herb-Probe*. THe origins of that program lie in an already existing version in the algorithm, but adapted to fit our needs[4].

To support systematic experimentation and fair comparison, Herb-Probe was implemented using a shared *budgeted search controller*, developed jointly with other project members. Among other uses it helps us abstract Probe to it's modular design choices, which is how we structure this section.

5.2 Shared Budgeted Search Controller

At the core of our implementation lies a budgeted search controller, implemented in the `HerbSearch` module. This controller abstracts over synthesis algorithms themselves and enforces a common execution structure across all approaches in the project. We depict it in Algorithm 1.

The controller maintains the following components:

- a synthesis problem,
- a program iterator used for enumeration,
- a synthesis function that evaluates candidate programs,
- a stopping condition,

Algorithm 1 Budgeted Search Controller

Require: Problem P, iterator I, synthesis function SYNTH, selector SELECT, updater UPDATE, stop checker STOP, attempts K
Ensure: Results $\mathcal{R}$, times $\mathcal{T}$
1: $\mathcal{R}, \mathcal{T} \leftarrow [\,]$
2: **for** $k = 1$ to K **do**
3: $(r, t) \leftarrow \text{SYNTH}(P, I)$
4: append r to $\mathcal{R}$, t to $\mathcal{T}$
5: **if** $\text{STOP}(r)$ **then**
6: **break**
7: **end if**
8: $I \leftarrow \text{UPDATE}(\text{SELECT}(r), I)$
9: **end for**
10: **return** $(\mathcal{R}, \mathcal{T})$

- a selection function that chooses promising candidates,
- an updater function that adapts the search state between attempts.

Execution proceeds in a fixed number of *attempts*, where each attempt corresponds to one synthesis cycle under a bounded budget. During each attempt, the controller:

1. runs the synthesis function on the current iterator,
2. records timing and result information,
3. checks whether a stopping condition has been met,
4. updates the iterator using the selected promising programs.

This design serves two purposes. First, it enforces a uniform experimental protocol across synthesis strategies, ensuring comparability. Second, it isolates algorithm-specific logic—such as grammar updates and selection heuristics—from the control flow, facilitating experimentation with different design choices.

5.3 The Iterator

Unlike the original Probe algorithm, which builds on Guided Bottom-Up Search, Herb-Probe employs probabilistic top-down enumeration via the `Most Likely First Search (MLFS)Iterator`[4] provided by HERB.JL.

In HERB.JL, programs are represented as abstract syntax trees derived from a context-sensitive grammar. Enumeration with this algorithm proceeds by incrementally expanding partial trees until a complete program is obtained. The `MLFSIterator` prioritizes enumeration based on grammar rule probabilities, enumerating programs in decreasing order of estimated likelihood.

The priority assigned to a partial or complete program is defined as the negative of the maximum log-probability achievable by any derivation consistent with the current tree:

$$\text{priority}(p) = -\max \log P(p \mid G),$$

where G denotes the grammar. Lower priority values are explored earlier, ensuring that programs composed of high-probability rules are enumerated first.

This probabilistic ordering fulfills a role analogous to cost-guided enumeration in Probe, but differs significantly in implementation. In particular, no explicit cost layers are maintained; instead, the enumeration order emerges directly from the grammar probabilities.

We chose to differ this implementation since when looking with an overview this seems like a clear abstract design choice, and a good version of it was already implemented.

5.4 Synthesis and Fitness Evaluation

For each synthesis attempt, Herb-Probe enumerates candidate programs using the current iterator and evaluates them against the specification. Each candidate program is interpreted on all examples in the problem specification, unless one of the examples is not compatible, and a fitness score is computed.

The fitness of a program p is defined as:

$$\text{fitness}(p) = \frac{\text{number of correctly solved examples}}{\text{total number of examples}}.$$

Programs with fitness strictly greater than zero are retained as *promising programs*. If a program achieves fitness 1.0, indicating that it solves all examples, the synthesis attempt terminates early. It also ends early if no promising solution is found. This is the stop function we initialize, whereas Probe doesn't stop in that case.

This mechanism mirrors Probe's notion of partial solutions but differs in execution. Fitness is computed eagerly during enumeration, and promising programs are stored explicitly as frozen abstract syntax trees for subsequent analysis and grammar updates. Both are a result of the differing enumeration strategy.

5.5 Selection Strategies

After each synthesis attempt, a subset of promising programs is selected to guide grammar adaptation. We implemented multiple selection strategies to study their impact on learning dynamics:

- **All-promising selection**: all programs with fitness greater than zero are retained. (baseline)
- **Best-only selection**: only the program with the highest fitness is retained. (top performing selector of Probe)[1]
- **Non-trivial selection**: only programs whose fitness exceeds a fixed threshold of 0.2 are retained.

These strategies allow us to investigate whether grammar learning benefits more from aggregating data across many partial solutions or from focusing on a single high-quality candidate. Does this have an effect with different iterators?

5.6 Grammar Adaptation (Herb-Probe)

The central contribution of Herb-Probe lies in its grammar update mechanism. Grammar rules in HERB.JL are associated with log-probabilities, which are updated between synthesis attempts based on the selected promising programs.

For each grammar rule r, we compute the maximum fitness among all selected programs in which r appears, denoted $f_{\max}(r)$. The rule's log-probability is then updated according to:

$$\log P'(r) = (1 - f_{\max}(r)) \cdot \log P(r).$$

This update rule is directly inspired by Probes as shown earlier, but adapted to deal with reverse calculation using -logs. After updating all rules, the grammar is normalized to ensure that probabilities sum to one for each non terminal.

5.7 Grammar Update Variants

To study the sensitivity of Probe-style learning to design choices, we implemented three variants of the grammar update mechanism where we determine the value of P(r):

1. **Iterative update**: grammar probabilities are updated based on previous probabilities.
2. **Original-style update**: probabilities are reset to a uniform baseline scaled by the number of rules. (How it is in Probe)
3. **Hybrid update**: a normalized copy of the original grammar is updated and applied.

These variants differ in how much historical information they retain and how strongly they bias future enumeration, while sharing the same underlying update rule.

5.8 Grammar Normalization

After each grammar update, probabilities are normalized to form a valid probabilistic context-sensitive grammar. Normalization is performed per nonterminal, ensuring that the probabilities of all rules expanding the same nonterminal sum to one.

If a grammar is not yet probabilistic, normalization initializes a uniform distribution. This step is crucial for maintaining stable enumeration behavior and ensuring that probabilistic ordering remains meaningful throughout synthesis.

5.9 Summary

Herb-Probe integrates Probe-inspired grammar learning into a probabilistic top-down synthesis framework. While it preserves the central idea of learning from partial solutions under a fixed budget, it departs from the original Probe algorithm in its enumeration strategy, grammar representation, and update mechanics.

This implementation provides a flexible experimental platform for studying the interaction between grammar learning and probabilistic enumeration, and for evaluating which components of Probe are essential to its effectiveness.

6 Evaluation

In this section we evaluate Herb-Probe and the shared budgeted-search framework using a variety of experiments, which are additive. We design these so that they can be used to answer the following research questions:

- **RQ1:** *Does increasing the total budget of a Herb-Probe run without any cycles lead to an increasing number of solved benchmarks?*
- **RQ2:** *At a fixed total budget, do increased Herb-Probe cycles solve more benchmarks than a single search?*
- **RQ3:** *Does the specific selector policy of Herb-Probe notably affect success?*

- **RQ4:** *Does Probe's original grammar update rule remain competitive for Herb-Probe?*
- **RQ5:** *Can the shared budgeted search framework successfully compare multiple differing implementations?*

6.1 Experimental Setup

Implementation and reproducibility. Herb-Probe and the shared budgeted-search framework were both implemented in Julia[1]. All code was implemented in in Herb.jl within the `HerbSearch` codebase. The experiments use the shared `BudgetedSearchController` and plug in Herb-Probe components via `synth_fn`, a `selector`, an `updater`, and a `stop_checker`. You can also deliver the enumerations per cycle and number of attempts. The benchmark execution file produces one JSON record per problem. In the original code extra outputs were added to record more data, including per-attempt fitness traces, timing, and snapshots of the grammar log-probabilities after each cycle. All tests were run once.

Benchmarks. We evaluate on the SyGuS PBE-SLIA Track 2019 dataset shipped with `HerbBenchmarks`. It contains 205 problems. Each benchmark provides a problem specification (input–output examples) and a grammar. We treat a benchmark as *solved* iff a synthesized program satisfies all examples.

Compute environment. Large-scale runs were executed as Slurm job arrays on the DelftBlue[2] cluster, dividing the benchmark suite into chunks and writing per-chunk JSONL output for later aggregation.[3]

6.2 RQ1: Does increasing the total budget of a Herb-Probe run without any cycles lead to an increasing number of solved benchmarks?

Results and discussion. For this experiment we decided to run about 10 different single cycle runs with maximum enumerations set from 10^4 until 10^7. You can see the mapped results in Figure 1. In the graph we note the percentage of the full benchmarks examples worth of problems fitness and their respective fitness that was reached during the cycle. We see that as enumerations increase the number of unsolvable problems decreases, while the number of solved ones increases. Those with a fitness between 0 and 1 retain a consistent share of the percentage at around 20%.

From this grammar we learn that we need quite a high enumeration count if we want to solve all problems. This is because Herb-Probe, if 0 promising problems are found in the first cycle, ends the run. For deciding a benchmark though the results are good news. They tell us that no matter which number of total iterations we choose 20% of the problems should be solvable.

Answer (RQ1). Yes. The greater the number of total enumerations, the greater the total solved problems. Based on the consistent 20% of solvable problems, we select 10^6 enumerations as a practical total budget for subsequent experiments balancing coverage and runtime.

[1]https://github.com/cdmpreda

[2]https://doc.dhpc.tudelft.nl/delftblue/

[3]memory partition, 15GB memory, and module julia/1.11.6

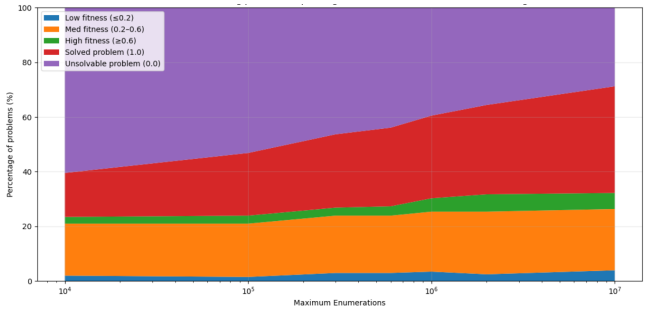

Figure 1: Shows an area plot comparing the total number of enumerations to the fitness achieved for its problems. All runs contain 1 cycle.

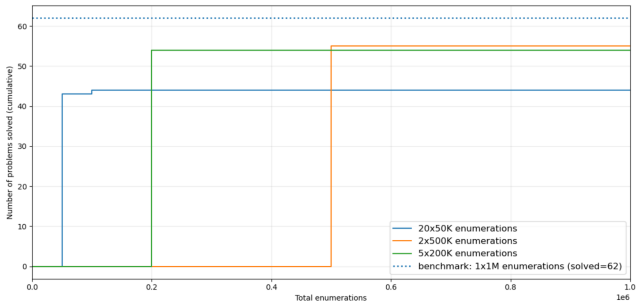

Figure 2: Solved benchmarks as a function of total enumerations. Also showing in which cycle which benchmark was solved. This figure pertains to cycle comparisons

6.3 RQ2: At a fixed total budget, do increased Herb-Probe cycles solve more benchmarks than a single search?

Results and discussion. Figure 2 compares Herb-Probe configurations that distribute a fixed total budget of 10^6 enumerations across multiple cycles. All multi-cycle configurations underperform the single-cycle baseline. Across all runs, almost all benchmarks are solved in the first cycle; only one benchmark is solved in a later cycle, and none achieve their best fitness beyond the second cycle. Analysis of per-benchmark logs shows that promising programs tend to repeat across cycles, while grammar entropy remains approximately constant, indicating little effective learning.

Among the multi-cycle variants, 2×500K performs best, which is largely explained by its longer first cycle. As shown in RQ1, larger uninterrupted searches reduce the probability that no promising programs are found at all. However, this advantage does not stem from improved cross-cycle learning: entropy measurements do not indicate stronger or more focused updates. When per-cycle budgets are too small, there is too little room for exploration; when they are large, most synthesis is already achieved in the first cycle.

Answer (RQ2). No. At a fixed total budget, increasing the number of Herb-Probe cycles does not improve synthesis performance over a single uninterrupted search. Despite 2×500K outperforming other multi-cycle configurations, we proceed with 5×200K to analyze the cycles effects.

6.4 RQ3: Does the specific selector policy of Herb-Probe notably affect success?

Results and discussion. We compare three selector policies: ALL (retain all partial solutions with $f > 0$), NON-TRIVIAL (retain solutions with $f > 0.2$), and BEST-ONLY (retain only the highest-fitness solution). Figure 4 shows that BEST-ONLY clearly outperforms the other selectors, solving seven additional benchmarks beyond the first cycle. All additional solves occur in the second cycle; no benchmark achieves its best fitness after cycle 2 under any selector.

Grammar entropy analysis provides insight into this behavior in Figure 3. When using ALL or NON-TRIVIAL, entropy stagnates or increases after the first update, indicating diffuse and noisy grammar updates. In contrast, BEST-ONLY produces a clear entropy decrease after the first cycle, suggesting a more focused learning signal. The subsequent rise in entropy after cycle 2 likely reflects limited diversity in the update signal, as only a single program contributes to learning.

Answer (RQ3). Yes. The selector policy significantly affects Herb-Probe's performance. Selecting only the highest-fitness partial solution yields the strongest learning signal and the best overall results, confirming this choice as a core design principle rather than an implementation artifact. We continue with BEST-ONLY.

6.5 RQ4: Does the updater rule matter, and does the original Probe update remain competitive?

Results and discussion. We compare three grammar update rules: ORIGINAL , ITERATIVE , and HYBRID.

As shown in Figure 5, the ORIGINAL update consistently achieves the highest number of solved benchmarks. The HYBRID and ITERATIVE variants perform similarly, with both solving fewer benchmarks than the original formulation. Across all variants, nearly all solved benchmarks are found in the first cycle, with only a small number appearing in the second.

While the ITERATIVE updater occasionally produces more promising partial solutions, this does not translate into additional solved benchmarks, suggesting that increased quantity of partial solutions alone is insufficient without strong probability reweighting. With one cycle we cannot make too many conclusions.

Answer (RQ4). Yes. The grammar update rule materially affects performance, and Probe's original update remains the most effective among those tested. Its reset-like behavior appears better suited to sparse or noisy learning signals than purely iterative alternatives.

6.6 RQ5: Can the shared budgeted-search framework successfully compare multiple differing implementations?

Results and discussion. To evaluate the flexibility of the shared budgeted-search framework, we compare Herb-Probe against two alternative budgeted synthesis approaches implemented by other project members: a structural sketch-based method and a promising subprogram-based method. All approaches are evaluated under identical conditions: 100 cycles with 10^4 enumerations per cycle (total budget 10^6), a selector threshold of $f > 0.2$, and the full set of 205 SyGuS PBE-SLIA benchmarks.

Approach	Solved Benchmarks
Herb-Probe (this work)	33 / 205
Structural Sketches	51 / 205
Promising Subprograms	31 / 205

Table 1: Comparison of different budgeted-search implementations under identical settings.

Table 1 summarizes the results. Despite substantial differences in internal representations and output formats, all implementations could be executed, logged, and evaluated using the same controller and benchmark pipeline. This enabled direct comparison without modifying the experimental infrastructure.

The absolute performance differences largely reflect configuration sensitivity rather than framework limitations. In particular, Herb-Probe's lower score is consistent with earlier findings (RQ1–RQ2): with only 10^4 enumerations per cycle, many benchmarks fail to produce any promising programs, limiting the effectiveness of grammar learning.

Answer (RQ5). Yes. The shared budgeted-search framework enables fair and reproducible comparison across heterogeneous budgeted synthesis strategies, fulfilling its goal as a unifying experimental platform.

6.7 Reflection Experiment: Matching a Stronger Baseline

Across RQ2–RQ4, we consistently observe that Herb-Probe rarely benefits from more than two cycles: almost all benchmarks achieve their best fitness in the first cycle, with a small number improving in the second and none thereafter. Grammar entropy stabilizes early, and the same promising programs are repeatedly selected in later cycles. This suggests that additional cycles contribute little new learning signal in the current setting.

To reflect on this observation, we evaluate whether Herb-Probe can still be competitive when restricted to the cycles that appear effective. We configure Herb-Probe using the best-performing choices identified earlier: two cycles, $2 \times 2 \cdot 10^5$ enumerations, the BEST-ONLY selector (RQ3), and the original Probe update rule (RQ4). We compare this against a single-search baseline with an equivalent total budget of $4 \cdot 10^5$ enumerations.

Under this comparison, Herb-Probe solves 58 benchmarks, while the baseline solves 55. Although the difference is small, Herb-Probe slightly outperforms the baseline despite operating with a lower per-cycle budget. Notably, according to RQ1, $2 \cdot 10^5$ enumerations fall in a regime with a higher proportion of unsolved benchmarks, making this result nontrivial.

Conclusion. While Herb-Probe does not exhibit sustained multi-cycle improvement, this experiment shows that, when restricted to its effective operating regime, Probe-style grammar learning can partially compensate for reduced enumeration budgets. This serves as a final validation of the im-

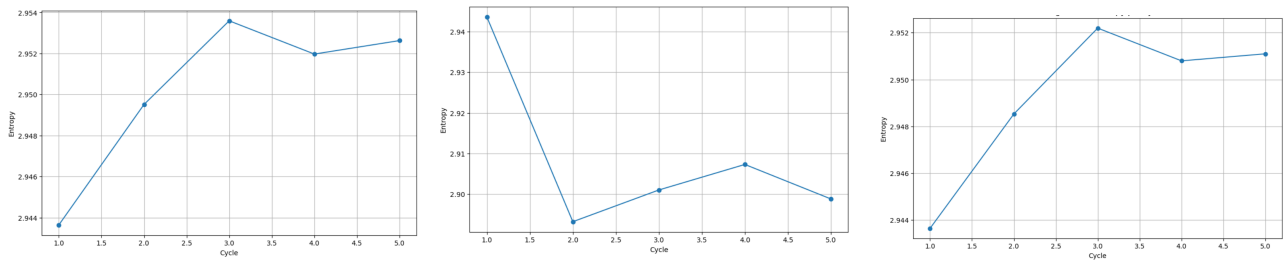

Figure 3: Shows the mean entropy evolution in the grammar cycles of the (left) select all, (middle) select best only, and (right) select not trivial schemes.

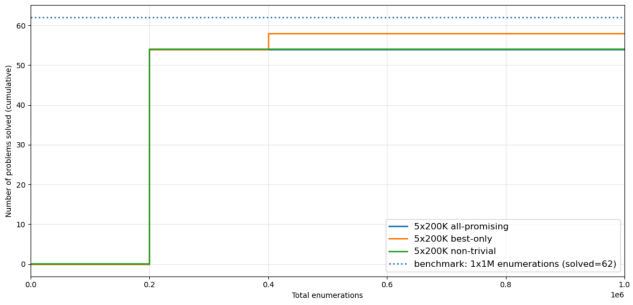

Figure 4: Solved benchmarks as a function of total enumerations. Also showing in which cycle which benchmark was solved. This figure pertains to selector scheme comparisons.

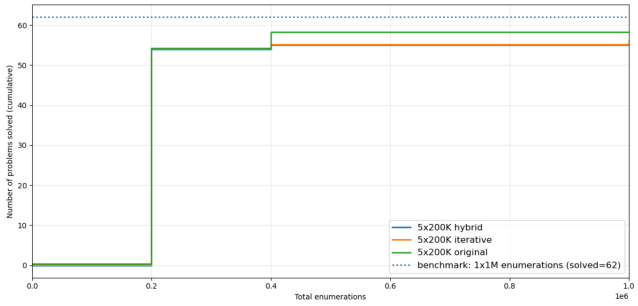

Figure 5: Solved benchmarks as a function of total enumerations. Also showing in which cycle which benchmark was solved. This figure pertains to updater comparisons

plementation and reinforces the conclusion that Probe's core design principles transfer to Herb-Probe, albeit with tightly bounded effectiveness.

7 Conclusions and Future Work

This work studied the robustness of Probe-style design principles when transferred to a different synthesis framework and search architecture. Rather than treating Probe as a fixed algorithm, we reinterpreted it as a set of modular choices and evaluated which of these remain effective under systematic variation.

We introduced *Herb-Probe*, a Probe-inspired synthesis approach implemented within a shared budgeted-search framework in Herb.jl. This framework enabled controlled experimentation and direct comparison across different budget allocations, selector policies, and grammar update rules.

Our results show that Herb-Probe aligns with several behaviors reported in the original Probe work. Selecting only the highest-fitness partial solution and using Probe's original update rule outperformed more permissive or iterative alternatives. At the same time, we find that the benefits of budgeted restarts are limited in this setting: most benchmarks are solved in the first cycle, with only modest gains in a second and no improvements thereafter. This could also be due to the harsh selector scheme. Naïvely splitting a fixed budget across many cycles often under performs a single uninterrupted search.

Despite these limitations, a final reflection experiment demonstrates that, when configured with the most effective design choices, Herb-Probe can match or slightly outperform a single-search baseline using the same number of enumerations. This confirms that Probe's core ideas remain practically useful, but that their impact is concentrated in early learning phases currently.

Beyond Herb-Probe itself, this work shows that the shared budgeted-search framework successfully supports fair and reproducible comparison across heterogeneous synthesis strategies. This contributes a reusable experimental infrastructure for future research on budgeted program synthesis.

Future Work. Several directions remain open. First, the enumeration strategy used in Herb-Probe differs from that of the original Probe implementation, making it difficult to isolate learning effects from search dynamics. Evaluating Probe-style updates with alternative enumerators may clarify this interaction. Second, all update rules studied here rely solely on fitness; incorporating decay, or unexplored program exploration may mitigate learning stagnation across cycles. A more permissive selection scheme would also be interesting to add. Finally, extending the shared framework with standardized outputs, richer diagnostics, and additional benchmarks would further enable large-scale comparative and hybrid synthesis studies.

8 Responsible Research

In running this research, due to its experimental and data driven nature, only a few ethical aspects were considered notable to mention. The most important are transparency of all ideas used in the paper that were not original or widespread, and giving them proper credit. In addition, being transparent on the use of Generative AI's to aid in this project and paper.

This paper and project makes use of artificial intelligence. Namely LLMs. These were used in all aspects of work: programming and writing. In the programming aspect LLMs were mostly used as a database for information, debugging errors, and occasionally a few lines of code. I think this is acceptable as it abstracts the problem for the researcher. It must be noted however that the LLMs input was always checked by me if I decided to include it and it was never used as the only source of information.

In the writing of the report LLMS were also used. Here it was mainly as a reviewer and generator of text. It must be noted that I would give the LLM the information I would want in a chapter, with a detailed list of my findings and ideas, followed by a structure it needed to follow. It is important to note only the organisation of information was left to the LLM. All ideas and information used were my own. This allowed me to keep more focus on the research and outsource difficult writing and structure which I struggle with. Note that LLMs were not used as a source of information or to generate their own findings for my requests. Also all outputs were reframed, and reworded by me and only what I wanted to say made it to the page.

Another ethical aspect that is absolutely vital is the reproducibility of this research; so others can confirm and elaborate on the research performed here. To that end, all the code and test data can be found on the my personal Github, which is mentioned in the experimental setup chapeter. Furthermore everything was programmed in Herb.jl[3], the framework that was used to run the experiments. The GitHub site provides extensive documentation on how to set it up and run it. In addition, extra care has been taken to ensure that the choices and details of the setup of the experiments are noted down for easy reproducibility.

A Example LLM Request for Writing

- I am now writing for X section. I want to put the following data and findings in this section.
- (List of data or detailed explanation of findings or argumentation I want)
- Please use this information in the shape of this style:
- (Author instruction details for structure)

References

[1] S. Barke, H. Peleg, and N. Polikarpova. Just-in-time learning for bottom-up enumerative synthesis. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–29, Nov. 2020.

[2] S. Gulwani, O. Polozov, and R. Singh. Program synthesis. *Foundations and Trends in Programming Languages*, 4(1–2):1–119, 2017.

[3] T. Hinnerichs, R. Gardos Reid, J. de Jong, B. Swinkels, P. Wochner, N. Filat, T. Magurescu, I. Hanou, and S. Dumancic. Herb.jl.

[4] T. Hinnerichs, R. G. Reid, J. de Jong, B. Swinkels, P. Wochner, N. Filat, T. Magurescu, I. Hanou, and S. Dumancic. Herb.jl: A unifying program synthesis library, 2025.

[5] U. Jaćimović. Identifying and leveraging structural sketches in budgeted program synthesis. Technical report, Delft University of Technology, 2026.

[6] A. Jelev. Exploring the effectiveness of using promising subprograms in program synthesis with budgeted search. Technical report, Delft University of Technology, 2026.

[7] J. Römer. Utilizing frequently occurring subprograms in program synthesis. Technical report, Delft University of Technology, 2026.