Probabilistic Delta Debugging

Anonymous Author(s)*

ABSTRACT

The delta debugging problem concerns how to reduce an object while preserving a certain property, and widely exists in many applications, such as compiler development, regression fault localization, and software debloating. Given the importance of delta debugging, multiple algorithms have been proposed to solve the delta debugging problem efficiently and effectively. However, the efficiency and effectiveness of the state-of-the-art algorithms are still not satisfactory. For example, the state-of-the-art delta debugging tool, CHISEL, may take up to 3 hours to reduce a single program with 14,092 lines of code, while the reduced program may be up to 2 times unnecessarily large.

In this paper, we propose a probabilistic delta debugging algorithm (named ProbDD) to improve the efficiency and the effectiveness of delta debugging. Our key insight is, the ddmin algorithm, the basic algorithm upon which many existing approaches are built, follows a predefined sequence of attempts to remove elements from a sequence, and fails to utilize the information from existing test results. To address this problem, ProbDD builds a probabilistic model to estimate the probabilities of the elements to be kept in the produced result, selects a set of elements to maximize the gain of the next test based on the model, and improves the model based on the test results.

We prove the correctness of ProbDD, and analyze the minimality of its result and the asymptotic number of tests under the worst case. The asymptotic number of tests in the worst case of ProbDD is $O(n)$, which is smaller than that of ddmin, $O(n^2)$ worst-case asymptotic number of tests. Furthermore, we experimentally compared ProbDD with ddmin on 40 subjects in HDD and CHISEL, two approaches that wrap ddmin for reducing trees and C programs, respectively. The results show that, after replacing ddmin with ProbDD, HDD and CHISEL produce 59.48% and 11.51% smaller results and use 63.22% and 45.27% less time, respectively.

ACM Reference Format:
Anonymous Author(s). 2018. Probabilistic Delta Debugging. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Delta debugging automatically reduces a set of elements while preserving a certain property [22], and has found applications in many domains, such as compiler debugging [23], regression fault localization [20], isolating the cause-effect chain of a failure [5, 21], and debloating software to reduce the size of a program while keeping certain desired functionalities [7].

Formally, delta debugging is defined as follows. Let $\mathbb{X}$ be a universe of all objects of interest, $\phi : \mathbb{X} \longrightarrow \{F, T\}$ be a test function determining whether an object exhibits a given property (T) or not (F), and $|X|$ be the size of an object $X \in \mathbb{X}$. Given an object $X \in \mathbb{X}$ that $\phi(X) = T$, the goal of delta debugging is to find another object $X^* \in \mathbb{X}$ such that $|X^*|$ is as small as possible and $\phi(X^*) = T$, i.e., X^* preserves the property. For example, in compiler development, delta debugging is used to find a smaller program that reproduces a compilation failure. Here $\mathbb{X}$ is a universe of programs, X is a possibly large program that leads a compilation failure, and ϕ tests whether the compilation failure still exists or not.

The state-of-the-art family of delta debugging approaches is built upon the ddmin algorithm [23]. The ddmin algorithm views an object $X \in \mathbb{X}$ as a sequence. In each iteration, ddmin splits X into n subsequences and tries to remove each subsequence and its complement from X. The number n starts with 2 and doubles in each iteration. Subsequent approaches in this family assume more complex domain-specific structures and apply ddmin to the sequences in the structures. For example, HDD [13] assumes the objects have a structure of a tree and applies ddmin only to the sequences of siblings. CHISEL [7] further considers the data and control dependency relations between elements in a C program and applies ddmin in a way that would not break the dependencies.

However, the efficiency and effectiveness of the state-of-the-art delta debugging algorithms are still not satisfactory. For example, as our evaluation will reveal later, the state-of-the-art delta debugging tool, CHISEL [7], may take up to 3 hours to reduce a single program with 14,092 lines of code, while the reduced program may be up to 2 times unnecessarily large.

In this paper, we aim to improve the effectiveness and efficiency of delta debugging. Our key insight is, the ddmin algorithm, the central component of many existing approaches, follows a predefined sequence of attempts to remove elements from the original object, and fails to utilize the information from existing test results. To address this problem, we propose a probabilistic delta debugging algorithm, ProbDD. ProbDD builds a probabilistic model to estimate the probability of each element to be kept in the produced result. In each iteration, ProbDD selects a subset of elements to maximize the gain of the next test based on the probabilistic model, and tests if the desired property is preserved in this subset. Then, ProbDD updates the probabilistic model based on the testing result.

We prove the result produced by ProbDD is correct and is minimal or minimum if the universe of objects satisfies certain conditions. We also analyze the asymptotic number of tests under the worst case. The asymptotic number of tests in the worst case of ProbDD is $O(n)$, which is smaller than that of ddmin, $O(n^2)$ worst-case asymptotic number of tests.

Furthermore, we evaluated ProbDD on 40 subjects in two application domains, i.e., trees and C programs, by substituting ProbDD

ESEC/FSE 2021, 23 - 27 August, 2021, Athens, Greece

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

for ddmin in two representative approaches for the two domains, HDD [13] and CHISEL [7]. The number of subjects in our evaluation is larger than all recent publications on delta debugging at top venues [7, 9–11, 13, 15, 17, 20, 21, 23] as far as we are aware. The results demonstrate that ProbDD significantly improves both the efficiency and the effectiveness of the representative approaches in the two domains. On average, after substituting ProbDD for ddmin, HDD and CHISEL produces 59.48% and 11.51% smaller results within the time limit, respectively. On the subjects where both versions finish within the time limit, after substituting ProbDD for ddmin, HDD and CHISEL use 63.22% and 45.27% less time, respectively.

In summary, this paper makes the following main contributions.

- We propose a novel probabilistic delta debugging algorithm, ProbDD, which dynamically learns a probabilistic model to efficiently and effectively reduce a sequence of elements.
- We prove the correctness of ProbDD, analyze the minimality of its result and the asymptotic number of tests under the worst case.
- We evaluate ProbDD in two application domains, demonstrating that ProbDD significantly improves the representative approaches in the two domains in both efficiency and effectiveness.

2 MOTIVATING EXAMPLE

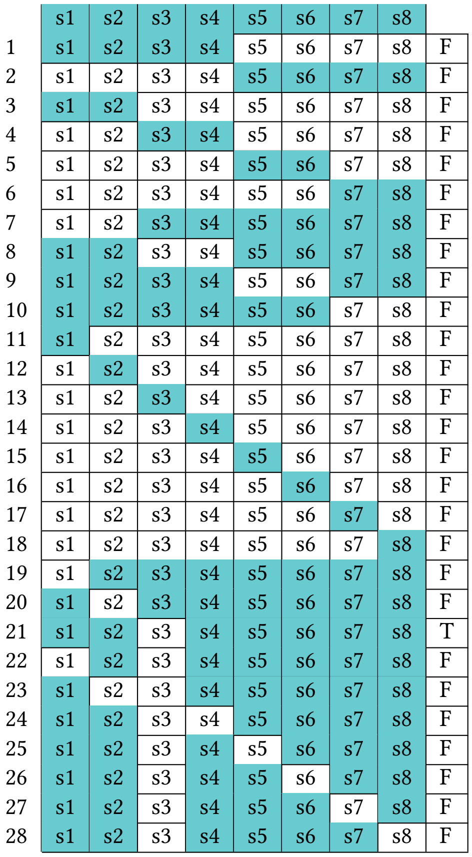
We use a program minimization example to illustrate how ddmin works. Listing 1 shows a real program from TensorFlow tutorials [2]. Let us assume that the function type() is faulty and any valid invocation to it will result in the same error. Now we would like to reduce the program such that the error is still produced. There are 8 statements in the program, and the goal of delta debugging is to find a subsequence of statements that still invokes type() and thus produces the error. Here we use s_i to denote the statement in Line i.

Listing 1: Example program to be reduced

```
1 import tensorflow as tf
2 x = tf.constant(3.0)
3 b = 1.0
4 with tf.GradientTape() as tape:
5   tape.watch(x)
6   y = x**2
7 b = tape.gradient(y,x)
8 print(type(b))
```

The ddmin algorithm views the set of elements as a sequence and proceeds as two nested loops. The outer loop reduces a variable n representing the length of the subsequence to be considered. The length n starts from 1/2 of all elements and reduces by half at each iteration until it reaches 1. The inner loop first tests all consecutive and disjoint subsequences of length n, and then tests the complements of these subsequences. If any test is successful, keep only this subsequence. If a subsequence or its complement has been tested before, skip it.

The tests that ddmin performs for this example are shown in Figure 1. At the end of each row, there is a T or an F, which means that the error is still produced (T) or not (F). First, n is 4, the two subsequences of length 4 are tested at lines 1 and 2. Both tests fail and their complements are all tested, so the first outer iteration finishes. Second, n is halved as 2, the four subsequences of length 2 are tested at lines 3 to 6. All the tests fail and the tests of their complements also fail, so the second outer iteration finishes. Third, n is halved as 1, the eight subsequences of length 1 are tested at lines 11 to 18. All these tests fail. Then, the complements are tested and the complement of $\{s_3\}$ passes the test at line 21. Since the test passes, the elements in the complement are kept and the seven subsequences and their complements need to be tested, so the algorithm continues with n as 1. However, none of the tests for the seven subsequences succeed, which are tested before, and none of their complements succeed (lines 22-28), so the third outer iteration finishes and the algorithm returns $\{s_1, s_2, s_4, s_5, s_6, s_7, s_8\}$. The returned set is 1-minimal because it cannot be further reduced by removing any single element from it.



	s1	**s2**	**s3**	**s4**	**s5**	**s6**	**s7**	**s8**	
1	**s1**	**s2**	**s3**	**s4**	s5	s6	s7	s8	F
2	s1	s2	s3	s4	**s5**	**s6**	**s7**	**s8**	F
3	**s1**	**s2**	s3	s4	s5	s6	s7	s8	F
4	s1	s2	**s3**	**s4**	s5	s6	s7	s8	F
5	s1	s2	s3	s4	**s5**	**s6**	s7	s8	F
6	s1	s2	s3	s4	s5	s6	**s7**	**s8**	F
7	s1	s2	**s3**	**s4**	**s5**	**s6**	**s7**	**s8**	F
8	**s1**	**s2**	s3	s4	**s5**	**s6**	**s7**	**s8**	F
9	**s1**	**s2**	**s3**	**s4**	s5	s6	**s7**	**s8**	F
10	**s1**	**s2**	**s3**	**s4**	**s5**	**s6**	s7	s8	F
11	**s1**	s2	s3	s4	s5	s6	s7	s8	F
12	s1	**s2**	s3	s4	s5	s6	s7	s8	F
13	s1	s2	**s3**	s4	s5	s6	s7	s8	F
14	s1	s2	s3	**s4**	s5	s6	s7	s8	F
15	s1	s2	s3	s4	**s5**	s6	s7	s8	F
16	s1	s2	s3	s4	s5	**s6**	s7	s8	F
17	s1	s2	s3	s4	s5	s6	**s7**	s8	F
18	s1	s2	s3	s4	s5	s6	s7	**s8**	F
19	s1	**s2**	**s3**	**s4**	**s5**	**s6**	**s7**	**s8**	F
20	**s1**	s2	**s3**	**s4**	**s5**	**s6**	**s7**	**s8**	F
21	**s1**	**s2**	s3	**s4**	**s5**	**s6**	**s7**	**s8**	T
22	s1	**s2**	s3	**s4**	**s5**	**s6**	**s7**	**s8**	F
23	**s1**	s2	s3	**s4**	**s5**	**s6**	**s7**	**s8**	F
24	**s1**	**s2**	s3	s4	**s5**	**s6**	**s7**	**s8**	F
25	**s1**	**s2**	s3	**s4**	s5	**s6**	**s7**	**s8**	F
26	**s1**	**s2**	s3	**s4**	**s5**	s6	**s7**	**s8**	F
27	**s1**	**s2**	s3	**s4**	**s5**	**s6**	s7	**s8**	F
28	**s1**	**s2**	s3	**s4**	**s5**	**s6**	**s7**	s8	F

Figure 1: Detailed iterations of ddmin

As we can see from the example, the sequence of attempts for ddmin is predefined and does not learn from past test results. For example, in this example statement, s_8 should not be removed. However, following the predefined order, the statement s_8 has been tried to remove 13 times, and all these attempts would fail. In fact, as studied by Zeller and Hildebrandt [23], the worst-case asymptotic number of tests in ddmin is $O(n^2)$, where n is the size of the initial set. Also, the reduced result contains seven statements, while the optimal result is $\{s_3, s_8\}$, containing only two statements.

3 APPROACH

From the analysis of the previous section, we can see that ddmin does not learn from the history of test results, and could keep removing an element though all historical removals of this element lead to test failures. To overcome this problem, our approach builds a probabilistic model to guide the tests and updates the probabilistic model based on the test results. The process continues until the probabilistic model predicts with 100% certainty that a subsequence is the optimal subsequence. In this way, the test history guides future tests through the probabilistic model. In this section, we describe (1) this model, (2) how this model should be updated based on the test results, and (3) how to use this model to guide tests.

3.1 The Probabilistic Model

3.1.1 Notations. Since our goal is to optimize ddmin, we also view the input object as a sequence and try to identify a subsequence that makes the test pass. In other words, the universe $\mathbb{X}$ is a n-dimensional Boolean space and an object X in the universe is a Boolean vector $X = \langle x_1, x_2, ..., x_n \rangle$ where $x_i \in \{0, 1\}$. Here $x_i = 1$ indicates that the ith element is included in the subsequence and $x_i = 0$ indicates that the ith element is excluded from the subsequence. To simplify the presentation, we also view a subsequence X as a set containing the indexes of the included elements, i.e., $\{i \mid x_i = 1\}$, so that the set operators such as $\subseteq$ apply to subsequences.

3.1.2 The Existence of the Optimal Subsequence. To simplify the probabilistic analysis, we assume two properties of the universe $\mathbb{X}$ which are often assumed or discussed in existing work [20, 23]. Please note the goal of assuming the two properties is to deduce the design of our probabilistic model, and the correctness and the time complexity of ProbDD do not depend on the two properties.

The monotony property says that if X fails the test, any subsequence of X fails the test.

Definition 3.1 (Monotony). $\forall X, X' \in \mathbb{X}, X' \subseteq X \wedge \phi(X) = F \Rightarrow \phi(X') = F$

The unambiguity property says that if two subsequences pass the test, their intersection passes the test.

Definition 3.2 (Unambiguity). $\forall X, X' \in \mathbb{X}, \phi(X) = T \wedge \phi(X') = T \Rightarrow \phi(X \cap X') = T$

We show that the above two properties imply the existence of an optimal subsequence where the test passes if and only if the elements in the subsequence are present.

Theorem 3.3. *If a universe $\mathbb{X}$ is both monotone and unambiguous, there exists an optimal subsequence X^* such that the following holds.*

$$\phi(X) = \begin{cases} T & X^* \subseteq X \\ F & \text{otherwise} \end{cases}$$

Proof. Let $X^* = \bigcap_{\phi(X)=T} X$. Based on unambiguity, we know that $\phi(X^*) = T$. Based on monotony, we know that any superset X of X^* makes the test pass, i.e., $\phi(X) = T$. Now let X be a subsequence that is not a superset of X^*. If we assume $\phi(X) = T$, we have $X \cap X^* = X^*$ by the definition of X^*, which contradicts with the fact that X is not a superset of X^*. Therefore, $\phi(X) = F$. □

3.1.3 The Model. Now we proceed to define the probabilistic model. Given the existence of the optimal subsequence X^*, the goal of delta debugging is to identify elements in X^*. Therefore, we assign the element at each index i a Bernoulli random variable θ_i to denote whether the ith element is in X^* or not. We use parameter p_i to denote the probability of the ith element is in X^*, i.e., $Pr(\theta_i = 1) = p_i$. Therefore, our probabilistic model is a n-dimensional vector of parameters $\langle p_1, p_2, \ldots, p_n \rangle$.

We further assume that the random variables are independent of each other, i.e., $\theta_i \perp \theta_j$ for any i and j. This assumption is reasonable because modern delta debugging approaches have considered the domain-specific structure of the objects, and if two elements depend on each other, e.g., they can be removed together but cannot be individually removed, such a dependency are likely to be captured by the outer approach wrapping ddmin. When ddmin is applied to a sequence, most of the elements in this sequence should not depend on each other. Based on this assumption, the probability of a vector X being equal to X^* is $\prod_i p_i^{x_i}(1-p_i)^{1-x_i}$.[1] This also implies that the delta debugging process of our approach stops when each p_i is either 1 or 0.

With this model, it is easy for us to calculate the probability of a test result. For example, the probability of X passing the test is the probability that no element in X^* is excluded from X, i.e., $Pr(\phi(X) = T) = \prod_i (1-p_i)^{1-x_i}$.

3.1.4 Prior Distribution. Since initially we do not have any knowledge about the individual elements, we uniformly set all p_i to σ, where $0 < \sigma < 1$ is a hyper-parameter of ProbDD. There are multiple ways to determine σ based on the properties of the problem domain. If the results usually have a fixed reduction ratio, we set σ to this ratio. If the reduced subsequences usually have a fixed length m, we can set σ to m/n, where n is the length of the input sequence.

3.2 Update the model

After a set of tests, we would like to calculate the posterior probabilities conditioned on the test results so as to guide future tests using the posterior probabilities. Now assume that we have performed a series of tests on $X_1, X_2, \ldots, X_m$ with test results $R_1, R_2, \ldots, R_m$. We denote the event that testing X_i returning R_i (i.e., $\phi(X_i) = R_i$) as T_i. Then we can calculate the posterior probability of θ_i as follows.

$$Pr(\theta_i = 1|T_1, T_2, ..., T_n) = \frac{Pr(\theta_i = 1, T_1, T_2, ..., T_n)}{Pr(T_1, T_2, .., T_n)}$$

A basic method to calculate the above two joint probabilities is to enumerate the universe of subsequences, and sum up the probability of a subsequence being the optimal one for each subsequence consistent with the events. A subsequence X is consistent with a test result $T = \langle X', R \rangle$ if $\phi(X') = R$ when X is the optimal one. More concretely, we define the following function to test whether

[1] In this paper we assume $0^0 = 1$.

a subsequence is consistent with the test results.

$$con(X, \langle T_1, \ldots, T_m \rangle) = \begin{cases} 1 & con'(X, T_1) \wedge \ldots \wedge con'(X, T_m) \\ 0 & \text{otherwise} \end{cases}$$

$$con'(X, \langle X', R \rangle) = \begin{cases} \bigwedge_{x'_i=0} x_i = 0 & R = T \\ \bigvee_{x'_i=0} x_i = 1 & R = F \end{cases}$$

Based on this function, we have the following result.

$$\frac{Pr(\theta_i = 1, T_1, T_2, ..., T_n)}{Pr(T_1, T_2, .., T_n)} = \frac{\sum_{X \in \mathbb{X}} \left(x_i * con(X, \langle T_1, \ldots, T_m \rangle) * \Pi_i p_i^{x_i} (1 - p_i)^{1-x_i} \right)}{\sum_{X \in \mathbb{X}} \left(con(X, \langle T_1, \ldots, T_m \rangle) * \Pi_i p_i^{x_i} (1 - p_i)^{1-x_i} \right)}$$

Calculating the above formula is a typical weighted model counting problem [4]: we need to sum up the weight of any solution X satisfying a constraint $con(X, \langle T_1, \ldots, T_m \rangle)$, and the weight of a solution is the product of the weight of individual assignments to x_i (i.e., p_i or $1 - p_i$). However, so far we still lack an efficient algorithm to solve weighted model counting: a state-of-the-art solver often takes thousands of seconds to solve a model of thousands of variables [4], which is typical in delta debugging. This is too slow to accelerate delta debugging.

Alternatively, instead of calculating the posterior probabilities conditioned on all test results, we calculate the posterior probabilities after every single test and update the model for future calculations. Concretely, we update p_i to $Pr(\theta_i = 1 \mid T)$ after a test T. This method ignores the interaction between different tests and is not as precise as the previous one, but can be calculated efficiently.

We show how to update p_i for each i. On the one hand, if the test fails, the posterior probability is as follows.

$$\begin{aligned} & Pr(\theta_i = 1 | \phi(X) = F) \\ = & \frac{Pr(\theta_i = 1) Pr(\phi(X) = F | \theta_i = 1)}{Pr(\phi(X) = F)} \\ = & \begin{cases} \frac{Pr(\theta_i=1) \cdot 1}{1 - \Pi_i (1 - Pr(\theta_i=1))^{1-x_i}} = \frac{p_i}{1 - \Pi_i (1-p_i)^{1-x_i}} & x_i = 0 \\ \frac{Pr(\theta_i=1) Pr(\phi(X)=F)}{Pr(\phi(X)=F)} = Pr(\theta_i = 1) = p_i & x_i = 1 \end{cases} \end{aligned}$$

On the other hand, if the test passes, the posterior probability is as follows.

$$\begin{aligned} & Pr(\theta_i = 1 | \phi(X) = T) \\ = & \frac{Pr(\theta_i = 1) Pr(\phi(X) = T | \theta_i = 1)}{Pr(\phi(X) = T)} \\ = & \begin{cases} \frac{Pr(\theta_i=1) \cdot 0}{Pr(\phi(X)=T} = 0 & x_i = 0 \\ \frac{Pr(\theta_i=1) Pr(\phi(X)=T)}{Pr(\phi(X)=T)} = Pr(\theta_i = 1) = p_i & x_i = 1 \end{cases} \end{aligned}$$

Based on the above equations, we update the parameter p_i for each i after a test $\phi(X) = R$ according to the following rules.

(1) p_i remains unchanged if the ith element is included in X.
(2) p_i is set to zero if the ith element is excluded from X and the test passes.
(3) p_i is set to $\frac{p_i}{1 - \Pi_i (1-p_i)^{1-x_i}}$ if the ith element is excluded from X and the test fails.

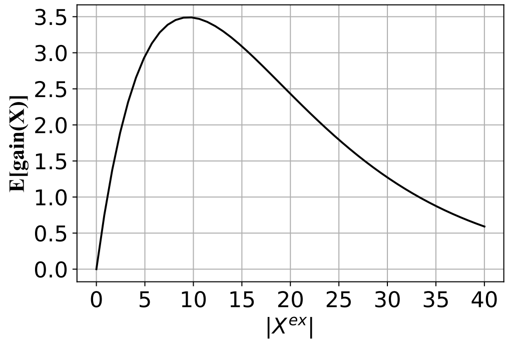


Figure 2: The relation between $\mathbb{E}[gain(X)]$ and $|X^{ex}|$

3.3 Select a Subsequence for Testing

We first define the gain of a test and then discuss how to maximize the expected gain.

3.3.1 The Gain of a Test. As we can see from the previous section, when X passes the test, the probabilities of the elements excluded from X would be set to zero, i.e., these elements should not be selected again for testing. As a result, each passed test excludes some elements from the final result. To measure how many elements a test can exclude, we define the *gain* of a test on subsequence X as the number of elements excluded if the test passes, and zero otherwise.

$$gain(X) = \begin{cases} |X^{ex}| & \phi(X) = T \\ 0 & \phi(X) = F \end{cases}$$

Here X^{ex} denotes the set of elements newly excluded when the test of X passes, i.e., $x_i^{ex} = 1$ iff $x_i = 0$ and $p_i > 0$.

Based on the probabilistic model $\langle p_1, p_2, \ldots, p_n \rangle$, we can calculate the expected gain of a test.

$$\mathbb{E}[gain(X)] = \left| X^{ex} \right| Pr(\phi(X) = T) = \left| X^{ex} \right| \Pi_i (1 - p_i)^{1-x_i}$$

Therefore, the goal of selecting a subsequence for a test is to select a subsequence X that maximizes $\mathbb{E}[gain(X)]$.

3.3.2 Maximizing the Expected Gain. Please note that simply selecting the subsequence that has the maximum probability to be equal to X^* does not necessarily lead to the maximum expected gain because the probability for it to pass the test maybe low.

To understand how to maximize the expected gain, let us first consider a simple situation where all probabilities p_i are equal. In this case, any subsequence of the same size leads to the same expected gain. Figure 2 shows the relation between $\mathbb{E}[gain(X)]$ and $|X^{ex}|$ when any p_i is 0.1. As we can see from the figure, when we remove more elements, the expected gain first increases and then decreases, with the maximum at the inflection. This is because $\mathbb{E}[gain(X)]$ is the product of two components, $|X^{ex}|$ and $\Pi_i (1 - p_i)^{1-x_i}$. The first one monotonously increases, but the rate of increase gradually decreases. The second one monotonously decreases, but the rate of decrease remains the same. Therefore, there must be a point at which the rate of decrease surpasses the rate of increase, which maximizes the expected gain.

Now let us consider the case where the probabilities are different. The first component, $|X^{ex}|$, is not affected by this change. The second component, $\Pi_i (1 - p_i)^{1-x_i}$, may lead to different values

	s1	s2	s3	s4	s5	s6	s7	s8	
	0.2500	0.2500	0.2500	0.2500	0.2500	0.2500	0.2500	0.2500	
1	s1	s2	s3	s4	s5	s6	s7	s8	F
	0.3657	0.3657	0.3657	0.2500	0.2500	0.2500	0.2500	0.3657	
2	s1	s2	s3	s4	s5	s6	s7	s8	T
	0.3657	0.3657	0.3657	0	0	0	0	0.3657	
3	s1	s2	s3	s4	s5	s6	s7	s8	F
	0.6119	0.3657	0.3657	0	0	0	0	0.6119	
4	s1	s2	s3	s4	s5	s6	s7	s8	F
	0.6119	0.6119	0.6119	0	0	0	0	0.6119	
5	s1	s2	s3	s4	s5	s6	s7	s8	T
	0.6119	0	0.6119	0	0	0	0	0.6119	
6	s1	s2	s3	s4	s5	s6	s7	s8	F
	0.6119	0	1	0	0	0	0	0.6119	
7	s1	s2	s3	s4	s5	s6	s7	s8	T
	0	0	1	0	0	0	0	0.6119	
8	s1	s2	s3	s4	s5	s6	s7	s8	F
	0	0	1	0	0	0	0	1	

Figure 3: Iterations of our algorithm

for different subsequences of the same length. To select the subsequence with the maximum value, we need to exclude the elements whose probabilities of being in X^* are the lowest.

Based on the above analysis, we use the following procedure to find a subsequence that has the maximum expected gain. Let X_p be the last subsequence that passes the test.

(1) Sort the elements in X_p ascending by their probabilities p_i.
(2) Exclude the elements one by one from X_p based on the above order until the expected gain begins to decrease.
(3) Return the subsequence with the highest expected gain.

Let $\hat{X}$ be the subsequence returned from the above procedure. The following theorem shows that $\hat{X}$ has the maximum expected gain.

Theorem 3.4. *$\mathbb{E}[gain(\hat{X})] \geq \mathbb{E}[gain(X)]$ for any $X \subseteq X_p$.*

Proof. Use $S(k)$ to denote the subsequence obtained after removing k elements in step (2). First, we prove $\forall X \subseteq X_p$, the subsequence $S(|X^{ex}|)$ which excludes the same number of elements as X but selects elements in order of increasing probability cannot have a worse expected gain. Second, we show the subsequence returned by the algorithm has the highest expected gain among $S(k)$ where $1 \leq k \leq |X_p|$. As a result $E[gain(\hat{X})] \geq E[gain(S(|X^{ex}|))] \geq E[gain(X)]$. The details can be found in Appendix. □

3.4 Revisiting the Motivating Example

Figure 3 shows a possible testing sequence of ProbDD for the motivating example. In Figure 3, each odd row represents each test, and the selected elements are shown in cells with darker colors. The last cell of each odd row shows the result of each test. Each even row represents the probability of each element after a test. The changes are shown in cells with darker colors.

Initially, all probabilities are set to 0.2500. In each iteration, ProbDD keeps excluding the element with the lowest probability until the expected gain begins to decrease. Since the initial probabilities are all equal, the selection of the first iteration is effectively random. ProbDD excludes s_1, s_2, s_3, and s_8 and the test fails, so ProbDD updates the probabilities of the removed elements based on rule (3) at the end of Section 3.2. At the second iteration, ProbDD selects four elements with the lowest probabilities, s_4, s_5, s_6, and s_7 to exclude. In this case, the test passes, so ProbDD directly sets the probabilities of the removed elements to zero according to rule (2). ProbDD sampled $\{s_2, s_3\}$ and $\{s_1, s_8\}$ to test at the third and fourth iteration, respectively. They all failed the test and the probabilities of the removed elements are updated accordingly. In the remaining iterations, the probabilities of the remaining elements have raised to a level such that only one element could be excluded at each time, and the probability of the removed element would be set to either 0 or 1 based on the test result. Finally, ProbDD stops when the probability of each element is either 1 or 0, and returns $\{s_3, s_8\}$.

As we can see from the above process, ProbDD learns from the history of tests: when removing s_8 fails, the probability of s_8 would be increased and s_8 would not be repetitively selected for removal. Furthermore, in the above process, ProbDD returns a much smaller result $\{s_3, s_8\}$ than ddmin. To reach this result, we need to remove s_4, s_5, s_6, and s_7, and the four elements need to be removed together, otherwise, the error could not be reproduced. Since ddmin uses fixed boundaries to partition subsequences, it would never remove the four elements together. On the other hand, ProbDD does not use fixed partitions, and could possibly test any subsequence in the universe $\mathbb{X}$.

4 PROPERTIES OF ProbDD

In this section, we discuss the efficiency, the correctness, and the minimality of the result.

4.1 Efficiency

Theorem 4.1. *Given input with size n, the asymptotic number of tests performed by ProbDD is bounded by O(n) in the worst case.*

Proof. First, there can be at most n passed tests as each passing test sets the probability of at least one element to 0. Second, it can be shown that there can be at most $O(n)$ failed tests before the probabilities of all elements are either zero or larger than 0.5. When the probabilities of all remaining elements are larger than 0.5, the algorithm will test elements one by one, so there could be at most $O(n)$ failed tests left. The details can be found in Appendix. □

Please note that this theorem implies that the ProbDD always terminates.

4.2 Correctness

Theorem 4.2. *The returned subset X_O of ProbDD will always maintain the property, i.e., $\phi(X_O) = T$.*

Proof. It is easy to see (1) if all tests fail, the input sequence is returned, (2) otherwise the subsequence used in the last passed test is returned. In either case, $\phi(X_O) = T$. □

4.3 Minimality

Theorem 4.3. *If monotony holds, the output of ProbDD X_O is minimal, i.e., $\forall X \subset X_O, \phi(X) = F$.*

Proof. Let s_i be the element in X_O but not in X. Since X_O is the output, we know that $p_i = 1$. It is easy to see $\frac{p_i}{1-\Pi_i(1-p_i)^{1-x_i}} = 1$ only when $\forall j \neq i, p_j = 0 \vee x_j = 1$, i.e., there exists a failed test on X' where only s_i is newly removed. Since X_O is the output, we know that $X \subset X_O \subseteq X' \cup \{s_i\}$. Since s_i is not in X, we know $X \subseteq X'$. Since $\phi(X') = F$, by monotony we have $\phi(X) = F$. □

Theorem 4.4. *If monotony and unambiguity both hold, the output of ProbDD X_O is minimum, i.e., $\forall X, |X| < |X_O| \Rightarrow \phi(X) = F$*

Proof. By the proof of Theorem3.3, minimum $X^* = \bigcap_{\phi(X)=T} X$. Since $\phi(X_O) = T$, we know $X^* \subseteq X_O$. Let us assume $X^* \subset X_O$. Then from Theorem4.3, we know that $\phi(X^*) = F$, which contradicts the definition of X^*. As a result, $X^* = X_O$, i.e., X_O is minimum. □

5 EVALUATION

As discussed before, many existing delta debugging approaches are domain-specific based on the ddmin algorithm. Specifically, a typical domain-specific delta debugging approach considers the constraints in the domain, and applies ddmin to subsequences of the elements such that the domain-specific constraints would not be violated. Since our goal is to improve ddmin, we would like to understand whether and how much ProbDD outperforms ddmin in different application domains, i.e., whether the performance of a domain-specific approach improves when replacing ddmin with ProbDD. Furthermore, we would like to investigate how ProbDD compares with ACTIVECOARSEN [12], which is the only randomized search algorithm that can be applied to delta debugging within our knowledge. To sum up, our evaluation addresses the following research questions.

- **RQ1:** How does ProbDD compare to ddmin in different application domains?
- **RQ2:** What is the impact of the parameter in ProbDD?
- **RQ3:** How does ProbDD compare with ACTIVECOARSEN?

5.1 Experiment Setup

Application Domains. Our evaluation considers the following two application domains, in each of which we picked the representative delta debugging approach based on ddmin as the target approach. We replaced the ddmin component with ProbDD in each target approach, and compared the performance with the original target approach with ddmin.

- **Trees**. The representative delta debugging approach for trees is HDD [13], and is often applied to programs where the abstract syntax tree (AST) of the program is available.
- **C Programs**. The representative debugging tool for C program is CHISEL [7], which relies on both the grammar of C language and the dependency relations between elements in programs.

We chose the two domains because they are actively studied in existing delta debugging research and there are publicly available implementations of the representative approaches based on ddmin.

To facilitate presentation, we call the original HDD with ddmin as *d-HDD*, and the version where ddmin is replaced with ProbDD as *p-HDD*. Similarly, the two versions of CHISEL is called *d-CHISEL* and *p-CHISEL*, respectively. We also use *d-version* and *p-version* if no specific approach is referred to.

Subjects. We mainly picked the subjects for evaluating the original approaches in existing publications to avoid selection bias. More specifically, we chose the following subjects.

- **Trees**. We used 30 subjects in the domain of trees. We used the 20 publicly available subjects in the benchmark for comparing HDD and Perses [17], which are C programs triggering crash and compilation bugs in GCC and Clang. The property to be preserved is to reproduce the reported bug without any undefined behavior. Since these subjects all fall into the application domain of C programs, we added 10 XML reduction tasks for diversity. We crawled a corpus of more than 1,000 XML files from repositories of XML files publicly available on the Internet, filtered out 73 of those that cannot be parsed, and randomly picked 10 XML files as subjects. The property to be preserved is to keep at least the original test coverage on an XML parser xmllint [3]. We did not use the benchmark in the original publication of HDD [13] because it is not publicly available.
- **C Programs**. We used 30 subjects in the domain of C programs. We used the benchmark for evaluating CHISEL [7], which includes 10 subjects that are C programs to be debloated to be used in embedded systems, as well as the same 20 subjects used for trees. The property for those 10 subjects is to compile successfully, pass given test cases, and contain specific functions. The property for those 20 subjects to keep is the same as that used in the application domain of trees.

In total, we used 40 subjects in our evaluation, and 20 of them are used in both application domains. The number of subjects in our evaluation is larger than all recent publications on delta debugging at top venues [7, 9–11, 13, 15, 17, 20, 21, 23] as far as we are aware. We made full use of 16 cores of the server, and the whole process of our evaluation took about 90 hours per core on average (1,441 hours in total).

Metrics. Following the existing work [7, 13, 17], we used three metrics to measure the effectiveness of a delta debugging approach in the study, i.e., *the size of the produced result*, *the processing time*, and *the number of tokens deleted per second*. We measured the size of the subjects in both domains using the number of tokens. We measured the processing time in seconds. The reduction process of each subject has a timeout limit of 3 hours. If timed out, the size of the produced result is the size of the smallest object in all passed tests and the processing time is not available. When calculating the average results, we calculated geometric means rather than arithmetic means because different subjects diverge significantly on the three metrics.

Process. To answer RQ1, we first recorded the original size for each subject. Then, we applied both d- and p-version of the approaches for each subject and recorded the size of the produced result and the processing time. Then we calculated the number of tokens deleted per second. What's more, we calculated the p-value of a paired sample Wilcoxon signed-ranked test given the size of the produced result, the number of tokens deleted per second, and the processing time of the subjects without timeout on the both p- and d-version to answer whether our approach achieves significant improvement in both effectiveness and efficiency compared to the original approaches, respectively.

Table 1: Comparison between ProbDD and ddmin

Summary	R_i	p-version			d-version			$\uparrow_R$	$p-value_R$	$\times S$	$p-value_S$	$\uparrow_T$	$p-value_T$
		R_p	S_p	T_p	R_d	S_d	T_d						
Trees	31,533	376	9	778	928	4	2,115	59.48%	0.0000	2.25	0.0000	63.22%	0.0015
C Programs	64,782	8,791	31	874	9,935	17	1,597	11.51%	0.0012	1.82	0.0000	45.27%	0.0000

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; p represents the p-versions; d represents the d-versions; $\uparrow$ denotes the improvement, where $\uparrow_X = (X_d - X_p)/X_d$; $\times S$ denotes the speedup, where $\times S = S_p/S_d$. In this table, all numbers are geometric means, and the means of process time are only calculated on the subjects where both p- and d-versions finish within the time limit.

To answer RQ2, we adjusted the values of the only parameter used in ProbDD, i.e., the initial value of probability σ. Since this experiment is time-consuming, we sampled a subset of subjects for this experiment. Considering the diversity of the reduction ratio (i.e., the ratio of the smallest returned size to the original size), we sorted the reduction ratio of all subjects and evenly selected 14 subjects with the reduction ratio from 0.005 to 0.899. These 14 subjects are xml-10, xml-5, xml-6, xml-3, clang-27747, gcc-64990, clang-27137, gcc-65383, gcc-71626, chown-8.2, mkdir-5.2.1, date-8.21, sort-8.16, and grep-2.19 as the ascending order of the reduction ratio. We ran the first half of subjects in the application domain of trees and the remaining subjects in the application domain of C programs. We changed the initial value of probability σ to 0.01, 0.05, 0.1, 0.15, 0.2, 0.25, and 0.3, respectively. For each setting, we measured the results using all the metrics. In this RQ, we did not conduct experiments on all subjects because it would take a very long time.

To answer RQ3, we selected ACTIVECOARSEN [12] as the representative random search algorithm and used the default setting in ACTIVECOARSEN. Then we created two versions of HDD and CHISEL by replacing ddmin with ACTIVECOARSEN, and the respective versions are called *a-HDD* and *a-CHISEL*. Then we compared the a-version and the p-version in all subjects in all domains.

The results of ProbDD, CHISEL, and ACTIVECOARSEN are also affected by randomness. To reduce the influence of randomness, we ran all versions affected by randomness 5 times and computed the average results. We chose 5 times because the standard deviation of the 5 running results for each subject and each approach is already less than 1% of their corresponding average results. In RQ1 and RQ3, we set σ in ProbDD to 0.1.

Implementation. We introduce the implementations for both application domains below.

- **Trees**. We adopted a recent implementation of HDD [1, 9] in Python as d-HDD and implemented p-HDD and a-HDD on top of this implementation.
- **C Programs**. We adopted the implementation of CHISEL in C++ by the original authors [7] as d-CHISEL and implemented p-CHISEL and a-CHISEL on top of it. In particular, the CHISEL implementation includes components for automatic dead code elimination (DCE) and dependency analysis (DA), and we disabled these components by using three command-line options, i.e., -skip_local_dep, -skip_global_dep, and -skip_dce, due to the following reasons. First, DCE is designed for program debloating in CHISEL and it fails most of the tests in other domains, e.g., compiler bugs triggered by unreachable code. Second, we found that the DA component produces incorrect results in some cases, e.g., when a function call is passed as a parameter, which exists in the subjects used in our evaluation. Please note that DCE and DA are disabled for all versions of CHISEL.

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.

5.2 Results and Analysis

5.2.1 Comparison between ProbDD and ddmin. Table 1 shows the overall performance of the p- and d-versions in terms of the three metrics. From Table 1, we can see that p-versions perform better than d-versions in all metrics. On average, p-versions delete 5 and 14 more tokens per second to obtain 59.48% and 11.51% smaller results than d-versions in the application domains of trees and C programs, respectively. On the subjects where both p- and d-versions finish within the time limit, p-HDD and p-CHISEL use 63.22% and 45.27% less time, respectively. All p-values are significant (< 0.05).

Detailed Results in Each Application Domain. We then investigate the detailed results of p-versions for each subject in both application domains. Table 2 shows the comparison results between p- and d-versions. From Table 2, the p-versions outperform the d-versions on 58 out of 60 subjects. Here we define the p-version outperforms the d-version on a subject if the p-version has better result in any of the three metrics and does not have worse result in any of the metrics. Only on 2 subjects, mkdir-5.2.1 and grep-2.19, the p-version performs worse than the d-version.

We analyzed the two subjects and found that, to preserve target properties, we have to keep consecutive subsequences in the returned result. In other words, if we know the element at index i is in X^*, elements at indexes $i-1$ and $i+1$ are also likely to be in X^*. This contradicts with our independence assumption, and thus p-CHISEL does not have better performance. Nevertheless, even on the two subjects where our independence assumption does not hold, the performance of p-CHISEL is only slightly worse than d-CHISEL.

> **RQ1:** On average, ProbDD improves HDD and CHISEL by deleting 5 and 14 more tokens per second to obtain 59.48% and 11.51% smaller results, respectively. On the subjects where both versions finish within the time limit, ProbDD reduces the execution time of HDD and CHISEL by 63.22% and 45.27%, respectively.

5.2.2 Impact of the parameter. We then investigate the impact of the only parameter in p-versions, i.e., the initial probability for each

Table 2: Comparison between ProbDD and ddmin: Detailed Data

D	Subject	p-version			d-version			↑$_R$	×S	↑$_T$
		R_p	T_p	S_p	R_d	T_d	S_d			
Trees	clang-22382	355	998	20.755	355	4,915	4.214	0.0%	4.9249	79.7%
	clang-22704	1,540	-	16.936	1,826	-	16.909	15.7%	1.0016	-
	clang-23309	1,327	-	3.456	13,782	-	2.302	90.4%	1.5009	-
	clang-23353	325	1,780	16.782	344	3,932	7.592	5.5%	2.2104	54.7%
	clang-25900	634	7,458	10.502	723	-	7.244	12.3%	1.4498	-
	clang-26760	397	-	19.369	624	-	19.348	36.4%	1.0011	-
	clang-27137	206	-	16.142	238	-	16.139	13.4%	1.0002	-
	clang-27747	227	4,256	40.792	315	-	16.067	27.9%	2.5389	-
	clang-31259	1,010	-	4.425	3,800	-	4.167	73.4%	1.0620	-
	gcc-59903	538	-	5.282	1,550	-	5.188	65.3%	1.0181	-
	gcc-60116	8,420	-	6.186	16,658	-	5.423	49.5%	1.1407	-
	gcc-61383	957	-	2.916	1,636	-	2.853	41.5%	1.0220	-
	gcc-61917	322	8,393	10.132	378	-	7.869	14.8%	1.2876	-
	gcc-64990	1,451	-	13.656	41,104	-	9.984	96.5%	1.3677	-
	gcc-65383	710	8,119	5.325	42,583	-	0.126	98.3%	42.3275	-
	gcc-66186	1,010	-	4.303	46,993	-	0.045	97.9%	95.1969	-
	gcc-66375	551	-	6.013	10,668	-	5.076	94.8%	1.1846	-
	gcc-70127	428	-	14.295	659	-	14.274	35.1%	1.0015	-
	gcc-70586	17,969	-	17.990	49,488	-	15.071	63.7%	1.1936	-
	gcc-71626	179	103	57.806	179	397	14.998	0.0%	3.8544	74.1%
	xml-1	30	486	11.152	170	3,731	1.415	82.4%	7.8804	87.0%
	xml-2	181	2,572	2.703	181	3,612	1.925	0.0%	1.4043	28.8%
	xml-3	236	2,508	3.462	236	3,378	2.571	0.0%	1.3469	25.8%
	xml-4	293	2,515	3.697	325	3,768	2.459	9.8%	1.5034	33.3%
	xml-5	46	509	11.493	230	2,765	2.049	80.0%	5.6086	81.6%
	xml-6	177	422	18.860	200	2,346	3.383	11.5%	5.5753	82.0%
	xml-7	54	382	12.421	54	807	5.880	0.0%	2.1126	52.7%
	xml-8	50	273	21.000	50	921	6.225	0.0%	3.3736	70.4%
	xml-9	179	2,707	1.768	195	3,240	1.472	8.2%	1.2009	16.5%
	xml-10	39	431	17.629	58	822	9.220	32.8%	1.9120	47.6%
C Programs	mkdir-5.2.1	8,321	1,429	18.530	8,291	1,417	18.709	-0.4%	0.9905	-0.8%
	rm-8.4	7,427	3,232	11.458	7,427	4,192	8.834	0.0%	1.2970	22.9%
	chown-8.2	7,445	3,704	9.834	8,286	5,297	6.718	10.1%	1.4639	30.1%
	grep-2.19	114,754	-	1.197	113,155	-	1.345	-1.4%	0.8899	-
	bzip2-1.05	51,123	-	1.797	64,686	-	0.541	21.0%	3.3208	-
	sort-8.16	51,848	-	3.354	55,336	-	3.031	6.3%	1.1066	-
	gzip-1.2.4	16,548	-	2.720	30,074	-	1.468	45.0%	1.8531	-
	uniq-8.16	14,045	9,242	5.390	14,201	-	4.598	1.1%	1.1723	-
	date-8.21	20,219	9,920	3.349	33,541	-	1.843	39.7%	1.8175	-
	tar-1.14	58,374	-	9.715	130,328	-	3.053	55.2%	3.1825	-
	clang-22382	4,028	150	113.600	4,028	453	37.616	0.0%	3.0200	66.9%
	clang-22704	1,224	2,917	62.811	1,742	3,749	48.733	29.7%	1.2889	22.2%
	clang-23309	7,267	1,006	31.193	7,272	2,365	13.266	0.1%	2.3513	57.5%
	clang-23353	7,698	694	32.418	7,741	1,911	11.750	0.6%	2.7589	63.7%
	clang-25900	2,988	1,270	59.821	3,016	2,343	32.413	0.9%	1.8456	45.8%
	clang-26760	4,970	2,286	89.504	5,241	3,345	61.087	5.2%	1.4652	31.7%
	clang-27137	14,400	2,907	55.087	15,373	4,995	31.865	6.3%	1.7288	41.8%
	clang-27747	6,270	717	233.710	6,324	1,011	165.693	0.9%	1.4105	29.1%
	clang-31259	4,166	633	70.510	4,166	1,197	37.287	0.0%	1.8910	47.1%
	gcc-59903	6,602	737	69.171	7,614	1,342	37.233	13.3%	1.8578	45.1%
	gcc-60116	9,453	886	74.234	9,611	1,750	37.493	1.6%	1.9799	49.4%
	gcc-61383	9,702	1,109	20.511	9,932	1,265	17.800	2.3%	1.1523	12.3%
	gcc-61917	13,691	947	75.679	13,691	1,607	44.597	0.0%	1.6969	41.1%
	gcc-64990	6,970	882	160.953	7,532	1,891	74.775	7.5%	2.1525	53.4%
	gcc-65383	5,065	397	97.927	5,065	1,509	25.763	0.0%	3.8010	73.7%
	gcc-66186	6,447	622	65.971	6,447	1,196	34.309	0.0%	1.9228	48.0%
	gcc-66375	5,169	854	70.631	5,169	2,243	26.892	0.0%	2.6265	61.9%
	gcc-70127	12,452	1,346	105.768	12,452	2,043	69.684	0.0%	1.5178	34.1%
	gcc-70586	8,383	908	224.533	8,383	1,677	121.572	0.0%	1.8469	45.9%
	gcc-71626	639	14	392.429	639	33	166.485	0.0%	2.3571	57.6%

element σ, based on the selected 14 subjects in both application domains (as presented in Section 5.1). The results are shown in Figure 4. The left sub-figures in Figure 4a and Figure 4b show the geometric means of the produced size, while the right sub-figures depict the geometric means of the number of tokens deleted per second. In each sub-figure, the blue line marks the performance of ProbDD. Also, we used the red line to mark the performance of the original approaches with ddmin for clear comparison.

We observe that though different σ values cause deviations in the performance, the p-versions stably outperform the d-versions with all studied σ values. Furthermore, the performance differences between different σ values is significantly smaller than the difference between the p-versions and the d-versions.

RQ2: The parameter σ has a small impact on the performance of ProbDD and ProbDD stably improves HDD and CHISEL in all parameter values we tested.

5.2.3 *Compared between ProbDD and ACTIVECOARSEN.*

Table 3 shows the overall comparison results between p-versions and a-versions on all the subjects in the application domains of trees and C programs. From this table, p-versions delete 3 and 21 more tokens per second to obtain 58.68% and 27.03% smaller size of produced result than a-versions on average in the application domains of trees and C programs, respectively. On subjects where both versions finish, the p-versions also use 58.77% and 68.65% less time. The detailed comparison results on each subject can be found in Table 4.

RQ3: On average, p-versions significantly outperform a-versions by deleting 3 and 21 more tokens per second to obtain 58.68% and 27.03% smaller results in the application domains of trees and C programs, respectively. On the subjects where both versions finish within the time limit, p-versions use 63.22% and 45.27% less processing time on the two domains, respectively.

5.3 Threats to Validity

The threat to *internal* validity mainly lies in the correctness of the implementation of p-versions and the experimental scripts. To reduce this threat, we have carefully reviewed our code.

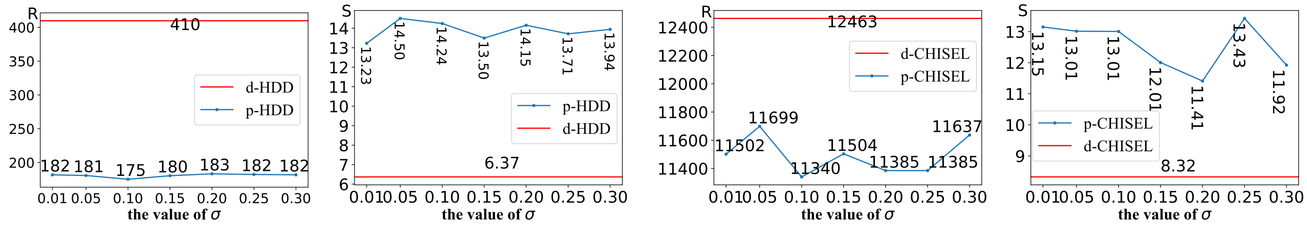
The threat to *external* validity mainly lies in the subjects and the target approaches. Regarding the subjects used in our study, we adopted the subjects used in existing publications for the two application domains, i.e., trees and C programs. Besides, to increase the subject diversity in the domain of trees, we additionally evaluated our approach on 10 XML files, which were randomly picked from the crawled corpus. In the future, we will evaluate ProbDD on more subjects. Regarding the target approaches, we adopted two representative approaches in domains of trees and C programs, i.e., HDD and CHISEL, as presented in Section 5.1.

The threat to *construct* validity mainly lies in randomness. The randomness may impact the performance of p-versions, a-versions, and d-CHISEL. To reduce this threat, we ran each of them on each subject 5 times and calculated the average results as presented in Section 5.1.

6 RELATED WORK

Delta Debugging Approaches built on ddmin. As the basic algorithm for delta debugging, ddmin was proposed by Zeller and Hildebrandt to minimize failure-inducing test inputs [23], which has been described in Section 1 and 2. Further, they proposed an extended version of ddmin, named dd, which aims to obtain a minimal difference between a passing test input and a failing test input rather than a minimal failure-inducing test input [23].

Subsequently, some approaches wrap ddmin for different domain-specific structures. Misherghi and Su [13] proposed HDD for more effective delta debugging on tree-structured data that has been described in Section 1. Inspired by HDD, modernized HDD [9], coarse



(a) Impact of σ in trees **(b) Impact of σ in C programs**

Figure 4: Impact of σ for ProbDD

Table 3: Comparison between ProbDD and ACTIVECOARSEN

Summary	R_i	p-version			a-version			$\uparrow_R$	$p-value_R$	$\times S$	$p-value_S$	$\uparrow_T$	$p-value_T$
		R_p	S_p	T_p	R_a	S_a	T_a						
Trees	31,533	376	9	778	910	6	1,887	58.68%	0.0000	1.5	0.0000	58.77%	0.0015
C Programs	64,782	8,791	31	874	12,597	9	2,788	27.03%	0.0000	3.33	0.0000	68.65%	0.0001

HDD [10], and HDDr [11] were proposed to further improve the performance of HDD. For example, HDDr is a recursive variant of HDD. Sun et al. [17] proposed Perses, which utilizes the formal syntax of a programming language to guide reduction and always produces syntactically valid subsequences. For each iteration of reduction, Perses invokes ddmin to prune the nodes in the parse tree for quantified nodes, and it proposes replacement strategies for regular nodes. CHISEL [7], implemented based on Perses, introduces dependency analysis to understand which elements need to be removed together. CHISEL also improves ddmin, and builds a decision tree model to prune the predefined sequences of ddmin during the reduction process.

Different from most existing approaches that wrap ddmin for different domains, our work aims to improve ddmin itself. Different from ddmin, ProbDD builds a probabilistic model to guide the tests and updates the model based on the test results. Our study has demonstrated that ProbDD significantly improves the performance of representative approaches built on ddmin in different application domains by replacing ddmin with ProbDD.

Among the existing approaches, CHISEL also builds statistical model to improve ddmin and thus is closely related to our work. However, CHISEL still relies on the predefined sequence of attempts in ddmin and only uses the statistical model to prioritize attempts in the sequence. Different from it, ProbDD directly selects elements based on the learned distribution. Our evaluation has demonstrated that ProbDD could significantly improve the performance of CHISEL.

Delta Debugging Approaches based on transformation templates. There are some approaches that employ transformation templates to transform an original object. GTR [8] defines two transformation templates for tree-structured data and can automatically choose which template to use in the reduction step by learning from a corpus of example data. C-Reduce [15] was proposed to solve the problem of test-case minimization, which employs plenty of source-to-source transformations for a more effective reduction on C, C++, and OpenCL programs. Although these transformation-template-based delta debugging approaches can further improve the reduction effectiveness in their domains [8, 15], they suffer from the serious efficiency problem based on the existing study [7, 17].

In this paper, we focus on solving the efficiency problem in the existing approaches built on ddmin. Improving the transformation-template-based approaches is future work.

Blackbox Optimization. Delta debugging is a blackbox optimization problem. Bayesian optimization is widely used to solve blackbox optimization problems. It builds a probabilistic model and updates the model with test results [14]. ProbDD can be viewed as a Bayesian optimization algorithm specifically designed for the delta debugging problem. Different from the classic Bayesian optimization algorithms that are designed for objective functions modeled by Gaussian process regression [6], ProbDD targets the delta debugging problem with *binary* test results. Although recently some Bayesian optimization approaches were proposed for binary objective functions [18, 24], they are designed for specific tasks, and to our knowledge, there is no existing Bayesian optimization approach that can be applied to solve the delta debugging problem.

Furthermore, heuristic search algorithms (such as the genetic search algorithms [16]) are also widely used to solve blackbox optimization problems, but similar to classic Bayesian optimization, classic heuristic search algorithms rely on continuous fitness functions. Since the test results are binary, how to design an effective fitness function to guide these algorithms is an open problem for future research.

The only heuristic search approach that can be applied to delta debugging within our knowledge is ACTIVECOARSEN [12]. It aims to find a minimal abstraction in the domain of static analyses using heuristic search. In our evaluation, we considered ACTIVECOARSEN as a baseline and the results that suggest that ProbDD outperforms ACTIVECOARSEN in improving the representative approaches in the two domains.

Recently, Xin et al. [19] also propose a software debloating approach, DEBOP, that is based on MCMC with Metropolis-Hastings sampling — a typical blackbox optimization approach. However, DEBOP targets a software debloating problem that is different from the standard one: instead of passing a testing function, the goal of

Table 4: Comparison between ProbDD and ACTIVE-COARSEN: Detailed Data

D	Subject	p-version			a-version			$\uparrow_R$	$\times S$	$\uparrow_T$
		R_p	T_p	S_p	R_a	T_a	S_a			
Trees	clang-22382	355	998	20.755	452	4,250	4.851	21.5%	4.2786	76.5%
	clang-22704	1,540	-	16.936	9,342	-	16.213	83.5%	1.0446	-
	clang-23309	1,327	-	3.456	2,836	-	3.316	53.2%	1.0422	-
	clang-23353	325	1,780	16.782	394	7,578	3.933	17.5%	4.2672	76.5%
	clang-25900	634	7,458	10.502	1,248	-	7.196	49.2%	1.4595	-
	clang-26760	397	-	19.369	544	-	19.355	27.0%	1.0007	-
	clang-27137	206	-	16.142	1,280	-	16.042	83.9%	1.0062	-
	clang-27747	227	4,256	40.792	945	-	16.009	76.0%	2.5481	-
	clang-31259	1,010	-	4.425	1,380	-	4.391	26.8%	1.0078	-
	gcc-59903	538	-	5.282	1,461	-	5.196	63.2%	1.0165	-
	gcc-60116	8,420	-	6.186	10,880	-	5.958	22.6%	1.0382	-
	gcc-61383	957	-	2.916	6,652	-	2.389	85.6%	1.2208	-
	gcc-61917	322	8,393	10.132	8,969	-	7.073	96.4%	1.4325	-
	gcc-64990	1,451	-	13.656	35,791	-	10.476	95.9%	1.3035	-
	gcc-65383	710	8,119	5.325	2,201	-	3.865	67.7%	1.3777	-
	gcc-66186	1,010	-	4.303	1,716	-	4.237	41.1%	1.0154	-
	gcc-66375	551	-	6.013	7,095	-	5.407	92.2%	1.1121	-
	gcc-70127	428	-	14.295	629	-	14.277	32.0%	1.0013	-
	gcc-70586	17,969	-	17.990	27,172	-	17.138	33.9%	1.0497	-
	gcc-71626	179	103	57.806	204	728	8.144	12.3%	7.0978	85.9%
	xml-1	30	486	11.152	70	1,116	4.821	57.1%	2.3134	56.5%
	xml-2	181	2,572	2.703	263	3,408	2.016	31.2%	1.3409	24.5%
	xml-3	236	2,508	3.462	236	6,971	1.246	0.0%	2.7795	64.0%
	xml-4	293	2,515	3.697	314	4,288	2.163	6.7%	1.7088	41.3%
	xml-5	46	509	11.493	46	1,950	3.000	0.0%	3.8310	73.9%
	xml-6	177	422	18.860	377	500	15.518	53.1%	1.2154	15.6%
	xml-7	54	382	12.421	54	603	7.869	0.0%	1.5785	36.7%
	xml-8	50	273	21.000	50	1,669	3.435	0.0%	6.1135	83.6%
	xml-9	179	2,707	1.768	207	2,900	1.641	13.5%	1.0776	6.7%
	xml-10	39	431	17.629	263	505	14.602	85.2%	1.2073	14.7%
C Programs	mkdir-5.2.1	8,321	1,429	18.530	8,398	-	2.445	0.9%	7.5798	-
	rm-8.4	7,427	3,232	11.458	15,867	-	2.647	53.2%	4.3280	-
	chown-8.2	7,445	3,704	9.834	18,120	-	2.384	58.9%	4.1245	-
	grep-2.19	114,754	-	1.197	114,871	-	1.186	0.1%	1.0091	-
	bzip2-1.05	51,123	-	1.797	55,790	-	1.365	8.4%	1.3166	-
	sort-8.16	51,848	-	3.354	71,271	-	1.555	27.3%	2.1563	-
	gzip-1.2.4	16,548	-	2.720	35,799	-	0.938	53.8%	2.9003	-
	uniq-8.16	14,045	9,242	5.390	47,209	-	1.542	70.2%	3.4958	-
	date-8.21	20,219	9,920	3.349	37,061	-	1.517	45.4%	2.2080	-
	tar-1.14	58,374	-	9.715	132,849	-	2.819	56.1%	3.4460	-
	clang-22382	4,028	150	113.600	5,626	496	31.133	28.4%	3.6488	69.8%
	clang-22704	1,224	2,917	62.811	2,288	6,425	28.351	46.5%	2.2155	54.6%
	clang-23309	7,267	1,006	31.193	7,937	5,553	5.530	8.4%	5.6403	81.9%
	clang-23353	7,698	694	32.418	7,850	3,062	7.298	1.9%	4.4421	77.3%
	clang-25900	2,988	1,270	59.821	4,462	1,971	37.797	33.0%	1.5827	35.6%
	clang-26760	4,970	2,286	89.504	5,489	5,653	36.103	9.5%	2.4792	59.6%
	clang-27137	14,400	2,907	55.087	14,456	6,597	24.266	0.4%	2.2701	55.9%
	clang-27747	6,270	717	233.710	6,322	4,072	41.139	0.8%	5.6810	82.4%
	clang-31259	4,166	633	70.510	5,894	3,089	13.890	29.3%	5.0765	79.5%
	gcc-59903	6,602	737	69.171	9,292	4,378	11.030	28.9%	6.2712	83.2%
	gcc-60116	9,453	886	74.234	12,786	4,505	13.860	26.1%	5.3561	80.3%
	gcc-61383	9,702	1,109	20.511	9,884	4,093	5.513	1.8%	3.7205	72.9%
	gcc-61917	13,691	947	75.679	14,705	4,455	15.860	6.9%	4.7718	78.7%
	gcc-64990	6,970	882	160.953	9,940	2,842	48.906	29.9%	3.2911	69.0%
	gcc-65383	5,065	397	97.927	9,127	1,377	25.283	44.5%	3.8732	71.2%
	gcc-66186	6,447	622	65.971	8,990	2,749	14.002	28.3%	4.7116	77.4%
	gcc-66375	5,169	854	70.631	6,672	3,515	16.733	22.5%	4.2211	75.7%
	gcc-70127	12,452	1,346	105.768	13,725	4,407	32.015	9.3%	3.3037	69.5%
	gcc-70586	8,383	908	224.533	11,625	3,010	66.656	27.9%	3.3685	69.8%
	gcc-71626	639	14	392.429	699	88	61.750	8.6%	6.3551	84.1%

this problem is to maximize a set of continuous objective functions. In other words, there is no binary test function and thus the MCMC algorithm applies.

7 FUTURE WORK

Unlike traditional delta debugging algorithms that search with a pre-defined order, our approach ProbDD builds a probabilistic model to estimate the probabilities of the elements to be kept in the produced result. A basic assumption of this model is that each element is independently related to the property to be preserved. However, elements may depend on each other due to structural constraints in the target domain. For example, in a tree structure, the existence of a child depends on the existence of its parent. In this paper, we ensure these structural constraints by building ProbDD into existing approaches such that these existing approaches apply ProbDD to only the subsets that would not violate the constraints. A more direct way to achieve this is to directly build these constraints in the probabilistic model. For example, in a tree of two elements, we can use two random variables to represent the probability of the parent and the conditional probability of the child when the parent is present. We do not need the conditional probability of the child when the parent is not present because we know the probability is zero. This is a future direction.

8 CONCLUSION

In this paper, we propose a probabilistic delta debugging algorithm, ProbDD, which builds a probabilistic model to estimate the probability of each element to be kept in the reduced result. ProbDD selects a subset of elements based on the probabilistic model to maximize the gain of the next test, tests whether the subset maintains the property, and improves the model based on the test results. Our algorithm terminates when the learned probabilities are either 1 or 0. Further, we prove the correctness of ProbDD, and analyze the minimality of its result and its worst-case asymptotic number of tests. We evaluated ProbDD in two application domains, i.e., trees and C programs. On average, after replacing ddmin with ProbDD, HDD and CHISEL produces 59.48% and 11.51% smaller results within the time limit, respectively. On the subjects where both versions finish within the time limit, HDD and CHISEL with ProbDD use 63.22% and 45.27% less time, respectively. The results demonstrate that learns from the test results based on a probabilistic model is a promising direction and call for future work.

Our tool and benchmarks are publicly available at:

https://github.com/anonymous-work/ProbDD

APPENDIX

The appendix contains the detailed proofs for the theorems can be found in the supplementary material.

REFERENCES

[1] Accessed: 2021. The implementation of modernized HDD. https://github.com/renatahodovan/picireny

[2] Accessed: 2021. Tensorflow tutorials. https://www.tensorflow.org/guide

[3] Accessed: 2021. xmllint. http://xmlsoft.org/xmllint.html

[4] Supratik Chakraborty, Dror Fried, Kuldeep S Meel, and Moshe Y Vardi. 2015. From Weighted to Unweighted Model Counting.. In *IJCAI*. 689–695.

[5] Holger Cleve and Andreas Zeller. 2005. Locating causes of program failures. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.* IEEE, 342–351.

[6] Peter I. Frazier. 2018. A Tutorial on Bayesian Optimization. arXiv:1807.02811 [stat.ML]

[7] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective program debloating via reinforcement learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 380–394.

[8] Satia Herfert, Jibesh Patra, and Michael Pradel. 2017. Automatically reducing tree-structured test inputs. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 861–871.

[9] Renáta Hodován and Ákos Kiss. 2016. Modernizing hierarchical delta debugging. In *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation*. 31–37.

[10] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. 2017. Coarse hierarchical delta debugging. In *2017 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 194–203.

[11] Ákos Kiss, Renáta Hodován, and Tibor Gyimóthy. 2018. 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*. 16–22.

[12] Percy Liang, Omer Tripp, and Mayur Naik. 2011. Learning minimal abstractions. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 31–42.

[13] Ghassan Misherghi and Zhendong Su. 2006. HDD: hierarchical delta debugging. In *Proceedings of the 28th international conference on Software engineering*. 142–151.

[14] Martin Pelikan, David E Goldberg, Erick Cantú-Paz, et al. 1999. BOA: The Bayesian optimization algorithm. In *Proceedings of the genetic and evolutionary computation conference GECCO-99*, Vol. 1. Citeseer, 525–532.

[15] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*. 335–346.

[16] Mandavilli Srinivas and Lalit M Patnaik. 1994. Genetic algorithms: A survey. *computer* 27, 6 (1994), 17–26.

[17] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. 2018. Perses: syntax-guided program reduction. In *Proceedings of the 40th International Conference on Software Engineering*. 361–371.

[18] Kevin Swersky, Yulia Rubanova, David Dohan, and Kevin Murphy. 2020. Amortized bayesian optimization over discrete spaces. In *Conference on Uncertainty in Artificial Intelligence*. PMLR, 769–778.

[19] Qi Xin, Myeongsoo Kim, Qirun Zhang, and Alessandro Orso. 2020. Program debloating via stochastic optimization. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*. 65–68.

[20] Andreas Zeller. 1999. Yesterday, my program worked. Today, it does not. Why? *ACM SIGSOFT Software engineering notes* 24, 6 (1999), 253–267.

[21] Andreas Zeller. 2002. Isolating cause-effect chains from computer programs. *ACM SIGSOFT Software Engineering Notes* 27, 6 (2002), 1–10.

[22] Andreas Zeller. 2009. *Why programs fail: a guide to systematic debugging*. Elsevier.

[23] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* 28, 2 (2002), 183–200.

[24] Yehong Zhang, Zhongxiang Dai, and Bryan Kian Hsiang Low. 2020. Bayesian optimization with binary auxiliary information. In *Uncertainty in Artificial Intelligence*. PMLR, 1222–1232.

APPENDIX

In this appendix, we give detailed proofs of the theorems.

Theorem 3.4. *$\mathbb{E}[gain(\hat{X})] \geq \mathbb{E}[gain(X)]$ for any $X \subseteq X_p$.*

Proof. To prove this theorem, we first prove two lemmas. We denote the sorted sequence of elements after step (1) as $\langle \hat{s}_1, \hat{s}_2, ..., \hat{s}_m \rangle$, where $\hat{s}_i$ denotes index of the ith elements in the original subsequence X_p. We further use $S(k)$ to denote the subsequence obtained after removing k elements in step (2), i.e., removing elements at indexes $\hat{s}_1, \hat{s}_2, \ldots, \hat{s}_k$ from X_p.

Lemma 1. *$\forall X \subseteq X_p, \mathbb{E}[gain(S(|X^{ex}|))] \geq \mathbb{E}[gain(X)]$.*

Proof. Recall that $\mathbb{E}[gain(X)] = |X^{ex}| \Pi_i (1 - p_i)^{1-x_i}$. To compare the two expected gains, we compare the two components of an expected gain, $|X^{ex}|$ and $\Pi_i (1 - p_i)^{1-x_i}$, separately.

- First, it is easy to see $|S(|X^{ex}|)^{ex}| = |X^{ex}|$.
- Second, we denote the indexes of the elements in X^{ex} in ascending order of probabilities as $s_1, s_2, ..., s_{|X^{ex}|}$. Then We have $p_{\hat{s}_1} \leq p_{s_1}, p_{\hat{s}_2} \leq p_{s_2}, ..., p_{\hat{s}_{|X^{ex}|}} \leq p_{s_{|X^{ex}|}}$, which implies $\prod_i (1 - p_{\hat{s}_i}) \geq \prod_i (1 - p_{s_i})$

Therefore, the lemma holds. □

Lemma 2. *For any $1 \leq i < j \leq m-1$, we have that $\frac{\mathbb{E}[gain(S(i))]}{\mathbb{E}[gain(S(i+1))]} < \frac{\mathbb{E}[gain(S(j))]}{\mathbb{E}[gain(S(j+1))]}$.*

Proof.

$$\begin{aligned}
& i < j \\
\Rightarrow & 1 - p_{\hat{s}_{i+1}} \geq 1 - p_{\hat{s}_{j+1}} \\
\Rightarrow & \frac{1}{1 - p_{\hat{s}_{i+1}}} \leq \frac{1}{1 - p_{\hat{s}_{j+1}}} \\
\Rightarrow & \frac{i}{(i+1)(1 - p_{\hat{s}_{i+1}})} < \frac{j}{(j+1)(1 - p_{\hat{s}_{j+1}})} \\
\Rightarrow & \frac{\mathbb{E}[gain(S(i))]}{\mathbb{E}[gain(S(i+1)]} < \frac{\mathbb{E}[gain(S(j))]}{\mathbb{E}[gain(S(j+1)]}
\end{aligned}$$

□

Based on the two lemmas, we prove our theorem. Let X be any subsequence such that $X \subseteq X_p$. From lemma 1 we know that $\mathbb{E}[gain(S(|X^{ex}|))] \geq \mathbb{E}[gain(X)]$. Below we will show $\mathbb{E}[gain(\hat{X})] \geq \mathbb{E}[gain(S(|X^{ex}|))]$.

- If $|X| \geq |\hat{X}|$, we know the above formula holds directly from step (2) of the procedure.
- Now let us consider the case where $|X| < |\hat{X}|$. From step (2), we know that
$$E[gain(\hat{X})] = E[gain(S(|\hat{X}^{ex}|))] > \mathbb{E}[gain(S(|\hat{X}^{ex}| + 1))].$$
From lemma 2, we know that
$$\frac{\mathbb{E}[gain(S(k))]}{\mathbb{E}[gain(S(k+1))]} > \frac{\mathbb{E}[gain(S(|\hat{X}^{ex}|))]}{\mathbb{E}[gain(S(|\hat{X}^{ex}| + 1))]} > 1$$
for any $k > |\hat{X}^{ex}|$. Therefore,
$$\mathbb{E}[gain(S(k))] > \mathbb{E}[gain(S(k+1))]$$
for any $k > |\hat{X}^{ex}|$. As a result,
$$\begin{aligned}
& \mathbb{E}[gain(S(|\hat{X}^{ex}|))] \\
> & \mathbb{E}[gain(S(|\hat{X}^{ex}| + 1))] \\
> & \ldots \\
> & \mathbb{E}[gain(S(|X^{ex}|))]
\end{aligned}$$

Putting the above together, we know that the original theorem holds. □

Theorem 4.1. *Given input with size n, the asymptotic number of tests performed by ProbDD is bounded by O(n) in the worst case.*

Proof. Recall the updating rules in section 3.2, in each iteration, we perform one test,

- If test passes, at least one element's probability will be set to 0.
- If test fails, at least one element's probability will be increased by
$$\frac{p_i}{1 - \Pi_i (1 - p_i)^{1-x_i}} - p_i = \frac{p_i * \Pi_i (1 - p_i)^{1-x_i}}{1 - \Pi_i (1 - p_i)^{1-x_i}}$$

In order to show the total number of tests is bounded by O(n), we consider the above two cases separately.

First, the operation to set the probability to 0 can be performed no more than n times as there are n elements in total.

Second, we will show the number of possible probability increases is also bounded by O(n).

Using the notations from the proof of Theorem 3.4, below we prove three lemmas first.

Lemma 3. *$|\hat{X}^{ex}| \leq \frac{1}{\sigma}$, where σ is the initial probability, $0 < \sigma < 1$.*

Proof. First, if $|\hat{X}^{ex}| = 1$, then $1 < \frac{1}{\sigma}$ is satisfied.
Second, if $|\hat{X}^{ex}| \geq 2$,

$$\begin{aligned}
& \mathbb{E}[gain(S(|\hat{X}^{ex}|))] \geq \mathbb{E}[gain(S(|\hat{X}^{ex}| - 1))] \\
\Rightarrow & (1 - p_{|\hat{X}^{ex}|}) * |\hat{X}^{ex}| \geq |\hat{X}^{ex}| - 1 \\
\Rightarrow & |\hat{X}^{ex}| \leq \frac{1}{p_{|\hat{X}^{ex}|}}
\end{aligned}$$

So $|\hat{X}^{ex}| \leq \frac{1}{\sigma}$ as $p_{|\hat{X}^{ex}|} \geq \sigma$. □

Lemma 4. *The number of probability increases is bounded by O(n) before the probabilities of all elements become zero or larger than 0.5.*

Proof. We will first show that the probability increase has a lower bound. Let us assume that there exist elements whose probabilities are larger than 0 and smaller than or equal to 0.5. Let the smallest probability among them be $0.5 - \epsilon$, where $\epsilon \geq 0$.

Then from Lemma 3,

$$\begin{aligned}
& \mathbb{E}[gain(\hat{X})] \geq E[gain(S(1))] = (1 - (0.5 - \epsilon)) * 1 \geq 0.5 \\
\Rightarrow & \Pi_i (1 - p_i)^{1-x_i} \geq \frac{0.5}{|\hat{X}^{ex}|} \geq 0.5\sigma
\end{aligned}$$

As a result, the probability increment

$$\begin{aligned} & \frac{p_i * \Pi_i(1-p_i)^{1-x_i}}{1-\Pi_i(1-p_i)^{1-x_i}} \\ \geq & \frac{\sigma}{\frac{1}{\Pi_i(1-p_i)^{1-x_i}} - 1} \\ \geq & \frac{\sigma}{\frac{1}{0.5\sigma} - 1} \end{aligned}$$

So if at least one element has a positive probability less than or equal to 0.5, the minimal probability increment is $\Delta = \frac{0.5\sigma^2}{1-0.5\sigma}$, which is a constant. As a result, the maximal number of probability increases before the probabilities of all elements become zero or large than 0.5 is at most $\frac{0.5n}{\Delta} = O(n)$. □

Lemma 5. $\mathbb{E}[S(1)] > \mathbb{E}[S(2)]$ *when* $p_i > 0.5 \vee p_i = 0$ *for any* i.

Proof. Let p_a be the smallest positive probability and p_b be the second smallest positive probability. Then we have $\mathbb{E}[S(1)] = (1-p_a) > 2(1-p_b) * (1-p_a) = \mathbb{E}[S(2)]$ □

As a result, from Lemma 4, it takes O(n) probability increases before the probabilities of all remaining elements are larger than 0.5. From Lemma 5, we know the algorithm removes one element for tests at a time after the probabilities of all remaining elements are larger than 0.5. Since a failed test with one removed element will set the probability of this element to 1, we know there is at most $O(n)$ probability increases after the probabilities of all remaining elements are larger than 0.5. Therefore, the number of probability increases is bounded by O(n).

Putting the above together, we know that the original theorem holds. □