Guiding Dynamic Programing via Structural Probability for Accelerating Programming by Example

RUYI JI, Peking University, China
YICAN SUN, Peking University, China
YINGFEI XIONG*, Peking University, China
ZHENJIANG HU, Peking University, China

Programming by example (PBE) is an important subproblem of program synthesis, and PBE techniques have been applied to many domains. Though many techniques for accelerating PBE systems have been explored, the scalability remains one of the main challenges: There is still a gap between the performances of state-of-the-art synthesizers and the industrial requirement. To further speed up solving PBE tasks, in this paper, we propose a novel PBE framework *MaxFlash*. *MaxFlash* uses a model based on structural probability, named topdown prediction models, to guide a search based on dynamic programming, such that the search will focus on subproblems that form probable programs, and avoid improbable programs. Our evaluation shows that *MaxFlash* achieves ×4.107 – ×2080 speed-ups against state-of-the-art solvers on 244 real-world tasks.

CCS Concepts: • **Software and its engineering → Software notations and tools**; **General programming languages**.

Additional Key Words and Phrases: Programming by Example, Dynamic Programming, Probabilistic Model

ACM Reference Format:
Ruyi Ji, Yican Sun, Yingfei Xiong, and Zhenjiang Hu. 2020. Guiding Dynamic Programing via Structural Probability for Accelerating Programming by Example. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 224 (November 2020), 29 pages. https://doi.org/10.1145/3428292

1 INTRODUCTION

Programming by example (PBE) is an important subproblem of program synthesis where the synthesis system is required to learn a program from input-output examples. PBE problem is important because (1) many practical synthesis problems are instances of PBE, and (2) the general synthesis problem of synthesizing a program from a logic specification can be converted into PBE by CEGIS framework [Solar-Lezama et al. 2006]. While the studies of building efficient PBE systems have been proceeding for four decades [Shaw et al. 1975], there is a surge of interest in applying PBE techniques to different domains in the past decade, such as string manipulation [Barowy et al.

*Corresponding author

Authors' addresses: Ruyi Ji, Key Lab of High Confidence Software Technologies, Ministry of Education Department of Computer Science and Technology, EECS, Peking University, Beijing, China, jiruyi910387714@pku.edu.cn; Yican Sun, Key Lab of High Confidence Software Technologies, Ministry of Education Department of Computer Science and Technology, EECS, Peking University, Beijing, China, sycpku@pku.edu.cn; Yingfei Xiong, Key Lab of High Confidence Software Technologies, Ministry of Education Department of Computer Science and Technology, EECS, Peking University, Beijing, China, xiongyf@pku.edu.cn; Zhenjiang Hu, Key Lab of High Confidence Software Technologies, Ministry of Education Department of Computer Science and Technology, EECS, Peking University, Beijing, China, huzj@pku.edu.cn.

2475-1421/2020/11-ART224
https://doi.org/10.1145/3428292

2015; Gulwani 2011], data wrangling [Chen et al. 2019; Feng et al. 2017; Yaghmazadeh et al. 2016], SQL queries [Wang et al. 2017; Zhang and Sun 2013].

Despite many existing applications, scalability remains one of the main challenges to apply PBE techniques. As summarized by Polozov and Gulwani [2016], many PBE systems are interactive systems, and a user-interacting PBE system of industrial quality should respond within 500 milliseconds. The state-of-the-art solvers fail to meet this requirement on many tasks. As will be shown later, the best solver in SyGuS competition 2019 only solves 123 out of 205 real-world string-manipulation tasks within 500ms in our evaluation.

To optimize the performance of PBE, many different approaches have been proposed. One important technique is dynamic programming. Dynamic-programming-based approaches deductively divide the synthesis problem into subproblems of synthesizing smaller programs, where the solutions to the subproblems can be reused. For example, the problem of synthesizing an expression returning "aa" can be divided into two subproblems, each of them synthesizes an expression returning "a", such that concatenating the two expressions gives a solution to the original problem. Moreover, the result of the first subproblem can be reused for the second one since they have the same requirement. A representative dynamic-programming-based approach is *PROSE* [Polozov and Gulwani 2015], which uses pre-defined rules over operators, called *witness functions*, to divide a synthesis problem into subproblems. *PROSE* framework has been used to construct many different applications [Barowy et al. 2015; Gulwani 2011; Kini and Gulwani 2015; Le and Gulwani 2014; Padhi et al. 2018; Singh and Gulwani 2012] including *FlashFill* [Gulwani 2011], a PBE system for automatically synthesizing string manipulation programs in spreadsheets.

Recent evidence suggests that probabilistic models based on structural probability could be used to accelerate program synthesis. Though in theory a rich space of programs can be written, in practice programs always fall into a small subspace that is predictable, and can be modeled by a statistical model that relies only on the structure of a program. For example, an expression $a + 1$ is more probable than $a - 1 + 2$. *Euphony* [Lee et al. 2018], a recently proposed approach, uses structural probability to accelerate enumerative search: It uses a learned probabilistic model to model the structural probability and enumerates the programs in the descending order of the probability until one is verified to be correct. The results show that *Euphony* achieves significant speed-ups. Compared with other probabilistic models such as a conditional probabilistic model over a context (e.g., a natural language description), the advantage of structural probability is that it can be easily modeled by a lightweight model, leading to great advantage on the speed.

Though *Euphony* successfully uses structural probability to accelerate the enumerative search, it is still unknown how to use structural probability to guide dynamic programming – one of the most important directions for accelerating PBE. In this paper we solve this problem by proposing a novel framework, *MaxFlash*, to utilize both dynamic programming and structural probability for efficiently solving PBE problems. *MaxFlash* follows *PROSE* to use witness functions to divide problems, and thus can be easily applied to a large number of existing *PROSE* applications where witness functions have already been defined. Our evaluation on 244 synthesis problems for string manipulation and matrix transformation shows that ***MaxFlash* achieves** $\times 4.107 - \times 2080$ **speed-ups against six state-of-the-art solvers**, namely *PROSE* [Polozov and Gulwani 2015], *Euphony* [Lee et al. 2018], *Eusolver* [Alur et al. 2017b], *CVC4* [Reynolds et al. 2019a], *Atlas* [Wang et al. 2018a] and *NGDS* [Kalyan et al. 2018]. Besides, we also compare the probabilistic model used in *MaxFlash* with *DeepCoder* [Balog et al. 2017], a state-of-the-art framework on training probabilistic models for synthesizers: The result demonstrates the advantage of *MaxFlash*.

Designing *MaxFlash* requires to address a series of significant challenges. **The first major challenge is the gap between the locality of subproblems in dynamic programming and the globality of structural probability**. Intuitively, we would like to use structural probability to avoid solving subproblems forming improbable programs. However, a subproblem contains only a fragment of a program, and we cannot know the structural probability of a whole program by examining only a local fragment. For example, an expression `Inc(a)` is probable, but becomes improbable when forming an improbable combination with other operations, e.g., `Dec(Inc(a))`, or used together with improbable components, e.g., `(a/0)*Inc(a)`.

To solve this problem, we introduce a novel subproblem definition that allows the local search of a program fragment to be aware of the global probability. More concretely, the new subproblem contains two additional parameters:

- The first one is a *context*, which captures the context of surrounding programs to calculate the probability of the current subprogram. For example, when the context is `Dec(?)`, `Inc(a)` is not a probable choice. In particular, we design a special probabilistic model, named topdown prediction model, for efficiently maintaining the context during dynamic programming: In a topdown prediction model, the context captures only the information from the ancestors but not siblings, such that the probability calculation of sibling subproblems are independent from each other, allowing subproblems to be searched independently.
- The second one is a *probability lowerbound*, which is a key component for performing branch-and-bound [Land and Doig 1960]. The probability lowerbound is propagated from parent subproblems to children subproblems to give a lowerbound on the probability for the current subprogram to form a globally probable program. For example, all the programs with form `(a/0)*?` will be ignored if `(a/0)` violates the probability lowerbound for its subproblem.

We use an iteratively deepening search to focus on probable programs: we start with a high probability lowerbound for the whole program and iteratively decrease the lowerbound until a solution is found. Besides, we use branch-and-bound to effectively prune off improbable search branches: We introduce a heuristic function which estimates the probability upperbound of valid programs to a subproblem, such that (1) a subproblem can be pruned off immediately once its heuristic value is smaller than the lowerbound, (2) the lowerbound of a subproblem can be better calculated by considering the heuristic values of its sibling subproblems.

The second major challenge is the chance of reusing subproblems. Standard dynamic programming reuses solutions of subproblems with exactly the same set of parameters. However, after adding the lowerbound, the chance for reusing a subproblem becomes extremely small.

To solve this problem, we turn the subproblems into optimization problems: Instead of searching for any valid program, we require to search for the program with the maximum probability. In this way, we can reuse the solution of a subproblem for another with a different lowerbound. To further boost the opportunities of subproblem reuse, we introduce two additional reuse mechanisms, including reusing existing solutions for better propagating the lowerbound and reusing solutions of the subproblems with fewer input-output constraints to solve subproblems with more constraints.

To sum up, this paper makes the following main contributions:

- A novel framework *MaxFlash* that combines dynamic programming and structural probability for efficiently solving PBE tasks. *MaxFlash* follows *PROSE* to use witness functions for dividing synthesis problems and can utilize existing witness functions in many applications of *PROSE*. In particular, *MaxFlash* includes
  - a novel subproblem definition that allows the local search of a program fragment to be aware of global structural probability while still allowing subproblem reuse,
  - a search algorithm that integrates iteratively deepening search and branch-and-bound, and
  - two additional reuse mechanisms to further boost subproblem reuse.
- An evaluation on a set of string manipulation and matrix transformation problems showing that *MaxFlash* has $\times 4.107 - \times 2080$ speed-ups against existing state-of-the-art solvers.

2 OVERVIEW

In this section, we introduce the basic idea of our approach via a motivating example, which will be discussed throughout this paper. In this example, we focus on synthesizing a program from a small domain-specific language L^{ex}:

Start symbol	S	$\rightarrow$	$N_{\mathbb{S}} \mid N_{\mathbb{Z}}$
String expr	$N_{\mathbb{S}}$	$\rightarrow$	Parameters $\mid (+\ N_{\mathbb{S}}\ N_{\mathbb{S}})$
		$\mid$	$(\textsc{CharAt}\ N_{\mathbb{S}}\ N_{\mathbb{Z}}) \mid$ '`.`'
Integer value	$N_{\mathbb{Z}}$	$\rightarrow$	$0 \mid 1$

Now, given an input-output example (`'John'`, `'Jonathan'`) $\rightarrow$ `'J.Jonathan'`, our goal is to find a program P in L^{ex} that outputs '`J.Jonathan`' when the input is ('`John`', '`Jonathan`'). Under these constraints, one valid program is:

$$(+\ \ (\textsc{CharAt}\ \ FS\ \ 0)\ \ (+\ \ \text{`}\texttt{.}\text{'}\ \ LS))$$

where FS and LS represent the two input strings respectively.

To start, we briefly introduce the dynamic-programming algorithm used in *PROSE*, as shown in Figure 1. In *PROSE*, a subproblem is to synthesize a program returning a specific value. *PROSE* uses a memoization search to reuse the subproblems.

(1) In the given example, to synthesize a program outputting '`J.Jonathan`', the synthesizer finds possibilities to divide this problem into subproblems. Each such possibility is denoted as a *scheme*. The synthesizer finds schemes by (1) enumerating possible syntactic forms: a parameter, a single constant, $(+\ N_{\mathbb{S}}\ N_{\mathbb{S}})$, and $(\textsc{CharAt}\ N_{\mathbb{S}}\ N_{\mathbb{Z}})$, and (2) enumerating possible outputs that the subprograms in the forms could produce. For example, when the form is $(+\ N_{\mathbb{S}}\ N_{\mathbb{S}})$, there are 9 possibilities: ('`J`', '`.Jonathan`'), ('`J.`', '`Jonathan`'), $\cdots$, ('`J.Jonatha`', '`n`').
(2) The synthesizer enumerates among schemes. Suppose the current scheme is ('`J.`', '`Jonathan`'). Then the synthesizer turns to synthesize a program P_1 that outputs `'J.'` and a program P_2 that outputs `'Jonathan'` . If both P_1, P_2 are found, $(+\ P_1\ P_2)$ will be a valid program that outputs '`J.Jonathan`' on input ('`John`', '`Jonathan`'). Tasks of synthesizing P_1 and P_2 keep the same form as the original task, and thus can be solved by recursively invoking the synthesizer.

As we can see from Figure 1, the subproblem whose target output is `'Jonathan'` is invoked twice but only searched once, through the memoization mechanism.

Such a search process is blind: the synthesis algorithm may explore a lot of poor subproblems before finding a valid program. To improve this, *MaxFlash* utilizes the structural probability of programs. Imagine, if we let human programmers write this program, there may be some common preferences in their programs. These commonalities can be modeled as a probabilistic model and thus guide the search. For example, programmers may prefer writing $(+\ P_1\ (+\ P_2\ P_3))$ rather than $(+\ (+\ P_1\ P_2)\ P_3)$ to make the first argument of + simpler. This pattern could be modeled as: when the whole program has the form $(+\ P_1\ P_2)$, the probability that the outmost operator of P_1 is + would be low.

Table 1 shows a potential probabilistic model $\mathcal{P}^{ex}$ for modeling structural probability, which describes the probability for a symbol to appear as a certain child of another symbol in AST. In the table, the rows represent the parent symbol and the index of the child, while the columns represent the child symbol. For example, the grid at row $(+, 1)$ and column FS represents the probability for FS to be used as the first child of +. Given model $\mathcal{P}^{ex}$, the probability of a program can be obtained by multiplying the probabilities of each part. For example, the probability of $(+\ (\textsc{CharAt}\ FS\ 0)\ (+\ \text{`}\texttt{.}\text{'}\ LS))$ is 0.01 under model $\mathcal{P}^{ex}$.

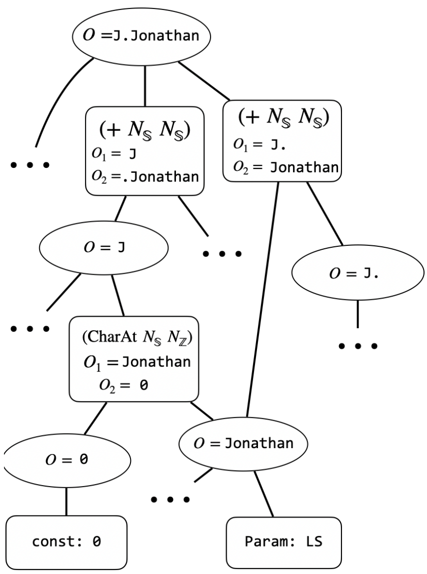


Fig. 1. Decision procedure of *PROSE*

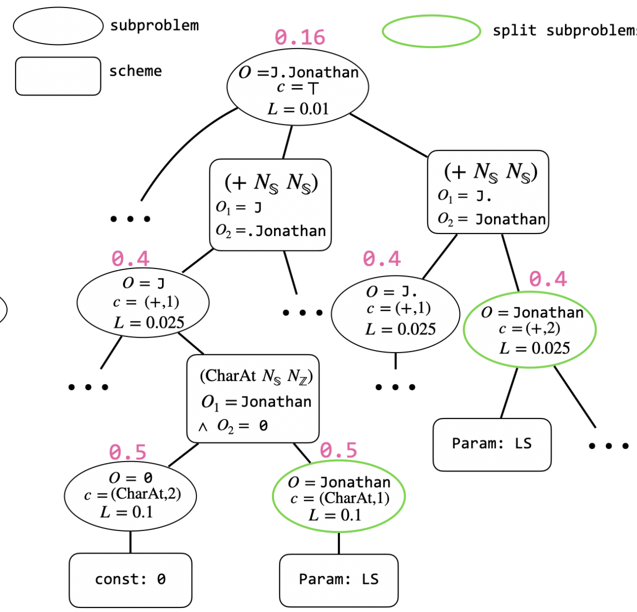


Fig. 2. Decision procedure of *MaxFlash*

Table 1. A conrete statistical model $\mathcal{P}^{ex}$

	CharAt	+	FS	LS	'.'	0	1
Start	0	1	0	0	0	0	0
(+, 1)	0.5	0.01	0.09	0	0.4	0	0
(+, 2)	0.05	0.5	0.05	0.4	0	0	0
(CharAt, 1)	0.05	0.05	0.5	0.4	0	0	0
(CharAt, 2)	0	0	0	0	0	0.5	0.5

This model reflects the globality of probability calculation: when calculating the probability of child symbol, we need to refer to its context: the parent symbol and the index. However, the subproblem definition in *PROSE* is local: when solving a subproblem, we do not concern its parent symbol. For example, in Figure 1, subproblem O = `Jonnathan` is used twice with different contexts: (CharAt, 1) and (+, 2). As a result, $\mathcal{P}^{ex}$ cannot be used in *PROSE* directly.

Such a conflict between the globality of structural probability and the locality of subproblems is the first challenge we met. To solve it, *MaxFlash* modifies the dynamic-programming algorithm in *PROSE* by further involving two parameters: the context of the subproblem and a probability lower bound. The decision procedure after involving these two parameters is shown in Figure 2.

Context. The context of a subproblem, denoted as c, is equal to its ancestral information required for our probabilistic model. After involving it, the local subproblem could partially access global information, and thus a large family of prediction models becomes applicable. For example, according to $\mathcal{P}^{ex}$, the context of a subprogram should be the symbol of its parent vertex and its index. With this definition, we attach the context to each subproblem in Figure 2. After that, subproblem O = `Jonnathan` in Figure 1 is split into two different subproblems, marked green in Figure 2, and thus the algorithm could correctly use the probabilities in $\mathcal{P}^{ex}$.

Probability Lowerbound. The probability lowerbound to a subproblem, denoted as L, is a requirement for the probability of the result. It means a valid program with a probability larger than L is required to produce a globally probable program. Each time when *MaxFlash* recurses into a subproblem, the lowerbound will also be propagated into it. In this way, the search is guided to consider only probable programs.

As mentioned before, *MaxFlash* utilizes iteratively deepening search: it starts with a high lowerbound to focus on the program with the highest probability. If no solution is found, it lowers the lowerbound by a constant value to allow more programs. *MaxFlash* repeats this procedure until a solution is found. *MaxFlash* also utilizes branch-and-bound: The lowerbounds are propagated among subproblems and are used to prune off improbable search branches. The propagation mechanism in *MaxFlash* is built upon a heuristic function, which overestimates the probability of the most probable valid program of each subproblem.

Here, we use a simple example to show how the probability lowerbounds propagate among subproblems and how it helps *MaxFlash* to prune off search branches. To begin with, we introduce a simple heuristic function: the heuristic value of a subproblem is defined as the probability of the most probable program to this subproblem under the context, i.e., the input-output constraints are ignored. We tag the heuristic value above every subproblem in Figure 2 as red numbers. Now, consider the procedure of synthesizing a program P with form $(+\ N_{\mathbb{S}}\ N_{\mathbb{S}})$ and lowerbound $L = 0.01$.

(1) While synthesizing the first argument, suppose its program is P_1, the lowerbound L could be raised to $\frac{0.01}{0.4} = 0.025$. This is because the probability of P is the product of those of its two arguments, and the probability of the second argument is no more than its heuristic value, 0.4.
(2) While enumerating the possible symbols for the root node of P_1, + can be skipped since its probability is lower than L. In this way, a large number of possible subproblems are pruned off.
(3) Suppose the current form of P_1 is $(\textsc{CharAt}\ N_{\mathbb{S}}\ N_{\mathbb{Z}})$. The synthesizer will then synthesize the first argument, denoted as $P_{1,1}$. At this time, L could be raised again to $\frac{0.025}{0.5\times 0.5} = 0.1$, which divides the probability that the form of P_1 is $\textsc{CharAt}$, 0.5, and the heuristic value of the second argument, 0.5. With this limit, the choices of $P_{1,1}$ remains only two possibilities: FS and LS.

However, after involving the probability lowerbound, the second major challenge emerges. Since the probability lowerbound is a real number and is integrated into subproblems, visiting the same subproblem twice becomes almost impossible. It turns out that the traditional reuse mechanism in dynamic programming becomes ineffective. To solve this problem, one intuitive idea is to reuse results between subproblems which only differ on the lowerbounds. However, this reuse is still very limited: we can reuse a solution only when the previously synthesized program has a probability higher than the new lowerbound, otherwise, we cannot say anything about the new problem.

To enable reusing a previous solution with a different lowerbound, *MaxFlash* modifies the search goal. It regards the synthesis problem as an optimization problem, which means it would not only produce a valid program but also find the one with the largest probability. In this way, the probability lowerbound can be ignored while reusing: if the previously synthesized program has a probability higher than the lowerbound, we return this program, otherwise, we report a failure.

Besides, we design two reuse mechanisms which can further reuse results and even reuse between subproblems with different input-output constraints. Here we just introduce the intuitions behind these two mechanisms. A detailed discussion can be found in Section 4.

(1) **Reusing through the heuristic function**: As many existing applications of branch-and-bound, *MaxFlash* will automatically refine the heuristic function using the newly obtained information during the synthesis. For example, if *MaxFlash* fails to synthesize a program for subproblem S under lowerbound L, the heuristic value of S will be improved to L. Moreover, *MaxFlash* takes advantage of a special relationship between subproblems in program synthesis,

and thus utilize the heuristic value of subproblems with fewer examples. In this way, the heuristic function becomes more and more accurate and thus its effect on pruning off useless search branches is constantly improved during synthesis.

(2) **Reusing results with fewer examples**: The time cost for the dynamic-programming algorithm grows dramatically when the number of examples increases since there are more different input-output constraints, and thus more different subproblems. Therefore *MaxFlash* tries to reuse results with fewer examples. For example, to synthesize a program from two examples E_1 and E_2, *MaxFlash* will firstly synthesize a program satisfying E_1 and then check whether it also satisfies E_2. If true, *MaxFlash* will return this program as the answer directly.

3 COMBINING DYNAMIC PROGRAMMING AND STRUCTURAL PROBABILITY

The core of *MaxFlash* is a combination of dynamic programming and structural probability. *MaxFlash* uses a series of methods to use structural probability to guide the search procedure of dynamic programming. In this section, we will introduce them step by step: Subsection 3.1 firstly introduces a basic dynamic-programming-based synthesizer. Then, the next four subsections 3.2, 3.3, 3.4, and 3.5 integrate contexts, iteratively deepening search and branch-and-bound, heuristic functions, and CEGIS framework into the dynamic-programming-based synthesizer in order.

We begin with a brief introduction to *Programming by Example (PBE)*. A PBE task is usually defined over a *Domain-Specific Language (DSL)*. Throughout this paper, we assume the syntax of a DSL is described by a context-free grammar $G = \langle N, \Sigma, s_0, R\rangle$ where N is a set of nonterminal symbols, Σ is a set of terminal symbols, s_0 represents the start symbol and R is the set of production rules. For simplicity, we further assume all production rules are in the form of $(s, f(x_1, \dots, x_k))$ which represents nonterminal symbol s can be expanded to a function using operator f and taking symbols $x_1, \dots, x_k$ as parameters. Specially, all production rules of constants and variables are regarded as functions with no parameter.

The task of PBE is to synthesize a program consistent with some given examples, from a given DSL. Different from most traditional synthesis approaches like *FlashFill*, *MaxFlash* regards a PBE task as an optimization problem: *MaxFlash* requires a *TopDown Prediction Model*, a special probabilistic model based on structural probability which will be introduced in Subsection 3.2, and always synthesizes the most probable program among all programs consistent with input-output examples.

3.1 Basic: A Dynamic-Programming-Based Synthesizer

The key idea of dynamic-programming is to divide the target problem into subproblems. Therefore, we start by defining the subproblems used in this subsection.

Definition 3.1 (PBE Subproblem). Given a grammar G, a PBE subproblem is a pair $(s, \mathbb{A})$, where: (1) s is a non-terminal symbol in G; (2) $\mathbb{A}$ is a list of input-output constraints $\{I_i \to O_i\}_{i=1}^{n}$, where I_i is an assignment to variables and O_i is a set of valid outputs.

The goal of a PBE subproblem $(s, \mathbb{A})$ is to find a valid program or determine there is no valid program. A valid program to PBE subproblem $(s, \mathbb{A})$ is a program p which is expanded from nonterminal s in G and is consistent with all input-output constraints in $\mathbb{A}$, i.e., $\forall (I_i, O_i) \in \mathbb{A}$, the output of p on I_i is a member of set O_i. i.e. $p(I_i) \in O_i$.

Algorithm 1 describes a basic dynamic-programming-based synthesizer. There are two functions used in Algorithm 1:

- `ValidProgram(`*problem*`)` always finds a valid program to a given PBE subproblem *problem*.
- `GetAllSchemes(`*problem*`)` returns a set of schemes of dividing *problem*. Each scheme is comprised of a grammar rule *form*, and k subproblems, where k is the number of arguments used in *form*. A scheme represents a way to divide *problem* into subproblems. `GetAllSchemes(·)` is implemented

Algorithm 1: A basic synthesizer based on dynamic programming

Input: A PBE task specified by a grammar $G = \langle N, \Sigma, s_0, R\rangle$ and a set of input-output examples E.
Output: A valid program $program^*$, or $\top$ for no valid program.

1 $memoTable \leftarrow \{\}$;
2 **Function** `ValidProgram`$(problem = (s, \mathbb{A}))$:
3   **if** $problem \in memoTable$ **then return** $memoTable[problem]$;
4   **for** $(form, subproblem_1, \dots, subproblem_k) \in$ `GetAllSchemes`$(problem)$ **do**
5         $subprogram_i \leftarrow$ `ValidProgram`$(subproblem_i)$ for each $i \in [1, k]$;
6   **if** $\forall i \in [1, k], subprogram_i \neq \top$ **then**
7             $result \leftarrow$ `ConstructProgram`$(form, subprogram_1, \dots, subprogram_k)$;
8             $memoTable[problem] \leftarrow result$;
9   **return** $result$;
10   **end**
11     $memoTable[problem] \leftarrow \top$;
12   **return** $\top$;
13 **return** `ValidProgram`$((s_0, E))$;

in the same way as *PROSE* [Polozov and Gulwani 2015]. For rule *form* and each input-output constraint $I_i \rightarrow O_i$ in *problem*, `GetAllSchemes(·)` uses pre-defined rules, named as witness functions, to obtain input-output constraints for subproblems. Function `GetAllSchemes(·)` enumerates on all combinations of resulting input-output constraints for different examples and thus gets a set of possible schemes.

Algorithm 1 maintains a memoization table *memoTable* which records valid programs for all visited subproblems. Each time when `ValidProgram(·)` receives a repetitive subproblem, it accesses *memoTable* and directly returns a valid program (Line 3). Otherwise, `ValidProgram(·)` enumerates on possible schemes of dividing *problem* into subproblems (Line 4). `ValidProgram(·)` recursively finds valid subprograms for subproblems (Line 5). If for every subproblem, a subprogram is found, `ValidProgram(·)` will merge these subprograms into a valid program to *problem* (Line 7) and will return this program (Lines $8-9$). Otherwise, if no valid program is found after dealing with all schemes, `ValidProgram(·)` will return $\top$ (Lines $11-12$). According to Definition 3.1, `ValidProgram`$((s_0, E))$ must be a valid program for the given PBE task (Line 13).

3.2 Integrating Contexts

As mentioned before, *MaxFlash* introduces a context to the definition of the subproblem to allow the calculation of structural probability and turns the subproblem into an optimization problem that searches for the most probable program. To efficiently maintain the context while dynamic programming, *MaxFlash* assumes the context captures only the information from the ancestors but does not include the information from siblings. In this way, when searching for the optimal solution for a subproblem, we can directly search for the optimal solution for each of its children independently, forming an efficient dynamic programming structure.

We use a topdown context model to represent the context information extracted from the ancestors. For each vertex, the context model constructs its context from the context of its parent, the grammar rule used on the parent, and the index of this vertex among all siblings.

Definition 3.2 (TopDown Context Model). Given a set of grammar rules R, a topdown context model $\mathcal{M}$ is a triple $\langle C, c_0, \tau\rangle$ where C represents a set of abstracted context, $c_0 \in C$ represents the start context, and τ is a transition function of type $(C \times R \times \mathbb{Z}^+) \rightarrow C$.

Given topdown context model $\langle C, c_0, \tau \rangle$ and program p, topdown context $C_p(v) \in C$ of vertex v on the AST of program p is defined as:

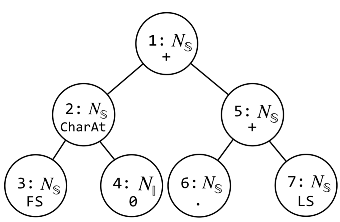
$$C_p(v) = \begin{cases} c_0 & v \text{ is the root} \\ \tau(C_p(f), r_f, x) & \text{Otherwise} \end{cases}$$

where f represents the parent vertex of v, r_f represents the rule applied to f, and x is the index of v among all children of f from left to right.

Example 3.3. Contexts used in Section 2 can be represented by a topdown context model $\mathcal{M}^{ex} = \langle C^{ex}, c_0^{ex}, \tau \rangle$. This model considers the information from the direct parent, i.e., the context of the parent is ignored. Model $\mathcal{M}^{ex}$ is specified over the rule set of L^{ex} (the DSL used in Section 2), and its contents are:

$$C^{ex} = \{\top\} \cup (R \times \mathbb{Z}^+) \quad c_0^{ex} = \top \quad \tau^{ex}(c, r, x) = (r, x)$$

This model captures the information for each node in AST about the rule applied to its parent and its index. To show this point, we apply this model and calculate the context for some vertices in the AST of the valid program discussed in Section 2. The results are shown in Figure 3.



Node	Parent	Index	$C_p(\cdot)$
1	/	/	$\top$
2	1	1	$(N_\mathbb{S} \to (+\ N_\mathbb{S}\ N_\mathbb{S}), 1)$
5	1	2	$(N_\mathbb{S} \to (+\ N_\mathbb{S}\ N_\mathbb{S}), 2)$
7	5	2	$(N_\mathbb{S} \to (+\ N_\mathbb{S}\ N_\mathbb{S}), 2)$

Fig. 3. An example of topdown contexts. The left figure shows the AST of program (+ (CharAt *FS* 0) (+ '.' *LS*)), and the right table lists the topdown contexts for some vertices of this AST.

Definition 3.4 (TopDown Prediction Model). Given a set of grammar rule R, a topdown prediction model (abbreviated as TPM) $\mathcal{P}$ is a topdown context model $\langle C, c_0, \tau \rangle$ combined with a function $\varphi : C \times R \mapsto \mathbb{R}^{\geq 0}$ which satisfies $\forall c \in C, \sum_{r \in R} \varphi(c, r) = 1$.

TPM uses $\varphi(c, r)$ to represent the probability for rule r to be applied to an AST vertex under context c. The probability of a program is equal to the product of the probabilities of all vertices on its AST. Throughout this paper, we use $\mathcal{P}_c[p]$ to denote the log-probability of program p under prediction model $\mathcal{P}$ and context c, and use this value to represent the structural probability.

Example 3.5. Continued with Example 3.3, model $\mathcal{P}^{ex}$ defined in Section 2 is exactly a TPM over $\mathcal{M}^{ex}$ and L^{ex}. According to $\mathcal{P}^{ex}$:

$$\mathcal{P}^{ex}_\top[(+\ (\text{CharAt } FS\ 0)\ \ (+\ `.\text{'}\ LS))] = \log 0.01 \approx -4.61$$
$$\mathcal{P}^{ex}_\top[(+\ (+\ (\text{CharAt } FS\ 0)\ `.\text{'})\ \ LS)] = \log 0 = -\infty$$

TPM keeps three important properties, making it suitable for dynamic programming:

(1) Under a topdown context model, sibling problems are independent from each other, allowing them to be searched independently.
(2) The calculation is local: The context can be transmitted only from the parent vertex, making the local transition of dynamic programming possible.
(3) The number of different contexts is limited (i.e., $|C|$) so that adding the context into subproblems would not significantly increase the number of different subproblems.

Algorithm 2: A dynamic-programming based synthesizer for optimization subproblems.

Input: A PBE task specified by a grammar $G = \langle N, \Sigma, s_0, R \rangle$, a set of input-output examples E and a TPM $\mathcal{P} = \langle C, c_0, \tau, \varphi \rangle$.

Output: A valid program $program^*$ with the highest probability according to $\mathcal{P}$.

1 $memoTable \leftarrow \{\}$;
2 **Function** `OptimalProgram`($problem = (s, \mathbb{A}, c)$):
3   **if** $problem \in memoTable$ **then return** $memoTable[problem]$;
4     $(bestProgram, bestProbability) \leftarrow (\top, -\infty)$;
5   **for** $(form, subproblem_1, \dots, subproblem_k) \in$ `GetAllSchemes`($problem$) **do**
6         $subprogram_i \leftarrow$ `OptimalProgram`($subproblem_i$) for each $i \in [1, k]$;
7   **if** $\forall i \in [1, k], subprogram_i \neq \top$ **then**
8             $candidate \leftarrow$ `ConstructProgram`($form, subprogram_1, \dots, subprogram_k$);
9   **if** $\mathcal{P}[candidate]_c > bestProbability$ **then**
10                 $(bestProgram, bestProbability) \leftarrow (candidate, \mathcal{P}_c[candidate])$;
11   **end**
12     $memoTable[problem] \leftarrow bestProgram$;
13   **return** $bestProgram$;
14 **return** `OptimalProgram`((s_0, E, c_0));

Now we are ready to define the subproblems used in *MaxFlash*. To distinguish from PBE subproblems, we name the new subproblems as *Optimization Subproblem*, abbreviated as *Subproblem*.

Definition 3.6 ((Optimization) Subproblem). Given a grammar G and a TPM $\mathcal{P}$, an optimization subproblem is a triple $(s, \mathbb{A}, c)$, where: (1) s is a non-terminal symbol in G. (2) $\mathbb{A}$ is a list of input-output constraints $\{I_i \to O_i\}_{i=1}^n$, where I_i is an assignment to variables and O_i is a set of valid outputs. (3) c is an abstracted context in $\mathcal{P}$.

The optimal program to subproblem $(s, \mathbb{A}, c)$ is a program p^* satisfying (1) **validity**: p^* can be expanded from symbol s in G and is consistent with all input-output constraints, i.e., $\forall (I_i, O_i) \in \mathbb{A}$, the output of p^* on I_i is a member of the set O_i. (2) **optimality**: for any valid program p to subproblem $(s, \mathbb{A}, c)$, the probability of p^* is always no smaller than p, i.e., $\mathcal{P}[p^*] \geq \mathcal{P}[p]$. If there is no valid program, the optimal program of $(s, \mathbb{A}, c)$ is defined as $\top$.

After determining the definition of subproblems, the way of extending the dynamic-programming algorithm becomes clear. Algorithm 2 describes a synthesizer for *Optimization Subproblems*. Algorithm 2 is almost the same as Algorithm 1, except for two differences:

- The main synthesis algorithm changes to `OptimalProgram(·)`, which always finds the optimal program for a given subproblem *problem* (Line 2).
- For each scheme, `OptimalProgram(·)` recursively finds optimal subprograms for subproblems (Line 6). By the optimality of `OptimalProgram(·)`, the program constructed from these subprograms must be the most probable program to this scheme (Line 8). `OptimalProgram(·)` will use this program to update results and ignore all other programs to this scheme (Lines 7 – 10) and will return the best program among all schemes as the result (Lines 12 – 13).

3.3 Integrating Iteratively Deepening Search and Branch-and-Bound

Algorithm 2 searches for the most probable but does not use structural probability to guide the search, which is our goal. To further utilize structural probability, we integrate iteratively deepening search [Korf 1985] and branch-and-bound [Land and Doig 1960], two efficient search strategies for

Algorithm 3: A synthesizer integrating iteratively deepening search

Input: A PBE task specified by a grammar $G = \langle N, \Sigma, s_0, R \rangle$, a set of input-output examples E, a TPM $\mathcal{P} = \langle C, c_0, \tau, \varphi \rangle$ and a step size *size*.

Output: A valid program $program^*$ with the highest probability according to $\mathcal{P}$.

```
1  memoTable ← {};
2  Function GetBound(problem = (s, 𝔸, c), lowerbound, form, scheme, i):
3      return lowerbound − log φ(c, form);
4  Function OptimalProgram(problem = (s, 𝔸, c), lowerbound):
5      if problem ∈ memoTable then
6          bestProgram ← memoTable[problem];
7          if 𝒫_c[bestProgram] ≥ lowerbound then return bestProgram else return ⊤;
8      end
9      if lowerbound > 0 then return ⊤;
10     (bestProgram, bestProbability) ← (⊤, −∞);
11     for scheme = (form, subproblem_1, ..., subproblem_k) ∈ GetAllSchemes(problem) do
12         for i ∈ [1, k] do
13             subBound_i ← GetBound(problem, lowerbound, form, scheme, i) for each i ∈ [1, k];
14             subprogram_i ← OptimalProgram(subproblem_i, subBound_i) for each i ∈ [1, k];
15         end
16         Update bestProgram. This part is the same as Lines 7 − 10 in Algorithm 2;
17         lowerbound ← max(lowerbound, bestProbability);
18     end
19     if bestProgram ≠ ⊤ then memoTable[problem] ← bestProgram;
20     if bestProbability ≥ lowerBound then return bestProgram else Return ⊤;
21 lowerbound ← 0;
22 while OptimalProgram((s_0, E, c_0), lowerbound) = ⊤ do
23     lowerbound ← lowerbound − step ;
24 end
25 return OptimalProgram((s_0, E, c_0), lowerbound);
```

finding an optimal solution, into the dynamic-programming-based synthesizer. The pseudo-code of the new synthesizer is shown in Algorithm 3.

We now elaborate on the differences between Algorithm 3 and Algorithm 2. The first difference is that the signature of `OptimalProgram(·)` changes: Besides a subproblem, `OptimalProgram(·)` in Algorithm 3 further requires a real number *lowerbound*. `OptimalProgram(`*problem, lowerbound*`)` searches for the optimal program only among valid programs with log-probabilities no smaller than *lowerbound*, and it will return $\top$ if the log-probability of the optimal program is smaller than *lowerbound*. In this way, the search space of `OptimalProgram(·)` is greatly reduced.

The second difference is that branch-and-bound is integrated into Algorithm 3: The lowerbound is propagated among subproblems and is used to prune off improbable search branches.

- Propagating. Given a form and a scheme, `GetBound(·)` propagates the lowerbound of *problem* to its subproblems (Line 13). In Algorithm 3, `GetBound(·)` is implemented conservatively: it only subtracts the probability for *form* to occur under context c, but ignores the probability of other subproblems. Besides `GetBound(·)`, the lowerbound will also be updated once a better program is found (Line 17), since we only need to focus on programs better than *bestProgram*.

- Pruning. The pruning method in Algorithm 3 is also conservative: Since the log-probability of a program cannot be larger than 0, it is safe to prune off a search branch when the lowerbound is greater than 0 (Line 9).

Like many applications of branch-and-bound, the efficiency of these two parts can be improved by involving a proper heuristic function. We will detailedly discuss this point in Section 3.4.

The next difference is that Algorithm 3 takes iteratively deepening search as the outer framework (Lines $21-25$). Algorithm 3 maintains a global lowerbound *lowerbound* (Line 21) and synthesizes in turns (Lines $22-24$). In each turn, it invokes `OptimalProgram(·)` to search for the target program among all programs with log-probabilities no smaller than *lowerbound* (Line 22). The lowerbound will be relaxed in turns until the target program is found (Line 23).

The last difference is on the reuse mechanism (Lines $5-8, 19$). In Algorithm 2, each subproblem is dealt with at most once: whether the first result is a program or $\top$, this result can be always reused when visiting the same subproblem again. However, after involving the lower bound, the reuse mechanism becomes more complex. Since `OptimalProgram(·)` only considers programs with log-probabilities at least *lowerbound*, we have to ignore all failed results since we cannot distinguish the case of no valid program and the case that the optimal program is not probable enough. We shall show how to use a heuristic function to indirectly reuse all failed results in Subsection 4.1.

3.4 Integrating Heuristic Function

To speed up branch-and-bound, heuristic function is a commonly used technique, as it can effectively improve the performance of both the propagating method and the pruning method. Therefore, we further integrate a heuristic function into Algorithm 3.

We begin with the definition of heuristic functions. For convenience, we define *bestProb*(*problem*) as the log-probability of the optimal program of problem *problem*. Specially, when the optimal program is $\top$, *bestProb*(*problem*) is defined as $-\infty$. The heuristic function used in *MaxFlash* is an over-approximation to *bestProb* for all possible subproblems.

Definition 3.7 (Heuristic Function). Function *heuristic* which maps subproblems to real numbers is a heuristic function if and only if for any subproblem *problem*, *heuristic*(*problem*) is no smaller than *bestProb*(*problem*).

In this subsection, we first assume the existence of a black-box heuristic function *heuristic*, and show how to use this heuristic function to speed up Algorithm 3. After that, we will implement a basic heuristic function $heuristic_0$. In Subsection 4.1, we will elaborate on the heuristic function used in *MaxFlash*, which is improved from $heuristic_0$.

Algorithm 4 shows a new version of `OptimalProgram(·)` which utilize a heuristic function while propagating lowerbounds and pruning off improbable search branches:

- Propagating. Algorithm 4 uses a new implementation of `GetBound(·)`(Lines $2-4$) to propagate lowerbounds. Comparing with Algorithm 3, `GetBound(·)` here considers the probability of sibling subprograms by subtracting the heuristic values of other subproblems, since $heuristic(subproblem_j)$ is an upper bound for the log-probability of the jth subprogram.
- Pruning. Algorithm 3 will directly return $\top$ once the heuristic value of *problem* is smaller than *lowerbound* (Line 10) (while 0 is used in Algorithm 3). Such an early termination is safe because the heuristic value is guaranteed to be no smaller than the log-probability of the best program.

Besides these two aspects, the heuristic function is also used to organize the order of enumerating schemes. Algorithm 4 enumerates schemes in the decreasing order of $\log \varphi(c, form_i) + \sum_{subp \in subproblems_i} heuristic(subp)$, i.e., the log-probability for the from to occur under context c plus the sum of heuristic values of subproblems. Clearly, this sum is an upper bound of the log-probability

Algorithm 4: A synthesizer integrating iteratively deepening search and a heuristic function

Input: A PBE task specified by a grammar $G = \langle N, \Sigma, s_0, R\rangle$, a set of input-output examples E, a TPM $\mathcal{P} = \langle C, c_0, \tau, \varphi\rangle$, a step size *size*.

Output: A valid program $program^*$ with the highest probability according to $\mathcal{P}$.

$memoTable \leftarrow \{\}$;

Function `GetBound`($problem = (s, \mathbb{A}, c)$, *lowerbound*, *form*, *scheme*, i)**:**

 $(subproblem_1, \dots, subproblem_k) \leftarrow scheme$;

 Return $lowerbound - \log \varphi(c, form) - \sum_{j \in [1,k], j \neq i} heuristic(subproblem_j)$;

Function `SortSchemes`($problem = (s, \mathbb{A}, c)$, $allSchemes = \{(form_i, subproblems_i)\}_{i=1}^{t}$)**:**

 Sort *allSchemes* in the decreasing order of $\log \varphi(c, form_i) + \sum_{subp \in subproblems_i} heuristic(subp)$.

 Return *allSchemes* ;

Function `OptimalProgram`($problem = (s, \mathbb{A}, c)$, *lowerbound*)**:**

 Reuse results in *memoTable*. This part is the same as Lines 5 – 8 in Algorithm 3.

 if $heurisitc(problem) < lowerbound$ **then** Return $\top$;

 $(bestProgram, bestProbability) \leftarrow (\top, -\infty)$;

 for $scheme = (form, subproblems) \in$ `SortSchemes`(*problem*, `GetAllSchemes`(*problem*)) **do**

  Deal with *scheme*. This part is the same with Lines 12 – 17 in Algorithm 3.

 end

 Store and return the result. This part is the same as Lines 19 – 20 in Algorithm 3.

The outer framework. This part is the same as Lines 21 – 25 in Algorithm 3.

of valid programs to $scheme_i$. Intuitively, the larger the upper bound is, the more probable finding a valid program from $scheme_i$ will be. Therefore we enumerate schemes in this order.

Now we introduce a basic heuristic function $heuristic_0$. For a subproblem $problem = (s, \mathbb{A}, c)$, $heuristic_0(problem)$ is defined as $bestProb((s, \emptyset, c))$, i.e., the log-probability of the most probable program expanding from non-terminal symbol s under context c.

Example 3.8. $heuristic_0$ is the logarithm of the heuristic function used in our motivating example. Figure 2 shows some examples of $heuristic_0$.

The validity of $heuristic_0$ is based on the following lemma:

LEMMA 3.9. *For any two subproblems* $problem_1 = (s, \mathbb{A}, c), problem_2 = (s, \mathbb{A}', c)$ *where* $\mathbb{A}' \subseteq \mathbb{A}$, *i.e., the input-output constraints in* $problem_1$ *is a subset of* $problem_2$, *we have that* $bestProb(problem_1)$ *is no larger than* $bestProb(problem_2)$.

PROOF. Let *program* be the optimal program for $problem_1$. Since $\mathbb{A}'$ is a subset of $\mathbb{A}$, *program* must be valid to $problem_2$. Therefore $bestProb(problem_1) = \mathcal{P}_c[program] \leq bestProb(problem_2)$ □

Since $\emptyset$ is a subset of any other set, $heuristic_0(problem)$ must be at least $bestProb(problem)$. In this way, we prove the validity of $heuristic_0$.

One advantage of $heuristic_0$ is that its value can be quickly obtained. Since $heuristic_0(\cdot)$ only relies on non-terminal symbol s and context c, the number of different heuristic values is at most $|N| \times |C|$, i.e., the number of nonterminal symbols times the number of contexts, which is a small number. In our implementation, all possible values of $heuristic_0$ are pre-calculated and cached for queries. An optimization algorithm proposed by Gallo et al. [1993] is used here, which could calculate all heuristic values in time complexity $O(|N||C| \log |C|)$.

Example 3.10. In Section 2, we have discussed Algorithm 4 with heuristic function $heuristic_0$. The workflow of Algorithm 4 on our motivating example is drawn as Figure 2.

Algorithm 5: Outer Framework of *MaxFlash*

Input: A TPM $\mathcal{P} = \langle C, c_0, \tau, \varphi \rangle$, a set $E = \{(I_i, O_i)\}_{i=1}^{n}$ of input-output examples and a step-size *step*.
Output: The target program *program**.

1 *counterExamples* $\leftarrow \emptyset$;
2 *lowerBound* $\leftarrow 0$;
3 **while** *True* **do**
4   **while** `OptimalProgram`(*counterExamples, lowerBound*) $= \top$ **do**
5   *lowerBound* $\leftarrow$ *lowerBound* $-$ *step*;
6   **end**
7   *program* $\leftarrow$ `OptimalProgram`(*counterExamples, lowerBound*);
8   *counterExample* $\leftarrow$ `GetCounterExample`(*program, E*);
9   **if** *counterExample* $= \top$ **then return** *program* ;
10   *counterExamples*.INSERT(*counterExample*);
11 **end**

3.5 Integrating CEGIS Framework

One shortage of dynamic-programming-based synthesizers is that their efficiencies are affected by the number of examples: since the input-output constraints of all examples are encoded into subproblems, the more examples are, the more subproblems will be, and thus the harder reusing results will be. A useful technique for this issue is CEGIS framework [Solar-Lezama et al. 2006], which allows the synthesis algorithm to consider only a subset of examples. In *MaxFlash*, we merge the outer framework of iteratively deepening search (Line $21 - 25$ in Algorithm 3) and CEGIS so that the lower bound is shared between different turns of CEGIS.

The outer framework of *MaxFlash* is shown in Algorithm 5, which is comprised of a double loop. The inner loop (Lines $4 - 6$) is the procedure of iteratively deepening search, and the outer one (Lines $3 - 11$) follows CEGIS framework. In the outer loop, a small set of examples *counterExamples* is maintained. In each turn, the inner loop finds a program *program* that is consistent with all examples in *counterExamples* (Line 4). After that, *MaxFlash* calls `GetCounterExample(·)` to verify whether *program* is consistent with all other examples in E (Line 8). If true, it will return *program* as the result (Line 9). Otherwise, it will add a counter-example to *counterExamples* (Line 12) and continue to synthesize for the new set of examples.

The trick in Algorithm 5 is that *lowerbound* is shared between different turns: by Lemma 3.9, the log-probability of the optimal program mustn't increase after adding new examples. Therefore, we could perform iteratively deepening search on the basis of previous CEGIS turns.

4 EXTRA REUSE MECHANISMS

The core of dynamic programming is a reuse mechanism for repetitive subproblems. In Section 3.3, we have discussed the basic reuse mechanism in *MaxFlash*, which reuses all results in which the optimal programs are found. This mechanism has two shortages:

- It only reuses success results and ignores all failed results. However, there is also a lot of useful information in failed results.
- It only reuses results for the same subproblem. However, Lemma 3.9 has shown that there are relations between subproblems with different input-output constraints.

Based on these two points, *MaxFlash* involves two novel reuse mechanisms: the first reuse mechanism updates the heuristic function according to obtained results and thus reuse the failed results

Algorithm 6: A synthesizer integrating iteratively deepening search and a heuristic function

Input: A PBE task specified by a grammar $G = \langle N, \Sigma, s_0, R\rangle$, a set of input-output examples E, a TPM $\mathcal{P} = \langle C, c_0, \tau, \varphi\rangle$, a step size *size*.

Output: A valid program $program^*$ with the highest probability according to $\mathcal{P}$.

1 $memoTable \leftarrow \{\}$;
2 $memoTable_h \leftarrow \{\}$;
3 **Function** `Heuristic`($problem = (s, \mathbb{A}, c)$)**:**
4     $heuristicValue \leftarrow heuristic_0(problem)$;
5   **for** $\mathbb{A}' \in$ `Prefix`($\mathbb{A}$) **do**
6   **if** $(s, \mathbb{A}', c) \in memoTable_h$ **then** $heuristicValue \leftarrow \min\left(heuristicValue, memoTable_h[(s, \mathbb{A}', c)]\right)$;
7   **end**
8   **return** $heuristicValue$;
9 **Function** `OptimalProgram`($problem = (s, \mathbb{A}, c), lowerbound$)**:**
10   Reuse results in *memoTable* and check the heuristic value. The same as Lines $9-10$ in Algorithm 4.
11   Solve subproblem *problem*. This part is the same as Lines $11-14$ in Algorithm 4.
12   **if** $bestProgram \neq \top$ **then**
13         $memoTable[problem] \leftarrow bestProgram$;
14         $memoTable_h[problem] \leftarrow bestProbability$;
15   **else**
16         $memoTable_h[problem] \leftarrow lowerbound$;
17   **end**
18   **if** $bestProbability \geq lowerBound$ **then return** *bestProgram* **else return** $\top$;
19 The outer framework. This part is the same as Algorithm 5.

indirectly; the second reuse mechanism utilizes the generality of synthesized programs and reuse the optimal programs with fewer input-output constraints.

4.1 Reusing Through the Heuristic Functions

In this subsection, we introduce the heuristic function *heuristic* used in *MaxFlash*, which is improved from $heuristic_0$, the basic heuristic function discussed in Section 3.4. Algorithm 6 shows a synthesizer with the full-version heuristic function, in which *heuristic* is implemented as `Heuristic(.)`.

The reuse mechanism through heuristic function can be divided into two parts. The first part follows a common idea among applications of branch-and-bound: Failed results can be used to update the heuristic function. Though failed results cannot be reused directly, it suggests that there is no valid program with log-probability at least *lowerbound*. Therefore if `OptimalProgram(·)` fails, *lowerbound* will be a safe heuristic value for *problem*. Similarly, for a successful result, *bestProbability* is safe, since it is exactly equal to $bestProb(problem)$. In Algorithm 6, all these values are stored in a new memoization table $memoTable_h$ (Lines 14, 16) and are used to improve $heuristic_0(problem)$.

The second part takes advantage of the relationship between subproblems established in Lemma 3.9, and reuses information from fewer constraints to more constraints (Lines $5-7$) according to the following two observations. Firstly, since *MaxFlash* adopts CEGIS framework, before solving subproblem $(s, \mathbb{A}, c)$, a large number of subproblems $(s, \mathbb{A}', c)$, where $\mathbb{A}'$ is a prefix of $\mathbb{A}$, have been dealt with. Secondly, according to Lemma 3.9, if $\mathbb{A}'$ is a prefix of $\mathbb{A}$, $bestProb((s, \mathbb{A}, c))$ must be at most $bestProb(s, \mathbb{A}', c)$. Therefore $memoTable_h[(s, \mathbb{A}', c)]$ is a valid heuristic value not only for $(s, \mathbb{A}', c)$ but also for $(s, \mathbb{A}, c)$. In this way, a tighter heuristic function is obtained:

$$heuristic((s, \mathbb{A}, c)) = \min(heuristic_0((s, \mathbb{A}, c)), \min_{\mathbb{A}' \text{ is a prefix of } \mathbb{A}} memoTable_h[(s, \mathbb{A}', c)])$$

Example 4.1. We now use two subproblems of the PBE task discussed in Section 2 to show the reuse mechanism discussed in this subsection. Consider the following two subproblems:

$$problem_1 = (N_{\mathbb{S}}, \{(\text{`}\texttt{John}\text{'}, \text{`}\texttt{Jonathan}\text{'}) \to \text{`}\texttt{J.Jonathan}\text{'}\}, \top)$$
$$problem_2 = (N_{\mathbb{S}}, \{(\text{`}\texttt{John}\text{'}, \text{`}\texttt{Jonathan}\text{'}) \to \text{`}\texttt{J.Jonathan}\text{'}, (\text{`}\texttt{John}\text{'}, \text{`}\texttt{Smith}\text{'}) \to \text{`}\texttt{J.Smith}\text{'})\}, \top)$$

Now, suppose `OptimalProgram(`$problem_1, -4$`)` and `OptimalProgram(`$problem_2, -3$`)` are invoked in order. Since the optimal program to $problem_1$ is (+ (CharAt *FS* 0) (+ '`.`' *LS*)) of which the log-probability is larger than -4, the first invocation fails. Therefore $memoTable_h[problem_1]$ is set to -4. During the second invocation, since the constraints of $problem_1$ is a prefix of the constraints of $problem_2$, `Heuristic(`$problem_2$`)` is evaluated to -4 which is smaller than the lowerbound -3. Therefore the second invocation will terminate with $\top$ immediately.

4.2 Reusing Results with Fewer Constraints

In this subsection, we turn to the last reuse mechanism, which also reuses results from subproblems with fewer constraints to subproblems with more constraints. This reuse mechanism is based on the following three observations. For any two subproblems $(s, \mathbb{A}, c), (s, \mathbb{A}', c)$ where $\mathbb{A}' \subset \mathbb{A}$:

(1) The first observation is a corollary to Lemma 3.9: If the optimal program to $(s, \mathbb{A}', c)$ is a valid program to $(s, \mathbb{A}, c)$, it must also be the optimal program to $(s, \mathbb{A}, c)$.
(2) The second observation is about the time cost: solving $(s, \mathbb{A}, c)$ is usually much more time-consuming than solving $(s, \mathbb{A}', c)$, since $(s, \mathbb{A}, c)$ considers more examples.
(3) The third observation is about the generality of an optimal program, which is inspired by CEGIS framework: The optimal program of $(s, \mathbb{A}', c)$ has a great chance to be valid for $(s, \mathbb{A}, c)$, since it is the most probable program under a prediction model based on structural probability.

Therefore, for a subproblem *problem* with multiple input-output constraints, *MaxFlash* will firstly solve another subproblem $problem_f$ with fewer constraints and checks whether the found program is valid for *problem*. If the found program is, the optimal program to *problem* is found immediately, and a lot of time will be saved. If it could not, there won't be too much wasted time since solving $problem_f$ is usually easier than solving *problem*.

The pseudo-code of this reuse mechanism is shown in Algorithm 7 (Lines $5 - 14$). The only detail worth discussing is that *MaxFlash* takes the result of removing the last constraint from *problem* as $problem_f$ (Lines $6 - 7$). Such a choice has two advantages: (1) The constraint set of $problem_f$ is one of the largest proper subsets of the constraint set of *problem*. Since almost all constraints are reserved, the optimal program to $problem_f$ is likely to be valid to *problem*. (2) According to CEGIS framework, a lot of results on the first $|\mathbb{A}| - 1$ examples are cached in the previous synthesis turn. Therefore the time cost of solving $problem_f$ should not be large.

Example 4.2. We continue to discuss subproblems $problem_1, problem_2$ introduced in Example 4.1. Consider the procedure of `OptimalProgram(`$problem_2, -5$`)`:

(1) At first, the last constraint of $problem_2$ is removed, and the result is $problem_1$ (Line 6).
(2) Then, `OptimalProgram(`$problem_1, -5$`)` is invoked. After finishing this invocation, the optimal program *candidate* = (+ (CharAt *FS* 0) (+ '`.`' *LS*)) to $problem_1$ is obtained (Line 7).
(3) Since *candidate* is a valid program to $problem_1$, it must also be the optimal program to $problem_2$. Therefore, `OptimalProgram(`$problem_2, -5$`)` returns *candidate* immediately (Lines $9 - 13$).

So far, we have illustrated all techniques and reuse mechanisms in *MaxFlash*. At the end of this section, we use a theorem to show the correctness of *MaxFlash*.

Algorithm 7: The pseudo code of *MaxFlash*.

Input: A PBE task specified by a grammar $G = \langle N, \Sigma, s_0, R \rangle$, a set of input-output examples E, a TPM $\mathcal{P} = \langle C, c_0, \tau, \varphi \rangle$, a step size *size*.
Output: A valid program $program^*$ with the highest probability according to $\mathcal{P}$.

```
memoTable ← {};
memoTable_h ← {};
Function OptimalProgram(problem = (s, 𝔸, c), lowerbound):
    Reuse results in memoTable and check the heuristic value. The same as Lines 9 − 10 in Algorithm 4.
    if |𝔸| > 1 then
        𝔸′ ← 𝔸.RemoveLast();
        candidate ← OptimalProgram((s, 𝔸′, c), lowerbound);
        if candidate = ⊤ then Return ⊤;
        if candidate is valid to problem then
            memoTable[problem] ← candidate ;
            memoTable_h[problem] ← 𝒫_c[candidate];
            return candidate ;
        end
    end
    Solve problem and update storages. This part is the same as Lines 11 − 18 in Algorithm 6.
The outer framework. This part is the same as Algorithm 5.
```

Theorem 4.3. *Given a PBE task and a topdown prediction model $\mathcal{P}$, let V be the set of all valid programs, the program $program^*$ synthesized by MaxFlash always satisfies (1)* ***validity:*** *$program^* \in V$, (2)* ***optimality:*** *$\forall program \in V, \mathcal{P}[program] \leq \mathcal{P}[program^*]$.*

Proof. The proof is in the appendix, which is available at https://github.com/jiry17/MaxFlash. □

5 IMPLEMENTATION

We implement *MaxFlash* over the string manipulation domain and matrix transformation domain. In this section, we shall briefly explain the details of our implementation. Our implementation and all experimental data are available at https://github.com/jiry17/MaxFlash.

When implementing the iteratively deepening search, we set the step size as 3, i.e., the global log-probability lowerbound will be relaxed by 3 after each iteration.

In our implementation, we take AST d-gram model as the topdown context model, i.e., the topdown context of a vertex is comprised of the rules applied to its first d ancestors and the positional relationship between them. Formally, we define a group of topdown context models $\mathcal{M}_d = \{C_d, c_{0,d}, \tau_d\}$, where:

- C_d contains all d-length sequences of which each element is either $\top$ or a pair in $R \times \mathbb{Z}^+$, representing the rule applied to a vertex and the index of the next vertex, i.e., $C_d = (\{\top\} \cup (R \times \mathbb{Z}^+))^d$.
- $c_{0,d}$ is the sequence containing only $\top$, i.e., $c_{0,d} = (\top)^d$.
- $\tau_d(c, r, i) = (e_2, \dots, e_d, (r, i))$, where e_i represents the ith element in context c.

In our implementation, we set d to 1 by default. At this time, the context model $\mathcal{M}_1$ is exactly the context model used in our motivating example. Please note that d-gram model on AST constitutes a proper subset of topdown prediction models. In our evaluation, we shall demonstrate that *MaxFlash* can achieve significant speed-ups even with these less expressive models.

We take a straight-forward way to train the TPM. Given a training set $\mathcal{T}$, for each program $p \in \mathcal{T}$ and each vertex v on the AST of p, we record the grammar rule on v and the topdown context of v. The prediction function $\varphi(c, r)$ is defined as the frequency for rule r to occur under context c, i.e.,

$$\varphi(c, r) = \frac{\sharp\{\text{AST vertices with rule } r \text{ and context } c\}}{\sharp\{\text{AST vertices with context } c\}}$$

Since different programs use different sets of variables and parameters, we abstract these rules in the following way:

- All variables with the same type are regarded as the same.
- All integer constants are regarded as the same.
- Over the string manipulation domain, string constants are divided into four categories: one for the constant that is a substring of both input and output, one for input only, one for output only, and one for others. All constants in the same category are regarded as the same.

Though the training method seems straight-forward and $\mathcal{M}_d$ does not fully utilize the expression ability of TPM, our evaluation results show that *MaxFlash* performs well even when both the prediction model and the training method are simple.

6 EVALUATION

To evaluate *MaxFlash*, we report several experiments designed to answer the following research questions:

- **RQ1**: How does *MaxFlash* compare against existing synthesis techniques?
- **RQ2**: How does the prediction model affect the performance of *MaxFlash*?
- **RQ3**: Is the performance of *MaxFlash* sensitive to the number of training data?
- **RQ4**: Do the two reuse mechanisms boost the efficiency of *MaxFlash*?

6.1 Experimental Setup

Baseline Solvers. We compare *MaxFlash* with six state-of-the-art synthesizers, which are selected according to the following criteria. First, we compare *MaxFlash* with *Eusolver* and *CVC4* because of their excellent performance in SyGuS-Comp.

- *Eusolver* [Alur et al. 2017b], the winner of the PBE track in SyGuS-Comp 2016 [Alur et al. 2016]. *Eusolver* uses an optimized enumeration strategy which makes it especially efficient on synthesizing if-statements.
- *CVC4* [Reynolds et al. 2019a], the winner of the PBE-string track in SyGuS-Comp from 2017 to 2019 [Alur et al. 2019, 2017a]. *CVC4* synthesizes programs by an SMT solver and an algorithm named counterexample-guided quantifier instantiation [Reynolds et al. 2015].

Second, since *MaxFlash* utilizes dynamic programming and structural probability to accelerate PBE, we compare *MaxFlash* with state-of-the-art synthesizers in these two categories.

- *PROSE* [Polozov and Gulwani 2015], a state-of-the-art framework for dynamic-programming-based PBE systems, and *Transformation.Text* (abbreviated as *TText*), an instantiation of *PROSE* over the string manipulation domain evolving from *FlashFill* [Gulwani 2011].
- *Euphony* [Lee et al. 2018], a synthesizer utilizing *PHOG* [Bielik et al. 2016], a model based on structural probability, to accelerate *Eusolver*.
- *NGDS* [Kalyan et al. 2018], a synthesizer combing a neural network with *PROSE*. Instead of structural probability used in *MaxFlash*, *NGDS* uses a scoring function conditioned on input-output constraints: For each subproblem $(s, \mathbb{A})$ in *PROSE*, *NGDS* uses a neural network to score each possible form according to $\mathbb{A}$, and then use the score to guide synthesis.

Besides, we also take *Atlas* [Wang et al. 2018a], a state-of-the-art solver in terms of performance reported in its publication, into account. *Atlas* is based on abstraction refinement, which reduces the number of subproblems by abstraction and thus boosts the efficiency of synthesis
Benchmarks. We evaluate *MaxFlash* over two domains: string manipulation and matrix transformation.

- **String**. We use the string dataset $\mathcal{D}_{\mathbb{S}}$ collected by Lee et al. [2018]. $\mathcal{D}_{\mathbb{S}}$ is comprised of 205 string manipulation benchmarks, collected from 108 string-related benchmarks in *SyGuS* competition, 37 questions by spreadsheet users in StackOverflow, and 60 articles about Excel programming in Exceljet. For each benchmark, a grammar and a set of examples are provided: the number of examples varies from 2 to 400, with an average number of 42.8. For some benchmarks in $\mathcal{D}_{\mathbb{S}}$, oracle programs are provided, which can be used for training. The oracle program is the program found by *Eusolver*. For those benchmarks which *Eusolver* cannot solve within 10 minutes, no oracle program is provided: these benchmarks will be ignored while training the model.
- **Matrix**. We use the matrix dataset $\mathcal{D}_{\mathbb{M}}$ collected by Wang et al. [2018b]. $\mathcal{D}_{\mathbb{M}}$ is comprised of 39 matrix transformation benchmarks, which are collected from StackOverflow and MathWorks. For each benchmark, a single input-output example is provided: the number of entries in the input matrix varies from 6 to 640, with an average number of 73.5. 29 out of these benchmarks involve matrices of dimension greater than 2. For each benchmark in $\mathcal{D}_{\mathbb{M}}$, an oracle program is provided, which can be used for training.

Configurations. All of the following experiments are conducted on Intel Core i7-8700 3.2GHz 6-Core Processor with 48GB of RAM. For each execution, *MaxFlash* synthesizes programs from the grammar provided in the benchmark and takes $\mathcal{M}_1$ as the topdown context model (defined in Section 5) by default. For all baseline solvers which utilize neural networks, we train models and conduct the experiments on GeForce GTX 1080Ti. For all instantiations of *PROSE*, we run them on *PROSE* SDK version 7.16.0, released on July 27th, 2020. All of the executions in the following experiments are under a time limit set to 5 minutes and a memory limit set to 8 GB.

6.2 Exp 1: Comparison of the Approaches (RQ1)

Procedure. Over the string manipulation domain, we compare *MaxFlash* with *Eusolver*, *CVC4*, *PROSE*, *Atlas*, *Euphony*, and *NGDS*. Due to the differences between the baseline solvers, the experiment settings are slightly different:

- *Eusolver* and *CVC4*. These baseline solvers do not require a training set. To compare our approach with these three baseline solvers over all 205 benchmarks $\mathcal{D}_{\mathbb{S}}$, we train *MaxFlash* via leave-one-out cross-validation, i.e., for each benchmark in $\mathcal{D}_{\mathbb{S}}$, the TPM used by *MaxFlash* is trained from all other benchmarks in $\mathcal{D}_{\mathbb{S}}$.
- *PROSE*. We compare two different instantiations of *PROSE* over the string manipulation domain: (1) *TText*, a closed-source tool on a fixed unpublished domain-specific language, representing the best performance of *PROSE* on string manipulation domain. In this experiment, we compare *MaxFlash* with *TText* contained in *PROSE* SDK version 7.16.0. (2) An instantiation of *PROSE* on the *SyGuS* grammar, providing a fair comparison between *PROSE* and *MaxFlash* on a fixed DSL. When instantiating *PROSE* on the *SyGuS* grammar, we contacted the *PROSE* group and they verified the correctness of some critical points in our implementation. We compare *MaxFlash* with these two instantiations over all 205 benchmarks in $\mathcal{D}_{\mathbb{S}}$, and use leave-one-out cross-validation to train the TPM for *MaxFlash*.
- *Euphony* and *NGDS*. These approaches require a set of benchmarks to train prediction models. Therefore, we split $\mathcal{D}_{\mathbb{S}}$ into a training set and a testing set: For each approach, the prediction model is trained on the training set, and only the results on the testing set are recorded. We

split $\mathcal{D}_{\mathbb{S}}$ in the same way as *Euphony* [Lee et al. 2018] does: The training set and the testing set contain 123 and 82 benchmarks respectively. We do not use leave-one-out cross-validation here because (1) the implementation of *Euphony* crashed on a different training set; (2) training the neural network used in *NGDS* is time-consuming. Note that since we use the same training set as *Euphony* does, we directly use the trained prediction model published with the implementation of *Euphony*.

Since the implementation of *NGDS* is not available, we re-implement it according to its paper [Kalyan et al. 2018], and carefully choose the missing parameters: (1) For the hyperparameters, we evaluate ten different settings and select the one with the best performance. Specifically, we set the embedding dimension of chars to 128, the hidden size of the LSTM to 256, the number of the layers in LSTM to 3, and the size of the last hidden layer to 256. (2) For the scoring function on programs, we take it as the size of a program, i.e., our implementation aims to find the correct program with a smaller size. Besides, since *NGDS* is designed only for synthesizing from a single example, we use the structure proposed by Devlin et al. [2017] to extend it to support multi-examples and use CEGIS to reduce the number of examples dealt by *NGDS*.

- *Atlas*. *Atlas* is built on a fixed grammar which contains fewer operators than the grammar provided in the benchmarks. For fairness, we instantiate *MaxFlash* on the grammar used by *Atlas*, and compare it with *Atlas*. Besides, since the grammar is changed, we recalculate the oracle programs for benchmarks in $\mathcal{D}_{\mathbb{S}}$ by running *MaxFlash* with a trivial prediction model: all rules always have the same probability. This comparison uses all 205 benchmarks in $\mathcal{D}_{\mathbb{S}}$ and also uses leave-one-out cross-validation to train the prediction model for *MaxFlash*.

Over the matrix transformation domain, since *Eusolver*, *Euphony*, *CVC4*, and *NGDS* cannot be applied to the matrix transformation domain directly, we compare *MaxFlash* with two baseline solvers: *Atlas* and an instantiation of *PROSE* (denoted as $PROSE^M$). Since both solvers do not require a training set, we use leave-one-out cross-validation to train the prediction model for *MaxFlash*.

One delicate point is that $\mathcal{D}_{\mathbb{S}}$ contains redundant benchmarks: some benchmarks are the same except the number of examples. Therefore, while performing leave-one-out cross-validation, if the examples of a benchmark is a superset or subset of the test benchmark, this benchmark will be excluded from the training set. In $\mathcal{D}_{\mathbb{M}}$, there is no redundant benchmark. Such a treatment is also performed in other experiments in this section.

In this experiment, we measure the number and the ratio of solved benchmarks for each synthesizer with two different time limits: the first one is 500 milliseconds, which represents the industrial requirement of a user-interacting PBE system [Polozov and Gulwani 2016]; the second one is 5 minutes, which is large enough to show the general synthesis ability. We also measure the size of the memory required by each synthesizer to solve each benchmark: For each execution, a background process is used to monitor its memory usage and report the spike.

Results. The results are summarized in Table 2 while more details are drawn as Figure 4a and Figure 4b. To compare the time cost, we record the average speed-up ratio of *MaxFlash* to each baseline solver. More concretely, in each comparison, for each benchmark solved by both *MaxFlash* and the baseline within 5 minutes, we record the ratio of the time cost of the baseline solver to the time cost of *MaxFlash*. The geometric mean of all these ratios is shown in the eighth column while the ninth line shows the geometric mean of only *hard* benchmarks: a benchmark is hard iff either the baseline or *MaxFlash* takes more than 0.5s on it. To compare the memory usage, in each comparison, for each benchmark solved by both *MaxFlash* and the baseline solver within 5 minutes, we record the ratio of the memory usage of *MaxFlash* to the memory usage of the baseline solver. The geometric mean of these ratios is listed in the tenth column. We shall compare the time cost at first, following with a discussion on memory usage.

Table 2. The results of comparing *MaxFlash* with baselines.

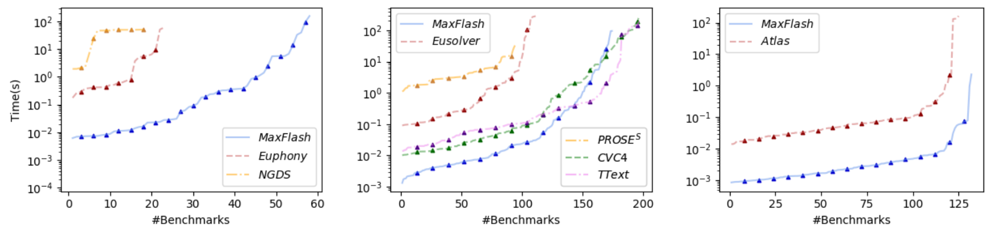
Baseline	$\mathcal{D}$	$\lvert\mathcal{D}_{\text{test}}\rvert$	#Solved in 500ms		#Solved in 5min		Speed-up		Memory
			MaxFlash	Baseline	*MaxFlash*	Baseline	All	Hard	Cost
Eusolver	$\mathcal{D}_{\mathbb{S}}$	205	143(70%)	63(31%)	174(85%)	111(54%)	×49.13	×126.2	35.63%
PROSE^S				0(0%)		94(46%)	×406.8	×406.8	2.551%
CVC4				123(60%)	174(85%)	196(96%)	×5.097	×8.670	66.53%
TText			143(70%)	154(75%)	174(85%)	197(96%)	×4.107	×1.148	17.05%
Atlas			130(63%)	115(56%)	132(64%)	125(61%)	×23.34	×95.98	3.883%
Euphony		82	43(52%)	9(11%)	58(71%)	23(28%)	×38.16	×55.30	37.26%
NGDS				0(0%)		18(22%)	×1043	×1043	23.10%
PROSE^M	$\mathcal{D}_{\mathbb{M}}$	39	34(87%)	0(0%)	39(100%)	33(85%)	×2080	×2080	8.137%
Atlas				29(74%)		38(97%)	×15.50	×3.663	3.776%

Comparing with *Eusolver* and *Euphony* over the string manipulation domain, *MaxFlash* not only solves many more benchmarks but also achieves a significant speed-up.

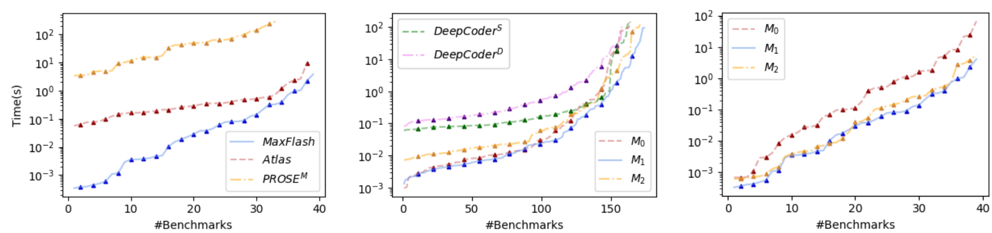
Comparing with *PROSE*, a state-of-the-art framework for dynamic-programming-based synthesizer, *MaxFlash* performs significantly better when both *MaxFlash* and *PROSE* are instantiated on the same grammar over both domains. However, the performance of *PROSE* can be greatly improved by a well-designed DSL: *TText* solved more benchmarks than *MaxFlash* whenever the time limit is 500 milliseconds or 5 minutes. We suspect that *TText* solves many more benchmarks because its DSL contains more powerful language constructors and witness functions such as regex, which make some hard benchmarks in $\mathcal{D}_{\mathbb{S}}$ much simpler. For example, on `univ_4.sl`, a benchmark on which both *PROSE*^{S} and *MaxFlash* failed but *TText* succeeded, *TText* uses regex ``‘‘egexPair((‘,’ or ‘and’) + ‘Upper Case’, ‘ϵ’)"`` to capture two cases, starting with `‘,’` and starting with `‘and’`, at the same time. However, in the *SyGuS* grammar, regex is unavailable, and thus a program has to use if-condition to deal with these cases, resulting in a program that is at least 2 times larger in size. This result suggests that a re-implementation of *MaxFlash* on the DSL used by *TText* may potentially achieve a better performance. Please note that even though *TText* is built on a much more powerful DSL, *MaxFlash* still achieves an average speed-up equal to ×4.107 on benchmarks solved by both *TText* and *MaxFlash*.

Comparing with *NGDS*, *MaxFlash* significantly outperforms it over the string manipulation domain. *MaxFlash* has two critical differences against *NGDS*: (1) The default prediction model $\mathcal{M}_1$ used by *MaxFlash* is much more lightweight than the neural network used by *NGDS*, allowing *MaxFlash* to explore more subproblems within the same time limit; (2) Comparing with the search algorithm used by *NGDS*, *MaxFlash* further utilizes heuristic functions and iterative deepening search, making *MaxFlash* more effective on cutting off useless search branches. In this experiment, our results are significantly different from the results reported by Kalyan et al. [2018]. Such a difference is caused by the difference in the experiment setting: In the experiment conducted by Kalyan et al. [2018], for each benchmark, only one example is provided to *NGDS*, while in our experiment, all examples are provided. Therefore, we further perform an experiment in which only the first example in each benchmark is considered. The results show that the performance of *NGDS* is greatly improved: *NGDS* finishes 41(50%) benchmarks within 5 minutes. But even comparing with the new results, *MaxFlash* still solves 17(21%) more benchmarks with an average speed-up equal to ×554.6 on benchmarks solved by both *MaxFlash* and *NGDS*.

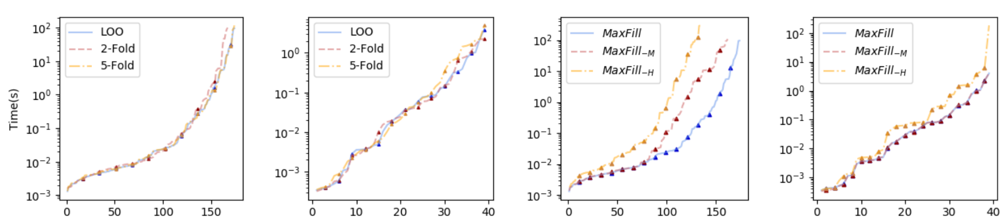
Comparing with *Atlas*, *MaxFlash* solves more benchmarks with faster speeds over both domains. Furthermore, over the string manipulation domain, *Atlas* is more limited than *MaxFlash*: *Atlas* is built on a fixed simple grammar and it is not clear how to extend its algorithm to the grammar



(a) Results of Exp 1 on $\mathcal{D}_{\mathbb{S}}$



(b) Results of Exp 1 on $\mathcal{D}_{\mathbb{M}}$. (c) Results of Exp 2 on $\mathcal{D}_{\mathbb{S}}$. (d) Results of Exp 2 on $\mathcal{D}_{\mathbb{M}}$.



(e) Results of Exp 3 on $\mathcal{D}_{\mathbb{S}}$ and $\mathcal{D}_{\mathbb{M}}$ respectively. (f) Results of Exp 4 on $\mathcal{D}_{\mathbb{S}}$ and $\mathcal{D}_{\mathbb{M}}$ respectively.

Fig. 4. The results of four experiments. For each approach, we sort its solved benchmarks in the increasing order of the time cost, and plot the ith benchmark as a point (i, t_i) where t_i is the time cost.

used in *SyGuS*. In contrast, *MaxFlash* can solve more benchmarks with the *SyGuS* grammar. Over the matrix transformation domain, *MaxFlash* outperforms *Atlas* on both time cost and the number of solved benchmarks, but on hard benchmarks, its advantage is not as significant as in $\mathcal{D}_{\mathbb{S}}$. The reason is that two extra reuse mechanisms in *MaxFlash* involve multi-example subproblems, but every benchmark in $\mathcal{D}_{\mathbb{M}}$ contains only one example. Therefore, both of them are less effective in $\mathcal{D}_{\mathbb{M}}$ than $\mathcal{D}_{\mathbb{S}}$.

Comparing with *CVC4*, *MaxFlash* still has an obvious speed-up and has a better performance when the time limit is 500ms, which demonstrates that *MaxFlash* performs better than *CVC4* in user-interacting scenarios. When the time limit is 5 minutes, *CVC4* solves 22 more benchmarks than *MaxFlash*. We investigated the results of other solvers and found most of them (19/22) cannot be solved by any other solver. One possible reason is that *CVC4* is built inside a constraint solver and could utilize specific theory solvers, which are probably critical to some benchmarks. Note that even though *CVC4* utilizes the theory solvers, *MaxFlash* still performs much faster on the tasks that are solved by both synthesizers, and works significantly better than *CVC4* within 500ms, the interaction limit.

Besides, we also compared the synthesized programs of *MaxFlash* with *CVC4* and found that *MaxFlash* could synthesize much simpler programs than *CVC4*: over the 171 benchmarks solved by both *MaxFlash* and *CVC4*, the programs found by *MaxFlash* uses 5.111 operators on average while the programs found by *CVC4* uses 124.4 operators on average. Moreover, the programs found by *MaxFlash* are often more natural than programs found by *CVC4*. For example, on `38871714.sl`, a benchmark aims to remove all angle brackets from the input string, the program found by *MaxFlash* is `(str.replace (str.replace input ">" "") "<" "")`, which is much more natural than the program found by *CVC4*, a program uses 159 operators. The reason for this difference is because *MaxFlash* always finds the most probable program according to a model based on structural probability, while *CVC4* only returns an arbitrary program that is consistent with all examples.

In terms of memory usage, though *MaxFlash* uses two memoization tables to reuse results, it still uses less memory space than all baselines. This is because the advanced search algorithm together with structural probability can effectively prune off subproblems, and thus only a small portion of subproblems is cached in the memoization tables. As for baseline solvers, they can be divided into two categories according to whether subproblems are used:

- The first category contains *PROSE*, *Atlas*, and *NGDS*, in which visited subproblems are recorded: Both *PROSE* and *Atlas* require a lot more memory space than *MaxFlash*. This is because *PROSE* always memorizes all possible subproblems; *Altas* uses an abstraction space to reduce the number of possible subproblems but it would still memorize all possibilities. Comparing with them, *NGDS* requires less memory because it uses a prune-off strategy to reduce the number of visited subproblems. However, according to the result, *NGDS* memorizes many more subproblems than *MaxFlash* does, implying the advantage of our search algorithm.
- The second category contains *Eusolver*, *CVC4*, and *Euphony*, in which other information is memorized instead of subproblems: *Eusolver* and *Euphony* stores all visited partial subprograms; *CVC4* records clauses learned from conflicts as an SMT solver. Though the result suggests that these approaches have advantages on the memory usage against those based on subproblems, *MaxFlash* still outperforms them with the help of the search algorithm and structural probability.

6.3 Exp 2: Comparison of Prediction Models (RQ2)

Procedure. In this experiment, we test how the choice of the prediction model affects the performance of *MaxFlash*. Here we consider three different topdown context models: $\mathcal{M}_0, \mathcal{M}_1$, and $\mathcal{M}_2$ (defined in Section 5). We train each topdown prediction model as discussed in Section 5 and use leave-one-out cross-validation to get the training set.

In addition, we compare TPM with *DeepCoder* [Balog et al. 2017], a state-of-the-art framework for using neural networks to guide programming-by-example. *DeepCoder* trains a neural network from random programs, and use it to predict the probability for each operator to be used (i.e., generates an $\mathcal{M}_0$ model) from some given input-output examples. *DeepCoder* can be combined with the search algorithm in *MaxFlash* and in this experiment, we consider two different ways of combining *DeepCoder* with *MaxFlash*.

- **Statically** (denoted as *DeepCoder*^{S}). For each benchmark, we firstly invoke *DeepCoder* to generate an $\mathcal{M}_0$ model, and then use it to guide *MaxFlash*.
- **Dynamically** (denoted as *DeepCoder*^{D}). For each benchmark, we invoke *MaxFlash* with context model $\mathcal{M}_0$ before determining the prediction model. Then for each subproblem $(s, \mathbb{A}, c)$ visited by *MaxFlash*, we invoke *DeepCoder* to generate a prediction specifically for it from $\mathbb{A}$.

We re-implement *Deepcoder* according to its paper [Balog et al. 2017] as its implementation is not published. Besides, since the neural network used by *DeepCoder* cannot be easily extended to the matrix transformation domain, we only instantiated *DeepCoder* over the string manipulation domain and measured the performance of $DeepCoder^S$ and $DeepCoder^D$ on dataset $\mathcal{D}_{\mathbb{S}}$.

Results. For each prediction model, we calculate the geometric mean of per-task speed-up ratios of *MaxFlash* with the default model (the ratio of the time cost of *MaxFlash* with each prediction model to that of *MaxFlash* with $\mathcal{M}_1$). The results are summarized in Table 3 while more details are drawn as Figure 4c and Figure 4d.

Table 3. The results of different prediction models.

Prediction Model	$\mathcal{D}_{\mathbb{S}}$				$\mathcal{D}_{\mathbb{M}}$			
	#Solved		Average Speed-up		#Solved		Average Speed-up	
	500ms	5min	All	Hard	500ms	5min	All	Hard
$\mathcal{M}_0$	134	164	×1.618	×6.124	23	39	×6.791	×10.52
$\mathcal{M}_1$ (Default)	143	174	×1.000	×1.000	34	39	×1.000	×1.000
$\mathcal{M}_2$	139	171	×2.571	×2.975	33	39	×1.428	×1.873
$DeepCoder^S$	140	163	×7.561	×2.027				
$DeepCoder^D$	98	158	×21.99	×20.51				

$\mathcal{M}_1$ performs best among all three topdown context models $\mathcal{M}_1$ outperforms $\mathcal{M}_0$ because the context in $\mathcal{M}_1$ is more refined which makes the prediction model more precise. $\mathcal{M}_1$ outperforms $\mathcal{M}_2$ because of the following two reasons:

(1) The training set is small and the training method is straight-forward, therefore $\mathcal{M}_2$ may overfit on the training set, since $\mathcal{M}_2$ is more complex than $\mathcal{M}_1$.
(2) The context set of $\mathcal{M}_2$ is larger than $\mathcal{M}_1$. Thus there are more subproblems for $\mathcal{M}_2$ than $\mathcal{M}_1$. The advantage of using a more complex prediction model is not enough to counter the disadvantage of having more subproblems.

The result suggests that the size of the context model affects the efficiency of *MaxFlash* on two sides. An ideal prediction model for *MaxFlash* should not only be precise but also use a small number of contexts. Furthermore, the difference between different context models is much smaller than the difference between *MaxFlash* and most baselines solvers, indicating the overall effectiveness of *MaxFlash*.

Comparing with *DeepCoder*, the default model $\mathcal{M}_1$ performs better than both $DeepCoder^S$ and $DeepCoder^D$: $DeepCoder^D$ performs the worst, while the performance of $DeepCoder^S$ is closer to that of $\mathcal{M}_1$. This difference is because $DeepCoder^S$ and $DeepCoder^D$ use the neural network differently: $DeepCoder^S$ invokes the network only once, while $DeepCoder^D$ invokes the network for each different subproblem. Therefore, this result demonstrates that reducing the time cost of the prediction model is critical to accelerating programming-by-example, implying the advantage of structural probability and TPM. Note that in this experiment, both $DeepCoder^S$ and $DeepCoder^D$ are built on *MaxFlash*: they have already benefited from the search algorithm proposed in this paper.

6.4 Exp 3: Comparison of the Number of Training Datas (RQ3)

Procedure. In this experiment, we test whether the performance of *MaxFlash* is sensitive to the number of training data.

In previous experiments, we use leave-one-out (LOO) cross-validation to train the prediction model for *MaxFlash* by default. In this experiment, we further consider two cross-validation methods: 2-fold cross-validation and 5-fold cross-validation. The procedure of n-fold cross-validation is: (1)

randomly divide the dataset into n subsets, (2) for each subset, the prediction model used by *MaxFlash* is trained on benchmarks from all other subsets. Therefore, 2-fold and 5-fold cross-validation use 50% and 20% fewer training data than leave-one-out cross-validation respectively.

For each cross-validation method, we run *MaxFlash* on all benchmarks in both datasets and measure the time costs.

Results. The results are summarized in Figure 4e. These two figures show that the size of the training set has a positive impact on the performance of *MaxFlash*, especially for hard benchmarks. This result shows that *MaxFlash* could perform even better when more training data is provided.

On the other hand, the impact of training data is not significant. After reducing the number of training data by 50%, *MaxFlash* only becomes 9.11% and 0.85% slower in average on $\mathcal{D}_{\mathbb{S}}$ and $\mathcal{D}_{\mathbb{M}}$ respectively. Just as shown in Section 6.2, comparing to the advantages of *MaxFlash* to other baseline solvers, such a loss is much smaller. This result demonstrates that even under the lack of training data, *MaxFlash* can still perform well.

6.5 Exp 4: Effects of Optimizations (RQ4)

Procedure. In this experiment, we test whether the two reuse mechanisms (reusing through heuristic function, reusing results with fewer constraints) speed up *MaxFlash*.

Here, we further implement two weakened solvers $MaxFlash_{-H}$, $MaxFlash_{-M}$, which disable reusing through heuristic function, reusing results with fewer constraints from *MaxFlash* respectively. We run these three solvers on all benchmarks in $\mathcal{D}$ and still use leave-one-out cross-validation to train the prediction model.

Results. For each reuse mechanism, we calculate the geometric mean of the per-task speed-up ratios (the ratio of the time cost of the weakened solver to that of *MaxFlash*). The results are summarized in Table 4 while more details are drawn as Figure 4f.

Table 4. The results of disabling each optimization module.

Disabled Module	$\mathcal{D}_{\mathbb{S}}$				$\mathcal{D}_{\mathbb{M}}$			
	#Solved		Average Speed-up		#Solved		Average Speed-up	
	500ms	5min	All	Hard	500ms	5min	All	Hard
Heuristic Function	96	134	×8.193	×116.1	29	39	×2.253	×5.628
Fewer Constraints	114	162	×3.252	×24.01	34	39	×1.000	×1.000

As shown in Table 4, over the string manipulation domain, both two reuse mechanisms boost the speed of *MaxFlash* significantly. Comparing between them, the effect of reusing through heuristic function is more obvious, because it not only utilizes the relationship between subproblems, but also helps to reuse all failed results. Besides, reusing through heuristic also produces a more precise heuristic function, which can help *MaxFlash* to prune off more search branches.

Over the matrix transformation domain, the second reuse mechanism does not work. The reason is that each benchmark in $\mathcal{D}_{\mathbb{M}}$ contains only a single example, and thus all optimizations for multi-example subproblems become useless on $\mathcal{D}_{\mathbb{M}}$. Even though, reusing through heuristic function still achieves a considerable speed-up, as shown in Table 4.

7 RELATED WORK

Accelerating program synthesis. There have been lots of techniques proposed to accelerate program synthesis. We summarize the techniques adopted by state-of-the-art solvers below:

- *Dynamic-programming* based synthesizers [Barowy et al. 2015; Gulwani 2011; Kini and Gulwani 2015; Le and Gulwani 2014; Padhi et al. 2018; Polozov and Gulwani 2015; Singh and Gulwani

2012], represented by *FlashFill* [Gulwani 2011], is closely related to *MaxFlash*, which also uses witness functions to divide synthesis tasks into subprograms and uses dynamic programming to speed up synthesis. *FlashFill* has been integrated into *PROSE* [Polozov and Gulwani 2015], a framework for dynamic-programming-based synthesizers, and has been constantly evolving as tool *Transformation.Text*.
- *Probabilistic-model* based synthesizers [Lee et al. 2018], represented by *Euphony* [Lee et al. 2018], model the probability for a program to be used as a probabilistic model and thus utilize structural probability to guide the synthesis. *Euphony* utilizes a probabilistic model based on structural probability, named *PHOG*, to accelerate an enumerative-based synthesizer, and thus achieves a significant speed-up.
- *Divide-and-conquer* based synthesizers [Alur et al. 2017b], represented by *Eusolver* [Alur et al. 2017b], models if-statements as decision trees, and uses a divide-and-conquer algorithm to synthesize predicates and branch expressions separately.
- *Abstraction refinement* based synthesizers [Wang et al. 2018a,b], represented by *Atlas* [Wang et al. 2018a], abstract the constraints to an abstraction space and thus reduce the number of possible search states. The effectiveness of abstraction heavily dependent on the DSL: All existing works are built on simple DSLs.
- *Refutation* based synthesizers [Reynolds et al. 2019a,b], represented by *CVC4* [Reynolds et al. 2019a], utilize an algorithm named counterexample-guided quantifier instantiation [Reynolds et al. 2015] to synthesize programs from unsatisfiability proofs given by the theory solvers. Relying on efficient theory solvers, these synthesizers could finish some tasks which are extremely hard for other techniques.

MaxFlash combines the first two techniques. We have evaluated *MaxFlash* against state-of-the-art solvers of all these directions in Section 6. The result demonstrates that *MaxFlash* achieves a faster speed than all these approaches for interactive tasks. Besides, it is remained as future work to study whether theory solvers and abstraction refinement can be utilized in *MaxFlash*, since these techniques optimize different aspects of program synthesis.

Program estimation. Program estimation [Xiong et al. 2018] is a problem related to program synthesis where the goal is not only to find a program satisfying the specification but also a program that is most likely under a context, such as a natural language description. This problem is also recognized as multi-layer specification problem [Chen et al. 2019] or multi-modal synthesis [Chen et al. 2020] in literature. Though program estimation approaches also utilize a probabilistic model and optimize the probability of the resulting program, their goal is different from acceleration, making their approaches difficult to be directly used for acceleration.

- Some approaches [Chen et al. 2019; Neelakantan et al. 2017] use natural language descriptions as context, which is not available when accelerating PBE.
- While some approaches [Balog et al. 2017; Devlin et al. 2017; Kalyan et al. 2018; Menon et al. 2013] use input-output examples as context, their probabilistic models are computationally complex because (1) examples are not easy to encode, and (2) it is more desirable to use a complex model to achieve better accuracy. As a result, evaluating the probabilistic model becomes a bottleneck of their performance, and thus prevents them from achieving significant speed-ups.

  NGDS [Kalyan et al. 2018] and *DeepCoder* [Balog et al. 2017] are two state-of-the-art approaches in this category. Both of them use neural networks to predict from input-output examples: *NGDS* uses a neural network to score each possible form for each subproblem, and use a basic branch-and-bound to prune off search branches; *DeepCoder* uses a neural network to generate an $\mathcal{M}_0$ model, and uses it to guide the synthesis process of a client program synthesizer. Comparing with them, *MaxFlash* is different in the following aspects: (1) Training a topdown prediction model does not require any domain knowledge, making *MaxFlash* easily applicable to different domains. (2) Evaluating a d-gram topdown prediction model is much faster than evaluating a neural network, allowing *MaxFlash* to explore more subproblems within the same time limit. (3) Comparing with the basic branch-and-bound used in *NGDS*, the search algorithm in *MaxFlash* utilizes more search strategies and is potentially more effective on pruning off useless search branches. We have established a detailed comparison between *MaxFlash* and these two approaches in Section 6. The result demonstrates that *MaxFlash* outperforms both of them on speed.

Probabilistic models based on structural probability. There have been several existing probabilistic models based on structural probability. We summarize some representative models below:

- *AST n-gram* constitutes a proper subset of topdown prediction models: An *AST n-gram* model can be regarded as a topdown prediction model with context model $\mathcal{M}_n$ (defined in Section 5).
- *Hidden Markov model*: A topdown prediction model is a special Hidden Markov model capturing the tree-path information from the root to the current vertex.
- *PHOG* [Bielik et al. 2016] constitutes a proper superset of topdown prediction models. A *PHOG* model uses a domain-specific language to extract contexts from the whole partial program, while a topdown prediction model only considers the path from the root. However, it would be hard to combine *PHOG* with dynamic programming since the context model in *PHOG* allows sibling subproblems to be dependent on each other.
- *code2vec* [Alon et al. 2019] also constitutes a proper superset of topdown prediction models. A *code2vec* model encodes AST paths into real-valued vectors instead of a finite set. However, it would also be hard to combine *code2vec* with dynamic programming since the number of possible contexts in *code2vec* is infinite, leading to an extremely large number of different subproblems.

8 CONCLUSION

We propose *MaxFlash*, a novel PBE framework, which uses topdown prediction models, a kind of probabilistic models based on structural probability, to guide a search based on dynamic programming. *MaxFlash* uses a series of methods to resolve two major challenges in combining these two techniques. To make local subproblems aware of structural probability, *MaxFlash* involves a topdown context and a probability lowerbound to subproblems. The context makes the structural probability calculable in subproblems, and the lowerbound helps *MaxFlash* to avoid improbable subproblems. To increase the chance of reusing results, *MaxFlash* turns the subproblems into optimization problems. Besides, to prune off search branches, *MaxFlash* uses an efficient search algorithm based on iteratively deepening search and branch-and-bound. To further boost the opportunities of reuse, *MaxFlash* involves two novel reuse mechanisms, reusing through heuristic function and reusing results with fewer constraints. We instantiate our framework over the string manipulation domain and the matrix transformation domain, and compare it with other state-of-the-art PBE synthesizers. Our results show that *MaxFlash* achieves $\times 4.107 - \times 2080$ speed-ups against these synthesizers on 244 real-world tasks.

ACKNOWLEDGEMENTS

We sincerely thank Xinyu Wang and Microsoft PROSE team for the help with the experiment setup and the anonymous OOPSLA reviewers for valuable feedback on this work. This work is supported in part by National Key Research and Development Program of China under Grant No. SQ2019YFE010068, National Natural Science Foundation of China under Grant Nos. 61922003, 61672045 and 61620106007.

REFERENCES

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: learning distributed representations of code. *Proc. ACM Program. Lang.* 3, POPL (2019), 40:1–40:29. https://doi.org/10.1145/3290353

Rajeev Alur, Dana Fisman, Saswat Padhi, Rishabh Singh, and Abhishek Udupa. 2019. SyGuS-Comp 2018: Results and Analysis. *CoRR* abs/1904.07146 (2019). arXiv:1904.07146 http://arxiv.org/abs/1904.07146

Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. 2016. SyGuS-Comp 2016: Results and Analysis. In *Proceedings Fifth Workshop on Synthesis, SYNT@CAV 2016, Toronto, Canada, July 17-18, 2016.* 178–202. https://doi.org/10.4204/EPTCS.229.13

Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. 2017a. SyGuS-Comp 2017: Results and Analysis. In *Proceedings Sixth Workshop on Synthesis, SYNT@CAV 2017, Heidelberg, Germany, 22nd July 2017.* 97–115. https://doi.org/10.4204/EPTCS.260.9

Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017b. Scaling Enumerative Program Synthesis via Divide and Conquer. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I.* 319–336. https://doi.org/10.1007/978-3-662-54577-5_18

Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. DeepCoder: Learning to Write Programs. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings.* https://openreview.net/forum?id=ByldLrqlx

Daniel W. Barowy, Sumit Gulwani, Ted Hart, and Benjamin G. Zorn. 2015. FlashRelate: extracting relational data from semi-structured spreadsheets using examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015.* 218–228. https://doi.org/10.1145/2737924.2737952

Pavol Bielik, Veselin Raychev, and Martin T. Vechev. 2016. PHOG: Probabilistic Model for Code. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016.* 2933–2942. http://proceedings.mlr.press/v48/bielik16.html

Qiaochu Chen, Xinyu Wang, Xi Ye, Greg Durrett, and Isil Dillig. 2020. Multi-Modal Synthesis of Regular Expressions. (2020).

Yanju Chen, Ruben Martins, and Yu Feng. 2019. Maximal multi-layer specification synthesis. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019.* 602–612. https://doi.org/10.1145/3338906.3338951

Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. 2017. RobustFill: Neural Program Learning under Noisy I/O. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017.* 990–998. http://proceedings.mlr.press/v70/devlin17a.html

Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based synthesis of table consolidation and transformation tasks from examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017.* 422–436. https://doi.org/10.1145/3062341.3062351

Giorgio Gallo, Giustino Longo, and Stefano Pallottino. 1993. Directed Hypergraphs and Applications. *Discret. Appl. Math.* 42, 2 (1993), 177–201. https://doi.org/10.1016/0166-218X(93)90045-P

Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011.* 317–330. https://doi.org/10.1145/1926385.1926423

Ashwin Kalyan, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. 2018. Neural-Guided Deductive Search for Real-Time Program Synthesis from Examples. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings.* https://openreview.net/forum?id=rywDjg-RW

Dileep Kini and Sumit Gulwani. 2015. FlashNormalize: Programming by Examples for Text Normalization. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015.* 776–783. http://ijcai.org/Abstract/15/115

Richard E. Korf. 1985. Depth-First Iterative-Deepening: An Optimal Admissible Tree Search. *Artif. Intell.* 27, 1 (1985), 97–109. https://doi.org/10.1016/0004-3702(85)90084-0

Ailsa H. Land and Alison G. Doig. 1960. An Automatic Method of Solving Discrete Programming Problems. *Econometrica* 28 (1960), 497–520.

Vu Le and Sumit Gulwani. 2014. FlashExtract: a framework for data extraction by examples. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014.* 542–553. https://doi.org/10.1145/2594291.2594333

Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. 2018. Accelerating search-based program synthesis using learned probabilistic models. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018.* 436–449. https://doi.org/10.1145/3192366.3192410

Aditya Krishna Menon, Omer Tamuz, Sumit Gulwani, Butler W. Lampson, and Adam Kalai. 2013. A Machine Learning Framework for Programming by Example. In *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*. 187–195. http://proceedings.mlr.press/v28/menon13.html

Arvind Neelakantan, Quoc V. Le, Martín Abadi, Andrew McCallum, and Dario Amodei. 2017. Learning a Natural Language Interface with Neural Programmer. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. https://openreview.net/forum?id=ry2YOrcge

Saswat Padhi, Prateek Jain, Daniel Perelman, Oleksandr Polozov, Sumit Gulwani, and Todd D. Millstein. 2018. FlashProfile: a framework for synthesizing data profiles. *PACMPL* 2, OOPSLA (2018), 150:1–150:28. https://doi.org/10.1145/3276520

Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: a framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*. 107–126. https://doi.org/10.1145/2814270.2814310

Oleksandr Polozov and Sumit Gulwani. 2016. Program synthesis in the industrial world: Inductive, incremental, interactive. In *5th Workshop on Synthesis (SYNT)*.

Andrew Reynolds, Haniel Barbosa, Andres Nötzli, Clark W. Barrett, and Cesare Tinelli. 2019a. cvc4sy: Smart and Fast Term Enumeration for Syntax-Guided Synthesis. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II*. 74–83. https://doi.org/10.1007/978-3-030-25543-5_5

Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli, and Clark W. Barrett. 2015. Counterexample-Guided Quantifier Instantiation for Synthesis in SMT. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*. 198–216. https://doi.org/10.1007/978-3-319-21668-3_12

Andrew Reynolds, Viktor Kuncak, Cesare Tinelli, Clark W. Barrett, and Morgan Deters. 2019b. Refutation-based synthesis in SMT. *Formal Methods in System Design* 55, 2 (2019), 73–102. https://doi.org/10.1007/s10703-017-0270-2

David E. Shaw, William R. Swartout, and C. Cordell Green. 1975. Inferring LISP Programs From Examples. In *Advance Papers of the Fourth International Joint Conference on Artificial Intelligence, Tbilisi, Georgia, USSR, September 3-8, 1975*. 260–267. http://ijcai.org/Proceedings/75/Papers/037.pdf

Rishabh Singh and Sumit Gulwani. 2012. Learning Semantic String Transformations from Examples. *PVLDB* 5, 8 (2012), 740–751. https://doi.org/10.14778/2212351.2212356

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. 2006. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*. 404–415. https://doi.org/10.1145/1168857.1168907

Chenglong Wang, Alvin Cheung, and Rastislav Bodík. 2017. Synthesizing highly expressive SQL queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. 452–466. https://doi.org/10.1145/3062341.3062365

Xinyu Wang, Greg Anderson, Isil Dillig, and Kenneth L. McMillan. 2018a. Learning Abstractions for Program Synthesis. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*. 407–426. https://doi.org/10.1007/978-3-319-96145-3_22

Xinyu Wang, Isil Dillig, and Rishabh Singh. 2018b. Program synthesis using abstraction refinement. *PACMPL* 2, POPL (2018), 63:1–63:30. https://doi.org/10.1145/3158151

Yingfei Xiong, Bo Wang, Guirong Fu, and Linfei Zang. 2018. Learning to Synthesize. In *International Genetic Improvement Workshop*. https://doi.org/10.1145/3194810.3194816

Navid Yaghmazadeh, Christian Klinger, Isil Dillig, and Swarat Chaudhuri. 2016. Synthesizing transformations on hierarchically structured data. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. 508–521. https://doi.org/10.1145/2908080.2908088

Sai Zhang and Yuyin Sun. 2013. Automatically synthesizing SQL queries from input-output examples. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, Ewen Denney, Tevfik Bultan, and Andreas Zeller (Eds.). IEEE, 224–234. https://doi.org/10.1109/ASE.2013.6693082

A APPENDIX

To prove Theorem 4.3, we involve the concept of *Synthesis State*. A synthesis state $\mathbb{S}$ is comprised of a subproblem *problem*, a probability limit *lowerbound*, a global memoization table *memoTable* and the auxiliary memoization table $memoTable_h$ for the heuristic function. A synthesis state represents an invocation `OptimalProgram(`*problem, lowerbound*`)` under memoization table *memoTable* and $memoTable_h$. Clearly, when *memoTable* and $memoTable_h$ is given, the procedure of `OptimalProgram(·)` is fixed. We denote its return value as $p(\mathbb{S})$.

We define the property *correct* for synthesis state $\mathbb{S}$, memoization table *memoTable* and auxiliary memoization table $memoTable_h$:

- A synthesis state $\mathbb{S} = (problem, lowerbound, memoTable, memoTable_h)$ is correct $\iff$ if the log-probability of the most probable program $program^*$ of *problem* is larger than *lowerbound*, $p(\mathbb{S}) = program^*$, otherwise $p(\mathbb{S}) = \top$.
- The memoization table *memoTable* is correct $\iff$ for any subproblem *problem* in *memoTable*, $memoTable[problem]$ is the most probable program of subproblem *problem*.
- The auxiliary memoization table $memoTable_h$ is correct $\iff$ for any subproblem *problem* in $memTable_h$, $memoTable_h[problem] \geq bestProb(problem)$.

Lemma A.1. *If $memoTable_h$ is correct, heuristic must be a valid heuristic function, i.e., for any subproblem problem, heuristic(problem) is no smaller than bestProb(problem).*

Proof. Suppose $problem = (s, \mathbb{A}, c)$. According to Section 4.1, *heuristic*(*problem*) is equal to $\min(heuristic_0((s,\mathbb{A},c)), \min_{\mathbb{A}' \text{ prefix of } \mathbb{A}} memoTable_h[(s,\mathbb{A}',c)])$. By correctness of $memoTable_h$ and Lemma 3.9, for any prefix $\mathbb{A}'$ of $\mathbb{A}$:

$$memoTable_h[(s,\mathbb{A}',c)] \geq bestProb(s,\mathbb{A}',c) \geq bestProb(s,\mathbb{A},c)$$

Since $heuristic_0$ is a valid heuristic function, we get the validity of *heuristic*, i.e., *bestProb*(*problem*) ≤ *heuristic*(*problem*). □

While synthesizing, *MaxFlash* invokes OptimalProgram many times. These invocations form some recursion trees: each recursion tree representing one iteration of the iteratively deepening search. To prove the correctness of *MaxFlash*, we only need to prove the invocations of OptimalProgram are always correct.

Lemma A.2. *For any synthesis state $\mathbb{S} = (problem, lowerbound, memoTable, memoTable_h)$, if memoTable, $memoTable_h$ are correct, then:*

- *memoTable and $memoTable_h$ will aways be correct during* `OptimalProgram(`*problem,lowerbound*`)`.
- *Synthesis state $\mathbb{S}$ will be correct.*

Proof. We prove it by structural induction on the recursion tree. Suppose all the recursive invocation of `OptimalProgram(`*S, lowerBound*`)` satisfies this property, we now prove this invocation itself also does. Note that the following proof is also held for the base case: since the base case have no self invocation, applying the following proof to base cases does not utilize the induction hypothesis.

Let p^+ (to distinguish it from *bestProgram* used in Algorithm 6) be the optimal program to *problem*. We discuss according to the return position of `OptimalProgram(`*problem,lowerBound*`)` in Algorithm 7:

- **Return at Line** 10 **in Algorithm 4.** By Lemma **??**, *heuristic*(*problem*) ≥ *bestProb*(*problem*). Therefore, if `OptimalProgram(·)` returns at this time, there must not be any valid program for *problem* with log-probability at least *lowerbound*. Besides, OptimalProgram does not modify *memoTable*, $memoTable_h$ in these case. Therefore, $\mathbb{S}$, *memoTable*, $memoTable_h$ are all correct.

- **Return at Line** 7 **in Algorithm 3.** Since *memoTable* is correct, returning *memoTable*[*problem*] directly must also be correct for *problem*.
- **Return at Line** 8 **or Line** 12 **in Algortihm 7.** Let $problem_l$ be the recursively invoked subproblem. By the induction hypothesis, *candidate* is correct, and *memoTable*, $memoTable_h$ are still correct after this invocation. There are two possible cases:
  - ***candidate* is** $\top$. Then $bestProb(problem) \leq bestProb(problem_l) \leq lowerbound$. Therefore, it's correct for `OptimalProgram(·)` to return $\top$.
  - ***candidate* is not** $\top$ **and *candidate* satisfies the last constraint**. At this time, *candidate* passes all the constraints, therefore *candidate* is a valid program for *problem*. Then, by Lemma 3.9, $\mathcal{P}_c[candidate] \leq bestProb(problem) \leq bestProb(problem_l) = \mathcal{P}_c[candidate]$, so *candidate* is also the most probable program to *problem*. Therefore it's correct for `OptimalProgram(·)` to return *candidate*.
- **Return at Line** 18 **of Algorithm 6.** Firstly, we show the value of *bestProgram* is always either $\top$ or a valid program to *problem*. Every time `OptimalProgram(·)` updates *bestProgram*, the new program *candidate* must be the most probable program to some scheme. By the induction hypothesis, each subprogram of *candidate* must satisfies the input-output constraint of the corresponding subproblem. Therefore, *candidate* must valid for *problem*.
  Next, we demonstrate that if $bestProb(probblem) > lowerbound$, the optimal program p^+ must be used to update *bestProgram* during `OptimalProgram(`*problem,lowerbound*`)`. Since *bestProbability* is either *lowerbound* or the log-probability of some program, when p^+ is found, *bestProbability* should be strictly smaller than $\mathcal{P}_c[p^+]$. At this time, p^+ will be used to update *bestProgram*.
  Therefore, if $bestProb(problem) > lowerbound$, *bestProgram* must be updated to p^+ at some time, and it must not be replaced by other programs since `OptimalProgram(·)` uses only valid program to update *bestProgram*. On the other hand, if $bestProb(problem) \leq lowerbound$, *bestProgram* will never be updated since no valid programs has log-probability larger than *lowerbound*. In conclusion, synthesis state $\mathbb{S}$ is correct at this time.

For *memoTable*, `OptimalProgram(·)` only updates it at Line 13 of Algorithm 6. Since $\mathbb{S}$ is correct, *bestProgram* must be the most probable program p^+ at this time. Therefore this update to *memoTable* is valid, i.e., *memoTable* remains correct.

For $memoTable_h$, `OptimalProgram(·)` uses *bestProbability* or *lowerbound* to update $memoTable_h$. Since $\mathbb{S}$ is correct, if the most probable program is found, *bestProbability* should be equal to $\mathcal{P}_c[p^+]$, i.e., $bestProb(problem)$; if not, *lowerbound* should be larger than $bestProb(problem)$. In both cases, $memoTable_h$ remains correct after this update. So far, we have discussed all possible cases and thus proved this lemma.

□

Now, we are ready to prove the main theorem.

Theorem A.3 (Theorem 4.3). *Given a PBE task and a topdown prediction model $\mathcal{P}$, let V be the set of all valid programs, the program $program^*$ synthesized by MaxFlash always satisfies (1)* ***validity:*** *$program^* \in V$, (2)* ***optimality:*** *$\forall program \in V, \mathcal{P}[program] \leq \mathcal{P}[program^*]$.*

Proof. Since the initial values of *memoTable* and $memoTable_h$ are both empty, at beginning, *memoTable* and $memoTable_h$ are both correct.

Then by Lemma **??**, $memoTable$, $memoTable_h$ and all synthesis states are all correct during the synthesis process. Therefore, the synthesized program must satisfy all examples and have the largest possible probability. □