# A Probabilistic Delta Debugging Approach for Abstract Syntax Trees

Guancheng Wang, Yiqian Wu, Qihao Zhu, Yingfei Xiong*, Xin Zhang and Lu Zhang
Key Laboratory of High Confidence Software Technologies, Ministry of Education (Peking University);
School of Computer Science, Peking University, 100871, P. R. China
{guancheng.wang,wuyiqian,zhuqh,xiongyf,xin,zhanglucs}@pku.edu.cn

***Abstract*—Delta debugging provides an efficient and systematic approach to isolate and identify a minimal subsequence that exhibit a specific property. A notable trend in the development of delta debugging is to address data with domain-specific structures, such as programs. However, the efficiency and effectiveness of domain-specific delta debugging algorithms still present challenges. Probabilistic delta debugging (ProbDD) enhances the ddmin algorithm, which forms the foundation of most domain-specific delta debugging approaches, by incorporating a probabilistic model. By replacing the ddmin component with ProbDD, algorithms relying on ddmin can achieve superior performance. Meanwhile, domain-specific delta debugging techniques, such as Perses, have been designed to cater to the abstract syntax tree (AST) and follow predefined sequences of attempts to minimize programs. These techniques benefit from the use of AST-based transformations, enabling them to achieve even smaller results efficiently. However, we observe that ProbDD assumes independence between elements, which may limit their performance in capturing syntactic relationships. Additionally, domain-specific approaches such as Perses rely on a predefined sequence of attempts the removal of the element and fail to utilize the information from existing test results.**

**In this paper, we propose T-PDD, a novel approach that addresses these limitations. T-PDD leverages the AST to construct a probabilistic model, both utilizing historical test results and capturing syntactic relationships to estimate the probabilities of elements being retained in the result. It selects a set of elements that maximizes the gain for the next test based on the model and updates the model using the test results.**

**In our evaluation, we assess our approach on 107 real-world subjects. The results demonstrate an average improvement of 26.95% in processing time and a 3.4x reduction in result size compared to Perses in the best-case scenario.**

***Index Terms*—Delta Debugging, Probabilistic Model, Abstract Syntax Tree**

## I. Introduction

Software bugs are an inevitable part of the software development process, causing disruptions, failures, and costly delays. Efficient bug localization and debugging techniques are crucial for improving software quality and reducing maintenance efforts. One such technique that has gained significant attention is delta debugging. Delta debugging [1]–[3] is a systematic and automated approach for isolating the root cause of a bug by iteratively reducing a complex input or program to a minimal, reproducible test case.

The concept of delta debugging was first introduced by Zeller and Hildebrandt in 1999 [1] as a method to simplify and narrow down inputs that trigger failures. Their pioneering work led to the development of the ddmin algorithm [3], which has since become the foundation for a range of delta debugging approaches and has found widespread application in various domains, including compiler debugging [3]–[5], regression fault localization [1], isolating the cause-effect chain of a failure [2], [6], and debloating software [7]–[9] to reduce the size of a program while keeping certain desired functionalities [10]. Delta debugging approaches operate on the principle that the presence of redundant or irrelevant components in a test case makes it harder to identify the root cause of a failure. By selectively removing or modifying these components, delta debugging approaches iteratively simplify the test case while preserving its ability to reproduce the bug.

A significant trend in the development of delta debugging techniques is to address data with domain-specific structures, such as programs. However, despite these advancements, the efficiency and effectiveness of domain-specific delta debugging algorithms still present challenges. One issue highlighted in [4] is that many domain-specific approaches, particularly those targeting program inputs, often generate objects that do not conform to the syntactic structure during the delta debugging process. This can result in wasted time and effort, as the produced objects may not be valid or executable programs.

ProbDD, proposed by Wang et al. [11], stands as the state-of-the-art delta debugging algorithm in the field. It introduces a probabilistic model to estimate the likelihood of each element being retained in the resulting output. In each iteration, ProbDD strategically selects a subset of elements that maximizes the expected gain in the subsequent test, leveraging the information provided by the probabilistic model. Subsequently, it verifies if the desired property is preserved within this subset and updates the probabilistic model based on the test outcome. In contrast to ddmin, which follows a predefined strategy for element removal from the original object, ProbDD enhances the testing process by incorporating insights from historical test results. By substituting the ddmin component with ProbDD, approaches that rely on ddmin can achieve superior performance. Experimental results reported in ProbDD [11] demonstrate significant time savings and reduced program sizes.

Meanwhile, there have been notable advancements in domain-specific delta debugging approaches, particularly in the domain of programs. These approaches such as Perses [4],

*Corresponding author.

focus on designing transformations based on AST. These transformations are tailored to ensure syntactic validity and offer the potential for further size reduction in the results and improved efficiency.

We observe that domain-specific approaches like Perses [4], despite their effectiveness, also rely on a predefined sequence of attempts to apply defined transformations to the original object. Additionally, probabilistic approaches such as ProbDD assume independence between elements, which may limit their performance in capturing syntactic relationships. To overcome these limitations, we propose a novel probabilistic delta debugging algorithm called T-PDD. T-PDD leverages a Bayesian network constructed from the AST, utilizing both existing test results and capturing syntactic relationships among elements to estimate the probability of each element being retained in the result. During each iteration, T-PDD selects a subset of elements that maximizes the expected gain in the subsequent test, guided by the probabilistic model. It then verifies if the desired property is preserved within this subset. Furthermore, T-PDD updates the probabilistic model based on the testing result, refining its estimation of element probabilities. The construction of the probabilistic model from the AST also allows T-PDD to perform probabilistic inference efficiently, as the structural complexity of the AST is relatively simple. As a result, we anticipate that T-PDD will surpass existing domain-specific delta debugging approaches in terms of both reducing the size of the produced result and improving time efficiency.

We conducted a comprehensive evaluation of T-PDD using a dataset of 107 subjects. Our evaluation dataset is larger than that of all recent publications on delta debugging at top venues, to the best of our knowledge [4], [10]–[13]. This dataset included 20 widely adopted C subjects from previous work [4], [11], [12]. In addition, we curated a diverse dataset of 87 subjects consisting of both C and Rust programs for evaluating T-PDD. This dataset allows us to thoroughly assess the effectiveness and efficiency of T-PDD. Our experimental results clearly demonstrate that T-PDD significantly improves the efficiency of Perses, a representative modern delta debugging approach. For example, in a typical case, Perses took up to 7 hours to reduce a C program with 371,277 tokens, whereas T-PDD accomplished the same reduction in just 1.5 hours. On average, T-PDD achieved the same size of produced results as Perses, while reducing the time consumption by 26.95%.

In summary, this paper presents the following key contributions:

1) We introduce a novel probabilistic model for the delta debugging algorithm, utilizing a tree-structured Bayesian network that incorporates the AST.
2) We curate a comprehensive dataset comprising 87 subjects specifically designed for evaluating our proposed approach. This dataset enables a thorough assessment of the effectiveness and efficiency of T-PDD.
3) We conduct extensive evaluations of T-PDD using both existing datasets and our collected dataset, encompassing a total of 107 subjects. The experimental results demonstrate that T-PDD significantly improves the efficiency of the representative delta debugging approach, while also producing results that are 3.4 times smaller in the best case.

## II. Motivating Example

We utilize a program simplification example to demonstrate the functionality of the domain-specific delta debugging approach. Among the available approaches, we choose Perses as the representative technique because it is the state-of-the-art approach for context-free languages, designed for general purposes. The code snippet presented in Listing 1 showcases the function that requires simplification. For the sake of clarity, we omit the definitions of the invoked functions. In this scenario, we assume that the while-loop has the potential to trigger a compiler crash bug. Furthermore, we assume that any valid program containing the `while` keyword triggers the aforementioned bug.

In this example, Perses operates on the AST depicted in Figure 1. The algorithm Perses starts by initializing a priority queue (Q) to maintain the nodes awaiting processing. In each iteration, Perses selects the node with the highest number of tokens from Q for further processing. Perses also distinguishes Kleene-Star nodes from regular nodes. A Kleene-star node has a list of children with variable length, such as a list of parameters in a function definition. When such a node is encountered, Perses applies the ddmin algorithm to attempt to remove its children. In the context of our paper, nodes like Kleene-Star are referred to as *quantifier nodes*, while others are *regular nodes*. For regular nodes, Perses first attempts to remove the node itself. If the test fails after removing the node, Perses proceeds to search for any descendant node that can replace the processed node without violating the syntax. After processing a node, its children are added to Q for future processing. A new iteration begins when Q becomes empty. The process continues until there are no nodes that can be removed within a single iteration.

Perses attempts to remove nodes in Figure 1 based on a sequence of `(1.TRANSUNIT, 2.COMPOUNDSTMT, 3.*, 4.WHILE_STMT, 5.INIT_DECL, 6.COMPOUNDSTMT, 8.*, 7.POST_EXP, ... ...)` in the first iteration. When processing regular nodes, i.e., all nodes except for `3.*` and `8.*` in our example, Perses tries to remove the node itself first. If the test fails after removing the node, Perses proceeds to search for any descendant node that can replace the processed node without violating the syntax. In our example, the node `2.COMPOUNDSTMT` and `4.WHILE_STMT` can be replaced by their descendant node `6.COMPOUNDSTMT`, respectively. When processing quantifier nodes, i.e., `3.*` and `8.*`, Perses utilizes ddmin to minimize the sequence of their children, respectively. In this way, the child of *8.** can be successfully removed in this iteration after processing all front nodes in the predefined sequence. In the second iteration, Perses follows a sequence of `(1.TRANSUNIT, 2.COMPOUNDSTMT, 3.*, 5.INIT_DECL, 4.WHILE_STMT, ... ...)` to attempt the node removal. In this iteration, the node

5.INIT_DECL can be successfully removed after processing the node 1.TRANSUNIT, 2.COMPOUNDSTMT, and 3.*. In the last iteration, Perses follows a sequence of (1.TRANSUNIT, 2.COMPOUNDSTMT, 3.*, 4.WHILE_STMT, ... ...) to attempt the node removal. As no more nodes can be removed in this iteration, Perses terminates the process.

From the above process, it is evident that Perses adheres to the predefined sequences for node removal and does not learn from historical test results. For example, the node 4.WHILE_STMT has been unsuccessfully attempted for removal multiple times in the three ddmin processes across all three iterations. If the simplification process were to learn from historical test results, as demonstrated later in Figure 3, the node 4.WHILE_STMT would have been attempted for removal twice, resulting in its probability of retention in the final result given that its parent reaches 1.0. While ProbDD utilizes a probabilistic model to determine the subsequence tested in the next iteration based on historical test results, it fails to address the aforementioned limitations in the following aspects.

On one hand, ProbDD assumes independence between elements, which can be problematic when directly applying it to AST leaf nodes. This approach considers a sequence of AST leaf nodes as input and may lead to the removal of nodes that violate the program's syntax. For instance, ProbDD uses two independent Bernoulli random variables to indicate whether the child nodes of 2.COMPOUNDSTMT, namely { and }, should be included in the final result or not. However, if the node { should not be retained in the final result, so should the node }. This indicates that these two nodes are dependent, and both nodes should be either removed or retained together. This highlights a limitation of ProbDD when dealing with interdependent nodes in the AST structure.

On the other hand, Perses employs the ddmin algorithm to attempt removing children of quantifier nodes such as 3.*. The ddmin algorithm is repeatedly applied in each iteration, trying different combinations of child removal until eventually attempting to remove each individual child. As an advanced approach to ddmin, it is possible to optimize Perses by substituting the ddmin algorithm with ProbDD. While ProbDD shows potential in leveraging a probabilistic model to guide the search process based on historical test results, similar to ddmin, it is also repeatedly applied to address the 3.* node in each iteration. In conclusion, Perses retains its predefined sequence of removal attempts across different iterations, regardless of whether ddmin or ProbDD is employed.

In contrast, our approach, T-PDD, takes a different approach by constructing a probabilistic model directly from the AST. The structural simplicity of the AST enables T-PDD to efficiently perform probabilistic inference and effectively guide the search process. By leveraging this lightweight probabilistic model, T-PDD is able to achieve efficient program simplification. It utilizes existing test results and captures the relationships among elements, resulting in improved efficiency and effectiveness in minimizing programs.

Listing 1: Example program to be simplified

```
1 int main() {
2   int* g_1 = safe_arr(config_1(),
      ↪ config_2());
3   while (1) {
4     g_1[0] = safe_val(config());
5   }
6   return 0;
7 }
```

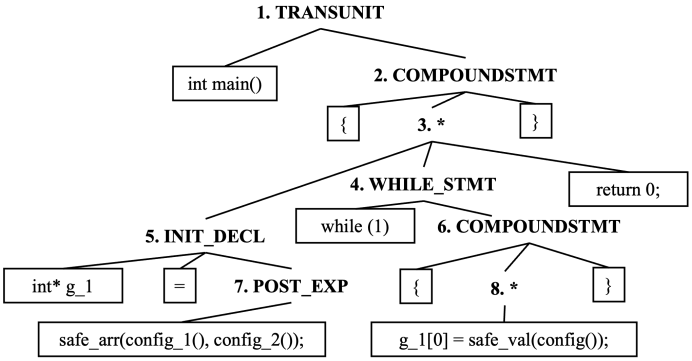



Fig. 1: The parse tree of Listing 1, simpilified for clarity.

Figure 2 illustrates our model constructed from the AST shown in Figure 1, following the steps in Section III-A2. Each node in Figure 2 represents a random variable that indicates whether the corresponding node in Figure 1 is included in the final result given its parent. To enhance clarity, we have omitted the annotations in this figure. Figure 3 demonstrates the steps of our algorithm for simplifying the program presented in Listing 1. The labels $n_1$ through $n_8$ indicate that the nodes are sequentially numbered from 1 to 8 in Figure 1. In Figure 3, each odd row represents a test, and the removed elements are depicted in cells with darker colors. The last cell of each odd row displays the result of the corresponding test. Each even row presents the probability of each node in Figure 2 being included in the final result after each test, given that its parent is included in the final result.

For each quantifier node and its children, a conditional probability table (CPT) is included to represent the probability of that node being in the final result when the parent node is in the final result. These CPTs are initialized according to the steps described in Section III-A4. This probabilistic model captures the syntactic relationships among nodes and ensures syntactic validity during the program simplification process, distinguishing T-PDD from ProbDD. During each step of the algorithm, T-PDD selects and excludes the node along with its descendant nodes that have the highest expected gain, as explained in Section III-B. T-PDD starts by excluding $n_3$ and its descendant elements $n_4$ to $n_8$. However, the test function fails, leading to an update of the CPT of $n_3$, as described

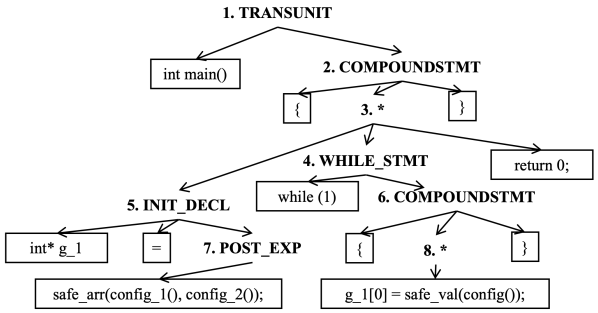



Fig. 2: Our model built from Figure 1, random variables do not be annotated for clarity.

| | $n_1$ | $n_2$ | $n_3$ | $n_4$ | $n_5$ | $n_6$ | $n_7$ | $n_8$ | |
|---|---|---|---|---|---|---|---|---|---|
| | 1.0 | 1.0 | 0.5 | 0.5 | 0.5 | 1.0 | 1.0 | 0.5 | |
| 1 | $n_1$ | $n_2$ | $n_3$ | $n_4$ | $n_5$ | $n_6$ | $n_7$ | $n_8$ | $\varphi(X^1) = F$ |
| | 1.0 | 1.0 | **0.89** | 0.5 | 0.5 | 1.0 | 1.0 | 0.5 | |
| 2 | $n_1$ | $n_2$ | $n_3$ | $n_4$ | $n_5$ | $n_6$ | $n_7$ | $n_8$ | $\varphi(X^2) = F$ |
| | 1.0 | 1.0 | 0.89 | **1.0** | 0.5 | 1.0 | 1.0 | 0.5 | |
| 3 | $n_1$ | $n_2$ | $n_3$ | $n_4$ | $n_5$ | $n_6$ | $n_7$ | $n_8$ | $\varphi(X^3) = T$ |
| | 1.0 | 1.0 | 0.89 | 1.0 | 0.5 | 1.0 | 1.0 | **0.0** | |
| 4 | $n_1$ | $n_2$ | $n_3$ | $n_4$ | $n_5$ | $n_6$ | $n_7$ | $n_8$ | $\varphi(X^4) = T$ |
| | 1.0 | 1.0 | 0.89 | 1.0 | **0.0** | 1.0 | 0.0 | 0.0 | |
| 5 | $n_1$ | $n_2$ | $n_3$ | $n_4$ | $n_5$ | $n_6$ | $n_7$ | $n_8$ | $\varphi(X^5) = F$ |
| | 1.0 | 1.0 | **1.0** | 1.0 | 0.0 | 1.0 | 0.0 | 0.0 | |

Fig. 3: Steps of our algorithm

in Section III-C. Subsequently, T-PDD proceeds to exclude $\{n_4, n_6, n_8\}$ at the second step, which fails the test, resulting in the necessary updates to the corresponding CPTs. At the third and fourth steps, T-PDD excludes $\{n_8\}$ and $\{n_5, n_7\}$ for testing, respectively. In both cases, the tests pass leading to setting the probabilities of $n_8$ and $n_5$ being included in the final result (given that their parents are included in the final result) to zero. During the fifth test, T-PDD focuses on the expected gain of $\{n_1, n_2\}$, which is the only object remaining to be tested. However, it fails the test function as well. After updating the CPT accordingly, each item in CPTs is either 0 or 1, prompting T-PDD to begin applying transformation templates to simplify the program. In this particular example, only two templates, namely replacing `2.COMPOUNDSTMT` with `6.COMPOUNDSTMT` and replacing `4.WHILE_STMT` with `6.COMPOUNDSTMT`, are applied. However, both of these transformations fail the test. Finally, the simplification process terminates and returns the simplified program. Please note that T-PDD does not define any transformation templates. As an example for the application of T-PDD to Perses, we traverse the transformation templates defined in Perses when processing the example program once after the convergence of T-PDD.

Indeed, our approach T-PDD demonstrates an efficient strategy for program minimization in this example. By prioritizing the removal of non-compulsory elements based on expected gain, T-PDD minimizes the number of tests required to achieve program simplification. Comparatively, domain-specific delta debugging approaches such as Perses follow the predefined sequences to attempt the removal of elments and requires 3 times ddmin application to process a single node `3.*` in the above example. While ProbDD shows improved performance compared to ddmin [11], it cannot capture syntactic relationships between nodes in AST. Even after replacing ddmin with ProbDD in Perses, it still requires three applications to address the node `3.*`. Moreover, Perses retains its predefined sequence of attempts to remove elements across different iterations, regardless of whether ddmin or ProbDD is employed. This discrepancy highlights the advantage of T-PDD in reducing the number of tests needed to simplify the program.

## III. Approach

In this section, we will describe the key components of our proposed approach, which includes the Bayesian network, the selection of subsequences for testing, and the update process based on historical test results.

### A. The Bayesian Network

*1) Notations:* Formally, the general delta debugging is defined as follows: Let $\mathbb{X}$ be the universe of all objects of interest. Let $\phi : \mathbb{X} \to \{F, T\}$ be a test function that determines whether an object exhibits a given property (T) or not (F). Also, let $|X|$ represent the size of an object $X \in \mathbb{X}$. The objective of delta debugging is to discover another object $X^* \in \mathbb{X}$, given an object $X \in \mathbb{X}$ where $\phi(X) = T$, such that $|X^*|$ is minimized while maintaining $\phi(X^*) = T$. In other words, $X^*$ retains the desired property.

Contrary to the general delta debugging approaches, our approach incorporates an initial parsing step, which transforms the input object $X$ into an AST denoted as $T$. The nodes of this tree, represented by $N = \langle n_1, n_2, \ldots, n_k \rangle$, can be categorized into two subsets: the leaf node set $N_l$ and the non-leaf node set $N_{nl}$. Each element in the original input sequence $X$ corresponds to an element in the leaf node set $N_l$. Our approach aims to identify the optimal structure $T^*$ that preserves the desired properties of the input while minimizing its size. By traversing the leaf nodes of the optimal structure $T^*$, the resulting optimized subsequence $X^*$ can be obtained. In essence, our approach leverages the structured representation of the input object to guide the simplification process, ensuring that the produced result adheres to the intended syntax structure while achieving the desired optimization.

*2) The Model:* Our model utilizes a tree-like Bayesian network, which is defined as a directed acyclic graph (DAG) denoted as $G = (\Theta, E)$. Here, $\Theta$ refers to the set of nodes in the graph, while $E$ represents the set of directed edges connecting the nodes. In our Bayesian network, each node $\theta_i \in \Theta$ corresponds to a Bernoulli random variable that indicates whether the corresponding element $n_i$ is included in $T^*$ or not. Therefore, $\theta_i$ and $n_i$ are correspond one-to-one. For leaf nodes $n_i \in N_l$, $\theta_i = 1$ indicates that the represented leaf node is included in $T^*$. For non-leaf nodes $n_i \in N_{nl}$, $\theta_i = 0$ signifies that the non-leaf node is not included in $T^*$,

and consequently, all nodes in its corresponding subtree are excluded. The edges in $E$ are derived from the AST. If there is an edge from $n_i$ to $n_j$ in $T$, an edge from $\theta_i$ to $\theta_j$ exists in $E$.

Let $S_{n_i}$ denote the parent of $n_i$ in $T$, and $S_{\theta_i}$ denote the parent of $\theta_i$ in $G$. Following the Bayesian network definition, we assign a conditional probability table (CPT) $P(\theta_i|S_{\theta_i})$ per node, specifying the probability of $\theta_i$ conditioned on its parent. Since $S_{\theta_i} = 0$ implies that all nodes in the corresponding subtree are not in $T^*$, we have $P(\theta_i = 1|S_{\theta_i} = 0) = 0$. Consequently, $P(\theta_i|S_{\theta_i})$ can be represented by a single probability value $p(\theta_i = 1|S_{\theta_i} = 1)$, denoted as $p_{\theta_i}$.

With the proposed Bayesian network model, we define the joint distribution of the graph $G$ as the product of the conditional probability distributions.

$$P(\theta_1, ..., \theta_k) = \prod_{i=1}^{k} P(\theta_i|S_{\theta_i}) \tag{1}$$

We define that $P(\theta_i|S_{\theta_i}) = P(\theta_i)$, if $\theta_i$ is the root of $G$. Let $\Theta_I$ be the set that includes $\theta_i$ and all its ancestors (the path from the root to $\theta_i$), the marginal probability of the $i$th variable is:

$$P(\theta_i = 1) = \prod_{\theta \in \Theta_I} p_\theta \tag{2}$$

*3) Inference:* In our approach, the Bayesian network is utilized to predict the probability of whether an object $X$ passes the test. Our core hypothesis is that $X$ will pass the test if and only if it includes all elements in the optimal subsequence $X^*$. According to this hypothesis, the probability of $X$ passing the test is equivalent to the probability that the deleted subtree $T_d$ does not contain any leaf nodes in $T^*$. Given a deleted subtree $T_d$, we can extend it to $T_{d-EX}$ by including all its ancestors up to the root, and the corresponding probabilistic subgraph can be denoted as $G_{d-EX} \subset G$. The probability that $T_d$ does not contain any leaf nodes in $T^*$ can be recursively defined as $Q(\theta_R)$, where $\theta_R$ is the root of $G_{d-EX}$ (also the root of $G$) and $Q$ is defined as follows.

$$Q(\theta) = \begin{cases} (1 - p_\theta), \ \theta \text{ is a leaf node} \\ (1 - p_\theta) + p_\theta \cdot \prod\limits_{\theta_c \in \Theta(G_{d-EX}), S_{\theta_c} = \theta} Q(\theta_c), \ \text{otherwise} \end{cases} \tag{3}$$

In this equation, $Q(\theta)$ represents the probability that all nodes in the subtree of $\theta$ are 0 if all the ancestors of $\theta$ are 1. The value $p_\theta$ represents $P(\theta = 1|S_\theta = 1)$, as described earlier.

The computation of $Q(\theta_R)$ can be performed in linear time, as the function $Q$ is called once for each node in $T_{d-EX}$. This allows for efficient prediction of the probability of a subsequence passing the test.

*4) The Priors:* The prior probabilities assigned to each node in the Bayesian network serve to incorporate our assumptions and knowledge about the syntax structure during the simplification process. Let us consider node $n_R$ in $T$, which represents the root node of $T$. We define $P(\theta_R = 1) = 1.0$, indicating that the root node is always included in the optimal subsequence. For the other nodes $n_i$ in $T$, we define the prior probability for each corresponding node in $G$ according to the following cases:

i) If $S_{n_i}$ is a quantifier node, we assign a prior probability of $p_{\theta_i} = \sigma$, where $\sigma$ is a hyperparameter in our approach. This hyperparameter can be tuned to control the probability of including node $n_i$ in the optimal subsequence. By adjusting $\sigma$, we can influence the likelihood of $n_i$'s children being retained or excluded during the simplification.

ii) If $S_{n_i}$ is a regular node, we set $p_{\theta_i} = 1.0$. In this case, if the parent node $S_{n_i}$ exists in the optimal subsequence $T^*$, node $n_i$ must also be included.

### B. Select a Subsequence for Testing

The gain of a test on a subsequence $X'$ is defined as the reduction in sequence size or the number of elements that would be excluded from the original sequence $X$ if the test passes. The gain function can be represented as follows:

$$gain(X', X) = \begin{cases} |X| - |X'|, & \phi(X') = T \\ 0, & \phi(X') = F \end{cases}$$

The gain is equal to the reduction in sequence size when the test passes ($\phi(X') = T$), and it is zero when the test fails ($\phi(X') = F$). To simplify the notation, we can use $gain(T_d)$ instead of $gain(X', X)$ since $X$ remains constant during the selection process, and $T_d$ corresponds to $X'$ one by one. This gain function helps quantify the impact of a test on the size of the sequence and guides the selection of optimal subsequences during the simplification process.

The expectation of the gain function $gain(T_d)$ can be computed by considering the number of leaf nodes in the deleted subtree $T_d$ and the probability that the test passes ($\phi(X') = T$). The expectation can be expressed as:

$$E(gain(T_d)) = |T_d|_l \cdot P(\phi(X') = T) \tag{4}$$

In this equation, $|T_d|_l$ represents the number of leaf nodes in the deleted subtree $T_d$. Please refer to Section III-A3 for details on how to compute $P(\phi(X') = T)$.

The goal of the selection is to find a deleted subtree $T_d$ to maximize $E(gain(T_d))$. We enumerate all the remaining subtrees in $T$ to find the largest expectation and delete that subtree to get $X'$ for testing.

### C. Update the Model

The process of updating the model based on historical test results involves computing the posterior probabilities of the variables in the Bayesian network given the observed test results. This task can be computationally challenging, as it falls under the category of weighted model counting, which is generally infeasible to solve exactly in polynomial time [11]. To address this challenge, we can approximate the posterior probabilities and iteratively update the model after each test, refining the probabilities based on the test results.

The procedure for updating the probabilities is as follows: after selecting a subsequence $X$ for testing, we obtain the result of whether it passes the test or not. Based on the result, we update the CPTs in the Bayesian network. Specifically, we focus on updating the posterior probabilities of the variables $\theta_i \in G_{d-EX}$, as described in Section III-A2.

On one hand, if the test passes, for $\theta_i \in G_{d-EX}$, we can update the posterior probability as:

$$P(\theta_i = 1|\phi(X) = T) = \frac{P(\theta_i = 1) \cdot P(\phi(X) = T|\theta_i = 1)}{P(\phi(X) = T)} \quad (5)$$

Here, $P(\theta_i = 1)$ represents the prior marginal probability, which can be computed using Equation 2. $P(\phi(X) = T)$ is the probability of the test passing, which has been obtained by computing $Q(\theta_R)$ in Equation 3. $P(\phi(X) = T|\theta_i = 1)$ can be obtained by computing $Q(\theta_R)$ after setting $\theta_i$ and all its ancestors to 1, and there is no need to recompute $Q$ for other $\theta$ that is not set to 1 in this process, as we have already computed those values when computing Equation 4 in Section III-B. Finally, the posterior conditional probability $\hat{p_{\theta_i}}$ is computed as $\hat{p_{\theta_i}} = \frac{P(\theta_i=1|\phi(X)=T)}{P(S_{\theta_i}=1|\phi(X)=T)}$.

On the other hand, if the test fails, the situation is similar. we update the posterior probability as:

$$P(\theta_i = 1|\phi(X) = F) = \frac{P(\theta_i=1)\cdot(1-P(\phi(X)=T|\theta_i=1))}{1-P(\phi(X)=T)} \quad (6)$$

The computation process is the same as when the test passes.

The updating process is performed in topological order, allowing for efficient computation in linear time. However, the sheer number of conditional probability tables (CPTs) that require updating poses a significant time-consuming challenge in practice, prompting us to seek ways to enhance efficiency. In theory, after a single failed test, the probabilities of the nodes along the path from the root to the deleted subtree root should ideally be 1.0. This would imply that these nodes are guaranteed to be retained in the final result. However, real-world scenarios often deviate from this ideal due to factors such as semantic dependencies, which our syntactic model fails to capture. To address this discrepancy, we adopt an optimization strategy focused on updating only the CPT of the deleted subtree root. This allows for the possibility of elements that should have been retained in the final result to be excluded in future tests, resulting in a smaller output size. Specifically, for $\theta_i \in G_{d-EX}$, when the test passes, we set $\hat{p_{\theta_i}} = 0$. Conversely, when the test fails, we set $\hat{p_{\theta_i}} = \frac{p_{\theta_i}}{P(\phi(X)=F)}$. Furthermore, if $\hat{p_{\theta_i}}$ exceeds 1.0 after this adjustment, $\hat{p_{\theta_i}}$ is then set to 1.0. While this optimization may slightly affect the model's accuracy, experimental results demonstrate its effectiveness. Further investigations can be pursued to explore additional optimizations that enhance the efficiency of the update process while preserving the accuracy of the results.

In our paper, we propose a novel approach called T-PDD for delta debugging. The core steps of our approach involve building a Bayesian network from the ASTs, selecting subsequences for testing based on expected gain, and updating the network after each test. This iterative process continues until each $p_\theta$ is either 0 or 1 or the expected gain falls below a predefined threshold (in our case, 1.0). In contrast to domain-specific delta debugging approaches, which apply all transformation templates throughout the process, in T-PDD, transformation templates are applied after the convergence. The termination condition is met when all templates have been applied once, and at this point, the simplified program is returned as the final result. By combining the exclusion of subtrees and the application of transformation templates, our algorithm provides a systematic and efficient approach to program simplification, yielding desirable results.

## IV. Evaluation

In our evaluation, we compare our proposed approach with Perses, a widely used domain-specific delta debugging approach in the field of compiler debugging. Perses is known for its state-of-the-art performance for context-free languages and is built upon the ddmin algorithm. We aim to assess the performance of our approach by conducting a comparative analysis with Perses. Given that ProbDD has demonstrated superior performance compared to the ddmin algorithm, we hypothesize that integrating ProbDD into Perses could potentially enhance its effectiveness. Therefore, we also evaluate our approach by comparing it against a variant of Perses that incorporates the ProbDD algorithm. We refer to this variant as p-Perses in our evaluation. Furthermore, we investigate the impact of the parameter settings in our approach. Specifically, our evaluation aims to address the following research questions:

- **RQ1.** How does T-PDD compare to Perses in terms of processing time and result size?
- **RQ2.** How does T-PDD compare to the variant of Perses (p-Perses) that incorporates the ProbDD algorithm in terms of processing time and result size?
- **RQ3.** What is the impact of the parameter in T-PDD?

### A. Experimental Setup

*1) Subjects:* Our evaluation dataset comprises a total of 20 subjects written in the C programming language. These subjects were selected from existing publications and are widely recognized and used in the field of software engineering. They serve as representative examples for evaluating the effectiveness of our approach.

In addition to the aforementioned subjects, we curated a comprehensive dataset consisting of 87 subjects. Among these, 82 subjects are written in the C programming language, while the remaining 5 subjects are written in the Rust programming language. These subjects cover a diverse range of codebases and programming constructs. In total, our study used 107 subjects. On average, each subject has 22,000 lines of code (LOC), ranging between 105 and 7,000 LOC.

To generate the C subjects, we utilized Csmith, a specialized fuzzing testing tool designed for C compilers. With Csmith, we generated a large number of C programs that were subsequently used to test 15 different stable versions of GCC and LLVM compilers over a period of one week. During

this extensive testing process, we identified a total of 133 buggy test cases that triggered crashes or resulted in wrong code errors in the compilers. These buggy test cases spread across 7 different stable versions of the compilers. To ensure efficiency and relevance in our evaluation, we further filtered out 51 test cases that exhibited excessively long compilation times. This step helped us focus on the subjects that were more manageable and representative of real-world scenarios. Ultimately, our evaluation dataset consisted of 82 subjects that successfully triggered bugs in 7 different stable versions of GCC and LLVM compilers. Among these subjects, 17 were responsible for triggering crash bugs in specific versions of GCC compilers (gcc-4.4.0 and gcc-4.6.0), while the remaining 65 subjects triggered wrong code bugs in various versions of GCC and LLVM compilers (gcc-4.4.3, gcc-4.5.0, gcc-4.6.0, gcc-6.2.0, gcc-7.1.0, and LLVM-6.0.1).

Regarding the Rust subjects, we specifically selected 5 subjects from the issue tracking system of Rust [14]. Out of these 5 subjects, one was found to trigger a crash bug in a particular version of the Rust compiler (rust-nightly-20191029). The remaining 4 subjects were responsible for triggering wrong code bugs in different versions of the Rust compilers (rust-1.20.0, rust-1.34.0, rust-nightly-20200210, and rust-nightly-20200922).

By incorporating both the curated and generated subjects, our evaluation dataset provides a comprehensive and diverse set of programs for assessing the effectiveness of our approach.

*2) Metrics:* In our study, we adopted three metrics commonly used in the evaluation of delta debugging approaches, as established by previous research [4], [10], [15]. These metrics include:

i) *Size of the produced result*: This metric measures the size of the minimized program or code snippet. We quantified the size using the number of tokens present in the result.
ii) *Processing time*: This metric quantifies the time taken by the delta debugging approach to minimize the program. We measured the processing time in seconds.
iii) *Number of tokens deleted per second*: This metric indicates the rate at which tokens are removed during the delta debugging process. It provides insights into the efficiency of the approach. We calculated this metric by dividing the total number of removed tokens by the processing time.

To ensure accurate representation of the results, we computed geometric means instead of arithmetic means when calculating the average results. This decision was made due to significant variations observed among different subjects in terms of the three metrics. By using geometric means, we provide a more balanced measure of the overall effectiveness of the delta debugging approach across the different subjects.

*3) Process:* To address RQ1, we utilized all 107 subjects in our evaluation. Firstly, we recorded the original size of each subject. Then, we applied Perses and T-PDD to each subject and recorded the size of the produced result as well as the processing time. Additionally, we calculated the number of tokens deleted per second. To determine the statistical significance of the improvements achieved by our approach in terms of effectiveness and efficiency compared to the original approaches, we conducted a paired sample Wilcoxon signed-ranked test and calculated the corresponding p-values using the sizes of the produced results, the number of tokens deleted per second, and the processing time.

To investigate RQ2, we compared the performance of T-PDD and p-Perses on a random selection of 25 subjects from our dataset, taking into consideration the time required to run all subjects.

To explore RQ3, we conducted experiments to assess the impact of the parameter $\sigma$ in T-PDD. We utilized the same subset of our dataset as in RQ2 for this experiment and varied the value of $\sigma$ (i.e., the initial value of conditional probability) to 0.3, 0.4, 0.5, 0.6, and 0.7. Due to the significant time required, we did not perform experiments on all subjects.

The results of both T-PDD and p-Perses are subject to randomness. To mitigate the influence of randomness, we executed both approaches five times and computed the average results. We selected five repetitions as the standard deviation of the five runs for each subject and approach was already below 1% of their respective average results. For RQ1 and RQ2, we set $\sigma$ in T-PDD to 0.5.

*4) Implementation:* For the compared approaches, we selected the latest version of Perses (v1.5) available at the beginning of our evaluation as the baseline approach for our evaluation. Additionally, we implemented p-Perses that incorporates the ProbDD algorithm. In this variant, we replaced the ddmin component of Perses (v1.5) with ProbDD and used the same hyper-parameter value as the one used in ProbDD [11]. Both approaches were executed on the subjects using the default settings of Perses.

Regarding our approach, T-PDD, we implemented the necessary functions for building the Bayesian network from an AST, updating posterior probabilities, and selecting objects for the next test. To ensure a fair comparison, we based our implementation of these functions on Perses (v1.5). However, we made a modification to the termination condition of T-PDD compared to Perses, and as a result, we added an extra command line option *–fixpoint false* when running T-PDD on the subjects. This option reflects the different termination conditions of T-PDD and Perses.

Our evaluation was performed on a Linux server with 16-core 32-thread Intel(R) Xeon(R) Gold 6130 CPU (3.7GHz), 128 Gigabyte RAM, and the operating system of Ubuntu Linux 16.04.

### B. Results and Analysis

*1) Comparison between T-PDD and Perses:* Table I presents the overall performance of T-PDD and Perses, highlighting their efficiency in terms of the three metrics. The results clearly indicate that T-PDD outperforms Perses in terms of efficiency. On average, T-PDD achieves a deletion rate of 12 more tokens per second than Perses across all 107 subjects, while maintaining the same size of the produced result. Importantly, this performance improvement is statistically

significant, as evidenced by both the $p{-}value_S$ and $p{-}value_T$ being less than 0.05.

*Detailed distribution of the improvements.* Due to space limitations, we are unable to list the results of each subject. However, we have published the detailed results in our GitHub repository for further reference. Additionally, we present the detailed distribution of improvements in Figure 4, focusing on the size of the produced result and the processing time. Each sub-figure showcases the improvement achieved by T-PDD over Perses on each subject, with the blue line representing the actual improvement and the orange line serving as a reference point (scale 0). Instances where T-PDD outperforms Perses are depicted above the orange line.

Regarding the size of the produced result, T-PDD performs worse than Perses on 24 subjects. This discrepancy can be attributed to the presence of unused variable definitions that remain in the output generated by T-PDD. These remnants contribute to an average increase of 22 tokens compared to the results produced by Perses. The nature of T-PDD, which terminates after a single traversal of transformation templates, can result in the retention of definitions associated with deleted invoked functions or variables. Concerning the processing time, T-PDD exhibits a relatively worse performance than Perses on 6 subjects, with 4 of these subjects yielding smaller size results when using T-PDD. Notably, there are only two subjects where T-PDD performs worse in terms of both the size of the produced result and the processing time. In such cases, it is observed that T-PDD struggles to efficiently eliminate large continuous chunks of dead code, a rare scenario in practice where the ddmin algorithm employed by Perses can handle that more effectively.

**RQ1:** In summary, our evaluation of T-PDD has demonstrated its significant efficiency improvement over Perses. Across a comprehensive set of 107 subjects, T-PDD outperforms Perses by consuming 26.95% less time and deleting 12 more tokens per second while achieving the same size of the produced result. These findings are statistically significant.

*2) Comparison between T-PDD and p-Perses:* Table II shows the detailed results and overall performance of T-PDD and p-Perses in terms of the three metrics. From Table II, we can see that T-PDD performs better than p-Perses in all metrics. On average, T-PDD deletes 8 more tokens per second to obtain 17.5% smaller results than p-Perses on a random selection of 25 subjects from our dataset. It is worth noting that p-Perses performs worse than T-PDD due to the predefined attempts to remove the elements as mentioned in Section II, which does not utilize the historical test results. However, there are two subjects where T-PDD performs worse than p-Perses in terms of the size of the produced result and the processing time, respectively. For the 20th subject, T-PDD only obtains one more token in the result compared to p-Perses, but consumes 8.077% less time. For the 25th subject, T-PDD requires more time than p-Perses. As discussed in Section IV-B1, T-PDD is not particularly effective in processing subjects that contain large sections of dead code, which is a rare scenario in the domain of program simplification.

**RQ2:** On average, T-PDD significantly outperforms p-Perses by deleting 8 more tokens per second to obtain 17.5% smaller results on a random selection of 25 subjects from our dataset.

*3) Impact of $\sigma$ in T-PDD:* We conducted an investigation into the impact of the initial conditional probability parameter, denoted as $\sigma$, in T-PDD. This investigation was based on a random selection of 25 subjects from our dataset. The results are presented in Figure 5, which consists of multiple sub-figures depicting the size of the produced result, the number of tokens deleted per second, and the processing time of T-PDD with different $\sigma$ values. In each sub-figure, the blue line represents the performance of T-PDD with the corresponding $\sigma$ value, while the orange line represents the performance of Perses for comparison purposes. It is worth noting that although different $\sigma$ values may lead to variations in performance, T-PDD consistently outperforms Perses across all studied $\sigma$ values. Furthermore, we observed that the performance differences between different $\sigma$ values are considerably smaller compared to the performance difference between T-PDD and Perses. This indicates that the choice of $\sigma$ has a relatively minor impact on the overall performance of T-PDD.

**RQ3:** Our experiments revealed that the parameter $\sigma$ has a minimal impact on the performance of T-PDD. Regardless of the specific values tested, T-PDD consistently outperformed Perses.

### C. Threats to Validity

To address the threat to internal validity, we have made efforts to ensure the correctness of the implementation of T-PDD and the experimental scripts. We conducted a thorough code review and testing to verify the accuracy and reliability of our implementation. By carefully examining the code, we aimed to minimize the potential for errors or inconsistencies that could affect the internal validity of our study.

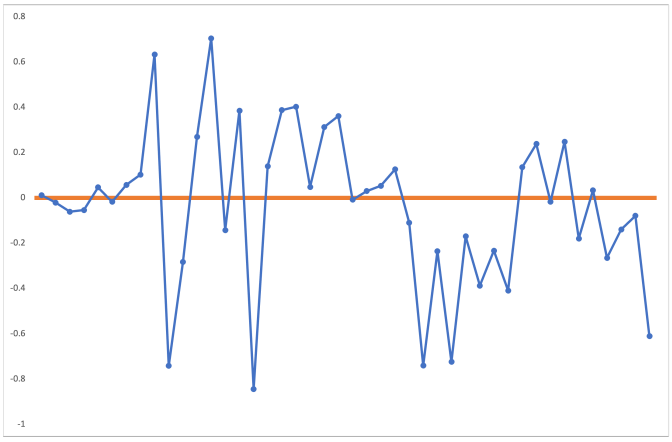
The threat to external validity is mainly associated with the subjects used in our study and the compared approaches. In order to address this, we have employed a combination of subjects from existing publications, which are commonly used and recognized in the field. Furthermore, to enhance the diversity of subjects, we have expanded our evaluation to include 5 Rust files and a selection of C files. This broader range of subjects enables us to investigate the generalizability of our approach. In the future, we plan to further extend the evaluation by incorporating additional subjects from various types of context-free grammars. Regarding the compared approaches, we have chosen Perses as a representative approach in the domain of compiler debugging, as discussed in Section IV-A.

The threat to construct validity primarily arises from randomness inherent in our experimental setup. Randomness can

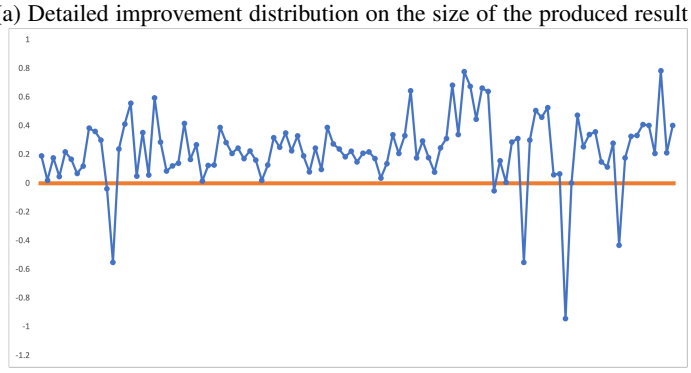
TABLE I: Comparison between T-PDD and Perses

| Summary | $R_i$ | T-PDD $R_t$ | T-PDD $S_t$ | T-PDD $T_t$ | Perses $R_p$ | Perses $S_p$ | Perses $T_p$ | $\uparrow_R$ | $p-value_R$ | $\times S$ | $p-value_S$ | $\uparrow_T$ | $p-value_T$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Dataset** | 45,612 | 48 | 43 | 1,000 | 48 | 31 | 1,369 | 00.00% | 0.9736 | 1.37 | 0.0000 | 26.95% | 0.0006 |

In this table and the tables in the rest of this section, $R$ represents the size of the results; $S$ represents the number of tokens deleted per seconds; $T$ represents the processing time in seconds; $R_i$ represents the size of the input; $t$ represents the T-PDD; $p$ represents the Perses; $\uparrow$ denotes the improvement, where $\uparrow_X = (X_p - X_t)/X_p$; $\times S$ denotes the speedup, where $\times S = S_t/S_p$. In this table, all numbers are geometric means.



(a) Detailed improvement distribution on the size of the produced result



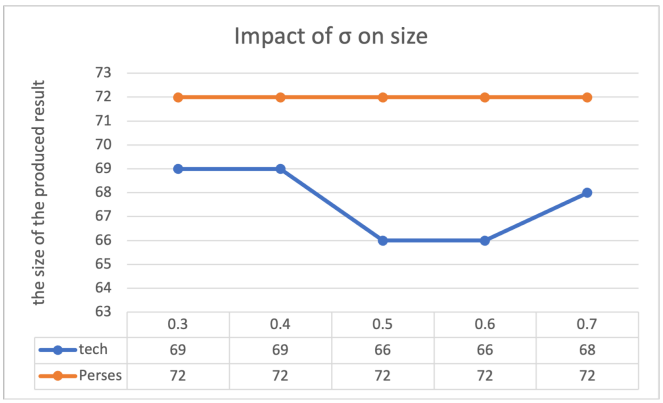
(b) Detailed improvement distribution on the processing time

Fig. 4: Detailed improvement distribution

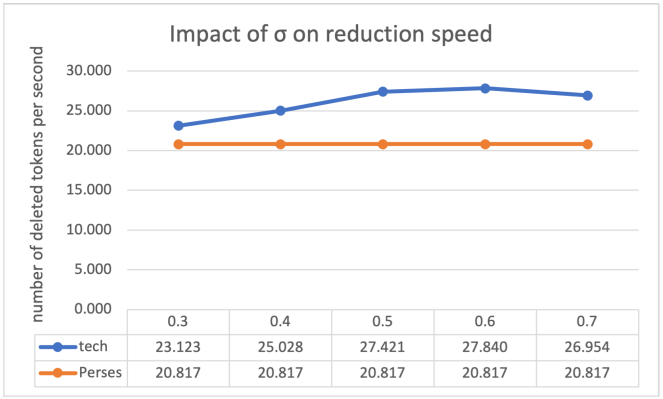
TABLE II: Comparison between T-PDD and p-Perses: Detailed Data

| D | Subject | T-PDD $R_t$ | T-PDD $S_t$ | T-PDD $T_t$ | p-Perses $R_p$ | p-Perses $S_p$ | p-Perses $T_p$ | $\uparrow_R$ | $\times S$ | $\uparrow_T$ |
|---|---|---|---|---|---|---|---|---|---|---|
| A dataset subset consisting of 25 subjects | 1 | 20 | 63.008 | 858 | 20 | 60.001 | 901 | 0.0% | 1.05 | 4.772% |
| | 2 | 20 | 87.628 | 494 | 20 | 62.827 | 689 | 0.0% | 1.395 | 28.302% |
| | 3 | 20 | 87.05 | 536 | 20 | 48.351 | 965 | 0.0% | 1.8 | 44.456% |
| | 4 | 20 | 208.426 | 1,075 | 20 | 89.053 | 2,516 | 0.0% | 2.34 | 57.273% |
| | 5 | 20 | 171.498 | 1,459 | 20 | 162.689 | 1,538 | 0.0% | 1.054 | 5.137% |
| | 6 | 20 | 110.01 | 630 | 20 | 103.288 | 671 | 0.0% | 1.065 | 6.11% |
| | 7 | 145 | 44.273 | 1,849 | 267 | 27.374 | 2,986 | 45.693% | 1.617 | 38.078% |
| | 8 | 103 | 14.41 | 1,769 | 189 | 13.095 | 1,940 | 45.503% | 1.1 | 8.814% |
| | 9 | 82 | 19.823 | 1,677 | 203 | 15.833 | 2,092 | 59.606% | 1.252 | 19.837% |
| | 10 | 72 | 35.444 | 1,041 | 79 | 22.549 | 1,636 | 8.861% | 1.572 | 36.369% |
| | 11 | 229 | 33.23 | 1,834 | 395 | 27.036 | 2,248 | 42.025% | 1.229 | 18.416% |
| | 12 | 351 | 23.326 | 2,077 | 410 | 20.443 | 2,367 | 14.39% | 1.141 | 12.252% |
| | 13 | 237 | 23.637 | 2,426 | 281 | 17.243 | 3,323 | 15.658% | 1.371 | 26.994% |
| | 14 | 260 | 74.224 | 2,003 | 301 | 42.2 | 3,522 | 13.621% | 1.759 | 43.129% |
| | 15 | 20 | 104.099 | 696 | 20 | 38.849 | 1,865 | 0.0% | 2.68 | 62.681% |
| | 16 | 20 | 87.099 | 497 | 20 | 62.555 | 692 | 0.0% | 1.392 | 28.179% |
| | 17 | 20 | 86.888 | 537 | 26 | 73.935 | 631 | 23.077% | 1.175 | 14.897% |
| | 18 | 20 | 108.886 | 1,038 | 20 | 63.425 | 1,782 | 0.0% | 1.717 | 41.751% |
| | 19 | 55 | 17.026 | 574 | 116 | 14.693 | 661 | 52.586% | 1.159 | 13.162% |
| | 20 | 60 | 4.166 | 808 | 59 | 3.83 | 879 | -1.695% | 1.088 | 8.077% |
| | 21 | 66 | 4.051 | 884 | 74 | 2.975 | 1,201 | 10.811% | 1.362 | 26.395% |
| | 22 | 54 | 27.526 | 1,549 | 55 | 16.107 | 2,647 | 1.818% | 1.709 | 41.481% |
| | 23 | 4,539 | 0.239 | 15,090 | 4,677 | 0.111 | 31,216 | 2.951% | 2.153 | 51.659% |
| | 24 | 99 | 0.127 | 55 | 99 | 0.047 | 150 | 0.0% | 2.702 | 63.333% |
| | 25 | 141 | 37.69 | 4,890 | 172 | 54.118 | 3,405 | 18.023% | 0.696 | -43.612% |
| | Summary | 66 | 27 | 1,104 | 80 | 19 | 1,572 | 17.5% | 1.421 | 29.771% |

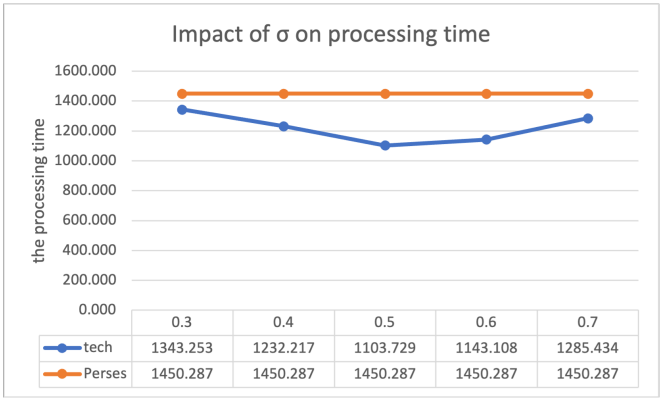
introduce variability in the performance of T-PDD and the variant of Perses used for comparison. To mitigate this threat, we have conducted multiple runs of each approach on each




(a) Impact of $\sigma$ on the size of the produced result




(b) Impact of $\sigma$ on the number of deleted tokens per second




(c) Impact of $\sigma$ on the processing time

Fig. 5: Impact of $\sigma$ for T-PDD

subject, specifically performing 5 repetitions. By calculating the average results, we have aimed to account for the impact of randomness and obtain more reliable performance metrics. Further details on the experimental setup and the results can be found in Section IV-A.

## V. Related Work

### A. Probabilistic Delta Debugging Approaches

One closely related work to our approach is ProbDD [11], which introduces a probabilistic model to guide the delta debugging process and achieves state-of-the-art performance in this field. Our approach, T-PDD, shares a similar motivation of utilizing historical test results to guide delta debugging. However, there are key differences between ProbDD and T-PDD. While ProbDD assumes independence between elements, T-PDD leverages the AST to construct a probabilistic model, capturing the relationships among elements. This distinction allows T-PDD to effectively address domain-specific delta debugging problems, while ProbDD focuses on a more general context. Another related work is CHISEL [10], which designs a Markov decision model to optimize the predefined sequences of the ddmin algorithm. Furthermore, CHISEL incorporates an analysis tailored to C programs, enabling it to identify def-use relationships. Pardis [16] is an approach that leverages machine learning techniques to enhance the delta debugging process for C programs. It trains a model to predict the semantic validity of objects, thereby improving the efficiency and effectiveness of the debugging process. DEBOP [13] is another probabilistic approach based on Markov chain Monte Carlo (MCMC) with Metropolis-Hastings sampling. However, DEBOP is designed for a different problem domain: software debloating, where the goal is to maximize a set of continuous objective functions instead of using a binary test function. Consequently, DEBOP is not suitable for addressing domain-specific delta debugging problems with binary test results, which is the focus of T-PDD.

### B. Domain-specific Delta Debugging Approaches

Different with the general delta debugging approaches such as ddmin and ProbDD, there exist several domain-specific delta debugging approaches tailored to specific programming languages or problem domains. Berkeley Delta [17] utilizes *topformflat* to identify nested structures and then perform ddmin on them. Hierarchical Delta Debugging (HDD) [15] is a program simplification technique that utilizes syntax trees as its foundation, building upon the ddmin algorithm. Since its introduction, several variants of HDD have been proposed to further enhance its capabilities. These include Coarse HDD [18], HDDr [19], and Modern HDD [20], [21]. These variants aim to improve upon the original HDD algorithm by introducing new strategies and optimizations for more effective program simplification. C-reduce [5] is one such approach that employs heuristics based on the C/C++ semantics obtained from Clang for efficient program reduction. Similar to C-reduce, there are other approaches in the domain of software debloating that leverage expert knowledge to guide the reduction process. Examples of such approaches include Trimmer [7], uTrimmer [8] and PRAT [9]. J-reduce and its improved version, proposed by Kalhauge and Palsberg [22], [23], are reduction tools for Java bytecode that models the reduction task as a problem of dependency graph reduction. GTR* [24] defines transformation templates for tree-structured data and filters out templates not present in a collected corpus of example data. Storm [25] designs transformation templates specifically for probabilistic program reduction. While these transformation template-based delta debugging approaches have shown improvements in reduction effectiveness within their respective domains, they often suffer from efficiency issues, as reported in existing studies [4]. Perses [4], on the other hand, is a general-purpose delta debugging approach that operates on the AST level. However, Perses follows predefined sequences of attempts for program simplification and does not leverage information from existing test results. In contrast, our approach T-PDD utilizes a Bayesian network constructed from the AST, incorporating existing test results and capturing the relationships among elements to estimate the probability of each element being retained in the result. As revealed in Section IV, T-PDD significantly outperforms Perses in terms of efficiency. More recently, Vulcan [12] has been proposed to perform aggressive program transformations using the formal syntax of the language. While Vulcan can achieve smaller sizes of the produced result, it is designed as a post-processing step for language-agnostic program reducers and the designed templates are highly time-consuming.

## VI. Conclusion

This paper introduces T-PDD, a novel approach designed to enhance the performance of domain-specific delta debugging techniques. By constructing a Bayesian network from the Abstract Syntax Tree, T-PDD leverages existing test results and captures the relationships among elements to estimate the probability of each element being retained in the result. In T-PDD, a strategic selection of elements is made to maximize the gain of the subsequent test, guided by the Bayesian network. The network is then updated based on the test results, continually refining the estimation process. The experimental results demonstrate the significant advantages of T-PDD over Perses, a representative domain-specific delta debugging approach. T-PDD achieves a remarkable 26.95% reduction in processing time on average, while also producing results that are 3.4 times smaller in the best case. Our tool and benchmarks can be found at: https://github.com/Amocy-Wang/T-PDD.

## Acknowledgments

We extend our sincere gratitude to the anonymous ISSRE reviewers for their insightful comments and dedicated efforts that significantly enhanced the quality of this paper. This research has been generously supported by the National Key Research and Development Program of China under Grant No. 2022YFB4501902 and the National Natural Science Foundation of China under Grant No. 62172017, No. 62232003.

## REFERENCES


[1] A. Zeller, "Yesterday, my program worked. today, it does not. why?" *ACM SIGSOFT Software engineering notes*, vol. 24, no. 6, pp. 253–267, 1999.

[2] ——, "Isolating cause-effect chains from computer programs," *ACM SIGSOFT Software Engineering Notes*, vol. 27, no. 6, pp. 1–10, 2002.

[3] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.

[4] C. Sun, Y. Li, Q. Zhang, T. Gu, and Z. Su, "Perses: syntax-guided program reduction," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 361–371.

[5] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for c compiler bugs," in *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, 2012, pp. 335–346.

[6] H. Cleve and A. Zeller, "Locating causes of program failures," in *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.* IEEE, 2005, pp. 342–351.

[7] H. Sharif, M. Abubakar, A. Gehani, and F. Zaffar, "Trimmer: application specialization for code debloating," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 329–339.

[8] H. Zhang, M. Ren, Y. Lei, and J. Ming, "One size does not fit all: security hardening of mips embedded systems via static binary debloating for shared libraries," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 255–270.

[9] R. Williams, T. Ren, L. De Carli, L. Lu, and G. Smith, "Guided feature identification and removal for resource-constrained firmware," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 2, pp. 1–25, 2021.

[10] K. Heo, W. Lee, P. Pashakhanloo, and M. Naik, "Effective program debloating via reinforcement learning," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 380–394.

[11] G. Wang, R. Shen, J. Chen, Y. Xiong, and L. Zhang, "Probabilistic delta debugging," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 881–892.

[12] Z. Xu, Y. Tian, M. Zhang, G. Zhao, Y. Jiang, and C. Sun, "Pushing the limit of 1-minimality of language-agnostic program reduction," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA1, pp. 636–664, 2023.

[13] Q. Xin, M. Kim, Q. Zhang, and A. Orso, "Program debloating via stochastic optimization," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*, 2020, pp. 65–68.

[14] "Rust," Accessed: 2023. [Online]. Available: https://github.com/rust-lang/rust/issues

[15] G. Misherghi and Z. Su, "Hdd: hierarchical delta debugging," in *Proceedings of the 28th international conference on Software engineering*, 2006, pp. 142–151.

[16] G. Gharachorlu and N. Sumner, "Leveraging models to reduce test cases in software repositories," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 2021, pp. 230–241.

[17] "Berkeley delta," Accessed: 2023. [Online]. Available: http://delta.tigris.org/

[18] R. Hodován, Á. Kiss, and T. Gyimóthy, "Coarse hierarchical delta debugging," in *2017 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 2017, pp. 194–203.

[19] Á. Kiss, R. Hodován, and T. Gyimóthy, "Hddr: a recursive variant of the hierarchical delta debugging algorithm," in *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, 2018, pp. 16–22.

[20] R. Hodován and Á. Kiss, "Modernizing hierarchical delta debugging," in *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation*, 2016, pp. 31–37.

[21] "The implementation of modernized hdd," Accessed: 2023. [Online]. Available: https://github.com/renatahodovan/picireny

[22] C. G. Kalhauge and J. Palsberg, "Binary reduction of dependency graphs," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 556–566.

[23] ——, "Logical bytecode reduction," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 1003–1016.

[24] S. Herfert, J. Patra, and M. Pradel, "Automatically reducing tree-structured test inputs," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 861–871.

[25] S. Dutta, W. Zhang, Z. Huang, and S. Misailovic, "Storm: program reduction for testing and debugging probabilistic programming systems," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 729–739.