Iterating the Minimizing Delta Debugging Algorithm

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

ABSTRACT

Probably the most well-known solution to automated test case minimization is the minimizing Delta Debugging algorithm (DDMIN). It is widely used because it "just works" on any kind of input. In this paper, we focus on the fixed-point iteration of DDMIN (named DDMIN*), more specifically whether it can improve on the result of the original algorithm. We present a carefully crafted example where the output of DDMIN could be reduced further, and iterating the algorithm finds a new, smaller local optimum. Then, we evaluate the idea on a publicly available test suite. We have found that the output of DDMIN* was usually smaller than the output of DDMIN. Using characters as units of reduction, the output became smaller by 67.94% on average, and in the best case, fixed-point iteration could improve as much as 89.68% on the output size of the original algorithm.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

test case minimization, delta debugging, iteration

ACM Reference Format:
Dániel Vince. 2022. Iterating the Minimizing Delta Debugging Algorithm. In *Proceedings of the 13th International Workshop on Automating Test Case Design, Selection and Evaluation (A-TEST '22), November 17–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3548659.3561314

1 INTRODUCTION

Automation is finding its ways nowadays in several fields of software engineering. The earliest automation approaches were probably related to testing: executing test suites, generating new test cases, and – when a test case fails – locating the fault in the software or finding the minimal part of the test case that induced the fault. Automated test case minimization has already seen several decades of work. One of the most well-known solution is Zeller and Hildebrandt's minimizing Delta Debugging algorithm (DDMIN) [9]. Although it is already more than twenty years old, it is still widely used because it "just works" on any kind of input. Since its introduction, there have been many approaches that tried to work better by being more "clever" – and they succeeded. HDD [5], Perses [7], ReduKtor [6], *etc.* could all reduce test cases faster than DDMIN or produce smaller output, but they typically needed some extra information about the structure of the test case, usually a grammar. This need for a grammar can act as an obstacle for some users of test case reducers: a grammar may not be readily available, and writing (or maintaining) it may not be a practical option. In such cases, the structure-unaware nature of the good-old DDMIN comes in very handy.

This is why we have investigated whether it was possible to make DDMIN itself work "better". Improving DDMIN is not an easy task, as it has already been proven that its result is 1-minimal (see Section 2). However, 1-minimality is not a global optimum, which signals that there may be some room for improvement. The question we sought the answer to was whether it was possible to make DDMIN look for and find another (and even better) 1-minimal result once a local optimum was found. A natural idea was to investigate whether it made sense to apply DDMIN multiple times, *i.e.*, whether it could give smaller and smaller 1-minimal results after each iteration. Thus, we wanted to see whether iterating DDMIN until a fixed-point works in theory and in practice. This paper elaborates on the above idea in more detail and presents experimental results of its application[1].

The rest of the paper is organized as follows: First, Section 2 gives a short overview of Delta Debugging to make this paper self-contained. Then, in Section 3, we show a motivational example where DDMIN can be improved upon, and describe the fixed-point iteration of DDMIN. In Section 4, we present the results of an experiment conducted on real test cases. In Section 5, we discuss related work, and finally, in Section 6, we conclude our paper and set directions for future research.

2 BACKGROUND

The minimizing Delta Debugging algorithm (DDMIN) [9] is a systematic iterative approach for reducing 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 about 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

A-TEST '22, November 17–18, 2022, Singapore, Singapore

ACM ISBN 978-1-4503-9452-9/22/11…$15.00
https://doi.org/10.1145/3548659.3561314

[1] Although the idea seems to be natural, or even trivial, interestingly, no work known to us has discussed it previously. The original works of Zeller and Hildebrandt do point out various improvement possibilities of DDMIN, but fixed-point iteration is not among them.

the splitting. The subsets of the new splitting are investigated again, as well as their complements, *i.e.*, 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) keeps 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, a local minimum where the removal of any unit from the set would cause the loss of the interesting property.

An input is composed of elementary 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 behavior, *pass* (also written as ✓) if the test succeeds, or *unresolved* (written as ?) if the result is indeterminate. The initial configuration that triggers the failing outcome is denoted by $c_{✗}$. For the formal definition of Zeller and Hildebrandt's latest formulation of DDMIN, the reader is referred to [9].

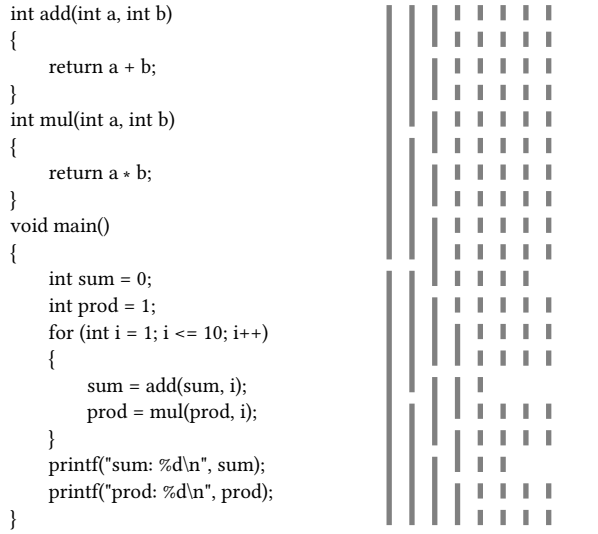
3 ITERATING DELTA DEBUGGING

The program in Listing 1 is a variant of a classic example of program slicing [8]. It computes both the sum and the product of the first ten 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 sub-program 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 [2] 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 Listing 1 is the input (the lines being the units) and the testing function is given as

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

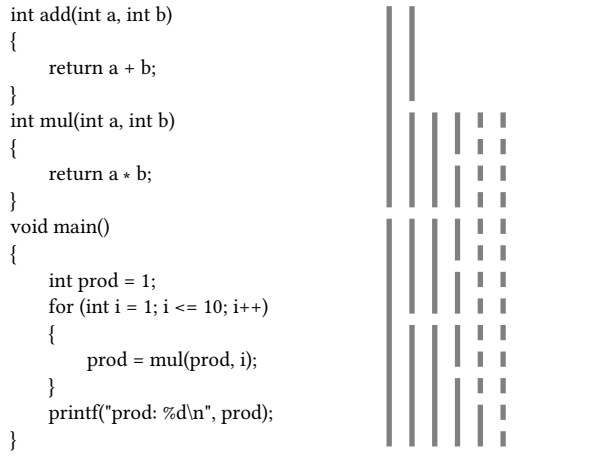
The gray bars on the right of the program code show the progress of DDMIN, from left to right. Every set of vertically aligned bars corresponds to a configuration of the algorithm and shows how that configuration is split into to subsets. This example shows that DDMIN could "slice away" the lines of 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

Listing 1: Example C program that computes the sum and product of the first ten natural numbers, and the execution of DDMIN on it while keeping 10! on the output.



```
1  int add(int a, int b)
2  {
3      return a + b;
4  }
5  int mul(int a, int b)
6  {
7      return a * b;
8  }
9  void main()
10 {
11     int sum = 0;
12     int prod = 1;
13     for (int i = 1; i <= 10; i++)
14     {
15         sum = add(sum, i);
16         prod = mul(prod, i);
17     }
18     printf("sum: %d\n", sum);
19     printf("prod: %d\n", prod);
20 }
```

Listing 2: The output of DDMIN on the program of Listing 1, and the re-execution of DDMIN.



```
1  int add(int a, int b)
2  {
3      return a + b;
4  }
5  int mul(int a, int b)
6  {
7      return a * b;
8  }
9  void main()
10 {
11     int prod = 1;
12     for (int i = 1; i <= 10; i++)
13     {
14         prod = mul(prod, i);
15     }
16     printf("prod: %d\n", prod);
17 }
```

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 Listing 2), 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 Listing 3. This is the global optimum for this example, so further executions of DDMIN are not visualized.

Listing 3: The output of DDMIN on the program of Listing 2.

```
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 }
```

Now, 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* [5] – and define it as follows:

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

Note that 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 so-called 1-tree-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.

4 EXPERIMENTAL RESULTS

To see how DDMIN* performs on real-life test cases, we have taken the test cases of the JerryScript Reduction Test Suite (JRTS)[2] and minimized them with both DDMIN and DDMIN*. For the experiment, we have used the Picire tool[3], an open-source Python implementation of DDMIN, and prototyped DDMIN* into it to evaluate its effects. The Picire framework is highly parametrizable, which we made use of: in the experiments, the "reduce to subset" step of DDMIN was disabled, the complement tests were performed in backward syntactic order, and the content-based caching was enabled.

Table 1 presents the results of reducing the test cases with characters as the unit of reduction. For each test case, the first group of values shows the properties of the inputs: the name and the size expressed in the unit of reduction. Then, the second group is the baseline data, *i.e.*, those measured using the traditional DDMIN algorithm: the number of testing steps needed to accomplish the reduction, and the size of the output. The last group of values contains the data collected during the executions of the fixed-point iterated algorithm (DDMIN*) on the test cases. In addition to the absolute numbers, we also give the changes relative to the baseline data. Plus, the number of iterations necessary to reach the fixed point is also shown.

If an input is not optimal (*i.e.*, it could be reduced somehow), then at least two iterations of DDMIN* are expected: whenever an iteration manages to reduce the configuration, there will be a next iteration that tries to continue the reduction. Only if the configuration cannot be reduced further in an iteration, will the algorithm halt. The iteration counts in Table 1 support this expectation. The highest number was 15 (for test case jerry-3479), and the lowest was 3 (also signaling that DDMIN* could always further reduce the result of DDMIN).

For all of the test cases, DDMIN* produced significantly smaller results than DDMIN. The output configuration got smaller by a minimum of 9.82% (in the case of jerry-3536) and by a maximum of 89.68% (for jerry-3437), and the average improvement was 67.94%. The cost of these improvements was an increased number of test executions, in the range of 15.82% and 337.17%, 111.41% being the average. Let us highlight the experiment of the jerry-3479 test case, which shows some impressive results. In that case, the input contained 5,201 characters, and DDMIN could reduce it to 2,383 characters, which is 45.81% of the input. However, the 15 iterations of DDMIN* could reduce it further to 433 characters, which is only 8.69% of the input size.

Based on the data and observations above, we can conclude from the experiments: 1) iterating DDMIN until a fixed-point was reached had a positive effect on the size of the result in general; 2) with character-based reduction, all inputs of the test suite could be reduced further with DDMIN*, the average and maximum configuration size improvements being 67.94% and 89.68%, respectively; and 3) these smaller 1-minimal results came at a cost of more testing steps: the reduction required 111.41% more steps on average than the baseline DDMIN.

5 RELATED WORK

One of the first works on automated test case reduction is Delta Debugging by Zeller and Hildebrandt, minimizing inputs of arbitrary format [9]. Gharachorlu and Sumner [4] proposed a modified version of DDMIN, the *One Pass Delta Debugging (OPDD)*, that skips the "reduce to complement" steps, and showed that it can also achieve 1-minimal results with linear complexity (if certain circumstances are met).

Artho investigated Delta Debugging in his "Iterative Delta Debugging" study [1]. However, despite the similarities between the titles, the two studies are unrelated. 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.

The cost of the generality of DDMIN is a 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

[2] https://github.com/vincedani/jrts
[3] https://github.com/renatahodovan/picire

Table 1: Results with Character Granularity

Test Case		DDMIN		DDMIN*				
Name	Chars	Steps	Chars	Iters	Steps		Chars	
jerry-3299	1,767	4,959	542	12	13,287	*+167.94%*	130	*-76.01%*
jerry-3361	1,953	2,276	427	12	9,950	*+337.17%*	175	*-59.02%*
jerry-3376	6,626	15,008	1,216	9	31,602	*+110.57%*	306	*-74.84%*
jerry-3408	2,681	4,869	557	7	9,707	*+99.36%*	178	*-68.04%*
jerry-3431	1,065	2,228	207	6	3,120	*+40.04%*	58	*-71.98%*
jerry-3433	961	588	74	4	681	*+15.82%*	14	*-81.08%*
jerry-3437	6,597	11,764	1,017	6	15,982	*+35.86%*	105	*-89.68%*
jerry-3479	5,201	17,391	2,383	15	66,097	*+280.06%*	452	*-81.03%*
jerry-3483	492	388	42	4	516	*+32.99%*	17	*-59.52%*
jerry-3506	3,760	5,797	658	8	12,322	*+112.56%*	192	*-70.82%*
jerry-3523	3,928	8,666	832	6	15,542	*+79.34%*	206	*-75.24%*
jerry-3534	1,927	3,407	378	6	6,512	*+91.14%*	128	*-66.14%*
jerry-3536	829	1,119	163	3	1,628	*+45.49%*	147	*-9.82%*

proposed to use information about the input format encoded in grammars, *i.e.*, converting test cases into a tree representation [5], and apply delta debugging to the levels of the tree, called Hierarchical Delta Debugging. This approach helped to 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. Motivated by their results, we could formalize the fixed-point iteration of DDMIN, called DDMIN*.

An analogy between test case reduction and program slicing was recognized by Binkley *et al.* [3]. 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's, however, they have not tried to start the algorithm over to find another, more optimal 1-minimal result.

6 SUMMARY

In this work, we have investigated whether the fixed-point iteration of DDMIN – denoted as DDMIN* – improves the effectiveness of the algorithm. We have presented an example input where DDMIN* improved the found local minimum, then we evaluated the idea on a publicly available test suite. We have found that the output of DDMIN* was usually smaller than the output of DDMIN. Using characters as units of reduction, the output became smaller by 67.94% on average. In the best case, fixed-point iteration could improve as much as 89.68% on the output size of the original algorithm. The cost of the smaller results is an increase in the number of testing steps.

As for future work, we have plans to continue this topic of research. We wish to conduct experiments with line based granularity, and with HDD as it uses DDMIN on the levels of its internal tree representation, and investigate whether utilizing the fixed-point iteration on the levels could improve the overall output of HDD*. Furthermore, we would like to explore additional optimization possibilities to improve the DDMIN*.

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.

REFERENCES

[1] Cyrille Artho. 2009. Iterative Delta Debugging. In *Hardware and Software: Verification and Testing (Lecture Notes in Computer Science, Vol. 5394)*. Springer, 99–113. https://doi.org/10.1007/978-3-642-01702-5_13

[2] David Binkley, Nicolas Gold, Mark Harman, Syed Islam, Jens Krinke, and Shin Yoo. 2014. ORBS: Language-Independent Program Slicing. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)* (Hong Kong, China). ACM, 109–120. https://doi.org/10.1145/2635868.2635893

[3] David Binkley, Nicolas Gold, Mark Harman, Syed Islam, Jens Krinke, and Shin Yoo. 2015. ORBS and the limits of static slicing. In *Proceedings of the 15th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2015)* (Bremen, Germany). IEEE, 1–10. https://doi.org/10.1109/SCAM.2015.7335396

[4] Golnaz Gharachorlu and Nick Sumner. 2018. Avoiding the Familiar to Speed Up Test Case Reduction. In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 426–437. https://doi.org/10.1109/QRS.2018.00056

[5] Ghassan Misherghi and Zhendong Su. 2006. HDD: Hierarchical Delta Debugging. In *Proceedings of the 28th International Conference on Software Engineering (ICSE '06)* (Shanghai, China). ACM, 142–151. https://doi.org/10.1145/1134285.1134307

[6] Daniil Stepanov, Marat Akhin, and Mikhail Belyaev. 2019. 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)* (San Diego, California, United States). IEEE, 317–326. https://doi.org/10.1109/ASE.2019.00038

[7] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. 2018. Perses: Syntax-Guided Program Reduction. In *Proceedings of the 40th ACM/IEEE International Conference on Software Engineering (ICSE '18)* (Gothenburg, Sweden). ACM, 361–371. https://doi.org/10.1145/3180155.3180236

[8] Frank Tip. 1995. A survey of program slicing techniques. *Journal of Programming Languages* 3, 3 (1995), 121–189.

[9] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Transactions on Software Engineering* 28, 2 (Feb. 2002), 183–200. https://doi.org/10.1109/32.988498