Two Irons in the Fire: Synthesizing Libraries of Programs by Optimizing an Auxiliary Function while Solving Problems

by

Habibur Rahman

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

Master of Science

Department of Computing Science

University of Alberta

Abstract

Program synthesis faces a significant challenge in exploring a vast program space to find a program that satisfies the user's intent. Prior studies have proposed using different methods to guide the synthesis process to address this challenge. We propose a method that offers search guidance which can be orthogonal to guidance provided by the previous functions. It only requires one to define an auxiliary objective function aligned with the synthesis task. Given a synthesis task, a domain-specific language (DSL), a programming language, and a simple domain-dependent auxiliary function, we search for a solution to the task in the language-defined program space using a baseline search algorithm. If the search fails to find a solution, we augment the language by incorporating a library of programs encountered during the search process that optimize the auxiliary function. Then we repeat the search with this library-augmented language. We continue augmenting the language and searching until it finds a solution to the given synthesis task or the system times out. We evaluate our approach in two domains: string manipulation and reverse drawing. In reverse drawing, our language augmentation method substantially increases the number of problems solved using the base algorithm, bottom-up search, outperforming DreamCoder. In string manipulation, our proposed method increases the number of problems solved for all base algorithms evaluated: Bottom-up Search (BUS), Bee Search, CrossBeam, and Bustle.

Preface

I am delighted to present this dissertation entitled *“Two Irons in the Fire: Synthesizing Libraries of Programs by Optimizing an Auxiliary Function while Solving Problems”* which represents my original and unpublished research conducted under the supervision of Dr. Levi Lelis and the guidance of Dr. Elham Parhizkar. A shorter version of this work is in the process of submission to the Transactions of Machine Learning Research (TMLR).

In recognition of this project’s collaborative nature, the first person plural is used throughout this document. Nevertheless, I take full responsibility for any technical or presentational errors that may occur.

Habibur Rahman

September 2023

To my parents, my wife, and my sons.

Acknowledgements

With earnest gratitude, I extend my sincerest respect to my supervisor, Dr. Levi Lelis, for his constant guidance, support, and encouragement. The wisdom and understanding of Dr. Elham Parhizkar have likewise been a beacon throughout this journey. Together, their guidance has been my compass, guiding me with unparalleled grace.

I owe thanks to my colleagues, Thirupathi Reddy and Kenneth Tjhia, for their collaboration and support. My friends Rubens O. Mores, Tales Carvalho, Saqib Ameen, Lucas N. Ferreira, Zaheen Farraz Ahmed, Mahdi Alikhasi, Zahra Bashir, Quazi Asif Sadmine, Spyros Orfanos, Ronaldo Vieira, and Mahdieh Mallahnezhad have enriched this work with their valuable suggestions and encouragement. Each of you has made an indelible mark on this dissertation.

A wave of appreciation goes to my all guardians and friends in Edmonton and back home in Bangladesh, including Jaydeb Sarker, Ziaur Rahman Hablu, Osman Gani Badsha, Syed Tauhid Zuhori, Md. Masum Billah, Julia Rahman, Biprodip Pal, S M Hasibur Rahman, Tanvir Ahmed Tomal, Habibur Rahaman, and Nafize Sadik. Your constant support and encouragement have been my anchor.

To my beloved wife, Asmaul Husna, your unwavering faith and love have been my strength. To my sons, Afnan Habib and Awwab Habib, your pure love has been my daily inspiration.

I must express my profound love to my parents, Mozibar Rahman and Hamida Banu, and all my siblings, in-laws, nephews, nieces, cousins, uncles, aunts, and family members. Your love and support have been the foundation upon which this dream was built.

Lastly, my gratitude to the University of Alberta, the Department of Computing Science, the Alberta Machine Intelligence Institute (Amii), and the Digital Research Alliance of Canada for providing the opportunity and resources to complete my research. You have been instrumental in shaping my academic pursuit.

To all of you who have been part of this incredible journey, I say thank you. Your impact resonates in every word, every page, and every discovery made.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Artificial intelligence research has long sought to develop techniques for program synthesis, the process of automatically generating programs from high-level specifications (Manna and Waldinger, 1971). In program synthesis, the goal is to produce a program that exhibits a specific behaviour as specified by the user. To achieve this, current state-of-the-art methods often employ search algorithms that explore a space of possible programs. A wide range of search methods have been studied in the literature, including bottom-up search (Udupa et al., 2013), stochastic search (Schkufza et al., 2013), genetic programming (Koza, 1994), beam search with a sequence-to-sequence neural network (Devlin et al., 2017), top-down search with prioritized grammar rules (Lee et al., 2018), and learned search based on partial executions (Ellis et al., 2019; Zohar and Wolf, 2018).

One significant challenge in program synthesis is to scale the search process to synthesize long and complex programs. As the size of the program being synthesized increases, the search space grows exponentially, making synthesis increasingly difficult and impractical. This is despite the use of guiding functions employed by current state-of-the-art methods (Odena et al., 2021; Barke et al., 2020; Shi et al., 2022b), which are still limited in their ability to guide the search process.

In this dissertation, we propose a new approach to program synthesis that simplifies the process by only requiring the definition of an auxiliary objective function. Given a synthesis task, a language defining the program space, and a domain-dependent auxiliary objective function, we use a search algorithm to explore the space of programs defined by the language to find a solution to the task. If the search is unsuccessful in finding a solution, we augment the language with a library of programs encountered during the search that optimize the auxiliary function. Then, this library-augmented language is used to repeat the search process. This iteration of search and language augmentation is repeated until a solution is found or the system reaches a pre-determined timeout limit.

By augmenting the language with programs that optimize the auxiliary function, our method can significantly reduce the complexity of the synthesis process. To illustrate, consider the reverse drawing domain where the system receives a drawing as input and produces a program that generates the drawing as output (Ellis et al., 2020). The auxiliary function in this domain measures how many pixels the program has left to draw. By augmenting the language with programs that optimize such an auxiliary function, we allow for the synthesis of simple programs that combine some of the programs from the library to generate the entire image provided as input to the task.

To evaluate the effectiveness of our proposed language augmentation approach, we conducted extensive experiments in two different domains: String Manipulation (Odena et al., 2021; Alur et al., 2013; Shi et al., 2022b) and Reverse Drawing (Ellis et al., 2020). In string manipulation, we used four baseline search algorithms: Bottom-up Search (BUS) (Udupa et al., 2013), Bustle (Odena et al., 2021), CrossBeam (Shi et al., 2022b), and Bee Search (Ameen and Lelis, 2023). We used bottom-up search (Udupa et al., 2013) as the baseline algorithm in the reverse drawing domain.

In those two domains, we use simple and intuitive auxiliary functions to build libraries of programs. The results of these evaluations demonstrate that our method of augmenting the program libraries can significantly improve the performance of all the baseline search algorithms. In string manipulation tasks, our library augmentation method can solve more problems than the evaluated-based algorithms. In the reverse drawing domain, our library augmentation method can substantially increase the number of problems the evaluated base algorithm, bottom-up search, can solve, outperformed DreamCoder (Ellis et al., 2020).

Chapter 2

Problem Formulation

In program synthesis, we can explore different programs through something called a domain-specific language (DSL), which is defined by a grammar denoted by $\mathcal{G}$. This grammar consists of three parts: non-terminals (V), terminals (Σ), relations (R) and start symbol (S), which together define the set of programs the DSL accepts.

There are two types of rules: non-terminal rules, which have at least one non-terminal symbol on the right side, and terminal rules, which don't have any non-terminal symbol. The term "arity" refers to the number of non-terminal symbols on the right side of a non-terminal rule, while for a terminal symbol, this number is 0. In Figure 2.1, we have a simple DSL with $V = \{S\}$, $\Sigma = \{\texttt{<}, \texttt{>}, \texttt{concat}, \texttt{replace}\}$, and R describing the rules (for example, $S \to \texttt{<}$).

The programs that $\mathcal{G}$ accepts make up the program space. For instance, the program $\texttt{replace}(\texttt{concat}(\texttt{<}, \texttt{>}), \texttt{<}, \texttt{>})$ is accepted by $\mathcal{G}$. We start with S and replace it with $\texttt{replace}(S, S, S)$. Then, we replace the leftmost S with $\texttt{Concat}(S, S)$, the one in the middle with $\texttt{<}$, the rightmost one with $\texttt{>}$, and so on.

$$S \to \ \texttt{<} \ \mid \ \texttt{>} \ \mid \texttt{concat}(S, S) \\ \mid \texttt{replace}(S, S, S)$$

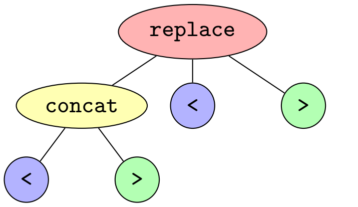


Figure 2.1: DSL and AST for `replace(concat(<, >),<,>)`.

Programs can be visualized using something called an abstract syntax tree (AST). The program's size is the number of nodes present in the AST of a program. Figure 2.1 (right) shows the AST for a specific program `replace(concat(<, >), <, >)`. Each circle or “node” in the AST represents a rule, and the nodes for non-terminal rules have lines to other nodes, called “children.” The nodes for terminal rules, which have no children, are the leaves of the tree. Each subtree within the AST represents a subprogram. So in our example, `concat(<, >)`, `<`, and `>` are all subprograms inside the main one. In this program, the size is 6 because there are 6 nodes in the AST.

Program synthesis refers to the task of generating a program using a specific domain-specific language (DSL), with a particular set of input values $\mathcal{I}$ and corresponding output values $\mathcal{O}$. The objective is to discover a program that is not only consistent with the defined grammar, but also capable of mapping the input values to the correct output values. To illustrate how the inputs are mapped to the output, let us examine the program $P_1 = \texttt{replace}(\texttt{i}_1, \texttt{i}_2, \texttt{i}_3)$, where $\mathcal{I} = \{[<, >, <], [>><, >, <]\ \}$ represents the input values and $\mathcal{O} = \{[<], [<<<]\ \}$ defines the desired outputs. Here, $\texttt{i}_1$, $\texttt{i}_2$, and $\texttt{i}_3$ are variables representing the input values to be processed by the program.

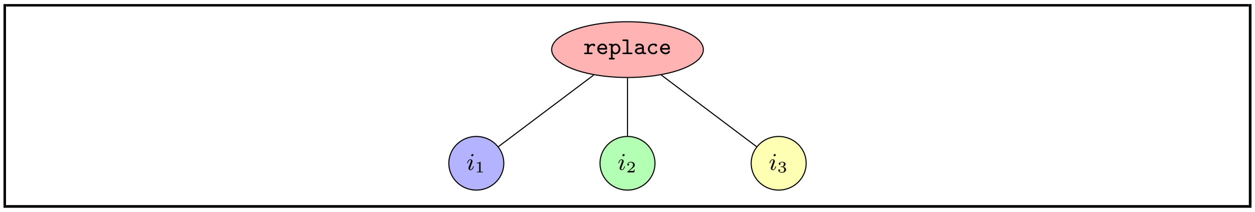


Figure 2.2: AST for $P_1 = \texttt{replace}(i_1, i_2, i_3)$.

Figure 2.2 represents the AST for P_1. The `replace` function is designed to replace occurrences of $\texttt{i}_2$ with $\texttt{i}_3$ within $\texttt{i}_1$. The operation can be understood in two parts:

- **Case 1:** Given $\texttt{i}_1 =$ “ $<, >, <$ ”, $\texttt{i}_2 =$ “ $><, >$ ”, and $\texttt{i}_3 =$ “ $<$ ”, the task is to replace segments matching $\texttt{i}_2$ in $\texttt{i}_1$ with $\texttt{i}_3$, leading to the output “[<]”. In mathematical notation, $\texttt{replace}(\texttt{i}_1, \texttt{i}_2, \texttt{i}_3) = \texttt{replace}(\text{“ } <, >, < \text{ ”}, \text{“ } ><, > \text{ ”}, \text{“ } < \text{ ”}) = \text{“ } < \text{ ”}$.

- **Case 2:** Given $\texttt{i}_1 =$ “ $><, >, <$ ”, $\texttt{i}_2 =$ “ $><, >$ ”, and $\texttt{i}_3 =$ “ $<$ ”, a similar transformation occurs, resulting in the output “[<<<]”. In mathematical notation, $\texttt{replace}(\texttt{i}_1, \texttt{i}_2, \texttt{i}_3) = \texttt{replace}(\text{“ } ><, >, < \text{ ”}, \text{“ } ><, > \text{ ”}, \text{“ } < \text{ ”}) = \text{“ } <<< \text{ ”}$.

These examples demonstrate how the `replace` function reads through $\texttt{i}_1$, identifies segments that match $\texttt{i}_2$, and substitutes them with $\texttt{i}_3$, thus generating the corresponding output for each given set of input values.

In this dissertation, we address the challenge of program synthesis in the string manipulation and reverse drawing domains. In the string manipulation domain, our goal is to generate programs that process input examples to produce outputs equivalent to those examples. In the reverse drawing domain, our aim is to generate a program capable of drawing an image that is the same as the given input. A program is deemed a solution if it can produce outputs identical to the examples in the string manipulation domain or replicate the image in the reverse drawing domain. We evaluate the programs in terms of the number of problems they can solve. Note that the outcome is binary, as the program either solves or does not solve the problem.

Chapter 3

Related Work

The challenge of synthesizing computer programs that meet a given specification has been widely discussed in the field of Computing Science (Manna and Waldinger, 1971; Summers, 1977), drawing significant attention from researchers in Artificial Intelligence (Balog et al., 2017; Devlin et al., 2017; Kalyan et al., 2018; Ellis et al., 2020) and Programming Language (Lee et al., 2018; Barke et al., 2020; Ji et al., 2020).

A variety of search methods have been investigated for solving synthesis tasks. One such method is to use constraint satisfaction algorithms, which involve converting the synthesis task into a constraint satisfaction problem that can be solved with readily available SMT solvers (Solar-Lezama, 2009). Additionally, synthesis tasks can be addressed using stochastic search algorithms, such as Simulated Annealing (Husien and Schewe, 2016) and genetic algorithms (Koza, 1992), which initiate with a potential solution and utilize mutation operators to generate alternate candidates that may be more proximate to a solution. Another strategy is to use enumerative algorithms that methodically evaluate all programs within the program space and guarantee a solution if enough resources are allowed in terms of time and memory. Enumerative techniques can be divided into bottom-up and top-down categories. Bottom-up search (BUS) algorithms begin with the smallest possible programs within a DSL and gradually build larger programs by combining the smaller ones (Udupa et al., 2013; Barke et al., 2020; Odena et al., 2021). One of the most notable benefits of BUS is that it generates complete programs during the search process, which can then be executed. This ability to execute programs enables the elimination of observational equivalent programs, or programs that produce the same output for a given input, thereby significantly reducing the number of programs generated during the search. For example, in the reverse drawing domain, we consider two programs to be observationally equivalent if they draw the same pixels on the canvas and the direction and states (up or down) of the pens are identical. In the string manipulation domain, if two programs produce exactly the same output, they are deemed observationally equivalent.

In case of such equivalency, we keep the shortest program. Top-down search algorithms begin with a high-level structure of the program and progressively detail the sub-structures. Top-down enumeration can only use weaker forms of equivalence (Wang et al., 2017; Lee et al., 2018) as many of the programs generated during the search process are unfinished and cannot be executed.

Due to the vastness of the program space induced by DSLs, a significant amount of research has been devoted to devising more efficient enumerative search algorithms for program synthesis tasks by employing pruning techniques to guide the search (Balog et al., 2017; Kalyan et al., 2018; Lee et al., 2018; Ji et al., 2020; Barke et al., 2020; Odena et al., 2021; Shi et al., 2022b). Empirical data indicates that guided top-down search methods do not surpass guided bottom-up search techniques (Barke et al., 2020).

Recent guided bottom-up search methods includes TF-Coder (Shi et al., 2022a), Probe (Barke et al., 2020), DreamCoder (Ellis et al., 2020), Bustle (Odena et al., 2021), Bee Search (Ameen and Lelis, 2023), CrossBeam (Shi et al., 2022b), LAPS (Wong et al., 2021) and Heap Search (Fijalkow et al., 2022). These systems use a combination of BUS and learned models to guide the search process using methods such as learned probabilistic models and neural models.

Of the methods mentioned, only DreamCoder and LAPS were evaluated on the reverse drawing domain, where one has to synthesize a program that produces the same image provided as input. DreamCoder starts with a base library of initial primitives and a dataset of training problems. Then, it returns a learned library augmented with programs and a learned neural model to guide a search procedure. The learning process is repeated with DreamCoder using the current library and neural model to find solutions to training problems and then updating the library and neural model based on the solved problems. It learns the library of programs for the whole domain depending on the training tasks. LAPS is also used in neural-guided search models for program synthesis along with natural language annotations to guide the library learning procedure.

Similarly to DreamCoder, Hernandez and Bulitko (2021) and Bulitko et al. (2022) introduced a library learning approach based on tasks in the problem domain, which incorporates the best-performing programs and their subprograms into the language. It starts by searching for the best programs on one portion of its dataset. Then it decomposes every subprogram of the best-performing programs and adds them to the language. The augmented language is then used to search for a program for the remaining problems in the dataset.

Our proposed approach differs from DreamCoder, LAPS, and the approach of Hernandez and Bulitko (2021) and Bulitko et al. (2022) in that we learn the library while solving a specific synthesis problem; we do not assume that there exists a set of problems from which we can learn. Moreover, we show that such a learned library allows a synthesizer to solve more problems than it would otherwise be able to solve. The disadvantage of our approach is that it requires an additional auxiliary function as input.

Similarly to DreamCoder and LAPS, CrossBeam also uses a neural model to guide the search process. It uses the neural model to combine the previously explored programs to generate new programs. Similarly to CrossBeam, Bustle also uses a trained neural model to prioritize the combination of previously explored programs.

Although efforts to use learned models to guide synthesis have been successful to a certain extent, some of these approaches are still limited by the need to explore a large portion of the search space due to the quality of guidance, making them impractical for synthesizing large programs in many scenarios.

Our approach involves encoding domain-knowledge through the use of auxiliary functions to guide the search for a solution program. Instead of learning libraries of programs for a whole problem domain, like DreamCoder does, we learn a library of programs by searching the solution program for a specific problem. We use the base synthesizer to search for a solution; if it fails, we add the best program to the library of programs that optimizes the auxiliary function. Later in the next iteration of the search, this program acts as a function. We hypothesized that this way of augmenting the language offers search guidance orthogonal to the previously added functions.

Chapter 4

Background

In this chapter, we first explain the Bottom-Up Search (BUS) and Top-Down Search algorithms before exploring unguided and guided bottom-up search methods for program synthesis.

Let us consider that we have the DSL illustrated in Figure 4.1 of the drawing domain as our running example, where we have `Forward` and `Repeat` operations. The `Forward` operation moves the pen in the canvas by a Length, L and ends after taking an angle, θ. The `Repeat` operation iterates the program S for T times and the `PenUp` operation executes the subprogram S inside without drawing anything to canvas.

```
S → Forward(L, θ) | Repeat(S, T) | S.S | PenUp(S)
L → 1 | 2
θ → 0°| 9°| 90°| ...  | 360°
T → 1 | 2 | ...  | 9
```

Figure 4.1: Sample DSL for Drawing Domain

BUS starts by generating the program of size 1. Subsequently, it generates programs of increasing sizes in each iteration until it finds the solution program.

Iterations	Set of Programs
1	`1, 2, 3.., 0°, 1°,..`
2	Φ
3	`Forward(1, 0°), Forward(2, 0°),..`
4	`PenUp(Forward(1,0°)), PenUp(Forward(2,0°)),..`
5	`Repeat(Forward(1,0°), 3), Repeat(Forward(2,90°), 4),..`
6	`Forward(1, 0°).Forward(1, 0°), Forward(1, 0°).Forward(2, 0°),..`
7	`PenUp(Forward(1,0°)).Forward(1,0°), PenUp(Forward(1,0°)).Forward(2,0°), ...`
...	...

Table 4.1: Illustration of Bottom-Up Search Synthesis

In Table 4.1, BUS starts from the terminal symbols and completed programs and gradually combines those programs based on the DSL rules to build up more complex programs. BUS can inadvertently generate a multitude of duplicate programs. One effective method to address these duplicate programs is by identifying and removing observational equivalent programs, which subsequently reduces the search space size. Observational equivalent programs are those that yield the same output for an identical set of inputs. For instance, `Forward(1,0°).Forward(1,0°)` and `Forward(2,0°)` are observational equivalent as both commands move the **pen** by 2 pixels in the same direction.

On the other hand, in top-down search synthesis, it starts with the start symbol. In subsequent iterations, it recursively replaces the start symbol with terminal or non-terminal symbols based on the rules.

Iterations	Set of Programs
1	S
2	S.S, Forward(L, θ), Repeat(S, T), PenUp(S)
3	S.S.S, S.Forward(L, θ), Forward(2, θ),.., Repeat(S, 4),.., PenUp(S.S),..
4	S.S.S.S,.., S.Forward(L, 0°), Forward(2, 0°), Repeat(Forward(L,T),4),..
...	...

Table 4.2: Illustration of Top-Down Search Synthesis

In Table 4.2, we can see that top-down search synthesis starts with the start symbol S and recursively replaces it based on the rules.

While both bottom-up and top-down searches are commonly used in program synthesis, we focus on bottom-up search synthesis because it generates complete programs at each level of the search tree. This:

- Enhances efficiency by enabling their reuse as subprograms in subsequent searches.
- Allows for observational equivalence checks. This contrasts with the top-down search, where such checks are challenging due to the presence of non-terminal symbols in the programs, which cannot be executed.

The following sections delve into more details about uninformed and guided bottom-up search synthesis.

4.1 Uninformed Bottom-Up Search Synthesis

In this section, we present the foundational synthesis approach that the proposed method is based on, which is bottom-up search (BUS) (Albarghouthi et al., 2013; Udupa et al., 2013).

BUS is a program synthesis method that tackles tasks by incrementally building programs of increasing size. The process begins by generating all programs defined by terminal symbols of the DSL, which have a size of 1 in terms of the number of nodes in the program's AST. Then, using the programs of size 1, BUS generates the programs of size 2 by applying the production rules of the DSL. This process continues using the programs of sizes 1 and 2 to generate programs of size 3, and so on. The search stops when a program is generated that can map the inputs to the task's outputs.

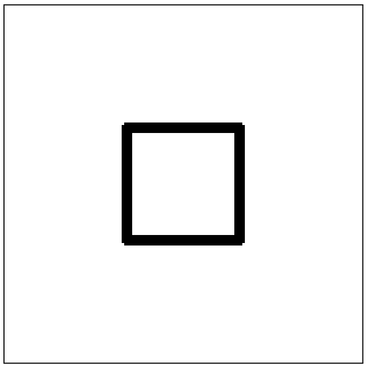


Figure 4.2: Sample Program Synthesis Task for Drawing a Square

Example 1. *Consider an example where a program is required to be synthesized to draw Figure 4.2 using the DSL depicted in Figure 4.1. The solution to this task, designed to draw the specified shape, is* `Repeat(Forward(2, 90°), 4)`*. Initially,* BUS *begins by generating and evaluating all programs of size* 1*, which are the constant lengths and angles like* $\{$`1, 2, ..., 0°, 1°, ...`$\}$*. The search then extends to generating the set of programs of size* 2*, which is empty. Continuing, it attempts to generate programs of size* 3*:* $\{$`Forward(1, 0°), Forward(2, 0°), ...`$\}$*. Since none of these programs successfully produces the desired image, BUS continues its search until it finds the solution with size* 5*.*

Algorithm 1 shows the operation of the uninformed BUS. This algorithm requires a context-free grammar, $\mathcal{G} = (V, \sum, R, S)$, a set of input-output examples $(\mathcal{I}, \mathcal{O})$, and an execution environment, labeled as $\mathcal{E}$ as its input. The grammar $\mathcal{G}$ is defined by a set of non-terminal symbols V, a set of terminal symbols $\sum$, a set of production rules R, and a start symbol S. The set of input-output examples $(\mathcal{I}, \mathcal{O})$ is used to evaluate the programs generated by the grammar. For every specific domain, an execution environment $\mathcal{E}$ is used to execute the programs generated by the grammar. For instance, in the reverse drawing domain, the execution environment $\mathcal{E}$ is a drawing canvas.

Algorithm 1 Uninformed Bottom-Up Search (BUS) Synthesis

```
Procedure: UNINFORMED-BUS($\mathcal{G}, (\mathcal{I}, \mathcal{O}), \mathcal{E}$)
Require: Grammar $\mathcal{G} = (V, \sum, R, S)$, a set of input-output examples $(\mathcal{I}, \mathcal{O})$ and execution environment, $\mathcal{E}$
Ensure: Solution program $p$ or $\perp$
 1: $s \leftarrow 1$                                          ▷ Initialize Program size
 2: $B \leftarrow \emptyset$                                  ▷ Initialize Program Bank
 3: $\mathcal{H} \leftarrow \emptyset$                        ▷ Initialize Output set of seen programs
 4: while not timeout do
 5:     for $p$ in GROW($\mathcal{G}, B, s$) do
 6:         $o \leftarrow$ EXECUTE($p, \mathcal{E}$)          ▷ Execute $p$ in $\mathcal{E}$
 7:         if $(p, o)$ satisfies $(\mathcal{I}, \mathcal{O})$ then
 8:             return $p$                                    ▷ Found solution
 9:         if $o \notin \mathcal{H}$ then
10:             $B[\text{s}] \leftarrow B \cup \{p\}$
11:             $\mathcal{H} \leftarrow \mathcal{H} \cup \{o\}$
12:     $s \leftarrow s + 1$                                  ▷ Increment size
13: return $\perp$                                            ▷ No solution found
Procedure: GROW($\mathcal{G}, B, s$)
Require: Grammar $\mathcal{G} = (V, \sum, R, S)$, program bank $B$, and size $s$
Ensure: program $p$ of size $s$
 1: for $r$ in $R$ do                                         ▷ Iterate over production rules
 2:     if arity($r$) $= 0$ and size($r$) $=$ s then          ▷ $r$ is a terminal symbol
 3:         yield $r$                                         ▷ Return $r$
 4:     else
 5:         if arity $> 0$ and size($r$) $< s$ then
 6:             for $(p_1, p_2, \ldots, p_k)$ in $B \times B \times \cdots \times B$ do   ▷ Operation over values of dictionary $B$
 7:                 $p \leftarrow r(p_1, p_2, \ldots p_k)$    ▷ Apply $r$ to $p_1, p_2, \ldots p_k$
 8:                 if size($p$) $= s$ and type($p_i$) $=$ type($r.arg_i$) for $i$ in $\{1, 2, \cdots, k\}$ then
 9:                     yield $p$                             ▷ Return $p$
```

Finally, BUS returns a program p that satisfies the input-output examples $(\mathcal{I}, \mathcal{O})$ or $\perp$ if such a program is not found within the time limit.

The algorithm begins by initializing the size s to 1 (line 1), the program bank B to $\emptyset$ (line 2), and the output set of the programs seen $\mathcal{H}$ to $\emptyset$ (line 3). After the initialization phase, it then enters a loop that continues until a solution is found or a timeout occurs. The loop begins by generating all programs of size s (line 5). Each program p is then executed in the execution environment $\mathcal{E}$ (line 6) using the EXECUTE procedure. The output of the program p is then checked against the output set $\mathcal{H}$ (line 7). If the output of the program p is not in the output set $\mathcal{H}$, then the program p is added to the program bank B (line 10) and the output of the program p is added to the output set $\mathcal{H}$ (line 11). This process of checking the output of the program p against the output set $\mathcal{H}$

ensures that the algorithm does not generate duplicate programs. The size s is then incremented (line 12) and the loop continues. If a solution is found, the algorithm returns the solution program p (line 8). If no solution is found within the time limit, the algorithm returns $\perp$ (line 13), as failure.

Grow is a procedure that generates all programs of size s using the production rules of grammar $\mathcal{G}$ and the current bank of programs. The procedure takes as input the grammar $\mathcal{G}$, the program bank B, and the size s. The procedure ensures a set of programs of size s. The procedure begins by iterating over the production rules R of the grammar $\mathcal{G}$ (line 1). For each production rule r, the procedure checks if the arity of the production rule r is 0 (line 6). If the arity of the production rule r is 0 and the size of r is s, then the production rule r is a terminal symbol and is returned (line 9). Otherwise, the procedure iterates over all possible combinations of programs of size s in the program bank B (line 6). The production rule r is applied to each combination of programs (line 7). If the type of all the subprograms p_i matches the type of production rule's arguments arg_i, i.e.,type(p_i) = type($r.arg_i$) for i in $\{1, 2, \cdots, k\}$ and the size of the program p is s, then the program p is returned (line 8).

4.2 Guided Bottom-Up Search Synthesis

In this section, we present the generic approach of guided bottom-up search synthesis that was introduced in Tf-coder (Shi et al., 2022a), Bustle (Odena et al., 2021), Crossbeam (Shi et al., 2022b), Probe (Barke et al., 2020), Brute (Cropper and Dumancic, 2020), Heap-Search (Fijalkow et al., 2022) and Bee Search (Ameen and Lelis, 2023).

All of the guided bottom-up search synthesis algorithms use a cost function to guide the search. The cost function is a function to calculate or approximate the cost of generating a particular program. The cost function is used to guide the search to generate programs with lower costs. Intuitively, the cost function assigns a higher cost to programs that are less likely to be part of a solution; on the other hand, it assigns a lower cost to the programs that are more likely to be part of a solution. This guided bottom-up search synthesis approach helps reduce the search space and find the solution faster.

Example 2. *Consider a task where we need to generate a program to draw the image shown in Figure 4.3. We can achieve this by using a Domain-Specific Language (DSL) shown in Figure 4.1. To calculate the cost of each program, we use a cost function, θ. For the given image, the solution program is:* `Repeat(Forward(2,90°),4).Forward(2,0°).Repeat(Forward(2,120°),3)`*. We use a cost-guided bottom-up search approach to generate the program. The search starts with generating terminal symbols, which are* {`1, 2,..., 0°, 1°,...`} *and have a cost of* 11*, which is the cheapest. As there are no possible programs that can be made with cost* $[12 - 22]$*, the next step is to generate programs costing* 23*, which are* {`Forward(1, 0°), Forward(2,90°),...`}*. For instance, the cost*

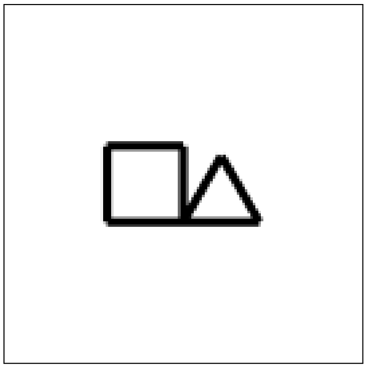


Figure 4.3: Sample Program Synthesis Task for Triangle Beside of Square

of `Forward(1, 0°)` is calculated as θ`(Forward(1, 0°)) =` θ`(1) +` θ`(0°) +` θ`(Forward) = 11 + 11 + 1 = 23`. In the fifth iteration, we get the solution program `Repeat(Forward(2,90°),4). Forward(2,0°).Repeat(Forward(2,120°),3)` with a cost of 71*, which generates the image.*

Algorithm 2 illustrates the generic cost-guided bottom-up search, which we have adopted from Ameen and Lelis (2023). This algorithm requires the same inputs as Algorithm 1 and a cost function θ as its input. In generic cost-guided bottom-up search, programs are enumerated in the order of increasing cost. The search enumerates all the programs of cost c before enumerating the programs of cost $c + 1$.

This algorithm begins by initializing a program bank B and a hash map $\mathcal{H}$ in lines 1 and 2 respectively. The program bank B is used to store the programs with the cost as a key. The set $\mathcal{H}$ is used to check for observational equivalent programs. It starts with a cost budget $c_b = 1$. Then, it goes over the next programs with cost c_b in line 5 which uses the Grow procedure. It executes the program in the execution environment $\mathcal{E}$ in line 6 and checks if the program satisfies the input-output examples $(\mathcal{I}, \mathcal{O})$ in line 7. If the program p satisfies the input-output examples $(\mathcal{I}, \mathcal{O})$, it returns p in line 8. Otherwise, it then checks if the subprogram is in the set $\mathcal{H}$ or not in line 9. If it is not, it adds the program p to the program bank B (line 10) and the output o in the set $\mathcal{H}$ (line 11). After iterating over all programs with cost c_b, it increments the cost budget c_b in line 12 and continues the process until it finds the solution program or the time limit is exceeded. Finally, it returns $\perp$ (line 13) if it does not find the solution program within the time limit.

Grow procedure of Algorithm 2 is used to generate new programs by applying production rules to the programs in the program bank B. This procedure requires the grammar $\mathcal{G}$, the program bank B, the cost budget c_b and the cost function θ as input. It returns a set of programs with cost c_b. In this procedure, it iterates over the production rules in line 1 and checks if the production rule is a terminal symbol or a non-terminal symbol. If the production rule r is a terminal symbol and the cost is equal to c_b, it returns r. Otherwise, if the production rule r is a non-terminal symbol

Algorithm 2 Guided Bottom-Up Search (BUS) Synthesis

```
Procedure: GUIDED-BUS($\mathcal{G}, (\mathcal{I}, \mathcal{O}), \mathcal{E}, \theta$)
Require: Grammar $\mathcal{G} = (V, \sum, R, S)$, a set of input-output examples $(\mathcal{I}, \mathcal{O})$, execution environment $\mathcal{E}$ and cost function $\theta$
Ensure: Solution program $p$ or $\perp$
 1: $B \leftarrow \emptyset$                                         ▷ Initialize Program Bank
 2: $\mathcal{H} \leftarrow \emptyset$                                         ▷ Initialize Output set of seen programs
 3: $c_b \leftarrow 1$                                         ▷ Initialize cost budget
 4: while not timeout do
 5:     for $p$ in GROW($\mathcal{G}, B, \mathcal{H}, c_b, \theta$) do          ▷ Iterate over the next set of programs
 6:         $o \leftarrow$ EXECUTE($p, \mathcal{E}$)                             ▷ Execute $p$ in $\mathcal{E}$
 7:         if $(p, o)$ satisfies $(\mathcal{I}, \mathcal{O})$ then
 8:             return $p$                                             ▷ Found solution
 9:         if $o \notin \mathcal{H}$ then
10:             $B[c_b] \leftarrow B[c_b] \cup \{p\}$                         ▷ Add $p$ to $B$ with cost $c_b$
11:             $\mathcal{H} \leftarrow \mathcal{H} \cup \{o\}$                         ▷ Add $o$ to $\mathcal{H}$ to check for duplicates
12:     $c_b \leftarrow c_b + 1$                                      ▷ Increment cost budget
13: return $\perp$                                                 ▷ No solution found
Procedure: GROW($\mathcal{G}, B, c_b, \theta$)
Require: Grammar $\mathcal{G} = (V, \sum, R, S)$, Program Bank $B$, Cost Function $\theta$, and Cost Budget $c_b$
Ensure: program of cost $c_b$
 1: for $r$ in $R$ do                                            ▷ Iterate over production rules
 2:     if arity($r$) $= 0$ and $\theta(r) = c_b$ then                   ▷ $r$ is a terminal symbol
 3:         yield $r$                                               ▷ Return $r$
 4:     else
 5:         if arity($r$) $> 0$ and $\theta(r) < c_b$ then
 6:             for $(p_1, p_2, \ldots, p_k)$ in $B \times B \times \cdots \times B$ do   ▷ Operation over values of dictionary $B$
 7:                 $p \leftarrow r(p_1, p_2, \ldots, p_k)$                    ▷ Apply $r$ to $p_1, p_2, \ldots p_k$
 8:                 if $\theta(p) = c_b$ and type($p_i$) = type($r.arg_i$) for $i$ in $\{1, 2, \cdots, k\}$ then
 9:                     yield $r(p_1, p_2, \ldots, p_k)$                  ▷ Return $r(p_1, p_2, \ldots, p_k)$
```

and the cost is less than c_b, it generates all possible subprograms by applying r to all the possible combinations of subprograms in the program bank B (line 7). Finally, if the program's cost is equal to c_b and the sub-programs used as parameters are consistent with the types of r, the procedure returns the program (line 9).

Algorithm 2 presents the pseudocode of guided BUS that describes algorithms PROBE Barke et al. (2020) and BUSTLE Odena et al. (2021). It is equivalent to PROBE if it receives the cost function of PROBE, and BUSTLE if it receives the cost function of BUSTLE. Next, we describe the cost function BUSTLE.

4.2.1 Bustle

Bustle (Odena et al., 2021) is an instantiation of Algorithm 2. Bustle uses a fully connected neural network model to compute the probability that a program is part of the solution. This neural model is a binary classifier that receives the task's input-output pairs $(\mathcal{I}, \mathcal{O})$ and the output of a program p for each of the input values in $\mathcal{I}$. Bustle assigns a higher cost to programs that are less likely to find the solution; on the other hand, assigns a lower cost to programs that are more likely to be a solution to the problem or task.

Bustle's cost function combines the cost associated with the production rule and the aggregate cost of all subprograms within a particular program. The computed cost is fine-tuned using a delta function, which discretizes the output of the neural network encoding the cost function. This neural network is specifically trained to predict the likelihood that a program will be a solution. The cost of any program is computed via the following formula:

$$\theta(p) = 1 + \sum_{p' \in \textsc{Sub}(p)} [\theta(p') + \delta(p')] \tag{4.1}$$

where, 1 is the cost associated with the production rule $\theta(r)$ to generate p (Bustle assumes all operators to cost 1), $\textsc{Sub}(p)$ represents the subprograms for the program p, and $\delta(p')$ is calculated using the bin scheme $B = \{0.0, 0.1, 0.2, 0.3, 0.4, 0.6, 1.0\}$ and the neural network's output probability. The value of $\delta(p')$ is calculated by the following equation:

$$\delta(p') = 5 - r(B, \hat{y}) \tag{4.2}$$

In this equation, $r(B, \hat{y})$ stands for the relative position of the output probability $\hat{y}$ within the bin scheme B. For example, if the output probability $\hat{y}$ is 0.3, then $r(B, \hat{y}) = 3$ as 0.3 belongs to the 3^{rd} bin, resulting in $\delta(p') = 5 - 3 = 2$.

Bustle trains the classifier to learn whether a program-generated output is intermediate between an input and output example pair. This is done by conditioning on two types of property signatures: one from the input to the output, and one from the intermediate output to the actual output. Figure 4.4 illustrates how the property signature is computed for the input-output examples in the string processing domain, which was introduced in Odena and Sutton (2020).

Each of the the property in X, demonstrated in Figure 4.4, results a representation belonging to the set of {`Mixed, AllFalse, AllTrue`} where `Mixed` represents a combination of `True` and `False` values, `AllFalse` represents all `False` values, and `AllTrue` represents all `True` values. In Figure 4.4, the first property returns *True* if the input is a substring of the output, the second property returns *True* for all pairs, the second property returns `Mixed` and the third property returns `False` for all

```
# input-output examples: (inp, outp)
io_pairs = [
            ("abc","AabcC"),
            ("123","123_"),
            ("aaB","aaB")
]

# property signatures
p1 = lambda inp, outp: inp in outp
p2 = lambda inp, outp: outp.endswith(inp)
p3 = lambda inp, outp: len(inp) > len(outp)

X = [p1, p2, p3]
```

Figure 4.4: Example of the Property Signatures for Input-Output Examples in the String Manipulation Domain

pairs (length of the input string is greater than the length of the output string). So, the resulting signature from the input to the output is {`AllTrue, Mixed, AllFalse`}. More precisely, if we have n input-output pairs and k properties, the property signature will be a vector of length k. Similarly, the signature for the intermediate output to the actual output is computed. It then concatenates the two signatures to form the final vector representation to train or test the neural network model. During the training process, BUSTLE randomly generated synthetic data.

4.2.2 BEE SEARCH

BUSTLE utilizes a strategy that rounds off the cost of programs by using a delta function. This is because this algorithm generates programs sequentially, in an order determined by increasing cost. However, a limitation of BUSTLE is that the actual order of programs does not necessarily reflect the true order of the associated costs. In summary, this can lead to the generation of programs in an arbitrary sequence, not necessarily reflective of an optimal order.

Instead of the rounding-off approach, BEE-SEARCH (Ameen and Lelis, 2023) employs a 'best-first' strategy that preserves the original order of the cost of the programs. By doing so, it maintains a more accurate representation of the cost-based hierarchy of the programs. Equation 4.3 shows the cost function used in BEE-SEARCH.

$$\theta(p) = 1 + \sum_{p' \in \textsc{Sub}(p)} [\theta(p') - \log_2 \mathbb{P}(p')] \tag{4.3}$$

Similar to BUSTLE's cost function, BEE SEARCH also considers the cost of the production rule r as

1 and considers the cost of the subprograms p' of the program p. However, it penalizes the cost of each subprogram by it's log-likelihood $\log_2 \mathbb{P}(p')$, where $\mathbb{P}(p')$ is the probability of the subprogram p' being part of the solution program.

4.2.3 CrossBeam

The CrossBeam method, as presented in (Shi et al., 2022b), introduces an innovative approach to guided bottom-up search synthesis, employing a trained model to create a probability distribution over the programs generated in the search process. Similarly to other techniques such as BUS, Bustle, and Bee Search, CrossBeam retains all generated programs in memory, enabling observational equivalence checks. Unlike the process of BUS, which generates all combinations of existing programs with a given production rule r, CrossBeam selectively samples subprograms p_i from the existing set. This sampling is used to determine the subsequent programs $r(p_1, \cdots, p_k)$ to be generated and evaluated.

The training procedure in CrossBeam involves a model that takes into account all programs generated during the search up to that level, the input-output pairs, and a specific production rule r. The model then constructs a probability distribution on all the programs encountered to define the next program $p = r(p_1, \cdots, p_k)$ evaluated in search. The subprograms p_i in p are sampled from the bank of programs by using methods such as Beam Search or UniqueRandomizer, as cited in (Shi et al., 2020). If a program p is found to be observationally equivalent to a previously encountered program, it is discarded. The system continues to iterate through all production rules r, sampling programs $r(p_1, \cdots, p_k)$, until a solution is found or the time limit is exceeded.

4.3 Limitation of Bottom-Up Search Synthesis

From the above sections on Uninformed and Guided BUS, we can see that the former is a complete search approach that generates all possible programs of a given size. However, it is not efficient to generate all the programs in the space of programs of a given size, and this can severely limit its applicability in practice as often one is unable to find a solution program due to the exponential growth of the search space.

On the other hand, guided BUS algorithms are more efficient and rely on the cost function to generate only the programs that are likely to be the solution. However, the guided BUS is also not guaranteed to find a solution. In cost-guided BUS synthesis, the cost function is used to guide the search toward the solution, but this cost function is also imperfect and the search space can still grow quite quickly with the size of the solution programs.

Chapter 5

Auxiliary-Guided Synthesis (AGS)

We introduce Auxiliary-Guided Synthesis (AGS) with the goal of simplifying the process of program synthesis through the use of an auxiliary objective function. Let us consider an example in the reverse drawing domain, where a system requires an image as input and produces a program that generates the drawing by controlling a pen (Ellis et al., 2020).

Example 3. *Our auxiliary function for the reverse drawing domain measures the number of pixels that overlap between the input image and the image drawn by a given program. Let the image shown Figure 5.2(c) as the input. Given this image, AGS searches for a solution in the language-defined program space using a base search algorithm (e.g., BUS). During this search, BUS finds the program P_1 that generates the image in Figure 5.2(a). As the search fails to yield the solution, AGS augments the language used to define the search space by adding P_1 to it as a symbol of size one. P_1 places the pen in the correct position by `PenUp(Forward(12, 0°))`, and then draws a line `Forward(12, 0°)`, and places the pen in the correct position to draw the first square `PenUp(Forward(12, 0°))`. In the next iteration of the search, the base search algorithm generates the program P_2 that includes P_1 as a subprogram; P_2 generates the image displayed in Figure 5.2(b). As this is still not the solution, AGS adds the program P_2 to the language. Finally, the base search finds the program P_3, which combines both P_1 and P_2 and solves the problem.*

As illustrated in Example 3, AGS uses an auxiliary objective function to build a library of programs that can be used to synthesize a program that solves the problem. The design of an auxiliary function is domain specific and should be chosen to capture relevant information about subprograms of the synthesis problem at hand. For example, in the reverse drawing domain, the auxiliary function evaluates a program's ability to partially draw a solution.

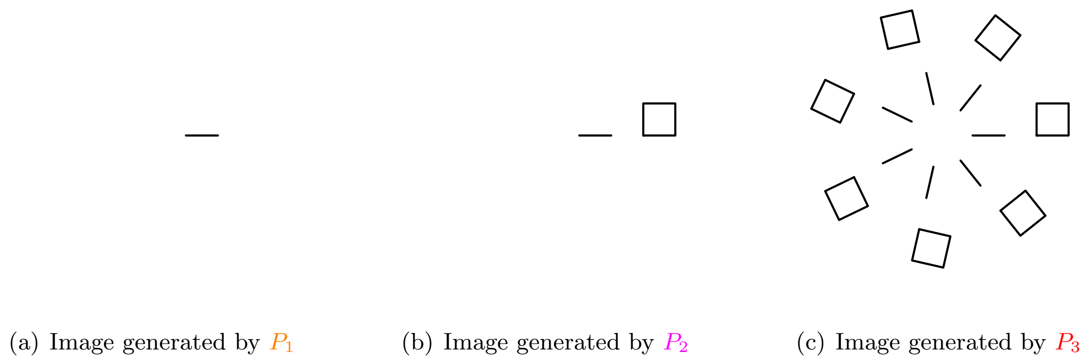
In Example 3, AGS's auxiliary function allowed it to synthesize a long program by performing searches that synthesized three short programs. The auxiliary function is helpful if it is able to capture relevant information about the subprograms of the solution program. In the best case

```
P1:
 PenUp
  Forward(12, 0°)
 Forward(12, 0°)
 PenUp
  Forward(12, 0°)
```

```
P2:
 P1
 Repeat(4):
   Forward(12, 90°)
```

```
P3:
 Repeat(7):
  P2
  Home(64, 64)
  Forward(0, 51.43°)
```

Figure 5.1: Programs synthesized by AGS in the reverse drawing domain.



(a) Image generated by P_1 (b) Image generated by P_2 (c) Image generated by P_3

Figure 5.2: Augmentation process in the reverse drawing domain.

possible, AGS is able to synthesize a program whose AST is of size n by performing n searches where one only considers programs of size at most 2 in each search. In this case, each search synthesizes a program that combines the program inserted in the library in the previous iteration of the algorithm with a production rule from the grammar to synthesize a program of one size larger than the largest program in the library. Performing n small searches is much more efficient than performing one search that synthesizes programs of size n because the number of programs evaluated grows quickly with the program size.

In practice we have to search for larger programs because the auxiliary function might fail to capture that small programs can be part of a solution program. For example, the auxiliary function is not able to capture that the subprogram `PenUp` of P_1 is a subprogram of a solution program because removing the pen from the canvas does not draw on it.

Algorithm 3 Auxiliary-Guided Synthesis (AGS)

```
Procedure: Auxiliary-Guided-Synthesis($\mathcal{G}, (\mathcal{I}, \mathcal{O}), g, B, \mathcal{E}$)
Require: Grammar $\mathcal{G} = (V, \sum, R, S)$, a set of input-output examples $(\mathcal{I}, \mathcal{O})$, Auxiliary Function
    $g$, a budget $\mathcal{B}$, and execution environment, $\mathcal{E}$
Ensure: Solution program $p$ or $\perp$
1: while not timeout and $g$-value is improving do
2:     $p$, solved $\leftarrow$ Base-Synthesizer($\mathcal{G}, (\mathcal{I}, \mathcal{O}), \mathcal{B}, \mathcal{E}$)
3:     if solved then
4:         return $p$
5:     $\mathcal{G} = \mathcal{G} \cup p$
   return $\perp$
```

5.1 Algorithm of Auxiliary-Guided Synthesis (AGS)

The AGS algorithm, presented in Algorithm 3, receives a problem specification through pairs $(\mathcal{I}, \mathcal{O})$, a DSL $\mathcal{G}$, an auxiliary function g, and a computational budget $\mathcal{B}$ (e.g., number of programs evaluated in the search) for base synthesizer. AGS returns either a solution program p or a failure signal $\perp$ as the output.

AGS uses the base synthesizer to explore the space of programs defined by $\mathcal{G}$ to find a solution to the task (line 2). If base synthesizer finds a solution in the search, then AGS returns p, which represents the solution (lines 4). Otherwise, p represents the program that optimizes the auxiliary function g and we add p to the DSL (line 5) as programs of size 1 (i.e., the programs can be used in future searches as ASTs of size 1). This process is repeated while there is still time available for synthesis ("not timeout") and while AGS finds program p that improves the value of the auxiliary function. AGS stops searching and returns failure, as repeating the search would return the same set of programs returned in the previous iteration or the time limit exceeded. The process of growing a library of programs is performed differently for different base synthesizers.

In the next sections, we explain how existing synthesizers can be used with AGS. We denoted the augmented versions of BUS, Bustle, Bee Search, and CrossBeam as A-BUS, A-Bustle, A-Bee, and A-CrossBeam, respectively.

5.1.1 A-BUS, A-Bustle, and A-Bee

The program, p added (line 5) to the language in A-BUS, A-Bustle, and A-Bee is treated as non-terminal symbol of the type p returns. For example, if p returns an integer value, then it can be used as an operation that returns an integer. This means that all programs p have a cost of 1 (size 1 in the context of BUS), so they are evaluated early in the search and can also be more easily combined with other programs in the search to generate the next programs.

5.1.2 A-CrossBeam

In CrossBeam, the program p used to augment the language is added to the initial set of programs the search has explored. When sampling the next program p' to be evaluated in the search, CrossBeam will be able to use p as one of the subprograms of p' starting in the first iteration of the search when it will restart the search.

The model CrossBeam uses to create the probability distribution over existing programs is trained on-policy. This means that CrossBeam uses the distribution of programs seen during the search to solve training problems to train the model. Since the programs p added to the language can be of arbitrary complexity, it is unlikely that the CrossBeam model has trained on search contexts similar to those A-CrossBeam induces (i.e., the set of existing programs might contain complex programs even in the first few iterations of search). We empirically evaluate whether CrossBeam's model is able to generalize to the search contexts of A-CrossBeam.

Since CrossBeam uses sampling, it can benefit from random restarts by avoiding local optima (Hoos and Stützle, 2004). That is, instead of running the system with a computational budget of X program evaluations, if we use N random restarts, we would sequentially perform N independent runs of the system with a budget of $\frac{X}{N}$ program evaluations each. Since A-CrossBeam implicitly performs random restarts, where in each restart it uses an augmented version of the language, it would not be clear from the results if performance improvements were due to random restarts or augmentation. Therefore, we also use a version of CrossBeam that uses random restarts as a baseline in our experiments. We denote as CrossBeam`(N)` the version with restarts, where $N > 1$.

Chapter 6

Empirical Evaluation

In our evaluation, we compare the performance of AGS with that of the base synthesizers on two domains: String Manipulation and Reverse Drawing. We hypothesize that AGS is capable of improving the performance of base synthesizers in terms of the number of solved problems.

6.1 String Manipulation

In the string manipulation domain, we have a set of input/output examples and the goal is to synthesize a program that can generate the desired output from the given input (Alur et al., 2013). Table 6.1 shows an example of such input/output examples. String manipulation tasks arguably represent the most successful application of program synthesis to a real problem through Microsoft Excel's FlashFill feature (Singh and Gulwani, 2015).

6.1.1 Benchmarks, DSL and SetUp

We evaluated AGS on two string manipulation data sets. The first data set is the SyGuS benchmark (Alur et al., 2013), which is a collection of 89 problems from the 2018 PBE Strings and the 2019 PBE SLIA Track. All the input-output examples of those problems are strings only. The second data set consists of 38 handcrafted problems, which are from the Bustle paper (Odena et al., 2021); the problems in this data set were designed to contain some difficult problems. Examples of tasks from these two datasets are shown in the Appendix A.3.

We employ the same DSL, evaluation criteria (meeting the input/output examples), and test problems as those utilized in the Bustle paper (Odena et al., 2021). See Appendix A.1.2 for the DSL used in our study. In this domain, we consider three types: Integers, Strings and Booleans. In our experiments, we set a maximum time limit of 24 hours for solving a problem using 64GB of

Input	Output
Nancy FreeHafer	N. FreeHafer
Andrew Cencici	A. Cencici
...	...
938-242-504	(938) 242 504
324-242-504	(324) 242 504
...	...
Honda125	Honda
Ducati1234	Ducati
...	...
90	90.00
10.0	10.00
...	...

Table 6.1: Example of input/output examples for the string manipulation domain.

RAM, 1 CPU core (Intel Gold 6148 Skylake @ 2.4 GHz) and 1 GPU (4 x NVidia V100SXM2 16G, connected via NVLink). For A-Bee, A-BUS, and A-Bustle, and CrossBeam we use the same time limit and computational resources. However, for A-BUS and BUS, we did not use GPU, since these algorithms do not use a model to guide the search.

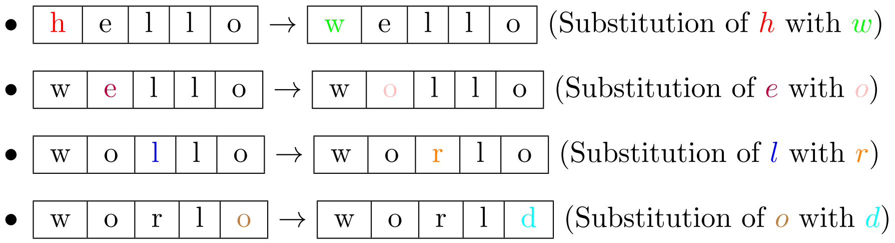
6.1.2 Auxiliary Function

We use the *Levenshtein distance* (Levenshtein, 1966) as our auxiliary function. The Levenshtein distance between two strings is the minimum number of single-character edits (insertions, deletions, or substitutions) required to change one word into the other. Our auxiliary function is the average Levenshtein distance $g(x, y)$ for strings x and y for all input-output examples of the problem. The distance is defined as follows.

$$\mathrm{g}(x,y) = \begin{cases} |x| & \text{if } |y| = 0, \\ |y| & \text{if } |x| = 0, \\ \mathrm{g}(\mathrm{tail}(x), \mathrm{tail}(y)) & \text{if } x[0] = y[0], \\ 1 + \min \begin{cases} \mathrm{g}(\mathrm{tail}(x), y) \\ \mathrm{g}(x, \mathrm{tail}(y)) \\ \mathrm{g}(\mathrm{tail}(x), \mathrm{tail}(y)) \end{cases} & \text{otherwise} \end{cases} \tag{6.1}$$

where the term “tail of string z” refers to a substring consisting of all characters in z, excluding the first character. The notation $z[n]$ denotes the n-th character of the string z, with the indexing starting at 0.

Let us consider two strings, S_1 = ``hello'' and $S_2$ = ``world'' and we want to calculate the Levenshtein distance between them. In Levenshtein distance, we can perform any of three operations at a certain time: insertion, deletion, and substitution.



- | h | e | l | l | o | → | w | e | l | l | o | (Substitution of h with w)
- | w | e | l | l | o | → | w | o | l | l | o | (Substitution of e with o)
- | w | o | l | l | o | → | w | o | r | l | o | (Substitution of l with r)
- | w | o | r | l | o | → | w | o | r | l | d | (Substitution of o with d)

Figure 6.1: Example of Levenshtein Distance Calculation for String Manipulation Domain

Figure 6.1 illustrates an example of Levenshtein distance calculation for two different strings. From the figure, we can see that the Levenshtein distance is 4 between S_1 and S_2.

6.1.3 Results and Discussion

For string manipulation tasks, we present our results in two parts. Firstly, we present the number of problems solved per number of evaluations, aiming to contrast the top-performing algorithm given a consistent expression evaluation budget. Secondly, we present the number of problems solved per running time in seconds to compare the best algorithms with the same computational budget in terms of running time. For the layer of search in A-Bee, A-BUS, and A-Bustle, we set the maximum depth to 7. During a complete search, we set the total number of expression evaluations to $14M$. We conduct the search using the augmentation process for 10 iterations, adding the best program ($k = 1$) from the base search. For base search algorithms (Bee Search, BUS, and Bustle), we do not have any such maximum limit. When presenting the number of problems solved per running time in seconds, we set the maximum time limit to 24 hours for each problem.

For CrossBeam, we set the overall search budget as in the CrossBeam paper (Shi et al., 2022b) which is $50K$ candidate programs and a maximum time limit of 24 hours for each problem. When presenting the number of problems solved per number of evaluations, we restart the search after $\frac{50K}{N}$ evaluations, where N is the number of restarts. When presenting the number of problems solved per running time, we restart the search after $50K$ evaluations and continue until we reach the maximum time limit.

The results of all methods except BUS and A-BUS show the mean result of 5 independent runs. For A-BUS and BUS, we run the search once because the base search algorithm, BUS is deterministic.

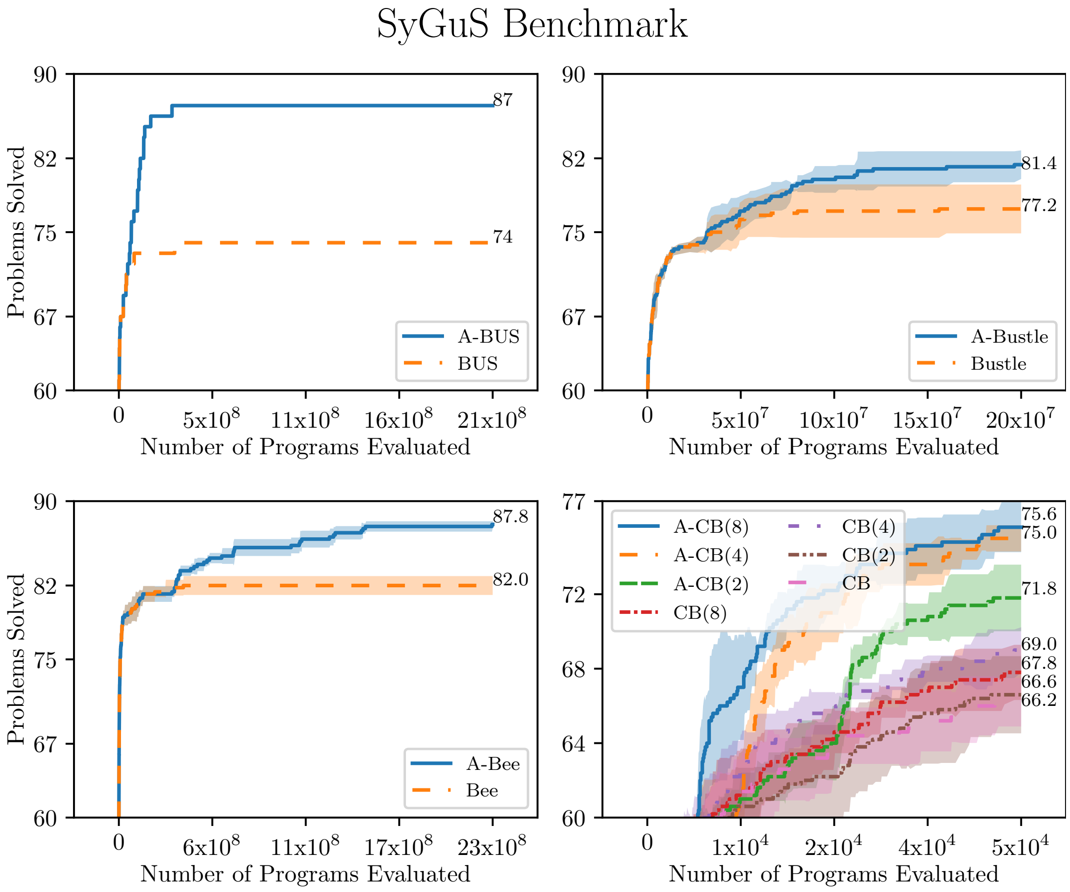


Figure 6.2: Number of problems solved per number of evaluations for the SyGuS benchmark, which has 89 problems. A-Y indicates the algorithm Y used with AGS, where Y is base search algorithm. CB indicates the CrossBeam method and A-CB(N) or CB(N) indicates the CrossBeam method with N restarts.

Figure 6.2 shows the number of problems solved per number of evaluations for the SyGuS benchmark. Each of the four distinct plots presents the results for the base search algorithm and their language-augmented version. The upper-left plot of the figure presents the comparison between A-BUS and BUS. Here, A-BUS emerges as the most successful algorithm, never performing worse than BUS. A-BUS solved 87 problems against BUS's 74.

Moving to the upper-right quadrant, the result of A-Bustle and Bustle reveals a similar pattern; A-Bustle bests Bustle in the number of solved problems, with the final of 81.4 to 77.2, respectively. In the bottom-left quadrant, a comparison between A-Bee and Bee Search is presented. It becomes evident that A-Bee demonstrates superior performance over Bee Search with respect to the number of problems solved. Specifically, A-Bee consistently equals or surpasses Bee Search, eventually solving 87.8 problems compared to Bee Search's 82.0.

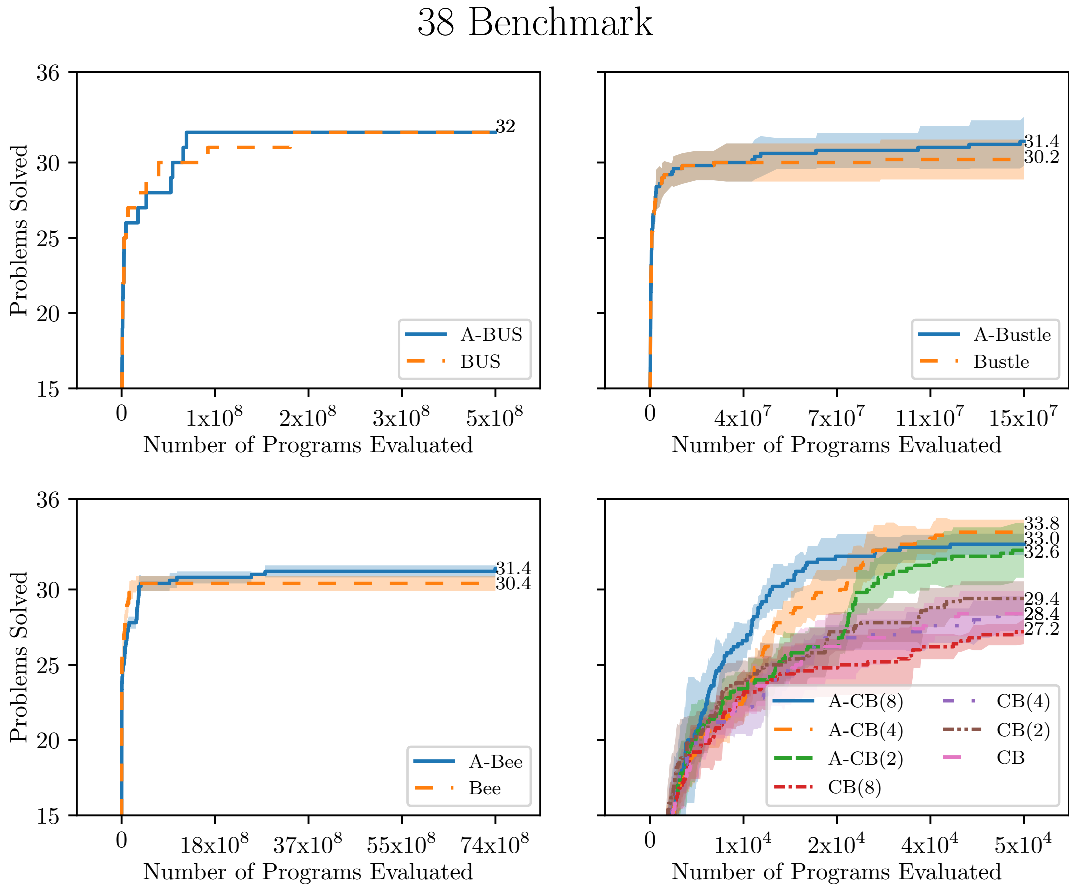


Figure 6.3: Number of problems solved per number of evaluations for the Bustle's 38 handcrafted string manipulation problems. A-Y indicates the algorithm Y with AGS, where Y is base search algorithm. CB indicates the CrossBeam method and A-CB(N) or CB(N) indicates the CrossBeam with N restarts.

Lastly, the bottom-right quadrant provides a comparative study of different versions of A-CrossBeam and CrossBeam across varied restart conditions. The results indicate that all A-CrossBeam variants surpassed the CrossBeam variants. Two specific cases, A-Crossbeam(2) and A-Crossbeam(4), exhibited slightly lower performance than Crossbeam(8) and Crossbeam(4), but all A-CrossBeam methods solved more problems when allowed more computation. Among these, the best performing A-CrossBeam variant was Crossbeam(8), solving 75.6 problems, while the best CrossBeam variant was A-Crossbeam(8), solving 69.0 problems.

Figure 6.3 depicts the evaluation of Bustle's 38 handcrafted benchmark. The plots show the number of problems solved by the number of programs evaluated. Similarly to the previous figure, Figure 6.3 also contains four different subplots. Beginning with the upper-left plot, the results show that both BUS and A-BUS solve the same number of problems once enough computational resources are given.

The upper-right plot shows that the result of A-Bustle marginally exceeds Bustle in the number of problems solved, with 31.4 and 30.2 problems solved, respectively. The bottom-left plot shows the results for A-Bee and Bee Search. Interestingly, although A-Bee starts on par with or even slightly behind Bee Search in the early stages of the search, it concludes with 31.4 solved problems, surpassing Bee Search's final count of 30.4. The final plot, located in the lower-right corner provides a comparative analysis of different variants of A-CrossBeam and CrossBeam with different number of restarts. All A-CrossBeam variants generally outperformed the CrossBeam variants. The top-performing A-CrossBeam variant was `A-Crossbeam(4)`, which solved 33.8 problems. The most successful CrossBeam variant was `Crossbeam(2)`, which solved only 29.4 problems.

In general, AGS-based approaches are never worse and often better than the base algorithms. These results support our hypothesis that the auxiliary function AGS uses for string manipulation tasks allows it to solve more problems than the base algorithms.

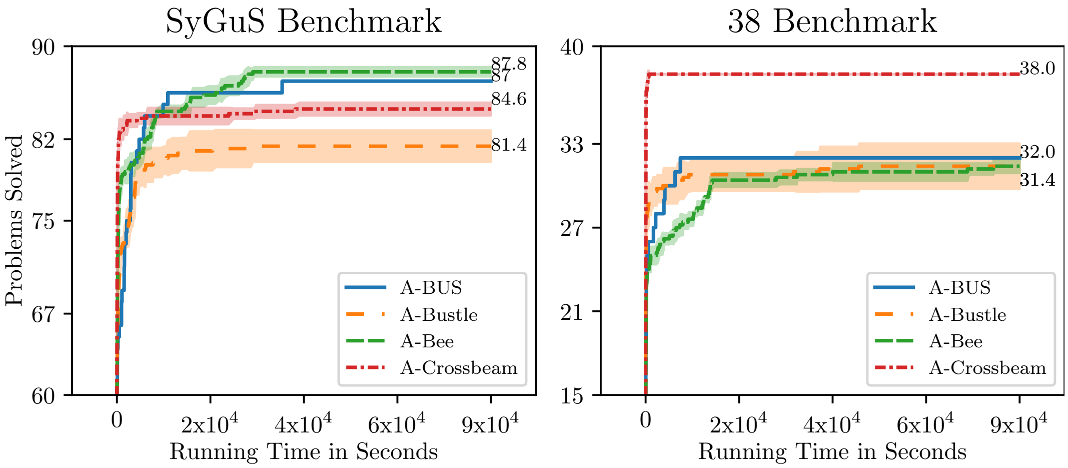


Figure 6.4: Number of problems solved per to running time in seconds for the SyGuS benchmark and the 38 handcrafted benchmark. `A-Y` indicates the algorithm `Y` used with AGS, where `Y` is base search algorithm.

Figure 6.1.3 shows the number of problems solved by running time in seconds, for both string manipulation benchmarks. In this figure, we present the best-performing method of each base algorithm for each benchmark. For the SyGuS benchmark, A-Bee outperforms all other methods, ultimately solving 87.8 problems. Although A-BUS initially performed worse than others, it eventually emerged as the second-best algorithm, solving 87 problems. In contrast, A-CrossBeam rapidly reached its final count of 84.6 problems solved, but did not progress beyond that point. A-Bustle remained consistent in its performance but trailed the others, solving 81.4 problems.

In the 38 benchmark, A-CrossBeam significantly outperform all other language-enhanced methods by solving all 38 problems in a short period of time. Although A-BUS lagged behind others during the initial stages, it quickly reached its final count of 32 problems, outpacing both A-Bee and A-Bustle. In particular, despite A-Bee falling behind, both A-Bee and A-Bustle eventually solved 31.4 problems each. The AGS version of Bee Search, A-Bee, solves almost all problems from the SyGuS competition, while A-CrossBeam solved all problems of the 38 benchmark. These two algorithms not only represent the current state of the art in these data sets but are also signalling that the community is in need of more challenging benchmark data sets.

6.2 Reverse Drawing

In the reverse drawing domain, we have a set of black-and-white images, and the goal is to synthesize a program that can generate the desired image from that program. Some examples of input images are shown in Figure 6.5.

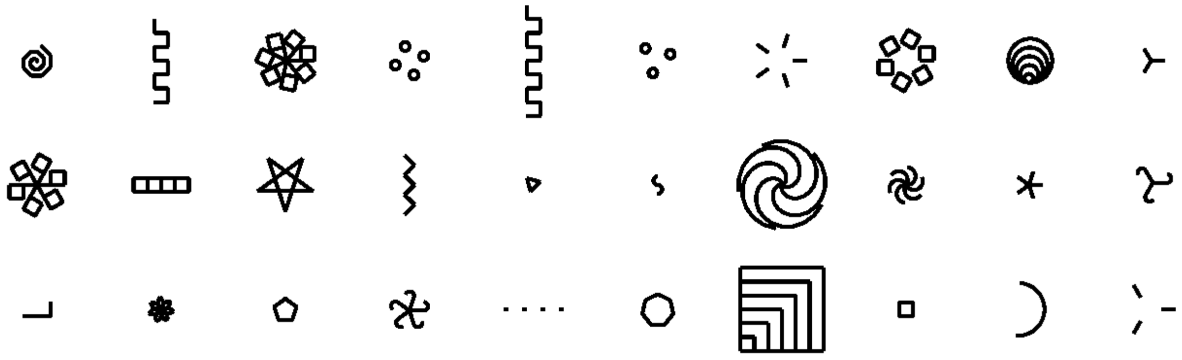


Figure 6.5: Example of Reverse Drawing tasks (Ellis et al., 2020).

6.2.1 Benchmarks, DSL and SetUp

For this experiments, we use the 111 test images from DreamCoder (Ellis et al., 2020) that are originally from Abelson and DiSessa (1986). However, we redraw those images in our customized Python turtle graphics environment with a line thickness of 1 pixel.

To compare with previous work, we adopted the same domain-specific language (DSL). This DSL comprises basic drawing instructions such as `Forward`, `Repeat`, `PenUp`, and `Home`, which are commonly used in the turtle graphics environment in Python. The DSL contains basic arithmetic operators such as `Add(+)`, `Subtract(-)`, `Multiply(*)`, and `Divide(/)`. We consider four types in this domain: Lengths, Angles, Booleans and Integers. See Appendix A.1.1 for a full description of the DSL used in our study.

For our experiment, we set a maximum time limit of `48 hours` for solving a problem using `64GB` of RAM, and 8 CPU cores (Intel Gold 6148 Skylake @ 2.4 GHz). We use the same time limit and computational resources for A-BUS, and BUS.

6.2.2 Auxiliary Function

The auxiliary function we use in our experiment measures the overlap between the two different images. Let I be the input image and $\hat{I}$ be the image drawn using the candidate program, and let $C = \hat{I} - I$. We consider the empty canvas as a matrix with zeros, and entries with 1 represent pixels drawn on the canvas. Our dissimilarity metric $d(C)$ is defined as follows:

$$d(C) = \begin{cases} \infty & \text{if } \exists (x, y) \text{ such that } C(x, y) = 1 \\ \sum_{x,y} \mathbf{1}[C_{x,y} = -1] & \text{otherwise} \end{cases} \tag{6.2}$$

Where, $C_{x,y}$ refers to the x-th row and y-th column of the matrix C and $\mathbf{1}[.]$ is the indicator function. In that case, the dissimilarity metric is set to ∞. We use three-step rules to break the ties among the candidate programs. Firstly, in case of a tie with dissimilarity, we prefer the program where the pen is located next to a pixel that still needs to be drawn; secondly, we prefer the program where the pen is located at the edge of a line; if the tie still remains, we prefer the program where the pen is located in a position (x, y) on the canvas with longest straight line starting from (x, y) with pixels yet to be drawn. The rest of the ties are broken randomly.

Let us consider the running example of calculating the dissimilarity metric in Figure 6.6 and Figure 6.7. If the original image is shown in Figure 6.6(b) and the image drawn by the program is shown in Figure 6.6(a), then the dissimilarity is calculated as shown in Equation 6.2:

0	0	0	0
0	1	0	0
0	0	1	0
0	0	0	0

(a) Image drawn

0	0	0	0
0	1	1	0
0	1	1	0
0	0	0	0

(b) Original image

0	0	0	0
0	0	-1	0
0	-1	0	0
0	0	0	0

(c) Difference

Dissimilarity = 2

Figure 6.6: Example of calculating the dissimilarity metric.

On the otherhand, if the original image is shown in Figure 6.7(b) and the image drawn by the program is shown in Figure 6.7(a), then the dissimilarity is calculated as shown in Equation 6.2:

1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	0

(a) Image drawn

0	0	0	0
0	1	1	0
0	1	1	0
0	0	0	0

(b) Original image

1	0	0	0
0	0	-1	0
0	-1	0	0
0	0	0	0

(c) Difference

Dissimilarity $= \infty$

Figure 6.7: Example of calculating the dissimilarity when the image drawn by the program draws a pixel that is not present in the original image.

6.2.3 Results and Disscussion

We choose BUS as our base algorithm in the reverse drawing domain, due to the lack of cost-guided search algorithms for this domain. For each problem, we set the maximum number of expression evaluations to `14M` for a complete search. However, we selected the maximum bound of the search tree from 4 to 7 for each problem. We select the best program ($k = 1$) for each depth and add it to the initial language. We perform the search with the augmentation process until we reach the maximum time limit of 48 hours.

In this experiment, we compared the performance of A-BUS with BUS, DreamCoder and variants of LAPS. DreamCoder learns a library of programs and trains a neural network model to guide the search on a set of 200 tasks. Wong et al. (2021) presented Language for Abstraction and Program Search (LAPS), which uses natural language annotations of the tasks (e.g., "draw a triangle next to a square"). Similarly to DreamCoder, LAPS also learns a library of programs with the help of those annotations and trains generative models to guide the search on a set of 200 tasks and annotations. "LAPS in neural search" is the LAPS variant that only performs a neural search where no language hints are used during testing. "LAPS + mutual exclusivity (ME)" is the LAPS variant that uses linguistic priors. "LAPS + ME + Compression" is the LAPS variant that uses the search mutual exclusivity (ME) constraint and language-program compression.

The results for DreamCoder and LAPS, along with their variants, are taken from Wong et al. (2021), which do not provide detailed information on the amount of computational resources used in their experiments. Therefore, the comparison we make with such systems is only approximate. While LAPS and its variants leverage both images and textual annotations, DreamCoder, BUS, and A-BUS, only receive the image as input. The numbers presented for BUS, and A-BUS, are for a single run of the systems, as they are deterministic. On the other hand, the numbers for DreamCoder, and LAPS, represent the average of four independent runs of these systems.

Method	Accuracy (%)
BUS	11.71
DreamCoder	42.64
A-BUS	52.25
LAPS in neural search	52.93
LAPS + mutual exclusivity (ME)	80.18
LAPS + ME + Compression	81.98

Table 6.2: Reverse Drawing task evaluation. Values are in the percentage of problems solved from a set with 111 problems.

In Table 6.2, we present the results of our proposed language-augmentation approach, which solved 52.25% of the 111 reverse drawing problems, where the base search algorithm BUS solved only 11.71%. The result shows that A-BUS outperformed BUS and DreamCoder, and is competitive with the variant of LAPS that only performs a neural search. The result also shows that the language-augmentation approach is effective in improving the performance of the base search algorithm.

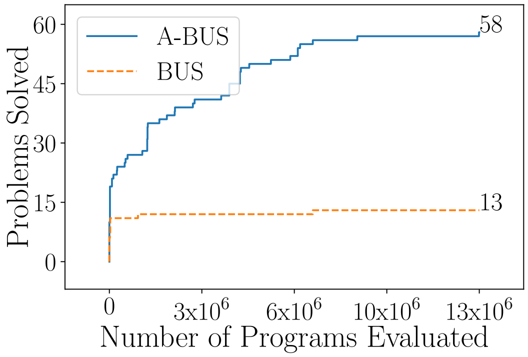


Figure 6.8: Reverse Drawing task evaluation. Number of problems solved per number of evaluations for the reverse drawing problems. `A-Y` indicates the algorithm `Y` with AGS, where `Y` is base search algorithm.

Figure 6.8 shows the number of problems solved by the number of programs evaluated in the reverse drawing domain. We only included the result from A-BUS and BUS, as we do not have access to the data from the experiments of DreamCoder and LAPS. This figure shows that A-BUS outperforms BUS even for small numbers of evaluations.

Chapter 7

Conclusions

In this dissertation, we introduced Auxiliary-Guided Synthesis (AGS), a framework designed to guide the search in the space of programs by learning libraries of programs while solving a problem. It uses a base synthesizer algorithm to find the solution to the problem; if it fails, it adds a program that best optimizes the auxiliary function. This process of search and augmentation continues until it finds a solution, cannot find a program with a better value of the auxiliary function or exceeds the predefined time limit.

The empirical results of the string manipulation domain show that our augmentation method offers orthogonal search guidance to existing functions by improving the performance of state-of-the-art systems, including BUS, Bustle, Bee Search, and CrossBeam. Although we use a simple auxiliary function, our augmentation approach A-BUS can outperform DreamCoder in the reverse drawing domain, if enough computational resources are provided. In conclusion, our empirical results suggest that our approach of augmentation can boost the performance of synthesizers by incorporating domain knowledge to it through a library-learning process.

Chapter 8

Future Work

In future work, we plan to dive deeper into the sensitivity of injecting more domain knowledge into the system. We will also explore the implications of using auxiliary-guided synthesis in other domains such as heuristic function synthesis and even areas like reinforcement learning where additional reward signals could be of value. Furthermore, we aim to investigate the effect of employing different auxiliary functions, like longest common subsequences, for string manipulation tasks. It would also be insightful to examine the influence of the library learning approach using auxiliary functions in the context of top-down search synthesis. Moreover, while manually checking the length of the synthesized programs, we have observed that some programs are longer than the shortest solution possible due to augmentation. Since we generally prefer shorter programs, exploring strategies to maintain compact program sizes in future research endeavors can be important.

References

H. Abelson and A. A. DiSessa. *Turtle Geometry: The Computer as a Medium for Exploring Mathematics*. MIT Press, 1986.

Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. Recursive program synthesis. In *Proceedings of 25th International Conference on Computer Aided Verification, Saint Petersburg, Russia*, pages 934–950, 2013.

Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo Martin, Mukund Raghothaman, Sanjit Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Proceedings of Formal Methods in Computer-Aided Design, FMCAD*, pages 1–17, 10 2013.

Saqib Ameen and Levi H.S. Lelis. Program synthesis with best-first bottom-up search. *Journal of Artificial Intelligence Research*, 2023.

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

Shraddha Barke, Hila Peleg, and Nadia Polikarpova. Just-in-time learning for bottom-up enumerative synthesis. *In Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–29, 2020.

JE Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 1965.

Vadim Bulitko, Shuwei Wang, Justin Stevens, and Levi HS Lelis. Portability and explainability of synthesized formula-based heuristics. In *Proceedings of the International Symposium on Combinatorial Search*, pages 29–37, 2022.

Andrew Cropper and Sebastijan Dumancic. Learning large logic programs by going beyond entailment. In Christian Bessiere, editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI*, pages 2073–2079, 2020.

Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy I/O. In *Proceedings of International Conference on Machine Learning*, pages 990–998, 2017.

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

Kevin Ellis, Catherine Wong, Maxwell I. Nye, Mathias Sablé-Meyer, Luc Cary, Lucas Morales, Luke B. Hewitt, Armando Solar-Lezama, and Joshua B. Tenenbaum. Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning. *CoRR*, 2020.

Nathanaël Fijalkow, Guillaume Lagarde, Théo Matricon, Kevin Ellis, Pierre Ohlmann, and Akarsh Potta. Scaling neural program synthesis with distribution-based search. In *AAAI*, 2022.

Sergio Poo Hernandez and Vadim Bulitko. Speeding up heuristic function synthesis via extending the formula grammar. In *Proceedings of the International Symposium on Combinatorial Search*, volume 12, pages 233–235, 2021.

H. H. Hoos and T. Stützle. *Stochastic Local Search: Foundations & Applications*. Elsevier / Morgan Kaufmann, 2004. ISBN 1-55860-872-9.

Idress Husien and Sven Schewe. Program generation using simulated annealing and model checking. In Rocco De Nicola and Eva Kühn, editors, *Proceedings of Software Engineering and Formal Methods*, pages 155–171, 2016.

Ruyi Ji, Yican Sun, Yingfei Xiong, and Zhenjiang Hu. Guiding dynamic programing via structural probability for accelerating programming by example. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–29, 2020.

Ashwin Kalyan, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. Neural-guided deductive search for real-time program synthesis from examples. In *Proceedings of International Conference on Learning Representations*, 2018.

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

John R Koza. Genetic programming as a means for programming computers by natural selection. *Statistics and computing*, 4(2):87–112, 1994.

Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. Accelerating search-based program synthesis using learned probabilistic models. *ACM SIGPLAN Notices*, 53(4):436–449, 2018.

Vladimir Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Proceedings of Soviet Physics Doklady*, pages 707–710. Soviet Union, 1966.

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

Augustus Odena and Charles Sutton. Learning to represent programs with property signatures. In *Proceedings of International Conference on Learning Representations*, 2020.

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

Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. *ACM SIGARCH Computer Architecture News*, 41(1):305–316, 2013.

Kensen Shi, David Bieber, and Charles Sutton. Incremental sampling without replacement for sequence models. In *Proceedings of International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 8785–8795, 2020.

Kensen Shi, David Bieber, and Rishabh Singh. Tf-coder: Program synthesis for tensor manipulations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 44(2):1–36, 2022a.

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

Rishabh Singh and Sumit Gulwani. Predicting a correct program in programming by example. In *Proceedings of International Conference on Computer Aided Verification*, pages 398–414, 2015.

Armando Solar-Lezama. The sketching approach to program synthesis. In *APLAS*, 2009.

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

Abhishek Udupa, Arun Raghavan, Jyotirmoy V Deshmukh, Sela Mador-Haim, Milo MK Martin, and Rajeev Alur. Transit: specifying protocols with concolic snippets. *ACM SIGPLAN Notices*, 48(6):287–296, 2013.

Stefan Van der Walt, Johannes L Schönberger, Juan Nunez-Iglesias, François Boulogne, Joshua D Warner, Neil Yager, Emmanuelle Gouillart, and Tony Yu. scikit-image: image processing in python. *PeerJ*, 2:e453, 2014.

Xinyu Wang, Isil Dillig, and Rishabh Singh. Program synthesis using abstraction refinement. *In Proceedings of ACM Programing Languages*, 2(POPL), Dec 2017.

Catherine Wong, Kevin M Ellis, Joshua Tenenbaum, and Jacob Andreas. Leveraging language to learn program abstractions and search heuristics. In *Proceedings of International Conference on Machine Learning*, pages 11193–11204, 2021.

Amit Zohar and Lior Wolf. Automatic program synthesis of long programs with a learned garbage collector. *Advances in neural information processing systems*, 31, 2018.

Appendix A

Appendix

A.1 Domain Specific Languages (DSLs)

A.1.1 DSL for Reverse Drawing Domain

Expression, S	$\rightarrow$	$\lambda \mid \lambda \cdot \lambda$
Initial Programs, λ	$\rightarrow$	$\text{Forward}(\mathcal{L}, \Theta, N) \mid \text{Repeat}(S, N, F) \mid \text{PenUp}(S) \mid \text{Home}(S)$
Angle, Θ	$\rightarrow$	$\Theta_u \mid \Theta_0 \mid \Theta_u \ominus N \mid \Theta \ominus N \mid \Theta \ominus \Theta \mid \Theta_\in$
Length, $\mathcal{L}$	$\rightarrow$	$\mathcal{L}_u \mid \mathcal{L}_0 \mid \mathcal{L}_u \ominus N \mid \mathcal{L} \ominus \mathcal{L} \mid \mathcal{L} \in$
Operator, $\ominus$	$\rightarrow$	$+ \mid - \mid * \mid /$
Iterations, Constants and Multiplier, N	$\rightarrow$	$1 \mid 2 \mid 3 \mid \ldots \mid 9 \mid 20$
Unit Length, $\mathcal{L}_u$	$\rightarrow$	12
Zero Length, $\mathcal{L}_0$	$\rightarrow$	0
Epsilon Length, $\mathcal{L}_\in$	$\rightarrow$	0.5
Unit Angle, Θ_u	$\rightarrow$	360°
Zero Angle, Θ_0	$\rightarrow$	0°
Epsilon Angle, $\Theta_\in$	$\rightarrow$	9°
Flag, F	$\rightarrow$	$\text{True} \mid \text{False}$

Figure A.1: DSL considered for the reverse drawing domain.

A.1.2 DSL for String Processing Domain

Expression, E	→	$S \mid I \mid B$
String expression, S	→	Concat$(S1, S2)$ $\mid$ Left(S, I) $\mid$ Right(S, I) $\mid$ Substr$(S, I1, I2)$
		$\mid$ Replace$(S1, I1, I2, S2)$ $\mid$ Trim(S) $\mid$ Repeat(S, I)
		$\mid$ Substitute$(S1, S2, S3)$ $\mid$ Substitute$(S1, S2, S3, I)$
		$\mid$ ToText(I) $\mid$ LowerCase(S) $\mid$ UpperCase(S) $\mid$ ProperCase(S)
		$\mid T \mid X \mid$ If$(B, S1, S2)$
Integer expression, I	→	$I1 + I2 \mid I1 - I2 \mid$ Find$(S1, S2)$ $\mid$ Find$(S1, S2, I)$ $\mid$ Len(S) $\mid J$
Boolean expression, B	→	Equals$(S1, S2)$ $\mid$ GreaterThan$(I1, I2)$
		$\mid$ GreaterThanOrEqualTo$(I1, I2)$
String constants, T	→	“” $\mid$ “ ” $\mid$ “,” $\mid$ “.” $\mid$ “!” $\mid$ “?” $\mid$ “(” $\mid$ “)” $\mid$ “[” $\mid$ “]”
		$\mid$ “<” $\mid$ “>” $\mid$ “{” $\mid$ “}” $\mid$ “-” $\mid$ “+” $\mid$ “_” $\mid$ “/” $\mid$ “$” $\mid$ “#”
		$\mid$ “:” $\mid$ “;” $\mid$ “@” $\mid$ “%” $\mid$ “0”
		$\mid$ string constants extracted from I/O examples
Integer constants, J	→	$0 \mid 1 \mid 2 \mid 3 \mid 99$
Input, X	→	$x_1 \mid \ldots \mid x_k$

Figure A.2: DSL considered for the string processing domain.

A.2 Bresenham's Line Drawing Algorithm for Reverse Drawing Domain

To draw the lines in reverse drawing domain, we use the Bressenham's line drawing algorithm (Bresenham, 1965). The algorithm is used to draw a line between two points in a grid. The algorithm is based on the idea of calculating the error at each step and then deciding whether to move in the x-direction or y-direction. The algorithm is shown in Algorithm 4:

Algorithm 4 Bressenham's line drawing algorithm

```
Procedure: BRESSENHAM($x_1, y_1, x_2, y_2$)
Require: Start Coordinate, $(x_1, y_1)$, End Coordinate, $(x_2, y_2)$
$dx \leftarrow x_2 - x_1$                          ▷ Change in x-coordinate
$dy \leftarrow y_2 - y_1$                          ▷ Change in y-coordinate
$d \leftarrow 2 * dy - dx$                         ▷ Initial error
$y \leftarrow y_1$
for $x \leftarrow x_1$ to $x_2$ do                 ▷ Iterate over x-coordinates
    $plot(x, y)$
    if $d > 0$ then
        $y \leftarrow y + 1$                       ▷ Move in y-direction
        $d \leftarrow d + 2 * (dy - dx)$           ▷ Update error
    else
        $d \leftarrow d + 2 * dy$                  ▷ Update error
```

We used skimage (Van der Walt et al., 2014) to implement the algorithm on custom python turtle (Abelson and DiSessa, 1986) environment.

A.3 Example of String Manipulation tasks from Dataset

```
; https=//stackoverflow.com/questions/2171308/
how-to-make-a-sub-string-selection-and-concatenation-in-excel
(set-logic SLIA)
(synth-fun f ((_arg_0 String)) String
 ( (Start String (ntString))
 (ntString String (
  _arg_0
  "" " "
  (str.++ ntString ntString)
  (str.replace ntString ntString ntString)
  (str.at ntString ntInt)
  (int.to.str ntInt)
  (ite ntBool ntString ntString)
  (str.substr ntString ntInt ntInt)
))
 (ntInt Int (

  1 0 -1
  (+ ntInt ntInt)
  (- ntInt ntInt)
  (str.len ntString)
  (str.to.int ntString)
  (ite ntBool ntInt ntInt)
  (str.indexof ntString ntString ntInt)
))
 (ntBool Bool (

  true false
  (= ntInt ntInt)
  (str.prefixof ntString ntString)
  (str.suffixof ntString ntString)
  (str.contains ntString ntString)
)) ))
(constraint (= (f "John Doe") "J Doe"))
(constraint (= (f "Mayur Naik") "M Naik"))
(constraint (= (f "Nimit Singh") "N Singh"))
(check-synth)
(define-fun f_1 ((_arg_0 String)) String (str.replace _arg_0
    (str.substr _arg_0 1 (str.indexof _arg_0 " " 1)) " "))
```

Figure A.3: Example of string manipulation task from 89 SyGuS string manipulation tasks dataset.

```
{
"name": "capitalizeCityAndState",
"description": "fix capitalization of city and state",
"trainExamples": [
    {
    "inputs": [
        "mountain view, ca"
    ],
    "output": "Mountain View, CA"
    },
    {
    "inputs": [
        "HOUSTON, TX"
    ],
    "output": "Houston, TX"
    }
],
"testExamples": [
    {
    "inputs": [
        "seattle, wa"
    ],
    "output": "Seattle, WA"
    },
    {
    "inputs": [
        "St. LOUIS, mo"
    ],
    "output": "St. Louis, MO"
    },
    ...,
    {
    "inputs": [
        "aa aaa a, bc"
    ],
    "output": "Aa Aaa A, BC"
    }
],
"tables": [],
"expectedProgram": "CONCATENATE(LEFT(PROPER(var_0), MINUS(LEN(var_0),
   1)), UPPER(RIGHT(var_0, 1)))",
"tags": []
}
```

Figure A.4: Example of string manipulation task from 38 Bustle dataset.