Guiding Dynamic Programming 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 $\times 4.107 - \times 2080$ 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

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

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

© 2020 Copyright held by the owner/author(s).
2473-1421/2020/11-ART224
https://doi.org/10.1145/3428292

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 \( \text{Inc}(a) \) is probable, but becomes improbable when forming an improbable combination with other operations, e.g., \( \text{Dec}(\text{Inc}(a)) \), or used together with improbable components, e.g., \( (a/0) \times \text{Inc}(a) \).

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 \( \text{Dec}(?) \), \( \text{Inc}(a) \) 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 \( (a/0) \times ? \) 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 $L^{ex}$:

<table>
<thead>
<tr>
<th>Start symbol</th>
<th>String expr</th>
<th>Integer value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>$N S$</td>
<td>$N Z$</td>
</tr>
<tr>
<td>$N S$</td>
<td>Parameters</td>
<td>$0$</td>
</tr>
<tr>
<td>$N Z$</td>
<td>$+$</td>
<td>$1$</td>
</tr>
<tr>
<td></td>
<td>$\text{CHARAt}$ $N S$ $N Z$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\cdot$</td>
<td></td>
</tr>
</tbody>
</table>

Now, given an input-output example (‘John’, ‘Jonathan’) → ‘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:

$$(+ (\text{CHARAt } FS 0) (+ \cdot LS))$$

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, ($N S N S$), and (CHARAt $N S N Z$), and (2) enumerating possible outputs that the subprograms in the forms could produce. For example, when the form is ($N S N S$), there are 9 possibilities: (‘J’, ‘. Jonathan’), (‘J’, ‘Jonathan’), ···, (‘J. Jonathan’, ‘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 $P_1$ and $P_2$ 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 $P^{ex}$ 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 $P^{ex}$, the probability of a program can be obtained by multiplying the probabilities of each part. For example, the probability of $(+ (\text{CHARAt } FS 0) (+ \cdot LS))$ is 0.01 under model $P^{ex}$.
Guiding Dynamic Programming via Structural Probability for Accelerating Programming by Example

Fig. 1. Decision procedure of PROSE

Fig. 2. Decision procedure of MaxFlash

Table 1. A concrete statistical model $P_{ex}$

<table>
<thead>
<tr>
<th></th>
<th>CHARAt</th>
<th>+</th>
<th>FS</th>
<th>LS</th>
<th>.</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(+, 1)</td>
<td>0.5</td>
<td>0.01</td>
<td>0.09</td>
<td>0</td>
<td>0.4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(+, 2)</td>
<td>0.05</td>
<td>0.5</td>
<td>0.05</td>
<td>0.4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(CHARAt, 1)</td>
<td>0.05</td>
<td>0.05</td>
<td>0.5</td>
<td>0.4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(CHARAt, 2)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.5</td>
<td>0.5</td>
<td></td>
</tr>
</tbody>
</table>

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 $O = \text{Jonnathan}$ is used twice with different contexts: ($\text{CHARAt}, 1$) and (+, 2). As a result, $P_{ex}$ 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 $P_{ex}$, 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 $O = \text{Jonnathan}$ in Figure 1 is split into two different subproblems, marked green in Figure 2, and thus the algorithm could correctly use the probabilities in $P_{ex}$.
**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 searches on improbable 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 $(+ N_5 N_5)$ and lowerbound $L = 0.01$.

1. While synthesizing the first argument, suppose its program is $P_1$, the lowerbound $L$ could be raised to $0.01 \times 0.01 = 0.0001$. 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 $(\text{CharAt } N_5 N_5)$. The synthesizer will then synthesize the first argument, denoted as $P_{1,1}$. At this time, $L$ could be raised again to $0.025 \times 0.5 = 0.1$, which divides the probability that the form of $P_1$ is $\text{CharAt}$, $0.5$, 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 program.

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, $\Sigma$ 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 $s \rightarrow f(x_1, \ldots, x_k)$ which represents nonterminal symbol $s$ can be expanded to a function using operator $f$ and taking symbols $x_1, \ldots, x_k$ 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 $(s, A)$, where:

1. $s$ is a non-terminal symbol in $G$;
2. $A$ is a list of input-output constraints $\{I_i \rightarrow O_i\}_{i=1}^n$, where $I_i$ is an assignment to variables and $O_i$ is a set of valid outputs.

The goal of a PBE subproblem $(s, A)$ is to find a valid program or determine there is no valid program. A valid program to PBE subproblem $(s, A)$ is a program $p$ which is expanded from nonterminal $s$ in $G$ and is consistent with all input-output constraints in $A$, i.e., $\forall (I_i, O_i) \in A$, 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(\cdot)` is implemented.
Algorithm 1: A basic synthesizer based on dynamic programming

**Input:** A PBE task specified by a grammar $G = (N, \Sigma, s_0, R)$ and a set of input-output examples $E$.

**Output:** A valid program $\text{program}^*$, or $\top$ for no valid program.

1. $\text{memoTable} \leftarrow \{\}$;
2. **Function** $\text{ValidProgram}(\text{problem} = (s, A))$:
   3. if $\text{problem} \in \text{memoTable}$ then return $\text{memoTable}[\text{problem}]$;
   4. for $(\text{form}, \text{subproblem}_1, \ldots, \text{subproblem}_k) \in \text{GetAllSchemes}(\text{problem})$ do
      5. $\text{subprogram}_i \leftarrow \text{ValidProgram}(\text{subproblem}_i)$ for each $i \in [1, k]$;
      6. if $\forall i \in [1, k], \text{subprogram}_i \neq \top$ then
         7. result $\leftarrow \text{ConstructProgram}(\text{form}, \text{subprogram}_1, \ldots, \text{subprogram}_k)$;
         8. $\text{memoTable}[\text{problem}] \leftarrow \text{result}$;
      9. return result;
   10. end
   11. $\text{memoTable}[\text{problem}] \leftarrow \top$;
   12. return $\top$;
13. return $\text{ValidProgram}((s_0, E))$;

in the same way as PROSE [Polozov and Gulwani 2015]. For rule $\text{form}$ and each input-output constraint $I_i \rightarrow O_i$ in $\text{problem}$, $\text{GetAllSchemes}(\cdot)$ uses pre-defined rules, named as witness functions, to obtain input-output constraints for subproblems. Function $\text{GetAllSchemes}(\cdot)$ 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 $\text{memoTable}$ which records valid programs for all visited subproblems. Each time when $\text{ValidProgram}(\cdot)$ receives a repetitive subproblem, it accesses $\text{memoTable}$ and directly returns a valid program (Line 3). Otherwise, $\text{ValidProgram}(\cdot)$ enumerates on possible schemes of dividing $\text{problem}$ into subproblems (Line 4). $\text{ValidProgram}(\cdot)$ recursively finds valid subprograms for subproblems (Line 5). If for every subproblem, a subprogram is found, $\text{ValidProgram}(\cdot)$ will merge these subprograms into a valid program to $\text{problem}$ (Line 7) and will return this program (Lines 8 – 9). Otherwise, if no valid program is found after dealing with all schemes, $\text{ValidProgram}(\cdot)$ will return $\top$ (Lines 11 – 12). According to Definition 3.1, $\text{ValidProgram}((s_0, E))$ 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 $M$ is a triple $(C, c_0, \tau)$ where $C$ represents a set of abstracted context, $c_0 \in C$ represents the start context, and $\tau$ 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:

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

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

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

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.

![Fig. 3. An example of topdown contexts. The left figure shows the AST of program \((+ (\text{CharAt FS} 0) (+ '.' LS))\), and the right table lists the topdown contexts for some vertices of this AST.](image)

**Definition 3.4 (TopDown Prediction Model).** Given a set of grammar rule \( R \), a topdown prediction model (abbreviated as TPM) \( P \) 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 \( P_c[p] \) to denote the log-probability of program \( p \) under prediction model \( P \) and context \( c \), and use this value to represent the structural probability.

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

\[
P^{ex}_T[ (+ (\text{CharAt FS} 0) (+ '.' LS)) ] = \log 0.01 \approx -4.61
\]

\[
P^{ex}_T[ (+ (\text{CharAt FS} 0) ('.' LS)) ] = \log 0 = -\infty
\]

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 $\mathcal{P} = \langle C, c_0, \tau, \phi \rangle$.

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

```plaintext
1 memoTable ← {};
2 Function OptimalProgram(problem = (s, A, c)):
3     if problem ∈ memoTable then return memoTable[problem];
4     (bestProgram, bestProbability) ← (τ, −∞);
5     for (form, subproblem$_1$, . . . , subproblem$_k$) ∈ GetAllSchemes(problem) do
6         subprogram$_i$ ← OptimalProgram(subproblem$_i$) for each $i ∈ [1, k]$;
7         if ∀i ∈ [1, k], subprogram$_i$ ≠ τ then
8             candidate ← ConstructProgram(form, subproblem$_1$, . . . , subproblem$_k$);
9             if $\mathcal{P}[candidate]_c > bestProbability$ then
10                (bestProgram, bestProbability) ← (candidate, $\mathcal{P}[candidate]_c$);
11         end
12     memoTable[problem] ← 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 *Subprogram*.

**Definition 3.6 ((Optimization) Subproblem).** Given a grammar $G$ and a TPM $\mathcal{P}$, an optimization subproblem is a triple $(s, A, c)$, where: (1) $s$ is a non-terminal symbol in $G$. (2) $A$ is a list of input-output constraints $I_i \rightarrow O_i |_{i=1}^n$, 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 $(s, A, c)$ 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., $\forall (I_i, O_i) ∈ A$, the output of $p^*$ on $I_i$ is a member of the set $O_i$. (2) *optimality*: for any valid program $p$ to subproblem $(s, A, c)$, the probability of $p^*$ is always no smaller than $p$, i.e., $\mathcal{P}[p^*] ≥ \mathcal{P}[p]$. If there is no valid program, the optimal program of $(s, A, c)$ is defined as $\tau$.

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 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
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 ⊤ 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 detailly 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 ⊤, 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 ⊤, bestProb(problem) is defined as −∞. 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, \( \text{heuristic}(\text{problem}) \geq \text{bestProb}(\text{problem}) \).

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 heuristic0. In Subsection 4.1, we will elaborate on the heuristic function used in MaxFlash, which is improved from heuristic0.

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 \( \text{heuristic}(@\text{subp}) \) is an upper bound for the log-probability of the jth subprogram.

• Pruning. Algorithm 3 will directly return ⊤ 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 \( \log \phi(c, form) + \sum_{\text{subp} \in \text{subproblems}} \text{heuristic}(\text{subp}) \), 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 $G = (N, Σ, s₀, R)$, a set of input-output examples $E$, a TPM $P = (C, c₀, r, ϕ)$, a step size $size$.

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

1 $memoTable ← \{\}$
2 Function GetBound(problem = $(s, A, c)$, lowerbound, form, scheme, $i$):
3 $\text{(subproblem}_1, \ldots, \text{subproblem}_k) ← \text{scheme}$
4 Return $\text{lowerbound} − \log ϕ(c, \text{form}) - \sum_{j \in [1, k], j \neq i} \text{heuristic(subproblem}_j)$
5 Function SortSchemes(problem = $(s, A, c)$, allSchemes = $\{(form_i, subproblems_i)\}_{i=1}^t$):
6 Sort allSchemes in the decreasing order of $\log ϕ(c, form_i) + \sum_{subp \in subproblems_i} \text{heuristic(subp)}$.
7 Return allSchemes;
8 Function OptimalProgram(problem = $(s, A, c)$, lowerbound):
9 Reuse results in memoTable. This part is the same as Lines 5 − 8 in Algorithm 3.
10 if heuristic(problem) < lowerbound then Return $T$;
11 $\text{(bestProgram, bestProbability)} ← (T, −\infty)$;
12 for scheme = $(form, subproblems) \in \text{SortSchemes(problem, GetAllSchemes(problem))}$ do
13 $\text{Deal with scheme. This part is the same with Lines 12 − 17 in Algorithm 3.}$
14 end
15 Store and return the result. This part is the same as Lines 19 − 20 in Algorithm 3.
16 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 $problem = (s, A, c)$, $heuristic_0(problem)$ 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 $problem_1 = (s, A, c), problem_2 = (s, A', c)$ where $A' \subseteq A$, 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 $A'$ is a subset of $A$, $program$ must be valid to $problem_2$. Therefore $bestProb(problem_1) = P_c[program] ≤ bestProb(problem_2)$

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 \( P = (C, c_0, r, \phi) \), a set \( E = \{(I_i, O_i)\}_{i=1}^N \) of input-output examples and a step-size \( \text{step} \).
Output: The target program \( \text{program}^* \).

1. \( \text{counterExamples} \leftarrow \emptyset \);
2. \( \text{lowerBound} \leftarrow 0 \);
3. while True do
   4.     while \( \text{OptimalProgram} (\text{counterExamples}, \text{lowerBound}) = \top \) do
   5.         \( \text{lowerBound} \leftarrow \text{lowerBound} - \text{step} \);
   6.     end
   7.     \( \text{program} \leftarrow \text{OptimalProgram} (\text{counterExamples}, \text{lowerBound}) \);
   8.     \( \text{counterExample} \leftarrow \text{GetCounterExample} (\text{program}, E) \);
   9.     if \( \text{counterExample} = \top \) then return \( \text{program} \);
   10.     \( \text{counterExamples}.\text{INSERT} (\text{counterExample}) \);
4. 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 \( \text{counterExamples} \) is maintained. In each turn, the inner loop finds a program \( \text{program} \) that is consistent with all examples in \( \text{counterExamples} \) (Line 4). After that, MaxFlash calls \( \text{GetCounterExample} (\cdot) \) to verify whether \( \text{program} \) is consistent with all other examples in \( E \) (Line 8). If true, it will return \( \text{program} \) as the result (Line 9). Otherwise, it will add a counter-example to \( \text{counterExamples} \) (Line 12) and continue to synthesize for the new set of examples.

The trick in Algorithm 5 is that \( \text{lowerBound} \) 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 $G = (N, \Sigma, s_0, R)$, a set of input-output examples $E$, a TPM $\mathcal{P} = (C, c_0, \tau, \phi)$, a step size size.

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

1. $\text{memoTable} \leftarrow \{}$;
2. $\text{memoTable}_h \leftarrow \{}$;
3. Function $\text{Heuristic}(\text{problem} = (s, A, c))$:
   1. $\text{heuristicValue} \leftarrow \text{heuristic}_0(\text{problem})$;
   2. for $A' \in \text{Prefix}(A)$ do
      1. if $(s, A', c) \in \text{memoTable}_h$ then $\text{heuristicValue} \leftarrow \min (\text{heuristicValue}, \text{memoTable}_h[(s, A', c)])$;
   3. return $\text{heuristicValue}$;

4. Function $\text{OptimalProgram}(\text{problem} = (s, A, c), \text{lowerbound})$:
   1. Reuse results in $\text{memoTable}$ and check the heuristic value. The same as Lines 9 – 10 in Algorithm 4.
   2. Solve subproblem $\text{problem}$. This part is the same as Lines 11 – 14 in Algorithm 4.
   3. if $\text{bestProgram} \neq \top$ then
      1. $\text{memoTable}[\text{problem}] \leftarrow \text{bestProgram}$;
      2. $\text{memoTable}_h[\text{problem}] \leftarrow \text{bestProbability}$;
   4. else
      1. $\text{memoTable}_h[\text{problem}] \leftarrow \text{lowerbound}$;
   5. if $\text{bestProbability} \geq \text{lowerBound}$ then return $\text{bestProgram}$ else return $\top$;

6. 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 $\text{heuristic}$ used in MaxFlash, which is improved from $\text{heuristic}_0$, the basic heuristic function discussed in Section 3.4. Algorithm 6 shows a synthesizer with the full-version heuristic function, in which $\text{heuristic}$ is implemented as $\text{Heuristic}(\cdot)$.

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 $\text{OptimalProgram}(\cdot)$ fails, lowerbound will be a safe heuristic value for problem. Similarly, for a successful result, $\text{bestProbability}$ is safe, since it is exactly equal to $\text{bestProb}(\text{problem})$. In Algorithm 6, all these values are stored in a new memoization table $\text{memoTable}_h$ (Lines 14, 16) and are used to improve $\text{heuristic}_0(\text{problem})$.

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 $(s, A, c)$, a large number of subproblems $(s, A', c)$, where $A'$ is a prefix of $A$, have been dealt with. Secondly, according to Lemma 3.9, if $A'$ is a prefix of $A$, $\text{bestProb}(s, A, c)$ must be at most $\text{bestProb}(s, A', c)$. Therefore $\text{memoTable}_h[(s, A', c)]$ is a valid heuristic value not only for $(s, A', c)$ but also for $(s, A, c)$. In this way, a tighter heuristic function is obtained:

$$\text{heuristic}(s, A, c) = \min(\text{heuristic}_0(s, A, c), \min_{A' \text{ is a prefix of } A} \text{memoTable}_h[(s, A', c)])$$
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:

\[ problem_1 = (N_S, \{('John', 'Jonathan') \rightarrow 'J.Jonathan'\}, \top) \]
\[ problem_2 = (N_S, \{('John', 'Jonathan') \rightarrow 'J.Jonathan', ('John', 'Smith') \rightarrow 'J.Smith'\}, \top) \]

Now, suppose \text{OptimalProgram}(problem\_1, -4) and \text{OptimalProgram}(problem\_2, -3) are invoked in order. Since the optimal program to \textit{problem}_1 is \((+ (\text{CharAt} FS 0) \ (\text{\textquoteleft}.'\text{\textquoteleft} LS))\) of which the log-probability is larger than \(-4\), the first invocation fails. Therefore \textit{memoTable}[\text{\textit{problem}}\_1] is set to \(-4\). During the second invocation, since the constraints of \textit{problem}_1 is a prefix of the constraints of \textit{problem}_2, \text{Heuristic}(\textit{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 \((s, \mathcal{A}, c)\), \((s, \mathcal{A}', c)\) where \(\mathcal{A}' \subset \mathcal{A}\):

1. The first observation is a corollary to Lemma 3.9: If the optimal program to \((s, \mathcal{A}, c)\) is a valid program to \((s, \mathcal{A}', c)\), it must also be the optimal program to \((s, \mathcal{A}, c)\).
2. The second observation is about the time cost: solving \((s, \mathcal{A}, c)\) is usually more time-consuming than solving \((s, \mathcal{A}', c)\), since \((s, \mathcal{A}, c)\) 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 \((s, \mathcal{A}', c)\) has a great chance to be valid for \((s, \mathcal{A}, c)\), since it is the most probable program under a prediction model based on structural probability.

Therefore, for a subproblem \textit{problem} with multiple input-output constraints, MaxFlash will firstly solve another subproblem \textit{problem}_f with fewer constraints and checks whether the found program is valid for \textit{problem}. If the found program is, the optimal program to \textit{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 \textit{problem}_f is usually easier than solving \textit{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 \textit{problem} as \textit{problem}_f (Lines 6 – 7). Such a choice has two advantages: (1) The constraint set of \textit{problem}_f is one of the largest proper subsets of the constraint set of \textit{problem}. Since almost all constraints are reserved, the optimal program to \textit{problem}_f is likely to be valid to \textit{problem}. (2) According to CEGIS framework, a lot of results on the first \(|\mathcal{A}| - 1\) examples are cached in the previous synthesis turn. Therefore the time cost of solving \textit{problem}_f should not be large.

Example 4.2. We continue to discuss subproblems \textit{problem}_1, \textit{problem}_2 introduced in Example 4.1. Consider the procedure of \text{OptimalProgram}(\textit{problem}_2, -5):

1. At first, the last constraint of \textit{problem}_2 is removed, and the result is \textit{problem}_1 (Line 6).
2. Then, \text{OptimalProgram}(\textit{problem}_1, -5) is invoked. After finishing this invocation, the optimal program \textit{candidate} = \((+ (\text{CharAt} FS 0) \ (\text{\textquoteleft}.'\text{\textquoteleft} LS))\) to \textit{problem}_1 is obtained (Line 7).
3. Since \textit{candidate} is a valid program to \textit{problem}_1, it must also be the optimal program to \textit{problem}_2. Therefore, \text{OptimalProgram}(\textit{problem}_2, -5) returns \textit{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 $G = (N, \Sigma, s_0, R)$, a set of input-output examples $E$, a TPM $P = (C, c_0, \tau, \phi)$, a step size $\ell$.

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

1. $\text{memoTable} \leftarrow \{\}$;
2. $\text{memoTable}_h \leftarrow \{\}$;
3. Function OptimalProgram($\text{problem} = (s, A, c), \text{lowerbound}$):
   4. Reuse results in memoTable and check the heuristic value. The same as Lines 9 – 10 in Algorithm 4.
   5. if $|A| > 1$ then
      6. $A' \leftarrow A.\text{RemoveLast}()$;
      7. candidate $\leftarrow$ OptimalProgram($s, A', c$, lowerbound);
      8. if candidate $= \top$ then Return $\top$;
      9. if candidate is valid to problem then
         10. memoTable[$\text{problem}$] $\leftarrow$ candidate;
         11. $\text{memoTable}_h[\text{problem}] \leftarrow P_c[\text{candidate}]$;
         12. return candidate;
   end
   13. Solve problem and update storages. This part is the same as Lines 11 – 18 in Algorithm 6.
14. The outer framework. This part is the same as Algorithm 5.

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

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 $M_d = \{C_d, c_{0,d}, \tau_d\}$, 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., $c_{0,d} = (\top)^d$.
- $\tau_d(c, r, i) = (e_2, \ldots, e_d, (r, i))$, where $e_i$ represents the $i$th element in context $c$.

In our implementation, we set $d$ to 1 by default. At this time, the context model $M_1$ 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 $T$, for each program $p \in T$ 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.,

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

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 $M_d$ 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 $(s, A)$ in PROSE, NGDS uses a neural network to score each possible form according to $A$, 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 \( \mathcal{D}_{S} \) collected by Lee et al. [2018]. \( \mathcal{D}_{S} \) 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 \( \mathcal{D}_{S} \), 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 \( \mathcal{D}_{M} \) collected by Wang et al. [2018b]. \( \mathcal{D}_{M} \) 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 \( \mathcal{D}_{M} \), 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 \( \mathcal{D}_{S} \), we train MaxFlash via leave-one-out cross-validation, i.e., for each benchmark in \( \mathcal{D}_{S} \), the TPM used by MaxFlash is trained from all other benchmarks in \( \mathcal{D}_{S} \).

- **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 \( \mathcal{D}_{S} \), 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 \( \mathcal{D}_{S} \) 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.
split $\mathcal{D}_S$ 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 $\mathcal{D}_S$ by running MaxFlash with a trivial prediction model: all rules always have the same probability. This comparison uses all 205 benchmarks in $\mathcal{D}_S$ 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 PROSEM). 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 $\mathcal{D}_S$ 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 $\mathcal{D}_M$, 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.
Comparing with \textit{Eusolver} and \textit{Euphony} over the string manipulation domain, \textit{MaxFlash} not only solves many more benchmarks but also achieves a significant speed-up.

Comparing with \textit{PROSE}, a state-of-the-art framework for dynamic-programming-based synthesizer, \textit{MaxFlash} performs significantly better when both \textit{MaxFlash} and \textit{PROSE} are instantiated on the same grammar over both domains. However, the performance of \textit{PROSE} can be greatly improved by a well-designed DSL: \textit{TText} solved more benchmarks than \textit{MaxFlash} whenever the time limit is 500 milliseconds or 5 minutes. We suspect that \textit{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 $D_0$ much simpler. For example, on $\text{univ}_v$.4.s1, a benchmark on which both \textit{PROSE} and \textit{MaxFlash} failed but \textit{TText} succeeded, \textit{TText} uses regex `\texttt{\textbackslash egexPair(\texttt{,} or \texttt{and}) + \texttt{Upper Case}, \texttt{e}')}" to capture two cases, starting with `\texttt{,}', and starting with `\texttt{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 \textit{MaxFlash} on the DSL used by \textit{TText} may potentially achieve a better performance. Please note that even though \textit{TText} is built on a much more powerful DSL, \textit{MaxFlash} still achieves an average speed-up equal to $\times 4.107$ on benchmarks solved by both \textit{TText} and \textit{MaxFlash}.

Comparing with \textit{NGDS}, \textit{MaxFlash} significantly outperforms it over the string manipulation domain. \textit{MaxFlash} has two critical differences against \textit{NGDS}: (1) The default prediction model $M_1$ used by \textit{MaxFlash} is much more lightweight than the neural network used by \textit{NGDS}, allowing \textit{MaxFlash} to explore more subproblems within the same time limit; (2) Comparing with the search algorithm used by \textit{NGDS}, \textit{MaxFlash} further utilizes heuristic functions and iterative deepening search, making \textit{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 \textit{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 \textit{NGDS} is greatly improved: \textit{NGDS} finishes 41(50\%) benchmarks within 5 minutes. But even comparing with the new results, \textit{MaxFlash} still solves 17(21\%) more benchmarks with an average speed-up equal to $\times 554.6$ on benchmarks solved by both \textit{MaxFlash} and \textit{NGDS}.

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

\begin{table}[h]
\centering
\small
\begin{tabular}{|l|l|l|l|l|l|l|}
\hline
Baseline & $D$ & $|D_{\text{test}}|$ & \#Solved in 500ms & \#Solved in 5min & Speed-up & Memory Cost \\
\hline
\textit{Eusolver} & $D_0$ & 205 & 143(70\%) & 63(31\%) & 174(85\%) & 111(54\%) & $\times 49.13 \times 126.2$ & 35.63\% \\
\textit{PROSE} & $D_0$ & 123(60\%) & 74(40\%) & 106(53\%) & 190(96\%) & $\times 5.097 \times 8.670$ & 66.53\% \\
\textit{CVC4} & $D_0$ & 143(70\%) & 154(75\%) & 174(85\%) & 197(96\%) & $\times 4.107 \times 1.148$ & 17.05\% \\
\textit{TText} & $D_0$ & 130(63\%) & 115(56\%) & 132(64\%) & 125(61\%) & $\times 23.34 \times 95.98$ & 3.883\% \\
\hline
\textit{Euphony} & $D_{ID}$ & 82 & 43(52\%) & 9(11\%) & 58(71\%) & 23(28\%) & $\times 38.16 \times 55.30$ & 37.26\% \\
\textit{NGDS} & $D_{ID}$ & 39 & 0(0\%) & 18(22\%) & 1043 & $\times 1043 \times 23.10$ & \\
\textit{PROSE} & $D_{ID}$ & 34(87\%) & 0(0\%) & 33(85\%) & 38(97\%) & $\times 2080 \times 2080$ & 8.137\% \\
\textit{Atlas} & $D_{ID}$ & 29(74\%) & 30(100\%) & 38(97\%) & $\times 15.50 \times 3.663$ & 3.776\% \\
\hline
\end{tabular}
\caption{The results of comparing \textit{MaxFlash} with baselines.}
\end{table}
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 DS. The reason is that two extra reuse mechanisms in MaxFlash involve multi-example subproblems, but every benchmark in DM contains only one example. Therefore, both of them are less effective in DM than DS.

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.s1, 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; Atlas 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: $M_0$, $M_1$, and $M_2$ (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 $M_0$ 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 $M_0$ model, and then use it to guide MaxFlash.
- **Dynamically** (denoted as DeepCoder$^D$). For each benchmark, we invoke MaxFlash with context model $M_0$ before determining the prediction model. Then for each subproblem $(s, h, e)$ visited by MaxFlash, we invoke DeepCoder to generate a prediction specifically for it from $h$. 

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 DeepCoderS and DeepCoderD on dataset Ds.

**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 M1). The results are summarized in Table 3 while more details are drawn as Figure 4c and Figure 4d.

<table>
<thead>
<tr>
<th>Prediction Model</th>
<th>Ds</th>
<th></th>
<th>Dm</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#Solved</td>
<td>Average Speed-up</td>
<td>#Solved</td>
<td>Average Speed-up</td>
</tr>
<tr>
<td></td>
<td>500ms</td>
<td>5min</td>
<td>All</td>
<td>Hard</td>
</tr>
<tr>
<td>M0</td>
<td>134</td>
<td>164</td>
<td>x1.618</td>
<td>x6.124</td>
</tr>
<tr>
<td>M1 (Default)</td>
<td>143</td>
<td>174</td>
<td>x1.000</td>
<td>x1.000</td>
</tr>
<tr>
<td>M2</td>
<td>139</td>
<td>171</td>
<td>x2.571</td>
<td>x2.975</td>
</tr>
<tr>
<td>DeepCoderS</td>
<td>140</td>
<td>163</td>
<td>x7.561</td>
<td>x2.027</td>
</tr>
<tr>
<td>DeepCoderD</td>
<td>98</td>
<td>158</td>
<td>x21.99</td>
<td>x20.51</td>
</tr>
</tbody>
</table>

M1 performs best among all three topdown context models M1 outperforms M0 because the context in M1 is more refined which makes the prediction model more precise. M1 outperforms M2 because of the following two reasons:

1. The training set is small and the training method is straight-forward, therefore M2 may overfit on the training set, since M2 is more complex than M1.
2. The context set of M2 is larger than M1. Thus there are more subproblems for M2 than M1. 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 M1 performs better than both DeepCoderS and DeepCoderD: DeepCoderD performs the worst, while the performance of DeepCoderS is closer to that of M1. This difference is because DeepCoderS and DeepCoderD use the neural network differently: DeepCoderS invokes the network only once, while DeepCoderD 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 DeepCoderS and DeepCoderD 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:
randomly divide the dataset into \( n \) subsets, (2) for each subset, the prediction model used by \textit{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 \textit{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 \textit{MaxFlash}, especially for hard benchmarks. This result shows that \textit{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\%, \textit{MaxFlash} only becomes 9.11\% and 0.85\% slower in average on \( \mathcal{D}_S \) and \( \mathcal{D}_M \) respectively. Just as shown in Section 6.2, comparing to the advantages of \textit{MaxFlash} to other baseline solvers, such a loss is much smaller. This result demonstrates that even under the lack of training data, \textit{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 \textit{MaxFlash}.

Here, we further implement two weakened solvers \textit{MaxFlash}_H, \textit{MaxFlash}_M, which disable reusing through heuristic function, reusing results with fewer constraints from \textit{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 \textit{MaxFlash}). The results are summarized in Table 4 while more details are drawn as Figure 4f.

<table>
<thead>
<tr>
<th>Disabled Module</th>
<th>( \mathcal{D}_S )</th>
<th>( \mathcal{D}_M )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#Solved</td>
<td>Average Speed-up</td>
</tr>
<tr>
<td></td>
<td>500ms</td>
<td>5min</td>
</tr>
<tr>
<td>Heuristic Function</td>
<td>96</td>
<td>134</td>
</tr>
<tr>
<td>Fewer Constraints</td>
<td>114</td>
<td>162</td>
</tr>
</tbody>
</table>

As shown in Table 4, over the string manipulation domain, both two reuse mechanisms boost the speed of \textit{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 \textit{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 \( \mathcal{D}_M \) contains only a single example, and thus all optimizations for multi-example subproblems become useless on \( \mathcal{D}_M \). 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 2015].
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] 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 toptdown 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 $M_n$ (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


A  APPENDIX

To prove Theorem 4.3, we involve the concept of Synthesis State. A synthesis state $S$ 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 $p(S)$.

We define the property correct for synthesis state $S$, memoization table memoTable and auxiliary memoization table memoTable$_h$:

- A synthesis state $S = (\text{problem, lowerbound, memoTable, memoTable}_h)$ is correct $\iff$ if the log-probability of the most probable program program$^*$ of problem is larger than lowerbound, $p(S) = \text{program}^*$, otherwise $p(S) = \top$.
- 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 $= (s, A, c)$. According to Section 4.1, heuristic(problem) is equal to $\min(\text{heuristic}_0((s, A, c)), \min_{A'} \text{prefix of } A \text{ memoTable}_h((s, A', c)))$. By correctness of memoTable$_h$ and Lemma 3.9, for any prefix $A'$ of $A$:

$$\text{memoTable}_h((s, A', c)) \geq \text{bestProb}(s, A', c) \geq \text{bestProb}(s, A, c)$$

Since heuristic$_0$ is a valid heuristic function, we get the validity of heuristic, i.e., bestProb(problem) $\leq$ 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 Optimal-Program are always correct.

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

- memoTable and memoTable$_h$ will always be correct during OptimalProgram(problem, lowerbound).
- Synthesis state $S$ 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) $\geq$ 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, $S$, 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 Algorithm 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 $\text{bestProb}(\text{problem}) \leq \text{bestProb}(\text{problem_l}) \leq \text{lowerbound}$. Therefore, it’s correct for $\text{OptimalProgram}(\cdot)$ 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, $\mathcal{P}_c[candidate] \leq \text{bestProb}(\text{problem}) \leq \text{bestProb}(\text{problem_l}) = \mathcal{P}_c[candidate]$, so `candidate` is also the most probable program to `problem`. Therefore it’s correct for $\text{OptimalProgram}(\cdot)$ 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 $\text{OptimalProgram}(\cdot)$ 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 $\text{bestProb}(\text{problem}) > \text{lowerbound}$, the optimal program $p^+$ must be used to update `bestProgram` during $\text{OptimalProgram}(\text{problem,lowerbound})$. 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 $\text{bestProb}(\text{problem}) > \text{lowerbound}$, `bestProgram` must be updated to $p^+$ at some time, and it must not be replaced by other programs since $\text{OptimalProgram}(\cdot)$ uses only valid program to update `bestProgram`. On the other hand, if $\text{bestProb}(\text{problem}) \leq \text{lowerbound}$, `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.

\[\square\]

**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 $\text{program}^*$ synthesized by MaxFlash always satisfies (1) validity: $\text{program}^* \in V$, (2) optimality: $\forall \text{program} \in V, \mathcal{P}[\text{program}] \leq \mathcal{P}[\text{program}^*]$.

**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 7?, `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. \[\square\]