Received: 27 June 2023 | Revised: 2 January 2024 | Accepted: 29 May 2024

DOI: 10.1002/smr.2702

RESEARCH ARTICLE - EMPIRICAL

Journal of Software: Evolution and Process WILEY

Evaluation of the fixed-point iteration of minimizing delta debugging

Dániel Vince | Ákos Kiss

Department of Software Engineering, University of Szeged, Szeged, Hungary

Correspondence
Dániel Vince, Department of Software Engineering, University of Szeged, Szeged, Hungary.
Email: vinced@inf.u-szeged.hu

Funding information
National Research, Development and Innovation Fund, Grant/Award Numbers: TKP2021-NVA-09, ÚNKP-22-3-SZTE-469, ÚNKP-23-3-SZTE-536

Abstract

The minimizing Delta Debugging (DDMIN) was among the first algorithms designed to automate the task of reducing test cases. Its popularity is based on the characteristics that it works on any kind of input, without knowledge about the input structure. Several studies proved that smaller outputs can be produced faster with more advanced techniques (e.g., building a tree representation of the input and reducing that data structure); however, if the structure is unknown or changing frequently, maintaining the descriptors might not be resource-efficient. Therefore, in this paper, we focus on the evaluation of the novel fixed-point iteration of minimizing Delta Debugging (DDMIN*) on publicly available test suites related to software engineering. Our experiments show that DDMIN* can help reduce inputs further by 48.08% on average compared to DDMIN (using lines as the units of the reduction). Although the effectiveness of the algorithm improved, it comes with the cost of additional testing steps. This study shows how the characteristics of the input affect the results and when it pays off using DDMIN*.

KEYWORDS
Delta Debugging, evaluation, test case minimization

1 | INTRODUCTION

Software engineers create software to automate all kinds of tasks and help people all over the world, but some parts of software engineering itself can be automated as well. Testing-related tasks are among these: executing test suites, generating new test cases, locating the fault in the software, finding the minimal part of the test case that induced the fault, and so forth. The latter task, that is, automated test case minimization, has been the focus of researchers, especially since security-related fuzz testing became popular. The importance of test case reduction is highlighted by several communities dealing with bug reports. The guidelines for bug reporting in the Chromium projects emphasize reduced bug reports:

> If a bug can be reduced to a simplified test, then create a simplified test and attach it to the bug.[1]

LLVM developers also require test case minimization in their documentation:

> ...narrow down the bug so that the person who fixes it will be able to find the problem more easily.
> Once you have a reduced test-case, ...[2]

One of the first algorithms was the minimizing Delta Debugging (DDMIN) by Zeller and Hildebrandt.[3–5] It is already more than two decades old and is still used with preference since it works on any kind of input. DDMIN is the de facto standard for most of the reduction-related tasks

that everyone adjusts to their usage. Since the first appearance of DDMIN, there have been many approaches that tried to be more "clever" by using some information about the input. HDD,[6] Perses,[7] Vulcan,[8] ReduKtor,[9] and so forth algorithms are more effective than DDMIN; however, usually extra information is needed about the input structure, for example, a grammar. This need for extra information can act as an obstacle for some use cases: Grammar may not be available, and writing (or maintaining) it may not be a practical option; for example, for programming languages that are being rapidly developed, where the specification is changing frequently, the grammar may not be able to keep up with the changes. Thus, in such cases, the structure-unaware nature of the good old DDMIN comes in very handy.

This is the reason why we investigated whether it is possible to make DDMIN more effective and proposed to use the fixed-point iteration variant of the algorithm (DDMIN*) in our previous work.[10] DDMIN is proven to find a 1-minimal result, which is a local minimum; however, DDMIN* tries to find other potentially smaller 1-minimal results once DDMIN has completed its work. We evaluated the algorithm on a carefully crafted example and on a publicly available small test suite with character-level granularity; however, the study lacked the exhaustive evaluation and the cost-effect analysis. Therefore, in this study, we extend our previous paper and fill these gaps in order to unfold the true behavior of DDMIN*. Furthermore, we conduct further experiments with different granularities on differently structured test suites. The goal of this study is to answer the following research questions:

Research Questions

RQ1. Is the effectiveness[*] of DDMIN* similar at different granularities (i.e., character and line level, or with a combined usage)?
RQ2. Whether the behavior (both effectiveness and efficiency[†]) of the algorithm variants depends on the input structure or size?
RQ3. Can DDMIN* compete with more sophisticated techniques?

The rest of the paper is organized as follows: First, in Section 2, we give a brief overview of DDMIN and HDD to make this paper self-contained. Section 3 presents a motivational example where DDMIN can be improved upon and gives an overview of the fixed-point iteration variant of minimizing Delta Debugging. Then, in Section 4, we outline the experimental setup and present the used test suites and algorithm parameters. In Section 5, we evaluate the effects of DDMIN* and present our experimental results. In Section 6, we discuss related work, and finally, in Section 7, we conclude our paper.

2 | BACKGROUND

The minimizing Delta Debugging algorithm (DDMIN)[3–5] is a systematic iterative approach to reduce arbitrary input while keeping a predefined property invariant. The input is split into atomic units and represented as a set of them. First, this set of units is split into two roughly equal-sized subsets, and both parts are investigated whether they still have the predefined property of the initial input. If the property is kept in any of the subsets, then the reduction step is considered successful, and a new iteration starts with that subset; otherwise, the granularity is refined by doubling the splitting. The subsets of the new splitting are investigated, as well as their complements; that is, it is checked whether keeping or removing any of the subsets leads to a smaller yet interesting test case. Again, if any of the investigated subsets (or their complements) keep the property in question, it will be used as the input for the next iteration; otherwise, the granularity is increased. The iteration continues until the granularity reaches the unit level when it is proven to have found a so-called 1-minimal result.

DDMIN has its roots in the isolation of failure-inducing code changes. Therefore, an input is composed of elementary changes (or deltas), denoted as δ_i, and a set of elementary changes is also called a configuration, usually denoted by c. The outcome of a program execution on a specific configuration is determined by a testing function, and it can be either *fail* (also written as ×) if the current input produced the original, *pass* (also written as ✓) if the test succeeds, or *unresolved* (written as **?**) if the result is indeterminate. The initial configuration that triggers the failure is denoted by $c_{@cross@}$. For the sake of completeness, Figure 1 gives Zeller and Hildebrandt's latest formulation of the minimizing Delta Debugging algorithm.[5]

1-minimality: a local minimum where the removal of any single element would cause the loss of the interesting property. Removing two or more elements at once might result in an even smaller still interesting configuration; however, every single element on its own is significant in reproducing the failure.[5] As test case reduction is an NP-complete problem, finding the global minimum is practically infeasible, and searching for a 1-minimal result is an effective goal.

If an input that is to be minimized has some mandatory structure over its units, which is typical for inputs to a program, DDMIN may work suboptimally. The configuration partitioning during the iterations may be unaligned with the boundaries of the structural elements of the input,

Let $test$ and $c_{\text{✗}}$ be given such that $test(\emptyset) = \text{✓} \wedge test(c_{\text{✗}}) = \text{✗}$ hold.
The goal is to find $c'_{\text{✗}} = ddmin(c_{\text{✗}})$ such that $c'_{\text{✗}} \subseteq c_{\text{✗}}$, $test(c'_{\text{✗}}) = \text{✗}$, and $c'_{\text{✗}}$ is 1-minimal.
The *minimizing Delta Debugging algorithm* $ddmin(c)$ is

$$ddmin(c_{\text{✗}}) = ddmin_2(c_{\text{✗}}, 2) \text{ where}$$

$$ddmin_2(c'_{\text{✗}}, n) = \begin{cases} ddmin_2(\Delta_i, 2) & \text{if } \exists i \in \{1, \ldots, n\} \cdot test(\Delta_i) = \text{✗ (“reduce to subset”)} \\ ddmin_2(\nabla_i, \max(n-1, 2)) & \text{else if } \exists i \in \{1, \ldots, n\} \cdot test(\nabla_i) = \text{✗ (“reduce to complement”)} \\ ddmin_2(c'_{\text{✗}}, \min(|c'_{\text{✗}}|, 2n)) & \text{else if } n < |c'_{\text{✗}}| \text{ (“increase granularity”)} \\ c'_{\text{✗}} & \text{otherwise (“done”).} \end{cases}$$

where $\nabla_i = c'_{\text{✗}} - \Delta_i$, $c'_{\text{✗}} = \Delta_1 \cup \Delta_2 \cup \ldots \cup \Delta_n$, all Δ_i are pairwise disjoint, and $\forall \Delta_i \cdot |\Delta_i| \approx |c'_{\text{✗}}|/n$ holds.
The recursion invariant (and thus precondition) for $ddmin_2$ is $test(c'_{\text{✗}}) = \text{✗} \wedge n \leq |c'_{\text{✗}}|$.

FIGURE 1 The minimizing Delta Debugging algorithm.[5]

```
1  procedure HDD(input_tree)
2      level ← 0
3      nodes ← tagNodes(input_tree, level)
4      while nodes ≠ ∅ do
5          minconfig ← DDMIN(nodes)
6          prune(input_tree, level, minconfig)
7          level ← level + 1
8          nodes ← tagNodes(input_tree, level)
9      end while
10 end procedure
```

FIGURE 2 The hierarchical Delta Debugging algorithm.[6]

leading to incorrectly formatted, thus non-reproducing, and therefore, useless test cases. The goal of the hierarchical Delta Debugging (HDD) algorithm[6] is to avoid such superfluous steps by not testing format-breaking configurations. It works on hierarchical tree-structured input representations (e.g., on parse trees, abstract syntax trees, or XML DOM trees) and applies the minimizing Delta Debugging algorithm to the levels of that tree, progressing downwards from the root to the leaves. The pseudocode formulation of HDD as defined by Misherghi and Su[6] is shown in Figure 2. The auxiliary routine tagNodes collects the nodes at a given level of the tree, then DDMIN is invoked on those nodes, and finally, prune applies the result of Delta Debugging to the tree. That is, for HDD, configurations are sets of tree nodes at a given level, and the removal of a node causes the removal of the whole subtree rooted from that node. In a latter variant of HDD, “pruning” of a node has been reinterpreted as its replacement with the minimal applicable syntactically correct fragment to reduce the number of test attempts at incorrectly formatted configurations even further.[11] If HDD is iterated until a fixed point is reached, denoted as HDD*, it gives a 1-tree-minimal result; that is, if any node is removed from the tree, the newly serialized test case would not be interesting anymore.

3 | ITERATING DELTA DEBUGGING

The program in Figure 3A is a variant of a classic example of program slicing.[12] It computes both the sum and the product of the first 10 natural numbers in a single loop. Using slicing terminology, we can say that we want to compute the (so-called static backward) slice of this program with respect to the criterion (19, *prod*), thus creating a subprogram that does not contain statements that do not contribute to the value of *prod* at line 19. This can be computed either by analyzing the control and data dependencies of the program—which is the classic slicing way—or by following the approach of observation-based slicing[13] that performs a systematic removal of code parts based on trial and error, much like what DDMIN does on its input. Actually, even DDMIN can be applied to such tasks. The two things that have to be remembered are that in such reduction scenarios, the inputs or test cases are also programs, and the interesting properties to keep are not program failures (but it is still an × that represents that the property is kept). So, we reformulate the classic slicing example as a test case minimization task, where the program in Figure 3A is the input (the lines being the units) and the testing function is given as

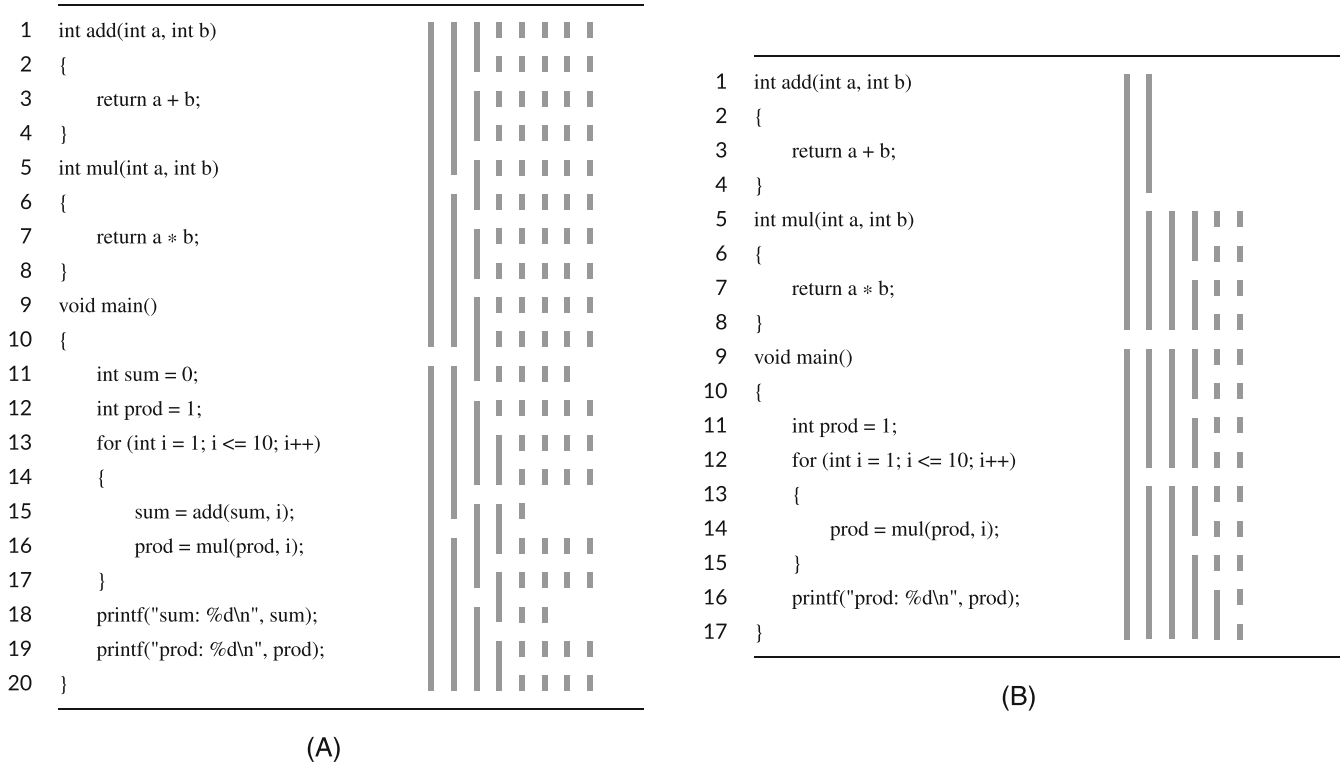


FIGURE 3 (A) Example C program that computes the sum and product of the first 10 natural numbers, and the execution of DDMIN on it while keeping 10! on the output. (B) The output of DDMIN on the program of (A) and the re-execution of DDMIN.

$$test(c) = \begin{cases} \checkmark & \text{if } c \text{ is syntactically incorrect} \\ \checkmark & \text{else if execution of } c \text{ does not terminate} \\ \checkmark & \text{else if execution of } c \text{ does not print prod : 3628800} \\ \times & \text{otherwise.} \end{cases}$$

The gray bars on the right of the program code show the progress of DDMIN, from left to right. Every set of vertically aligned bars corresponds to a configuration of the algorithm and shows how that configuration is split into subsets. This example shows that DDMIN could "slice away" the lines of the *main* function that did not contribute to the computation of *prod*. However, the algorithm could not remove the *add* function, because when the configuration contained no call to it anymore (at line 15), the granularity had already reached line (i.e., unit) level. But *add* could only be removed as a whole, not line-by-line, as removing any single line would cause syntax errors. (Note that this is one of the shortcomings of DDMIN that HDD and other grammar-based reducers wanted to fix.) So, DDMIN has produced a 1-minimal result (shown in Figure 3B), but it is clearly not a global minimum. What we can realize when looking at this result is that we could re-execute DDMIN on this program with the same testing function as the first time and we may be able to remove the superfluous *add* function as well. Again, the gray bars on the right of the program code show the progress of DDMIN, and indeed, the subsets of the second configuration aligned well with the structure of this input and made further reduction possible. The result of the second execution of DDMIN is given in Figure 4. This is the global optimum for this example, so further executions of DDMIN are not visualized.

Motivated by this example, we can formalize the intuition that DDMIN could be executed multiple times. Since it cannot be told a priori how many executions are needed for a given input, we propose to iterate DDMIN until a fixed point is reached. We will denote the fixed-point iteration of DDMIN as DDMIN*—following the notation used for HDD and HDD*[6]—and define it as follows:

$$ddmin^*(c_\times) = \begin{cases} c'_\times & \text{if } c_\times = c'_\times \\ ddmin^*(c'_\times) & \text{otherwise} \end{cases}$$
$$\text{where } c'_\times = ddmin(c_\times).$$

Although the asterisk notation is the same for the two algorithms and even its meaning is identical in both cases (i.e., fixed-point iteration), its purpose is fundamentally different for HDD and DDMIN. A single execution of HDD has no minimality guarantees, only HDD* produces 1-tree-

```
1  int mul(int a, int b)
2  {
3      return a * b;
4  }
5  void main()
6  {
7      int prod = 1;
8      for (int i = 1; i <= 10; i++)
9      {
10         prod = mul(prod, i);
11     }
12     printf("prod: %d\n", prod);
13 }
```

FIGURE 4 The output of DDMIN on the program of Figure 3B.

minimal results. However, even a single execution of DDMIN is guaranteed to give a 1-minimal result. The purpose of iterating it further is to find an even better 1-minimum. (Note that re-executing DDMIN does not guarantee better results in all cases, only if the configuration aligns well with the structure of the input.)

4 | EXPERIMENTAL SETUP

To evaluate the effects of DDMIN*, the existing prototype implementation has been used in the state-of-the-art Picire[‡] reducer. Picire is an open-source Python implementation of DDMIN that supports several configuration options, which were taken advantage of: the "reduce to subset" step was disabled, and complement tests were performed in backward syntactic order (i.e., the removal of the syntactically last subset of the test input was tested first since previous experiences have shown that the order in which the elements of a configuration are investigated can affect performance[14,15]), and the resource-efficient content-based caching was enabled.[16,17] Although the tool has options for parallel reduction, the experiments were conducted sequentially for the sake of reproducibility. With these settings, multiple executions yield the same results in terms of output and testing steps.

DDMIN can be impractically slow on huge input configurations (e.g., large input at character-level granularity), but fortunately, an extra option is available in Picire: a combined reduction pass that may achieve smaller outputs faster. The first pass splits the input into lines and uses them as a configuration. Line-based reduction is faster than character-based; however, it may produce larger results, as superfluous characters might be removed from the lines with finer granularity. The second pass then uses the output of the first, then continues the reduction at character-level granularity to achieve smaller results. We have not found a reference to this technique in the literature (and there is no theoretical guarantee that will be faster for all inputs); however, it is really useful from a practical perspective, and therefore, we have used it in our experiments. Picire is implemented to maximally utilize each reduction pass: The line-based reduction continues until the fixed point is reached, and then the character-based reduction does the same thing. Seemingly, this causes extra testing steps, but the experiments show that using this two-pass reduction is beneficial (see Section 5.1).

DDMIN may also give noticeably suboptimal results when the input has some well-defined structure over its units, which is quite typical for inputs for programs. During its iterations, DDMIN can separate the configuration in a way that is completely unaligned with the structure of the input, leading to incorrectly formatted, thus nonreproducible, therefore, completely unusable intermediate test cases. Several studies have made efforts to address this problem, one of the most well-known approaches is the Hierarchical Delta Debugging (HDD) algorithm. However, if DDMIN* forces the reduction further with the help of the fixed-point iteration, it raises the question *whether DDMIN* can compete with more sophisticated techniques (e.g., HDD*) in terms of effectiveness?* To examine this question, HDD* is included in the experiments using the Picireny[§] project, which is a hierarchical test case reduction framework on top of Picire, also written in Python, that supports ANTLR v4[¶] grammars and already contains an implementation of the HDD algorithm.

As inputs, test cases from different sources have been collected, all of which have already been used in the literature for benchmarking reduction. The first test suite is the Perses Test Suite[#] (PTS), which contains fuzzer-generated C sources that cause various internal compiler errors in the Clang and GCC compilers. The second set of tests is the JerryScript Reduction Test Suite[‖] (JRTS), which also contains fuzzer-generated

JavaScript files that cause failures in the JerryScript lightweight JavaScript engine. In the case of both test suites, the interesting property of test cases to keep during reduction is the failure that they induce. A typical test case contains two different elements:

- *input_file.{c, js}*: the input file to reduce, contains fuzzer-generated constructions in the appropriate programming language,
- *oracle*: a script that takes an input_file and decides whether it keeps the interesting property of the initial input or not.

The properties of the test cases are shown in Table A1 of Appendix A. As test cases are text files, there are two natural size metrics that can be used: the number of lines (column "Lines") and the number of characters or bytes (column "Size"). These are also the possible units of reduction (the δ_i of Section 3). However, to avoid bias from formatting (e.g., spaces or tabs used for indentation, or any whitespace in general), we also count non-whitespace characters as this value might reflect better what a human software engineer may be interested in (column "Chars"). To be able to feed test cases to HDD, they have to be processed to a tree structure. "Tree Height" represents the height of the parse tree built from the input, "Rules" shows the number of nonterminals, and "Tokens" shows the number of terminals in it. The parse tree representation of each test case was built using the grammar available for the input format from the official ANTLR v4 grammars repository.[**] Furthermore, Picireny has applied the flattening of the recursive structures[18] and the squeeze of the linear tree components[16] optimizations to the trees.

The experiments include two-pass DDMIN, DDMIN*, and the tree-based HDD* algorithm; therefore, a unified unit of measure must be used. Using the above-described non-whitespace characters as the "absolute" unit helps us to make a fair comparison between the different algorithms and their variants.

The workstation used to carry out the experiments was equipped with an Intel® Xeon® CPU E5-2680 v4 CPU clocked at 2.4 GHz and 128 GB RAM. The machine was running Ubuntu 22.04 with Linux kernel 5.15.0, and it was having no other load during the experiments.

5 | EXPERIMENT RESULTS

5.1 | RQ1: Effectiveness of DDMIN*

5.1.1 | Character-level granularity

Previously, the character-level reduction has been evaluated on JRTS,[10] where DDMIN* produced 67.94% smaller results than DDMIN on average. The cost of the improvement was the increase in the number of testing steps, which increased by 111.41% on average. It was also shown that if an input could be reduced somehow, then at least two iterations of DDMIN* are present: Whenever an iteration manages to reduce the configuration, there will be a next iteration that tries to continue the reduction (by definition of the fixed-point iteration). If the configuration cannot be reduced further in an iteration, the algorithm halts. Also, when the number of iterations is exactly two, that means that DDMIN* is not able to produce smaller results than DDMIN.

Figure 5 presents the character-level reduction process of *jerry-3479* over 15 iterations from the previous study. Changes in configuration size are represented along the vertical axis, from the input size to the end result of the last iteration. Test executions are represented along the horizontal axis, and each dashed vertical line represents the end of an iteration. The leftmost dashed line corresponds to the result of the first

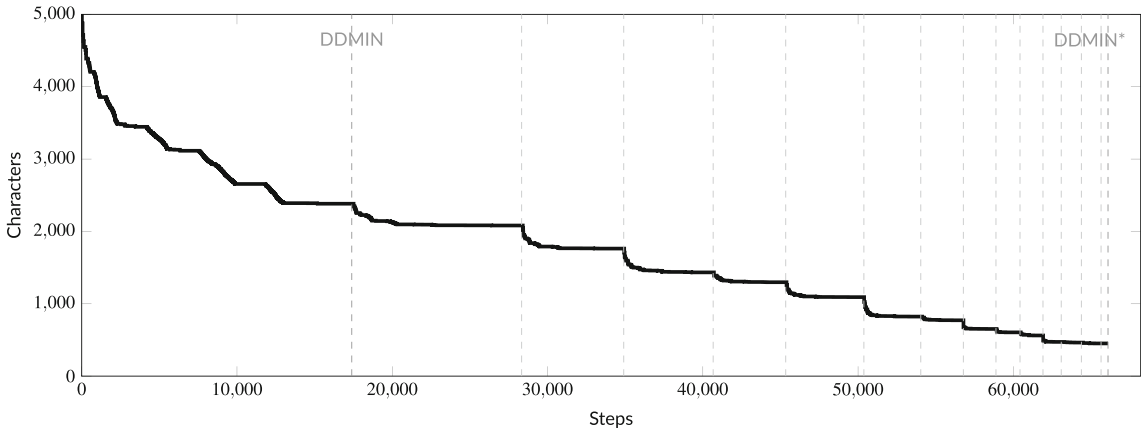


FIGURE 5 The process of reduction of the jerry-3479 test case with DDMIN* using character granularity through 15 iterations: the change of configuration size over testing steps.

iteration, which is also the result of DDMIN (labeled DDMIN). The following iterations yield gradual reductions until the 15th iteration cannot reduce its input further; therefore, the iteration halts (rightmost dashed line labeled with DDMIN*). Bigger parts from the configuration could be removed close to the beginning of each iteration (as the splitting of the configuration is reset to 2), and then only smaller chunks could be removed as the process progresses (flat parts of the figure). Then again, larger removals can be observed at the beginning of each additional DDMIN* iteration, which is responsible for the improvement. The first iteration is responsible for 26.31% of the steps, then the second for 16.55%, and the following iterations gradually perform fewer steps.

5.1.2 | Line-level granularity

Table A2 of Appendix A presents the results of the line-level reduction of the test cases of JRTS and PTS. The first group of values shows the baseline data, that is, those measured using DDMIN: The number of testing steps needed to accomplish the reduction, the amount of time needed to perform these steps on the workstation used for the experiment, the size of the output expressed in lines, and the number of non-whitespace characters in the output. (The number of testing steps includes actual test case evaluations only; i.e., does not include the re-evaluation of already seen configurations. As discussed in Section 4, content-based caching was used.) The second group of values contains data collected using DDMIN*. In addition to the absolute numbers, we also give the changes relative to the baseline data.

The algorithm behavior in terms of efficiency is similar on both test suites. Not surprisingly, DDMIN* required more testing steps than DDMIN, the average increase is 66.08%, with 519.21% being the maximum (*gcc-61917*). Compared to this, the increase in wall clock time is only 28.59% with 280.25% being the maximum (also *gcc-61917*). For most of the test cases, DDMIN* produced smaller results than DDMIN. Since the configuration units are *lines*, the results should be compared in this unit. The output configuration got smaller by a maximum of 96.45% (*clang-27747*), and the average improvement is 48.08%. Interestingly, two different behaviors can be observed: DDMIN* improves the effectiveness of the reduction to a great extent on PTS (68.70%), which contains 86 times larger inputs (in terms of lines) than JRTS on average. However, the improvement is only observable in 5 of 13 cases of JRTS, and the average improvement is only 19.53%. After a manual examination of the output files, it turned out that where DDMIN* produced the same output as DDMIN, that was the global optimum and could not be reduced further at line-level. For smaller configurations (in our experiments, 18–118 units) DDMIN *might* give a global optimum; however, if the input configuration was bigger (36–20,617 units), DDMIN* could be very helpful in creating smaller outputs. An overlap can be observed between the intervals, where relatively small configurations could also be reduced further with DDMIN*: but eventually, the behavior of the algorithm is test case specific, and we could not unambiguously generalize the phenomenon.

5.1.3 | Two-pass reduction

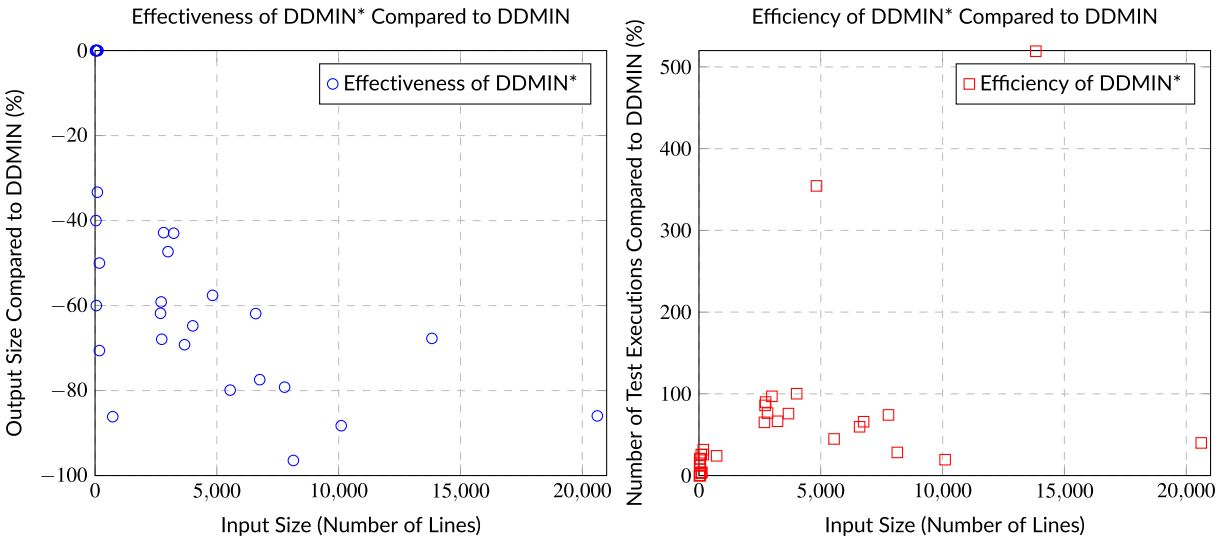
Not only the manual test case minimization can be time-consuming when the input is huge, but also DDMIN can take a lot of time. Although line-level reduction had reasonable time requirements, the character-level reduction was completely unacceptable for practical use in the case of PTS. However, the line-level reduction does not exploit the full potential of the algorithm, and there is an intermediate way, which might be "fast enough" and the output is still smaller. (E.g., Picire reduced the *gcc-71626* benchmark program to 4652 non-whitespace characters with DDMIN at line-level in 1437 steps; furthermore, at character level, it could be reduced to 8713 non-whitespace characters in *121,968* steps. Using more resources and the result is even worse, which is not the best combination.) Therefore, as discussed in Section 4, a combined reduction pass has been utilized that first reduces the input with line granularity and then reduces further with character-level granularity. The reduction time became more acceptable using this technique as the line-level granularity.

Table A3 of Appendix A shows the results of this two-pass reduction with DDMIN and DDMIN*. The structure is similar to Table A2; only the main basis of comparison is changed to non-whitespace characters (column "Chars") to avoid misinterpretation of the results. The average increase in the testing steps is 96.41%, with 615.33% being the maximum (*clang-23309*), which means that a maximum of seven times more testing steps are needed to reduce tests from our suite. The increase in wall clock time is 68.91% with 711.25% being the maximum (*clang-23309*). Compared to the line-level experiment, DDMIN* produced smaller results in 8 of 13 (+3) on JRTS, meaning the line-level global minimum could be reduced further with finer granularity. The output got smaller by a maximum of 88.27% (*clang-27747*), and the average improvement is 45.76%. Based on the two-pass reduction, it is definitely worth using DDMIN* if the goal is to produce as small outputs as possible with the least information about the input structure (i.e., lines and characters only), since the size of the output halved on average. Results of Table A3 can be interpreted in a way that results from Table A2 were given to Picire to further reduce it with character-level DDMIN*. Comparing the results from the two tables, the average improvement is *53.96%*, but it has its price in the increased time needed to accomplish the reduction: 15 times more wall-clock time is needed on average. (This might sound a lot, however, the slowest reduction in JRTS needed 6 min, which can be made even more acceptable with parallel execution in a fuzzer ecosystem.)

5.2 | RQ2: Input-dependent behavior of algorithm variants

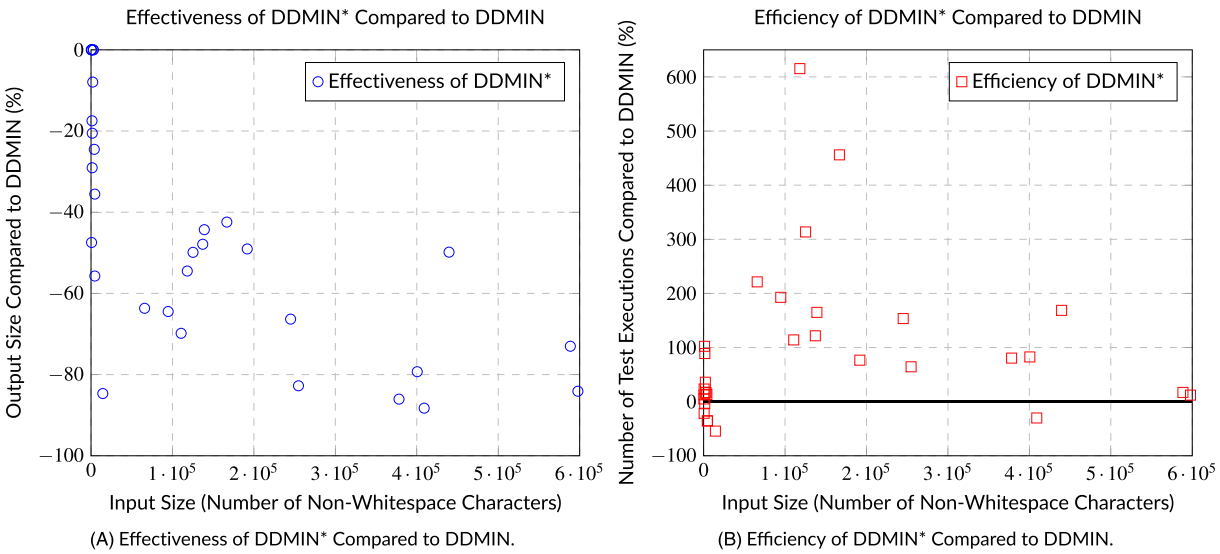
Figure 6 visualizes the raw data from Table A2. Each mark on the charts represents a test case reduced with DDMIN*, and the position of the mark reflects how the fixed-point iteration affected the reduction compared to DDMIN. The input configuration size is represented along the horizontal axis, while the effects of DDMIN* are represented along the vertical axis (all relative to the size and number of testing steps of the baseline). Figure 6A corresponds to the effectiveness, and it shows that DDMIN* produced exactly the same output as DDMIN in some cases; however, the outputs are the fraction of the baseline for the majority of benchmarks. Figure 6B shows the effect on the efficiency of the reduction. DDMIN* yielded the smaller outputs slower (as expected), the additional cost in the number of testing steps is not expensive for inputs with configuration size less than 1000 (*13.28%*), and the effort is doubled for larger inputs (*109.56%* on average including the outliers, and *66.08%* when they are excluded). There are only a handful of outliers where DDMIN* required much more testing steps than the baseline, namely *gcc-61383* and *gcc-61917*.

Figure 7 visualizes the raw data from Table A3. Its structure is similar to Figure 6, and the only difference is the metrics of the configuration size represented along the horizontal axis: The number of lines has been replaced by the number of non-whitespace characters. Figure 7A shows similar results as Figure 6A, DDMIN* produces smaller outputs for the vast majority of benchmarks, and some small inputs cannot be reduced



(A) Effectiveness of DDMIN* Compared to DDMIN. (B) Efficiency of DDMIN* Compared to DDMIN.

FIGURE 6 Effect of line-level DDMIN* as a function of the *Size* of the input.



(A) Effectiveness of DDMIN* Compared to DDMIN. (B) Efficiency of DDMIN* Compared to DDMIN.

FIGURE 7 Effect of two-pass DDMIN as a function of the *Size* of the input.

further with fixed-point iteration. Figure 7B shows surprising results: some inputs could be reduced faster. The reason behind the efficiency improvement is that Picire can produce smaller results at line level in a reasonable amount of testing steps with DDMIN*, and then the character-level reduction can work further starting from this smaller input configuration. However, the general case is that DDMIN* requires more testing steps; furthermore, unlike the line-level reduction, the increase did not only double but quadrupled on average with two-pass reduction. There are two, but different outliers where DDMIN* required much more testing steps, namely, *clang-23309* and *gcc-59903*.

5.3 | RQ3: Relation to more sophisticated techniques

Seeing these promising results, the question can be raised whether DDMIN* can compete with HDD* in terms of output size. There is no doubt that neither DDMIN nor DDMIN* can compete with HDD* in terms of the number of required testing steps. HDD takes the input structure into account, which makes it highly efficient compared to structure-unaware algorithms. The answer can be found in Table A4 of Appendix A. While the output of traditional DDMIN is 720% larger than the output of HDD* on average (in these test suites), DDMIN* brings the results much closer. The output of DDMIN* is 139% larger than the output of HDD*, which is not that bad for a structure-unaware technique. However, DDMIN* is still far away from being a competitor to HDD*, and should be optimized further.

5.4 | Threats to validity

The following threats were identified in this study, and the following actions were considered to avoid or minimize their impacts.

5.4.1 | Selection of benchmarks

Two suites of benchmarks were used, one in C and one in JavaScript programming language, the Perses Test Suite (PTS) and JerryScript Reduction Test Suite (JRTS), respectively. As the formats of the test cases are similarly structured and come from the same domain of programming languages, the findings may not generalize to all types of test cases. However, we believe that the results of these test suites are indicative since they contain real-world test cases and have been used in reduction-related studies.[7,19,20] By comparing their sizes, JRTS contains fewer test cases, which are also smaller in size than PTS, threatening the unbiased presentation of the results, as the effectiveness of a reduction algorithm might depend on the size of the input. We examined the relative effects of the algorithm to avoid the misinterpretation of the results; furthermore, where the results were different, we discussed them in detail.

5.4.2 | Correctness of implementation

In order to ensure that the implementation of the experiments is correct and accurate, we conducted a code review. On selected C and JavaScript examples, we traced the behavior of the implementation to validate that it works as intended. Furthermore, the implementation is based on open-source and well-maintained repositories such as the Picire and Picireny frameworks that have been used in several studies[10,14–18,21–24] and ANTLR v4.

6 | RELATED WORK

One of the first works on automated test case reduction is Delta Debugging by Zeller and Hildebrandt, minimizing inputs of arbitrary format.[3–5] Gharachorlu and Sumner[25] observed that after a successful "reduce to complement" step, DDMIN revisits the previously investigated subsets, and these steps might be inefficient in practice. Therefore, they proposed a modified version of DDMIN, *One Pass Delta Debugging (OPDD)*, which skips these steps and shows that if certain circumstances are met, it can also achieve a 1-minimal result. Their goal was to achieve linear time complexity, and they identified three independent conditions that can eliminate the need for revisiting: common dependence order, unambiguity, and deferred removal. Hodován and Kiss proposed using DDMIN in a parallel way that can reduce the time required to perform the minimization.[14] They observed that parallel DDMIN can yield different-sized 1-minimal outputs as a result of independent parallel executions. Furthermore, they also observed that the "reduce to subset" step is not even necessary for 1-minimality and they reordered the "reduce to complement" steps. Kiss proved[20] that if the split factor of DDMIN is well chosen (2 was used in the original author's work), then the reduction can be sped up significantly.

Artho investigated Delta Debugging in his "Iterative Delta Debugging" study.[26] Although the title of the study and the methodology discussed in this study are similar, the studies are not related. It used the Delta Debugging algorithm (not DDMIN) to find the failure-inducing

changes in the version history. He raised the issue that DD is only applicable if the version that *passes* a test is known, which may not be the case for newly discovered defects. Therefore, he proposed the iterative DD, called IDD, that successively backports fixes to earlier defects, and one may eventually obtain a version that is capable of executing the test in question correctly.

Most of the published works target textual inputs; however, test case reduction can be applied to other scenarios as well. Several authors have minimized faulty event sequences originating from various sources: Scott et al[27] minimized event sequences of distributed systems, Bársony[24] reduced OpenGL API traces, and Clapp et al[28] aimed at Android GUI event sequences with a variant of DDMIN. Kalhauge and Palsberg[29,30] modeled the dependencies of Java bytecode via propositional logic and reduced them with their Generalized Binary Reduction algorithm. Furthermore, Brummayer and Biere[31] and Kremer et al[32] even used DDMIN to minimize SMT solver formulas.

The cost of the generality of DDMIN is the lowered performance on inputs that have strict formatting rules, because many format-breaking incorrect test cases are generated and evaluated during the reduction process. To overcome this problem, Miserghi and Su proposed using information about the input format encoded in grammars, that is, converting test cases into a tree representation,[6] and applying delta debugging to the levels of the tree, called hierarchical Delta Debugging. This approach helped remove parts of the input that aligned with its syntactic unit boundaries. As a further improvement, they proposed the fixed-point iteration of HDD, denoted as HDD*: HDD is repeatedly applied until it fails to remove further elements from the tree producing a 1-tree-minimal result.

Binkley et al[13,33–37] recognized an analogy between test case reduction and program slicing. The concepts of program slicing (the slicing criterion) could be reformulated as concepts of reduction (interestingness property); therefore, their proposed approach avoids building dependence graphs for a program and can work at the syntactic level only. They have also recognized that a single iteration of their Observation-Based Slicing (ORBS) algorithm does not guarantee 1-minimality, since certain lines become removable only after other lines have been deleted. Therefore, they iterated the body of the algorithm as long as the previous iteration deleted some lines from the input. The approach is similar to DDMIN*; however, they have not tried to start the algorithm over to find another, more optimal, 1-minimal result.

7 | CONCLUSIONS

In this work, we have evaluated the fixed-point iteration of minimizing Delta Debugging (DDMIN*) in two slightly different settings. The test suites used are publicly available and have already been used in reduction-related studies. First, the reduction of test cases was performed with line granularity, and the experiments show that DDMIN* can produce 48.08% smaller outputs on average (68.70% on Perses Test Suite and 19.53% on JerryScript Reduction Test Suite). The price of this improvement is the increased number of steps, which was 66.08% on average.

Then, a "combined" two-pass reduction was performed where test cases were first reduced with line granularity, then these intermediate results were reduced further with character granularity as fine-tuning. DDMIN* overcome DDMIN with this setting as well and could reduce inputs further by 45.76% on average. Surprisingly some inputs could be reduced faster with DDMIN* as the line-level reduction produces results in a reasonable amount of steps, and then the character-level reduction can work further from this smaller input configuration.

If grammar is not available or maintaining it is not a beneficial option, we would recommend using the combined two-pass DDMIN*, since it was shown that the fixed-point iteration results in smaller outputs in exchange for some extra CPU cycles. We also recommend using Picire,[††] which contains our reference implementation and is ready to use out of the box from PyPI[‡‡] via Python's pip package manager.

Encouraged by the promising results, we have compared the output of DDMIN* to the output of HDD*, to see whether a structure-unaware algorithm can compete with a "more clever" one. In terms of required testing steps, the answer is simply no; however, in terms of size, DDMIN* brought the results much closer to each other, from 9 times larger outputs (DDMIN) to only 3 times larger ones. There is still room for improvement in those situations where the structure information is missing or changing rapidly. As future work, we wish to conduct further experiments to make DDMIN* more "clever" without the need for grammar, guiding the reduction to which units should be removed first.

Based on the experimental data and observations above, we can conclude the contributions of this study and answer the research questions:

Answer to Research Question #1

Is the effectiveness of DDMIN similar at different granularities (i.e., character and line level, or with a combined usage)?*

DDMIN* is most effective with character-level granularity based on Vince[10] and our experimental results (67.94% smaller outputs) compared to line-level reduction (48.08% smaller outputs). However, character-level reduction can be unacceptably slow for "large" inputs and line-level reduction leaves unnecessary parts in its output; therefore, we used a combined approach, where DDMIN* produced 45.76% smaller outputs. Furthermore, the two-pass reduction produced 53.96% smaller outputs than the line-level approach in our experiments, on average.

Answer to Research Question #2

Whether the behavior of the algorithm variants depends on the input structure or size?

DDMIN* incurs an additional cost (number of testing steps), which appears in most cases. (Some tests from our experimental setup could be reduced with fewer steps, but these are exceptions.) This additional cost is related to the size of the test case, but it does not grow beyond all limits. We identified that a maximum of seven times more testing steps are needed with DDMIN* in the used benchmark suites.

The effectiveness of the reduction shows similar patterns: the larger the input configuration, the larger the potential to reduce. There were some small test cases (*JRTS*) where DDMIN* could not reduce the input further; however, this cannot be completely generalized. If the input configuration has some superfluous items, DDMIN* can reduce it further regardless of its size.

Answer to Research Question #3

Can DDMIN compete with more sophisticated techniques?*

Encouraged by the promising results, we have compared the output of DDMIN* to the output of HDD, whether a structure-unaware algorithm can compete with a "more clever." In terms of required testing steps, the answer is simply no; however, DDMIN* brought the results much closer to each other, from 9 times larger outputs (DDMIN) to only 3 times larger ones.

ACKNOWLEDGMENTS

This research was supported by project TKP2021-NVA-09, implemented with the support provided by the Ministry of Innovation and Technology of Hungary from the National Research, Development and Innovation Fund, financed under the TKP2021-NVA funding scheme. Dániel Vince was supported by the ÚNKP-22-3-SZTE-469 and ÚNKP-23-3-SZTE-536 New National Excellence Program of the Ministry for Culture and Innovation from the source of the National Research, Development and Innovation Fund.

DATA AVAILABILITY STATEMENT

The source code of the implemented algorithms is openly available in the Picire project at https://github.com/renatahodovan/picire. Input data to the implemented algorithms are openly available in the JerryScript Reduction Test Suite project at https://github.com/vincedani/jrts and in the Perses project at https://github.com/choderalab/perses/. Raw execution log data of the implemented algorithms are available from the corresponding author upon request.

ORCID

Dániel Vince https://orcid.org/0000-0002-8701-5373

ENDNOTES

* By effectiveness, we mean the effect on the size of reduced test cases.

† By efficiency, we mean the effect on the number of testing steps.

‡ https://github.com/renatahodovan/picire

§ https://github.com/renatahodovan/picireny

¶ https://github.com/antlr/antlr4

https://github.com/uw-pluverse/perses

‖ https://github.com/vincedani/jrts

** https://github.com/antlr/grammars-v4

†† https://github.com/renatahodovan/picire

‡‡ https://pypi.org

REFERENCES

1. Chromium: Bug Life Cycle and Reporting Guidelines. Accessed October 4, 2023. https://www.chromium.org/for-testers/bug-reporting-guidelines
2. LLVM: How to submit an LLVM bug report. Accessed October 4, 2023. https://llvm.org/docs/HowToSubmitABug.html
3. Zeller A. Yesterday, my program worked. Today, it does not. Why? In: Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE '99), Lecture Notes in Computer Science, vol. 1687. Springer; 1999:253-267.

4. Hildebrandt R, Zeller A. Simplifying failure-inducing input. In: Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '00). ACM; 2000:135-145.
5. Zeller A, Hildebrandt R. Simplifying and isolating failure-inducing input. *IEEE Trans Softw Eng*. 2002;28(2):183-200.
6. Misherghi G, Su Z. HDD: Hierarchical delta debugging. In: Proceedings of the 28th International Conference on Software Engineering (ICSE '06). ACM; 2006:142-151.
7. Sun C, Li Y, Zhang Q, Gu T, Su Z. Perses: Syntax-guided program reduction. In: Proceedings of the 40th ACM/IEEE International Conference on Software Engineering (ICSE '18). ACM; 2018:361-371.
8. Xu Z, Tian Y, Zhang M, Zhao G, Jiang Y, Sun C. Pushing the limit of 1-minimality of language-agnostic program reduction. *Proc ACM Program Lang*. 2023;7(OOPSLA1):636-664.
9. Stepanov D, Akhin M, Belyaev M. ReduKtor: How we stopped worrying about bugs in Kotlin compiler. In: Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE 2019). IEEE; 2019:317-326.
10. Vince D. Iterating the minimizing delta debugging algorithm. In: Proceedings of the 13th International Workshop on Automating Test Case Design, Selection and Evaluation (A-Test 2022). ACM; 2022:00-04.
11. Misherghi GS. Hierarchical delta debugging. *Master's Thesis*: University of California, Davis. California; 2007.
12. Tip F. A survey of program slicing techniques. *J Programm Lang*. 1995;3(3):121-189.
13. Binkley D, Gold N, Harman M, Islam S, Krinke J, Yoo S. ORBS: Language-independent program slicing. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014). ACM; 2014:109-120.
14. Hodován R, Kiss A. Practical improvements to the minimizing delta debugging algorithm. In: Proceedings of the 11th International Joint Conference on Software Technologies (ICSOFT 2016) – Volume 1: ICSOFT-EA. SciTePress; 2016:241-248.
15. Kiss A, Hodován R, Gyimóthy T. HDDr: A recursive variant of the hierarchical delta debugging algorithm. In: Proceedings of the 9th ACM SIGSOFT International Workshop on Automating Test Case Design, Selection, and Evaluation (A-TEST 2018). ACM; 2018:16-22.
16. Hodován R, Kiss A, Gyimóthy T. Tree preprocessing and test outcome caching for efficient hierarchical delta debugging. In: Proceedings of the 12th IEEE/ACM International Workshop on Automation of Software Testing (AST 2017). IEEE; 2017:23-29.
17. Vince D, Kiss A. Cache optimizations for test case reduction. In: Proceedings of the 22nd IEEE International Conference on Software Quality, Reliability, and Security (QRS 2022). IEEE; 2022:442-453.
18. Hodován R, Kiss A, Gyimóthy T. Coarse hierarchical delta debugging. In: Proceedings of the 33rd IEEE International Conference on Software Maintenance and Evolution (ICSME 2017). IEEE; 2017:194-203.
19. Gharachorlu G, Sumner N. PARDIS: Priority aware test case reduction. In: Proceedings of the 22nd International Conference on Fundamental Approaches to Software Engineering (FASE 2019), Lecture Notes in Computer Science, vol. 11424. Springer; 2019:409-426.
20. Kiss A. Generalizing the split factor of the minimizing delta debugging algorithm. *IEEE Access*. 2020;8:219837-219846.
21. Vince D, Hodován R, Bársony D, Kiss A. Extending hierarchical delta debugging with hoisting. In: Proceedings of the 2nd ACM/IEEE International Conference on Automation of Software Test (AST 2021). IEEE; 2021:60-69.
22. Vince D, Hodován R, Kiss A. Reduction-assisted fault localization: Don't throw away the by-products! In: Proceedings of the 16th International Conference on Software Technologies (ICSOFT 2021). SciTePress; 2021:196-206.
23. Vince D, Hodován R, Bársony D, Kiss A. The effect of hoisting on variants of hierarchical delta debugging. *J Softw: Evol Process*. 2022;34(11):e2483:1-e2483:26.
24. Bársony D. OpenGL API call trace reduction with the minimizing delta debugging algorithm. In: Proceedings of the 13th International Workshop on Automating Test Case Design, Selection and Evaluation (A-Test'22). ACM; 2022:53-56.
25. Gharachorlu G, Sumner N. Avoiding the familiar to speed up test case reduction. In: 2018 IEEE International Conference on Software Quality, Reliability and Security (QRS). IEEE; 2018:426-437.
26. Artho C. Iterative delta debugging. *Hardware and software: Verification and testing*, Lecture Notes in Computer Science, vol. 5394: Springer; 2009: 99-113.
27. Scott C, Brajkovic V, Necula G, Krishnamurthy A, Shenker S. Minimizing faulty executions of distributed systems. In: Proceedings of the 13th USENIX symposium on networked systems design and implementation (NSDI '16). USENIX Association; 2016:291-309.
28. Clapp L, Bastani O, Anand S, Aiken A. Minimizing GUI event traces. In: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016). ACM; 2016:422-434.
29. Kalhauge CG, Palsberg J. Binary reduction of dependency graphs. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2019. Association for Computing Machinery; 2019; New York, NY, USA:556-566.
30. Kalhauge CG, Palsberg J. Logical bytecode reduction. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2021. Association for Computing Machinery; 2021; New York, NY, USA:1003-1016.
31. Brummayer R, Biere A. Fuzzing and delta-debugging SMT solvers. In: Proceedings of the 7th International Workshop on Satisfiability Modulo Theories (SMT '09). ACM; 2009:1-5.
32. Kremer G, Niemetz A, Preiner M. DDSMT 2.0: Better delta debugging for the SMT-LIBV2 language and friends. In: Silva A, Leino KRM, eds. *Computer aided verification*: Springer International Publishing; 2021:231-242.
33. Binkley D, Gold N, Harman M, Islam S, Krinke J, Yoo S. ORBS and the limits of static slicing. In: Proceedings of the 15th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2015). IEEE; 2015:1-10.
34. Yoo S, Binkley D, Eastman R. Seeing is slicing: Observation based slicing of picture description languages. In: Proceedings of the 14th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2014). IEEE; 2014:175-184.
35. Gold NE, Binkley D, Harman M, Islam S, Krinke J, Yoo S. Generalized observational slicing for tree-represented modelling languages. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017). ACM; 2017:547-558.
36. Binkley D, Gold N, Islam S, Krinke J, Yoo S. Tree-oriented vs. line-oriented observation-based slicing. In: Proceedings of the 17th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2017). IEEE; 2017:21-30.
37. Binkley D, Gold N, Islam S, Krinke J, Yoo S. A comparison of tree- and line-oriented observational slicing. *Empir Softw Eng*. 2019;24(5):3077-3113.

How to cite this article: Vince D, Kiss Á. Evaluation of the fixed-point iteration of minimizing delta debugging. *J Softw Evol Proc*. 2024; 36(10):e2702. https://doi.org/10.1002/smr.2702

APPENDIX A: EXPERIMENTAL DATA

In the tables below, characters serve as an absolute measure (column "Chars"), and it is expressed as the number of non-whitespace characters to avoid bias from indentation or other formatting differences.

TABLE A1 Properties of the inputs used for benchmarking.

Test	Size	Chars	Lines	Tree Height	Rules	Tokens
jerry-3299	1767	1208	54	33	608	140
jerry-3361	1953	1520	43	28	562	163
jerry-3376	6626	4647	178	36	2194	473
jerry-3408	2681	2100	44	28	778	228
jerry-3431	1065	648	36	30	527	130
jerry-3433	961	652	36	24	378	82
jerry-3437	6597	4623	178	36	2188	471
jerry-3479	5201	3998	95	25	1326	347
jerry-3483	492	326	18	19	193	48
jerry-3506	3760	2735	100	28	1278	343
jerry-3523	3928	2802	118	28	1416	345
jerry-3534	1927	1409	53	28	641	176
jerry-3536	829	592	27	23	310	71
clang-22382	80,210	65,786	2993	242	29,344	6573
clang-22704	723,495	597,827	20,617	272	255,972	61,255
clang-23309	147,879	118,178	2815	288	52,183	11,570
clang-23353	134,381	94,734	4011	185	44,100	9989
clang-25900	328,729	245,065	5546	292	106,751	23,406
clang-26350	467,008	378,160	6759	304	168,324	25,790
clang-26760	793,470	588,548	10,104	340	288,964	60,762
clang-27747	541,699	409,083	8141	265	238,604	46,295
clang-31259	179,380	137,161	2736	331	66,291	14,590
gcc-59903	217,161	166,754	3225	298	76,531	17,322
gcc-61383	142,054	110,643	4824	303	46,786	9070
gcc-61917	343,503	254,742	13,827	254	115,834	24,508
gcc-64990	554,312	439,587	6593	342	200,107	45,000
gcc-65383	158,731	125,221	2687	254	58,846	13,237
gcc-66186	177,924	139,087	2713	258	65,228	14,434
gcc-66375	248,824	191,827	3674	282	86,512	19,216
gcc-70127	540,224	400,556	7780	293	210,039	44,942
gcc-71626	18,975	14,465	724	20	8044	2047

TABLE A2 Results with line granularity.

	DDMIN				DDMIN*							
Test Case	Steps	Time (s)	Lines	Chars	Steps		Time (s)		Lines		Chars	
jerry-3299	54	4.33	13	307	65	+20.37%	5.24	+21.02%	13	–	307	–
jerry-3361	28	3.87	4	165	29	+3.57%	3.88	+0.26%	4	–	165	–
jerry-3376	144	15.22	17	361	181	+25.69%	18.14	+19.19%	5	−70.59%	113	−68.70%
jerry-3408	22	1.74	3	165	22	–	1.81	+4.02%	3	–	165	–
jerry-3431	29	3.24	5	104	35	+20.69%	3.89	+20.06%	3	−40.00%	56	−46.15%
jerry-3433	23	3.15	2	57	23	–	3.29	+4.44%	2	–	57	–
jerry-3437	186	20.27	20	497	245	+31.72%	24.93	+22.99%	10	−50.00%	252	−49.30%
jerry-3479	148	16.72	15	515	186	+25.68%	19.30	+15.43%	10	−33.33%	364	−29.32%
jerry-3483	13	1.90	2	56	13	–	1.90	–	2	–	56	–
jerry-3506	33	4.21	3	93	34	+3.03%	4.33	+2.85%	3	–	93	–
jerry-3523	49	6.39	4	90	51	+4.08%	6.67	+4.38%	4	–	90	–
jerry-3534	85	8.95	10	223	95	+11.76%	10.17	+13.63%	4	−60.00%	121	−45.74%
jerry-3536	33	3.22	7	158	38	+15.15%	3.66	+13.66%	7	–	158	–
clang-22382	4936	1514.18	596	14,705	9722	+96.96%	2032.92	+34.26%	314	−47.32%	10,451	−28.93%
clang-22704	11,930	5957.04	919	9572	16,695	+39.94%	6541.71	+9.81%	129	−85.96%	1748	−81.74%
clang-23309	8736	4439.29	803	23,616	15,414	+76.44%	5975.74	+34.61%	459	−42.84%	20,048	−15.11%
clang-23353	6117	2127.34	670	11,337	12,243	+100.15%	2769.17	+30.17%	236	−64.78%	5147	−54.60%
clang-25900	13,086	7662.78	1000	13,031	18,948	+44.80%	8496.11	+10.88%	201	−79.90%	5783	−55.62%
clang-26350	21,027	20,445.25	1485	63,740	34,880	+65.88%	23,424.97	+14.57%	335	−77.44%	16,673	−73.84%
clang-26760	35,647	25,824.05	1782	17,170	42,565	+19.41%	22,886.70	−11.37%	209	−88.27%	6755	−60.66%
clang-27747	17,309	15,970.98	1407	16,825	22,208	+28.30%	16,837.55	+5.43%	50	−96.45%	1905	−88.68%
clang-31259	9172	5510.85	764	15,395	17,421	+89.94%	6951.33	+26.14%	245	−67.93%	9742	−36.72%
gcc-59903	8677	7453.63	805	15,900	14,440	+66.42%	8504.36	+14.10%	459	−42.98%	13,823	−13.06%
gcc-61383	21,857	7391.74	2847	45,370	99,322	+354.42%	22,975.37	+210.82%	1207	−57.60%	16,685	−63.22%
gcc-61917	51,876	26,518.21	6975	92,919	321,223	+519.21%	100,834.93	+280.25%	2251	−67.73%	27,094	−70.84%
gcc-64990	14,500	15,754.00	1173	21,002	23,157	+59.70%	17,466.87	+10.87%	447	−61.89%	17,528	−16.54%
gcc-65383	8536	4106.35	715	16,034	14,093	+65.10%	4932.27	+20.11%	273	−61.82%	11,297	−29.54%
gcc-66186	9326	9297.62	803	17,442	17,334	+85.87%	10,373.58	+11.57%	328	−59.15%	11,837	−32.14%
gcc-66375	11,352	11,476.84	1062	18,363	19,958	+75.81%	13,354.05	+16.36%	327	−69.21%	11,219	−38.90%
gcc-70127	20,547	17,735.92	1620	21,367	35,790	+74.19%	19,839.98	+11.86%	337	−79.20%	9183	−57.02%
gcc-71626	1437	161.77	130	4652	1784	+24.15%	184.27	+13.91%	18	−86.15%	611	−86.87%

TABLE A3 Results with combined granularity.

	DDMIN			DDMIN*					
Test Case	Steps	Time (s)	Chars	Steps		Time (s)		Chars	
jerry-3299	1092	81.94	143	2209	+102.29%	165.22	+101.64%	118	−17.48%
jerry-3361	625	54.56	112	1181	+88.96%	95.33	+74.73%	89	−20.54%
jerry-3376	1549	137.97	201	993	−35.89%	82.94	−39.89%	89	−55.72%
jerry-3408	423	31.97	63	575	+35.93%	42.12	+31.75%	58	−7.94%
jerry-3431	456	44.45	59	358	−21.49%	32.61	−26.64%	31	−47.46%
jerry-3433	153	16.57	21	172	+12.42%	17.37	+4.83%	21	–
jerry-3437	1837	167.81	166	1189	−35.27%	108.42	−35.39%	107	−35.54%
jerry-3479	1875	162.08	253	2124	+13.28%	174.25	+7.51%	191	−24.51%
jerry-3483	40	5.30	4	42	+5.00%	5.20	−1.89%	4	–
jerry-3506	282	328.47	48	329	+16.67%	386.24	+17.59%	48	–
jerry-3523	377	35.52	63	440	+16.71%	39.69	+11.74%	63	–
jerry-3534	1079	92.67	148	1037	−3.89%	84.73	−8.57%	105	−29.05%
jerry-3536	683	56.08	148	842	+23.28%	67.73	+20.77%	148	–
clang-22382	97,305	7018	7431	312,664	+221.32%	64,387.15	+153.26%	2700	−63.67%
clang-22704	48,929	5493	4365	54,819	+12.04%	10,341.69	−19.90%	694	−84.10%
clang-23309	128,663	9971	11,736	920,361	+615.33%	268,369.71	+711.25%	5341	−54.49%
clang-23353	69,955	5041	7352	204,809	+192.77%	35,926.18	+144.34%	2612	−64.47%
clang-25900	107,132	10,842	7647	271,594	+153.51%	57,745.12	+66.18%	2574	−66.34%
clang-26350	571,649	66,923	47,655	1,031,666	+80.47%	275,089.75	−11.17%	6646	−86.05%
clang-26760	175,770	19,461	9456	205,756	+17.06%	54,985.43	−4.77%	2551	−73.02%
clang-27747	101,777	14,389	8988	71,072	−30.17%	23,191.19	−42.01%	1054	−88.27%
clang-31259	97,311	10,840	8156	215,736	+121.70%	54,647.29	+83.45%	4250	−47.89%
gcc-59903	108,168	11,997	10,080	601,419	+456.00%	145,742.81	+355.33%	5803	−42.43%
gcc-61383	374,540	49,046	32,544	801,846	+114.09%	243,083.31	+34.40%	9817	−69.83%
gcc-61917	1,025,065	174,767	65,357	1,685,350	+64.41%	576,466.30	−20.89%	11,268	−82.76%
gcc-64990	294,382	16,967	10,051	791,075	+168.72%	224,668.02	+145.05%	5042	−49.84%
gcc-65383	106,058	9438	7593	438,661	+313.60%	105,516.61	+272.20%	3805	−49.89%
gcc-66186	95,930	11,196	9236	254,214	+165.00%	76,236.37	+124.48%	5144	−44.30%
gcc-66375	110,042	14,186	9680	194,366	+76.63%	51,084.57	+35.09%	4932	−49.05%
gcc-70127	149,023	21,531	12,062	272,148	+82.62%	62,918.22	+11.66%	2498	−79.29%
gcc-71626	17,879	1172	1644	8145	−54.44%	568.14	−59.84%	252	−84.67%

TABLE A4 Results with HDD* and combined granularity DDMIN*.

	HDD*		DDMIN*						
Test Case	Steps	Time (s)	Chars	Steps		Time (s)		Chars	
jerry-3299	171	17.80	92	2209	+1191.81%	165.22	+828.20%	118	+28.26%
jerry-3361	141	12.40	97	1181	+737.59%	95.33	+668.79%	89	−8.25%
jerry-3376	116	11.71	70	993	+756.03%	82.94	+608.28%	89	+27.14%
jerry-3408	164	12.00	62	575	+250.61%	42.12	+251.00%	58	−6.45%
jerry-3431	52	5.44	31	358	+588.46%	32.61	+499.45%	31	–
jerry-3433	10	1.87	21	172	+1620.00%	17.37	+828.88%	21	–
jerry-3437	38	5.43	42	1189	+3028.95%	108.42	+1896.69%	107	+154.76%
jerry-3479	225	22.56	120	2124	+844.00%	174.25	+672.38%	191	+59.17%
jerry-3483	67	6.29	38	42	−37.31%	5.20	−17.33%	4	−89.47%
jerry-3506	112	10.76	52	329	+193.75%	386.24	+3489.59%	48	−7.69%
jerry-3523	109	9.72	63	440	+303.67%	39.69	+308.33%	63	–
jerry-3534	170	14.32	96	1037	+510.00%	84.73	+491.69%	105	+9.38%
jerry-3536	146	11.95	123	842	+476.71%	67.73	+466.78%	148	+20.33%
clang-22382	14,842	4618.19	582	312,664	+2006.62%	64,387.15	+1294.21%	2700	+363.92%
clang-22704	10,530	11,665.69	168	54,819	+420.60%	10,341.69	−11.35%	694	+313.10%
clang-23309	24,594	22,821.60	3582	920,361	+3642.22%	268,369.71	+1075.95%	5341	+49.11%
clang-23353	14,585	5739.18	374	204,809	+1304.24%	35,926.18	+525.98%	2612	+598.40%
clang-25900	14,737	8753.28	1562	271,594	+1742.94%	57,745.12	+559.70%	2574	+64.79%
clang-26350	16,748	21,945.57	1613	1,031,666	+6059.94%	275,089.75	+1153.51%	6646	+312.03%
clang-26760	12,925	13,059.59	586	205,756	+1491.92%	54,985.43	+321.03%	2551	+335.32%
clang-27747	7164	5147.64	419	71,072	+892.07%	23,191.19	+350.52%	1054	+151.55%
clang-31259	18,900	19,467.98	2174	215,736	+1041.46%	54,647.29	+180.70%	4250	+95.49%
gcc-59903	18,646	18,169.24	1726	601,419	+3125.46%	145,742.81	+702.14%	5803	+236.21%
gcc-61383	17,279	13,640.33	1704	801,846	+4540.58%	243,083.31	+1682.09%	9817	+476.12%
gcc-61917	17,276	12,082.91	1764	1,685,350	+9655.44%	576,466.30	+4670.92%	11,268	+538.78%
gcc-64990	19,258	27,572.61	2866	791,075	+4007.77%	224,668.02	+714.82%	5042	+75.92%
gcc-65383	11,836	8757.59	1028	438,661	+3606.16%	105,516.61	+1104.86%	3805	+270.14%
gcc-66186	15,649	18,205.31	2617	254,214	+1524.47%	76,236.37	+318.76%	5144	+96.56%
gcc-66375	21,171	35,216.11	2963	194,366	+818.08%	51,084.57	+45.06%	4932	+66.45%
gcc-70127	21,562	46,974.24	1763	272,148	+1162.16%	62,918.22	+33.94%	2498	+41.69%
gcc-71626	4210	454.84	168	8145	+93.47%	568.14	+24.91%	252	+50.00%