Information Theory Approach to Reducing Failure Inducing Input

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

Minimized reproductions of software issues are easier to debug. This paper presents a new algorithm "entropy debugging" faster than the predecessor "delta debugging". To do so, it treats the problem of crash minimization as a set minimization/simplification problem. The fitness tests themselves on possible subsets to consider can then be viewed as an information stream, which lends itself well to statistical analysis and data compression techniques. This means "entropy debugging" is likely to approach a hypothetical lower bound of performance given an accurate statistical model of the underlying information.

1 Introduction

Debugging a crash is easier with an example crashing input, and easiest when that example reproduction is minimal. These inputs can be found by users, developers, and/or tools like fuzz testing.

When these crash reproductions are not minimized, it would be ideal if there were an algorithm that could create the minimized input. Removing one character (or symbol of some other kind) at a time from the input is a naive algorithm to reduce an input. If the reproduction is small and the crash reporduces quickly, this naive algorithm works.

Delta Debugging [1] is another option. Delta Debugging resembles a binary search where large chunks of the input are removed at a time, potentially reducing the input greatly in few tests. However, Delta Debugging is slower than the naive approach when the input is already close to minimal. This fits 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 an option for cases where the input can be formed into a tree. Hierarchical Delta Debugging yields great results but not all simplifications can be treated as a tree, and the time spent turning the failure case into a tree-like structure may not be worth the improved performance. For instance, a greybox/blackbox fuzz testing framework does not understand the program it's fuzzing well enough to create a hierarchical structure of the failures it finds. And malformed inputs may only crash because the input violates the tree structure that the program expects (for instance, a parser may crash for a particular type of mismatched parenthesis). Lastly, Yu et. al found that both Hierarchical Delta Debugging did not yield definite improvements over Delta Debugging in efficiency in real world application [2].

In spite of these issues, simplification of failure cases and simplification of fuzz corpora by code coverage measures are both still tasks performed by frameworks and those frameworks need algorithms with good performance.

Entropy Debugging is a new algorithm presented here which aims to achieve close to optimal average case performance for all types of inputs, by taking a statistical/information theory approach to the problem of set simplification over an arbitrary fitness test.

1.1 Formalizations

Delta Debugging [1] and the algorithm presented here are both algorithms that solve the following general problem.

1.1.1 Set Minimization

We define the problem of **set minimization** as $sm(\alpha,t) = \alpha'$, where α is an ordered or unordered set, t is a blackbox function, and α' is the smallest set s such that $s \subset \alpha$ and $t(s)$. (Zeller's "local minimum" [1]).

Given that t is an abritrary black box function, its performance is unbounded. Therefore an important measure of complexity of an algorithm performing set minimization is the number of invocations of t.

Set minimization is the true goal of both Delta Debugging and the algorithm presented in this paper, however, the complexity of $sm(\alpha,t)$ is $2^{|\alpha|}$ [1].

Therefore a different definition is needed for practical purposes; for this there is **n-minimality** [1].

1.1.2 1-minimality

Zeller defines **n-minimality** as $min(\alpha,t,n) = \alpha'$ where $\forall c \subset \alpha \cdot |c| - |\alpha| \leq n \Rightarrow t(c)$. Zeller proposes that 1-minimality is the ideal target for an algorithm approximating set minimization.

In less formal terms, n-minimality can be defined as searching for a result where no n items in the set could be removed while still satisfying the test function. In the case of 1-minimal, this means that no individual item in the set can be removed on its own while satisfying t, though it is possible that some pair of entries in the set (or tuples of size $n > 1$) could be.

This paper disputes the value of 1-minimality, though the algorithm presented can help efficiently find 1-minimal results.

To do: link to disputing the value of 1-minimality

Finding a 1-minimal set is $O(n^2)$.

1.1.3 Set Simplification

There is also a new presented definition, which is core to the ideas presented in this paper. We define the problem of **set simplification** as $simplify(\alpha, t) = \alpha'$ as a subproblem of set minimization. In simplification, it is required that t satisfies $\forall c \supset \alpha' \Rightarrow t(c)$.

Theorem 1.1. *The worst case complexity of set simplification is $O(n)$.*

Axiom 1.2. $t(\alpha' \setminus c)$ *iff* $c \in \alpha'$.

Axiom 1.3. $|\alpha'| \leq |\alpha|$.

Lemma 1.4. $simplify(\alpha, t) = \{c | c \in \alpha, t(\alpha \setminus c)\}$

Proof. Since set simplification can be translated to a list comprehension on α, it cannot take more than $|\alpha|$ steps to find. □

Theorem 1.5. *The result of set simplification is also n-minimal for all n.*

Proof. Assume that α' is not n-minimal for some $n > 0$. That means $\exists c \subset \alpha' \cdot |c| - |\alpha'| > n \wedge t(c)$. That implies $|c| < |\alpha'|$. However, if $t(c)$ and $|c| < |\alpha'|$, then $|\alpha'|$ is not the smallest s such that $t(s)$. This is a contradiction. Therefore if the set simplification result α' is n-minimal for all n. □

In practice, few nontrivial tests satisfy the requirements to make set simplification applicable. However, this paper uses set simplification as a benchmark of algorithmic performance and suggests it as a good approximation for many use cases where set minimization is desired.

1.2 Information Theory Approach

The novel approach taken to find a better algorithm of approximating set minimization is to translate test function results into an information stream. This paper explores how that information stream can be compressed efficiently.

Expressing the problem as a data stream is easier from the lens of *set simplification* than *set minimization*. (Later, *set minimization* will be compared to noisy data streams).

To do: eventually compare set minimization to noisy data streams.

From here on, this paper assumes that α is a *totally ordered set*.

1.2.1 Simplification as Information

For each member $e_i \in \alpha'$, the "underlying state" can be viewed as important ($e_i \in \alpha'$) or unimportant ($e_i \notin \alpha'$). As a character coding, this may look like *uuiuuiuui*. Of course, a better encoding is a binary encoding, 001001001.

This binary encoding is has some significance. It represents the *brute force encoding* of performing set minimization on the input. In other words, if each character is tested individually and in order for membership of α' (by $t(\alpha \setminus e_i)$), the ordered results of these invocations of t will be true or false corresponding to the 0s and 1s above.

The field of information theory suggests that this stream should be encodable in a more optimal encoding.

In the example above, one such encoding would be $0 = 00, 1 = 1$. Under this encoding, the previous example can be compressed to 010101, saving four bits of information.

1.2.2 Statistical Modeling of the Simplification Data

Having a statistical model of the simplification data is required in order to create an optimal encoding of that data.

The simplest statistical model of this data is merely having $p(e \in \alpha')$ for any $e \in \alpha$.

This model would not assume any correlation between the individual members of the minimal set. While this is reasonable given than t is an arbitrary function, it does not match what is typical of real world scenarios.

The cases given by Zeller [1] exemplify this. The simplest failure inducing inputs studied for *delta debugging* typically represent contiguous subsets of the original input.

Therefore this paper suggests markov transitions as a simple but more powerful model. A set of markov transition probabilities (from u to i and i to u) are powerful enough to capture correlation, discorrelation, and independence between bits in the data stream.

Now it is possible to create an optimal coding against this markov model.

1.2.3 Optimal Markov Encoding

Given a markov model and an input length, it is now possible to construct an optimal encoding for the sequence using huffman coding. Where β is the set of all possible sequences of items u and i,

$optimal = Huffman(\beta)$.

If we had this code, we would be able to optimally represent inputs like *uuiuuiuui*, for any arbitrary length, given any arbitrary underlying markov model.

Note: this is optimal *if the markov model is the best statistical model*. There may be other more accurate statistical models that lead to more optimal encodings still.

However, this approach is not scalable, as $|\beta| = 2^{|n|}$.

1.2.4 Simplification Chunks

To simplify the construction of our huffman code, we must construct a smaller set of symbols that makes progress on the whole input. One possibility is to construct huffman codes over each "character" (i and u) in the input. This would merely result in the unimproved encodings ($i = 0$, $u = 1$ or vice versa), regardless of the entropy of the markov model. This can be thought of as "quantization error," where an input entropy below 1 will still yield an average performance of at least 1 in a huffman tree.

There is a natural way to segment the simplification data that matches practical observations of real world use cases for these algorithms, as well as having this "quantization error" end up matching the worst case scenario of simplification.

The best case performance of simplification is $O(|\alpha'|)$. It therefore is reasonable to choose the huffman code symbol sets such that any quantization error that results from the set having entropy less than 1, will not necessarily decrease algorithmic worst case performance.

It is therefore this paper's suggestion that huffamn codes be used to encode *sequences of repeated u ending in a single i or end of input*, in the style of Golomb-Rice run-length encoding [4].

Here's such an encoding: $i = 0$, $ui = 10$, $uui = 11$, $uu\varepsilon = 11$.

This example is of course, the identity encoding.

As an example of an improved encoding, our original example may be optimally encoded (with ..., $uui = 0$, ...), to 000.

1.2.5 Runnable Identity Encodings

Unfortunately, the problem of compressing our original simplification data is not yet done. The encodings created by huffman's algorithm are not guaranteed to create a *runnable identity encoding*.

Take the above example, where $uui = 0$. This is not runnable in its indentic form. There is no single value v such that $t(v)$ iff $uui \in \alpha'$. Through two tests this is possible (test that $t(\{i\}) \wedge \neg t(\{\})$), however, those two tests imply the *runnable encoding* for this sequence is 10 and not 0. Given that they do not share an identity, $uui = 0$ is not a *runnable identity encoding*.

Theorem 1.6. *Huffman coding is not guaranteed to return a runnable identity encoding.*

Lemma 1.7. *If $p(i) = 0.25$, $p(ui) = 0.5$, and $p(uui) = 0.25$, $huffman(\{i, ui, uui\})$ produces $ui = 0$, $i = 10$, $uui = 11$.*

Axiom 1.8. *If this is a runnable identity encoding, then some value v exists such that $t(v)$ iff $ui \wedge \neg(i \vee uui)$.*

Axiom 1.9. *$t(v) = r$, and r is either $true$ or $false$.*

Axiom 1.10. *$t(v)$ is false in all cases where $v \not\supset \alpha'$ v and true in all cases where $v \supset \alpha'$.*

Axiom 1.11. *$v \supset \alpha'$ or $v \not\supset \alpha'$.*

Axiom 1.12. *α must be at least three items $\{..., a, b, c, ...\}$*

Axiom 1.13. *$t(v)$ must be r in the case that $b \in \alpha'$ and $a \notin \alpha'$.*

Axiom 1.14. *$t(v)$ must be $\neg r$ in the case then $a \in \alpha'$.*

Axiom 1.15. *r may be true or false: if r is always true or always false then $t(v)$ distinguishes nothing.*

Lemma 1.16. *If $a \in v$ and $r = true$, $v \supset \alpha'$, so it is possible that $a \in \alpha'$ or $a \notin \alpha'$. Since $r = true$ distinguishes between $b \in \alpha'$ or $a \notin \alpha'$, it must be that $a \notin v$.*

Lemma 1.17. *Since $a \notin v$, $r = true$ implies that $a \notin \alpha'$. Therefore whenever $r = true$ it must also be true that $b \in \alpha'$ in all cases.*

Lemma 1.18. *When $b \in \alpha'$ and $r = true$, then $v \supset \alpha'$, and therefore also $b \in v$.*

Proof. When $b \in v$ and $v \supset \alpha'$ and $b \notin \alpha'$, $r = true$, which is a contradiction. □

Constructing a runnable identity encoding requires constructing a decision tree in which every branch in the tree (or bit in the encoding) corresponds to a single invocation of t for some derivable input v.

For the sequence encoding described here, this is equivalent to an *ordered tree* (that is to say, a tree whose leaves maintain a known ordering).

There is a very simple ordering that ensures a *runnable identity encoding*: given two symbols a and b, we say $a < b$ iff $|\{u | e \in a, e = u\}| < |\{u | e \in b, e = u\}|$ (that is, the symbols are ordered in terms of how many unimportant characters that symbol finds).

Theorem 1.19. *An ordered tree over this relation is a runnable identity encoding.*

Lemma 1.20. *Let our symbol set be i, ui, uui, $u..._n i$.*

Lemma 1.21. *There is some root of the tree $r = Branch(r', r'')$. Since the tree is ordered, all symbols in r' are i, ui, uui, ... $uuu..._j i$ for some j, and all symbols in r'' are $uuu..._{j+1} i$, $uuu..._{j+2} i$, $uuu..._n i$.*

Lemma 1.22. *$uuu..._e i$ "describes" α' when $\alpha[e + 1 + c] \in \alpha'$ (where c is the offset of the encoder in the data stream), and $\forall w \leq e > 0 \rightarrow \alpha[w + c] \notin \alpha'$.*

Lemma 1.23. *There is some value v such that $t(v)$ is true for all possible α' such that $uuu..._{j+q} i$ "describes" α' where $q > 0$. Additionally, for that value v, $t(v)$ is false for all $u..._{j+q} i$ where $q \leq 0$. (Note that this implies the tree searches rightwards on $true$).*

Lemma 1.24. *Assume v is the smallest superset of all sets that are "described by" $uuu..._{j+q} i$" for all $q > 0$. Since $\alpha[j + q + 1 + c] \in \alpha'$, 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 1.25. *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, ... $uuu..._j i$, $uuu..._{j+1} i$, ... $uuu..._n i$ 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*. □

1.2.6 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

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

A naive algorithm to brute force the lowest cost tree is

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 $result$
```

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.

1.2.7 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.

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

1.2.8 Information Gain Coding

A common machine learning approach to generating decision trees is to split the data set recursively on maximum information gain [5]. Information gain $IG(Y,X)$ is the reduction of entropy $H(Y)$ given new information.

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
```

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

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

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

$$\begin{aligned} H'(Y,n) = \sum_{y \in Y \leq n} -p(y)log_2 \frac{p(y)}{\sum_{y' \in Y \leq n} p(y')} \\ + \sum_{y \in Y \geq n} -p(y)log_2 \frac{p(y)}{\sum_{y' \in Y \geq n} p(y')} \end{aligned} \quad (1)$$

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.

1.2.9 Other forms of coding

Other lossless compression schemes exist which can outperform huffman code, for example, arithmetic coding

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 $Branch(left, right)$
```

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.

2 Entropy debugging

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

2.1 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 $\frac{n+1}{s+2}$.

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
$p_u \leftarrow (succeses_u + 1)/(trials_u + 2)$
$p_{uu} \leftarrow (succeses_{uu} + 1)/(trials_{uu} + 2)$
```

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

$$p(i) = 1 - p(u) \tag{2}$$
$$p(ui) = 1 - p(uu) \tag{3}$$
$$p(iu) = 1 - p(ii) \tag{4}$$

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

$$\begin{aligned} p(ii)p(i) + p(u)p(ui) &= p(i) \\ p(ii)p(i) &= p(i) - p(u)p(ui) \\ p(ii) &= \frac{p(i) - p(u)p(ui)}{p(i)} \end{aligned} \tag{5}$$

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

2.2 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

```
$state \leftarrow$ "unknown"
for $i \leftarrow 0 to |items|$ do
    $symbols \leftarrow \{uuu..._n i | n \in \{0, 1, ...|items|\}\}$
    $p_{first} \leftarrow \{p(i|state)\}$
    $p_{rest} \leftarrow \{p(u|state) * p(uu)^(n-1) * p(ui) | n \in \{1, 2, ...|items|\}\}$
    $symbols_p \leftarrow p_{first} + p_{rest}$
    $tree \leftarrow buildTree(symbols, symbols_p)$
    while $tree$ is a branch do
        $pivot \leftarrow$ rightmost leaf of the left branch of $tree$)
        $candidate \leftarrow items[...i, i + |pivot|...]$
        if $test(candidate)$ then
            $tree \leftarrow$ right branch of $tree$
        else
            $tree \leftarrow$ left branch of $tree$
        end if
    end while
    $outcome \leftarrow tree$
    $items \leftarrow items[...i, i + |outcome| - 1...]$
    $successes_u \leftarrow successes_u + |outcome| - 1$
    $trials_u \leftarrow trials_u + |outcome|$
    $successes_{uu} \leftarrow successes_{uu} + |outcome| - 2$
    $trials_{uu} \leftarrow trials_u + |outcome| - 3$
    $state \leftarrow$ 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.

2.3 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.

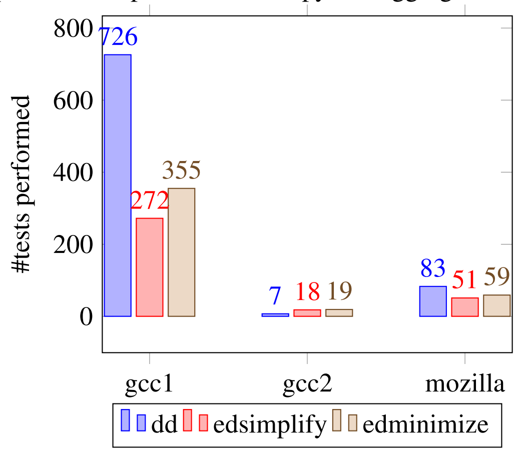
A minimized input is almost by definition a high entropy input. Aside from pathological cases (not seen in any experiment done for this paper), it holds that it is simpler and faster to merely choose a brute force approach immediately.

See discussions on the value of minimization over simplification.

3 Results

3.1 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

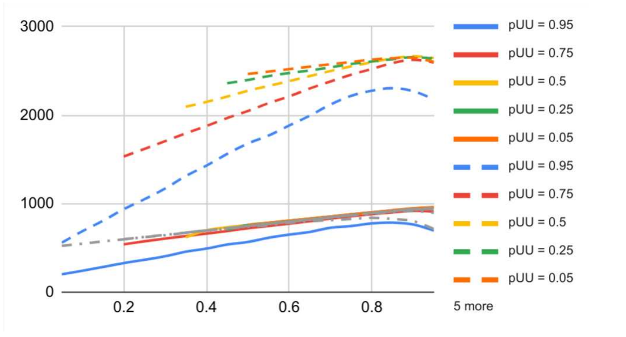


Fig. 1. Delta debugging (dashed) vs brute force (grey), Entropy Debugging (solid) for medium probabilities

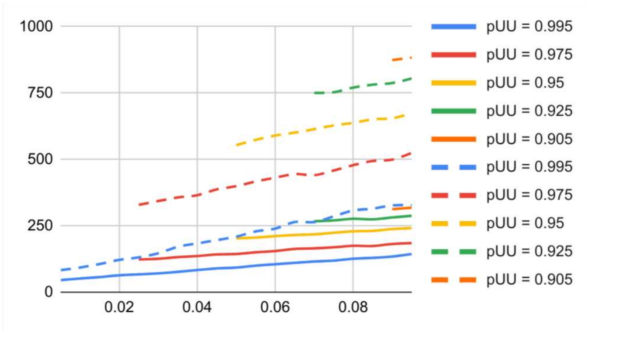


Fig. 2. Delta debugging (dashed) vs Entropy Debugging (solid) for small probabilities

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.

3.2 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.

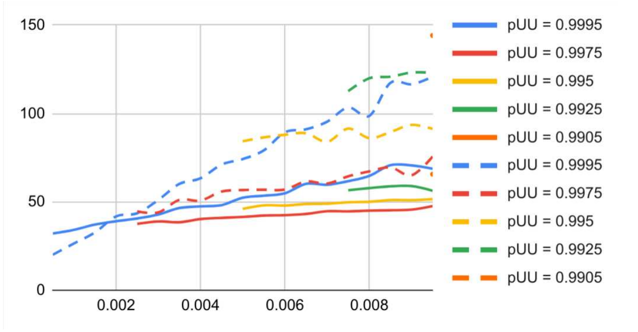


Fig. 3. Delta debugging (dashed) vs Entropy Debugging (solid) for very small probabilities

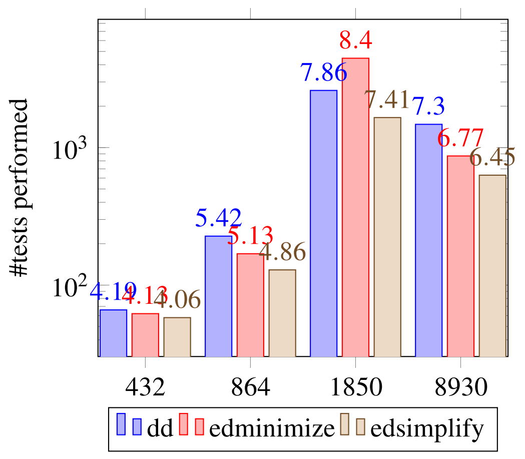
3.3 Fuzz Testing Coverage Corpus

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

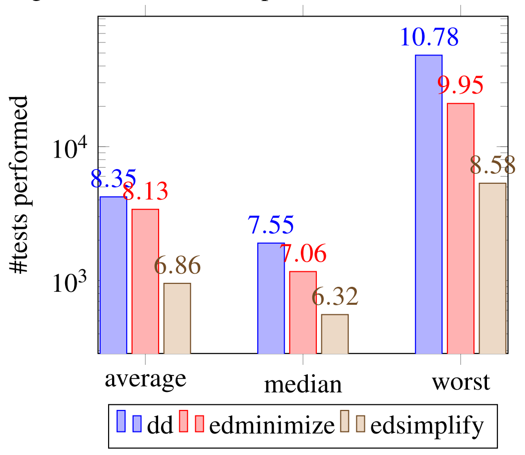
The fuzzing library Dust is a code coverage guided fuzz testing framework for Dart. This has been used to generate random Dart programs that may crash the Dart analyzer. To do so, a corpus of interesting programs is maintained such that those interesting programs each exercise some unique code path of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

3.4 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.

3.5 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.

3.6 Optimality analysis

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

3.7 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.

4 Conclusions

Conclusions TBD

5 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] Malvar, H., 2006. "Adaptive run-length/golomb-rice encoding of quantized generalized gaussian sources with unknown statistics". pp. 23 – 32.

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