Program Synthesis with Best-First Bottom-Up Search

by

Saqib Ameen

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

Abstract

Cost-guided bottom-up search (BUS) algorithms use a cost function to guide the search for solving program synthesis tasks. In this thesis, 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 the cost function. We introduce a novel best-first bottom-up search algorithm, which we call BEE SEARCH, that does not suffer from 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 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 utilizes the information provided by an existing cost model. Empirical results on string manipulation and bit-vector tasks show that BEE SEARCH can outperform the state-of-the-art cost-guided BUS approaches when employing more complex domain-specific languages (DSLs); BEE SEARCH and previous approaches perform equally well with simpler DSLs. Further, our new cost function with BEE SEARCH outperforms previous cost functions on string manipulation tasks.

Preface

This dissertation is original work of the author done in collaboration with Levi Lelis. This work is currently under review for publication. Due to the collaborative nature of this work, pronoun “we” is used in this script. However, I remain solely responsible for any technical or presentational errors present.

Saqib Ameen
December, 2022

To my parents.

Acknowledgements

I owe my deepest gratitude to my advisor Levi Lelis for his constant support, patience, and guidance throughout my research work. He did not only influence my research, but helped me become a clear thinker, better writer, and a well-rounded person in general.

I am thankful to my friends and colleagues who made my stay at the University of Alberta memorable and made harsh winters and COVID time bearable. In particular, I would like to thanks, in no order, Justin Stevens, Abdul Wahab, Lucas N. Ferreira, Rubens O. Moraes, Tales Carvalho, Thirupathi Reddy, Abdul Ali Bangash, Kamran Janjua, Rayhan Kabir, Shihab Anik, Nazmus Sakeef, Humayun Kabir, Habib Rahman, Jiamin He, Kenneth Tjhia, and Aidan Bush. I am also grateful to my friends in Pakistan — for always having my back and all the facetimes to make help me endure the distance from them.

For their endless support and love, I owe this to the greatest people in my life: my parents and my siblings.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 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., 2017; 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 thesis, we consider programming-by-example (PBE) 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.

Program synthesis has been successfully applied to various problems. One of its prominent applications is data wrangling tools such as FlashFill (Gulwani, 2011) in Excel, which allows users to perform string transformations by giving input-output examples. FlashExtract (Le & Gulwani, 2014) is another such tool that can extract fields from web pages and semi-structured log files. Programming is also a significant domain where synthesis is applied to develop tools for code suggestions (Zhang et al., 2016; Chen et al., 2021), code repair (Juniwal et al., 2014; Singh et al., 2013), and auto-completion (Raychev et al., 2014). Among other applications of program synthesis, some notable ones are numbers transformations (Singh & Gulwani, 2012), circuits transformations (Barthe et al., 2014), programmatic strategies for games (Butler et al., 2017; Mariño et al., 2021; Medeiros et al., 2022), and SQL queries synthesis (Wang et al., 2017a; Bastani et al., 2019; Takenouchi et al., 2021; Zhou et al., 2022).

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 programs space DSLs induce can be very large and a considerable amount of research has been devoted to developing more effective search algorithms for solving 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 programs of the DSL, and it iteratively generates larger programs by combining the smaller ones the algorithm has generated (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.

Nevertheless, 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 as 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).

Despite their superior performance to BUS, Cost-guided BUS algorithms used in TF-Coder, Probe, and Bustle, suffer from the same problem: they can lose some of the information given by the guiding cost function, hence they do not perform the search in best-first order with respect to the cost of the programs. That is, these algorithms might evaluate more expensive programs before evaluating cheaper ones. Heap Search is a cost-guided BUS algorithm which attempts to solve this problem by searching in a 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 large number of programs that are more expensive than the solution program. Moreover, as we show in this work, Heap Search sacrifices the detection of observational equivalence 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 this dissertation, we examine an alternate approach to best-first bottom-up search for synthesis using the idea of an abstract search space. We introduce this idea in a search algorithm that we call Bee Search; it does not suffers from the shortcomings of Heap Search and can perform best-first search with respect to the generation of the programs for a given cost function. It is able to fully utilize the information given by a guiding cost function, and can be used with a family of cost functions from the literature.

$$I \rightarrow \texttt{concat}(I, I) \mid \texttt{1} \mid \texttt{2} \mid \cdots \mid \texttt{1000}$$

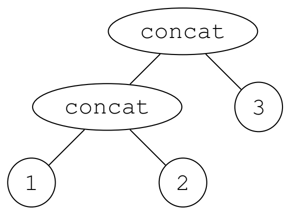

Figure 1.1: DSL and AST for `concat(concat(1, 2), 3)`, which produces the output `123`.

1.1 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 1.1 shows a DSL with $V = \{I\}$, $\Sigma = \{\texttt{concat}, \texttt{1}, \texttt{2}, \cdots, \texttt{1000}\}$, R represents the production rules (e.g., $I \rightarrow \texttt{1}$); 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 in the rule's righthand side. For example, the arity of rule $I \rightarrow \texttt{concat}(I, I)$ is 2. The arity of terminal rules is 0. The programs $\mathcal{G}$ accepts determines the programs space. For example, $\mathcal{G}$ accepts `concat(concat(1, 2), 3)`: I is replaced with $\texttt{concat}(I, I)$; then the leftmost I with $\texttt{concat}(I, I)$ 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`.

Search algorithms represent programs as abstract syntax trees (ASTs). Figure 1.1 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 `concat(concat(1, 2), 3)`. We say that a program is generated in search when the program's AST is created and stored in memory.

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 to (i) derive a program that $\mathcal{G}$ accepts and (ii) that correctly maps each of the input values to its corresponding output value. For example, $\texttt{concat}(\texttt{in}_1, \texttt{in}_2)$ correctly produces the output value for the following problem: $\mathcal{I} = \{[\texttt{1, 2}], [\texttt{10, 10}]\}$ and $\mathcal{O} = \{[\texttt{12}], [\texttt{1010}]\}$, where $\texttt{in}_1$ and $\texttt{in}_2$ are the two input values. In Appendix A, we present the DSLs used in our work.

1.2 Contributions

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 functions. We also show how to overcome the weaknesses of previous cost-guided BUS algorithms with Bee Search. 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 programs that are more expensive than the solution program. Note that other best-first algorithms such as A* (Hart, Nilsson, & Raphael, 1968) and Dijkstra's algorithm (Dijkstra, 1959) can generate states that are more costly than the goal state. To the best of our knowledge, Bee Search is the first search algorithm able to perform best-first search with respect to generation. 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 is able to perform observational equivalence checks, as regular BUS algorithms. Another contribution of our work is the introduction of a novel cost function based on the neural network model used in the Bustle system. In contrast with the 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. In order to evaluate our hypothesis, we compare the number of problems Bee Search solves while using the same cost functions Probe and Bustle use string and in a bit-vector domains. The results show that Bee Search is never worse than Probe and Bustle and that 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 on all settings evaluated. Finally, the results also show that Bee Search with our novel cost function outperforms all systems tested in the SyGuS string manipulation domain (Alur et al., 2013). Further, our work is first attempt where all these algorithms are written in same notation and compared. Heap Search, in its original paper, was only compared to the other best-first top-down approaches; it was not compared with any bottom-up approaches. Similarly, Probe and Bustle systems which use two different type of guiding cost-functions were not compared before.

Chapter 2

Background

In this chapter, we describe informed and uninformed versions of BUS for program synthesis. We also define the taxonomy of cost-functions used in the literature to guide BUS and discuss a few guided bottom-up search algorithms in detail: Probe, Bustle, Heap Search, and Brute.

2.1 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 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 size-based BUS was shown to be more effective than height-based BUS in the string manipulation domain we examine in this thesis (Barke et al., 2020), we only consider that in our work and implicitly refer to it as BUS.

Example 1. *Let us consider an example where we need to synthesize a program that produces the output `100010001000` with the DSL shown in Figure 1.1 (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. Next,* BUS *generates all programs of size 3:* $\{$*`concat(1, 1),`* $\cdots$*`, concat(1000, 1000)`*$\}$*. This process continues until the solution is generated while* BUS *produces the programs of size 5.*

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: $s \leftarrow 1$
2: while not timeout do
3:     for $p$ in NEXT-PROGRAM($\mathcal{G}, B, s$) do
4:         $o \leftarrow$ EXECUTE($p, \mathcal{I}$)
5:         if $o$ equals $\mathcal{O}$ then
6:             return $p$
7:         if $p$ is not equivalent to any program in $B$ then
8:             $B[s]$.add($p$)
9:     $s \leftarrow s + 1$
10: return $\perp$
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 $r \in R$ do
12:     if arity(r) $= 0$ then
13:         yield $r$
14:     else
15:         for $(p_1, \cdots, p_k)$in$\{B[s_0] \times \cdots \times B[s_k]\} \mid$
            $size(r(p_1, \cdots, p_k) = s \wedge type(p_i) = type(r.arg_i)\}$ do
16:             yield $r(p_1, \cdots, p_k)$
```

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 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. 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 programs' AST size, the size of target program s. NEXT-PROGRAM generates programs of size s using the production rules of the grammar $r \in R$ (lines 11-16). NEXT-PROGRAM returns the production rule r if it is terminal (lines 12-13). Otherwise, if arity is greater than 0, it takes the Cartesian product of all the previously seen programs in programs bank B using r, such that size of resultant program's AST is equal to the desired size s, i.e., $size(r(p_1, \cdots, p_k) = s$ and the type of all the subprograms p_i matches with the type of the production rule's arguments arg_i, i.e., $type(p_i) = type(r.arg_i)$. Here, $size$ gives us the size of the program's AST, k represents the arity of the production rule r, and the type check ensures that the type of subprograms, $p_1, \cdots, p_k$ match the type of the arguments required by r. For instance, `concat` can only take arguments of the string type; a valid combination of programs for the production rule r satisfy the type and cost conditions.

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 the solution for the problem. Otherwise, it checks for observational equivalence, i.e., whether or not the search has previously seen a program with 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 the it has not timed out and a solution program is not found.

2.2 Informed 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, first we explain the cost function then, we explain the working of PROBE, BUSTLE, HEAP SEARCH, and BRUTE; they are all considered in our experiments. Since TF-CODER requires a manually crafted set of weights for each operation of the language, we do not considered it in our experiments.

2.2.1 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 from Figure 1.1, a pre-generation function would determine the cost of program `concat(1, 2)` as a function of the cost of the production rule $I \rightarrow \texttt{concat}(I, I)$ and of the subprograms `1` and `2`. The cost functions used in TF-CODER and PROBE 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 that AST of a program must be in memory to compute its cost.

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

Figure 2.1: 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($\mathbb{P}_r$)).

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 of r being part of a solution. Consider the PCFG shown in Figure 2.1, 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:

$$w(p) = w(r) + \sum w(p_i) \tag{2.1}$$

Example 2. *Let us consider program $p =$ `concat(1, 2)`. The cost of 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. Further,* Probe *rounds off the cost of the programs to the nearest integer. For instance, 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 the increasing order of integer w-values; it first enumerates all programs of cost 1, then the ones with integer w-value 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 it 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 inputs set $\mathcal{I}$ to the outputs set $\mathcal{O}$.The budget Lim is defined as a constant d, which is defined manually, multiplied by the largest cost of a production rule in the current PCFG.

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

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 $d = 6$ in Probe paper. If the search is unable to find a solution after restarting and no partial solution is found, then the budget is increased to $\textsc{Lim}_i = \textsc{Lim}_{i-1} + l \times d$, where $\textsc{Lim}_{i-1}$ is the budget of the previous iteration.

Probe's PCFG starts with a uniform probability distribution to all the 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

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

Here, $\mathbb{P}_u(r)$ represents the probability of rules as given by a 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: 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. Hence, 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.

Bustle Cost Function ($w_{\textsc{Bustle}}$)

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

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

```
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 2.2: A set of input output pairs (`io_pairs`) and property signatures list (`ps`). The first property returns `True` for all pairs (the lower-case version of the output string is in the lower-case version of the input string for all input-output 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.

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.

$$w'(p) = w(p) + 5 - \delta(p) \quad \text{where } \delta(p) \in \{0, \cdots, 5\} \tag{2.2}$$

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 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 instance, consider p with probability 0.05, then $\delta(p) = 0$, and $w'(p) = w(p) + 5$. This will delay the use of subprogram p to generate further programs. Note that like Probe, Bustle uses a discretization scheme to ensure that the cost values w and w' are integers.

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.*

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.

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 2.2.3 and 2.2.4, respectively.

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: $c \leftarrow 1$
 2: while not timeout do
 3:     for $p$ in NEXT-PROGRAM($\mathcal{G}, B, c, w$) do
 4:         $o \leftarrow$ EXECUTE($p, \mathcal{I}$)
 5:         if $o$ equals $\mathcal{O}$ then
 6:             return $p$
 7:         if $p$ is not equivalent to any program in $B$ then
 8:             if post-generation cost function then
 9:                 $c \leftarrow w'(p)$
10:             $B[c]$.add($p$)
11:     $c \leftarrow c + 1$
12: return $\perp$
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 $r \in R$ do
14:     if arity $= 0$ and $w(r) = c$ then
15:         yield $r$
16:     else if arity $> 0$ and $w(r) < c$ then
17:         $P \leftarrow \{B[c_1] \times \cdots \times B[c_k] \,|w(r(p_1, \cdots, p_k)) = c \wedge type(p_i) = type(r.arg_i)\}$
18:         for $(p_1, \cdots, p_k)$ in $P$ do
19:             yield $r(p_1, \cdots, p_k)$
```

2.2.2 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 the 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 4. *Consider the DSL shown in Figure 2.3 along with the cost of each production rule $r \in R$, where the goal is to synthesize the program `concat(concat(1000, 1000), 1000)`. The costs are assigned in a way that they bias the search towards the solution. 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. Further, cost of each production rule is rounded-off to the nearest integer value,*

	Cost
$I \rightarrow$ `concat`(I, I)	14.28771
$\mid$ `1` $\mid$ `2` $\mid \cdots \mid$ `999`	9.860138
$\mid$ `1000`	9.165895

Figure 2.3: A context-free grammar (CFG) for string manipulation task with the cost of each production rule $r \in R$ as given by a cost function w, with costs slightly biased towards the solution program `concat(concat(1000, 1000), 1000)` as `1000` is cheaper than `1`, `2`, $\cdots$, and `999`.

for instance, 9.860138 *is rounded to* 10 *and* 9.165895 *is rounded to* 9*. Cost-guided* BUS *will start by enumerating all programs with cost* 9*: {`1000`}, since it is the cheapest set of programs which can be generated. No program can be generated with costs in* $[1, 8]$*. Next, it generates programs with cost* 10*: {`1`, `2`, $\cdots$, `999`}, followed by programs with cost* 32*: {`concat(1000, 1000)`} since the cost of `concat` is* 14 *and the cost of `1000` is* 9*. In the next iteration, it generates programs with cost* 33*: {`concat(1000, 1)`, $\cdots$, `concat(999, 1000)`}, followed by programs of cost* 34*: {`concat(1, 1)`, $\cdots$, `concat(999, 999)`}. Then in the fifth iteration, it generates programs of cost* 55 *and it finds the solution program `concat(concat(1000, 1000), 1000)`.*

Contrary to the size-based enumeration, cost-guided BUS is biased towards cheaper programs according to the cost function w, which often allows the search to solve the problem while possibly generating many fewer programs as 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 is incremented 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 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 target cost c given by the rule r with the parameters given by the Cartesian product of all programs in bank B. While computing the Cartesian product, NEXT-PROGRAM considers only the programs from B that match the type of the arguments of r (lines 16–19). For example, when generating programs p with the production rule $I \rightarrow$ `concat`(I, I), NEXT-PROGRAM only considers programs that return strings as subprograms of the programs 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 program p is added to B, the main loop is repeated until the search finds the solution program p or it exhausts the time it has allowed for synthesis.

Algorithm 2 generalizes PROBE and BUSTLE: it is equivalent to PROBE if it receives PROBE's cost function and it is equivalent to BUSTLE if it receives BUSTLE's cost function.

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 2.1, `1000` has a lower cost than `1 | 2 |···| 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 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 $[1, 8]$ in Example 4); 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 for 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 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 thesis, 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 Chapter 3 for details).

2.2.3 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 provably evaluates programs in a best-first ordering according to a pre-generation cost function. It achieves best-first enumeration by using a set of priority queues Heap, one for each non-terminal symbol T. We say that programs derived of T are of type T. Each 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 5. *Let us 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 popped out of 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 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 program 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 popped out of* 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

Algorithm 3 Heap Search

```
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 $\bot$
 1: for all non-terminal symbols $T \in V$ do
 2:     Create an empty min heap $\text{Heap}_T$
 3:     Create an empty hash table $\text{Succ}_T$
 4:     Create an empty set $\text{Seen}_T$
 5:     for all derivation rules of $T \to p$ do
 6:         Add $p$ to $\text{Heap}_T$ with priority $w(p)$
 7:         Add $p$ to $\text{Seen}_T$
 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 $\text{Heap}_T$ with priority $w(p)$
11:         Add $p$ to $\text{Seen}_T$
12: $p \leftarrow \emptyset$
13: while not timeout do
14:     $p \leftarrow \text{Query}(p, I)$
15:     $o \leftarrow \text{Execute}(p, \mathcal{I})$
16:     if $o$ equals $\mathcal{O}$ then
17:         return $p$
18: return $\bot$
Procedure: Query($p$, $T$)
Require: A program $p$, and type $T$ of the program.
Ensure: Next cheapest program $p'$
19: if $p$ is a key in $\text{Succ}_T$ then
20:     return $\text{Succ}_T[p]$
21: else
22:     $p' \leftarrow pop(\text{Heap}_T)$ # $p'$ is of form $r(p_1, \cdots, p_k)$
23:     $\text{Succ}_T[p] \leftarrow p'$
24:     for all $i \in [1, \cdots, k]$ do
25:         $y_i = \text{Query}(p_i, T_i)$
26:         $p'_i = r(p_1, \cdots, p_{i-1}, y_i, p_{i+1}, \cdots, p_k)$
27:         if $p'_i$ is not in $\text{Seen}_T$ then
28:             Add $p'_i$ to $\text{Heap}_T$ with priority $w(p'_i)$
29:             Add $p'_i$ to $\text{Seen}_T$
30: return $p'$
```

program of each type T, which are 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 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 intialized 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 in 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 19 and 20). If the successor p' is not in SUCC_T, then it is popped out of HEAP_T and p' is set as the successor of p in SUCC_T. In Example 5, the successor of program `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 25). 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 27–29). 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 program `1 + 1` because it is observational equivalent to `2`, HEAP SEARCH considers both programs in search.

Limitations of HEAP SEARCH

HEAP SEARCH sacrifices the ability of performing observational equivalence 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 `1 + 1`'s children, 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 24-29) that might never be evaluated for being more expensive that cost of a solution program. Further, HEAP SEARCH was

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: Add $n_0$ to $Q$ with priority 0 (highest priority possible)
 3: B = {}
 4: while not timeout do
 5:     $n \leftarrow Q$.pop() # node $n$ can represent one or multiple programs
 6:     for $p$ in $n$ do
 7:         $o \leftarrow$EXECUTE($p, \mathcal{I}$)
 8:         if $o$ equals $\mathcal{O}$ then
 9:             return $p$
10:         if $o$ not in $B$ then
11:             $B$.add($p, o$)
12:     for each child $n'$ of $n$ do # only considers non-equivalent programs
13:         Add $n'$ to $Q$ with priority $w(p')$, where $p'$ is the program $n'$ represents
14: return $\perp$
```

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 (Fijalkow et al., 2022).

2.2.4 BRUTE

BRUTE is a best-first search algorithm we adapted to inductive program synthesis from the inductive logic programming system with the same name (Cropper & Dumančic, 2020). Let us consider an example using the same DSL used in Example 5: $I \rightarrow 1 \,|\, 2 \,|\, I + I$.

In BRUTE, the root of the tree represents all programs given by symbols appearing in terminal production rules. In our example, the root represents two programs: `1` and `2`. The root of the tree is the only node possibly representing multiple programs; all other nodes in the tree represents a single 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 of the tree is 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` is 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 terminal rules (line 1); n_0 is added to a priority queue Q and it is popped out of it in the first iteration of the algorithm. In every iteration of the algorithm, Brute pops the cheapest node n from Q (line 5) and it checks if any of the programs n represents is a solution to the problem (lines 6–9). If the programs are not observational equivalent to other programs in the bank B, then they are added to B (lines 10 and 11). Finally, Brute adds all children n' of n into Q (lines 13), as described in our example. Children are only generated and added to B for the programs $p \in n$ that are not equivalent.

Limitations of Brute

Similar 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 children. 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.

Chapter 3

Best-First Bottom-Up Search (Bee Search)

In this chapter we introduce Bee Search; it is organized as follows. First, we discuss how we adapt w_{Probe} and w_{Bustle} to Bee Searchand introduce a novel cost function to be used with Bee Search, which is based on w_{Bustle}. Then, we explain the cost-tuple space used by Bee Search algorithm to attain best-first ordering with respect to the generation of programs, and lastly we explain the Bee Search algorithm.

3.1 Cost Functions for Bee Search

The difference between Probe's cost function (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 as w_{Probe}. If used in the context of Bee Search, then we are referring to the function that does not truncate the values; we refer to the original Probe function in all other contexts.

Since Bee Search handles real-valued cost functions, we adapt Bustle's w function by interpolating the δ values 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 x and y values: $(0.00, 0)$, $(0.15, 1), (0.25, 2), (0.35, 3), (0.50, 4), (1.00, 5)$ and using a cubic spline to obtain an interpolant. Then, given a probability value returned by the model, the interpolant returns the δ-value used to obtain w', as shown in Equation 2.2. 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.

$$w_{\text{U}}(p) = 1 + \sum_{p'_i \in p} w'_{\text{U}}(p'_i) \tag{3.1}$$

where,

$$w'_{\text{U}}(p'_i) = w_{\text{U}}(p'_i) - \log_2 \mathbb{P}(p') \tag{3.2}$$

Here, we abuse notation in the summation and use the program p as the set of subprograms p' composing p. $\mathbb{P}(p)$ is the probability of p being part of a solution according to BUSTLE's neural network. This function computes the cost w of a program as the sum of the w' costs of its subprograms added of 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 of the negative of the log of the probability of p being 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.

3.2 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 smallest to largest cost. Whenever clear from the context, we use the words 'state' and 'cost-tuple state' interchangeably.

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 program encountered in search. Considering the costs of the PCFG shown in Figure 2.1, program `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 the state represents. Although only `1000` costs 9.165895, 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.

The 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; so 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 represent a solution to the program synthesis task, then all children of n is generated and inserted in Q. The children of n are given by the set of states the 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 is generated. Let us consider the following example.

Example 6. *Table 3.1 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 2.1. The table shows updates to the* Bee Search*'s costs 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-tuple 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 length of cost-tuple corresponds to the arity of the production rule; the empty cost-tuple $(\,)$ represents all programs generated with a terminal rule and cost-tuple with length two, e.g., $(0, 0)$, represent the entries for* `concat`.

Bee Search *starts by initializing C with the cost of the the cheapest program generated with a non-terminal rule, and Q with cost-tuple states for all terminal rules. 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* $[33.9207, (0, 0)]$ *represents the program* `concat(1000, 1000)`.[1] *In 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 pops the state* $[9.9660, ()]$ *and generates 999 programs* `1`, $\cdots$, `999` *with cost* 9.9660*; both costs are added to C.*

Next, the search pops $n =$ $[33.9207, (0, 0)]$ *from Q and generates* `concat(1000, 1000)` *with a cost of* 33.9208. *Here,* 0 *in the cost-tuple refers to the* 0*-th index of the cost set C and the only program we have with that cost is* `1000`*;* $(0, 0)$ *indicates that both arguments of* `concat` *should be*

[1]In an actual implementation, Bee Search would 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 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.

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,` $\cdots$`, 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 3.1: Trace of the solution to `concat(1000, concat(1000, 1000))`, containing cost list C, priority queue Q, programs generated, and the count of programs generated at each iteration, using Bee Search; see text for more information about state $[9.9660, ()]$ in iteration 1.

of cost 9.8165*, hence the program* `concat(1000, 1000)` *whose cost equals to the sum of the costs of 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* $[(34.0702, [1, 0]), (34.0702, [0, 1])]$. *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.*

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 the search running time. As an example, Probe is unable to distinguish 58.0249 (used in sixth iteration of the example) and 58.1744 (next cheapest cost) as both values are truncated to 58. As a result, Probe evaluates approximately one billion programs to find the solution. By contrast, Bee Search's best-first 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, we do not have to generate the programs to identify the ones that are to be evaluated next in 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

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:     $Q$.push($[S, (1, \cdots, 1)]$)
 5: for each terminal symbol $S$ in $\mathcal{G}$ do
 6:     $Q$.push($[S]$)
 7: while not timeout do
 8:     $p, c \leftarrow$ Next-Program($B, Q$)
 9:     $o \leftarrow$ Execute($p, \mathcal{I}$)
10:     if $o$ equals $\mathcal{O}$ then
11:         return $p$
12:     $B[c]$.add($p$)
13: return $\perp$
```

the program space. We show empirically the advantages of performing best-first search with respect to the generation of programs.

3.3 Bee 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 either 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). The ordering of Q is defined by the w-values of the cost-tuple states.

In every iteration of the algorithm (iteration of the while loop in Algorithm 5), it generates the next program p (see Algorithm 6) and we execute p 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 programs' cost (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

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: $n \leftarrow Q$.pop()
 2: if pre-generation cost function then
 3:     $C \leftarrow C \cup w(n)$
 4: if $n$ represents a terminal symbol then
 5:     return $r(n)$, $w(n)$
 6: for $i$ in $\{1, \cdots, k\}$ do
 7:     $n' \leftarrow n$
 8:     $n'[i] \leftarrow n'[i] + 1$
 9:     if $n'$ is not a duplicate then
10:         $Q$.push($n'$)
11: for $(p_1, \cdots, p_k)$ in $\{B[C[n[1]]] \times \cdots \times B[C[n[k]]] \wedge T(p_i) = T(r(n).arg_i)\}$ do
12:     $p \leftarrow r(n)(p_1 \cdots, p_k)$
13:     if $p$ is equivalent to any program in $B$ then
14:         continue
15:     if post-generation cost function then
16:         Insert $w'(p)$ in $C$ while keeping $C$ sorted
17:         if $w'(p) < C[i]$, where $i = \max_j \{j \in n | n \in Q\}$ then
18:             Heapify $Q$
19:         yield $p$, $w'(p)$
20:     else
21:         yield $p$, $w(p)$
```

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 $r(n)$ 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 $r(n)$ 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 $r(n)$ by taking the Cartesian product $B[C[n[1]]] \times \cdots \times B[C[n[k]]]$ (line 11). BEE SEARCH performs

[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.

type checking while generating programs with the Cartesian product operation. For example, if the type of the i-th non-terminal symbol of rule r, denoted $T(r(n).arg_i)$, is a string, as in the concat operation, then the program used to replace the i-th non-terminal symbol of concat, denoted $T(p_i)$, must return a string. We denote the newly generated programs as $r(n)(p_1, \cdots, p_k)$. Since all programs $B[C[n[i]]]$ have the same cost, the w-value of all programs $r(n)(p_1, \cdots, p_k)$ matches the w-value of n. We discard observational equivalent programs (lines 13 and 14).

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 programs cost 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 16. 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 performed only occasionally. 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 3.4). Each program and its cost are returned as an iterator in lines 19 and 21. 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 evaluation in batch is not shown in the pseudocode.

3.4 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 are monotonically increasing 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 that $w(n) = K + \sum_{i=1}^{k} C[i]$, 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 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 smallest w-value. Since w is additive, no state n can have smaller w-value 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 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 step) 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, as n_1 has the smallest w-value in Q. Thus, the search expands 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). $\square$

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 entails in 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 smallest w-value and the w function is additive, no other program has smallest w-value, so $C[1]$ has its minimum value. The inductive hypothesis states that $C[i]$-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 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 that will generate a program p whose $w'(p)$-value will be assigned to $C[i]$, and before this assignment happens, we have that $C[i] > w'(p)$.

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 is monotonically increasing for 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. $\square$

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.

Theorem 3. *Given enough memory and time, if a solution program p exists in the search space defined by the grammar $\mathcal{G}$* BEE SEARCH *will find it.*

Proof. BEE SEARCH considers all possible cost-tuple states in the search—it does not leave any state to be 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. $\square$

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

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 by 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 growing 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 monotonically increasing, $w(n) > C[i_{\text{MAX}}]$ (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 grows by 1 and all children of n have valid indexes. $\square$

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

Theorem 4. *If* BEE SEARCH *returns a solution program p, then p is correct as it solves the program synthesis task: p satisfies 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 SEARCH checks for these constraints in line 10 of Algorithm 5 and only returns the solution program p (line 11) if the constraints are satisfied. $\square$

Chapter 4

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 original paper (Odena et al., 2021), and (iii) 27 bit-vector problems from the Hacker's Delight book (Warren, 2013). Note that all sets of problems include real programming problems, such as problems posted by developers on StackOverflow. A few problems from the benchmarks are added in the Appendix B to give an intuition of the tasks that we are trying to solve. 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 where 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 $d = \{1, 2, \cdots, 7\}$ on 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 Bustle's neural network. 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 the 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, 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 bit-vector domain, we used 130 literals for all problems, obtained by taking the union of literals from all problems in the set, and adding other 116 random literals. The goal of experimenting with the larger DSLs is to evaluate the algorithms on large search spaces. The larger DSLs also simulate scenarios in which one does not have access to the set of literals required to solve a problem, e.g., FlashFill (Gulwani, 2011), and more literals can increase the chances of defining spaces that contain a solution. All experiments were run on 2.4 GHz CPUs with 16 GB of RAM. The algorithms had 120 minutes for each task.

Figures 4.1 and 4.2 present the results for the string manipulation tasks from 205 SyGuS competition and 38 handcrafted benchmarks, respectively. Figure 4.3 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 number of programs evaluated and by the running time in seconds. 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).

4.1 Discussion

Bee Search solves the largest number of problems in all domains and settings tested. For the SyGuS benchmark (Figure 4.1), Bee Search with w_{U} solves 184 tasks for the smaller DSL and 132 for the larger DSL. The second best algorithm is Bustle, which solves 180 tasks for the smaller DSL and 117 tasks for the larger DSL. Although the difference in terms of number of tasks solved might 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 4.2), both Probe and Bee Search with w_{Probe} solve the largest number of tasks for the smaller DSL (29 tasks) and for the larger DSL (17 tasks). Bustle solves 28 for the smaller DSL and 12 for the larger DSL. A similar pattern is observed in the bit-vector domain (Figure 4.3), where Probe and Bee Search with w_{Probe} solve 21 and 12 tasks with the larger and smaller DSLs, respectively. The second best performing algorithm in bit-vector is Heap Search, with 15 and 8 problems solved for the smaller and larger DSLs, respectively.

For a given cost function, Bee Search never performs worse than other search algorithms in

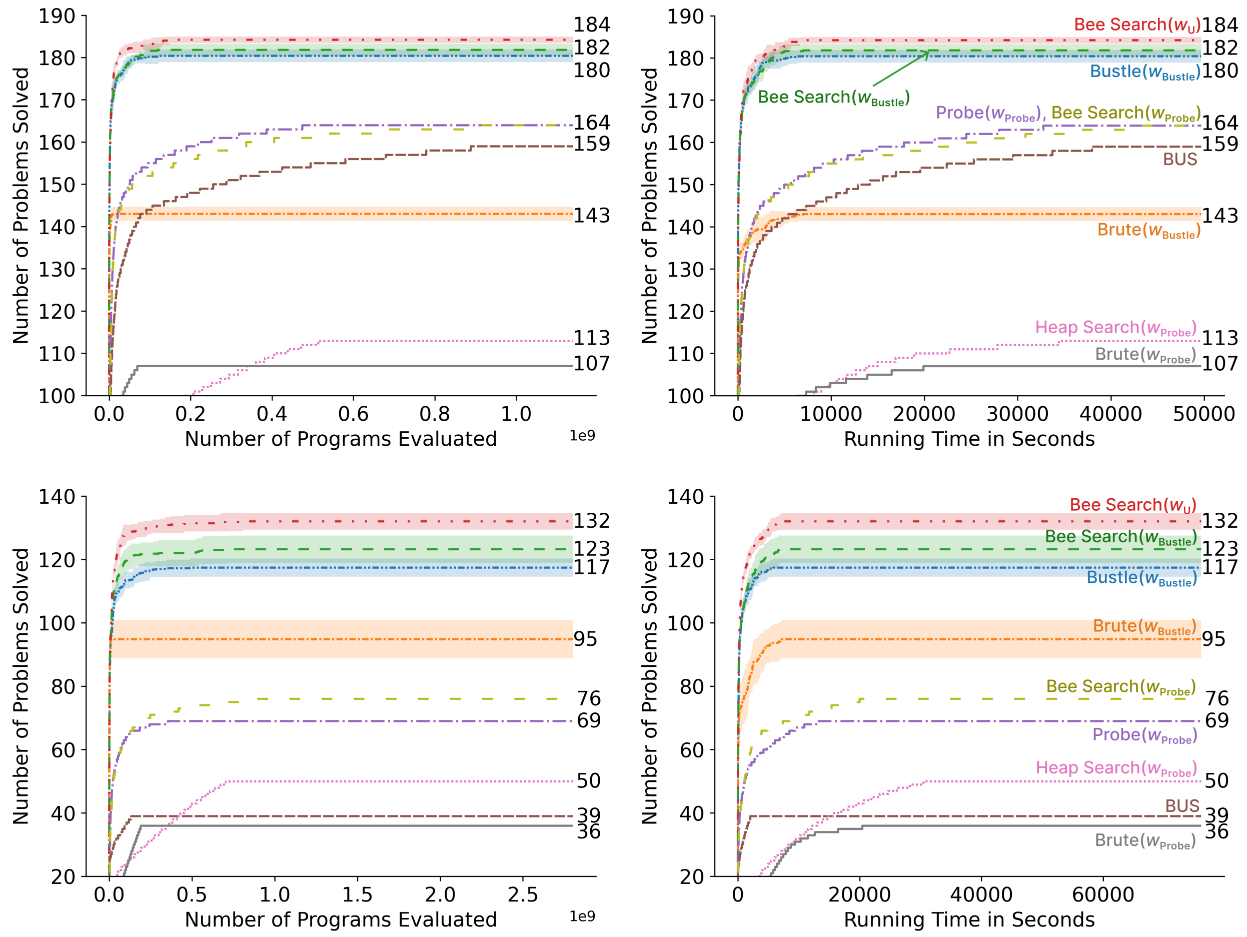

Figure 4.1: Number of problems solved per to number of evaluations and running time for 205 string domain problems of SyGuS. The two plots at the top show the results for the smaller DSL; the ones at the bottom for the larger DSL.

terms of number of tasks solved and it often performs better than the other algorithms. For the SyGuS benchmark (Figure 4.1), 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 w_{Bustle}. 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 it outperforms all algorithms in the larger DSL. We observe similar results in the 38 tasks (Figure 4.2) and in the bit-vector tasks (Figure 4.3).

Bee Search with w_{U} outperforms all systems tested in the SyGuS string domain with both DSLs. Bee Search with w_{U} outperforms both Bee Search and Bustle when they employ w_{Bustle} in the 38 handcrafted tasks. Function w_{U} relies more on the probability values the model returns than w_{Bustle} and Bee Search exploits such information with its best-first search. The much

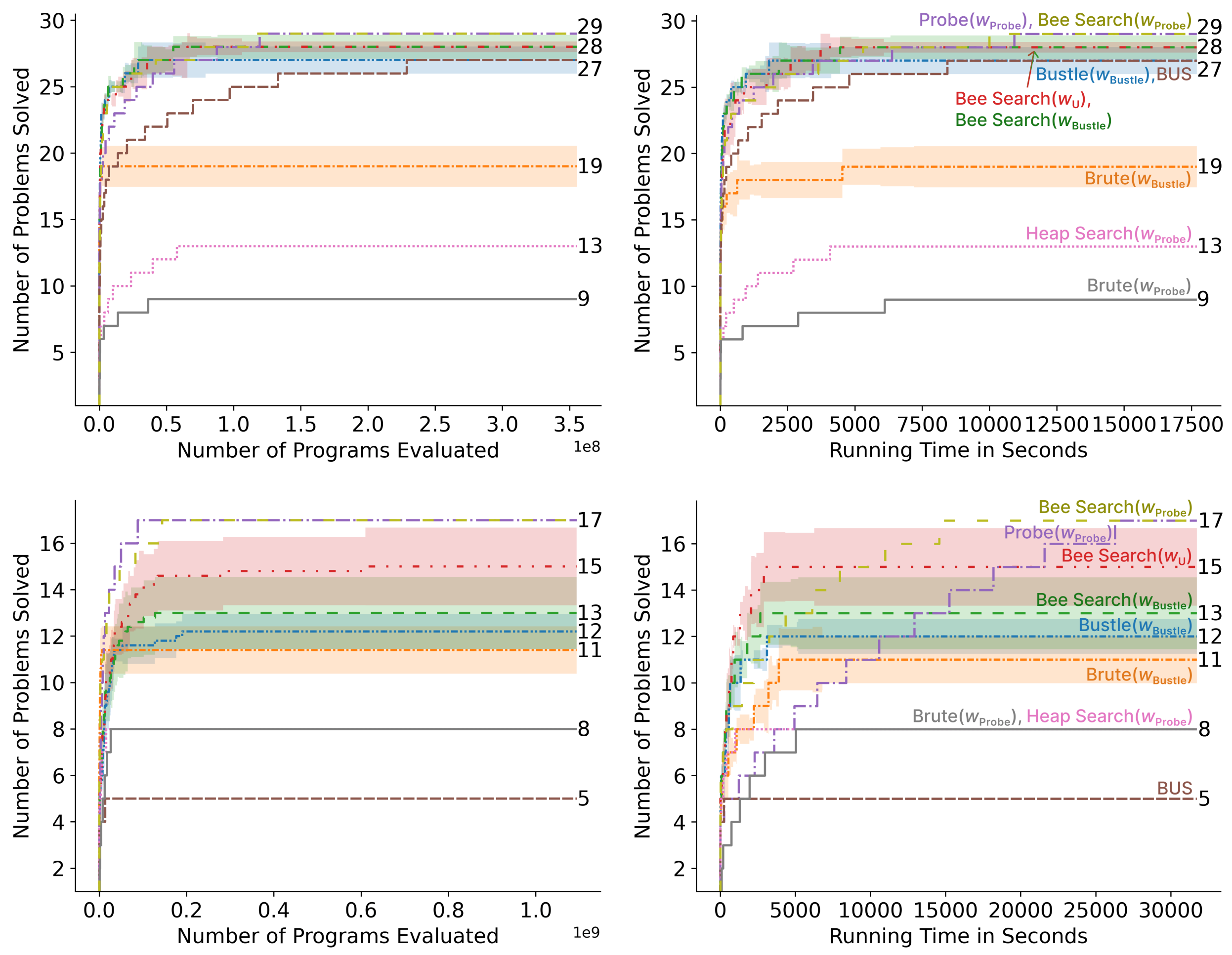

Figure 4.2: Number of problems solved per to number of evaluations and running time for the handcrafted 38 strings benchmarks by Odena et al.. The two plots on the top show the results for the smaller DSL; the ones on the bottom for the larger DSL.

simpler w_{PROBE} cost function performs better than the neural network-based functions (w_{BUSTLE} and w_{U}) in the 38 handcrafted tasks. We conjecture that the property signatures do not represent well the problems from this benchmark.

These results suggest that BEE SEARCH's best-first scheme is able to better use of the information the 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 the generation of programs. BEE SEARCH generates cost-tuple states that are never expanded, but the number of such states is much smaller

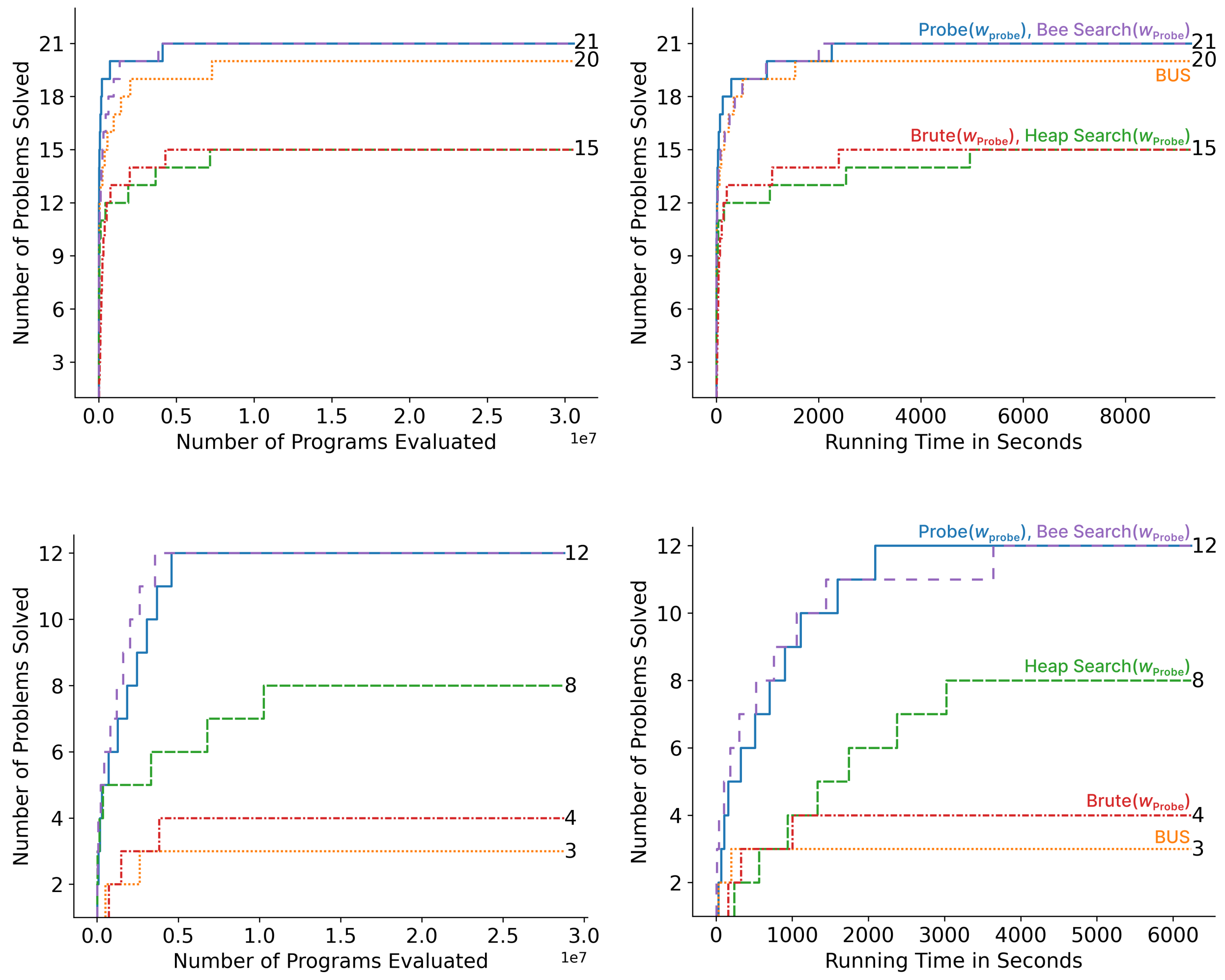

Figure 4.3: Number of problems solved per to number of evaluations and running time for bit-vector domain. The two plots on the top show the results for the smaller DSL; the ones on the bottom for the larger DSL.

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. Heap Search is unable to perform equivalence check, and this is the reason why it performs poorly in our experiments. In some cases, such as in the SyGuS benchmark, even the BUS substantially outperforms both Brute and Heap Search.

In the context of Inductive Logic Programming (ILP), Brute uses Answer Set Programming (ASP) constraints to reduce the branching factor of search. The Brute version we evaluated in this work 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 work, we evaluated only the sequential version of all algorithms.

Chapter 5

Related Work

Program synthesis has been studied for many years, starting with Waldinger and Lee (1969), Smith (1976), 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), and synthesis of fault-tolerant circuits (Eldib et al., 2016).

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 the programs in the space of programs. We focus on enumerative algorithms as Bee Search is an enumerative method.

5.1 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 smallest possible programs and use the rules of the symbolic language to generate bigger programs by combining the smaller ones. BUS is an attractive search strategy because the programs generated are complete and thus can be executed, which allows one to perform observational equivalence checks (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, 2017b) since most programs generated in the search are incomplete and cannot be executed.

5.2 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 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 explores first 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 past the (Chen, Liu, & Song, 2019; Zohar & Wolf, 2018; Bunel, Hausknecht, Devlin, Singh, & Kohli, 2018; Devlin et al., 2017; Wang et al., 2017b), 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 bottom-up search (BUS) algorithm for synthesizing programs. 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 TF-Coder in our experiments as we would have to manually choose the weights to problem domains used in our experiments. TF-Coder also suffers from the loss of information, similarly to Bustle and Probe as it considers only positive integer cost values during enumeration.

Chapter 6

Conclusions

In this dissertation, we showed that current state-of-the-art guided BUS algorithms, Bustle and Probe, 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 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 of eliminating observational equivalent programs. We presented an algorithm inspired on 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 to 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 costlier 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 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 Bustle's neural model. 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 it can substantially outperform them, especially in larger program spaces. Bee Search outperformed Heap Search and Brute by a large margin in both domains. The empirical results also showed that Bee Search with our cost function was the best performing system in the string domain.

We hope Bee Search will enable more research in cost functions for guiding the BUS algorithms. This is because Bee Search eliminates factors such as how much information is lost with truncation and how much time is wasted in sterile iterations; Bee Search allows for a reliable measurement of the quality of cost functions.

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 *TACAS*.

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.

Barthe, G., Dupressoir, F., Fouque, P.-A., Grégoire, B., & Zapalowicz, J.-C. (2014). Synthesis of fault attacks on cryptographic implementations. *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*.

Bastani, O., Zhang, X., & Solar-Lezama, A. (2019). Synthesizing queries via interactive sketching. *ArXiv*, *abs/1912.12659*.

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*.

Butler, E., Torlak, E., & Popovic, Z. (2017). Synthesizing interpretable strategies for solving puzzle games. *Proceedings of the 12th International Conference on the Foundations of Digital Games*.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Ponde, H., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D. W., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W. H., Nichol, A., Babuschkin, I., Balaji, S. A., Jain, S., Carr, A., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M. M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., & Zaremba, W. (2021). Evaluating large language models trained on code. *ArXiv*, *abs/2107.03374*.

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. (2017). Robustfill: Neural program learning under noisy I/O. In *Proceedings of the 34th International Conference on Machine Learning*, Vol. 70 of *Proceedings of Machine Learning Research*, pp. 990–998. PMLR.

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. *AAAI*, *abs/2110.12485*.

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).

Juniwal, G., Donzé, A., Jensen, J. C., & Seshia, S. A. (2014). Cpsgrader: Synthesizing temporal logic testers for auto-grading an embedded systems laboratory. In *2014 International Conference on Embedded Software (EMSOFT)*, pp. 1–10.

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.

Le, V., & Gulwani, S. (2014). Flashextract: A framework for data extraction by examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, p. 542–553, New York, NY, USA. Association for Computing Machinery.

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*, p. 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.

Medeiros, L. C., Aleixo, D. S., & Lelis, L. H. S. (2022). What can we learn even from the weakest? learning sketches for programmatic strategies. In *AAAI*.

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.

Raychev, V., Vechev, M., & Yahav, E. (2014). Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, p. 419–428, New York, NY, USA. Association for Computing Machinery.

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.

Singh, R., Gulwani, S., & Solar-Lezama, A. (2013). Automated feedback generation for introductory programming assignments. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pp. 15–26.

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

Solar-Lezama, A. (2009). The sketching approach to program synthesis. In *Asian Symposium on Programming Languages and Systems.*

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.

Takenouchi, K., Ishio, T., Okada, J., & Sakata, Y. (2021). Patsql: Efficient synthesis of sql queries from example tables with quick inference of projected columns. *Proc. VLDB Endow.*, *14*(11), 1937–1949.

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.

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, C., Cheung, A., & Bodík, R. (2017a). Synthesizing highly expressive sql queries from input-output examples. *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation.*

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

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

Zhang, H., Jain, A., Khandelwal, G., Kaushik, C., Ge, S., & Hu, W. (2016). Bing developer assistant: Improving developer productivity by recommending sample code. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, p. 956–961, New York, NY, USA. Association for Computing Machinery.

Zhou, X., Bodik, R., Cheung, A., & Wang, C. (2022). Synthesizing analytical sql queries from computation demonstration. In *Proceedings of the 43rd ACM SIGPLAN International Conference* *on Programming Language Design and Implementation*, PLDI 2022, p. 168–182, New York, NY, USA. Association for Computing Machinery.

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

Appendix

A Domain Specific Languages (DSLs)

A.1 String Domain DSL

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

Figure 6.1: Baseline DSL for string domain used in this thesis

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 6.2: Baseline DSL for bitvector domain used in this thesis

B Benchmark Problems

Count Consecutive Monthly Orders

Input	Output
7 0 0 5 4 4	3
0 0 2 3 3 0	3
5 6 4 6 0 7	4
0 4 5 0 0 2	2
3 0 3 0 1 2	2
5 3 2 5 6 1	6

Table 6.1: Input-output examples for the problem of finding the count consecutive monthly orders

Format Data

Input	Output
R/V<208,0,32>	R/V 208 0 32
R/S<184,28,16>	R/S 184 28 16
R/B<255,88,80>	R/B 255 88 80

Table 6.2: Input-output examples for the problem of formatting data

Address Formatting

Input	Output
Phialdelphia, PA, USA	Phialdelphia, PA, USA
Los Angeles, CA	Los Angeles, CA, USA
Ithaca, New York, USA	Ithaca, NY, USA
College Park, MD	College Park, MD, USA
Ann Arbor, MI, USA	Ann Arbor, MI, USA
New York, NY, USA	New York, NY, USA
New York, New York, USA	New York, NY, USA

Table 6.3: Input-output examples for the problem of formatting address

Extract Version Number

Input	Output
AIX 5.1	5.1
VMware ESX Server 3.5.0 build-110268	3.5
Linux Linux 2.6 Linux	2.6
Red Hat Enterprise AS 4 <2.6-78.0.13.ELlargesmp>	2.6
Microsoft <R> Windows <R> 2000 Advanced Server 1.0	1.0
Microsoft Windows XP Win2008R2 6.1.7601	6.1

Table 6.4: Input-output examples for the problem of extracting version number from string

Format Phone Numbers

Input	Output
938-242-504	(938) 242-504
308-916-545	(308) 916-545
623-599-749	(623) 599-749
981-424-843	(981) 424-843
118-980-214	(118) 980-214

Table 6.5: Input-output examples for the problem of formatting phone numbers

Extract Year

Input	Output
April 1 1799	1799
April 11 1867	1867
February 12 1806	1806
February 21 1798	1798
February 28 1844 as Delaware Township	1844
February 5 1798	1798
February 7 1892 Verona Township	1892
February 9 1797	1797
January 19 1748	1748
July 10 1721 as Upper Penns Neck Township	1721
March 15 1860	1860
March 17 1870 <as Raritan Township>	1870
March 17 1874	1874
March 23 1864	1864
March 5 1867	1867
April 28th 1828	1828

Table 6.6: Input-output examples for the problem of extracting year from string