

DDMin versus QuickXplain – An Experimental Comparison of two Algorithms for Minimizing Collections

Oliver A. Tazl
Institute of Software Technology
Graz University of Technology, Austria
Graz, Austria
oliver.tazl@ist.tugraz.at

Christopher Tafeit
Institute of Software Technology
Graz University of Technology
Graz, Austria
christopher.tafeit@gmail.com

Franz Wotawa
Institute of Software Technology
Graz University of Technology
Graz, Austria
wotawa@ist.tugraz.at

Alexander Felfernig
Institute of Software Technology
Graz University of Technology
Graz, Austria
alexander.felfernig@ist.tugraz.at

Abstract—About two decades ago, two algorithms, i.e., DDMin and QuickXPlain, for minimizing collections, were independently proposed and gained attention in the two research areas of Software Engineering and Artificial Intelligence, respectively. Whereas DDMin was developed for reducing a given test case, QuickXPlain was intended to be used for obtaining minimal conflicts efficiently. In this paper, we compare the performance of both algorithms with respect to their capabilities of minimizing collections. We found out that one algorithm outperforms the other under given prerequisites and vice versa. These findings help to select the suitable algorithm for a given task.

Index Terms—test case minimization, conflict minimization, software testing, application to diagnosis and configuration

I. INTRODUCTION

There are many important tasks in different areas of Software Engineering (SE) and Artificial Intelligence (AI) requiring minimizing collections. In SE, localizing faults may require reducing inputs that lead to crashes or other unexpected behavior. For example, if a compiler crashes due to a given input program, fault localization becomes way more easy when knowing only those parts of the textual input that reveal the bug. In AI and there, for example, in Model-based Diagnosis (MBD) [1], [2] we rely on minimal conflicts used to compute minimal diagnoses. In any case, we have to deal with obtaining a – at least smaller – sub-collection that still fulfills the same criteria (or properties) as the original collection.

In SE and AI independently, two algorithms for minimizing a test case and a conflict were introduced about 20 years ago, substantially influencing in their respective research fields. In SE, Zeller and Hildebrandt [3] described the Delta Debugging algorithm DDMin and its use for simplifying failure-inducing inputs. In AI, Junker [4] suggested QuickXPlain for minimizing conflicts. Both algorithms rely on the general divide and

conquer approach for minimization. Both algorithms aim at providing a smaller collection but come – at least partially – with limited guarantees regarding finding a smaller or even minimal solution. This is due to the fact that the computational complexity required would be exponential in the size of the collection.

Interestingly, these algorithms have not been considered in the respected research field of the other algorithm. Moreover – to our knowledge – nobody has ever compared these algorithms with respect to execution time and their capabilities of minimizing original collections. In this paper, we want to close this gap and provide an experimental evaluation where we compare DDMin with QuickXPlain on the same input collections and properties. The properties have been designed in a way allowing to make conclusions regarding in which cases one or the other algorithm behaves superior. Such an analysis has an impact to both SE as well as AI allowing to decide which algorithm to use under which circumstances. More concrete questions we want to answer in this paper are whether we can use DDMin for minimizing conflicts like QuickXPlain, or to use QuickXPlain for minimizing a test case instead of DDMin.

In summary, the content of this paper provides the following contributions: (i) it comes up with a framework allowing to compare DDMin and QuickXPlain directly (although they have been originally designed to fit different purposes), and (ii) it experimentally compares two different algorithms using a parametric set of inputs aiming at providing more insights regarding superiority of an algorithm in a particular application context.

We organize the remainder of the paper as follows: In Section II, we introduce the basic foundation of DDMin and QuickXPlain. Afterwards in Section III, we outline the underlying implementation and the experimental evaluation

procedure. Furthermore, we discuss the obtained evaluation results and derive some concluding remarks regarding the comparison between DDMin and QuickXPlain. Finally, we give an overview of related research and conclude the paper.

II. BASIC FOUNDATIONS

To be self-contained, we outline the underlying foundations and the two algorithms DDMin and QuickXPlain. We start defining the underlying minimization problem to be solved. For this purpose, we assume that we have: (i) a collection of elements $C = \{e_1, \dots, e_n\}$ where each element $e \in C$ is from a domain D , and (ii) a function *test* that takes a collection of elements as input and returns either \surd or \times , which is defined as follows:

$$test(x) = \begin{cases} \times & \text{if } x \text{ fulfills the given criteria or properties} \\ \surd & \text{otherwise} \end{cases}$$

Note that we assume $test(C)$ returns \times for the original collection of elements. Furthermore, in the original definition of Zeller and Hildebrandt [3] $test(x)$, with $x \neq C$ may also return $?$ in case there is an unexpected behavior of x but which diverges from the behavior of C . However, in DDMin $?$ and \surd are treated equivalently so there is no need to distinguish these two cases.

The problem of minimization of C with respect to a given *test* function is to find an ideally smaller $C' \subseteq C$ (if it exists) for which $test(C') = \times$. If we want to have a really minimal C' , we may come up with two corresponding definitions:

subset minimal: A collection $C' \subseteq C$ is called subset minimal if and only if there is no $C'' \subset C'$ where $test(C'') = \times$ holds. There might be more than one subset minimal solution. In the context of delta debugging [3] this type of minimum is referred to as local minimum.

cardinality minimal: Alternatively, a collection $C' \subseteq C$ is cardinality minimal if and only if there exists no smaller C'' , i.e., $|C'| > |C''|$ where $test(C'')$ holds. In delta debugging, cardinality minimums are called global minimums.

Obviously, finding either subset minimal or cardinality minimal solutions is exponential in the size of the input collection C because we have to check all subsets of C . Hence, in practice, we may be more interested in finding a smaller solution if such a solution exists instead of a minimal one. It is worth noting in this context that DDMin only guarantees to return a solution with one element less, if such a solution exists. However, the evaluation indicates that in practice DDMin is way more efficient in removing unnecessary parts of a collection. Instead, QuickXPlain guarantees subset minimality but not cardinality minimality. Hence, when evaluating both algorithms, we are interested in how far away provided solutions are from the minimal one.

In the following, we describe the algorithms. Note that the used pseudo-code is adapted from the original one for allowing to use the collection C as well as the *test* function as input

directly. However, we did neither improve the algorithms nor change their originally stated behavior.

We first define helper functions that are used in the algorithms, namely *split* and *complement*.

split: *split* is able to divide a given collection of elements C into n parts leading to new collections $C_1 \dots C_n$. Functions *split* returns collections that are pairwise disjoint ($C_1 \cap C_2 = \emptyset$), completely represent all elements of the original collection, i.e., $C_1 \cup C_2 \dots \cup C_n = C$. Moreover, all the sub-collection have approximately the same size, i.e., For all i , and j : $|C_i| = |C_j| + x$ with $x \in \{0, 1\}$.

complement: The complement of a sub-collection C_1 is a collection comprising all elements of the original collection C that are not in C_1 . I.e., *complement* is defined as follows: $complement(C_1) = \{x \mid x \in C \wedge x \notin C_1\}$.

In Algorithm 1, we depict the pseudo-code of the Delta Debugging (DD) algorithm DDMin of Zeller and Hildebrandt [3], which reduces the input collection using a divide and conquer strategy. The algorithm uses to some extent ideas from binary search for trilling down the failure inducing input systematically. The overall process implemented by DDMin has four steps: *reduce to subset*, *reduce to complement*, *increase granularity* and *done*. These steps split the input into smaller parts and combine them if needed to narrow down the faulty input. Correct parts were cut off to focus on the remaining faulty ones in order to produce a smaller input that triggers the faulty behaviour.

QuickXPlain (QXP) as shown in Algorithm 2, was introduced by [4] to solve over-constrained problems by providing explanations. Those are also calculated by a divide and conquer approach. The input is a problem instance which comprises an analysed set/collection (\mathcal{A}) and a background set/collection (\mathcal{B}). For our experiments we assume \mathcal{B} to be empty. The function *test* is then used to determine the necessity of executing the algorithm. In the trivial case of a correct or not dividable input, the execution is stopped before entering the recursion. Next, the recursion is started. The procedure starts by partitioning the analyzed set into two, in our case equal-sized, subsets and analyze these subsets recursively until a minimal solution can be provided.

III. IMPLEMENTATION AND EXPERIMENTAL EVALUATION

In this section, we present the prerequisites behind the experimental evaluation in order to assure that results can be reproduced.

A. Prerequisites

The prerequisites comprise implementation details, the input collections used for the evaluation, and the execution environment. The objective behind the experimental evaluation was to answer the following 2 research questions:

RQ1 "Does *QuickXPlain* behave superior compared to *DDMin* with respect to the execution time or vice-versa?"

RQ2 "Do both algorithms *DDMin* and *QuickXPlain* deliver minimal solutions?"

Algorithm 1 DDMIN [3]

Input: A collection c_f where $test(c_f) = \times$

Output: A potentially smaller sub-collection of c_f where $test$ returns \times .

```
1: procedure DDMIN( $c_f$ )
2:   if  $test(c_f) = \times$  then
3:      $ddmin2(c_f, 2)$ 
4:   end if
5:   return  $c_f$ 
6: end procedure
7: procedure DDMIN2( $c_f, n$ )
8:   if  $|c_f| = 1$  then
9:     return
10:  end if
11:   $\Delta_1, \dots, \Delta_n \leftarrow split(c_f, n)$ 
12:  while  $i = 1 \dots n$  do ▷ reduce to subset
13:    if  $test(\Delta_i) = \times$  then
14:       $ddmin2(\Delta_i, 2)$ 
15:    return
16:  end if
17: end while
18:   $\nabla_1, \dots, \nabla_n \leftarrow complement(c_f, \Delta_1, \dots, \Delta_n)$ 
19:  while  $i = 1 \dots n$  do ▷ reduce to complement
20:    if  $test(\nabla_i) = \times$  then
21:       $ddmin2(\nabla_i, max(n - 1, 2))$ 
22:    return
23:  end if
24: end while
25:  if  $n < |c_f|$  then ▷ increase granularity
26:     $ddmin2(c_f, min(|c_f|, 2 * n))$ 
27:  return
28: end if
29:  return  $c_f$  ▷ done
30: end procedure
```

In order to answer these research questions we implemented both algorithms, i.e., DDMIN and QuickXPlain, using the programming language Java 17 using libraries for logging data as a foundation. As stated we implemented the algorithms without manual optimisations for improving the execution time. Hence, we relied on the pseudo-code and description provided by the respective originators of the algorithms.

To implement the $test$ function required, we implemented a configurable test oracle, which uses JSON files to load and store input configurations. These configurations store the given collection of elements e_1, \dots, e_n of cardinality n , where each element e_i is marked as either neutral or failure-inducing element. The test oracle now implements the function $test$ as

Algorithm 2 QuickXPlain [4], [5]

Input: a pair $\langle \mathcal{A}, \mathcal{B} \rangle$ where \mathcal{A} is the analyzed set and \mathcal{B} is the background, which we assume to be the empty set in the context of this paper.

Output: a minimal set wrt. $\langle \mathcal{A}, \mathcal{B} \rangle$, if existent; pass, otherwise

```
1: procedure QXP( $\langle \mathcal{A}, \mathcal{B} \rangle$ )
2:   if  $test(\mathcal{A} \cup \mathcal{B}) = \surd$  then
3:     return pass
4:   else if  $|\mathcal{A}| = \emptyset$  then
5:     return  $\emptyset$ 
6:   else
7:     return  $QXP^*(\mathcal{B}, \langle \mathcal{A}, \mathcal{B} \rangle)$ 
8:   end if
9: end procedure
10: procedure QXP( $\mathcal{C}, \langle \mathcal{A}, \mathcal{B} \rangle$ )
11:  if  $\mathcal{C} \neq \emptyset$  and  $test(\mathcal{B}) = \times$  then
12:    return  $\emptyset$ 
13:  end if
14:  if  $|\mathcal{A}| = 1$  then
15:    return  $\mathcal{A}$ 
16:  end if
17:   $\mathcal{A}_1, \mathcal{A}_2 \leftarrow split(\mathcal{A}, 2)$ 
18:   $X_2 \leftarrow QXP^*(\mathcal{A}_1, \langle \mathcal{A}_2, \mathcal{B} \cup \mathcal{A}_1 \rangle)$ 
19:   $X_1 \leftarrow QXP^*(X_2, \langle \mathcal{A}_1, \mathcal{B} \cup X_2 \rangle)$ 
20:  return  $X_1 \cup X_2$ 
21: end procedure
```

follows: The test oracle fails on a particular not necessarily strict subset C of the collection e_1, \dots, e_n , if C contains all failure-inducing elements of in e_1, \dots, e_n . In this case the test oracle returns \times and otherwise \surd .

We use the configurations for coming up with different test inputs for DDMIN and QuickXPlain. For this purpose, we created the configurations automatically using a program comprising the size n of the collection, and the number of failure-inducing elements (fail elements for short) k as inputs. For the experiments, we generated two types of inputs. The first type comprises tests where clusters of k fail elements arrange at the beginning of the collection, after k elements, after $2 \cdot k$ elements, etc. until reaching the end of the collection. In this type, which we refer as **cluster test input**, we only have one cluster of fail elements in each collection. In the second type of inputs, which we refer to as **random test input**, we generate collections of size n and randomly select k elements in this collection to be fail elements.

For the experiments, we created 3 sets of cluster test inputs of size 10,000 with 50, 500, and 1,000 fail elements respectively. For each of these sets we moved the cluster from the beginning to the end to obtain all cluster test inputs. For the random cluster with also relied on collections of size $n = 10,000$ considering 50, 500, and 1,000 randomly selected fail elements. For the random cluster, we generated 300 of such different configurations. In total, we executed about 400 different configurations. It is worth noting that we selected

collections of size 10,000 because of obtaining reasonable execution times ranging from milliseconds to less than 1 hour.

We published the implementation of the algorithms, the used configurations for the experimental analysis, the batch programs for running the experiments, as well as all data obtained is on Github to be used for further research.

B. Results

In order to answer the two research questions, we carried out the experiments, where we executed each input configuration ten times. Reported results are averaged to reduce the side effects of the execution environment on the results. In the following, we report execution time in milliseconds (ms) or seconds (s). The Java implementation of the algorithms was executed using the OpenJDK 17.0.1 Hotspot JVM using a computer with the following configuration: AMD Ryzen 9 3900x 12-Core 3.8 GHz processor and 64 GB RAM running Windows 11. Note also that the experimental evaluation is automated allowing to re-execute it on demand.

To answer **RQ1**, we measured the execution time required for minimizing the different input configurations. As already said we used two different types of configuration, i.e., the cluster test input and the random test input. In Figure 1, we display the result of the cluster test input where we combined a collection of 10,000 elements with 50, 100, and 1,000 fail elements respectively. All fail elements are in the same cluster, which we move from the left to the right of the collection for obtaining different input collections. Note when using a block of 1,000 fail elements, we obtain input collections, where the fail element cluster comprises elements 0-999, 1,000-1,999, etc.

The results for the cluster test input allow us to derive the following differences in the behavior of DDMin and QuickXPlain. First, DDMin comes with an almost constant execution time behavior with the exception of a cluster of fail elements in the middle. In this case, depending on the number of fail elements, the behavior of DDMin may even be worst when compared to QuickXPlain. This behavior might be due to the fact that DDMin requires to increase cardinality as well as to compute complements in order to find the cluster. QuickXPlain, however, has a more or less linear execution time behavior with respect to the position of the fail element cluster. Fail elements at the beginning (i.e., at the left side of the collection) lead to an execution time similarly to DDMin, whereas DDMin is superior when the fail element cluster is located on the right side of the collection.

In Figure 2, we summarize the results of the random test input configurations containing 10,000 elements and 50, 100 and 1,000 random distributed fail elements within. We see that QuickXPlain performs exceptionally well in this case compared to DDMin. In all cases, QuickXPlain outperforms DDMin.

Research question **RQ1** can now be answered as follows. QuickXPlain does not necessarily outperform DDMin when

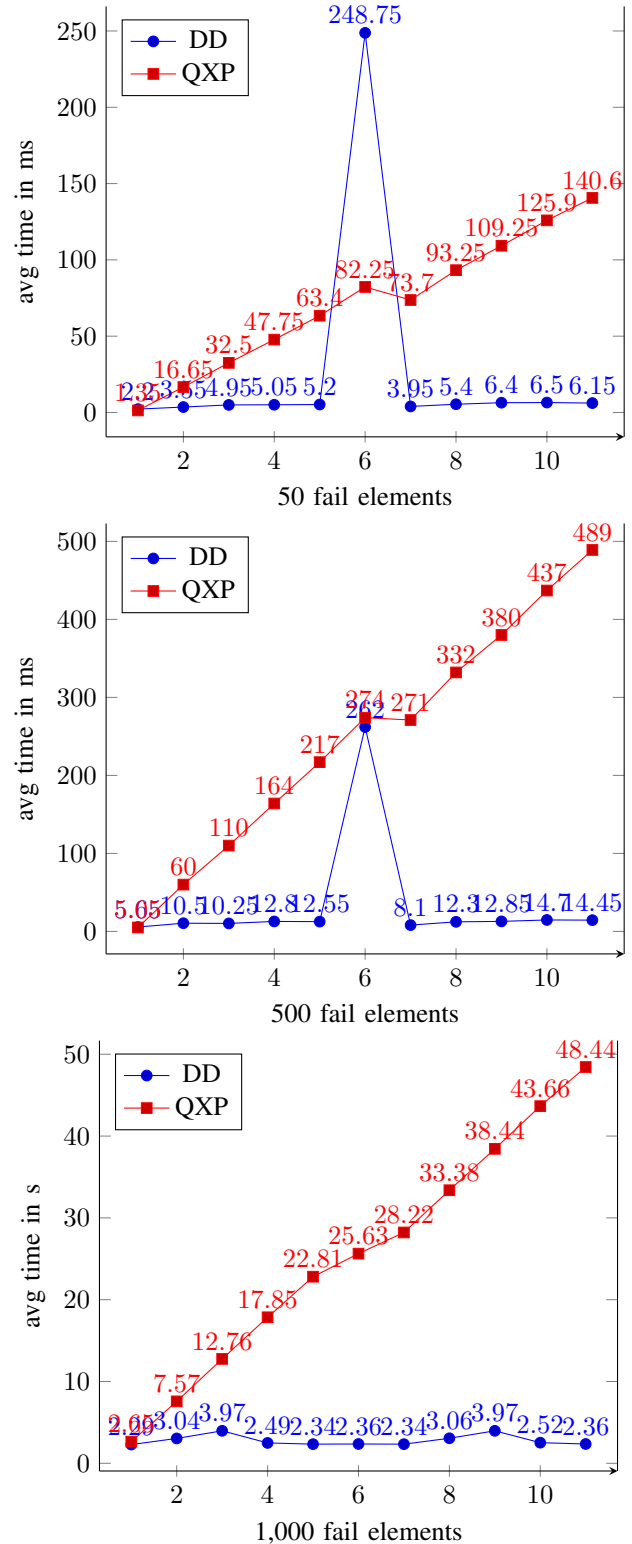


Fig. 1: Execution time results obtained using the cluster test inputs comprising 10,000 elements in combination with 50, 100 and 1000 fail elements respectively.

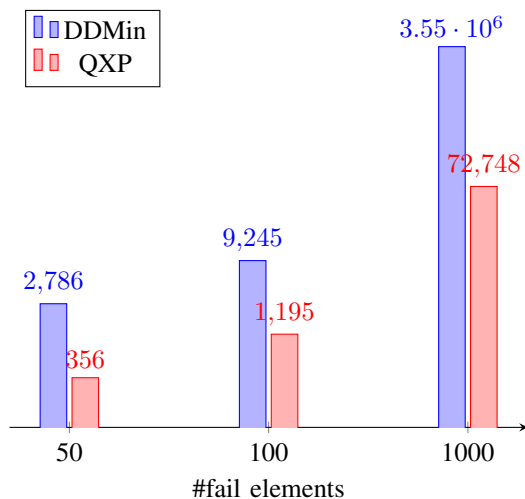


Fig. 2: Execution time results obtained using the random test inputs comprising 10,000 elements, and randomly selected 50, 100, and 1,000 fail elements. The depicted execution time is the average execution time in milliseconds.

considering execution time. In case of a single cluster of fail elements within a given collection DDMin seems to be superior, but execution time depends on the position of the cluster in the collection as well as on the number of the fail elements. If the number is larger, DDMin seems to be a better choice. However, in case of many clusters (like in the case of the random test input), QuickXPlain is faster than DDMin. Hence, depending on the given input arguments either QuickXPlain or DDMin should be chosen. For test case minimization (especially when considering inputs of a compiler), where there is most likely one cluster of interest, DDMin seems to be more appropriate. For conflict minimization, where there are many different randomly distributed small clusters, QuickXPlain seems to be the algorithm of choice. However, further experimental evaluations are required considering real-world test inputs instead of synthetically generated example inputs as we used in our experimental evaluation.

To answer research question **RQ2**, we further compared the output, i.e., the minimized collection, of DDMin and QuickXPlain obtained for all the different test inputs with the optimal solution, i.e., the fail elements in the respective collection. For all test inputs, both algorithms returned the optimal solution as outcome. Hence, for the given test inputs, we can answer **RQ2** with yes. However, this result seems not to be conclusive and further experiments are required, considering real-world test inputs as well as more sophisticated synthetic examples.

C. Threads to Validity

Like for all experimental evaluations, there are different threats to internal and external validity, which we discuss. Regarding internal validity, we have to mention the computational environment comprising hardware and software used. This includes the operating systems as well as the programming language used. In particular, when relying on Java

and its virtual machine, we know mechanisms like garbage collection and just in time optimization that we are not able to control, but influencing measured execution time. We try to mitigate those effects by repeating the tests and averaging the obtained time results. Moreover, it is worth mentioning that we implemented both algorithms in the same language making use of the same libraries not using any optimizations. Hence, we do not expect any bias in the measured execution time that originates from an implementation.

Regarding external validity, we have to mention that the evaluation is based on solely two different test input categories namely the cluster and the random test input. For the random case, the outcome was always identifying QuickXPlain as the fastest algorithm. For the cluster, DDMin can be said to be superior in most of the cases. Although, these results allow to state superiority on average in some more specific cases, such results may not be generally valid in all contexts including considering real-world example inputs. However, at least for those examples close to the synthetic one, we would expect a similar outcome.

IV. RELATED RESEARCH

The overall goal of this paper is to compare two algorithms that support the minimization of conflicts. Basic related minimality properties are *subset minimality* and *minimal cardinality* where the latter is more restrictive, i.e., also takes into account the criteria of subset minimality. Which criteria should be applied depends on the corresponding application context. Subset minimality is useful, for example, if a preference relationship can be defined over the given set of conflict candidates (e.g., given component failure probabilities or user preferences with regard to a set of product properties) [6], whereas minimal cardinality is useful in the case of non-available preference relationships (e.g., when searching failure-inducing inputs in the context of software testing) [3]. Especially in real-time scenarios, minimality criteria have to be relaxed to find a trade-off between conflict identification costs (time efforts) and costs for conflict resolution [7].

The algorithms discussed in this paper can be regarded as specific instances of so-called explanation algorithms [8]. DDMin [3] as well as QuickXPlain [4] support the determination of *minimal conflict sets* (fulfilling the criteria of subset minimality) which are also denoted as *minimal unsatisfiable subsets* (MUS) [9] or *minimal unsatisfiable cores* (MUC) [10]. Minimal conflict sets are well-suited for supporting the identification of minimizing collections in the context of test case minimization but as well in explorative interactive settings such as knowledge-based configuration [11] where users should be better supported in understanding relationships between different product properties.

In other scenarios, we are more interested in explanations that help to restore consistency, for example, [12], [6] focus on consistency restoration of inconsistent knowledge bases. In such scenarios, conflict sets are used as input for a hitting set algorithm [1] that helps to determine minimal diagnoses which are also denoted as *minimal correction subset* (MCS) [9]. In

contrast to hitting set based conflict resolution (diagnosis), direct diagnosis helps to determine hitting sets without the need of predetermining minimal conflicts sets. An example algorithm is FastDiag [13] which follows a divide-and-conquer based approach for identifying minimal hitting sets.

There is also a natural relationship between minimal conflict sets and minimal diagnoses in terms of a duality property [14]: for a given set CS of minimal conflicts we are able to determine a corresponding set DS of minimal diagnoses using a HSDAG based approach [1]. Vice-versa, we are able to derive exactly CS if we construct a HSDAG for DS .

Finally, the complement of a minimal hitting set, i.e., a minimal correction subset (MCS), is a so-called *maximal satisfiable subset* (MSS) [9]. Whereas MCSs Δ are characterized by the property that no subset of Δ fulfills the property that all conflicts can be resolved, MSSs Γ are characterized by the property that no extension of Γ remains satisfiable.

Summarizing, the two algorithms analyzed in this paper help to determine minimal conflicts sets which can then be exploited to determine corresponding minimal correction subsets as well as maximal satisfiable subsets.

In the context of software engineering and in particular testing, the minimization of collections is an important task. Besides minimizing a particular test case using Delta Debugging, the optimization of test suites (see, e.g., [15]) is of interest. There have been several algorithms proposed including Greedy algorithms [16] adopting solutions for the well-known set covering problem. In any of these cases, there is more information available than only a collection and a *test* function. For test suite minimization, we usually know the influence of each test to the execution of a program, i.e., the statements that are executed. In the case of minimizing one test case, such knowledge is usually not available. Therefore, we focused solely on Delta Debugging for comparing it with QuickXPlain.

V. CONCLUSIONS

In this paper, we presented the outcome of an experimental evaluation of two algorithms for minimizing collections, i.e., QuickXPlain and DDMin, which originated from the different research areas of Artificial Intelligence and Software Engineering. In order to allow algorithm comparison, we provided a framework that takes a collection and a test function as input, and calls the algorithms for computing a sub-collection that still returns the same test function result. The underlying objective behind the research was to clarify whether one of the algorithms is superior with respect to execution time or the obtained minimization output.

To answer this question, we came up with different test examples generated automatically based on certain parameters. Based on our experiments, we were able to come up with the following results: (i) in cases where minimization has to deal with one cluster, i.e., a set of elements in the collection, which are in close proximity, DDMin provides an almost constant execution time behavior outperforming QuickXPlain. (ii) in case of random distribution of elements to be selected

in a collection, QuickXPlain is superior with respect to its execution time. (iii) both algorithms always returned the smallest possible sub-collection. Hence, it seems that both algorithms were well designed for their particular area of use, i.e., minimizing a test case (DDMin), and minimizing conflicts (QuickXPlain). However, it is required to carry out further experiments considering real-world examples from the application domains of test case and conflict minimization, as well as more different generated examples for identifying additional parameters influencing the execution time behavior as well as the capabilities of finding a minimal solution.

ACKNOWLEDGEMENT

The research was supported by ECSEL JU under the project H2020 826060 AI4DI - Artificial Intelligence for Digitising Industry. AI4DI is funded by the Austrian Federal Ministry of Transport, Innovation and Technology (BMVIT) under the program "ICT of the Future" between May 2019 and April 2022. More information can be retrieved from <https://iktderzukunft.at/en/bmvti>.

REFERENCES

- [1] R. Reiter, "A theory of diagnosis from first principles," *Artificial Intelligence*, vol. 32, no. 1, pp. 57–95, 1987.
- [2] J. de Kleer and B. C. Williams, "Diagnosing multiple faults," *Artificial Intelligence*, vol. 32, no. 1, pp. 97–130, 1987.
- [3] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, feb 2002.
- [4] U. Junker, "Quickxplain: Preferred explanations and relaxations for over-constrained problems," in *Proceedings of the 19th National Conference on Artificial Intelligence*, ser. AAAI'04. AAAI Press, 2004, p. 167–172.
- [5] P. Rodler, "Understanding the quickxplain algorithm: Simple explanation and formal proof," *CoRR*, vol. abs/2001.01835, 2020. [Online]. Available: <http://arxiv.org/abs/2001.01835>
- [6] A. Felfernig, G. Friedrich, D. Jannach, and M. Stumptner, "Consistency-based diagnosis of configuration knowledge bases," *Artificial Intelligence*, vol. 152, no. 2, pp. 213 – 234, 2004.
- [7] A. Felfernig, R. Walter, J. Galindo, D. Benavides, M. Atas, S. Polat-Erdeniz, and S. Reiterer, "Anytime diagnosis for reconfiguration," *Journal of Intelligent Information Systems*, vol. 51, pp. 161–182, 2018.
- [8] S. Gupta, B. Genc, and B. O'Sullivan, "Explanation in constraint satisfaction: A survey," in *13th International Joint Conference on Artificial Intelligence, IJCAI-21*, 2021, pp. 4400–4407.
- [9] M. H. Liffiton and K. A. Sakallah, "Algorithms for computing minimal unsatisfiable subsets of constraints," *Journal of Automated Reasoning*, vol. 40, pp. 1–33, 2008.
- [10] I. Lynce and J. P. M. Silva, "On computing minimum unsatisfiable cores," in *7th International Conference on Theory and Applications of Satisfiability Testing*, Vancouver, BC, Canada, 2004.
- [11] U. Junker, "Configuration," in *Handbook of Constraint Programming*, F. Rossi, P. van Beek, and T. Walsh, Eds. Elsevier, 2006, pp. 837–873.
- [12] R. Bakker, F. Dikker, F. Tempelman, and P. Wogmim, "Diagnosing and solving over-determined constraint satisfaction problems," in *IJCAI'93*. Morgan Kaufmann, 1993, pp. 276–281.
- [13] A. Felfernig, M. Schubert, and C. Zehentner, "An efficient diagnosis algorithm for inconsistent constraint sets," *AI for Engineering Design, Analysis, and Manufacturing (AIEDAM)*, vol. 26, no. 1, pp. 53–62, 2012.
- [14] R. Stern, M. Kalech, A. Feldman, and G. Provan, "Exploring the duality in conflict-directed model-based diagnosis," in *AAAI'12*. AAAI, 2012, pp. 828–834.
- [15] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011.
- [16] L. Zheng, M. Harman, and R. Hierons, "Search algorithms for regression test case prioritization," *IEEE Trans. Softw. Eng.*, vol. 33, no. 4, p. 225–237, Apr. 2007. [Online]. Available: <https://doi.org/10.1109/TSE.2007.38>