ECO SEARCH: A No-delay Best-First Search Algorithm for Program Synthesis

Théo Matricon[1], **Nathanaël Fijalkow**[1], **Guillaume Lagarde**[2]

[1] LaBRI, CNRS, France,
[2] LaBRI, Université de Bordeaux, France,

Abstract

Many approaches to program synthesis perform a combinatorial search within a large space of programs to find one that satisfies a given specification. To tame the search space blowup, previous works introduced probabilistic and neural approaches to guide this combinatorial search by inducing heuristic cost functions. Best-first search algorithms ensure to search in the exact order induced by the cost function, significantly reducing the portion of the program space to be explored. We present a new best-first search algorithm called ECO SEARCH, which is the first constant-delay algorithm for pre-generation cost function: the amount of compute required between outputting two programs is constant, and in particular does not increase over time. This key property yields important speedups: we observe that ECO SEARCH outperforms its predecessors on two classic domains.

Code — https://github.com/SynthesisLab/DeepSynth2/tree/eco_search_aaai

Extended version — https://doi.org/10.48550/arXiv.2412.17330

1 Introduction

Program synthesis is one of the oldest dream of Artificial Intelligence: it automates problem solving by generating a program meeting a given specification (Manna and Waldinger 1971; Gulwani, Polozov, and Singh 2017). A very classical scenario for user-based program synthesis, known as programming by example (PBE), uses input output examples as specification. For PBE, combinatorial search for program synthesis has been an especially popular technique (Alur, Radhakrishna, and Udupa 2017; Balog et al. 2017; Alur et al. 2018; Shi, Steinhardt, and Liang 2019; Barke, Peleg, and Polikarpova 2020; Zohar and Wolf 2018; Ellis et al. 2021; Odena et al. 2021; Fijalkow et al. 2022; Shi, Bieber, and Singh 2022; Shi et al. 2022; Ameen and Lelis 2023).

To scale combinatorial search for program synthesis, many approaches rely on defining a heuristic cost function assigning to every program a numerical value, such that the programs with least scores are the most likely to satisfy the specification. For example, DeepCoder (Balog et al. 2017) and TF-Coder (Shi, Bieber, and Singh 2022) use neural models, while BUSTLE (Odena et al. 2021) leverages probabilistic methods for defining a heuristic cost function. Very recently, LLMs have been used for guiding combinatorial search (Li, Parsert, and Polgreen 2024; Li and Ellis 2024).

Best-first search algorithms explore the space in the exact order induced by the cost function: this significantly reduces the portion of the program space to be explored. Since EUPHONY (Alur, Radhakrishna, and Udupa 2017)'s use of A^* algorithm, several best-first search algorithms have been constructed (Shi, Bieber, and Singh 2022; Ellis et al. 2021; Fijalkow et al. 2022; Ameen and Lelis 2023).

The major issue of best-first search algorithms is that they *slow down over time*. This is because in order to ensure optimality they need to consider a growing frontier of potentially next-to-be-generated programs in their data structures, which quickly become enormous. The notion of *delay* captures this behaviour: it quantifies the amount of compute required between outputting two programs. The first best-first search algorithm had linear delay (Alur, Radhakrishna, and Udupa 2017), and the state of the art algorithms achieve logarithmic delay (Fijalkow et al. 2022; Ameen and Lelis 2023): the compute required between outputting the t^{th} and the $(t+1)^{\text{th}}$ program is bounded by $O(\log(t))$.

The fundamental question explored in this paper is whether **there exist best-first search algorithms with constant delay**, also called "no-delay". We answer this question positively by constructing the first constant-delay best-first search algorithm called ECO SEARCH for pre-generation cost function. Importantly, ECO SEARCH performs a *bottom-up search*, which implies that it can take advantage of classical observational equivalence techniques. Technically, ECO SEARCH relies on the "cost tuple representation" introduced in (Ameen and Lelis 2023). A key novelty of ECO SEARCH is a new frugal expansion built on top of the one introduced in that paper, which ensures that it only considers programs when they need to be evaluated. Combined with novel data structures it enables ECO SEARCH to achieve constant delay.

We demonstrate the effectiveness of ECO SEARCH in two classic domains: the DeepCoder (Balog et al. 2017) domain of integer list manipulations and in the FlashFill (Gulwani 2011) domain of string manipulations. In our experiments, ECO SEARCH solves twice as many tasks in the same amount of time than previous methods. To summarize, our

contributions are the following:

- We introduce ECO SEARCH, a new best-first bottom-up search algorithm;
- Through a theoretical analysis we show that ECO SEARCH has constant delay;
- Experimentally, we observe that ECO SEARCH provides significant improvements over existing algorithms.

2 Background

Cost-guided Combinatorial Search

We consider a set P of elements, which for our applications in program synthesis is the class of all programs. Given a specification φ we write $p \models \varphi$ when the program p satisfies the specification φ: we say that p is a solution program. Note that this definition is independent of the type of specification: a logical formula, a set of input output examples, or any other type of specifications discriminating between solutions and not solutions.

The goal of *combinatorial search* for program synthesis is given a specification φ to find a solution program. Sometimes it is useful to find more than one solution program, or even all of them; in this paper we focus on finding a single one, but the algorithms naturally extend to finding a finite number of solution programs.

In *cost-guided combinatorial search*, we further assume a cost function $w : P \to \mathbb{R}_{>0}$, mapping each program to a (positive) cost. The cost function w is used as a heuristic: the smaller the cost of a program, the more likely it is to be a solution program. In this context, *best-first search algorithms* enumerate programs by increasing costs.

Domain-specific Languages and Context-free Grammars

Let us now be more specific about how programs are represented. A domain-specific language (DSL) is a programming language designed to solve a specific set of tasks. Classically, we represent DSLs using context-free grammars (CFGs), and more precisely deterministic tree grammars. We let Σ denote the set of primitive symbols, which include variables. Each symbol has a fixed arity (variables and constants have arity 0). The set of non-terminal symbols is Γ, and $S \in \Gamma$ is the initial non-terminal. Derivation rules have the following syntax:

$$X \to f(X_1, \ldots, X_k),$$

where $f \in \Sigma$ has arity k and $X, X_1, \ldots, X_k \in \Gamma$. The CFG is deterministic if given X and f, there is a unique derivation rule from X with f. A grammar acts as a generator: it generates trees, which we call programs.

To make things concrete, let us consider a small example. Our DSL manipulates strings and integers, hence it uses two types: `string` and `int`. It has three primitives:

```
cast: int -> string
concat: string -> string -> string
add: int -> int -> int
```

Let us add constants `"Hello", "World": string` and `1: int`. We also add a variable `var: int`. The

r_1 :	`str`	$\to$	"Hello"	cost: 1.1
r_2 :	`str`	$\to$	"World"	cost: 2.0
r_3 :	`str`	$\to$	`cast(int)`	cost: 4.4
r_4 :	`str`	$\to$	`concat(str, str)`	cost: 5.3
r_5 :	`int`	$\to$	`var`	cost: 1.8
r_6 :	`int`	$\to$	`1`	cost: 3.3
r_7 :	`int`	$\to$	`add(int, int)`	cost: 5.3

Figure 1: A simple DSL.

class of programs of type `int -> string` is generated by the CFG given in Figure 1, which uses two non-terminals, `string` and `int`, with the former being initial. An example program generated by this grammar is `concat("Hello", cast(add(var,1)))`. Using the natural semantics for `concat`, `cast`, and `1`, this program concatenates `"Hello"` to the result of adding 1 to the input variable and casting it as a string.

Pre-generation Cost Functions

In most cases cost functions are of a special nature: they are computed recursively alongside the grammar and induced by defining $\text{COST}(r)$ for each derivation rule r (see Figure 1 for an example). Note that $\text{COST}(r)$ can be any positive real number. Consider a program $\text{P} = f(\text{P}_1, \ldots, \text{P}_k)$ generated by the derivation rule $r : X \to f(X_1, \ldots, X_k)$, meaning that P_i is generated by X_i, then

$$\text{COST}(\text{P}) = \text{COST}(r) + \sum_{i=1}^{k} \text{COST}(\text{P}_i).$$

What makes pre-generation cost functions special is that they do not depend on executions of the programs, in fact they do not even require holding the whole program in memory since they are naturally computed recursively. Pre-generation cost functions is a common assumption (Balog et al. 2017; Ellis et al. 2021; Fijalkow et al. 2022).

3 ECO SEARCH

We present the four key ideas behind ECO SEARCH: cost tuple representation, per non-terminal data structure, frugal expansion, and buckets. The full description and pseudocode is given in the extended version which also includes a complete description and pseudocode with worked out examples of the two predecessors HEAP SEARCH and BEE SEARCH.

To make our pseudocode as readable as possible we use the generator syntax of Python. In particular, the **yield** statement is used to return an element (a program in our case) and continue the execution of the code. The main function is called OUTPUT, its goal is to output one or more programs. It is informally decomposed into a *generation* part, which is in charge of generating programs, and an *update* part, which updates the data structures.

Cost Tuple Representation

Let us take as starting point the BEE SEARCH (Ameen and Lelis 2023) algorithm, and its key idea: the *cost tuple representation*. BEE SEARCH algorithm maintains three objects:

Algorithm 1 The generation part of BEE SEARCH

```
1: function OUTPUT():
2:     (skip part of the code)
3:     t = (r : X → f(X_1, ..., X_k), n : (n_1, ..., n_k))
4:     for P_1, ..., P_k in ⨂_{i=1}^k GENERATED[INDEX2COST[n_i]] do
5:         if for all i ∈ {1, ..., k}, P_i is generated by X_i then
6:             P ← f(P_1, ..., P_k)
7:             add P to GENERATED[c]
8:             yield P
```

- GENERATED: stores the set of programs generated so far, organised by costs. Concretely, it is a mapping from costs to sets of programs: GENERATED$[c]$ is the set of generated programs of cost c.
- INDEX2COST: a list of the costs of the generated programs. Let us write INDEX2COST $= [c_1, \dots, c_\ell]$, then $c_1 < \cdots < c_\ell$ and GENERATED$[c_i]$ is defined.
- QUEUE: stores information about which programs to generate next. Concretely, it is a priority queue of *cost tuples* ordered by costs, that we define now.

Cost tuples are an efficient way of representing sets of programs. A *cost tuple* is a pair consisting of a derivation rule $r : X \to f(X_1, \dots, X_k)$ and a tuple $n = (n_1, \dots, n_k) \in \mathbb{N}^k$. For derivation rules $r : X \to a$, cost tuples are of the form $(r, \emptyset)$. A cost tuple represents a set of programs: (r, n) represents all programs generated by the rule r where the i^{th} argument is any program in GENERATED[INDEX2COST$[n_i]$]. The cost of a cost tuple $t = (r, n)$ is defined as

$$\text{COST}(t) = \text{COST}(r) + \sum_{i=1}^{k} \text{INDEX2COST}[n_i].$$

A single call to OUTPUT generates **all** programs represented by the cost tuple t found by popping QUEUE. Let us consider the case of a cost tuple $t = (r : X \to f(X_1, \dots, X_k), n : (n_1, \dots, n_k))$, the pseudocode addressing this case is given in Algorithm 1. The idea is to fetch all programs P_i in GENERATED[INDEX2COST$[n_i]$] and to form the programs $f(\text{P}_1, \dots, \text{P}_k)$. The issue is here that not all such programs are derived from the grammar: we additionally need to check whether each P_i was generated from X_i so we can apply the rule $r : X \to f(X_1, \dots, X_k)$. This means that many programs are discarded at this step, and it may even happen that no program is generated by a call to OUTPUT.

Taking a step back, the issue is that GENERATED contains all generated programs, losing track of which non-terminal were used to generate them.

Per Non-terminal Data Structure

Enters the HEAP SEARCH algorithm, which introduces the second key idea: *per non-terminal data structure*. Simply put, instead of a general data structure, HEAP SEARCH maintains independent objects for each non-terminal. Let us apply this philosophy and define the data structures for ECO SEARCH. Our algorithm maintains three objects for each non-terminal X:

Algorithm 2 The generation part in ECO SEARCH

```
1: function OUTPUT(X, ℓ):
2:     (skip part of the code)
3:     t = (r : X → f(X_1, ..., X_k), n : (n_1, ..., n_k))
4:     for P_1, ..., P_k in ⨂_{i=1}^k OUTPUT(X_i, n_i) do
5:         P ← f(P_1, ..., P_k)
6:         add P to GENERATED_X[c]
7:         yield P
```

- GENERATED$_X$: stores the set of programs generated from X so far, organised by costs. Concretely, it is a mapping from costs to sets of programs: GENERATED$_X[c]$ is the set of generated programs of cost c.
- INDEX2COST$_X$: a list of the costs of the generated programs from X. Let us write INDEX2COST$_X$ $= [c_1, \dots, c_\ell]$, then $c_1 < \cdots < c_\ell$ and GENERATED$_X[c_i]$ is defined.
- QUEUE$_X$: stores information about which programs to generate next. Concretely, it is a priority queue of *cost tuples* ordered by costs, that we define now.

We naturally adapt the definition as follows. A cost tuple represents a set of programs: (r, n) represents all programs generated by the rule r where the i^{th} argument is any program in GENERATED$_X$[INDEX2COST$_{X_i}[n_i]$].

This makes the generation part in ECO SEARCH very efficient, solving the limitation discussed above in BEE SEARCH. In Algorithm 2 we spell out part of the function OUTPUT, which takes as input a non-terminal X and a natural number ℓ (and becomes recursive). To formulate its specification let us write for a non-terminal X the set of costs $c_1 < c_2 < c_3 < \dots$ of all programs generated from X, we say that c_ℓ is the ℓ-smallest cost for X. The output of OUTPUT(X, ℓ) is the set of all programs generated from X with ℓ-smallest cost.

Frugal Expansion

We have presented the data structures of ECO SEARCH, the way it generates programs, and the specification of its main function OUTPUT. We now focus on the update part of OUTPUT. The third key idea is frugal expansion, which addresses the main issue with HEAP SEARCH: the number of recursive calls to OUTPUT. Indeed, to maintain the invariants on the data structures, we need to add cost tuples to the queue. As fleshed out in Algorithm 3, for a tuple $n : (n_1, \dots, n_k)$ we consider the k tuples obtained by adding 1 to each index $i \in [1, k]$: $m^i : (n_1, \dots, n_i + 1, \dots, n_k)$.

The issue is that this happens recursively as written in Algorithm 2, leading to many recursive calls. Two things can happen for a call to OUTPUT(X, ℓ):

- Either the result was already computed (if INDEX2COST$_X[\ell]$ is defined) and its answer is read off the data structure;
- Or it was not, and we perform some recursive calls as described in Algorithm 3.

Algorithm 3 Update part in ECO SEARCH

```
1: function OUTPUT(X, ℓ):
2:     (skip part of the code)
3:     t = (r : X → f(X_1, ..., X_k), n : (n_1, ..., n_k))
4:     (skip generation part of the code)
5:     for i from 1 to k do
6:         n' ← n
7:         n'_i ← n_i + 1
8:         if t' = (r, n') not in QUEUE_X then
9:             if n'_i not in INDEX2COST_{X_i} then
10:                c' ← COST(PEEK(QUEUE_{X_i}))
11:                INDEX2COST_{X_i}[n'_i] ← c'
12:            add t' to QUEUE_X with value COST(t')
```

The key property of frugal expansion is that when calling $\text{OUTPUT}(X, \ell)$, for each non-terminal Y, at most one recursive call $\text{OUTPUT}(Y, \_)$ falls in the second case. This analysis was already done in the arxiv version Section C.2 Lemma 2 of (Fijalkow et al. 2022), therefore we only give an overview.

At this point we have a simplified version of ECO SEARCH: as we will see in the experiments, it already outperforms HEAP SEARCH and BEE SEARCH, but it does not yet have constant delay. We will later refer to this algorithm as "ECO SEARCH without buckets".

Buckets

To introduce our main innovation, we need to state and prove some theoretical properties on the costs of programs induced by pre-generation cost functions. First some terminology: let us fix a non-terminal X, and P, P' two programs generated by X. We say that P' is a successor of P if $\text{COST}(\text{P}) < \text{COST}(\text{P}')$ and there does not exist P'' generated by X such that $\text{COST}(\text{P}) < \text{COST}(\text{P}'') < \text{COST}(\text{P}')$. In other words, P' has minimal cost among programs of higher cost than P generated by X. Note that a program may have many successors, but they all have the same costs. We write $\text{COST-SUCC}(\text{P})$ for the cost of any successor of P.

We first prove that successors in the cost tuple spaces are close in the cost space. Proofs of both lemmas below can be found in the extended version.

Lemma 1. *There exists a constant $M \geq 0$ such that for any program* P *we have* $\text{COST-SUCC}(\text{P}) - \text{COST}(\text{P}) \leq M$.

A consequence of Lemma 1 is a similar bound, this time applying to the queue in ECO SEARCH.

Lemma 2. *There exists a constant $M' \geq 0$ such that in* ECO SEARCH *at a any given time, for any non-terminal X, all programs* P *in the queue* QUEUE_X *satisfy:*

$$\text{COST}(\text{P}) - \min_{\text{P}' \in \text{QUEUE}_X} \text{COST}(\text{P}') \leq M'.$$

Let us make a simplifying assumption: the cost function takes integer values, meaning $w : P \to \mathbb{N}_{>0}$. Let us analyse the time complexity of $\text{OUTPUT}(X, \_)$. As discussed above frugal expansion implies that for each non-terminal Y, at most one call to $\text{OUTPUT}(Y, \_)$ yields to recursive calls. Hence the total number of recursive calls is bounded by the number of non-terminals, and we are left with analysing the time complexity of a single call. It is bounded by the time needed to pop and push a constant number (bounded by the maximum arity in the CFG) of cost tuples from a queue. If the queues are implemented as priority queues, the time complexity of these operations is $O(\log N)$, where N is the number of elements in the queue.

However, thanks to Lemma 2, there are at most M possible costs in the queue at any given time. Therefore, we can implement the queues as "bucket queues" (a classical data structure, see for instance (Thorup 2000)). Concretely, a bucket queue is an array of M lists, each containing cost tuples with the same cost. We keep track of the index j of the list that contains programs of minimal cost. To pop a cost tuple, we iterate over the j^{th} list. If the list at index j is empty, we increment $j \mod M$ until we find a non-empty list. To push an element that has a cost k plus from the current minimal cost, we simply add it to the list at index $(j + k) \mod M$. The time complexity of popping and pushing an element in this implementation is constant with lists implemented as single linked lists for example.

Theorem 1. *Assuming integer costs,* ECO SEARCH *has constant delay: the amount of compute between generating two programs is constant over time.*

4 Experiments

To investigate whether the theoretical properties of ECO SEARCH bear fruits we ask the following questions:

Q1: Does ECO SEARCH improve the performance of enumerative approaches on program synthesis tasks?

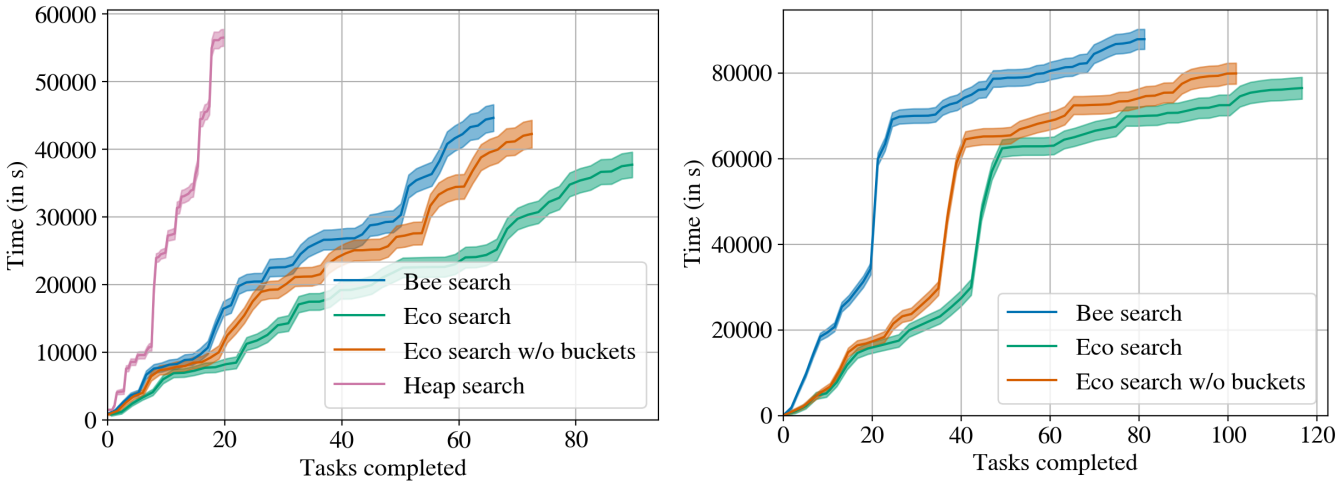
Q2: How does the performance of these algorithms scale with the complexity of the grammar?

Datasets. We consider two classic domains: string manipulations and integer list manipulations. For string manipulations we use the same setting as in BEE SEARCH (Ameen and Lelis 2023): FlashFill's 205 tasks from SyGuS. The DSL has 3 non-terminals, one per type. For integer list manipulation we use the DeepCoder (Balog et al. 2017) dataset comprised of 366 tasks. The DSL has 2 non-terminals, again one per type.

The cost functions used are the same for all algorithms, following (Fijalkow et al. 2022). Predictions are obtained with the help of a neural network outputting probability for each derivation rule. The neural networks are trained on the same synthetic dataset (one for each domain).

Implementation. All algorithms are re-implemented in Python. The code is made available as supplementary material, it contains the seeds used, the cost functions and all other additional minor experimental details. All experiments were run on a 16 GB RAM machine with an Intel Xeon(R) W-1270 CPU running at up to 3.40GHz, running Ubuntu Jellyfish (no GPUs were used). They were run on at least five different seeds and we report the mean performance along with the 95% confidence interval.

Algorithms. We compare ECO SEARCH against the two state of the art best-first search algorithms: HEAP SEARCH and BEE SEARCH. Since they are all bottom-up algorithms



(a) String manipulations from SyGuS using FlashFill's DSL

(b) Integer List Manipulation using DeepCoder's DSL

Figure 2: Main results: cumulative time for completing tasks

they all use observational equivalence (pruning programs with same outputs on all input examples). None of them have hyperparameters except for the rounding off procedure for costs. For BEE SEARCH we follow the original implementation and round off cost values to 10^{-2} in log space (since our cost function are probabilities). For ECO SEARCH we need to discretize costs, as follows. We discretize probabilities in log space up to 10^{-5} By default we use a bucket size of 20. We also experiment with other values, and for comparison, we also consider ECO SEARCH without buckets. When the constant M is less than 1000, we use M instead of the given bucket size. Those parameters were not tuned, we chose these constant as a naive trade-off.

Does ECO SEARCH improve the performance of enumerative approaches on program synthesis tasks?

We run all best-first search algorithms on our benchmarks. with a timeout of five minutes ($300s$) per task. We plot the mean cumulative time used and the 95% confidence interval with respect to the number of tasks solved on Figure 2a for string manipulations and Figure 2b for integer list manipulations.

First, for string manipulation, we observe that HEAP SEARCH is far outperformed by other algorithms with ECO SEARCH achieving the same score in 14% of the time. This is why we did not include HEAP SEARCH in integer list manipulation because it times out on most tasks.

Second, ECO SEARCH without buckets outperfoms BEE SEARCH. The increase in performance is small on string manipulation with a bit less than 10 more tasks solved but on integer list manipulation it solves more than 20 more tasks compared to BEE SEARCH. To explain why the gap in performance is different in the two domains, we will see in the next experiment that BEE SEARCH scales poorly with the number of non-terminals in the DSL, which is larger for string manipulation.

Finally, ECO SEARCH outperforms all other algorithms by a large margin, solving 13 more tasks on integer list manipulations and 20 more tasks than its variant without buckets. Comparing to BEE SEARCH, it reaches the same number of tasks solved in slightly more than half the time for string manipulation and 78% of the time for integer list manipulation, while solving at least 20 new tasks compared to BEE SEARCH on both datasets.

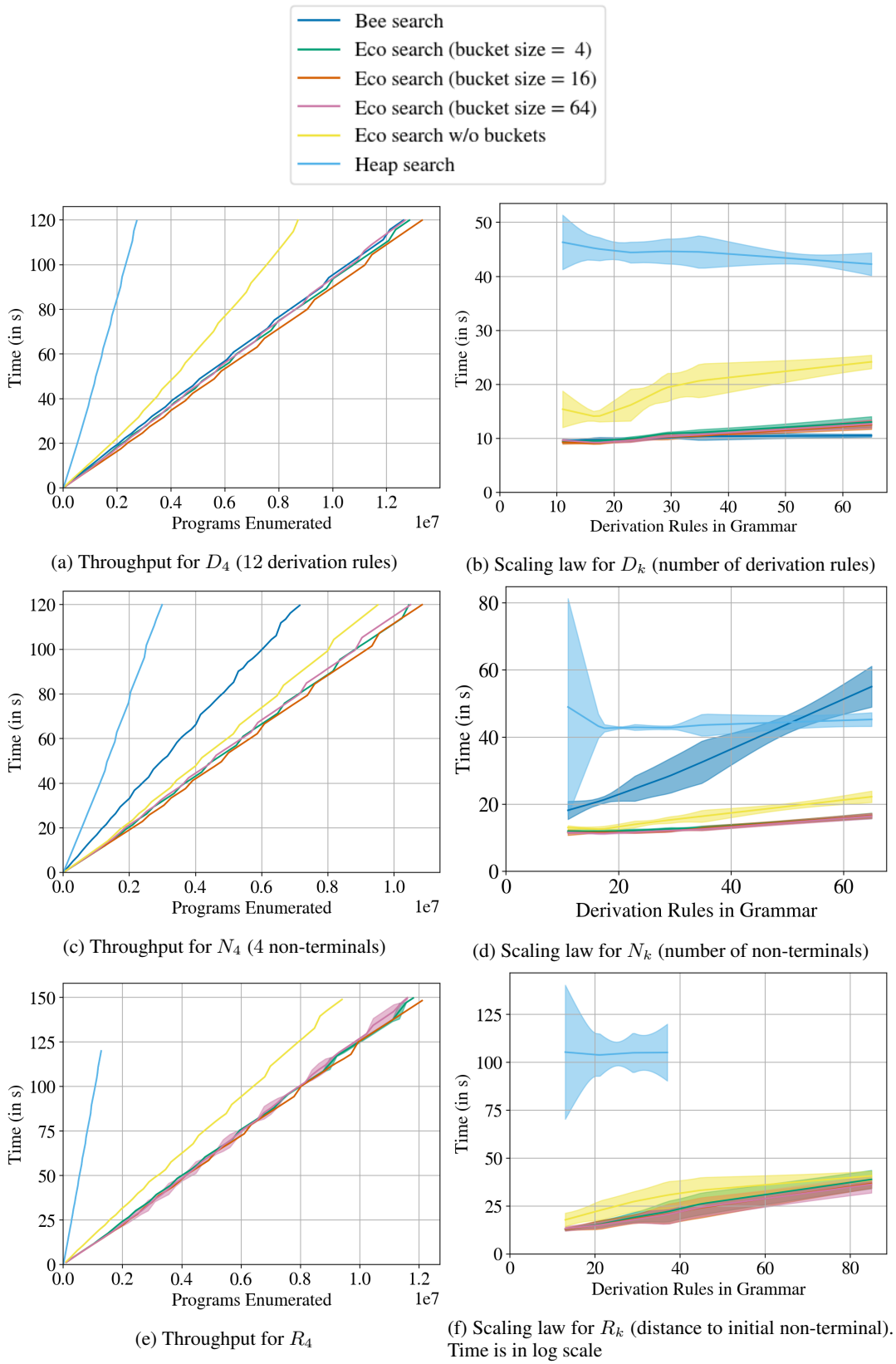
Summary

> ECO SEARCH outperfoms all other algorithms including its variant without buckets, solving as many tasks in 66% of the time and solving 30% more tasks in total.

How does the performances of these algorithms scale with the complexity of the grammar?

The goal of these experiments is to understand how well our algorithms perform on more complicated grammars. However there is no agreed upon definition of "grammar complexity" as different measures can be used. A bad proxy is the number of programs it generates: in most cases it is infinite, and grows extremely fast as a function of depth hence it cannot be accurately compared. We identify three parameters: the number of derivation rules, the number of non-terminals and he maximal distance from a non-terminal to the start non-terminal, meaning the number of derivation rules required to reach the non-terminal. In our experiments, we measure the performance of our algorithms for pure enumeration: the programs are not evaluated on input examples, enumeration continues for a fixed amount of time. For each parameter, we created parametric grammars:

- The grammar D_k has $3k$ derivation rules. It uses a single non-terminal S. The primitives are: k primitives f_i (arity 2), k primitives g_i (arity 1), and k constants h_i (arity 0).



(a) Throughput for D_4 (12 derivation rules)

(b) Scaling law for D_k (number of derivation rules)

(c) Throughput for N_4 (4 non-terminals)

(d) Scaling law for N_k (number of non-terminals)

(e) Throughput for R_4

(f) Scaling law for R_k (distance to initial non-terminal). Time is in log scale

Figure 3: Scaling against the three parameters: throughput and scaling laws.

The derivation rules are, for each $i \in [1, k]$:

$$S \to f_i(S, S) \quad ; \quad S \to g_i(S) \quad ; \quad S \to h_i$$

- The grammar N_k has k non-terminals, called $S_1, \ldots, S_k$, with S_1 initial. The primitives are: $2k$ primitives f_i, g_i (arity 1) and k constants h_i (arity 0). The derivation rules are, for each $i \in [1, k]$:

$$S_1 \to f_i(S_i) \quad ; \quad S_i \to g_i(S_1) \quad ; \quad S_i \to h_i$$

- The grammar R_k has k non-terminals, called $S_1, \ldots, S_k$, with S_1 initial. The primitives are: k primitives f_i (arity 3), k primitives g_i (arity 2), k primitives h_i (arity 1), and k constants k_i (arity 0). The derivation rules are, for each $i \in [1, k]$:

$$\begin{array}{ll} S_i \to f_i(S_{i-1}, S_i, S_{i+1}) & ; \quad S_i \to g_i(S_1, S_i) \\ S_1 \to h_i(S_i) & ; \quad S_i \to k_i \end{array}$$

For each of the three parameters, we consider two scenarios: *throughput*: for a fixed grammar, how many programs are enumerated as a function of time; *scaling law*: for a range of values of the parameter, how long does it take to enumerate one million programs. We plot the results for the three parameters and both scenarios on Figure 3.

First, we look at the evolution with the number of derivation rules. ECO SEARCH and BEE SEARCH perform equally well, irrespective of the bucket size. However, removing buckets makes ECO SEARCH much slower. When we look at the scaling law, the same result is observed, and there is little to no influence of the number of derivation rules.

Second, looking at the number of non-terminals, for the throughput scenario the results are the same as for the main experiments: HEAP SEARCH $<$ BEE SEARCH $<$ ECO SEARCH without buckets $<$ ECO SEARCH. On the scaling law, we observe that BEE SEARCH is outperformed by HEAP SEARCH for grammars with more than 15 non-terminals. The same growth is observed for all variants of ECO SEARCH albeit at a slower pace. This suggests that BEE SEARCH scales badly with the number of non-terminals: increasing the number of non-terminals 4x, BEE SEARCH takes 3x more time, while ECO SEARCH takes only 2x more.

Finally, looking at the distance to the starting non-terminal. BEE SEARCH is missing since we failed to enumerate $10K$ programs within the timeout, even for R_4. Similarly, HEAP SEARCH was not plotted for larger parameters because it failed to enumerate 1M programs. For the throughput, except for the disappearance of BEE SEARCH the results are as expected. For the scaling law, we observe that the distance has a significant impact: ECO SEARCH takes 6x more time for R_{22} compared to R_4.

Moreover, Figure 3 highlights the slowing down over time of the different algorithms. If we compare how the throughput evolves with the number of programs enumerated, then all algorithms but ECO SEARCH slow down faster due to their logarithmic delay. It is highlighted on Figure 3e, where HEAP SEARCH fails to generate 100.000 programs in the last 20 seconds of the experiment, and BEE SEARCH simply fails to do so in the first 20 seconds. The slope for ECO SEARCH without buckets clearly increase faster than for ECO SEARCH indicating a faster slow down.

Summary

ECO SEARCH scales better in terms of number of non-terminals and distance to starting non-terminal, and as well as BEE SEARCH for the number of derivation rules. ECO SEARCH slows down less than alternatives and is robust to the number of buckets.

5 Related Works

Combinatorial search for program synthesis has been an active area (Alur et al. 2018), and a powerful tool combined with neural approaches (Chaudhuri et al. 2021). In particular, cost-guided combinatorial search provides a natural way of combining statistical or neural predictions with search (Menon et al. 2013; Balog et al. 2017).

By exploring the space in the exact order induced by the cost function, best-first search algorithms form a natural family of algorithms. The first introduced was an A^* algorithm (Alur, Radhakrishna, and Udupa 2017). ECO SEARCH can be thought of as the unification of HEAP SEARCH (Fijalkow et al. 2022) and BEE SEARCH (Ameen and Lelis 2023), both best-first search bottom-up algorithms.

Best-first search algorithms were also developed for Inductive Logic Programming (Cropper and Dumancic 2020).

Importantly, ECO SEARCH follows the bottom-up paradigm, where larger programs are obtained by composing smaller ones (Udupa et al. 2013). Such algorithms have been successfully combined with machine learning approaches, for instance PC-Coder (Zohar and Wolf 2018), Probe (Barke, Peleg, and Polikarpova 2020), and DreamCoder (Ellis et al. 2021). In these works, machine learning improves combinatorial search while BUSTLE (Odena et al. 2021) and Execution-Guided Synthesis (Chen, Liu, and Song 2019) guide the search process with neural models. Alternatively, CROSSBEAM (Shi et al. 2022) and LambdaBeam (Shi et al. 2023) leverage Reinforcement Learning for this purpose. Interestingly, LambdaBeam solve tasks that LLMs cannot solve thanks to its ability to perform high-level reasoning and program composition. Together with recent approaches using LLMs for guiding search (Li, Parsert, and Polgreen 2024; Li and Ellis 2024), this motivates developing faster algorithms for cost-guided combinatorial search.

6 Conclusions

We introduced ECO SEARCH a new best-first bottom-up search algorithm, and proved it has constant-delay, meaning that the amount of compute required from outputting one program to the next is constant. On two classic domains it enables solving twice as many tasks in the same amount of time than previous methods.

Our experiments reveal an important research direction: search algorithms suffer drops in performance when the complexity of the grammar increases. Often the grammar remains small and this limitation is not drastic. However, recent applications of program synthesis use large grammars, for instance Hodel (2024) constructs a large DSL to solve the Abstraction Reasoning Corpus (Chollet 2019). We leave as an open question to construct best-first search algorithms that can operate at scale on such DSLs.

Acknowledgements

This work was partially supported by the SAIF project, funded by the "France 2030" government investment plan managed by the French National Research Agency, under the reference ANR-23-PEIA-0006.

References

Alur, R.; Radhakrishna, A.; and Udupa, A. 2017. Scaling enumerative program synthesis via divide and conquer. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 319–336. Springer.

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

Ameen, S.; and Lelis, H. L. 2023. Program Synthesis with Best-First Bottom-Up Search. *Journal of Artificial Intelligence Research*, 77.

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*.

Barke, S.; Peleg, H.; and Polikarpova, N. 2020. Just-in-Time Learning for Bottom-Up Enumerative Synthesis. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

Chaudhuri, S.; Ellis, K.; Polozov, O.; Singh, R.; Solar-Lezama, A.; and Yue, Y. 2021. Neurosymbolic Programming. *Foundations and Trends in Programming Languages*, 7(3): 158–243.

Chen, X.; Liu, C.; and Song, D. 2019. Execution-Guided Neural Program Synthesis. In *International Conference on Learning Representations (ICLR)*.

Chollet, F. 2019. On the Measure of Intelligence. *CoRR*, abs/1911.01547.

Cropper, A.; and Dumancic, S. 2020. Learning Large Logic Programs By Going Beyond Entailment. In *International Joint Conference on Artificial Intelligence, IJCAI*, 2073–2079. ijcai.org.

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*.

Fijalkow, N.; Lagarde, G.; Matricon, T.; Ellis, K.; Ohlmann, P.; and Potta, A. N. 2022. Scaling Neural Program Synthesis with Distribution-Based Search. *Proceedings of the AAAI Conference on Artificial Intelligence*, 36(6): 6623–6630.

Gulwani, S. 2011. Automating string processing in spreadsheets using input-output examples. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*.

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

Hodel, M. 2024. Addressing the Abstraction and Reasoning Corpus via Procedural Example Generation. arXiv:2404.07353.

Li, W.; and Ellis, K. 2024. Is Programming by Example solved by LLMs? In *Advances in Neural Information Processing Systems (NeurIPS)*.

Li, Y.; Parsert, J.; and Polgreen, E. 2024. Guiding Enumerative Program Synthesis with Large Language Models. In *International Conference on Computer Aided Verification, CAV*.

Manna, Z.; and Waldinger, R. J. 1971. Toward automatic program synthesis. *Communications of the ACM*, 14(3): 151–165.

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*.

Odena, A.; Shi, K.; Bieber, D.; Singh, R.; Sutton, C.; and Dai, H. 2021. BUSTLE: Bottom-Up Program Synthesis Through Learning-Guided Exploration. In *International Conference on Learning Representations (ICLR)*.

Shi, K.; Bieber, D.; and Singh, R. 2022. TF-Coder: Program synthesis for tensor manipulations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 44(2): 1–36.

Shi, K.; Dai, H.; Ellis, K.; and Sutton, C. 2022. CrossBeam: Learning to Search in Bottom-Up Program Synthesis. In *International Conference on Learning Representations (ICLR)*.

Shi, K.; Dai, H.; Li, W.; Ellis, K.; and Sutton, C. 2023. LambdaBeam: Neural Program Search with Higher-Order Functions and Lambdas. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Shi, K.; Steinhardt, J.; and Liang, P. 2019. FrAngel: Component-Based Synthesis with Control Structures. *Proceedings of the ACM on Programming Languages*, 3(POPL).

Thorup, M. 2000. On RAM Priority Queues. *SIAM Journal on Computing*, 30(1): 86–109.

Udupa, A.; Raghavan, A.; Deshmukh, J. V.; Mador-Haim, S.; Martin, M. M. K.; and Alur, R. 2013. TRANSIT: Specifying protocols with concolic snippets. In *Conference on Programming Language Design and Implementation (PLDI)*.

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