Journal of Artificial Intelligence Research 77 (2023) 1275-1310 Submitted 11/2022; published 08/2023

Program Synthesis with Best-First Bottom-Up Search

Saqib Ameen saqib.ameen@ualberta.ca
Levi H. S. Lelis levi.lelis@ualberta.ca
Department of Computing Science,
Alberta Machine Intelligence Institute (Amii),
University of Alberta, Canada

Abstract

Cost-guided bottom-up search (BUS) algorithms use a cost function to guide the search to solve program synthesis tasks. In this paper, we show that current state-of-the-art cost-guided BUS algorithms suffer from a common problem: they can lose useful information given by the model and fail to perform the search in a best-first order according to a cost function. We introduce a novel best-first bottom-up search algorithm, which we call Bee Search, that does not suffer information loss and is able to perform cost-guided bottom-up synthesis in a best-first manner. Importantly, Bee Search performs best-first search with respect to the *generation* of programs, i.e., it does not even create in memory programs that are more expensive than the solution program. It attains best-first ordering with respect to generation by performing a search in an abstract space of program costs. We also introduce a new cost function that better uses the information provided by an existing cost model. Empirical results on string manipulation and bit-vector tasks show that Bee Search can outperform existing cost-guided BUS approaches when employing more complex domain-specific languages (DSLs); Bee Search and previous approaches perform equally well with simpler DSLs. Furthermore, our new cost function with Bee Search outperforms previous cost functions on string manipulation tasks.

1. Introduction

Synthesizing computer programs that satisfy a specification is a long-standing problem in Computing Science (Waldinger & Lee, 1969; Manna & Waldinger, 1979; Fraňová, 1985; Deville & Lau, 1994; Colón, 2004; Solar-Lezama et al., 2006; Singh & Gulwani, 2012) that has received much attention from the Artificial Intelligence (Balog et al., 2016; Devlin et al., 2017a; Kalyan et al., 2018; Shin et al., 2019; Ellis et al., 2020) and the Programming Language communities (Albarghouthi et al., 2013; Udupa et al., 2013; Lee et al., 2018; Barke et al., 2020; Ji et al., 2020). In this paper, we consider programming-by-example problems in which a system receives a set of input-output examples and it attempts to synthesize a program that maps each input to the desired output.

One approach to solving program synthesis tasks is to search for a solution over the space of programs defined by a domain-specific language (DSL). The program space that DSLs induce can be very large, and a considerable amount of research has been devoted to developing more effective search algorithms to solve program synthesis tasks (Odena et al., 2021; Barke et al., 2020; Lee et al., 2018; Alur et al., 2017; Albarghouthi et al., 2013). Bottom-up search (BUS) is a successful search strategy that starts with the smallest possible DSL programs and iteratively generates larger programs by combining the smaller

©2023 The Authors. Published by AI Access Foundation under Creative Commons Attribution License CC BY 4.0.

ones generated by the algorithm (Albarghouthi et al., 2013; Udupa et al., 2013; Alur et al., 2013).

One of the key advantages of BUS over other search algorithms such as top-down search approaches (Lee et al., 2018; Alur et al., 2017) is that BUS generates complete programs during the search, which means that the programs can be executed and evaluated. The ability to execute programs allows one to discard observational equivalent programs (Albarghouthi et al., 2013); two programs are observational equivalent if they produce the same output value for a given set of input values. The detection of observational equivalent programs can substantially reduce the number of programs generated during the search.

However, due to the size of the search space, BUS is only able to find solutions to problems that can be solved with short programs. Cost-guided BUS algorithms are able to solve more problems than BUS because they use a cost function to guide the search toward more promising programs (Shi, Bieber, & Singh, 2020). A cost function receives a program and returns a cost value. Programs with low-cost values are deemed more promising than programs with high-cost values and are given preference to be used as subprograms of other programs, thus biasing the search. Several systems use cost-guided BUS algorithms: TF-Coder (Shi et al., 2020), Probe (Barke et al., 2020), Bustle (Odena et al., 2021), and Heap Search (Fijalkow et al., 2022).

In this paper, we show that, despite their superior performance to BUS, the guided search algorithms used in TF-Coder, Probe, and Bustle suffer from the same problem: they can lose some of the information given by the cost function because they round off the costs of the programs. As a result, they do not necessarily perform the search in best-first order with respect to the cost of the programs and might evaluate more expensive programs before evaluating cheaper ones. Heap Search is an existing cost-guided BUS algorithm that searches in best-first order with respect to a cost function. However, Heap Search searches in a best-first order with respect to the evaluation of the programs. This means that it can generate a very large number of programs that are more expensive than the solution program. Moreover, as we show in this paper, Heap Search sacrifices the detection of observational equivalent programs to attain its best-first ordering and it is only able to search in best-first order while using some of the cost functions from the literature.

In our work, we present a taxonomy for the cost functions used in previous work, where we divide them into two families of functions: pre-generation and post-generation. Pre-generation functions are those able to evaluate the cost of a program before the program is even created in memory. Post-generation functions require the program to be in memory so that the program can be executed as part of the computation of its cost.

In addition to our taxonomy, another contribution of this paper is a best-first bottom-up search algorithm we call Bee Search, which overcomes the weaknesses of previous cost-guided BUS algorithms. Bee Search performs search in a best-first ordering according to cost functions from both the pre-generation and post-generation families of functions. Moreover, Bee Search's best-first search is with respect to the generation of programs. That is, Bee Search does not even create in memory programs that are more expensive than the solution program. Note that other best-first algorithms such as A* (Hart et al., 1968) and Dijkstra's algorithm (Dijkstra, 1959) can generate states that are more costly than the goal state, which can hurt performance in domains with a large branching factor. Bee Search's generation-time best-first search is achieved by searching in an abstract cost-tuple

space. Each state in the cost-tuple space informs which programs should be generated next in search, such that the best-first ordering of programs is attained. Unlike Heap Search, Bee Search performs observational equivalence checks as regular BUS algorithms.

To highlight Bee Search's ability to use a wide range of cost functions, we introduce a novel cost function based on the neural network model used in the Bustle system. In contrast to Bustle's cost function, our cost function "relies" more on the prediction of the neural model and less so on the size of the evaluated programs.

We hypothesize that Bee Search is able to solve more problems than Probe and Bustle due to the information these algorithms lose during search. To evaluate our hypothesis, we compare the number of problems Bee Search solves while using the same cost functions Probe and Bustle used in a set of string manipulation tasks and in a set of bit-vector manipulation tasks. The results show that Bee Search is never worse than Probe and Bustle and it can solve more problems than them when searching in larger program spaces. We also evaluate Bee Search's generation-time best-first search by comparing it with Heap Search and with a search algorithm based on the best-first search algorithm used in Brute, an Inductive Logic Programming system (Cropper & Dumančic, 2020). Both Heap Search and Brute perform best-first search, but not with respect to the generation of programs, as Bee Search does. Bee Search outperforms both Heap Search and Brute by a large margin in all the settings evaluated. Finally, the results also show that Bee Search with our novel cost function outperforms all systems tested in the string manipulation domain.

This paper is organized as follows. We start by defining the program synthesis problem (Section 2), then we present existing uninformed and cost-guided bottom-up search algorithms for synthesis and discuss the limitations of a few contemporary cost-guided BUS algorithms (Sections 3 and 4). In Section 4.1, we discuss two cost functions and use them to describe the taxonomy of the cost functions used in the literature, then we present our bottom-up best-first search algorithm Bee Search (Section 5) and prove the guarantees it provides. In Section 6, we present empirical results, followed by related work (Section 7) and conclusions (Section 8). In Appendix A, we present the DSLs we used.

2. Problem Formulation

In program synthesis tasks, one is given a DSL in the form of a context-free grammar $\mathcal{G} = (V, \Sigma, R, I)$. Here, V, Σ, and R are sets of non-terminals, terminals, and relations defining the production rules of the grammar, respectively. I is $\mathcal{G}$'s initial symbol. Figure 1 shows a DSL with $V = \{I\}$, $\Sigma = \{$`concat`, `1`, `2`, $\cdots$, `1000`$\}$, R represents the production rules (e.g., $I \to$ `1`); we call non-terminal a production rule whose righthand side contains at least one non-terminal symbol and we call terminal a production rule whose righthand side does not contain a non-terminal symbol. The arity of a non-terminal rule is the number of non-terminal symbols on the rule's righthand side. For example, the arity of rule $I \to$ `concat`(I, I) is 2. The arity of terminal rules is 0. The programs $\mathcal{G}$ accepts determine the programs space. For example, $\mathcal{G}$ accepts `concat(concat(1, 2), 3)`: I is replaced with `concat`(I, I); then the leftmost I with `concat`(I, I) and the rightmost I with `3`, and so on. The DSL of this example treats all numbers as strings and `concat(concat(1, 2), 3)` returns `123`.

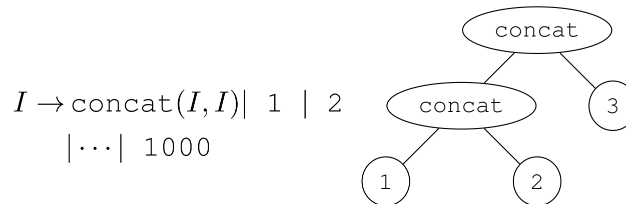

Figure 1: DSL and AST for concat(concat(1, 2), 3), which produces the output 123.

Search algorithms represent programs as abstract syntax trees (ASTs). Figure 1 shows the AST of the program concat(concat(1, 2), 3). Each node in the AST represents a production rule. Nodes representing a non-terminal rule have a number of children equal to the number of non-terminal symbols in the rule. For example, concat has two children because the rule $I \rightarrow \texttt{concat}(I, I)$ contains two symbols I. Nodes representing production rules of terminals are leaves in the AST. Note that each subtree in the AST represents a program. We call the subtrees rooted at a child of node p the subprograms of p. For example, concat(1, 2) and 3 are subprograms of the root node of the tree in Figure 1. We say that a program is generated in search when the program's AST is created and stored in memory. We say that a program is evaluated when it is executed.

In addition to a DSL, a program synthesis task is composed of a set of input values $\mathcal{I}$ and output values $\mathcal{O}$. The task is (i) to derive a program that $\mathcal{G}$ accepts and (ii) to correctly map each of the input values to its corresponding output value. For example, consider a DSL represented with a grammar $\mathcal{G}$ that is identical to the one shown in Figure 1 but augmented with the rules $I \rightarrow \texttt{in}_1|\texttt{in}_2$, where $\texttt{in}_1$ and $\texttt{in}_2$ are two input values. The program $\texttt{concat}(\texttt{in}_1, \texttt{in}_2)$ correctly produces the output value for the following problem: $\mathcal{I} = \{[1, 2], [10, 10]\}$ and $\mathcal{O} = \{[12], [1010]\}$.

3. Uninformed Bottom-Up Search (BUS)

BUS solves program synthesis tasks by enumerating all programs of size i before enumerating programs of size $i+1$, where size is the number of nodes in the program's AST. BUS starts by generating all programs defined by the terminal symbols of the DSL (size 1). Then, it uses the programs of size 1 to generate programs of size 2 through the production rules of the DSL; then it uses the programs of size 1 and 2 to generate programs of size 3, and so on. The search stops when it generates a program that maps the inputs to the outputs or it times out. Instead of size, BUS can also be height-based, where the height of the program's AST is considered. Since it has been shown that size-based BUS is more effective than height-based BUS in the string manipulation domain (Barke et al., 2020), which we consider in this paper, we only consider the size-based version in our work and call it BUS.

Example 1. *Consider an example where we need to synthesize a program that produces the output 100010001000 with the DSL shown in Figure 1 (the input set is empty). The solution to this problem is concat(concat(1000, 1000), 1000).* BUS *first generates and evaluates all programs of size 1: $\{1, 2, \cdots, 1000\}$. Since none of these programs correctly generates the desired output,* BUS *generates the set of programs of size 2, which is empty.*

Algorithm 1 Uninformed Bottom-Up Search (BUS)

Procedure: Uninformed-BUS($\mathcal{G}, (\mathcal{I}, \mathcal{O})$)
Require: $\mathcal{G} = (V, \Sigma, R, I)$, and a set of input-output examples $(\mathcal{I}, \mathcal{O})$.
Ensure: Solution program p or $\perp$

```
1: s ← 1
2: while not timeout do
3:    for p in Next-Program(G, B, s) do
4:       o ← Execute(p, I)
5:       if o equals O then
6:          return p
7:       if p is not equivalent to any program in B then
8:          B[s].add(p)
9:    s ← s + 1
10: return ⊥
```

Procedure: Next-Program($\mathcal{G}, B, s$)
Require: $\mathcal{G} = (V, \Sigma, R, I)$, programs bank B, and program size s.
Ensure: Program of size s

```
11: for r ∈ R do
12:    if arity(r) = 0 and size(r) = s then
13:       yield r
14:    else if arity > 0 and size(r) < s then
15:       for (p_1, ··· , p_k) in B × ··· × B do #operation over the values of the dictionary B
16:          if size(r(p_1, ··· , p_k)) = s and (p_1, ··· , p_k) is type-consistent with r then
17:             yield r(p_1, ··· , p_k)
```

Next, BUS *generates all programs of size 3:* {`concat(1, 1)`, $\cdots$, `concat(1000, 1000)`}*. This process stops when the solution is generated while* BUS *produces programs of size 5.*

The pseudocode for the uninformed bottom-up search is given in Algorithm 1. It receives a grammar $\mathcal{G} = (V, \Sigma, R, I)$, and set of input-output examples $(\mathcal{I}, \mathcal{O})$ and returns a program p that is able to map the inputs to the outputs. A failure $\perp$ is returned if no solution program is found. BUS starts by initializing the size $s = 1$ (line 1), then it enters the main loop, where, in each iteration, it calls Next-Program procedure to generate programs of size s. The variable s is incremented by one for the next iteration (line 9).

Next-Program receives the grammar $\mathcal{G}$, bank of programs B, which are indexed by the AST size of the programs, and the size of the target program s. Next-Program generates programs of size s using the production rules of the grammar $r \in R$ (lines 11-19). Next-Program returns the production rule r if it is terminal (lines 12-13). Otherwise, if r's arity is greater than 0, it generates programs with production rule r by taking the Cartesian product of all programs in the bank of programs B such that the following constraints are satisfied: (i) size($r(p_1, \cdots, p_k) = s$, where function size($\cdot$) returns the number of nodes in the program's AST, k represents the arity of the production rule r, and (ii) $(p_1, \cdots, p_k)$ is type-consistent with rule r, i.e., the type of subprograms, $p_1, \cdots, p_k$ matches the type of the arguments required by r (e.g., `concat` can only take arguments of the string type).

	$\mathbb{P}_r$	Cost
$I \rightarrow$ `concat`(I, I)	0.00005	14.28771
| `1` | `2` |$\cdots$| `999`	0.00099984	9.966013
| `1000`	0.00110895	9.816589

Figure 2: A probabilistic context-free grammar (PCFG) for string manipulation task with probability of each rule ($\mathbb{P}_r$) and its cost which is negative log of the probability (-log($\mathbb{P}_r$)).

Once a program p is yielded to the main loop, Uninformed-BUS executes p (line 4) and, if the output o of p matches the desired output $\mathcal{O}$, Uninformed-BUS returns the program p as a solution to the problem. Otherwise, it checks for observational equivalence, i.e., whether the search has previously seen a program with the same output set o. If not, the search adds the program p to the bank of programs B, indexed by the program size s. The search continues while it has not timed out and a solution program is not found.

4. Guided Bottom-Up Search

TF-Coder (Shi et al., 2020), Probe (Barke et al., 2020), Heap Search (Fijalkow et al., 2022), and Bustle (Odena et al., 2021) use a cost function w to guide the bottom-up search. The function w these systems employ favors programs that are "more likely" to lead to a solution. For example, in the problem described above, a cost function could favor programs that produce outputs with digits `1` and `0` as they appear in the desired output. In this section, we explain existing cost functions and then explain Probe, Bustle, Heap Search, and Brute, which are used as baselines in our experiments. Since TF-Coder's cost function requires a manually crafted set of weights for each operation of the language, we did not consider it in our experiments.

4.1 Cost Functions

We divide the cost functions from the literature into two types: *pre-generation* and *post-generation*. Pre-generation cost functions define the cost of a program p based on the production rule used to generate p and on the subprograms of p. For example, considering the DSL in Figure 1, a pre-generation function would determine the cost of the program `concat(1, 2)` as a function of the cost of the production rule $I \rightarrow$ `concat`(I, I) and of the subprograms `1` and `2`. The cost functions used in TF-Coder, Probe, and Heap Search are pre-generation functions where the cost of a program p is given by the sum of the cost of the production rule used to generate p and the cost of p's subprograms. We call these functions pre-generation because one can compute the cost of a program before generating the program. Post-generation functions determine the cost of a program p while using information that requires the execution of p. The cost function used in Bustle is post-generation because it uses the output of p to compute its cost. We call these functions post-generation because the AST of the program must be in memory to compute its cost.

4.1.1 Probe Cost Function (w_{Probe})

Probe uses a pre-generation cost function (w_{Probe}) based on a probabilistic context-free grammar (PCFG). The PCFG assigns a value to each production rule r denoting the probability that r is part of a solution. Consider the PCFG shown in Figure 2, Probe transforms the probability of a rule r, denoted by $\mathbb{P}_r$, into cost by taking $-\log_2(\mathbb{P}_r)$. The cost of each rule is shown in the column "Cost". The cost of a program $p = r(p_1, \cdots, p_k)$, denoted by $w(p)$, is given by the sum of the costs of its subprograms and the rule r used to derive it:

$$w(p) = w(r) + \sum_{i=1}^{k} w(p_i) \tag{1}$$

Example 2. *Consider program $p =$ `concat(1, 2)`. The cost of the program is given as $w(p) = 14.28771 + 9.966013 + 9.966013 = 34.219736$ because the cost of `concat`, `1`, and `2` is* -log(0.00005) = 14.28771*,* -log(0.00099) = 9.96601*, and* -log(0.001108) = 9.81658 *respectively. Similarly, the cost of $p =$ `concat(1000, 1000)` is $w(p) = 14.28771 + 9.816589 + 9.816589 = 33.920888$. Furthermore,* Probe *rounds off the cost of the programs to the nearest integer. For example, the cost of $p =$ `concat(1, 2)` would be* 34 *as given by w_{Probe}.*

The cost function w_{Probe} rounds off the cost value to the nearest integer because Probe enumerates the programs in increasing order of integer w values: first it enumerates all the programs of cost 1, then the ones with integer w-values of 2 and so on, until a solution is found. Probe learns the PCFG while searching. It runs the search until a budget Lim is exhausted and uses the partial solutions encountered in this search to train the PCFG. A partial solution is a program p, which maps at least one input from the input set $\mathcal{I}$ to its corresponding output in the set $\mathcal{O}$. The budget Lim is defined as a constant d, which is defined manually, multiplied by the highest cost l of a production rule in the current PCFG.

$$\text{Lim} = l \times d, \text{ where } l = \max_{r \in R}(w(r)).$$

After training the PCFG with partial solutions, the search is restarted with the updated PCFG. The parameter d allows one to define how often the system trains the PCFG model and restarts the search; Barke et al. used $d = 6$. If the search cannot find a solution after restarting and no partial solution is found, then the budget is increased to $\text{Lim}_i = \text{Lim}_{i-1} + l \times d$, where Lim_{i-1} is the budget of the previous iteration.

Probe's PCFG starts with a uniform probability distribution to all production rules, and it updates the probability distribution with partial solutions (programs) as follows. Probe selects a subset of partial solutions from all the partial solutions encountered in the current iteration that satisfy the following: (i) it is the first cheapest program according to the current cost model, (ii) satisfies a unique subset of input-output examples, and (iii) was not encountered in previous iterations. Then it updates the probability of all production rules $r \in R$, $\mathbb{P}(r)$ with

$$\mathbb{P}(r) = \frac{\mathbb{P}_u(r)^{1-\text{Fit}}}{Z} \text{ where } \text{Fit} = \max_{\{p \in PSol | r \in tr(p)\}} \frac{|o(p) \cap \mathcal{O}|}{|\mathcal{O}|}$$

Here, $\mathbb{P}_u(r)$ represents the probability of rules as given by the uniform distribution, Z represents the normalization factor, $PSol$ indicates a subset of partial solutions selected

using the aforementioned criteria, $tr(p)$ represents the trace of a program, which is the sequence of production rules used to derive the program p, $o(p)$ indicates the output of the partial program p, and Fit indicates the highest proportion of input-output examples solved by any partial solution $p \in PSol$ derived using production rule r. This way, Probe increases the probability of production rules r that solve the maximum number of input-output examples. Once Probe updates the PCFG, it stores $PSol$ in memory and maintains it across the restarts to ensure that partial solutions selected in previous iterations are not selected again to update the grammar. Therefore, PCFG is only updated when a set $PSol$ with novel partial solutions is found. Due to Probe's update rule, the probabilities are in the open interval $(0.0, 1.0)$. This way, the model is never "certain" that a symbol must or must not be used in the solution of a problem; no symbol in the language costs 0 and w is monotonically increasing.

4.1.2 Bustle Cost Function (w_{Bustle})

Bustle's cost function, w_{Bustle}, uses a neural network to compute the probability that a program is part of a solution. The network is a binary classification model that receives the input-output pairs $(\mathcal{I}, \mathcal{O})$ of the task and the output of a program p to each of the input values in $\mathcal{I}$ and returns the probability that p is a subprogram of a solution to the task.

Bustle's cost function is defined using two functions: w and w'. For program $p = r(p_1, \cdots, p_k)$ the function $w(p)$ is defined as

$$w(p) = 1 + \sum_{i=1}^{k} w'(p_i) \,.$$

Here, 1 is the cost of production rule $w(r)$ used to generate p (Bustle assumes all operators to cost 1), and $w'(p_i)$ is the cost of the subprogram p_i as given by the following equation.

$$w'(p) = w(p) + 5 - \delta(p) \quad \text{where } \delta(p) \in \{0, \cdots, 5\} \tag{2}$$

The value of $\delta(p)$ is an integer value that is based on the probability of p being a subprogram of a solution that the neural model returns. The integer value $\delta(p)$ returns is defined according to a binning scheme. Consider the values $\{0.0, 0.1, 0.2, 0.3, 0.4, 0.6, 1.0\}$; if the probability that the neural model returns is within the first two values, i.e., $[0.0, 0.1)$, then $\delta(p) = 0$, if it is within the second and third values, then $\delta(p) = 1$, and so on. The value of δ is used to penalize p by changing its cost according to the probability given by the neural network; lower probabilities will result in higher costs. For example, consider p with probability 0.05, then $\delta(p) = 0$, and $w'(p) = w(p) + 5$. This will delay the use of the subprogram p to generate further programs. Note that similarly to Probe, Bustle uses a discretization scheme to ensure that the cost values w and w' are integers. w_{Bustle} also increases monotonically.

Example 3. *Consider the generation of* `concat(concat(1, 3), 2)`. Bustle *computes its* w*-value as* $1 + w'$`(concat(1, 3))`$+ w'$`(2)`*, i.e., the cost of* 1 *for the operator* `concat` *plus the* w'*-costs of its subprograms* `concat(1, 3)` *and* `2`*. Once the program* `concat(concat(1, 3), 2)` *is generated, a neural network is used to calculate its* w'*-value, which is used when it appears as a subprogram in another program.*

```
io_pairs = [("hello world", "hello"),
            ("FOO BAR", "foo"),
            ("switch", "switch")]
ps = [lambda inp, out: out.lower() in inp.lower(),
      lambda inp, out: out in inp,
      lambda inp, out: len(inp) < len(out)]
```

Figure 3: A set of input output pairs (io_pairs) and property signature list (ps). The first property returns True for all pairs; the second property returns True for the first and third pairs and False for the second pair; and the third property returns False for all pairs.

Property Signatures The binary classification model that defines w_{Bustle} receives as input the set of input-output pairs, which could be of varied length. Instead of training a recurrent model to handle inputs of varied size, w_{Bustle} uses property signatures (Odena & Sutton, 2020) to define the input to a simpler fully connected feed-forward neural model. A property is defined as a function f that takes as input the input-output pair of a program p and returns a Boolean: $f(i, o) \rightarrow \{0, 1\}$. A property is used to define some aspect of p. For example, given an input-output pair (`hello world`, `hello`), a property function $f(i, o)$ that checks whether o is in i returns True to the input-output pair. Similarly, when a list of input-output pairs of a program p is evaluated with a list of k properties, we get a feature vector of length k, where each entry indicates the result of a property for all input-output pairs. Each entry of this vector has a value in $\{-1, 0, 1\}$, where the values of -1 and 1 indicate that the property returned either False or True, respectively, to all input-output pairs; the value of 0 indicates that the property returned True to some pairs and False to others.

Example 4. *Consider the set of input-output pairs and three properties, written in Python, shown in Figure 3. The first lambda function returns True and the third False to all pairs. The second property returns True to the first and third pairs and False to the second. Thus, this set of input-output pairs have the property signature vector* $[1, 0, -1]$.

The model w_{Bustle} receives an input of size fixed by a number of properties. That way, the number of input-output pairs can vary, but the input size remains the same.

4.1.3 Cost Functions for Bee Search

In this section, we show how we adapt w_{Probe} and w_{Bustle} to Bee Search. We also introduce a novel cost function to be used with Bee Search, which is based on w_{Bustle}.

The difference between Probe's w and Bee Search's version of it is that in the latter the costs are not rounded off. We denote both versions of the cost function by w_{Probe}. If used in the context of Bee Search, then we refer to the function that does not truncate the values; we refer to the original function of Probe in all other contexts.

Since Bee Search handles real-valued cost functions, we adapt the w function of Bustle by interpolating the values of δ from 0 to 5 according to Bustle's binning scheme. We interpolate $x = \{0.00, 0.15, 0.25, 0.35, 0.50, 1.00\}$ and $y = \{0, 1, 2, 3, 4, 5\}$ by pairing the values x and y: $(0.00, 0), (0.15, 1), (0.25, 2), (0.35, 3), (0.50, 4), (1.00, 5)$ and using a cubic spline to obtain an interpolant. Then, given a probability value returned by the model, the inter-

polant returns the δ-value used to obtain w', as shown in Equation 2. In the context of Bee Search, we use w_{Bustle} to refer to the interpolated version of the original w_{Bustle}.

We also introduce a novel cost function based on w_{Bustle} defined as follows.

$$w_{\text{U}}(p) = 1 + \sum_{i=1}^{k} w'_{\text{U}}(p_i) \tag{3}$$

where,

$$w'_{\text{U}}(p_i) = w_{\text{U}}(p_i) - \log_2 \mathbb{P}(p_i) \tag{4}$$

$\mathbb{P}(p_i)$ is the probability that p_i is part of a solution according to Bustle's neural network. This function computes the cost w of a program as the sum of the costs w' of its subprograms added to 1, the cost of a production rule; the cost $w_{\text{U}}(p)$ for all terminal symbols p is 1. The cost $w'_{\text{U}}(p_i)$ is given by $w_{\text{U}}(p_i)$ added to the negative of the log of the probability that p_i is part of a solution. Bustle's cost function limits how much the neural model can change the cost of a program by mapping the probabilities to a value between 0 and 5. Our cost function leaves the influence of the neural model unbounded by using $-\log_2 \mathbb{P}(p)$. The subscript U in w_{U} stands for "unbounded".

We call w_{Bustle} and w_{U} *penalizing functions* because the post-generation w'-value is not smaller than the w-value. We call w_{Probe}, w_{Bustle}, and w_{U} *additive* cost functions because the w-value of a program p is computed by adding the cost of the subprograms of p. The adapted versions of w_{Probe} and w_{Bustle}, and w_{U} are monotonically increasing cost functions.

In the next section, we describe a generic cost-guided bottom-up search algorithm that can be used to instantiate Probe and Bustle by changing the cost function the search uses. Heap Search and Brute are described in Sections 4.3 and 4.4, respectively.

4.2 Generic Cost-Guided Bottom-Up Search

In cost-guided bottom-up search, programs are enumerated in the order of increasing cost c. The cost is assigned by a cost function w, such that programs that are more likely to lead to the solution have a lower cost. The search enumerates all programs of cost c before enumerating programs of cost $c + 1$. It begins by enumerating all programs of cost 1, then in the second iteration, it uses the production rules $r \in R$ to combine the programs with cost 1 and generate programs of cost 2, and so on. The search continues until the solution program p is found, or the search budget is exhausted and a failure $\perp$ is returned.

Example 5. *Consider the DSL shown in Figure 2 along with the cost of each production rule $r \in R$, where the goal is to synthesize the program `concat(concat(1000, 1000), 1000)`. Costs are assigned in a way that they bias the search toward the solution (the symbol `1000` is cheaper than `1, 2, ···, 999`). The cost of program $p = r(p_1, \cdots, p_k)$ given by rule r is equal to the sum of the costs of its subprograms $p_1, \cdots, p_k$ and the production rule r. Furthermore, the cost of each program is rounded to the nearest integer value; for instance, the cost* 9.816589 *for program `1000` is rounded to* 10. *Cost-guided* BUS *will start by enumerating all programs with cost* 10*: {`1, 2, ···, 1000`}, since it is the cheapest set of programs that can be generated. No program can be generated with costs in* $[1, 9]$*. Next, it generates programs with cost* 34*: {`concat(1, 1), concat(1, 2), ···, concat(1000, 1000)`}. For example, the cost of `concat(1000, 1000)` is calculated as follows. Since*

Algorithm 2 Generic Cost-Guided Bottom-Up Search

Procedure: Cost-Guided-BUS($\mathcal{G}, (\mathcal{I}, \mathcal{O}), w$)
Require: $\mathcal{G} = (V, \Sigma, R, I)$, set of input-output examples $(\mathcal{I}, \mathcal{O})$, and a cost function w.
Ensure: Solution program p or $\perp$

```
1: c ← 1
2: while not timeout do
3:    for p in Next-Program(G, B, c, w) do
4:       o ← Execute(p, I)
5:       if o equals O then
6:          return p
7:       if p is not equivalent to any program in B then
8:          if post-generation cost function then
9:             c ← w'(p)
10:         B[c].add(p)
11:   c ← c + 1
12: return ⊥
```

Procedure: Next-Program($\mathcal{G}, B, c, w$)
Require: $\mathcal{G} = (V, \Sigma, R, I)$, programs bank B, cost c, and a cost function w.
Ensure: Program of cost c

```
13: for r ∈ R do
14:    if arity = 0 and w(r) = c then
15:       yield r
16:    else if arity > 0 and w(r) < c then
17:       for (p_1, ··· , p_k) in B × ··· × B do #operation over the values of the dictionary B
18:          if w(r(p_1, ··· , p_k)) = c and (p_1, ··· , p_k) is type-consistent with r then
19:             yield r(p_1, ··· , p_k)
```

the cost of `concat` *is* 14.28771 *and the cost of* `1000` *is* 9.8165*, which gives* $14.28771 + 2 \times 9.8165 = 33.920888 \approx 34$*. In the third iteration, it generates programs of cost* 58 *and it finds the solution program* `concat(concat(1000, 1000), 1000)`*.*

Unlike the size-based enumeration, the cost-guided BUS is biased toward cheaper programs according to the cost function w, which often allows the search to solve the problem while possibly generating many fewer programs compared to the size-based methods.

The pseudocode for generic cost-guided BUS is given in Algorithm 2. It receives a grammar $\mathcal{G}$, a set of input-output examples ($\mathcal{I}$, $\mathcal{O}$), and a cost function w to guide the search. It starts by initializing cost $c = 1$, and in each iteration of the main loop (lines 2-11), it calls the procedure Next-Program to generate programs with a target w-value of c. The cost c increases by one after each iteration (line 11). Next-Program procedure iterates over all the rules $r \in R$ of grammar $\mathcal{G}$ to generate programs with the target cost c. If the arity of the rule r is 0, i.e., it is a terminal rule, and the cost of the rule $w(r) = c$, then it returns the program given by rule r (lines 14-15). Otherwise, if the arity of the production rule is greater than 0, and the cost of the rule is less than the desired cost $w(r) < c$, then Next-Program generates all programs with the target cost c given by the rule r with

the parameters given by the Cartesian product of all programs in the bank B. Similarly to Algorithm 1, Next-Program considers only the programs $p_1, \cdots, p_k$ from B that are type-consistent with r. For example, when generating programs p with rule $I \rightarrow \texttt{concat}(I, I)$, Next-Program only considers programs that return strings as subprograms of p.

Once the program p is yielded to the Cost-Guided-BUS procedure, the program is executed (line 4) and, if it satisfies the desired output $\mathcal{O}$, then it is returned as a solution to the task. Otherwise, the algorithm checks for observational equivalence, if p is not observational equivalent to any other program in the bank B, then p is stored in B indexed by its cost c. Post-generation functions, such as the one used in Bustle, use a temporary cost $w(p)$ during the generation of programs p in Next-Program (line 16) and assign a new cost $w'(p)$ once the program is generated (line 9). The new cost $w'(p)$ is then used to store the programs in B. Once the program p is added to B, the main loop is repeated until the search finds the solution program p or it exhausts the time allowed for synthesis.

Algorithm 2 generalizes Probe, Bustle, and TF-Coder. It is equivalent to Probe if it receives the cost function of Probe. We note that Probe learns a cost function during the search, while Algorithm 2 assumes a fixed pre-generation cost function. It is equivalent to Bustle if it receives the post-generation cost function of Bustle. Finally, it is equivalent to TF-Coder if it receives TF-Coder's hand-crafted cost function, which is also a pre-generation function.

4.2.1 Lack of Best-First Ordering for Probe and Bustle

Both Probe and Bustle lose information because they round off the costs of the programs. As a result, the order in which they search over programs of a given cost is arbitrary, not best-first. In the PCFG given in Figure 2, `1000` has a lower cost than `1`, `2`, $\cdots$, `999`, but when rounded, their costs become equal, i.e., 10, and they are enumerated in an arbitrary order, while according to the cost function `1000` should be evaluated before `1`, `2`, $\cdots$, `999`. Since the number of programs with the same rounded cost can increase rapidly as the search grows, the algorithms' rounding off scheme can substantially slow down the synthesis process. In the example of synthesizing `concat(concat(1000, 1000), 1000)`, depending on how the ties are broken, Probe might evaluate more than 910,000,000 programs before finding the solution. By contrast, Bee Search evaluates 1,001,001 programs to find the solution.

One could achieve a "near best-first search" if the costs were multiplied by a large constant (e.g., 1,000,000) before being rounded off. The issue with this approach is that, due to the large number of different costs, there would be many iterations of Probe and Bustle that no program would be generated (similar to how Algorithm 2 did not generate any program with cost $[1, 9]$ in Example 5); we refer to these iterations as *sterile iterations*. Probe and Bustle still pay the computational cost of checking whether there are programs to be generated of a particular cost. Given that the target program cost of a given iteration is c and that the production rule r with k non-terminal symbols costs c', Probe checks all combinations of programs $(p_1, \cdots, p_k)$ whose added cost is $c - c'$, so that the non-terminal symbols of r can be replaced by $(p_1, \cdots, p_k)$ and the total cost of the generated program is c; Bustle follows the same approach, but using its cost function. The task of finding a subset of numbers that adds to a target value is NP-Complete (Garey & Johnson, 1990)

and finding such subsets is exponential in the number of non-terminals k. Although k can be small (e.g., for `concat` $k = 2$), computing the subsets can still hamper the performance of the algorithm if the subsets have to be computed many times during the search.

We performed preliminary experiments with a modified version of Probe that uses the "large-constant trick" and discovered that the approach is too slow to be practical: most of the computational effort is spent computing the subsets with target cost values for which no program can be generated. The algorithm we introduce in this paper, Bee Search, bypasses the NP-Complete problem of finding a subset of numbers that adds to a target value by performing a search in a cost-tuple space (see Section 5 for details).

4.3 Heap Search

Heap Search (Fijalkow et al., 2022) performs best-first bottom-up synthesis with respect to a cost function w defined with a PCFG. It achieves best-first enumeration by using a set of priority queues, which we denote as Heap, one for each non-terminal symbol T. We say that the programs derived from T are of type T. Each queue in Heap contains programs of type T that are sorted according to the programs' costs. Heap Search also uses a set of programs already seen in search (Seen) and a hash table (Succ), to store the *successors* of all programs of type T seen in search. The successor of a program p, denoted p', is the next cheapest program of type T to be generated, i.e., a program generated with a production rule for T with $w(p') > w(p)$ such that there is no p'' with $w(p') > w(p'') > w(p)$.

Example 6. *Consider an example of* Heap Search *using the following DSL.*

$$I \to 1 \,|\, 2 \,|\, I + I\,.$$

Here, $w(1) < w(2) < w(I \to I + I)$, and similarly to Probe*, the cost of a program p is given by the sum of the cost of the production rule and its subprograms p_i.* Heap Search *first generates programs `1`, `2`, and `1+1` (the cheapest program which can be generated using $I + I$) and add them to* Heap_I*, a heap structure storing programs of type I (this DSL only has programs of type I). The programs are sorted according to their cost w. Then, it first evaluates `1` (cheapest program), followed by the next cheapest program, `2`. Once `2` is removed from the heap,* Heap Search *sets `2` as the successor of `1` in the* Succ *hash table, i.e.,* Succ[`1`] = `2`*. Next, it pops `1+1` out and `1+1` is assigned as the successor of `2`. Since `1+1` was derived from a non-terminal production rule ($I \to I + I$),* Heap Search *generates `1+1`'s children by replacing each subprogram p of `1+1` with p's successor. That is, it first replaces `1` (the first subprogram) with its successor (`2`) to generate `2+1`. Then,* Heap Search *replaces the second subprogram with its successor and `1+2` is generated. Both of these programs are added to* Heap_I*. In the next iteration,* Heap Search *pops out the next program from* Heap_I *and continues the search until a solution program p is removed from* Heap_I *or it times out and it returns failure.*

The pseudocode for Heap Search, adapted from Fijalkow et al. (2022), is shown in Algorithm 3. The algorithm starts by initializing all data structures Heap_T, Seen_T, and Succ_T with the programs given by the terminal symbols p of $\mathcal{G}$. The structures are also initialized with the cheapest program of each type T, which is given by production rules $T \to r(T_1, \cdots, T_k)$. Each subprogram of type T_i in $r(T_1, \cdots, T_k)$ is given by the cheapest

Algorithm 3 Heap Search Algortihm

Procedure: HEAPSEARCH($\mathcal{G}$, $(\mathcal{I}, \mathcal{O})$, w)
Require: $\mathcal{G} = (V, \Sigma, R, I)$, input-output examples $(\mathcal{I}, \mathcal{O})$, and a cost function w.
Ensure: Solution program p or $\perp$

1: **for all** non-terminal symbols $T \in V$ **do**
2:   Create an empty min heap HEAP_T
3:   Create an empty hash table SUCC_T
4:   Create an empty set SEEN_T
5:   **for all** derivation rules of $T \to p$ **do**
6:     Add p to HEAP_T with priority $w(p)$
7:     Add p to SEEN_T
8:   **for all** derivation rules $T \to r(T_1, \cdots, T_k)$ **do**
9:     $p \leftarrow r(\text{HEAP}_{T_1}.\text{top}(), \cdots, \text{HEAP}_{T_k}.\text{top}())$
10:     Add p to HEAP_T with priority $w(p)$
11:     Add p to SEEN_T
12: $p \leftarrow \emptyset$
13: $T \leftarrow I$
14: **while** not timeout **do**
15:   $p \leftarrow \text{QUERY}(p, T)$
16:   $o \leftarrow \text{EXECUTE}(p, \mathcal{I})$
17:   **if** o equals $\mathcal{O}$ **then**
18:     **return** p
19:   $T \leftarrow \text{type}(p)$
20: **return** $\perp$

Procedure: QUERY(p, T)
Require: A program p, and type T of the program.
Ensure: Next cheapest program p'

21: **if** p is a key in SUCC_T **then**
22:   **return** $\text{SUCC}_T[p]$
23: **else**
24:   $p' \leftarrow pop(\text{HEAP}_T)$ # *p' is of form $r(p_1, \cdots, p_k)$*
25:   $\text{SUCC}_T[p] \leftarrow p'$
26:   **for all** $i \in [1, \cdots, k]$ **do**
27:     $y_i = \text{QUERY}(p_i, T_i)$
28:     $p'_i = r(p_1, \cdots, p_{i-1}, y_i, p_{i+1}, \cdots, p_k)$
29:     **if** p'_i is not in SEEN_T **then**
30:       Add p'_i to HEAP_T with priority $w(p'_i)$
31:       Add p'_i to SEEN_T
32: **return** p'

program generated with a terminal symbol of type T_i. For example, for type T_i we use $\text{HEAP}_{T_i}.top()$ as all heaps are already initialized by the terminal symbols. In our example,

the rule $I \to I + I$ generated the program `1 + 1` because program `1` was the cheapest program generated with a terminal symbol.

Heap Search invokes Query while there is still time allowed for synthesis. Query receives a program p and its type T as input; it returns the successor of p. Heap Search initially calls Query with an empty program and the initial symbol of the grammar, I. Then, it calls Query with the program it returned on its last call. Each call to Query returns the next program according to the best-first ordering of the programs given by w. Each program Query returns is evaluated and, if it represents a solution, the program is returned; Heap Search returns failure, $\perp$, if it times out before finding a solution.

The Query procedure returns the successor of the program p passed as input and it recursively generates the children of p's successor. The base case of the recursion is when the successor of p is already stored in $\text{Succ}_T[p]$ (lines 21 and 22). If the successor p' is not in Succ_T, then it is removed from Heap_T and p' is set as the successor of p in Succ_T. In Example 6, the successor of `2` is `1 + 1`; the latter was popped out of Heap_T when Query was invoked for `2`. Query then generates all children of p', by replacing each of p''s k subprograms with the successors of its subprograms. The subprogram successors are obtained by calling Query recursively (line 27). In our example, the children of `1 + 1` were `2 + 1` and `1 + 2`. The subprogram `2` of `2 + 1` was obtained by calling Query with the program `1`; program `2` was returned as the base case of the recursion as $\text{Succ}_T[1] = 2$.

Heap Search only inserts a newly generated program if it has not been added to a Heap before. This is achieved by storing in the hash table Seen all programs generated in search (lines 29–31). Note that Heap Search only does not re-insert in a Heap the exact programs that were seen before. This is different from the equivalence check BUS algorithms perform. While BUS algorithms would disregard the program `1 + 1` because it is observational equivalent to `2`, Heap Search considers both programs in search. Heap Search provably evaluates programs in a best-first ordering according to a pre-generation cost function.

4.3.1 Limitations of Heap Search

Heap Search sacrifices the ability of remove observational equivalent programs to attain best-first ordering with respect to a pre-generation cost function. If Heap Search performed equivalence check, it would no longer be a complete algorithm as it could fail to find a solution even for solvable problems. In our example, `1 + 1` would be eliminated as it is observational equivalent to `2` and Heap Search would not generate the children of `1 + 1`, which cannot be generated through another branch of the search.

Moreover, Heap Search is guaranteed to *evaluate* programs in a best-first order, but it does not *generate* programs in the best-first order. Whenever Query is called, it returns a single program that is evaluated, however, it generates many other programs (lines 26-31) that might never be evaluated because they are more expensive than the cost of a solution program.

In addition, Heap Search was not designed to search with post-generation functions such as w_{Bustle}. If the algorithm is modified to handle post-generation cost functions, then it would not be able to search in a best-first order as its proof implicitly assumes a pre-generation cost function.

Algorithm 4 Brute Search

Procedure: BRUTE($\mathcal{G}, (\mathcal{I}, \mathcal{O}), w$)
Require: $\mathcal{G} = (V, \Sigma, R, I)$, input-output examples $(\mathcal{I}, \mathcal{O})$, and a cost function w.
Ensure: Solution program p or $\perp$

1: $n_0 \leftarrow \{T | T \in \Sigma \wedge I \rightarrow T\}$
2: **if** a program p in n_0 is a solution **then**
3:    **return** p
4: Add n_0 to Q with priority 0 (highest priority possible)
5: B = {}
6: **while** not timeout **do**
7:    $n \leftarrow Q$.pop() # *node n can represent one or multiple programs*
8:    **for** each child program p of n **do**
9:       $o \leftarrow$EXECUTE$(p, \mathcal{I})$
10:       **if** o equals $\mathcal{O}$ **then**
11:          **return** p
12:       **if** o not in B **then**
13:          B.add(p, o)
14:          Add node representing p to Q with priority $w(p)$
15: **return** $\perp$

4.4 Brute Search

BRUTE is a best-first search algorithm we adapted to inductive program synthesis that is loosely inspired by the inductive logic programming system of the same name (Cropper & Dumančic, 2020). Let us consider an example with the DSL from Example 6: $I \rightarrow 1 \,|\, 2 \,|\, I{+}I$.

In BRUTE, the root of the tree represents all programs given by symbols appearing in terminal production rules. In our example, the root represents the programs `1` and `2`. The root is the only node representing multiple programs; all other nodes in the tree represent one program. The set of children of a node n in the BRUTE search tree is defined as follows. For each non-terminal production rule we generate all possible programs given by the Cartesian product of all programs seen in search where at least one of the subprograms of the children is given by a program n represents. In our example, the children of the root are given by the Cartesian product of programs `1` and `2` with rule $I \rightarrow I + I$: `1 + 1`, `1 + 2`, `2 + 1`, and `2 + 2`. The children nodes representing programs `1 + 1` and `2 + 1` are pruned because they are observational equivalent to `2` and `1 + 2`, respectively. The next layer of the tree is generated following the same procedure. For example, the children of the node representing `1 + 2` are given by the Cartesian product of all programs observed in search as subprograms of the production rule $I \rightarrow I + I$, where at least one subprogram is `1 + 2`. The children of the node representing `1 + 2` are: `1 + (1 + 2)`, `2 + (1 + 2)`, `(1 + 2) + 1`, `(1 + 2) + 2`, `(1 + 2) + (2 + 2)`, `(2 + 2) + (1 + 2)`; after pruning observational equivalent programs, we obtain: `2 + (1 + 2)` and `(1 + 2) + (2 + 2)`.

The pseudocode of BRUTE is shown in Algorithm 4. The root of the tree, n_0, is defined as the set of programs given by the terminal rules (line 1); if any of these programs p represents a solution, then p is returned. Otherwise, n_0 is added to a priority queue Q. In every

iteration of the algorithm, Brute pops the cheapest node n from Q (line 7) and generates its children, as illustrated in our example above. Each child of n represents a program p. If p is a solution, then it is returned. Otherwise, if p is not observational equivalent to another program in B (line 12), then (i) Brute adds p to the bank of programs B (line 13) and (ii) a node representing p is added to the queue with priority $w(p)$ (line 14).

4.4.1 Limitations of Brute

Similarly to Heap Search, Brute only evaluates programs in best-first order and does not generate programs in best-first order. In each iteration, it evaluates a single program but possibly generates many more. In Brute the branching factor can be very large compared to Heap Search since it considers all the programs evaluated so far in the Cartesian product used to generate the children of a node. This can substantially slow down the synthesis process and increase its memory usage because it generates many programs that will never be evaluated, as they can be more expensive than the solution program.

5. Best-First Bottom-Up Search (Bee Search)

Bee Search attains best-first ordering with respect to the generation of programs by performing a search in a cost-tuple space, which is explained in the next section.

5.1 Cost-Tuple Space

Bee Search searches over a set of cost-tuple spaces to determine the next program to be generated during search. We define one cost-tuple space for each non-terminal rule. A state in a cost-tuple space is defined by a tuple with k integers in $\mathbb{N}$ (we use 1 as the index of the first element of an array), where k is the number of non-terminal symbols in the production rule. For example, (i_1, i_2) represents a state in the cost-tuple space of the rule $I \rightarrow \texttt{concat}(I, I)$; i_1 and i_2 represent indexes in an ordered set C that contains the cost of all programs generated in search and it is sorted from the smallest to the largest cost. We use the words 'state' and 'cost-tuple state' interchangeably. Since each state is related to a production rule r, we denote by $n.r$ the production rule associated with n.

Each cost-tuple state represents a set of programs that are to be generated. For example, the state $(1, 1)$ of the space for $I \rightarrow \texttt{concat}(I, I)$ represents all programs that can be generated by replacing the first and second non-terminal symbols of `concat` by the cheapest programs encountered in search. According to the costs of the PCFG shown in Figure 2, `1000` is the program with the lowest w-value encountered in the space defined by the DSL (w-value of 9.8165). Initially, $C = \{9.8165\}$ and the cost-tuple state $n = (1, 1)$ for $I \rightarrow \texttt{concat}(I, I)$ represents the program `concat(1000, 1000)` and the cost-tuple state's w-value can be computed as $w(n) = 14.28771 + 9.8165 + 9.8165 = 33.92071$. For additive cost functions, the w-value of the state is equal to the w-value of the programs that the state represents. Although only `1000` costs 9.8165, each state $n = (i_1, i_2, \cdots, i_k)$ can represent multiple programs, as there might be multiple programs with the i-th cost in C.

Bee Search uses a priority queue Q that is initialized with one cost-tuple state $(1, \cdots, 1)$ for each non-terminal rule, where the size of the tuple matches the number of non-terminal rules on the right-hand side of the rule. Each cost-tuple state represents a set of programs

with a given cost w; therefore, the priority queue is sorted according to the w-value of each cost-tuple state. In every iteration, BEE SEARCH pops the cheapest cost-tuple state $n = (i_1, i_2, \cdots, i_k)$ from Q and generates all programs n represents. If none of these programs represents a solution to the program synthesis task, then the children of n are generated and inserted in Q. The children of n are given by the set of states that differ from n with the addition of 1 to an entry of n: $\{(i_1+1, i_2, \cdots, i_k), (i_1, i_2+1, \cdots, i_k), \cdots, (i_1, i_2, \cdots, i_k+1)\}$. We say that a cost-tuple state n is expanded when its children are generated.

Let us consider the following example.

Example 7. *Table 1 shows a trace of* BEE SEARCH *for the problem of synthesizing the program* `concat(1000, concat(1000, 1000))`*. In this example, we will consider the cost function* w_{PROBE} *described in Figure 2. The table shows updates to the* BEE SEARCH*'s cost list* C *and priority queue* Q*, programs generated, and the number of programs generated in each iteration (Count). The entries in column* Q *are cost-tuples of the form [cost, cost-tuple]; costs are truncated to four decimal places and the name of production rule is omitted from the cost-tuple for brevity. The number of entries in a cost-tuple indicates the arity of the production rule; the empty cost-tuple* $()$ *represents all programs generated with a terminal rule and cost-tuples with two entries, e.g.,* $(1, 1)$*, represent states for* `concat`*.*

Itr. #	C	Q	Programs	Count
1	$\{9.8165\}$	$\{[9.9660, ()], [33.9207, (1, 1)]\}$	`1000`	1
2	$\{9.8165, 9.9660\}$	$\{[33.9207, (1, 1)]\}$	`1, 2, ···, 999`	999
3	$\{9.8165, 9.9660, 33.9207\}$	$\{[34.0702, (2, 1)], [34.0702, (1, 2)]\}$	`concat(1000,1000)`	1
4	$\{9.8165, 9.9660, 33.9207, 34.0702\}$	$\{[34.2197, (2, 2)], [58.0249, (1, 3)], [58.0249, (3, 1)]\}$	`concat(1000, 1), ... concat(999, 1000)`	1998
5	$\{9.8165, 9.9660, 33.9207, 34.0702, 34.2197\}$	$\{[58.0249, (1, 3)], [58.0249, (3, 1)], [58.1744, (3, 2)]\}$	`concat(1, 1), ...concat(999, 999)`	998001
6	$\{9.8165, 9.9660, 33.9207, 34.0702, 34.2197, 58.0249\}$	$\{[58.1744, (3, 2)], [58.1744, (2, 3)], [58.1744, (4, 1)], [58.1744, (1, 4)]\}$	`concat(1000, concat(1000, 1000))`	1

Table 1: The table shows the trace of BEE SEARCH for finding `concat(1000, concat(1000, 1000))` with the cost function described in Figure 2. The table shows the cost list C, priority queue Q, the generated programs, and the count of programs generated at each iteration.

BEE SEARCH *starts by initializing* C *with the cost of the cheapest program generated with a terminal rule.* Q *is initialized with empty cost-tuple states, with one state for each terminal rule that is not the cheapest, and with one cost-tuple state for each non-terminal rule. In this example, all programs generated with production rules* $I \rightarrow j$ *for* $j \in \{$`1, 2,` $\cdots$ `, 999`$\}$ *are represented in state* $[9.9660, ()]$*, while state* $[33.9207, (1, 1)]$ *represents the program* `concat(1000, 1000)`*.*[1] *In the first iteration,* BEE SEARCH *pops the cheapest node of* Q*, which represents the program* `1000`*, whose cost is* 9.8165*. In the second iteration, it removes the state*

1. BEE SEARCH would actually insert one cost-tuple state for each program generated with a terminal rule: $[9.9660, (), \texttt{1}], [9.9660, (), \texttt{2}], \cdots, [9.9660, (), \texttt{999}]$. For the interest of space we represent all these

$[9.9660, ()]$ *and generates 999 programs* `1, ⋯, 999` *with cost* 9.9660*; both costs are added to* C*.*

Next, the search removes $n = [33.9207, (1, 1)]$ *from* Q *and generates* `concat(1000, 1000)` *with a cost of* 33.9207*. Here,* 1 *in the cost-tuple refers to the first index of the cost set* C *and the only program we have with that cost is* `1000`*;* $(1, 1)$ *indicates that both arguments of* `concat` *should be of cost* 9.8165*, hence the program* `concat(1000, 1000)` *whose cost equals the sum of the costs of the subprograms* `1000` *and the cost of the operation* `concat`*. Since* n *represents a non-terminal rule, the node is expanded, thus generating the cost-tuple states* $\{[34.0702, (2, 1)], [34.0702, (1, 2)]\}$*. Similarly, in fourth and fifth iterations, the search generates all programs with cost* 34.0702 *and* 34.2197*, respectively. After generating all programs of each node, it adds their children to* Q*.* Bee Search *finds the solution program with cost* 58.0249 *in iteration 6. Note that there are other programs with similar cost (i.e.,* 58.1744*) and* Bee Search *is able to distinguish them from the solution node. The programs with cost* 58.1744 *are not even generated in memory, but only their cost-tuple states.*

Bee Search's ability to distinguish programs with similar cost values (e.g., 58.0249 and 58.1744) can make a substantial difference in terms of search running time. As an example, Probe is unable to distinguish 58.0249 (used in the sixth iteration of the example) and 58.1744 (the next cheapest cost) as both values are truncated to 58. As a result, Probe evaluates approximately one billion programs to find the solution for our example. By contrast, Bee Search evaluates only approximately one million programs to find the solution.

Further, the search in the cost-tuple space allows for a best-first search with respect to the generation of programs. At each iteration, Bee Search only generates the programs with the next cheapest possible cost. Unlike other best-first search algorithms such as Brute and Heap Search, Bee Search does not have to generate the programs to identify the ones that will be evaluated next in the search. This is achieved at the cost of generating cost-tuple states that might never be evaluated in search (i.e., cost-tuple states for which we do not generate their programs). This feature is an advantage because the branching factor in the cost-tuple space is much smaller than the branching factor in the program space. We show empirically the advantages of performing best-first search with respect to the generation of programs.

5.2 Search Algorithm

Algorithms 5 and 6 show the pseudocode for Bee Search. The algorithm receives a DSL, a set of input-output examples, and a monotonically increasing cost function w; it returns a solution p or failure $\perp$. The search starts by adding in C the w-value of the cheapest program generated with a terminal rule and initializing a priority queue Q with one state $(1, \cdots, 1)$ for each non-terminal rule (line 4 of Algorithm 5). Q also receives one state for each terminal rule; these states do not have a tuple associated with them (line 6) because they do not generate children in the cost-tuple space. The ordering of Q is defined by the w-values of the cost-tuple states.

cost-tuple states as $[9.9660, ()]$ in this example. Bee Search would also spend one iteration with each of these states, which we also simplify in this example to a single iteration.

Algorithm 5 Bee Search

Procedure: BeeSearch($\mathcal{G}, (\mathcal{I}, \mathcal{O}), w$)
Require: $\mathcal{G} = (V, \Sigma, R, I)$, input-output examples $(\mathcal{I}, \mathcal{O})$, and a monotonically increasing cost function w.
Ensure: Solution program p or $\perp$

1: $C \leftarrow \{\min_{T \in \Sigma} w(T)\}$
2: $Q \leftarrow \emptyset$ *# Sorted according to the w-values*
3: **for** each non-terminal rule S in $\mathcal{G}$ **do**
4: $\quad Q$.push($[S, (1, \cdots, 1)]$)
5: **for** each terminal symbol S in $\mathcal{G}$ **do**
6: $\quad Q$.push($[S]$)
7: **while** not timeout **do**
8: $\quad p, c \leftarrow$ Next-Program(B, Q)
9: $\quad o \leftarrow$ Execute($p, \mathcal{I}$)
10: $\quad$ **if** o equals $\mathcal{O}$ **then**
11: $\quad\quad$ **return** p
12: $\quad B[c]$.add(p)
13: **return** $\perp$

In every iteration of the algorithm (iteration of the while loop in Algorithm 5), it generates the next program p (see Algorithm 6), which is executed with the input values $\mathcal{I}$, thus obtaining the outputs o. If o matches the desired output $\mathcal{O}$, then p is a solution, and it is returned (line 11). If p is not a solution, it is added to the bank of programs B, which is indexed by the cost of the programs c (line 12). This process continues until either a solution is found or the search times out, in which case failure $\perp$ is returned (line 13).

Algorithm 6 defines the program that is evaluated next in search. We remove a state n with the smallest w-value from Q (line 1). If the state represents a terminal rule, then the state has no children (i.e., it does not have non-terminal symbols that can be used to generate new programs). In this case, we just return the program $n.r$ and its cost $w(n)$ (line 5 of Algorithm 6), where function r returns the righthand side of the rule state n represents. We expand n if it represents a non-terminal rule, i.e., we generate all children n' of n and we add them to Q if they were not already inserted in Q (lines 6–10).

If n represents a non-terminal rule, we generate the set of programs n represents and we return each program p as an iterator to Algorithm 5.[2] The programs n represents are generated by replacing each non-terminal symbol of n's rule with a program from B. Let $n[j]$ be the j-th value of n, the j-th non-terminal symbol of $n.r$ is replaced by a program with cost $C[n[j]]$. Given that B is indexed by the programs' cost, we obtain the programs $(p_1, \cdots, p_k)$ that replace the non-terminals of $n.r$ by taking the Cartesian product $B[C[n[1]]] \times \cdots \times B[C[n[k]]]$ (line 11). Similarly to Algorithms 1 and 2, Bee Search performs type checking while generating programs with the Cartesian product operation. We denote newly generated programs as $n.r(p_1, \cdots, p_k)$. Since all programs $B[C[n[i]]]$ have the

2. An iterator is implemented with the keyword "yield" and it allows a function f_1 return a value to a function f_2 and, once f_2 calls f_1 again, the execution in f_1 continues from the last "yield" executed.

Algorithm 6 NEXT-PROGRAM procedure

Procedure: NEXT-PROGRAM(B, Q)
Require: Bank of programs B, priority queue Q
Ensure: Next cheapest program p of cost c
1: $n \leftarrow Q.\text{pop}()$
2: **if** pre-generation cost function **then**
3: $\quad C \leftarrow C \cup w(n)$
4: **if** n represents a terminal symbol **then**
5: $\quad$ **return** $n.r$, $w(n)$
6: **for** i in $\{1, \cdots, k\}$ **do**
7: $\quad n' \leftarrow n$
8: $\quad n'[i] \leftarrow n'[i] + 1$
9: $\quad$ **if** n' is not a duplicate **then**
10: $\quad\quad Q.\text{push}(n')$
11: **for** $(p_1, \cdots, p_k)$ **in** $B[C[n[1]]] \times \cdots \times B[C[n[k]]]$ **do**
12: $\quad$ **if** $(p_1, \cdots, p_k)$ is type-consistent with $n.r$ **then**
13: $\quad\quad p \leftarrow n.r(p_1 \cdots, p_k)$
14: $\quad\quad$ **if** p is equivalent to any program in B **then**
15: $\quad\quad\quad$ **continue**
16: $\quad\quad$ **if** post-generation cost function **then**
17: $\quad\quad\quad$ Insert $w'(p)$ in C while keeping C sorted
18: $\quad\quad\quad$ **if** $w'(p) < C[i]$, where $i = \max_j\{j \in n | n \in Q\}$ **then**
19: $\quad\quad\quad\quad$ Heapify Q
20: $\quad\quad\quad$ **yield** p, $w'(p)$
21: $\quad\quad$ **else**
22: $\quad\quad\quad$ **yield** p, $w(p)$

same cost, the w-value of all programs $n.r(p_1, \cdots, p_k)$ matches the w-value of n. We discard observational equivalent programs (lines 14 and 15).

BEE SEARCH treats post-generation and pre-generation functions differently. For pre-generation functions, the w-value of all programs generated from state n is identical to the w-value of n. Thus, the cost of the programs can be added to the set C before generating them (lines 2 and 3). For post-generation functions, the w'-values are known only after generating the programs, so they are inserted in C in line 17. While for pre-generation functions the cost values are inserted in increasing order in C and they can simply be appended at the end of C, the costs w' are not necessarily generated in increasing order (programs generated from the same tuple n can have different w'-values). BEE SEARCH maintains C sorted with post-generation functions by inserting the w'-values in their correct positions in C. Let i be the largest index in a cost-tuple state in Q. If the w'-value inserted in C is smaller than the i-th value in C (denoted $C[i]$ in the pseudocode), then Q needs to restore its heap structure through a "heapify" operation. This is because, once the w'-value is inserted in C, some of the indexes j in the cost-tuple states in Q might not refer to the same $C[j]$ they referred to prior to the insertion of the new w'-value. In the worst case, the insert operation is linear in

the size of C. The heapify operation is more expensive because it is linear in the size of Q and Q tends to be much larger than C. However, in practice, the heapify operation is rarely performed. By keeping C sorted and Q with a valid heap structure, we can prove that BEE SEARCH is a best-first search algorithm also for post-generation functions (see Section 5.3). Each program and its cost are returned as an iterator in lines 20 and 22. If the w' function is encoded in a neural network, instead of evaluating one program at a time, we evaluate all programs generated from n in a batch for efficiency; the batch evaluation is not shown in the pseudocode.

5.3 Theoretical Guarantees

BEE SEARCH is complete, correct, and performs best-first bottom-up search with respect to an additive cost function w that can be either of the pre-generation or the post-generation type. In this section, we provide the proofs for these properties.

Lemma 1. *Let w be an additive cost function. If the values in C are sorted from smallest to largest, then the w-values of the cost-tuple states increase monotonically in* BEE SEARCH.

Proof. The children n' of a state $n = (i_1, i_2, \cdots, i_k)$ are identical n, except for one entry j in n that is incremented by 1. For additive functions we have $w(n) = K + \sum_{i=1}^{k} C[i]$, where K is the cost of the production rule n represents. Then, we have the following.

$$
\begin{aligned}
w(n) &= K + \sum_{i=1}^{k} C[i] \\
&= K + C[j] + \sum_{\substack{i=1 \\ i \neq j}}^{k} C[i] \\
&< K + C[j+1] + \sum_{\substack{i=1 \\ i \neq j}}^{k} C[i] \\
&= w(n')\,.
\end{aligned}
$$

The inequality is due to C being sorted from smallest to largest and the values in C being unique. □

The following theorems state that BEE SEARCH performs a best-first search with respect to the generation of programs for a family of w functions that includes w_{PROBE} (Theorem 1) and for a family of w functions that includes w_{BUSTLE} and w_{U} (Theorem 2).

Theorem 1. BEE SEARCH *generates programs in best-first order with respect to an additive pre-generation cost function w.*

Proof. We prove by induction in the iterations of search that BEE SEARCH expands all cost-tuple states n in best-first order with respect to w, which implies that the programs p generated from n are evaluated in best-first order because w is additive and thus $w(p) = w(n)$

for all p. The base case is Bee Search's first iteration when C is initialized with the w-value of a terminal symbol with the smallest w value. Since w is additive, no state n can have a value of w lower than $C[1]$. The inductive hypothesis states that all cost-tuple states up to the j-th expansion (excluding the j-th expansion) are processed in best-first order with respect to w. Since w is a pre-generation function, the cost values $w(n)$ are added to C once a cost-tuple state n is expanded, so C must be sorted in increasing order prior to the j-th expansion.

For the inductive step, we consider the j-th expansion. In the j-th expansion, Bee Search expands a cost-tuple state n_1 with the smallest w-value in Q. Let us suppose that there is another state n_2 that was not expanded before n_1 and $w(n_2) < w(n_1)$ (i.e., Bee Search would have to expand n_2 instead of n_1 to attain best-first ordering). Since C is sorted from smallest to largest (inductive hypothesis) and $w(n) < w(n_2) < w(n_1)$ for any ancestor n of n_2 (Lemma 1), either n_2 or one of its ancestors n would have the smallest w-value in Q and not n_1, which is a contradiction, since n_1 has the smallest w-value in Q. Thus, the search expands the cost-tuple states in best-first order with respect to w, which implies that it generates programs in best-first order with respect to w ($w(n) = w(p)$ for all programs p generated from n). □

Theorem 2. Bee Search *generates programs in best-first order with respect to an additive and penalizing post-generation cost function* w*.*

Proof. During a Bee Search search with penalizing post-generation cost functions, the values of $C[i]$ for a fixed i can change across iterations due to the penalization term of w' and the sorting Bee Search performs. We prove by induction in the iterations of the search that, at the time of expansion of a cost-tuple state $n = (i_1, i_2, \cdots, i_k)$, $C[i]$ has its minimum value for all indexes i in n. By proving that the $C[i]$-entries have their minimum values, we show that the $C[i]$-values cannot change later in search for all i in n. Since Bee Search maintains C sorted and the $C[i]$-values cannot change, we can use Lemma 1 to show that the cost-tuple expansions happen in best-first order with respect to w, which implies a best-first order for the generation of programs as $w(n) = w(p)$ for all p generated from n. In our proof we consider cost-tuple states representing non-terminal rules since states representing terminal rules do not generate children, and thus the priority queue alone guarantees the best-first ordering for such states. The base case is the first cost-tuple state $n = (1, \cdots, 1)$ expanded. Since $C[1]$ contains the cost of the terminal symbol with the smallest w-value and the w function is additive, no other program can have a smaller w-value. The inductive hypothesis states that the $C[i]$-values have their minimum value and are sorted for all indexes i of states Bee Search expands up to the j-th expansion.

Let n_1 be the cost-tuple state with the smallest w-value in Q in the j-th expansion. Let us suppose that, at a given iteration, due to the order in which Bee Search inserts cost values in C, there is an i in n_1 for which $C[i]$ does not have its minimum value. That is, there exists a cost-tuple state n_2 that was not expanded yet and that will generate a program p whose $w'(p)$-value will be assigned to $C[i]$, and before this assignment happens, we have $C[i] > w'(p)$. Penalizing cost functions guarantee that the value of a program cannot be smaller than the value of the cost-tuple state that generated the program, i.e., $w'(p) \geq w(n_2)$, for a p generated from n_2. Since $w(n_1)$ is given by the sum of costs of its subprograms (w is additive) and $w'(p)$ is one of the terms of the sum that results in $w(n_1)$,

then $w'(p) < w(n_1)$ and thus $w(n_2) < w(n_1)$. Since Bee Search always maintains Q with a valid heap structure and the cost function increases monotonically for the cost-tuple states (inductive hypothesis and Lemma 1), n_2 and all its ancestors must have been expanded prior to n_1 and the $C[i]$-values for all i in n_1 must be at their minimum value when n_1 is expanded.

Since the $C[i]$-values are sorted and final for all i in the states Bee Search expands, Lemma 1 gives us that the cost-tuple states are expanded in best-first order according to w, which implies that the programs are generated in best-first order according to w. □

The next theorem shows that Bee Search search is complete, i.e., if there is a solution is the space of programs $\mathcal{G}$ defines, Bee Search will eventually find it.

Property 1. *Given enough memory and time, if a solution program p exists in the search space defined by the grammar $\mathcal{G}$, then* Bee Search *will find it.*

Proof. Bee Search considers all possible cost-tuple states in the search—it does not leave any state unchecked. Since every program that can be derived from $\mathcal{G}$ is mapped to a cost-tuple state, Bee Search considers all possible programs during search. □

We begin by discussing the correctness of Bee Search by showing that all indexes stored in cost-tuple states refer to valid positions in the cost set C, despite C being dynamically constructed during search. This means that Bee Search never accesses an index that is outside the range of $[1, |C|]$, and thus it does not throw a runtime error for that reason.

Property 2. *For monotonically increasing cost functions, the indexes i_j in the cost-tuples $(i_1, i_2, \cdots, i_k)$ generated during the* Bee Search *search are valid, i.e., i_j in $[1, |C|]$.*

Proof. The proof is by induction in the iterations of search. C is initialized with the cost of the cheapest terminal symbol, so all tuples $(1, \cdots, 1)$ refer to a valid index. The inductive hypothesis is that prior to the j-th iteration of Bee Search all indexes i in all tuples $(i_1, i_2, \cdots, i_k)$ generated thus far in search are valid. In the j-th iteration of Bee Search the cost-tuple state n is to be expanded and, by the inductive hypothesis, all its indexes are valid, including the largest index, denoted i_{MAX}. Since all indexes are valid, we know that $|C| \geq i_{\text{MAX}}$. If $|C| > i_{\text{MAX}}$, then all children of n are trivially valid because each index in n grows at most 1 in n's children. If $|C| = i_{\text{MAX}}$, all children are also valid because when n is expanded, its cost is added to C, thus increasing the size of C by 1. The cost $C[i_{\text{MAX}}]$ is the largest in C before n is expanded. Since the cost function is increasing monotonically, $w(n) > C[i_{\text{MAX}}]$ (or $w'(p) > C[i_{\text{MAX}}]$, where p is a program generated from n, for post-generation cost functions). Thus, when n is expanded, the size of C increases by 1 and all children of n have valid indexes. □

The following theorem states that the program Bee Search returns is correct, i.e., it solves the program synthesis task.

Property 3. *If* Bee Search *returns a solution program p, then p is correct as it solves the program synthesis task: p satisfies the semantic and syntactic constraints of the task.*

Proof. It is trivial to establish that Bee Search is correct, the program p it returns must satisfy the semantic and syntactic constraints of the synthesis problem, as Bee Searchchecks for these constraints in line 10 of Algorithm 5 and only returns the solution program p (line 11) if the constraints are satisfied. □

6. Empirical Results

We evaluate Bee Search on three benchmark problems set: (i) 205 string manipulation problems—108 programming by example string problems from 2017 SyGuS competition, 37 real problems faced by people and posted on StackOverflow, and 60 spreadsheet problems from Exceljet (Lee et al., 2018) (we call this benchmark the SyGuS benchmark), (ii) 38 handcrafted string manipulation problems from Bustle's paper (Odena et al., 2021), and (iii) 27 bit-vector problems from the Hacker's Delight book (Warren, 2013). We have implemented Probe, Bustle, Brute with w_{Probe} and w_{Bustle}, Bee Search with w_{Probe}, w_{Bustle}, and w_{U}, Heap Search with w_{Probe}, and BUS.

Probe uses an online learning scheme in which the probabilities of the PCFG are updated as more input-output examples are solved. Probe uses a parameter to determine when to update the probabilities; we tested the values of $d = \{1, 2, \cdots, 7\}$ in all algorithms using w_{Probe}, and report the results for the value that performed best for each algorithm. Bustle uses a neural network with property signatures (Odena & Sutton, 2020). Property signatures are domain dependent and Odena et al. (2021) described properties only for the string manipulation domain, so we limit the experiments with techniques using w_{Bustle} and w_{U} to string manipulation problems only. However, we evaluate the approaches using w_{Probe} on both string and bit-vector problems. We report the average and standard deviation over 5 independent runs of all results involving the neural network of Bustle. BUS is deterministic, so is Probe's learning scheme, hence we report the results of a single run for them.

We are interested in comparing Bee Search using w_{Probe} with all other algorithms using w_{Probe} (Probe, Brute, and Heap Search) and Bee Search using w_{Bustle} with all other algorithms using w_{Bustle} (Bustle and Brute). We are also interested in comparing all algorithms performing best-first search: Bee Search, Brute, and Heap Search. Lastly, we are interested in comparing Bee Search using w_{U} with all other algorithms. We performed two sets of experiments: one with a smaller DSL and another with a larger one (see Appendix A for the DSLs). The larger DSLs are defined as follows. For each problem in the string domain of SyGuS benchmarks, instead of using only the literals given in the problem's specification, we use literals from all problems in the set and all letters in the English alphabet. For the 38 string manipulation problems, in addition to the aforementioned literals, 50 randomly generated strings of length selected uniformly at random from the range $[2, 7]$, and 10 integers selected uniformly at random from the range $[10, 100]$ are also added. For the bit-vector domain, we used 250 literals for all problems, obtained by taking the union of literals from all problems in the set and adding other 236 random literals. The goal of experimenting with larger DSLs is to evaluate the algorithms in larger search spaces. Larger DSLs also simulate scenarios in which one does not have access to the set of literals required to solve a problem, and more literals can increase the chances of defining spaces

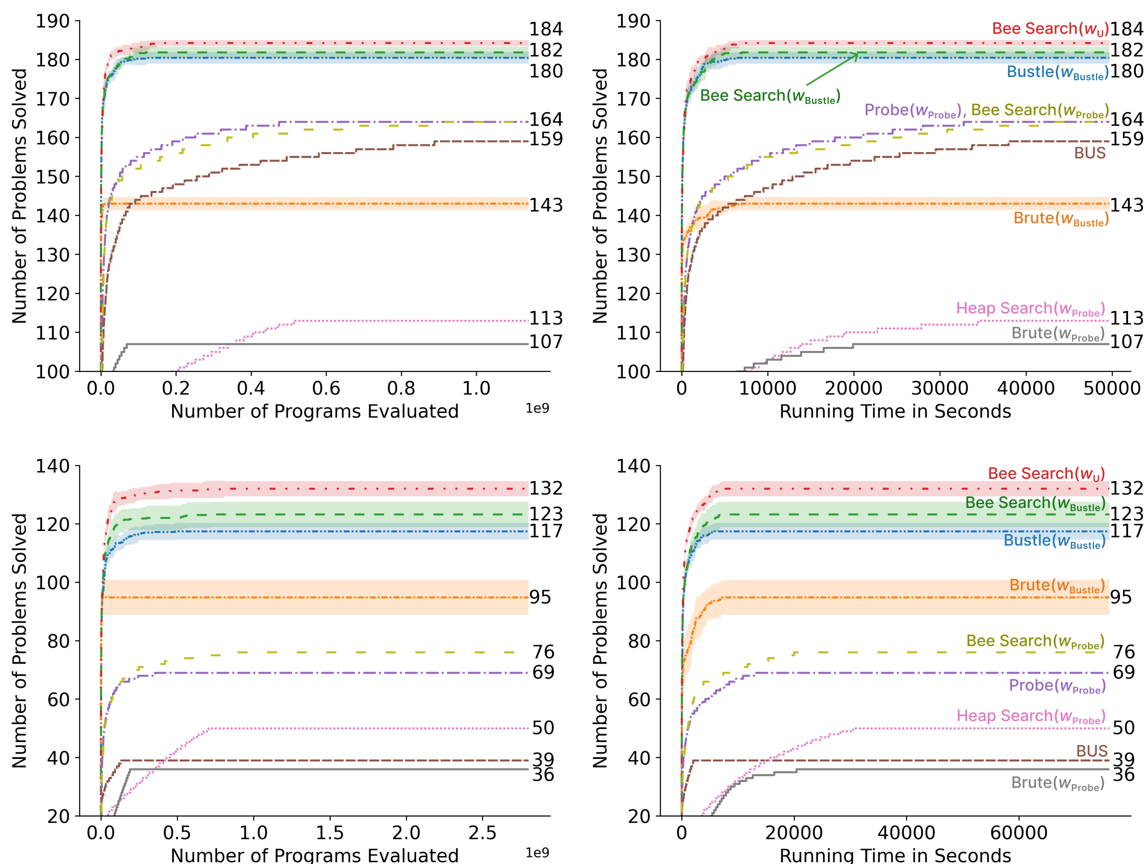

Figure 4: Number of problems solved per number of evaluations and running time for 205 string domain problems of SyGuS. The two plots at the top show the results for the smaller DSL; the plots at the bottom show the results for the larger DSL.

that contain a solution. All experiments were run on 2.4 GHz CPUs with 16 GB of RAM. The algorithms had 120 minutes to solve each task.

Figures 4 and 5 present the results for the string manipulation tasks from 205 SyGuS competition and 38 handcrafted benchmarks, respectively. Figure 6 shows the results of bit-vector domain. In each figure, the two plots at the top present the results for the smaller DSL, while the plots at the bottom present the results for the larger DSL. We present the number of problems solved by the number of programs evaluated and by the running time in seconds. While running time offers a fair evaluation of the different algorithms, the number of evaluations is machine- and implementation-independent, which allows it to be more easily used by others. The plots were generated by sorting the solved instances according to each algorithm's running time (or number of evaluations); the y-axis shows the total number of problems solved and the x-axis the sum of running times (or sum of number of evaluations).

6.1 Discussion

Our discussion is divided by our key findings.

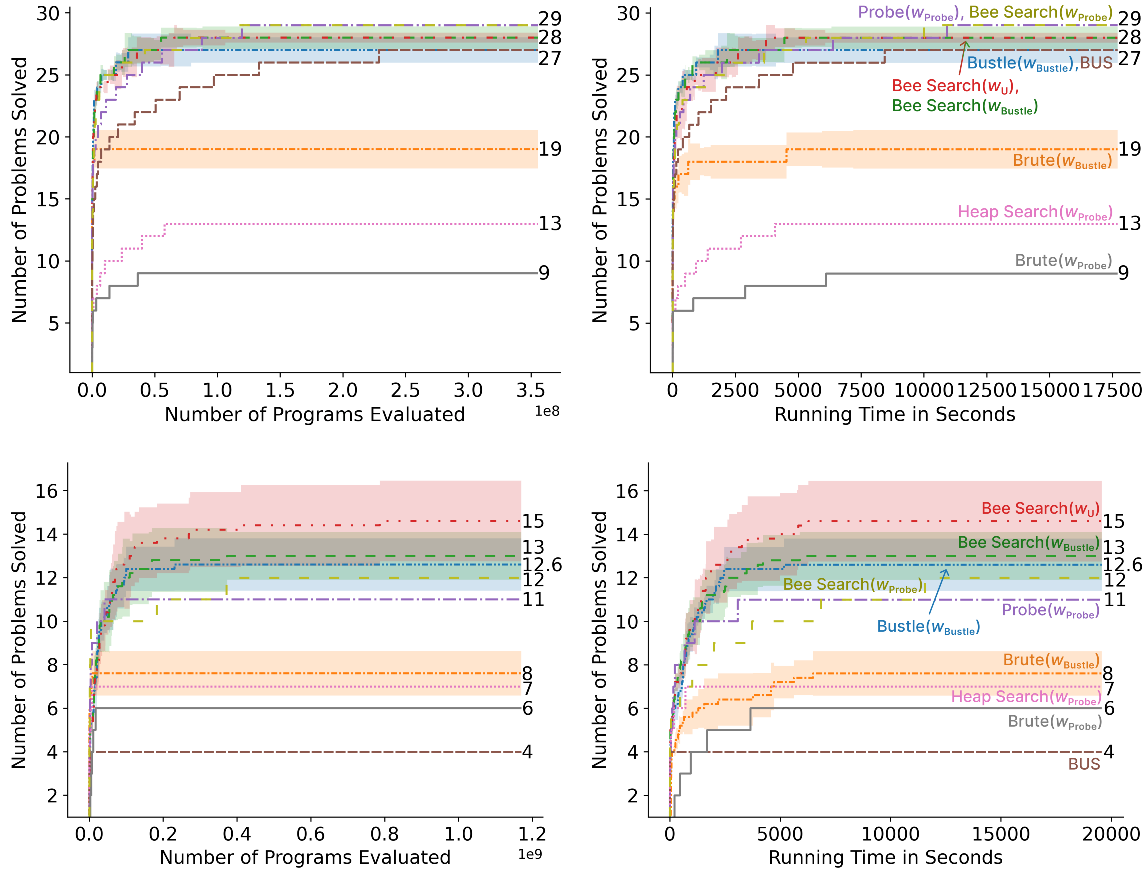

Figure 5: Number of problems solved per number of evaluations and running time for the handcrafted 38 strings benchmarks by Odena et al.. The two plots at the top show the results for the smaller DSL; the ones at the bottom show the results for the larger DSL.

Bee Search is not worse and often superior to others for a given cost function

For a given cost function, Bee Search never performs worse in terms of the number of tasks solved, and often performs better than the other algorithms. For the SyGuS benchmark (Figure 4), Bee Search with w_{Bustle} solves 182 and 123 tasks for the smaller and larger DSL, respectively, while Bustle solves 180 and 117. Brute solves only 143 and 95 tasks with the same function. Similarly, Bee Search with w_{Probe} is never worse than the other algorithms using w_{Probe}: it solves the same number of problems Probe solves for the smaller DSL and outperforms all algorithms in the larger DSL. We observe similar results in the 38 tasks (Figure 5) and in the bit-vector tasks (Figure 6).

Bee Search with either w_{U} or w_{Probe} performs best in the evaluated domains

For the SyGuS benchmark (Figure 4), Bee Search with w_{U} solves more tasks than any other algorithm: 184 with the smaller DSL and 132 with the larger DSL. The second best algorithm in this domain is Bustle, which solves 180 tasks with the smaller DSL and 117 tasks with the larger DSL. Although the difference in terms of the number of tasks solved

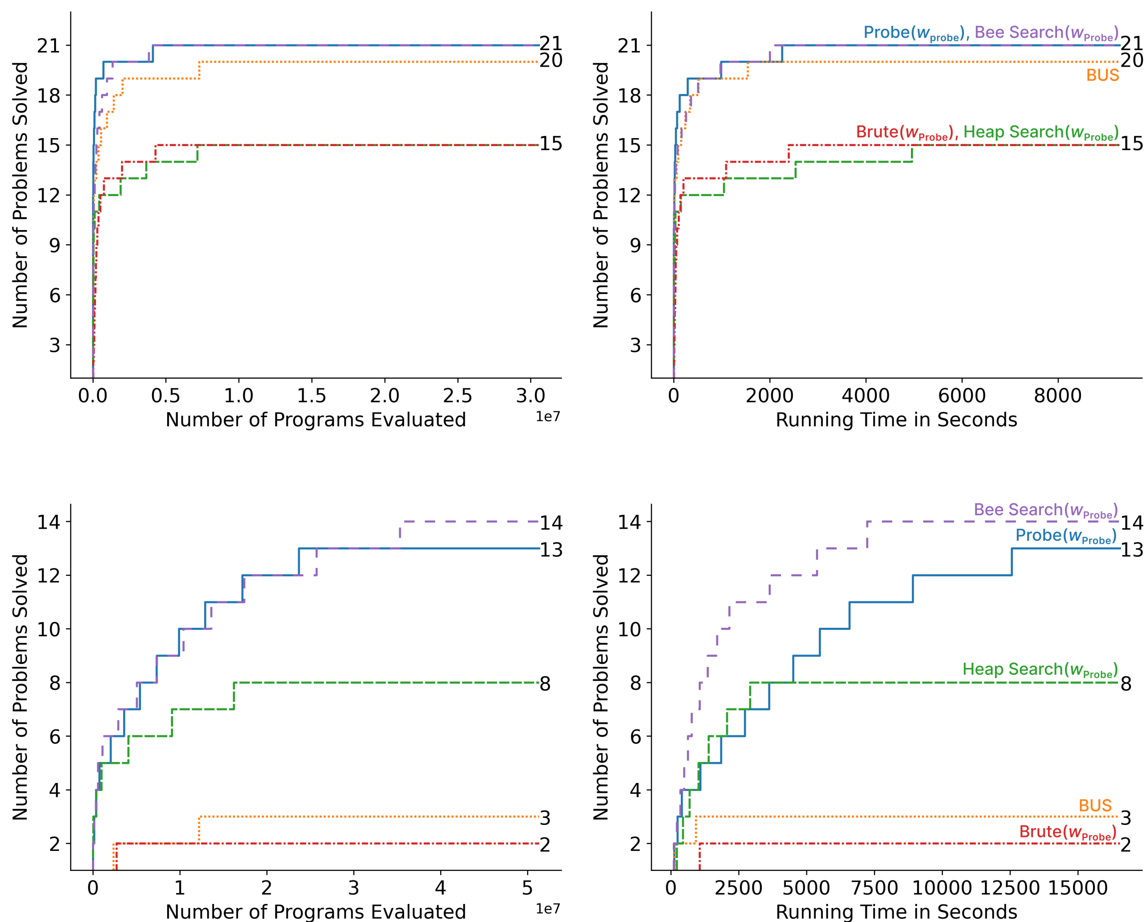

Figure 6: Number of problems solved per number of evaluations and running time for bit-vector domain. The two plots at the top show the results for the smaller DSL; the ones at the bottom show the results for the larger DSL.

may seem small, it is substantial given that the tasks Bee Search solves and Bustle fails to solve are hard. For the 38 handcrafted string tasks (Figure 5), Probe and Bee Search with w_{Probe} solve the largest number of tasks with the smaller DSL (29 tasks), while Bee Search with w_{Bustle} and w_{U} solve 28 tasks. Bustle comes next with 27 tasks solved. For the larger DSL, Bee Search with w_{U} solves the largest number of tasks, 15, and is followed by Bee Search with w_{Bustle} with 13; Bustle solves 12.6. A similar pattern is observed in the bit-vector domain (Figure 6), where Probe and Bee Search with w_{Probe} solve 21 tasks with the smaller DSL. For the larger DSL, Bee Search with w_{Probe} solves more problems than all other methods. The second best performing algorithm in bit-vector is Probe, with 13 problems solved, while Heap Search is the third best algorithm with 8 problems solved.

w_{U} performs better than w_{Bustle}

Bee Search with w_{U} outperforms all systems tested with w_{Bustle}, the other neural-based function, in the two evaluated domains: SyGuS benchmark and 38 string problems.

Bee Search overcomes the weaknesses of previous algorithms based on BUS

These results suggest that Bee Search's best-first scheme is able to better use the information cost functions provide to the search than the "truncation-based" algorithms Probe and Bustle. The results also suggest that Bee Search's scheme of searching in the cost-tuple space is effective as it outperforms Brute and Heap Search by a large margin in all domains. Brute suffers from the fact that it generates a large number of programs that are never evaluated in search, which increases the algorithm's memory and time requirements. Bee Search does not suffer from this problem because its best-first search is with respect to program generation. Bee Search generates cost-tuple states that are never expanded, but the number of such states is much smaller than the number of programs Brute generates and are not evaluated. This is because the cost-tuple space is an abstraction of the original program space, where many programs are mapped to the same cost-tuple space. We conjecture that Heap Search performs poorly in our experiments because it is unable to perform observational equivalence. In the SyGuS benchmark, even BUSsubstantially outperforms both Brute and Heap Search.

6.2 Limitations of Evaluation

In the context of ILP, Brute uses Answer Set Programming constraints to reduce the branching factor of search. The Brute version we evaluated in this paper is only an approximation of the original algorithm, as it is not clear how to adapt to Inductive Program Synthesis all the search enhancements developed in the context of ILP. Similarly, Heap Search was originally evaluated in the context of parallel programming. In this paper, we evaluated only the sequential version of all algorithms.

In some of our experiments, we observed that Bee Search performs as well as truncation-based algorithms (e.g., Probe in the bit-vector domain with the smaller DSL). We conjecture that these results can be explained by the nature of the cost function. For example, if the cost function does not provide helpful information for guiding the search, then both the exact and the truncated cost values will not be helpful for guiding the search. As another example, if a cost function is coarse-grained (e.g., all cost values are integers), then Bustle, Probe, and Bee Searchwill receive the same search signal to guide the search.

7. Related Work

The synthesis of programs has been studied for many years, starting with Waldinger and Lee (1969), Smith (1976), and Summers (1977). It was used to solve tasks such as synthesis of database transactions (Qian, 1990, 1993), system verification (Musser, 1989; Dybjer & Sander, 1990), logic programming (Kodratoff et al., 1990; Deville & Lau, 1994), manipulation of bits (Solar-Lezama et al., 2005; Gulwani & Venkatesan, 2009), strings (Gulwani, 2011), numbers (Singh & Gulwani, 2012), synthesis of fault-tolerant circuits (Eldib et al., 2016), and programmatic policies (Verma et al., 2018; Bastani et al., 2018; Mariño et al., 2021).

Similarly to the application domains, there is also a large diversity of strategies for solving synthesis tasks. In constraint satisfaction algorithms, one transforms the synthesis task into a constraint satisfaction problem that can be solved with off-the-shelf SAT solvers (Solar-Lezama, 2009). Stochastic search algorithms such as Simulated Annealing (Husien & Schewe, 2016) and genetic algorithms (Koza, 1992) have also been applied to solve synthesis tasks. Stochastic search algorithms start with a candidate solution and use mutation operators to change that candidate into other candidates that might be closer to a solution. Enumerative algorithms systematically evaluate programs in the space defined by the DSL. We focus on enumerative algorithms since Bee Search is an enumerative method.

7.1 Enumerative Methods

Enumeration-based search has proven to be an effective approach and is used in many synthesizers (Odena et al., 2021; Barke et al., 2020; Lee et al., 2018; Albarghouthi et al., 2013; Udupa et al., 2013), including winners of SyGuS competitions (Alur et al., 2016, 2017). Enumerative methods can be classified into two categories: bottom-up and top-down. Bottom-up search (BUS) algorithms start with the shortest possible programs and use the rules of the symbolic language to generate longer programs by combining the shorter ones. BUS is an attractive search strategy because the programs generated are complete and thus can be executed, allowing observational equivalence checks to be performed (Albarghouthi et al., 2013; Udupa et al., 2013; Lee et al., 2018; Odena et al., 2021; Barke et al., 2020). Top-down search algorithms start with a high-level structure of the program and enumerate the low-level structures. Top-down enumeration can only utilize weaker forms of equivalence (Lee et al., 2018; Wang, Dillig, & Singh, 2017) since most of the programs generated in the search are incomplete and cannot be executed.

7.2 Guided Enumerative Search

In guided enumerative search, instead of enumerating programs according to their AST size, the algorithms prioritize programs according to a function. One of the first guided search methods for program synthesis, DeepCoder (Balog et al., 2016), uses a top-down search. It uses a learned model to define a probability distribution over symbols in the language. Then, it performs a depth-first search that first explores the branches with a higher probability according to the model. Euphony also uses a probability distribution over production symbols of the underlying context-free grammar defining the programming language to guide a top-down search (Lee et al., 2018), however, Euphony's model considers the context in which a production rule is to be applied using the idea of probabilistic higher-order grammar. While different variations of using a learned model to guide top-down search algorithms have been introduced in the past (Chen, Liu, & Song, 2019; Zohar & Wolf, 2018; Bunel, Hausknecht, Devlin, Singh, & Kohli, 2018; Devlin, Uesato, Bhupatiraju, Singh, rahman Mohamed, & Kohli, 2017b; Wang et al., 2017), empirical evidence shows that they fail to outperform guided bottom-up search techniques (Barke et al., 2020).

TF-Coder (Shi et al., 2020) was the first system to utilize a function to guide a BUS algorithm. TF-Coder requires one to manually assign weight values to the operations based on their usage and complexity. During the search, TF-Coder prefers to combine programs with lower weights than programs with larger weights, thus biasing the search; the weight

of a program is defined as the sum of the weights of the production rules used to generate the program. Since TF-Coder requires one to manually set weights for each operation, we did not consider it in our experiments. Similarly to Bustle and Probe, TF-Coder also suffers from loss of information, since it considers only integer cost values.

8. Conclusions

In this paper, we showed that some of the current guided BUS algorithms suffer from a common problem: they can lose useful information given by the cost function because they only consider integer-valued costs. As a result, these algorithms do not perform best-first search with respect to the cost function used in the search. Heap Search is a best-first guided bottom-up search algorithm that provably does not lose information from the cost function. However, Heap Search sacrifices a key feature of BUS algorithms, which is the ability to eliminate observational equivalent programs. We presented an algorithm loosely inspired by the system Brute, from the ILP literature, to program synthesis, which we also referred to as Brute. Brute is able to perform search in best-first order and eliminate observational equivalent programs. However, Brute's search is best-first with respect to the evaluation of programs. As a result, many programs are generated but never evaluated, as they are more expensive than the solution program. We introduced Bee Search, a novel guided BUS algorithm that is guaranteed to perform search in best-first order when employing additive pre-generation cost functions and penalizing additive post-generation functions. In addition to performing search in best-first order, Bee Search is able to eliminate observational equivalent programs and its best-first search is with respect to the generation of programs. That is, Bee Search does not generate programs that are more expensive than the solution program. We also introduced a cost function that uses the neural model of Bustle. The difference between our function and Bustle's is that the former does not bound the penalty applied in the post-generation evaluation, as Bustle's cost function does. Empirical results on string manipulation and bit-vector problems showed that Bee Search was never worse than Probe and Bustle and can substantially outperform them, especially in larger program spaces. Bee Search outperformed Heap Search and Brute by a large margin in both domains. Empirical results also showed that Bee Search with our cost function was the best performing system in the string manipulation domain.

Acknowledgements

We thank Thirupathi Reddy Emireddy for implementing the Bustle algorithm used in our experiments and the anonymous reviewers for their constructive feedback. This research was supported by Canada's NSERC and the CIFAR AI Chairs program. This research was enabled in part by support provided by the Digital Research Alliance of Canada.

Appendix A. Domain Specific Languages (DSLs)

A.1 String Domain DSL

$Start \rightarrow S \mid I \mid B$
$S \rightarrow \texttt{replace}(S, S, S) \mid \texttt{concat}(S, S) \mid \texttt{substr}(S, I, I)$
$\quad \mid \texttt{ite}(B, S, S) \mid \texttt{intToStr}(I) \mid \texttt{charAt}(S, I)$
$\quad \mid \texttt{toLower}(S) \mid \texttt{toUpper}(S) \mid \texttt{arg0} \mid \texttt{arg1} \mid \ldots$
$\quad \mid \texttt{lit}{-}0 \mid \texttt{lit}{-}1 \mid \ldots$
$I \rightarrow \texttt{strToInt}(S) \mid \texttt{add}(I, I) \mid \texttt{sub}(I, I) \mid \texttt{mul}(I, I)$
$\quad \mid \texttt{mod}(I, I) \mid \texttt{length}(S) \mid \texttt{indexOf}(S, S, I)$
$\quad \mid \texttt{ite}(B, I, I) \mid \texttt{find}(S, S) \mid \texttt{arg0} \mid \texttt{arg1} \mid \ldots$
$\quad \mid \texttt{lit}{-}0 \mid \texttt{lit}{-}1 \mid \ldots$
$B \rightarrow \texttt{true} \mid \texttt{false} \mid \texttt{isEqual}(I, I) \mid \texttt{isLess}(I, I)$
$\quad \mid \texttt{isGreater}(I, I) \mid \texttt{contains}(S, S)$
$\quad \mid \texttt{isSuffixOf}(S, S) \mid \texttt{isPrefixOf}(S, S)$

Figure 7: Baseline DSL for string domain used in this paper

A.2 BitVector Domain DSL

$Start \rightarrow BV \mid B$
$BV \rightarrow \texttt{xor}(BV, BV) \mid \texttt{and}(BV, BV) \mid \texttt{or}(BV, BV)$
$\quad \mid \texttt{neg}(BV) \mid \texttt{not}(BV) \mid \texttt{add}(BV, BV) \mid \texttt{mul}(BV, BV)$
$\quad \mid \texttt{udiv}(BV, BV) \mid \texttt{urem}(BV, BV) \mid \texttt{lshr}(BV, BV)$
$\quad \mid \texttt{ashr}(BV, BV) \mid \texttt{shl}(BV, BV) \mid \texttt{sdiv}(BV, BV)$
$\quad \mid \texttt{srem}(BV, BV) \mid \texttt{sub}(BV, BV) \mid \texttt{ite}(B, BV, BV)$
$\quad \mid \texttt{arg0} \mid \texttt{arg1} \mid \ldots \mid \texttt{lit}{-}0 \mid \texttt{lit}{-}1 \mid \ldots$
$B \rightarrow \texttt{true} \mid \texttt{false} \mid \texttt{isEqual}(BV, BV) \mid \texttt{ult}(BV, BV)$
$\quad \mid \texttt{ule}(BV, BV) \mid \texttt{slt}(BV, BV) \mid \texttt{sle}(BV, BV)$
$\quad \mid \texttt{ugt}(BV, BV) \mid \texttt{redor}(BV) \mid \texttt{and}(BV, BV)$
$\quad \mid \texttt{or}(BV, BV) \mid \texttt{not}(BV) \mid \texttt{uge}(BV, BV)$
$\quad \mid \texttt{sge}(BV, BV) \mid \texttt{sgt}(BV, BV)$

Figure 8: Baseline DSL for bit-vector domain used in this paper

References

Albarghouthi, A., Gulwani, S., & Kincaid, Z. (2013). Recursive program synthesis. In *International Conference Computer Aided Verification, CAV*, pp. 934–950.

Alur, R., Bodik, R., Juniwal, G., Martin, M., Raghothaman, M., Seshia, S., Singh, R., Solar-Lezama, A., Torlak, E., & Udupa, A. (2013). Syntax-guided synthesis.. pp. 1–17.

Alur, R., Fisman, D., Singh, R., & Solar-Lezama, A. (2016). Sygus-comp 2016: Results and analysis. In Piskac, R., & Dimitrova, R. (Eds.), *Proceedings Fifth Workshop on*

Synthesis, SYNT@CAV 2016, Toronto, Canada, July 17-18, 2016, Vol. 229 of *EPTCS*, pp. 178–202.

Alur, R., Radhakrishna, A., & Udupa, A. (2017). Scaling enumerative program synthesis via divide and conquer. In *Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems conference*, pp. 319–336. Springer Berlin Heidelberg.

Balog, M., Gaunt, A. L., Brockschmidt, M., Nowozin, S., & Tarlow, D. (2016). Deepcoder: Learning to write programs. *CoRR*, *abs/1611.01989*.

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

Bastani, O., Pu, Y., & Solar-Lezama, A. (2018). Verifiable reinforcement learning via policy extraction. In *Advances in Neural Information Processing Systems*, pp. 2499–2509.

Bunel, R., Hausknecht, M., Devlin, J., Singh, R., & Kohli, P. (2018). Leveraging grammar and reinforcement learning for neural program synthesis. In *International Conference on Learning Representations*.

Chen, X., Liu, C., & Song, D. (2019). Execution-guided neural program synthesis. In *International Conference on Learning Representations*.

Colón, M. A. (2004). Schema-guided synthesis of imperative programs by constraint solving. In *Proceedings of the 14th International Conference on Logic Based Program Synthesis and Transformation*, LOPSTR'04, p. 166–181, Berlin, Heidelberg. Springer-Verlag.

Cropper, A., & Dumančic, S. (2020). Learning large logic programs by going beyond entailment. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence*, pp. 2073–2079. International Joint Conferences on Artificial Intelligence Organization.

Deville, Y., & Lau, K.-K. (1994). Logic program synthesis. *The Journal of Logic Programming*, *19-20*, 321–350. Special Issue: Ten Years of Logic Programming.

Devlin, J., Uesato, J., Bhupatiraju, S., Singh, R., Mohamed, A., & Kohli, P. (2017a). Robustfill: Neural program learning under noisy I/O. In Precup, D., & Teh, Y. W. (Eds.), *Proceedings of the 34th International Conference on Machine Learning*, Vol. 70 of *Proceedings of Machine Learning Research*, pp. 990–998. PMLR.

Devlin, J., Uesato, J., Bhupatiraju, S., Singh, R., rahman Mohamed, A., & Kohli, P. (2017b). Robustfill: Neural program learning under noisy i/o. In *ICML*.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, *1*(1), 269–271.

Dybjer, P., & Sander, H. (1990). A functional programming approach to the specification and verification of concurrent systems. In Rattray, C. (Ed.), *Specification and Verification of Concurrent Systems*, pp. 331–343, London. Springer London.

Eldib, H., Wu, M., & Wang, C. (2016). Synthesis of fault-attack countermeasures for cryptographic circuits.. Vol. 9780.

Ellis, K., Wong, C., Nye, M. I., Sablé-Meyer, M., Cary, L., Morales, L., Hewitt, L. B., Solar-Lezama, A., & Tenenbaum, J. B. (2020). Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning. *CoRR*, *abs/2006.08381*.

Fijalkow, N., Lagarde, G., Matricon, T., Ellis, K., Ohlmann, P., & Potta, A. (2022). Scaling neural program synthesis with distribution-based search. In *AAAI*.

Fraňová, M. (1985). A methodology for automatic programming based on the constructive matching strategy. In Caviness, B. F. (Ed.), *EUROCAL '85*, pp. 568–569, Berlin, Heidelberg. Springer Berlin Heidelberg.

Garey, M. R., & Johnson, D. S. (1990). *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA.

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

Gulwani, S., & Venkatesan, R. (2009). Component based synthesis applied to bitvector circuits. Tech. rep. MSR-TR-2010-12.

Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, *SSC-4(2)*, 100–107.

Husien, I., & Schewe, S. (2016). Program generation using simulated annealing and model checking. In De Nicola, R., & Kühn, E. (Eds.), *Software Engineering and Formal Methods*, pp. 155–171. Springer International Publishing.

Ji, R., Sun, Y., Xiong, Y., & Hu, Z. (2020). Guiding dynamic programing via structural probability for accelerating programming by example. *Proceedings of the ACM on Programming Languages*, *4*(OOPSLA).

Kalyan, A., Mohta, A., Polozov, O., Batra, D., Jain, P., & Gulwani, S. (2018). Neural-guided deductive search for real-time program synthesis from examples. In *International Conference on Learning Representations*.

Kodratoff, Y., Franova, M., & Partridge, D. (1990). Logic programming and program synthesis. In *Systems Integration '90. Proceedings of the First International Conference on Systems Integration*, pp. 346–355.

Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA.

Lee, W., Heo, K., Alur, R., & Naik, M. (2018). Accelerating search-based program synthesis using learned probabilistic models. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 436–449. Association for Computing Machinery.

Manna, Z., & Waldinger, R. (1979). Synthesis: Dreams-programs. *IEEE Transactions on Software Engineering*, *SE-5*(4), 294–328.

Mariño, J. R. H., Moraes, R. O., Oliveira, T. C., Toledo, C., & Lelis, L. H. S. (2021). Programmatic strategies for real-time strategy games. *Proceedings of the AAAI Conference on Artificial Intelligence*, *35*(1), 381–389.

Musser, D. R. (1989). *Automated Theorem Proving for Analysis and Synthesis of Computations*, pp. 440–464. Springer New York, New York, NY.

Odena, A., Shi, K., Bieber, D., Singh, R., Sutton, C., & Dai, H. (2021). BUSTLE: bottom-up program synthesis through learning-guided exploration. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*.

Odena, A., & Sutton, C. (2020). Learning to represent programs with property signatures. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net.

Qian, X. (1990). Synthesizing database transactions. In *Proceedings of the 16th International Conference on Very Large Data Bases*, VLDB '90, p. 552–565, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Qian, X. (1993). The deductive synthesis of database transactions. *ACM Trans. Database Syst.*, *18*(4), 626–677.

Shi, K., Bieber, D., & Singh, R. (2020). Tf-coder: Program synthesis for tensor manipulations. *CoRR*, *abs/2003.09040*.

Shin, R., Kant, N., Gupta, K., Bender, C. M., Trabucco, B., Singh, R., & Song, D. X. (2019). Synthetic datasets for neural program synthesis. *ArXiv*, *abs/1912.12345*.

Singh, R., & Gulwani, S. (2012). Synthesizing number transformations from input-output examples. In *Proceedings of the 24th International Conference on Computer Aided Verification*, CAV'12, p. 634–651, Berlin, Heidelberg. Springer-Verlag.

Smith, D. C. (1976). Pygmalion: A creative programming environment. Tech. rep..

Solar-Lezama, A. (2009). The sketching approach to program synthesis. In *APLAS*.

Solar-Lezama, A., Rabbah, R., Bodík, R., & Ebcioğlu, K. (2005). Programming by sketching for bit-streaming programs. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, p. 281–294, New York, NY, USA. Association for Computing Machinery.

Solar-Lezama, A., Tancau, L., Bodik, R., Seshia, S., & Saraswat, V. (2006). Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, p. 404–415, New York, NY, USA. Association for Computing Machinery.

Summers, P. D. (1977). A methodology for lisp program construction from examples. *Journal of the ACM*, *24*(1), 161?175.

Udupa, A., Raghavan, A., Deshmukh, J. V., Mador-Haim, S., Martin, M. M., & Alur, R. (2013). Transit: Specifying protocols with concolic snippets. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 287–296. ACM.

Verma, A., Murali, V., Singh, R., Kohli, P., & Chaudhuri, S. (2018). Programmatically interpretable reinforcement learning. In *Proceedings of the International Conference on Machine Learning*, pp. 5052–5061.

Waldinger, R. J., & Lee, R. C. T. (1969). Prow: A step toward automatic program writing. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence*, IJCAI'69, p. 241–252, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Wang, X., Dillig, I., & Singh, R. (2017). Program synthesis using abstraction refinement. *Proc. ACM Program. Lang.*, *2*(POPL).

Warren, H. S. (2013). *Hacker's delight*. Addison-Wesley.

Zohar, A., & Wolf, L. (2018). Automatic program synthesis of long programs with a learned garbage collector. *CoRR*, *abs/1809.04682*.