Generalized Reduction of Tree-Structured Test Inputs

Anonymous Author(s)

ABSTRACT

Localizing a fault triggered by a given test input is a time-consuming task. The well-known delta debugging algorithm and its derivatives automate this task by repeatedly reducing the given input. Unfortunately, these approaches are limited to blindly removing parts of the input and cannot reduce the input by restructuring it. This paper presents the Generalized Tree Reduction algorithm, an effective and efficient technique to reduce arbitrary test inputs that can be represented as a tree, such as program code, HTML pages, and XML documents. The algorithm combines an extensible set of tree transformations with delta debugging and a greedy backtracking algorithm. To reduce the size of the considered search space, the approach automatically specializes the tree transformations applied by the algorithm based on examples of input trees. We evaluate the approach by reducing Python files that cause interpreter crashes and JavaScript files that cause browser inconsistencies. The algorithm reduces these files by 50% and 96%, respectively, outperforming both Delta Debugging and another state-of-the-art algorithm.

ACM Reference format:
Anonymous Author(s). 2017. Generalized Reduction of Tree-Structured Test Inputs. In *Proceedings of ISSTA conference, Santa Barbara, California, USA, July 2017 (ISSTA'17),* 11 pages.
DOI: 10.475/123_4

1 INTRODUCTION

Debugging is a time-consuming and tedious process. Pinpointing the cause of a bug often requires more time than fixing it [21]. Often, the input is responsible for exposing a bug and a corresponding bug report will contain a failure-inducing input as a test case. One example could be a program that crashes the compiler or interpreter when given as an input. A different example could be an HTML page that is rendered incorrectly in a browser. The larger this test case is, the more complex it becomes to distinguish irrelevant portions from crucial ones, making the debugging process cumbersome.

To ease the task of isolating the cause of a bug triggered by a given test input, several automated techniques have been proposed. Given a test input and an oracle that determines whether a reduced version of the input still triggers the same bug, these techniques automatically reduce the input. With a reduced test input, the developer is likely to find (and fix) the root cause of the bug faster. Furthermore, the reduced test input can easily be turned into a regression test case after the bug has been fixed.

Existing techniques roughly fall into two categories. On the one hand, delta debugging [25] and its derivatives [17] reduce inputs in a language-independent way by repeatedly removing parts of the input until no further reduction is possible. While being simple and elegant, these approaches disregard the language of the input and therefore miss opportunities for input reduction. In particular, these techniques cannot restructure inputs, which often enables further reductions.

As an example, consider the following JavaScript code and suppose that it triggers a bug, e.g., by crashing the underlying JavaScript engine.

```
for (var i = 0; i < 10; i++) {
  if (cond1 || cond2) {
    partOfBug();
  }
  if (cond3) {
    otherPartOfBug();
  }
}
```

Further suppose that the two function calls are sufficient to trigger the bug. That is, the following code is sufficient as a test input to enable a developer to reproduce and localize the bug:

```
partOfBug();
otherPartOfBug();
```

Unfortunately, existing language-independent techniques are challenged by this example. The original delta debugging algorithm blindly removes parts of the program, which is likely to lead to a syntactically invalid program or to a local minimum that is larger than the fully reduced example. Hierarchical delta debugging [17], a variant of delta debugging that considers the tree structure of the input, fails to find the reduced input because it can remove only entire subtrees, but it cannot restructure the input.

On the other hand, some techniques [19] exploit domain knowledge about the language of the test input. While being potentially more effective, hard-coding language knowledge into the approach limits it to a single kind of test input.

This paper presents the Generalized Tree Reduction algorithm (GTR), a language-independent technique to reduce arbitrary tree-structured test inputs. The approach is enabled by two key observations. First, we observe that transformations beyond removing entire parts of the input are beneficial in reducing inputs. GTR exploits this observation by incorporating tree transformations into the reduction process. The challenge is how to know which transformations to apply without hard-coding knowledge about a particular language. Second, we observe that for most relevant input formats, there are various examples that implicitly encode language information. For example, there are various programs in public code repositories, millions of HTML files, and many publicly available XML documents. GTR exploits this situation to automatically specialize a generic set of transformations for a particular language by learning from a corpus of example data.

Our work focuses on inputs that can be represented as a tree. This focus is motivated by the fact that the inputs of many program have an inherent tree structure, e.g., XML documents, or can be easily converted into a tree, e.g., the abstract syntax tree of source code. The input to GTR is a tree with a desirable property, such as triggering a bug, and an oracle that determines whether a reduced

ISSTA'17, Santa Barbara, California, USA
2016. 123-4567-24-567/08/06...$15.00
DOI: 10.475/123_4

version of the tree still has the desirable property. The algorithm reduces the tree level by level, i.e., it considers all nodes of a level to minimize the whole tree, before continuing with the next level. The output of the algorithm is a reduced tree that has the desirable property according to the oracle.

At the core of our approach is an extensible set of tree transformations that modify a tree into a new tree with fewer nodes. We describe two transformation patterns that we find to be particularly effective. The first pattern removes a node and all its children, drastically shrinking the tree's size. As deleting nodes alone is insufficient for various inputs, the second pattern replaces a node with one of its children, i.e., it pulls up a subtree to the next level of the tree. In principle, these transformation patterns are applicable to arbitrary kinds of nodes in the tree. To reduce the size of the search space considered by GTR, i.e., ultimately the time required to reduce an input, we specialize the transformation patterns to a specific input language by learning from a corpus of example data. Since the learning is fully automatic, the approach remains language-independent yet exploits language-specific language. While we find the above two transformation patterns to be effective, the algorithm is easily extensible with additional patterns.

To evaluate GTR, we apply the algorithm to a total of 48 inputs in the form of Python and JavaScript programs. The Python programs each trigger a bug in the Python interpreter, while the JavaScript programs cause inconsistencies between browsers. We find that GTR reduces the Python and JavaScript files by 50% and 96%, respectively, clearly outperforming the two closest existing techniques, Delta Debugging [25] and Hierarchical Delta Debugging [17].

To summarize, we make the following contributions:

- We identify the lack of restructuring as a crucial limitation of existing language-independent input reduction techniques.
- We present a novel tree reduction algorithm that transforms trees based on an extensible set of tree transformation patterns. If a set of example inputs is available, the approach automatically specializes the patterns to the language of the input.
- We show the presented algorithm to be significantly more effective than two state of the art techniques, without sacrificing efficiency.

The implementation of our approach is available for download.[1]

2 BACKGROUND

In this section we provide the necessary background information on previous work that our approach builds upon. In particular, we explain the Delta Debugging algorithm and its derivative Hierarchical Delta Debugging.

2.1 Delta Debugging

Zeller and Hildebrandt proposed Delta Debugging (DD) [25], a greedy algorithm isolating failure inducing inputs. In a nutshell, DD splits the input in chunks of decreasing sizes, trying to remove some chunks while maintaining a property of the input. "Chunk" can refer, e.g., to individual characters or lines of a document. Often but not necessarily, the property is that the input induces a bug when fed to a program. DD is fully automatic. The only input to the algorithm is the document to minimize and an oracle that checks whether a reduced test input still has the desired property.

DD does not guarantee to find the smallest possible input but instead ensure *1-minimality.* This property guarantees that no single part of the input can be removed without loosing the property of interest. For example, when applying line-based delta debugging to reduce a program that triggers a compiler bug, 1-minimality means that removing any line of the input will cause the input to not trigger the bug anymore.

DD has an important disadvantage for structured input. The algorithm disregards the structure of the input when splitting the input into chunks. As a result, the algorithm may generate various invalid inputs and invoke the oracle unnecessarily. For instance, when applying DD to the example from the introduction, the algorithm may delete a closing bracket without removing its counterpart, generating a syntactically invalid program. Since each candidate input is given to the oracle, such invalid inputs increase the execution time of the algorithm.

2.2 Hierarchical Delta Debugging

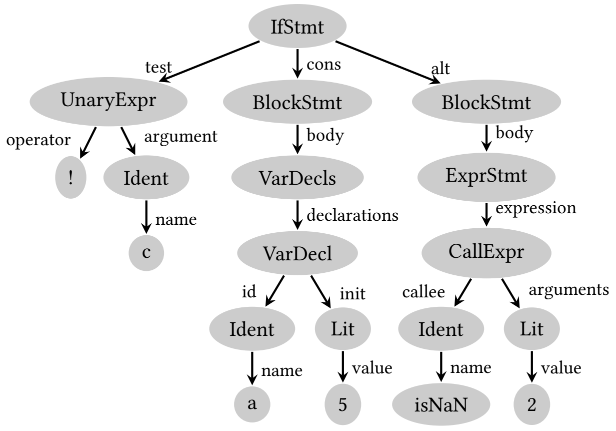
Hierarchical Delta Debugging (HDD) [17] addresses the limitation that DD disregards the structure of the input. To this end, the algorithm considers the input to be a tree, which is a natural way to interpret various inputs. For example, when referring to code as the input, one natural way to convert it into a tree is using the abstract syntax tree (AST). HDD starts from the root of the tree and treats each level successively. At every level, the algorithm applies the original DD algorithm to all nodes at this level to find the smallest set of nodes necessary on the particular level. The algorithm terminates after running DD on the last level of the tree.

HDD has the potential to purge large parts of the input early, which often leads to reduced inputs that are smaller than with DD, while also requiring fewer oracle invocations. In contrast to DD, HDD does not provide 1-minimality. To guarantee this property, another algorithm called HDD* is presented [17]. HDD* repeatedly uses HDD, until no more changes to the tree are performed.

While HDD improves over DD for tree inputs, it is limited to removing nodes (and all their children), but it does not use any other tree transformations. For an input where the important part is deeply nested in the input tree, HDD produces far-from-minimal results. An example is the code excerpt provided in the introduction. Here, HDD attempts to remove the entire if-branches, which yields a program that does not trigger the bug anymore. The algorithm yields the following code as the reduced input:

```
for (;;) {
  if (cond1 || cond2) {
    partOfBug();
  }
  if (cond3) {
    otherPartOfBug();
  }
}
```

[1] Note for double-blind reviewing: The following link points to a repository that reveals the identity of the author(s): https://goo.gl/0p2YJf



```
if(!c) {
  var a = 5; // root cause of failure
} else {
  isNaN(2);
}
```

Figure 1: An abstract syntax tree and code for our running example.

3 PROBLEM STATEMENT

Previous approaches for reducing failure-inducing inputs miss opportunities for reduction because they ignore the structure of the input and because they are limited to removing parts of the input. Motivated by these limitations, we aim for an algorithm that exploits the structure of the inputs, and that finds near-minimal results even when the root cause of a failure is deeply nested inside the input. This section introduces a running example and uses it to define the problem we are addressing.

Figure 1 shows our running example: a small piece of JavaScript code that triggers a bug, e.g., in a JavaScript engine. Suppose that the bug is triggered by the statement at line 2.

Our work focuses on inputs that can be represented as a tree.

Definition 3.1. A *labeled ordered tree* is a recursive data structure (l, c), where l is a textual label and c is the (possibly empty) ordered list of outgoing edges. An edge $e \in c$ is a tuple (l, t), where l is a textual label for the edge and t is the child node, which itself is a labeled ordered tree. We use $\mathcal{T}$ to refer to the set of all trees.

The example input can be represented as a tree – in this case, the abstract syntax tree, as shown in Figure 1.

We will refer to a labeled ordered tree hereafter simply as a tree or node, depending on the context. Trees have several properties. The *size* of a tree is the number of its nodes: $size : \mathcal{T} \rightarrow \mathbb{N}$. The tree in Figure 1 has a size of 19. The *context* of a tree is a partial function that returns the label of the parent node and the label of the incoming edge: $context : \mathcal{T} \rightarrow (String \times String)$. The context of the root node is undefined. For the example, the context of the *UnaryExpr* node on the left side of our tree is (*IfStmt*, *test*). The *level* of a node in a tree is the edge-distance from the node to the tree's root node. All nodes of a particular level in a tree can be obtained by a function $level : (\mathcal{T} \times \mathbb{N}) \rightarrow P(\mathcal{T})$, where P denotes the power set. $level(t, 0)$ will return just $\{t\}$. The *depth* of a tree is defined as the maximum distance of a leaf node to the root: $depth : \mathcal{T} \rightarrow \mathbb{N}$. The example tree has a depth of 5. Finally, we say that a tree t' is derived from another tree t, written $derived(t', t)$, if one can build t' from t by (i) deleting nodes and edges and by (ii) moving nodes and edges within the tree without changing a single label.

Definition 3.2. An *oracle* o is a function that, given a tree, decides whether the tree provides a desired property: $o : \mathcal{T} \rightarrow Bool$. We use $\mathcal{O}$ to denote the set of all oracles.

A tree t is *minimal* w.r.t. an oracle o and a source tree st if t satisfies the oracle and if there is no smaller derived tree that also satisfies the oracle. Formally, t is minimal if $derived(t, st) \wedge o(t) =$ true $\wedge \left(\nexists t' \neq t : derived(t', st) \wedge o(t') = true \wedge size(t') < size(t)\right)$.

Definition 3.3. A *tree reduction algorithm* is a function $A : (\mathcal{T} \times \mathcal{O}) \rightarrow \mathcal{T}$ that, given a tree t and an oracle o where $o(t) =$ true, returns another tree t' for which $o(t') =$ true and $size(t') \leqslant size(t)$.

Put simply, A tries to find a smaller tree that still provides a property of interest, as decided by the oracle. If a reduction algorithm cannot further reduce a tree, it will return the same tree.

The goal of this work is to provide a tree reduction algorithm that returns near-minimal trees with respect to the given oracle, while maintaining the number of oracle invocations low. A small number of oracle invocations is important, as they can be costly operations, such as running a compiler, that significantly increase the overall runtime of the algorithm. In general, finding the minimal tree is impractical because the number of trees to check with the oracle grows exponentially with the size of the input tree.

4 THE GENERALIZED TREE REDUCTION ALGORITHM

This section introduces a novel tree reduction algorithm, called *Generalized Tree Reduction* or GTR. Figure 2 shows the components of the approach and how they interact with each other. Given an input tree (step 1), the algorithm traverses the tree from top to bottom while applying transformations to reduce the tree. For example, a transformation may remove an entire subtree or restructure the nodes of the tree. The transformations are based on an extensible set of tree transformation templates (step 2) that specify a set of candidate transformations (step 3). To specialize a generic template to a particular input format, the approach filters these candidates based on knowledge inferred from a corpus of example inputs (step 4). The algorithm applies the transformations and queries the oracle to check whether a reduced tree preserves the property of interest, e.g., whether it still triggers a particular bug (steps 5 and 6). The algorithm repeatedly reduces the tree until no more tree reductions are found. Finally, GTR returns the reduced tree (step 7).

We call the GTR algorithm "generalized" because it can express a variety of different tree reduction algorithms, depending on the provided tree transformation templates. For example, by providing a single template that reduces entire subtrees, GTR is equivalent to the existing HDD algorithm [17] (Section 4.5).

Before delving into the details of GTR, we illustrate its main ideas using the running example in Figure 1. Given the tree representation of the input, the algorithm analyzes the tree level by

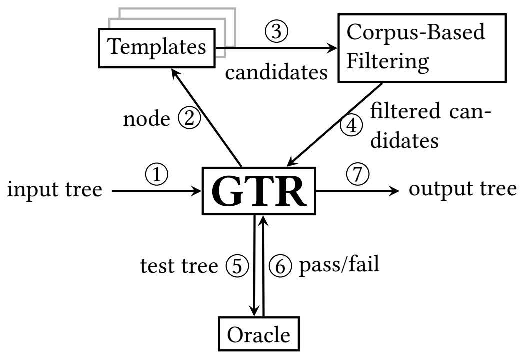


Figure 2: High-level overview of the GTR approach.

Algorithm 1 Deletion template

```
Input: a tree tree
Output: a set of candidates trees
1: function DEL_TEMPLATE(tree)
2:     return {DEL}
```

level, starting at the root node. For example, the algorithm considers a transformation that removes the root node *IfStmt* and all its children, but discards this transformation because the reduced tree (an empty program) does not trigger the bug anymore. As another example, the algorithm considers transformations that replace the root node with one of its children. Replacing the root node with the *BlockStmt* that represents the then-branch yields a smaller tree that still triggers the bug. Therefore, the algorithm applies this transformation and continues to further reduce the remaining tree. Eventually, the algorithm reaches a tree that represents only the statement `var a = 5;`, which cannot be reduced without destroying the property of interest.

The remainder of this section explains the GTR algorithm in detail. At first, we present the tree transformations applied by the algorithm (Sections 4.1 and 4.2). Then, we describe how GTR combines different transformations into an effective tree reduction algorithm (Section 4.3).

4.1 Tree Transformation Templates

The core ingredient of GTR are transformations that reduce the size of a tree. We specify such transformations with templates:

Definition 4.1. A *transformation template* is a function $\mathcal{T} \rightarrow P(\mathcal{T}) \cup \{DEL\}$ that returns a set of candidate trees that are the result of transforming a given input tree. In addition to candidate trees, the template may return the special symbol *DEL*, which indicates that the tree should be removed rather than modified.

In general, templates may return multiple candidates of transformed trees for a single given tree. We call templates that always return only one candidate tree *at-most-one templates.* In this paper, we focus on two transformation templates, which yield a tree reduction algorithm that is more effective than the best existing algorithms. The GTR approach is easily extensible with additional templates.

Algorithm 2 Substitute-by-child template

```
Input: a tree tree
Output: a set of candidates nodes
1: function SBC_TEMPLATE(tree)
2:     $candidates \leftarrow \emptyset$
3:     for $i \in [0, |tree.c|]$ do
4:         $c \leftarrow tree.c[i].t$
5:         $candidates \leftarrow candidates \cup \{c\}$
6:     return candidates
```

Deletion template. The first template addresses situations where an entire subtree of the input given to GTR is irrelevant for the property of interest. In our running example (Figure 1), the subtree rooted at the right-most *BlockStmt* node is such a subtree. To enable GTR to remove such subtrees, the deletion template (Algorithm 1) simply suggests for each given tree to delete it by returning the special *DEL* symbol. This template is an at-most-one template because it returns exactly one candidate for each given tree.

Substitute-by-child template. The second template addresses situations where simply removing an entire subtree is undesirable because the subtree contains nodes relevant for the property of interest. We observe that a common pattern is that the root node of a subtree is irrelevant but one of its children is important for the property of interest. In the running example, the tree rooted at the *IfStmt* matches this pattern, because the if statement is irrelevant, but the variable declaration nested inside it is crucial. To address this pattern, the substitute-by-child template (Algorithm 2) returns each child of a given tree's root node as a candidate for replacing the given tree. The template iterates over all children of the given tree and adds each of them to the set of candidates. Applying this transformation template to the *IfStmt* of the running example yields a set of three candidates, namely the three subtrees rooted at nodes *UnaryExpr*, *BlockStmt*, and *BlockStmt*.

Beyond these two templates, additional templates can be easily integrated into GTR, enabling the approach to express different tree reduction algorithms. For example, other transformation templates could swap sibling nodes inside a given tree or replace the root note of a tree with one of its grand-children.

4.2 Corpus-Based Filtering

The templates defined above are completely language independent. When applying these templates to a tree that ought to conform to a specific input format, many of the candidates may be rejected by the oracle simply because they violate the input format. For example, when the deletion template suggests to remove the *UnaryExpr* from the tree in Figure 1, the resulting tree corresponds to syntactically invalid JavaScript code because every if statement requires a condition. Suggesting such invalid candidates does not influence the effectiveness of our approach because the oracle rejects all invalid candidates. However, a high number of invalid candidates negatively influences the efficiency of the approach since invoking the oracle often imposes a significant runtime cost.

To address the challenge of invalid candidates, we enhance the approach with a language-dependent filtering of candidates trees that rejects invalid trees before invoking the oracle. To preserve the

Algorithm 3 Specialized deletion template

```
1: function DEL_TEMPLATE($tree$)
2:     $(P, e) \leftarrow context(tree)$
3:     if $e \notin mandatory\_edges(P)$ then
4:         return $\{DEL\}$
5:     else
6:         return $\{\}$
```

language-independence of GTR, the filtering is based on knowledge that gets automatically inferred from a corpus of example inputs in the specific input format. For example, the approach learns from a set of JavaScript programs that if statements require a condition, and therefore, will filter any candidate that violates this requirement.

Deletion template. To specialize the deletion template to a particular language, we need to know which edges are mandatory for particular node types. We analyze the code corpus to find a set of mandatory edge labels for each node label L. An edge is considered mandatory, if it appears on *all* nodes with the label, inside the whole corpus. This set for a particular label can be obtained by a function $mandatory\_edges$. In the next step, the deletion template is modified so that a node can only be deleted if it is not a mandatory child of its parent node. Algorithm 3 shows the specialized variant of the deletion template.

For the running example, consider again the candidate that suggests to remove from the *IfStmt* the *UnaryExpr* subtree. The corpus analysis finds that the set of mandatory edges for an `IfStmt` is {`test`, `cons`} (Section 5.4 provides details on the code corpora used). Based on this inferred knowledge, the algorithm will not attempt to delete the `UnaryExpr` any more, but rejects this candidates before needlessly passing the tree to the oracle.

Substitute-by-child template. To specialize the substitute-by-child template, we gather information on the parent node labels and incoming edge labels of nodes. In particular, for each node label L we collect a set of pairs (P, e) where P is the label of the parent, and e is the label of the ingoing edge. This set of pairs is equivalent to all distinct contexts (defined by the function *context*) of nodes with that label. The set can be obtained by a function *contexts*, given the node label. We then replace line 5 of the substitute-by-child template (Algorithm 2) with the following steps:

```
5: if $context(tree) \in contexts(tree.c[i].l)$ then
6:     $candidates \leftarrow candidates \cup \{c\}$
```

The specialized variant of the template checks if the child that we replace the node with can also appear in the same context as the node. For example, the approach infers that there is one valid context for a *VarDecl* node, namely *(VarDecls, declarations)*. Since *(BlockStat, body)* is not a valid context, the algorithm will immediately reject a candidate that tries to substitute `VarDecls` with `VarDecl`.

Inferring from a corpus of examples how to specialize language-independent transformation templates to an input format is a deliberate design decision. The rationale is the observation that for most input formats used in practice, there are sufficiently many examples to learn from available. An alternative approach could be to use a formal grammar of the input language to filter syntactically

Algorithm 4 Generalized tree reduction

```
Input: tree $t$, oracle $o$, set $\mathcal{L}$ of templates
Output: reduced tree
1: for $i \in [0, depth(t)]$ do
2:     for $l \in \mathcal{L}$ do
3:         $t \leftarrow$ APPLYTEMPLATE($t, i, o, l$)

4: function APPLYTEMPLATE($t, i, o, l$)
5:     $levelNodes \leftarrow level(t, i)$          ▷ All nodes of level $i$
6:     if $l$ is at-most-one template then
7:         $newNodes \leftarrow$ apply DD to replace $levelNodes$ using $l$
8:         return tree where $newNodes$ replace $levelNodes$
9:     else
10:        return REDUCELEVELNODES($t$, $levelNodes$, $o$, $l$)
```

invalid trees. We rejected this idea because (i) a grammar may not be available, e.g., for proprietary formats, (ii) the checks performed by the specialized transformation templates are more lightweight than parsing the entire input tree with a grammar. Our evaluation assesses the effectiveness and efficiency of GTR with and without the corpus-based filtering of candidate trees (Section 5.4).

4.3 GTR Algorithm

Based on the transformation templates described above, the GTR algorithm reduces a given tree by applying transformations at each level of the tree. Algorithm 4 summarizes the main steps. Starting at the root node, the algorithm considers each level of the tree and applies all available transformation templates to each level using a helper function $applyTemplate$ (lines 1 to 3).

Handling At-Most-One Templates via Delta Debugging. When applying a template to the nodes at a particular level, the algorithm distinguishes between at-most-one templates and other templates. Since an at-most-one template suggests at most one candidate for each node, the algorithm needs to decide which of the suggested replacements to apply. This problem can be reduced to Delta Debugging (DD). The chunks needed as input for DD are the nodes of the level. DD then tries to combine as many replacements as possible while querying the oracle to check if a replacement preserves the property of interest. For each node n for which the transformation template returns a node n', DD will try to replace n with n'. For each node, where the symbol DEL is returned instead, DD will try to delete n. After deciding on the replacements, the result is a new list of nodes for the current level. The helper function $applyTemplate$ replaces the nodes on the level with the new nodes and returns the resulting tree to the main loop of the algorithm (lines 7 and 8).

Backtracking-based Search for Other Templates. For templates that may return more than one candidate, the algorithm must decide not only whether to apply a candidate replacement but also which of the suggested candidate replacements to apply. This problem cannot be easily mapped to DD because DD assumes to have exactly one option per chunk (typically, whether to delete it or not). Instead, we present a backtracking-based algorithm that searches for a replacement of nodes on a particular level that reduces the

Algorithm 5 Backtracking-based reduction of level nodes

```
Input: tree $t$, list of $nodes$ on the same level, oracle $o$, template $t$
Output: reduced tree
    ▷ Maps each node to its current replacement:
 1: $conf \leftarrow$ empty map
 2: for $n \in nodes$ do
 3:     $conf.put(n, n)$
 4: repeat
 5:     $improvementFound \leftarrow false$
 6:     for $n \in nodes$ do
 7:         $currentRep \leftarrow conf.get(n)$
 8:         for $n' \in l(n)$ where $size(n') < size(currentRep)$ do
 9:             $t' \leftarrow t$ with each $n$ replaced by $n'$
10:             $conf.put(n, n')$
11:             if $oracle(t')$ then
12:                 $improvementFound \leftarrow true$
13:                 $currentRep \leftarrow n'$
14:             else
15:                 $conf.put(n, currentRep)$                ▷ Backtrack
16: until $\neg improvementFound$
17: return $t'$
```

overall tree. Similar to DD, the algorithm is a greedy search for a local optimum.

Algorithm 5 summarizes the main steps of the backtracking-based search for replacements of nodes on a particular level. The algorithm is called at line 10 of the main GTR algorithm. The central idea is to try different *configurations* that specify which replacements to use for each node. The algorithm starts with a configuration that replaces each node with itself (lines 1 to 3). Then, the algorithm iterates through all nodes (line 6) and tries all configurations where the replacement candidate is smaller than the currently chosen replacement. That is, the algorithm avoids invoking the oracle for replacements that are less effective than an already found replacement. If the oracle confirms that replacing a node preserves the property of interest, an improvement was found w.r.t to the current replacement (lines 12 and 13). Otherwise, the algorithm must revert the replacement and backtracks to the previous configuration (line 15).

The algorithm repeats the search for a replacement of any of the nodes on the current level until no further improvement is found. The reason for repeatedly considering the list of nodes is that using an effective replacement at a later node may enable using previously impossible replacements at previous nodes, which have already been tested in the current iteration. For example, consider the following input, where the `crash(b);` call ensures the property of interest:

```
a = b = 0;
if (a)
  crash(b);
```

During the first iteration of the main loop (lines 4 to 16), the algorithm cannot reduce the assignments in the first line but reduces the input by substituting the if-statement with the `crash(b);` call. Now, during the second iteration, the algorithm again considers the assignment statement and successfully reduces it to `b = 0;`, which yields the following reduced input:

```
b = 0;
crash(b);
```

The search for a reduction of the nodes in the current level guarantees to find a local optimum, i.e., a configuration where using any other replacement that yields a smaller subtree would dissatisfy the oracle. As the search is greedy, we may miss a configuration that yields a smaller overall tree satisfying the oracle. Searching for a global optimum would require to explore all possible configurations, which is exponential in the number of candidates suggested.

Example. We illustrate GTR on the running example. Recall that only line 2 of Figure 1 is relevant for reproducing the bug. The algorithm starts on level 0, which contains only the `IfStmt`, and invokes *applyTemplate* with the deletion template. Deletion is an at-most-one template. Therefore, the algorithm applies DD to the node on this level; and tries to delete it with all its children, but this would make the bug disappear and is discarded by the oracle.

Next, `applyTemplate` is invoked with the substitute-by-child template. Since this template is not an at-most-one template, the algorithm invokes the backtracking-based *reduceLevelNodes* function, i.e., Algorithm 5. There is only one node to consider in line 6 of Algorithm 5, and in line 8 three different candidates are tested. The first is the `UnaryExpr` on the left side. This candidate has a size of 4. But, since the important code piece is removed, line 15 reverts this change. The next candidate is the `BlockStatement` in the middle. It has a size of 7. The oracle returns true for this transformation, so *currentRep* is updated in line 13. The third candidate is the `BlockStatement` on the right. Since it also has a size of 7, which is not smaller than the size of *currentRep*, the candidate is not tested. Now that an improvement was found, the main loop (lines 4 to 16) is repeated. As there is only one node, nothing new will be tested. After having finished both templates on level 0, GTR will advance to level 1 and continue in the same manner.

4.4 GTR* Algorithm

The existing DD and HDD* algorithms guarantee 1-minimality (or 1-tree-minimality). In essence, this property states that, given a reduced input, there is not single reduction step that can further reduce the input without destroying the property of interest. Similar to this property, we define a similar property for GTR:

Definition 4.2. A tree t is called *1-transformation-minimal* w.r.t. an oracle o and a set of templates $\mathcal{L}$ if $o(t)$ = true $\land \forall n$ in t and $\forall l \in \mathcal{L}$, there is no candidate n' in $l(n)$ that, when replacing n with n' yields a tree t' with $o(t') = true \land size(t') < size(t)$.

In other words, using the oracle to try all trees that are obtained by single replacements of one node of the tree will always return $false$ for 1-transformation-minimal trees. The main difference to the existing 1-minimality [25] and 1-tree-minimality [17] properties is to consider arbitrary tree transformations.

The GTR algorithm does not guarantee to find a 1-transformation-minimal tree. The reason is that by optimizing a tree on one level, a transformation on a higher level, which had been rejected by the oracle before, can become possible. To guarantee 1-transformation-minimality, we present a variant of GTR, the GTR* algorithm

Algorithm 6 GTR*

```
Input: tree $t$, oracle $o$, set $\mathcal{L}$ of templates
Output: 1-transformation-minimal tree
1: $current \leftarrow t$
2: repeat
3:     $previous \leftarrow current$
4:     $current \leftarrow GTR(previous, o, \mathcal{L})$
5: until $size(previous) = size(current)$
6: return $current$
```

(Algorithm 6). GTR* repeats GTR until the tree does not change its size any more, which indicates that no transformation can be applied thereafter.

4.5 Generalization of HDD and HDD*

GTR and GTR* generalize the existing HDD and HDD* algorithms, respectively. To obtain HDD, we configure GTR to include only the deletion template (Algorithm 1), without specializing the template to a particular language. The resulting algorithm applies DD on every level of the input tree by deleting a subset of the nodes on this level. This behavior is exactly what HDD does, i.e., the reduced tree is the same as returned by HDD. This generalization also applies to HDD*, where we simply run GTR* on the variant of GTR that is equivalent to HDD. Beyond the existing HDD and HDD* algorithms, GTR can express other tree reduction algorithms by including different sets of transformation templates.

5 EVALUATION

We evaluate GTR by applying it to two input formats: programs written in Python and JavaScript, which trigger some kind of misbehavior in their respective execution environment. The evaluation compares GTR and GTR* to the existing DD, HDD, and HDD* algorithms. We focus on three research questions:

- RQ1: How effective is the approach in reducing trees?
- RQ2: How efficient is the approach in reducing trees?
- RQ3: What is the effect of specializing transformation templates to an input format, compared to omitting this part of the approach?

5.1 Experiment Setup

Input Data and Oracles. We consider two sets of inputs: 7 Python files that crash the Python interpreter and 41 JavaScript files that cause inconsistencies between browsers. Both input formats can be naturally represented as a tree based on their abstract syntax trees. We use the Python parser from the AST library and the Esprima parser to transform source code files into trees.

To obtain failure-inducing Python files, we searched the official bug tracker of the for segmentation faults and stack overflows reported along with code to reproduce it. Table 1a lists the Python files we found, which version of the Python interpreter they break, and links to the corresponding issues. Because these files have been reported by users or developers, they are likely to have been manually reduced, presenting a non-trivial challenge to any input reduction algorithm. As the oracle to check whether a reduced Python file preserves the property of interest we check the status code returned by the Python interpreter when executing the file.

To obtain JavaScript files that cause inconsistencies between browsers, we use files generated by TreeFuzz [18], an existing fuzz testing technique. We configure TreeFuzz to generate 3,000 files and keep all files that trigger a browser inconsistency, which results in 41 files. Table 1b summarizes their size and number of lines of code. Since these files are automatically generated, they generally contain parts that are not required to trigger the property of interest, i.e., a browser inconsistency, providing a good data set to complement the manually written Python files. As the oracle we compare the runtime behavior of a JavaScript file in Firefox 25 and Chrome 48, as described in [18].

Corpora of Input Data. To automatically specialize the transformation templates to a particular language, our approach learns from a corpus of examples of input data. We use publicly available code corpora or code examples for our two target languages. The Python corpus consists of 900 files gathered from popular GitHub projects. The JavaScript corpus comprises around 140,000 files provided as part of an existing "big code" corpus [5].

State of the Art Approaches. We compare our approach to our own implementations of the existing DD, HDD, and HDD* algorithms. The DD implementation works on the line-level, i.e., each line of the program is a chunk considered by DD. The HDD implementation uses the same tree representation of the inputs as the GTR implementation.

5.2 Effectiveness

To evaluate effectiveness, we apply GTR and GTR* to reduce the given Python and JavaScript files while preserving their problem-inducing behavior. To compare to the state of the art, we also apply DD, HDD, and HDD* to these files. Figures 3a and 3b summarize the results by showing how much each algorithm reduces the inputs. Each vertical bar represent one file in its original size. Each symbol on that bar shows how much a particular algorithm reduces that file. Figure 3a uses a logarithmic scale, because the size of the test inputs varies significantly.

For the Python data set, we observe that GTR* consistently produces the smallest files and GTR the second smallest. Some files can be reduced significantly. For example, itertools.py was cut from 1,575 characters to 438 characters using GTR*. Other files, such as mroref.py, cannot be reduced that much (383 characters before and 318 after applying GTR*). The reason is that these inputs are diverse in size and that some of them have been reduced manually before reporting them to the Python developers, which leaves little room for any subsequently applied tree reduction algorithm.

For the JavaScript data set, we also observe even larger reductions (by all algorithms), sometimes removing more than 99% of the file. The main reason is that these files are generated by a fuzz tester and have not been processed by a human. Similar to the Python data set, GTR and GTR* consistently outperform all other algorithms.

To summarize the comparison of the effectiveness, Figure 3c shows the relative reduction of file sizes for all five algorithms. For Python, taking the median, GTR reduces files 50%, whereas the best existing algorithm, HDD, achieves only a median reduction

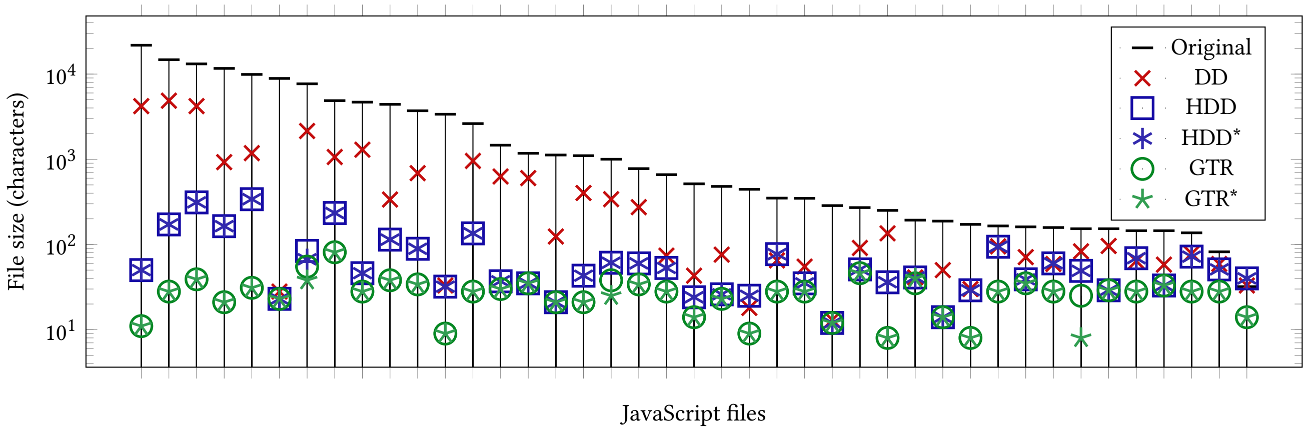
Name	Bytes	Lines	Py. version	Issue URL
ackermann.py	483	19	3.4.3	http://stackoverflow.com/questions/27269576
alloc.py	1125	44	2.7.6	http://bugs.python.org/issue26595
dict.py	265	21	2.7.6	http://hg.python.org/cpython/file/default/Lib/test/crashers/underlying_dict.py
itertools.py	1574	73	2.7.6	http://bugs.python.org/issue24482
mroref.py	383	22	2.7.6	http://svn.python.org/view/python/trunk/Lib/test/crashers/loosing_mro_ref.py
recursion.py	212	22	3.4.3	http://bugs.python.org/issue6717
so.py	664	27	3.4.3	http://stackoverflow.com/questions/19127777

(a) Files in the Python data set.

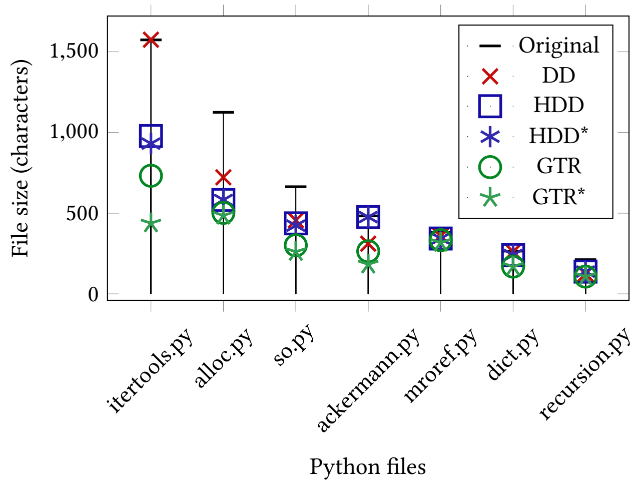
	Bytes	Lines
Minimum	32	1
Maximum	21806	458
Average	3016	86

(b) Summary of files in the JavaScript data set.

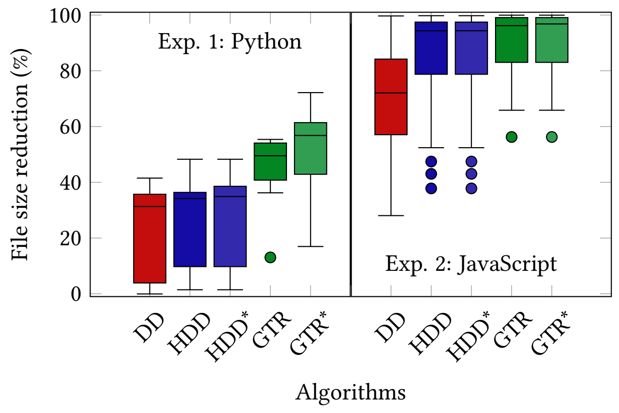
Table 1: Files in the Python and JavaScript data sets.



(a) Sizes of the JavaScript files, before and after applying tree reduction algorithms. Note the logarithmic scale.



(b) Sizes of the Python files, before and after applying tree reduction algorithms.



(c) File size reduction achieved by different tree reduction algorithms. The boxes indicate the median and the first and third quartiles. The whiskers include up to 1.5 inter-quartile-ranges above and below the box.

Figure 3: Effectiveness evaluation with two experiments.

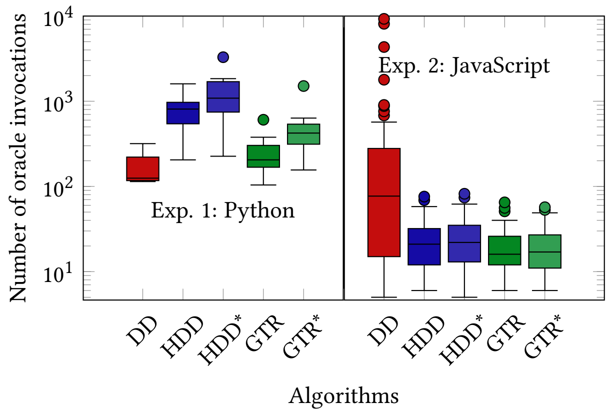


Figure 4: Number of oracle invocations. Note the logarithmic scale. The boxes indicate the median and the first and third quartiles. The whiskers include up to 1.5 inter-quartile-ranges above and below the box.

of 34%. GTR* is even more effective, removing 57% of the original files. For JavaScript, GTR and GTR* reduce file sizes by 96 and 97%. In contrast, DD achieves only 72% median reduction; HDD and HDD* reduce the files by 94%. Even though the difference in percentage between GTR and HDD is relatively small for JavaScript, it is important to note that this difference corresponds to up to 309 characters, and 25 characters taking the median, which makes a significant difference for debugging.

5.3 Efficiency

We evaluate the efficiency of our approach by measuring the number of oracle invocations required to reduce a tree. Using this metric instead of, e.g., wall clock time, is motivated by two reasons. First, invoking the oracle typically is the most important operation during automated input reduction, because it often involves running a complex piece of software, such as a compiler or interpreter, on non-trivial inputs, such as large programs. Second, wall clock time is highly dependent on the implementation of the tree reduction and the oracle. To reassure ourselves this metric is appropriate, we preliminarily compared the time during oracle invocations versus the time in other parts of the algorithm with our JavaScript data set. For each algorithm, the oracle invocation time dominates and comprises more than 98% of the total execution time, on average.

Figure 4 shows how many oracle invocations GTR, GTR*, and the other algorithms require to reduce a single file. Using the median, for the Python data set, GTR needs 64% more invocations than DD but HDD needs 295% more invocations than GTR. The GTR* and HDD* algorithms both need more invocations than their *-less counterparts, which is unsurprising because they run the algorithm, including oracle invocations, multiple times. For the JavaScript data set, GTR needs fewer invocations than all other algorithms. The large difference in the results for DD can be explained by the size of the input files. Since the Python files are relatively small and cannot be reduced as much as the relatively large JavaScript files, DD reduces them quickly. In contrast, the structure-unaware search of DD takes significantly more oracle invocations for larger files. To summarize, GTR is either more efficient or only slightly less efficient than the best existing approach.

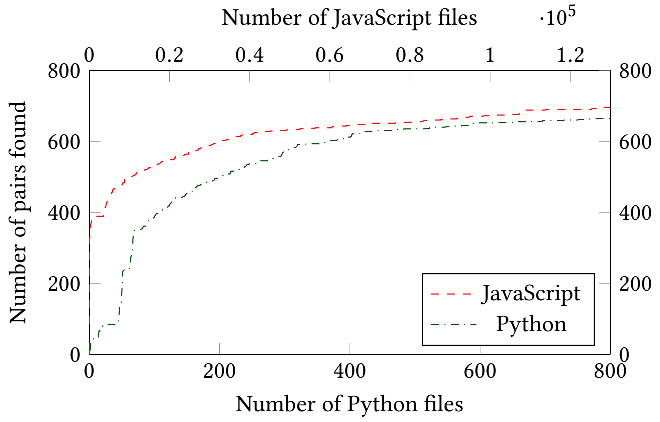


Figure 5: Learning curve for Python and JavaScript corpora.

5.4 Benefits of Corpus-Based Filtering

Our approach specializes language-independent transformation templates to a specific input language by learning filtering rules from a corpus of examples of inputs. To evaluate how the corpus-based filtering influences the effectiveness and efficiency of GTR, we compare the approach with a variant of GTR that does not filter any candidate transformations. For both variants, we perform the same experiments as described in Sections 5.2 and 5.3.

The comparison shows that both variants of GTR are equally effective and that corpus-based filtering significantly improves the efficiency of the algorithm:

The GTR variant without filtering of candidates achieves the same effectiveness for JavaScript and slightly higher reductions (5%) for Python. The reason is that the corpus does not mirror all facets of the target languages, which may cause the filtering to overly constrain the transformations. For example, if the corpus would not contain any if-statement without an else-branch, then GTR would not consider removing the else-branch. Fortunately, the results show that such overly constrained filtering is very unlikely.

The GTR variant without filtering needs significantly more oracle invocations. For the Python data set, the variant needs 659.1 invocations, whereas the full GTR approach needs only 265.3 invocations, on average. For the JavaScript data set, the results are 33.5 and 21.0 invocations without and with filtering, respectively.

We conclude that the corpus-based filtering achieves its goal of reducing the number of oracle invocations without sacrificing the effectiveness of the algorithm.

To better understand what size a corpus of examples should have, we measure how much knowledge about the input format our approach infers depending on the corpus size. We focus this experiment on the knowledge inferred for the substitute-by-child template, where the approach gathers pairs (P, e), where P is the label of a parent node and e is the label of the corresponding incoming edge (Section 4.2). A corpus with too few examples may prevent

GTR from performing a beneficial transformation, simply because there was no (P, e) pair in the corpus that makes the transformation plausible. Figure 5 shows the number of (P, e) pairs over the number of analyzed files. For both Python and JavaScript, the curve rises steeply in the beginning and flattens out afterwards, stalling at around 700 entries. Only very few new entries are found towards the right of the graph. These results suggest that a relatively small corpus of examples is sufficient to infer most knowledge.

Finally, we measure how long extracting language-specific information from the corpus of examples takes. In total, extracting this knowledge takes around 19 seconds and 21 minutes for the Python and JavaScript data set, respectively.

6 RELATED WORK

We divide the related work into three different categories: test input reduction, fault localization, and test suite reduction.

6.1 Minimizing Test Inputs

Delta Debugging (DD) [25] set the foundations to automatize the process of minimizing test inputs. In contrast to GTR, it cannot handle large structured inputs effectively. Recently, a modernized version explores parallelization of DD [11], which is orthogonal to our contributions.

The hierarchical version HDD [17] applies DD on hierarchical documents. However, HDD fails to restructure trees in a way that allows obtaining significantly smaller results, and also performs less efficient than GTR. An improved HDD version proposed later uses a different kind of grammars. This targets the conversion of code documents to trees and is complementary to our findings. Another contribution that both HDD and GTR could benefit from, is automatic input syntax derivation with dynamic analysis [16]. This could substitute a grammar in HDD (or a language parser in GTR), so that targeting new languages becomes easier.

C-Reduce [19] is an implementation of DD, that applies domain-specific transformations when reducing C code. These transformations include changing identifiers and constants, removing pairs of parentheses or curly braces, or inlining functions. C-Reduce is built as an extension to a randomized test-case generator [23]. One big advantage of C-Reduce is never producing any input with non-deterministic or undefined behavior. There are two big differences in comparison to our approach. First, the transformations are source-to-source and not tree-to-tree. Second, C-Reduce loses generality by applying domain-specific changes to the document. However, many of the transformation included by C-Reduce can be expressed in a more general way and are included in our approach. For example, “removing an operator and one of its operands (e.g., changing a+b into a or b)” is equivalent to replacing the operator node with one of its children in the tree.

When the input to a program is not as complex as a code document itself, more efficient techniques can be employed. One possibility is to minimize the path constraints of the input that led to a particular failure [4]. However, the set of path constraints grows exponentially for more complex inputs, rendering this approach unfeasible for code inputs.

Another interesting approach taints parts of each input to identify the parts relevant to a failure [7]. The default setting taints each byte independently, making it possible to also account for complex inputs. At the same time, this disregards the structure of the document, similar to DD, and thus becomes overly expensive. Another setting tracks inputs on a per-entity basis, which is insufficient for test input reduction.

6.2 Fault localization

A set of different techniques aims at locating the fault in the buggy program itself, instead of the input. Zeller applied DD also to the program with this goal [24]. There are various other approaches for fault localization not building upon DD [2, 12, 20].

One approach, which minimizes a program while maintaining its behavior with respect to a particular variable, is program slicing [22]. Ideally the smaller slice contains the bug and eases its localization. The dynamic variant [1] focuses on the subset of the program that give a variable its value *with the current input*. Just slicing the variables that appear in the line causing the bug (if known) does not guarantee to obtain a program that produces the same buggy behavior, though. The combination of DD with dynamic forward and backward slicing has also been explored previously [9].

Yet another technique reduces recorded traces to find shorter program executions with the observed buggy behavior [6, 10, 13], which ultimately also reveals probable locations for the bug.

Fault localization and test input reduction have different goals. In a first step, a tester confronted with a failure needs a small (and fast running) input to reproduce the failure. In a second step, the bug must be located in the program and fixed. Finally, the small input can be turned into a regression test. Thus, both techniques complement each other.

6.3 Minimizing Test Suites

While randomly generating tests, high code coverage can be achieved. The tests in randomly generated suites are often rather big and can also benefit from a reduction with DD [14, 15]. To maintain the good coverage of the test suite, the oracle can be modified to account for that, instead of testing for particular failures [3, 8].

7 CONCLUSION

We presented GTR, a novel algorithm to reduce tree-structured test inputs in a generalized and language-independent way. Our algorithm makes use of arbitrary tree transformations and applies them hierarchically to obtain a smaller test input. It uses both Delta Debugging and a greedy backtracking-based search to choose which transformations to apply. To improve over transforming blindly, we specialize the transformation templates by analyzing large code corpora, increase the efficiency.

We evaluate our approach against the baseline for reducing test input: Delta Debugging; and another hierarchical algorithm: HDD. Using a total of 48 test inputs in Python and JavaScript we find that GTR outperforms existing techniques regarding its reduction. The algorithm reduces these files by 50% and 96%, respectively. GTR is either slightly less or more efficient than the best existing approach.

REFERENCES

[1] Hiralal Agrawal and Joseph R Horgan. 1990. Dynamic program slicing. In *ACM SIGPLAN Notices*, Vol. 25. ACM, 246–256.

[2] Hiralal Agrawal, Joseph R Horgan, Saul London, and W Eric Wong. 1995. Fault localization using execution slices and dataflow tests.. In *ISSRE*, Vol. 95. 143–151.

[3] Mohammad Amin Alipour, August Shi, Rahul Gopinath, Darko Marinov, and Alex Groce. 2016. Evaluating non-adequate test-case reduction. In *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*. IEEE, 16–26.

[4] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Danny Dig, Amit Paradkar, and Michael D Ernst. 2008. Finding bugs in dynamic web applications. In *Proceedings of the 2008 international symposium on Software testing and analysis*. ACM, 261–272.

[5] Multiple authors. 2016. Learning from "Big Code". (2016). Retrieved January 19, 2017 from http://learnbigcode.github.io/datasets/

[6] Martin Burger and Andreas Zeller. 2011. Minimizing reproduction of software failures. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, 221–231.

[7] James Clause and Alessandro Orso. 2009. Penumbra: automatically identifying failure-relevant inputs using dynamic tainting. In *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM, 249–260.

[8] Alex Groce, Mohammed Amin Alipour, Chaoqiang Zhang, Yang Chen, and John Regehr. 2014. Cause reduction for quick testing. In *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on*. IEEE, 243–252.

[9] Neelam Gupta, Haifeng He, Xiangyu Zhang, and Rajiv Gupta. 2005. Locating faulty code using failure-inducing chops. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. ACM, 263–272.

[10] Mouna Hammoudi, Brian Burg, Gigon Bae, and Gregg Rothermel. 2015. On the use of delta debugging to reduce recordings and facilitate debugging of web applications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 333–344.

[11] Renáta Hodován and Akos Kiss. 2016. Practical Improvements to the Minimizing Delta Debugging Algorithm. In *Proceedings of the 11th International Joint Conference on Software Technologies*. 241–248.

[12] Tom Janssen, Rui Abreu, and Arjan JC van Gemund. 2009. Zoltar: A toolset for automatic fault localization. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 662–664.

[13] Manu Jose and Rupak Majumdar. 2011. Cause clue clauses: error localization using maximum satisfiability. *ACM SIGPLAN Notices* 46, 6 (2011), 437–446.

[14] Yong Lei and James H Andrews. 2005. Minimization of randomized unit test cases. In *Software Reliability Engineering, 2005. ISSRE 2005. 16th IEEE International Symposium on*. IEEE, 10–pp.

[15] Andreas Leitner, Manuel Oriol, Andreas Zeller, Ilinca Ciupa, and Bertrand Meyer. 2007. Efficient unit test case minimization. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 417–420.

[16] Zhiqiang Lin and Xiangyu Zhang. 2008. Deriving input syntactic structure from execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. ACM, 83–93.

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

[18] Jibesh Patra and Michael Pradel. 2016. *Learning to Fuzz: Application-Independent Fuzz Testing with Probabilistic, Generative Models of Input Data*. Technical Report TUD-CS-2016-14664. TU Darmstadt, Department of Computer Science.

[19] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 335–346.

[20] Manos Renieres and Steven P Reiss. 2003. Fault localization with nearest neighbor queries. In *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*. IEEE, 30–39.

[21] Iris Vessey. 1985. Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies* 23, 5 (1985), 459–494.

[22] Mark Weiser. 1981. Program slicing. In *Proceedings of the 5th international conference on Software engineering*. IEEE Press, 439–449.

[23] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 283–294.

[24] Andreas Zeller. 2002. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*. ACM, 1–10.

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