Information Theory Approach to Reducing Failure Inducing Input

Mike Fairhurst
Google Inc., Email: mfairhurst@google.com

Debugging software is easier with examples of minimal inputs that reproduce defects. This paper introduces "entropy debugging'" an improved algorithm to simplify failure inducing inputs that outperforms its predecessor "delta debugging". By taking a statistical approach to a generalizable set minimization problem, we are able to treat the pass/fail results of fitness tests as a stream of binary information. Entropy debugging leverages data compression techniques to minimize this stream, meaning fewer fitness tests are performed and the algorithm approaches the upper bound of possible performance in both high entropy (already well minimized) and low entropy (not well minimized) cases.

1 Introduction

Known reproductions of software defects often facilitate efficient debugging of that software, however, reproductions often contain extraneous input, complicating the troubleshooting process. Fuzz testing, a technique where randomized inputs are fed to a program in search of unwanted behavior such as crashes, is particularly known to produce overly complex defect-inducing inputs. Therefore, it is sensible to employing algorithms capable of minimizing these. Additionally, these algorthms can be valuable for for simplifying reproductions discovered by developers or end-users.

One approach to simplifying failure-inpucing input is employing a simple iterative algorithm which removes one character of that input at a time. For each character that is removed, the algorithm checks that the program still exhibits the defect of interest with the simpler input, and in this case the extraneous character is removed.

Due to the complexity of real world programs, the removal of one character at the end of the input may affect the necessity of a character at the beginning. Consider a parser that crashes with unclosed parentheses given the input string: '(((()))'. None of the open parenthesis may be removed from the input without suppressing the crash until all closing parentheses are first removed. To produce higher qualitity minimizations, which we refer to as 1-minimal (no single character may be removed from the reproduction without compromising it), the simple iterative algorithm can be iterated to a fixed point. This concept of n-minimality is defined by Zeller [1].

Reaching 1-minimality in this way requires $O(n^2)$ steps where n is the size, of the failure-inducing input, and each step requires running the defective program, which may be computationally expensive. It is also worth noting that in real world use cases, extraneous input is also often clustered and may constitute the majority of the reproduction. A strategy that only removes one character at a time cannot exploit these patterns to achieve improved efficiency in real-world scenarios.

Delta Debugging is an algorithm by Zeller [1] which can offer improved real-world performance by removing one or more characters per test against a target program. Delta Debugging resembles a binary search where at the nth stage, 1/nth of the failure-inducing input is deleted and checked against the target program. Eventually, one nth of the failure-inducing input will be a single character. At this point the behavior of the Delta Debugging algorithm matches the naive iterative algorithm so that a 1-minimal reproduction may be found.

The bisection approach taken by Delta Debugging has the potential to greatly reduce a reproduction in few tests when the failure-inducing input contains large volumes of extraneous content. However, if a failure-inducing input is already close to minimal, the algorithm typically does not produce working reproductions when removing multiple characters. In cases where the algorithm is simplifying reproduction which are already close to 1-minimal, Delta Debugging effectively functions as a slower version of the single character iteration algorithm. This inconsistency in potential performance is consistent with the finding by Yu et. al that the usefulness of Delta Debugging can vary wildly in real world scenarios [2].

Hierarchical Delta Debugging [3] is a proposed variation that may improve the effectiveness of these early steps in the Delta Debugging algorithm for certain target programs. HDD may be employed when the target program operates on tree-like data such as parsers and certain data decoders. A developer may create a module which produces an abstract syntax tree (AST) from a failure-inducing input. HDD then operates by removing branches of this tree, which is an effective approach to quickly reduce failure-inducing inputs in certain cases. This technique does require additional effort by the developer, and is less suitable for many programs such

as image decoders. It also may be less useful to fuzz testers who wish to produce and simplify malformed inputs. Lastly, Yu et. al found that Hierarchical Delta Debugging did not yield definite improvements over Delta Debugging in efficiency in real world application [2].

Modern "Grey Box" fuzz testing software presents a scenario that necessitates a simplification algorithm that efficiently reduces both noisy and highly reduced data samples. Grey Box fuzz testing is a widely utilized and highly efficient technique to automate the discovery of defects in software. Grey box fuzz testing frameworks execute a target piece of software with code coverage collection enabled. This method is also known as "code coverage guided fuzz testing." This method involves maintaining a corpus of inputs which execute different code paths inside the target program. This corpus is then mangled and spliced to produce new candidate inputs, which are added to the corpus if the corresponding execution involves new branches in the target program. For example, if the target program is a compiler, one sample may hold a class

these two samples eventually yields a method definition, which is added to the corpus. A minimized corpus is more likely to yield novel new samples during the random splicing process. Extraneous data, such as lengthy comments in the case of fuzz testing compilers, reduce the likelihood that splicing involves data of interest (e.g. the class and function definitions). Grey Box fuzz testing software may then wish to employ an algorithm which is capable of efficiently simplifying all samples before adding them to the corpus, where these unminimized samples range from nearly being 1-minimal, to containing large sections of extraneous input.

In this paper we present "Entropy Debugging," a new algorithm which demonstrates superior performance as compared to Delta Debugging in various scenarios. Through an information theory based approach, our algorithm achieves more efficient most of Zeller's original data examples [1] in significantly fewer tests. Moreover, our algorithm demonstrates near optimal performance on inputs which are already near 1-minimal. To support our claims we include quantitative results on generated data as well as Zeller's test cases. Based on these findings, we argue our algorithm represents a compelling choice for reducing failure-inducing inputs in various scenarios, and anticipate its effectiveness in other related areas as well.

2 Improved Algorithm Design: A Data Compression Approach

To construct an algorithm with improved efficiency, we apply optimistic assumptions that simplify the problem of reducing a failure-inducing input. By applying proper design considerations, these assumptions make it possible to produce data compression mappings which may be reinterpreted as test invocations in search of a 1-minimal failure inducing input. Our algorithm demonstrates high real-world performance even when these optimistic assumptions do not hold, as evidenced by quantatative data analysis conducted in this study. The following section provides a detailed overview of our information-theory based approach.

For this section we introduce the concept of a minimal ordered set α. We define α to be the smallest possible ordered set which satisfies condition $f(\alpha') = true$ for some function f. To facilitate algorithm design, we assume that any superset α' of the minimal set α also satisfies the condition $f(\alpha') = true$. These definitions and assumptions form a foundation for our proposed problem of set simplification. In this problem, α' is a known non-minimal reproduction of a defect, and $f(i)$ is true when the program of interest crashes in the relevant manner given input i. The objective of our algorithm is to find the minimal reproduction, represented by α, in minimal invocations of function f.

2.1 Simplification Stream Forms and Performance Lower Bounds

Simplifying a non-minimal set α' involves identifying the important and unimportant members within α'. In order to leverage data compression techniques in the search for the minimal set α it is necessary to translate α into a binary stream.

We introduce the concept of *simplification stream form*, which represents each member e in α' as "i" if $e \in \alpha$ and "u" if $e \notin \alpha$. For example, a non-minimal set α' which contains three "important" members is expressed as iii in simplification stream form. Three unimportant members is expressed as uuu. A pair of important members followed by a pair of unimportant members is represented as $iiuu$. This form enables the application of information theory techniques to the problem of simplifying failure-inducing input.

The problem of minimizing a non-minimal failure-inducing input α' can therefore be expressed as a search for $ssf(\alpha')$ in as few invocations of test function f as possible. The simplification stream form encapsulates all the essential information required to derive the minimized input α from α' without additional invocations of f.

Simplification stream forms also make clear an upper bound on the maximum steps required by a simplification algorithm. The simplification stream form of any input can always be derived in $|\alpha'|$ steps, where at every ith step invokes test function f with the ith member of α' is removed.

$ssf(\alpha') = [f(\alpha[0:i] + \alpha'[i+1:L]) for i in\ range(0, L)]$

Algorithms such as Delta Debugging, a predecessor to our approach, have demonstrated the ability to minimize failure-inducing inputs in fewer steps than this maximum bound. However, the Delta Debugging algorithm will exceed this maximum bound on certain inputs, particularly those that are close to being 1-minimal. Our algorithm is designed not to exceed this bound.

2.1.1 Simplification Stream Forms and Entropy

Simplification stream forms possess a key property from the field of information theory: Entropy. The entropy of a simplification stream form represents the amount of information contained in it, and can be seen as the minimum number of yes or no questions required to derive the form.

When $ssf(\alpha')$ is low entropy and contains limited information, it suggests the potential to reduce α' in fewer than $|\alpha'|$ invocations of f. Conversely, high entropy indicates that the minimum execution steps to reduce α' may be close to the maximum bound of steps to reduce it.

Considering the practical implications and limitations, we propose the use of of entropy to establish a lower bound the required steps to simplify a failure-inducing input. It is important to note that the validity of this measure as a lower bound depends heavily on the exact properties and constraints of the test function f, as well as further developments in the field. However, we propose that $H(ssf(\alpha'))$ represents a practical lower bound on the fewest steps required to minimize the failure-inducing input α', where the entropy function H quantifies the information contained in $ssf(\alpha')$.

Quantifying the entropy of a simplification stream is a statistical process that is discussed in detail later in this study.

2.1.2 Lower Bounds of Near-Minimal Inputs

The entropy of a simplification stream form is only one lower bound in our problem domain. Consider the simplification stream form of 1-minimal failure inducing input α. Every member of α is important by definition. Therefore the simplification stream form of α is *iii...i* with a length of n where $n = |\alpha|$. This simplification stream form has very low entropy, however, it cannot be distinguished from other simplification stream forms in fewer than n tests.

Theorem 2.1. *Let α be the minimal failure-inducing input and α' be the non-minimized failure-inducing input of length L.*

The simplification stream form s of input α' cannot be determined in fewer than n invocations of f where n is the length of α.

Lemma 2.2. *If $L = 1$, then $f()$ will evaluate to true iff α is and false iff α is α'. This uniquely determines s.*

Lemma 2.3. *Let s' be a simplification stream form of length $L-1$, and let tests $f(i_0), f(i_1), \ldots f(i'_n)$ be the minimal set of tests to uniquely determine s to be one of $s'i$ or $s'u$. If s is $s'i$ and test $f(i_x)$ which evaluates to true, then $\alpha[L-1] \in i_x$. If s' is $s'i$, and $\alpha[L-1] \notin i_x$, then $f(i_x)$ will always evaluate to false. This represents a contradicton, as $f(i_x)$ was stated to be one of a minimal set of tests, and a test which differentiates nothing cannot be part of the minimal set of tests.*

Assume s is $s'u$ and one of the aforementioned minimal steps $f(i_x)$ evaluates to true. If $\alpha[L-1] \notin i_x$, then this test proves s cannot be $s'i$. If $\alpha[L-1] \in i_x$, then this test may be substituted with test $f(i_x - \alpha[L-1])$, which will still evaluate to true, and therefore the altered set of tests will still uniquely determine s' and disprove $s = s'i$.

Therefore, $s'i$ requires $n'+1$ tests to differentiate against all other possible simplification stream forms of length L.

Corrollary 2.4. *The above argument can be applied where s is one of is' or us', and can be applied where there exists s' is divided into two halves s'' and s''', and s is one of $s''is'''$ or $s''us'''$.*

Proof. The theorem that a simplification stream form cannot be determined in fewer than n invocations of f where n is the length of $alpha$ is proved by induction.

For any s, it can be determined in no fewer than $n'+0$ steps where s' has n important components, or $n'+1$ steps where s' has $n-1$ important components. Therefore, s can be determined in a minimum of n steps where n is equal to the number of important components in s. □

This proof indicates that not all low entropy simplification stream forms indicate opportunities to more quickly simplify failure-inducing input. A simplification stream form such as *uuuiuuu* may be possible to minimize in very few steps, however the opposite form *iiiuiii* cannot.

2.1.3 Calculating Lower Bound Performance

With these two lower bounds identified, we can combine them into a single, more precise equation:

$$lb(\alpha, \alpha') = max(|\alpha|, H(ssf(\alpha'))) \quad (1)$$

2.1.4 Statistical Models of Simplification Stream Forms

Quantifying the entropy of a non-minimized set α' requires dividing the simplification stream into discrete events e and estimating probability $p(e)$.

In the simplest statistical model, we say only $p(i) = c$ for some constant c. This model assumes only that each member of α' in stream simplification has a certain probability of being important, and inverse probability of being unimportant. This statistical model does not assume any correlation between the adjacent members α'.

In real world scenarios, important and unimportant sections of an input tend to be clustered into contiguous sections. This is true in the original examples provided by Zeller [1] and this matches our experience. In this paper, we propose the use of Markov transitions as a powerful model for predicting simplification stream forms to capture these correlations between the importance of neighboring characters in a failure-inducing input. A Markov transition model requires estimating probabilities $p(iu)$ and $p(ui)$.

The entropy rate of a markov model M is defined as

$$H(M) = -\sum ijp(i) * p(ij) * log(p(ij))$$

Therefore, the entropy of $ssf(\alpha')$ is the entropy of the initial state, summed with the entropy rate per subsequent states. Therefore,

$$H(ssf(\alpha')) = H(M) * (L-1) - \sum ip(i) * log(p(i))$$

Additionally, the length of the length of the minimal input α may be estimated as $L' = p(i) * L$. This yields a lower bound for minimizing failure inducing inputs that are modeled by markov chains as follows:

$$lb(M, \alpha') = max(p(i) * |\alpha'|, H(ssf(\alpha'))) \quad (2)$$

2.2 f-Guided Decision Trees

Each bit of data in a compressed artifact provides information that the decoder uses to reconstruct the original content. This can be described as a process where all possible sets of original content are considered, and each bit in the compressed data narrows down the set of possibilities. Eventually, the original content is the only remaining possibility and the artifact has been fully decompressed.

In the case of simplifying failure-inducing input, we wish to create a decision tree in which every branch represents the result from a single invocation of our function f. The leaves of this tree should contain a simplification stream form. Reaching a leaf therefore allows us to derive the minimal failure-inducing input α from the non-minimal input α'.

We define f-guided decision trees as follows. Recall that α is the minimal subset of α' such that $f(\alpha) = true$, and the exact value α is not known. Let $ssf(\alpha', x)$ be the simplification stream form for a candidate minimal input $x \subseteq \alpha'$. For example, if α' is *xy*, then α is one of , *x*, *y*, or *xy*, which have corresponding simplification stream forms *uu*, *iu*, *ui*, and *ii*. The simplification stream form function also allows for an inverse $ssf^{-1}(ssf(\alpha', x), \alpha') = x$ which produces the candidate minimum input associated with a candidate simplification stream form. With these definitions, let branch(left, right) be a branch in a valid f-guided decision tree. In this case there must exist a single test input v such that the test $f(v)$ evaluates to *true* if $\alpha \in ssf^{-1}(\alpha', l)$for all$l$inleaves(left), and $f(v)$ evaluates to false if $\alpha \in ssf^{-1}(\alpha', l)$for all$l$inleaves(right). In other words, there must be a single test execution that eliminates all leftward leaves as possible simplification stream forms of the input, or all rightward simplification stream forms of the input.

In this scenario, the average case performance of our algorithm in terms of invocations of function f is equal to the depth of each leaf multiplied by the probability of its corresponding simplification stream form. For example, reaching a leaf of depth 3 requires three invocations of f. If this leaf is selected 10% of the time, it contributes on average 0.3 executions of function f to the overall cost of the algorithm. To optimize the performance of our algorithm, we wish to minimize the following sum:

$\sum_{\text{leaf} \in \text{tree}} \text{depth}(\text{leaf}) * p(ssf(\text{leaf}))$

The probability of a leaf being selected by the decision tree can be estimated by evaluating the probability of its simplification stream form for a given Markov transition model.

2.2.1 Dividing Stream Simplification Forms

Constructing an f-guided decision tree containing all simplification stream forms of length L becomes impractical for large values of L due to the exponential growth of possible forms, reaching 2^L. It is therefore a practical requirement to first divide α' into smaller segments.

Each decision tree should efficiently distinguish the simplification stream form for a subset of the unminimized input α' when evaluated. After all decision trees for all subsets have been evaluated, the complete simplification stream form of α' is discovered and the minimized form α may be easily derived.

One approach for dividing a stream form into segments is to choose a length $L' < L$, and divide α' into chunks of size L'. However, this approach is not well suited for low entropy non-minimized failure-inducing inputs. Consider an unreduced input α' of length $1,000$, with a corresponding minimized input α of length 1. L/L' segments of α' are each reduced by the execution of one decision tree. By design, the fewest number of invocations of f required to evaluate an f-guided decision tree is 1. Therefore, reducing α' requires a minimum of L/L' invocations of f, while the size of L' is still limited by exponential growth. Effectively, the 1000 character unminimized sample can only be reduced in small chunks such as 20 characters at a time. This is undesirable when the unminimized input α' is sufficiently large and the minimized input α is sufficiently small.

2.2.2 Binary Run Length Encoding

There is a more fitting approach that involves only a modest increase in design complexity, using a binary run length encoding. In our binary run length encoding, a simplification stream form may be expressed as a sequence of integers indicating the count of *u*s between each *i*. For example, *uuii* is expressed as the integer sequence 2 0, or two *u*s followed by an *i* and then zero *u*s followed by an *i*.

We define the function $rl(\text{form})$ to equal the count of u in a form if form matches the expression $^u * i\$$. That is, $rl(i) = 0$, $rl(ui) = 1$, $rl(uui) = 2$, etc.

With this binary run length coding, α' is divided into n sections dynamically, where $n = |\alpha|$. Each decision tree therefore produces an integer result indicating the run length of the current section. For instance, a decision tree may decide between 0 (*i*), 1 (*ui*), 2 (*uui*), ... up to some reasonable maximum integer. The probability of each run may be estimated by a suitable markov model, and a probabiltiy threshold may cap the maximum run length. Additionally, the maximum run length should not exceed the remaining length of α'. Lastly, an arbitrary maximum can be chosen. Since the size of the tree grows linearly with run length, rather than exponentially, a larger maximum cap can be practically selected with this approach.

Reducing each segment of an unminimized input α' involves use of a decision tree which must invoke the target function f at least once. With sufficiently high caps on run length, these decision trees will be evaluated n times where $n = |\alpha|$. This is a desirable property because this is exactly the upper bound of the maximum tests that should be required to simplify α'.

2.2.3 f-Guided Leaf Order

An f-guided decision tree is guaranteed to be valid if its leaves, and those leaves' corresponding run length simplifications stream forms, follow the order relation $a \prec b \iff rl(a) < rl(b)$.

For example, if a branch contains left leaves *ui*, *uui*, and the right contains *uuui*, *uuuui*, this is properly ordered. To evaluate this branch point in this tree, we may test the removal of the first three characters of α' by evaluating $f(\alpha[3:L])$. If $f(\alpha[3:L])$ evaluates to true, then $ssf(\alpha')$ begins with *uuu* and therefore cannot be *ui* or *uui*. If the test evaluates to $false$, then $ssf(\alpha')$ cannot begin with *uuu*, and must be one of *ui* or *uui*. This is therefore a valid step in a valid f-guided decision tree. If, however, the left branch contains leaf *uuuuui* or the right branch contains leaf *i*, then the test $f(\alpha[3:L])$ will not always exclude the entire set of leftward leaves or the entire set of rightward leaves, and is not a valid f-guided tree.

Generally, test value v for invocation $f(v)$ in such ordered trees is determinable by computing $d = \min_{\text{leaf} in \text{right}} rl(\text{leaf})$. This minimum integer value d represents the fewest unimportant characters of all rightward cases, and exceeds maximum unimportant characters in all leftward cases. Therefore, if $f(\alpha'[d:L])$ evaluates to $true$, then none of input characters $\alpha'[0]$ through $\alpha'[d]$ are members of the minimal input α, and one of the rightward leaves accurately describes the current segment. If $f(\alpha'[d:L])$ evaluates to $false$, then at least one of input characters $\alpha'[0]$ through $\alpha'[d]$ is a member of the minimal input α, and instead one of the leftward leaves accurately describes the current segment.

Theorem 2.5. *An f-guided decision tree of run length simplification stream forms is guaranteed to be valid if the assocatiated forms of its leaves follow the order $a \prec b \iff rl(a) < rl(b)$.*

Lemma 2.6. *Let the minimal failure-inducing input α be a subset of the non-minimized input α'.*

Let the leaves of our tree be runlength forms $rl(0)$ (i), $rl(1)$ (ui), $rl(2)$ (uui), ... $rl(n)$ ($uu...u_ni$).

Let there be an non-negative integer offset $p < |\alpha'|$.

Let each leaf $rl(x)$ correspond to all candidate sets such that $c = ssf^{-1}(P \| rl(x) \| S)$ for any P, S, where $|P| = p$ and $|ssf(c)| = |\alpha'|$.

Recall that our test function $f(v)$ is assumed to be defined as $v \subseteq \alpha$. Our proof holds if for every branch of the tree there is a test value v, such $v \supseteq \alpha$ when α is a leaf of the left branch of the tree

Lemma 2.7. *For every branch in the tree, let j be an integer such that all left leaves are run lengths of form form $rl(x)$ where $x < j$, and all right leaves are of form $rl(x)$ where $x \geq j$. This establishes that the leaves are properly ordered.*

Lemma 2.8. *Let c be a candidate minimized input, corresponding to a runlength form $rl(e)$, such that $c = ssf^{-1}(P \| rl(e) \| S)$. If candidate c is the actual minimum input α, then $\alpha'[e] \in \alpha$ and for all members $m = \alpha'[0:e]$, $m \notin \alpha$.*

Lemma 2.9. *Let α be $ssf^{-1}(uuu...u_ai)$" for some $a < n$. If $a < j$ then one of $\alpha \in v$, Since $\alpha[j+q] \in \alpha'$, then $\forall e > j + q + c \rightarrow e \in \alpha'$. Since $t(v)$ is true and $t(x) = x \supset \alpha'$, it holds that $\forall e > j + q + c \rightarrow e \in v$.*

Lemma 2.10. *Assume that $\forall e \leq j + q + c \rightarrow e \notin v$. In this case, v is not a superset of any set "described by" any $uuu...j-1$. Therefore, the there is some value v that bisects all symbols in one invocation of t. (Once again, note that this implies rightwards traversal on $true$).*

Proof. Since a value v exists which bisects the symbol set *i*, *ui*, *uui*, ... $uuu..._ji$, $uuu..._{j+1}i$, ... $uuu..._ni$ for any j given $t(v)$, this problem is functionally equivalent to any ordered decision tree on integers using $>$, and the resulting tree will be equivalent to a *runnable identity encoding*. □

2.2.4 Huffman Coding

The Huffman coding algorithm produces a tree with minimum average depth for a set of symbols with corresponding probabilities. In the design of our algorithm, we explored whether Huffman Coding could generate optimal f-guided trees for an alphabet that consists of run-length codes (*i*, *ui*, *uui*, ...). We found that this is not the case without alteration to the Huffman algorithm.

Huffman Coding is a greedy algorithm which takes the two least likely symbols x and y and removes them from the alphabet. A new symbol $branch(x,y)$ is inserted into the alphabet with associated probability $p(branch(x,y)) = p(x) + p(y)$. The Huffman algorithm repeats this until the alphabet has size 1.

For a huffman code to be a valid f-guided tree, then for each branch there must exist a single value v such that $f(v)$ reduces the possible simplification stream forms of α' in agreement with the branch taken.

We prove that huffman codes are not all valid f-guided trees by example. Let $tree = \text{Huffman}(i, ui, uui)$. Let $p(i) = 0.2$, $p(uui) = 0.2$), $p(ui) = 0.6$. The two least likely symbols are *i* and *uui*, and therefore $tree = branch(ui, branch(i, uui))$. No value v exists to guide the first branch which we prove by exhaustion. Let $\alpha' = x,y,z$. $f(x,y,z)$ is always true. $f()$ is always false if $ssf(\alpha') \in i, ui, uui$. $f(x)$ is false if $ssf(\alpha') \in ui, uui$. $f(y)$ is true if $ssf(\alpha') \in i, ii$.

2.2.5 f-Guided Leaf Ordering

2.2.6 Optimal ordered decision trees

Much like huffman code, there must be an optimal ordered decision tree. The optimal ordered tree is the tree that has the lowest possible cost as defined by

$$cost(t) = \sum_{leavesoft} p(x) depth(x)$$

A naive algorithm to brute force the lowest cost tree is

This algorithm is not scalable, as it takes $O(n!)$ steps to complete. Howver, for small trees it is suitable for use and guarantees minimal tests.

Algorithm 1 Naively build optimal ordered decision tree

if $|items| = 1$ **then**
  return $items[0]$
else if $|items| = 2$ **then**
  return $Branch(items[0], items[1])$
end if
$result \leftarrow null$
$left \leftarrow items[0]$
$right \leftarrow items[1...]$
while $|right| > 0$ **do**
  $candidate \leftarrow Branch(optimal(left), optimal(right)))$
  if $result = null$ or $cost(candidate) < cost(result)$ **then**
    $result \leftarrow candidate$
  end if
  $left \leftarrow left + right[0]$
  $right \leftarrow right[1...]$
end while
return $result$

2.2.7 Ordered huffman coding

The following is a variant of huffman coding which completes in $O(n^2)$. However, does not produce optimal trees.

Like standard huffman code, this algorithm finds the smallest pair and creates a branch. However, unlike standard huffman code, this algorithm ensures that the order is maintained by finding the smallest *adjacent* pair. Similarly, while standard huffman code sorts the new set of items & branches on each iteration, this variant will put the branch into the tree in place.

Algorithm 2 Ordered huffman code

loop
  if $|items| = 1$ **then**
    return $items[0]$
  else if $|items| = 2$ **then**
    return $Branch(items[0], items[1])$
  end if
  $pSmallest \leftarrow null$
  $iSmallest \leftarrow null$
  for $i \leftarrow 0$ to $|items| - 1$ **do**
    $pCandidate \leftarrow p(items[i]) + p(items[i+1])$
    if $pSmallest = null$ or $pCandidate < pSmallest$ **then**
      $iSmallest \leftarrow i$
      $pSmallest \leftarrow pCandidate$
    end if
  end for
  $b \leftarrow Branch(items[iSmallest], items[iSmallest + 1])$
  $items \leftarrow items[0...iSmallest] + b + items[iSmallest + 2...]$
end loop

In profiling, ordered huffman coding tended to produce close to optimal trees.

2.2.8 Information Gain Coding

A common machine learning approach to generating decision trees is to split the data set recursively on maximum information gain [4]. Information gain $IG(Y,X)$ is the reduction of entropy $H(Y)$ given new information.

$$IG(Y,X) = H(Y) - H(Y,X)$$

$$H(Y,X) = \sum_{x \in X, y \in Y} -p(x,y) log_2 \frac{p(x,y)}{p(x)}$$

Since this is an ordered tree, a decision X has outcomes $\{y < n, y >= n\}$. In this case, join probabilities easily simplify. $p(y < n, y)$ corresponds to $p(y)$ if $y < n$, $p(y >= n, y)$ is $p(y)$ if $i >= n$, and $p(x', y)$ for all other x' is 0.

Therefore it is possible to reduce $H(Y,X)$ in this case, given test decision n, to

$$\begin{aligned} H'(Y,n) = &\sum_{y \in Y \leq n} -p(y) log_2 \frac{p(y)}{\sum_{y' \in Y \leq n} p(y')} \\ &+ \sum_{y \in Y \geq n} -p(y) log_2 \frac{p(y)}{\sum_{y' \in Y \geq n} p(y')} \end{aligned} \tag{3}$$

and can easily accomodate upper and lower bounds beyond this. Therefore it is possible to generate an information gain decision tree with the following algorithm.

Information gain seems to yield slightly better results than ordered huffman coding, however, it is slower. Therefore huffman coding is primarily used in Entropy Debugging.

2.2.9 Other forms of coding

Other lossless compression schemes exist which can outperform huffman code, for example, arithmetic coding and ANS. However, a general statement could be made that these algorithms outperform huffman code by dealing with information in an even more abstract manner than huffman codes. However, a *runnable identity encoding* must deal with that information concretely. Therefore these were not applied to Entropy Debugging, but there may be room to improve Entropy Debugging by attempting to do so.

3 Entropy debugging

Entropy debugging runs in three stages: presampling, adaptive consumption, and then minimization.

Algorithm 3 Information Gain Ordered Decision Tree

$pSubset(items') = \sum_{item \in items'} p(item)$
$hSubset(items') = \sum_{item \in items'} -p(item) log_2 \frac{p(item)}{pSubset(items')}$
if $|items| = 1$ **then**
  return $items[0]$
else if $|items| = 2$ **then**
  return $Branch(items[0], items[1])$
end if
$hAll \leftarrow hSubset(items)$
$maxIg \leftarrow null$
$maxIgIndex \leftarrow null$
for $i \leftarrow 0$ to $|items| - 1$ **do**
  $hLeft \leftarrow hSubset(items[...i])$
  $hRight \leftarrow hSubset(items[i+1...])$
  $ig \leftarrow hAll - hLeft - hRight$
  if $maxIg = null$ or $ig > maxIg$ **then**
    $maxIg \leftarrow ig$
    $maxIgIndex \leftarrow i$
  end if
end for
$left \leftarrow igTree(items[...maxIgIndex])$
$right \leftarrow igTree(items[maxIgIndex+1...])$
return $Branch(left, right)$

Algorithm 4 Presample

$choices \leftarrow$ choose n random entries from $items$.
$successes_u \leftarrow 0$
$successes_{uu} \leftarrow 0$
$trials_u \leftarrow n$
$trials_{uu} \leftarrow 0$
for $choice \leftarrow choices$ **do**
  if $t(items - choice)$ **then**
    $successes_u++$
    remove $item$ from $items$
    $next \leftarrow$ the entry in $items$ following $item$
    $trials_u++$
    $trials_{uu}++$
    if $t(items - next)$ **then**
      $successes_u++$
      $successes_{uu}++$
      remove $next$ from $items$
    end if
  end if
end for
$p_u \leftarrow (succeses_u + 1)/(trials_u + 2)$
$p_{uu} \leftarrow (succeses_{uu} + 1)/(trials_{uu} + 2)$

3.1 Presampling

First, entropy debugging will presample the input to generate a basic markov model.

It is unclear what the perfect number of presamples is. If the presampling count is high, this prevents later stages from making large gains in few tests. However, the presampling stage is the only stage that has true statistical randomness. The current default hyperparameter value is to do five presamples.

Five random pairs of entries in the input are selected, and individually checked for removability against the input function. Those that may be removed while still satisfying the input function have their subsequent neighbor checked as well.

Bayesian inference is used to infer the underlying $p(u_n)$ and $p(u_n|u_{n-1})$. The base probability distribution is found via the Laplace Rule of Succession, so if n evenst are observed out of s trials, the inferred underlying probability of the next event is $\frac{n+1}{s+2}$.

With these data on probabilities, we infer $p(u)$ and $p(uu)$. We may then infer $p(i)$, $p(ii)$, $p(iu)$, and $p(ui)$.

$$p(i) = 1 - p(u) \tag{4}$$
$$p(ui) = 1 - p(uu) \tag{5}$$
$$p(iu) = 1 - p(ii) \tag{6}$$

This trivially covers all but $p(ii)$, which may also be inferred with slightly more work:

$$\begin{aligned} p(ii)p(i) + p(u)p(ui) &= p(i) \\ p(ii)p(i) &= p(i) - p(u)p(ui) \\ p(ii) &= \frac{p(i) - p(u)p(ui)}{p(i)} \end{aligned} \tag{7}$$

It is now possible to estimate the probabilty of any sequence, with or without a preceeding state.

3.2 Adaptive Consumption

The adaptive consumption stage of Entropy Debugging sweeps the input from left to right, forming decision trees which are walked to find an outcome. That outcome is read to reduce the input, and update the markov model.

The symbol set and tree size is capped, though the following algorithm writeup elides that detail. When the tree is capped, a symbol $uuuu...n$ may be built, which requires slight offset & probability adjustments to be correct, but otherwise does not significantly deviate from what's written below. By capping symbol and tree size, extremely large and sparse inputs may be less optimally solved, however, it reduces pathological cases that may lead to $O(n^3)$ complexity. In Entropy Debugging, tree size caps are hyperparameters. The default values are to cap the tree to 1000 symbols, where each symbol has at least a 30% probability.

Trees may be built with any algorithm. This behavior may be customized as a hyperparameter. The default behavior is to build optimal trees for symbol sets of length 10 or less, huffman-like trees for symbol sets of length 51 or greater, and between that, a combinator of information gain, optimal, and huffman like is used: trees of size six or less are

built optimally, and for other trees, both huffman and information gain trees are built, of which the lowest cost tree is used (and this process recurses).

Algorithm 5 Adaptive consume

```
state ← "unknown"
for i ← 0to|items| do
  symbols ← {uuu..._n i|n ∈ {0,1,...|items|}}
  p_first ← {p(i|state)}
  p_rest ← {p(u|state) * p(uu)^(n − 1) * p(ui)|n ∈ {1,2,...|items|}}
  symbols_p ← p_first + p_rest
  tree ← buildTree(symbols, symbols_p)
  while tree is a branch do
    pivot ← rightmost leaf of the left branch of tree)
    candidate ← items[...i, i + |pivot|...]
    if test(candidate) then
      tree ← right branch of tree
    else
      tree ← left branch of tree
    end if
  end while
  outcome ← tree
  items ← items[...i, i + |outcome| − 1...]
  successes_u ← successes_u + |outcome| − 1
  trials_u ← trials_u + |outcome|
  successes_uu ← successes_uu + |outcome| − 2
  trials_uu ← trials_u + |outcome| − 3
  state ← final state in tree
end for
```

Note that these updates to the markov model contain sampling bias. In benchmarks, performance is still good. A reasonable explanation is that some sampling bias is a good thing, as it allows the algorithm to adapt somewhat to local variations in the distribution.

The algorithm can be run without adjusting the markov model. This is reasonable if a statistical model for a corpus has already been built.

3.3 Minimization

The final stage of entropy debugging is the minimization phase. It is optional in our algorithm. At this point, the problem of *set simplification* is solved, however, the minimization phase provides *set minimization*.

In tests, this phase can behave somewhat pathologically, which is to be expected since it is $O(n^2)$ complexity instead of linear time.

The process is a very simple character by character sweep until a pass completes making no progress.

The reason why this phase is so naive is simple.

Consider the alternative case where entropy debugging is performed iteratively until entropy debugging makes no progress. For a low entropy input, entropy debugging will outperform a brute force search. However, for a high or medium entropy input, there is no way to beat a brute force search, and therefore entropy debugging will revert to that approach automatically.

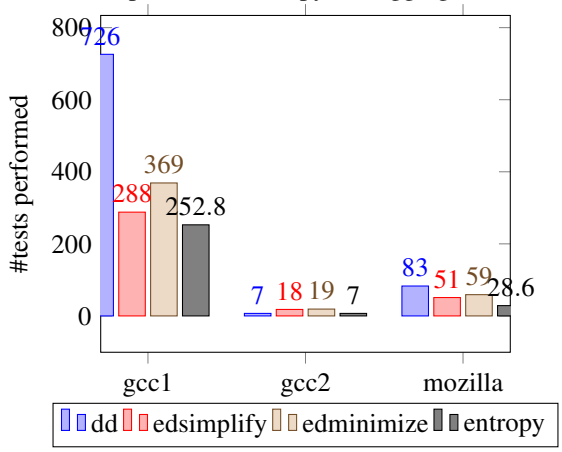
A minimized input is almost by definition a high entropy input. Aside from pathological cases (not seen in any experiment done for this paper), it holds that it is simpler and faster to merely choose a brute force approach immediately.

See discussions on the value of minimization over simplification.

4 Results

4.1 Delta Debugging Case Studies

Zeller [1] included some case studies in the original publication of Delta Debugging which have been reproduced. Two are related to a gcc bug, and one is a crash in firefox when printing a $< SELECT >$ tag. These offer a good starting point of comparison to Entropy Debugging.



As shown above, entropy debugging is able to minimize the first gcc crash input in 52% faster than delta debugging – it is also able to simplify it in 37% of the time. Similarly, entropy debugging is 29% faster at minimizing the mozilla crash input, and can simplify it in 61% of the time.

For the second gcc crash, entropy debugging takes nearly twice as long. Why? The difference comes down to assumptions made by the algorithms. The input contains 30 items, and the minimal subset is a single item. Delta Debugging assumes this sort of input is likely, and the search it performs works out to be a perfect binary search. On the other hand, Entropy Debugging assumes nothing about the underlying information. It takes 10 checks just to presample the input (with default hyperparameters), already losing to Delta Debugging. It then executes 8 checks to simplify or 9 checks to minimize the input, which is slightly behind Delta Debugging. In essence, the difference is the assumptions made by the algorithms, combined with the small size of the input.

It is possible, in circumstances like this, to run entropy debugging without any presampling, and to seed a custom starting probability distribution. In this case, the second gcc crash input can be simplified by entropy debugging in six checks. In general, this is not necesary, as Entropy Debugging beats Delta Debugging for a very large range of markov model probabilities.

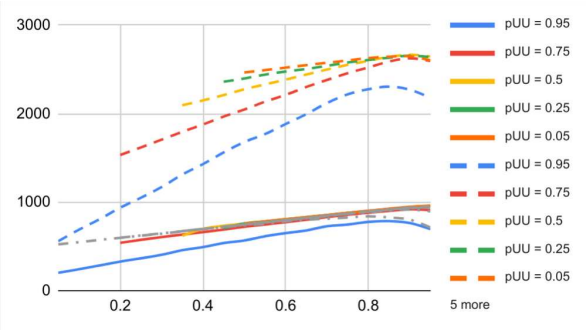


Fig. 1. Delta debugging (dashed) vs brute force (grey), Entropy Debugging (solid) for medium probabilities

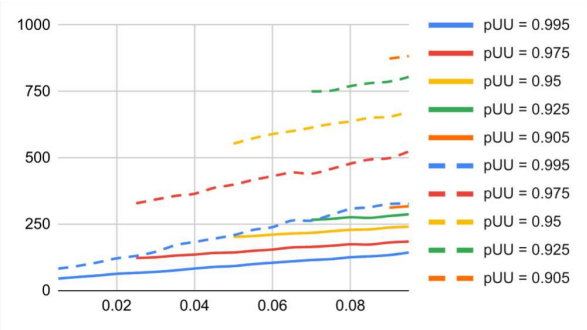


Fig. 2. Delta debugging (dashed) vs Entropy Debugging (solid) for small probabilities

4.2 Generated Data

To more directly measure the point at which Delta Debugging may perform better, than Entropy Debugging, generated data from various markov models are fed into both algorithms and compared below. In addition, a naive brute force is plotted. Delta Debugging often exceeds the naive approach by nearly 3x, while Entropy Debugging fits itself to the curve.

In the above chart, Entropy Debugging is compared to Delta Debugging and a naive brute force algorithm for "normal" markov probabilities. The x axis represents the overall percentage of waste in an input, and the style of the line marks the algorithm in question (solid = entropy debugging, dashed = Delta Debugging, grey dash dot = brute force). Each algorithm is plotted for multiple transition properties (correlation between waste/non-waste in the input): 95% correlation, 75% correlation, no correlation (50%), 75% discorrelation, 95% discorrelation.

For this range of probabilities, Delta Debugging only manages to approach the performance of the naive solution in the most extreme case of 95% waste with 95% correlation. Indeed, Entropy Debugging is only considerably better than the naive approach in the 95% correlation plot (and never considerably worse).

This chart clearly does not capture a set of inputs that Delta Debugging was designed to handle. As probabilities get further and further from these "normal" (by some definition) probabilities, Delta Debugging gets closer and closer to Entropy Debugging, with the two meeting when around 99.8% of an input is waste with around 99.9% correlation.

4.3 Fuzz Testing Coverage Corpus

Entropy Debugging was used to minify a fuzz testing corpus while maintaining the same code coverage.

The fuzzing library Dust is a code coverage guided fuzz testing framework for Dart. This has been used to generate random Dart programs that may crash the Dart analyzer. To do so, a corpus of interesting programs is maintained such that those interesting programs each exercise some unique code path of the fuzz target.

For the Dart language, there are hundreds of test files that cover most of the language specification. This is therefore an ideal starting place to find this set of interesting Dart

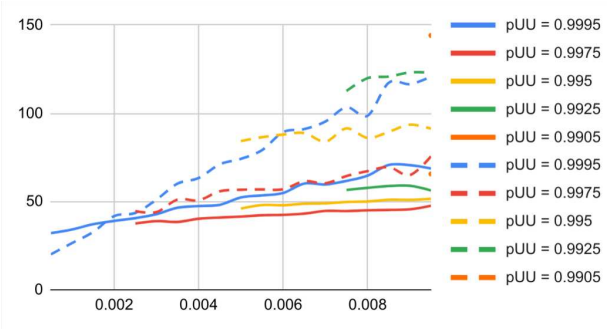
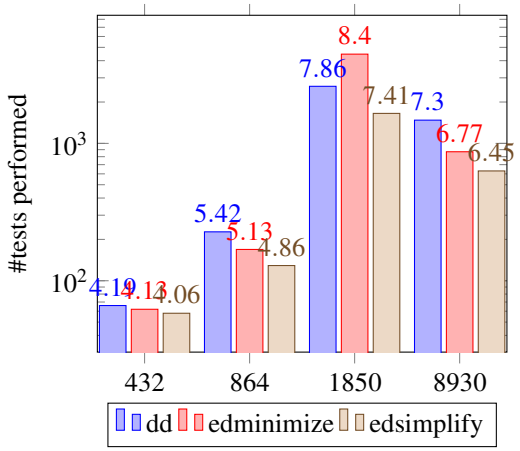


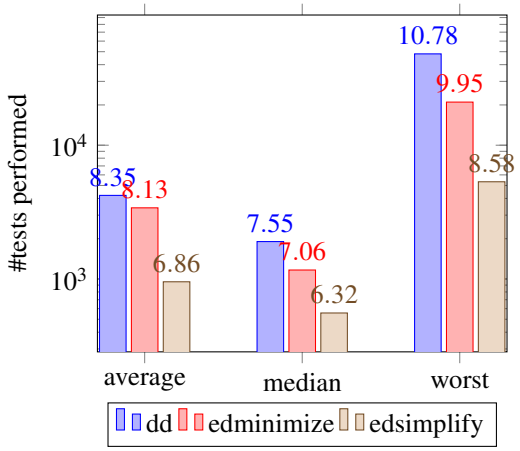
Fig. 3. Delta debugging (dashed) vs Entropy Debugging (solid) for very small probabilities

programs. However, the corpus contains a lot of redundancy from the perspective of any given fuzz target. The goal is to therefore quickly minimize this corpus, a task that in theory should be a good fit for either Delta Debugging or Entropy Debugging.

Here are four example simplifications, charted by input size to runs required by the two algorithms. In the case of entropy debugging, it has been split out by time required to simplify vs minimize the input.



And here are the median, average, worst results of the three algorithms over all samples.



These data not only show not that Delta Debugging takes 80% of the time in this example in the average case, but it also took 43% of the time in the worst case.

4.4 Is miminization useful?

The previous numbers also raise the question of whether or not *minimization* is better than *simplification*.

On average, minimization yielded an 11% smaller input than simplification, even though it took 3.5x as many tests. For context, the 11% difference in the result is a mere 3% of the input – and requires 33 additional tests per additional removed character.

Of course, minimization is still posible with Entropy Debugging, and still faster in this data set than Delta Debugging (especially in the median and worst case). However, this paper finds no way to speed up minimization. Under the statistical and compression theory approach described above, there simply is no better approach than a character by character search when the entropy of a sample is high. And at 89% of the way simplified, that counts as high entropy. And if a pass at this entropy ends up finding any items in the set that can be removed, a new pass is required, creating pathological cases in both entropy debugging as well as delta debugging.

Why is the average only 20% less for Entropy Debugging even though its median performance is a 40% improvemnet from Delta Debugging, which also has the highest worst case result? This is not a quantitative explanation, but it seems the answer here is that an inefficient simplification stage leads to a better minimization stage.

Recall that minimization in this paper specifically refers to Zeller's definition of 1-minimal [1]. Stricter minimums can be sought such as 2-minimal and beyond. Anecdotally, it seems that acheiving 1-minimal may frequently require removing *pairs* or larger chunks of the input, more in the realm of 2-minimal and beyond. In a character by character sweep, it may take n passes for some n to acheive 1-minimality.

On the other hand, it seems that Delta Debugging is more likely to have run additional suboptimal tests in the form of attempting to delete optimistically large chunks of contiguous characters. These ultimately still result in a larger average case performance, but they do decrease the likelihood of the minimization stage taking a large number of passes. (Note that this effect also does not keep Delta Debugging from claiming the largest recorded worst case result).

It seems compelling that simplification is a good approximation of 1-minimal that can be found in significantly less time. Future research to improve minimization techniques seems prodent, and it seems likely that Delta Debugging could also benefit from a simplification mode in addition to 1-minimal search.

Similarly, the case studies on mozilla and gcc could be even more drastically improved if only a *simplified* reproduction were sought instead of a *minmiized* one.

4.5 Performance takeaways

The benchmarking performed lends itself to a selection process between Entropy Debugging and Delta Debugging.

If a true 1-minimal result is needed, it is possible that entropy-debugging will not lead to significantly improved results without some improvement over the process of minimization. However, it still appears to be the better algorithm in the average and worst case.

If the input is likely close to simplified, Entropy Debugging will yield very large improvements in performance over Delta Debugging.

If the input is small (approaching 30 entries) and also far from the minimal result, Delta Debugging may yield better results due to Entropy Debugging's limitation of presampling (though this can be tuned).

If the input is very very far from the minimal result (less than 1% of the input, is expected to be part of the minimal result), then Delta Debugging may yield better results due to Delta Debugging's more aggressive approach. This may also be tunable as a hyperparameter of Entropy Debugging.

This paper proposes that in all other cases, Entropy Debugging seems to outperform Delta Debugging in all other cases.

This paper also proposes that if a true *minimal* result is *not* needed, that a simplification from the Entropy Debugging is a much more efficient approximation.

4.6 Optimality analysis

To do: analyze how close entropy debugging gets to the lower bound of algorithmic performance.

4.7 Future Work

There is a large body of future work to be explored in addition to this paper's findings.

Entropy Debugging currently uses markov models as the only statistical understanding of the simplification stream's information. This limits the peformance of Entropy Debugging on inputs that could be described better by better statistical models, such as machine learning models. While more advanced models generally require more data, and data collection is itself throttled in Entropy Debugging, it is possible that new statistical models could lead to performance improvements, especially in simplifying large corpora of inputs.

Approaches to unify Entropy Debugging with hierarchical approaches such as HDD [3] could yield even larger efficiency. One possibility of such a unification is to integrate the hierarchical nature of data into the statistical models of Entropy Debugging rather than changing the overall structure of the algorithm itself.

Delta Debugging is also capable of comparing minimum failing inputs with maximum succeeding inputs. While Yu et. al. found that these isolated changes are typically failure-irrelevant in practical usage [2], work could be done to find these inputs as part of Entropy Debugging as well for applications where it may be useful.

An algorithm remains to be found that can efficiently build an optimal runnable identity encoding, or in general, an optimal ordered probabilistic tree. Doing so may have implications for Decision Tree Learning in general, and would also improve the performance of Entropy Debugging.

More advanced compression techniques than huffman code exists. Indeed, it is the simplicity of huffman codes that make them easily applicable to this particular simplification problem. However, it is possible that these more advanced compression techniques such as Arithmetic Coding and ANS could be adapted to produce runnable identity codes that could be used by Entropy Debugging.

5 Conclusions

Conclusions TBD

6 Discussions

Discussion TBD

Acknowledgements

Acknowledgement TBD

References

[1] Zeller, A., and Hildebrandt, R., 2002. "Simplifying and isolating failure-inducing input". *IEEE Trans. Softw. Eng.,* **28**(2), Feb., p. 183–200.

[2] Kai Yu, Mengxiang Lin, J. C., 2010. "Towards automated debugging in software evolution: Evaluating delta debugging on real regression bugs from the developers' perspectives".

[3] Misherghi, G., and Su, Z., 2006. "Hdd: Hierarchical delta debugging". In Proceedings of the 28th International Conference on Software Engineering, ICSE '06, Association for Computing Machinery, p. 142–151.

[4] Quinlan, J. R., 1986. "Induction of decision trees".