Published version: Tsukasa Yagi, Shinpei Hayashi, "Toward Interactive Optimization of Source Code Differences: An Empirical Study of Its Performance,", in Proceedings of the 24th IEEE International Conference on Source Code Analysis and Manipulation, pp. 224–234, 2024, https://doi.org/10.1109/SCAM63643.2024.00030

Toward Interactive Optimization of Source Code Differences: An Empirical Study of Its Performance

John Doe
Anonymous
Affiliation

***Abstract*—A source code difference (diff) indicates the changes made by comparing the new and old source code versions, which can be utilized in code reviews to help developers understand the changes. Although many diff generation methods have been proposed, existing automatic methods may generate nonoptimal diffs, which hinder reviewers from understanding the changes. In this paper, we propose an interactive approach for optimizing diffs. In this method, users can provide feedback for the points of a diff that should not be matched but are or parts that should be matched but are not. Based on this feedback, the editing graph is updated, allowing users to obtain more optimal diff. To investigate the potential of this approach, we simulated our method by applying search algorithms to empirically assess the number of feedback required and the amount of diff optimization resulting from feedback via the proposed method. Results via 23 GitHub projects show that 92% of nonoptimal diffs could be addressed with less than four feedback actions in the ideal case.**

***Index Terms*—source code difference, interactive, empirical study**

I. Introduction

Code review is a process in which developers manually inspect source code to identify defects and enhance the software's quality [1]. Because this process requires significant effort, modern code review is a typical way for code review, which focuses the inspection on the modified parts of the source code [2].

To properly conduct code reviews, understanding the changes is crucial [3]. Review tools display the changes made to the source code using line differences [2]. These line differences facilitate developers' understanding of the changes [3].

However, existing methods for generating line differences may not accurately identify changes in the source code [4]. Moreover, for a better understanding of changes, it is preferable to display differences that represent logically coherent changes [5], [6]. However, there may be instances where the grouping of change is not optimized. Nonoptimal differences can also impact tools that provide additional information based on line differences [4], [7].

To address the issue of unoptimized line differences, we propose an interactive optimization method. In this method, users review the output line differences and provide feedback where the change identification is deemed inappropriate. Subsequently, users can obtain more optimized differences by generating new line differences incorporating this feedback.

A significant divergence from other automated difference generation methods is that this method requires human manipulation. Therefore, to discuss the method's practicality, it is important to evaluate whether the effort required for feedback operations is realistic. However, direct quantification of effort with human subjects requires significant cost, which is beyond the scope of this paper. Rather, in this paper, we preliminary attempt to quantify the optimization performance in two aspects: the amount of feedback needed to obtain appropriate differences and the amount of improvement in differences via feedback. We formulated the process of repetitive feedback in the interactive optimization as a graph search problem and simulated it by applying a search algorithm. Source code changes from 23 projects hosted on GitHub were used as subjects for the study. The evaluation of feedback performance was conducted based on the simulation results. Results show that feedback was done on average 1.73 times for optimization in the ideal case, and 4.87 locations were optimized simultaneously with an instance of feedback. The amount of optimization by average feedback was 68% compared to the ideal case. These results indicate that both in ideal and average cases, this method can reduce the effort required to optimize differences, and it is meaningful to continue research towards practical application.

The main contributions of this paper are as follows:

- Proposal of an interactive optimization method for differences
- Empirical study of the potential performance of the interactive optimization method, revealing the optimization performances in ideal cases and in average cases.

The remainder of this paper is outlined as follows: Section II discusses the motivation for this study. Section III explains the method for generating feedback-based differences. Section IV describes the simulation method used for the empirical study. Section V discusses the results of the optimization performance investigation. Section VI summarizes related work on code differencing techniques. Finally, Section VII outlines the contents of this paper and future challenges.

II. Motivation

Code review is the process in which an individual other than the author manually inspects the source code to identify defects and improve the maintainability of software. The modern code review method reduces the workload by utilizing

```
1  public interface Block{                    1  public interface Block{
2      int getCount();                        
3      /**                                    2    /**
4       * Whether the values in this block are sorted
                                              3     * Gets the number of positions in the block
5       */                                   4     */
6      boolean isSorted();                    
                                              5    int getCount();
7      /**                                    6    /**
8       * Whether the block contains a single value
                                              7     * Gets the start and end positions of the block
9       */                                   8     */
10     boolean isSingleValue();               
11     Range getRange();                      9    Range getRange();
12 }                                          10 }
```

(a) Nonoptimal diff.

```
1  public interface Block{                    1  public interface Block{
                                              2    /**
                                              3     * Gets the number of positions in the block
                                              4     */
2      int getCount();                        5    int getCount();
3      /**
4       * Whether the values in this block are sorted
5       */
6      boolean isSorted();
7      /**
8       * Whether the block contains a single value
9       */
10     boolean isSingleValue();
                                              6    /**
                                              7     * Gets the start and end positions of the block
                                              8     */
11     Range getRange();                      9    Range getRange();
12 }                                          10 }
```

(b) Optimal diff.

Fig. 1. Two diffs from the same source code pair.

tools and focusing the inspection on the changes made to the source code [2]. Code review is conducted in numerous organizations, and several empirical studies have demonstrated its effectiveness in enhancing code quality [1], [3], [8].

The greatest challenge in code reviews is understanding the changes [2]. Accurately comprehending changes leads to more effective code reviews [3]. Numerous code review support tools offer source code difference (diff), which developers use to understand modifications. For instance, GitHub implements line diff using Myers' algorithm [9].

However, diffs generated by existing methods may not always be optimal for the user. Even though the diffs correctly describe the changes in calculations, there are instances where they do not accurately reflect the modifications made by developers [4]. Moreover, to effectively understand changes, concise and logically cohesive diffs are preferable [5], [6]. However, in nonoptimal diffs, cohesion may be compromised.

Figure 1 shows different source code diffs for the same changes: the removal of two methods and the addition of comments to the remaining methods. Figure 1a depicts the diff generated by the Myers' algorithm [9], a widely used method, whereas Fig. 1b represents the optimal diff for expressing the changes. Nonoptiminal diffs are unintelligible and disregard the intent of changes. For example, because the developer has not modified the *getCount* method, as illustrated in Fig. 1b, the method should correspond between the two revisions; however, it does not in Fig. 1a. Furthermore, comments were added to the new version's getCount and getRange methods. However, the diff in Fig. 1a correlates the comments between the new and old versions and thus fails to reflect the changes made by the developer accurately. Existing methods for generating diff can produce such confusing diffs. In fact, not only Myers but also other line differencing approaches, e.g., Histogram, cannot generate the diff shown in Fig. 1b.

One might think that code diffs are just outcomes of automated differencing tools and not the target of manual optimization. However, we believe that developers often meet a situation where the correction of diffs is useful. For example, developers submit a source code diff to a mailing list or an issue tracking system of an open-source software project as a patch [10]. Patches are the targets of the reviews from other developers. Reducing the understandability of a diff by a patch developer may contribute to reducing the time spent by multiple reviewers. In this context, it is important that the effort of improving understandability by correcting a diff be low enough. It might be hard or time-consuming to force developers to provide the correct mapping of elements when correcting a diff. Therefore, a technique is useful for developers to provide a diff by an input that is easy to prepare without specifying the desired outcome itself.

III. Interactive Optimization

This paper proposes an interactive optimization for source code differences (diffs) as a solution to a problem that is difficult to solve with existing automatic diff generation.

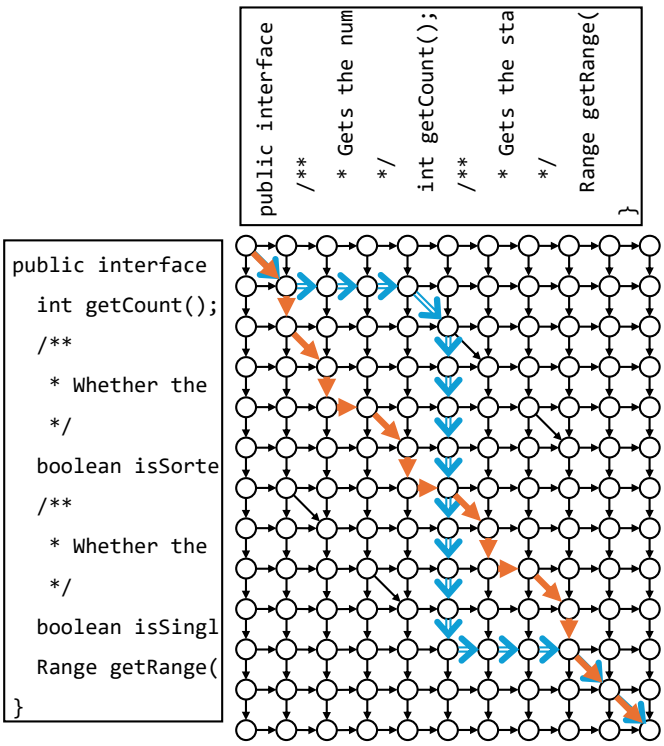


Fig. 2. Example of an edit graph.

A. Preliminary: Edit Graph and Difference

An *edit graph* [11], [12][1] is an oriented graph $G = (V, E)$ consisting of sets of nodes V and edges $E \subset V \times V$ where

$$
\begin{aligned}
V &= \{v_j^i \mid 0 \le i \le N \wedge 0 \le j \le M\}, \\
E &= \{\langle v_j^{i-1}, v_j^i \rangle \mid 1 \le i \le N \wedge 0 \le j \le M\} \cup \\
&\quad \{\langle v_{j-1}^i, v_j^i \rangle \mid 0 \le i \le N \wedge 1 \le j \le M\} \cup \\
&\quad \{\langle v_{j-1}^{i-1}, v_j^i \rangle) \mid 1 \le i \le N \wedge 1 \le j \le M \wedge eq(i, j)\}.
\end{aligned}
$$

Here, N and M respectively denote the numbers of lines of code in the old and new versions of source code. A node v_j^i expresses a state that i-th and j-th lines of old and new versions of source code are read, respectively. Edges are in between adjacent nodes. We use three types of edges: *vertical*, *horizontal*, and *diagonal*. A horizontal edge $\langle v_j^{i-1}, v_j^i \rangle$ expresses that i-th line of the old version of code was not associated with the lines in the new version and skipped (regarded as *deleted*). A vertical edge $\langle v_{j-1}^i, v_j^i \rangle$ expresses that j-th line of a new version of the code was not associated with the lines in the old version and skipped (regarded as *added*). A diagonal edge $\langle v_{j-1}^{i-1}, v_j^i \rangle$ expresses that i-th line in the old version was associated with j-th line in the new version. The predicate $eq(i, j)$ holds if and only if i-th line in the old version is the same as j-th line in the new version. An example of an edit graph is shown in Fig. 2, comparing the two versions of source code shown in Fig. 1.

[1]The definition in this paper does not strictly follow the ones in these references, but it is an essentially equivalent one.

A *path* from the node v_0^0 to the node v_M^N expresses a mapping between lines in the old and new versions, i.e., a *diff*. A diff $d \in D\ (\subseteq 2^E)$ can also be represented as a set of edges used to form the path. We typically use one of the shortest paths as a preferable difference. A set of bold edges in Fig. 2 represents the shortest path in the graph, which corresponds to the difference shown in Fig. 1a. The distance of this shortest path is 15. The path includes five horizontal edges, which means that the difference includes the deletion of five lines. Calculating the shortest paths on an edit graph is equivalent to calculating the *longest common sequence* (LCS) of lines of code in the old and new versions. If more than one shortest path is found, one of them is picked.

Many algorithms to calculate LCS between lines are proposed. For example, a simple algorithm based on dynamic programming works under $O(NM)$. More efficient algorithms are also proposed, e.g., $O(nd)$ [12].

B. Basic idea

The basic idea of the proposed technique is simple; users modify the given edit graph based on their feedback. The usage process of the proposed technique is shown in Fig. 3. In our technique, the user first obtains a path, i.e., diff, between old and new versions using an automated differencing technique. Next, the user reads the diff and provides feedback indicating its unfavorable points, i.e., the points that do not match with his/her intention. In this figure, the lines A and B were correctly associated, but the line C was not associated and was regarded as the deletion and the insertion. When the user thinks that the associations of A and B are inappropriate and points out these inappropriate associations, the edges in the edit graph are updated, and the user obtains an updated diff based on the updated graph. The thin solid edges in the graph represent the shortest path in the first analysis whereas the thick edges represent the shortest path after the correction. By updating the edit graph, the new shortest path follows, avoiding the use of the removed edges by the update. Thus, it includes the mapping of the line C rather than the lines A and B. The updated shortest path is automatically calculated based on the feedback using the same differencing algorithm.

The proposed approach is designed to give users feedback on areas of dissatisfaction with the generated diff, rather than giving them feedback about the information on how the ideal diff will be. This approach allows users who do not necessarily have the ideal diff in their mind to reach the ideal one by simply expressing their dissatisfaction with the current result.

C. Updating Edit Graph

We provide three types of feedback actions: *mismatch*, *old-orphan*, and *new-orphan*. Feedback actions can be expressed as a pair of two indices with a special wildcard symbol: $a \in A\ (\subset (\mathbb{N}^+ \cup \{*\})^2)$.

Mismatched lines. Consider a situation that j-th line in the new version was associated with i-th line in the old version (See Fig. 4a). If the user believes that this situation is inappropriate, he/she indicates that they were undesired

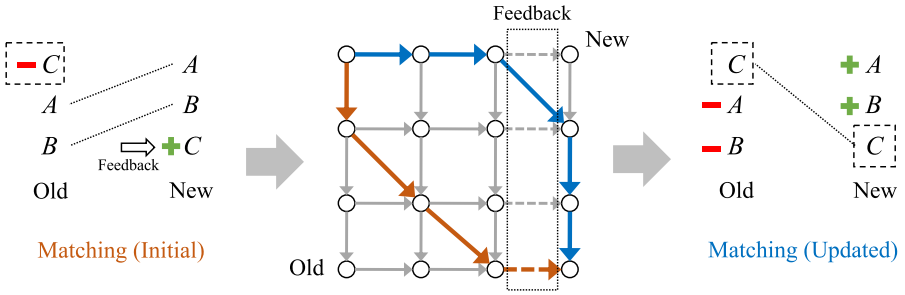
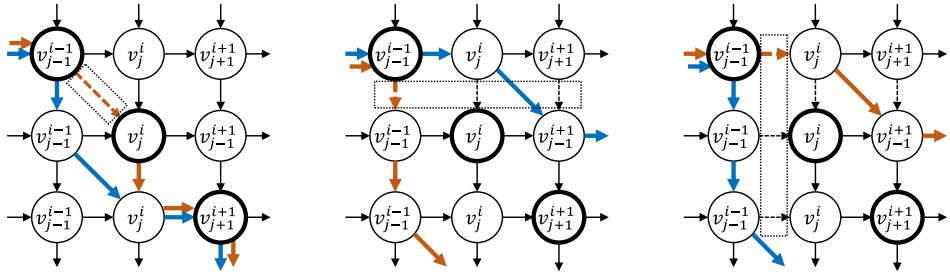


Fig. 3. Overview of the interactive optimization.



(a) Mismatch. (b) Old-orhphan. (c) New-orphan.

Fig. 4. Types of feedback actions.

mismatched, providing a feedback action of $(i, j) \in A$. The user desires either of the following cases for the target line in the new version: 1) another mapping to another line in the old version, or 2) no mapping to the lines in the old version, and it should be regarded as being inserted. Therefore on this, we update the graph by removing the indicated diagonal edge:

$$E \leftarrow E \setminus \{\langle v^{i-1}_{j-1}, v^i_j \rangle\}.$$

In Fig. 4a, due to this removal of the diagonal edge, the original orange-colored path becomes infeasible, and the new blue-colored path is newly obtained by running the automated differencing algorithm again.

Old-orphan lines. Consider a situation that i-th line in the old version was not associated with any line in the new version (See Fig. 4b). If the user believes that this situation is inappropriate, he/she indicates that this was an undesired *old-orphan* line, providing a feedback action of $(i, *) \in A$. The user should consider the disagreement of not only the current mapping but also the any mappings that the old target line is not associated with any of the lines in the new version. Therefore, we update the graph by removing all the horizontal edges that i-th line is regarded as deleted, as follows:

$$E \leftarrow E \setminus \{\langle v^{i-1}_j, v^i_j \rangle \mid 1 \leq j \leq M\}.$$

New-orphan lines. Consider a situation that j-th line in the new version was not associated to any line in the old version (See Fig. 4c). If the user believes that this situation is inappropriate, he/she indicates that this was an undesired *new-orphan* line, providing a feedback action of $(*, j) \in A$. The user should consider the disagreement of not only the current mapping but also the mappings that the target new line is not associated with any of the lines in the old version. Therefore, we update the graph by removing all the vertical edges that j-th line is regarded as inserted, as follows:

$$E \leftarrow E \setminus \{\langle v^i_{j-1}, v^i_j \rangle \mid 1 \leq i \leq N\}.$$

The interactive optimization mechanism can be implemented through a very simple user interface that allows users to click on a line of the displayed diff. Any line on a diff, which can be added, deleted, or matched in the two versions, corresponds to an edge on the edit graph to be used to generate the diff. Therefore, clicking on a line on the diff specifies a specific edge in the graph, with the intention of the dissatisfaction with the treatment of the specified line. Then, a feedback action can be generated from the specified edge to update the diff. More precisely, the following *action*: $E \to A$ provides a feedback action from the specified edge:

$$action(\langle v^{i'}_{j'}, v^i_j \rangle) = \begin{cases} (i, j) & \text{if } i' \neq i \wedge j' \neq j \text{ (diagonal)}, \\ (i, *) & \text{if } i' = i \text{ (horizontal)}, \\ (*, j) & \text{if } j' = j \text{ (vertical)}. \end{cases}$$

After applying the above corrections, we calculate the paths again using the updated edit graph and obtain the new diff based on the calculated paths.

D. Benefits and Weaknesses

We consider that the proposed technique has several benefits and weaknesses:

- **Benefit: Simplicity and understandability.** The principle idea, i.e., changing the edges to be used in the based differencing technique according to the feedback actions,

is simple. It is easy for users to understand the effect of the optimization.

- **Benefit: Use of line-based difference.** On the one hand, the proposed technique can be applied to the line-based differencing techniques. We believe that this leads to wide acceptance of the obtained outcome for practitioners. Also, since line-based differencing does not require code parsing, it has high interoperability; it can accept the source code including syntax errors or those written in new programming languages. Almost accurate differencing algorithms [13]–[18] require structural inputs. Syntax errors of the given texts or lacks of parsers may cause the failure of the application of differencing.
- **Benefit: High applicability.** On the other hand, our technique can be applied to not only a sequence of lines but also that of any elements in principle. For example, we can apply an accurate differencing algorithm named Semantic Diff [19] for a sequence of program statements, and our technique could be combined with it.
- **Weakness: Bound to the restrictions of based differencing technique.** The proposed technique uses the LCS-based differencing technique as its basis. Therefore, it inherits some of the weaknesses of the based technique. For example, the basic line-based approach only deals with deletion and insertion as its primitive edit script, and does not deal with the *move* or the *copy* of code fragments. This weakness also appears when large changes are applied to versions because such changes are sometimes inappropriate to represent just deleted and inserted elements.

IV. Simulation Methodology

A. Objective

Interactive diff optimization is helpful because it allows users to update diff as intended. However, unlike existing automatic methods, this method requires user feedback. Therefore, to discuss the practicality, not only the quality of diffs generated by the method should be considered, but also the effort required to optimize diff.

Although it is necessary to conduct a developer study to directly quantify the effort, we have not even yet had a grasp of the fundamental properties of the interactive optimization approach. Therefore, as a preliminary, this paper investigates the capability to optimize the diff of the proposed method using two metrics. The first metric measures the number of feedback actions needed to obtain optimal diff. Moreover, in interactive diff optimization, one feedback instance could correct multiple nonoptimal locations simultaneously. The second metric is the number of these simultaneously corrected spots. If one feedback action optimizes more inappropriate locations, it can be expected that the overall effort to optimize diff will be more minor.

We can obtain the two metrics mentioned above by mechanically simulating the interactive optimization process. Instead of efforts that are costly to quantify, we will concretely investigate these easily quantifiable optimization performances. For the investigation, we answer two research questions concerning different feedback strategies.

***RQ*$_1$: What is the optimization performance in the ideal case?** The first strategy is where users provide ideal feedback and minimize the number of iterations. The objective in this case is to measure the highest performance of interactive diffs optimization methods. This study provides information on how much the required effort for feedback can be reduced with this method, making it a meaningful stepping stone for discussions.

***RQ*$_2$: What is the optimization performance in average cases?** When considering the scenarios in which users utilize the tool, it is only sometimes guaranteed that they can give ideal feedback. Therefore, it is necessary to consider various feedback cases to evaluate the effort required for this method more practically. The average cases of feedback are investigated to examine these variations.

B. Basic Design

In the original interactive method, users optimize a diff by checking it and repeatedly providing feedback on nonoptimal parts. This repetitive process was formulated as a search problem for empirical study, enabling it to be mechanically simulated.

Figure 5 shows an overview of the simulation methodology. This figure represents the process from the initial diff shown in Fig. 1a, iterating through feedback until obtaining the diff shown in Fig. 1b. This simulation is divided into two major parts. One part simulates the flow from checking the diff to providing feedback, as shown on the upper side of the figure. The other part simulates various patterns of providing feedback, as shown on the lower side of the figure.

The following subsections of this section will explain the elements necessary for formulating search problems, using the optimization from Figure 1a to Fig. 1b as an example.

C. Diff Generation Reflecting Feedback

In the simulation, the process of generating diff was simplified. Rather than being incrementally updated via interactive feedback in the editing graph, it was updated through multiple simultaneous feedback per diff. Users can recreate the same diff by reproducing the simultaneous feedback in sequence. Therefore, this simplification does not affect the outcome of the investigation.

Figure 6 demonstrates an example where the simulator provides feedback simultaneously for the removal of `getCount` on Line 2 of the old version and the matching of `/**` on Lines 3 and 6 of the old and new versions, respectively. Based on this feedback, the corresponding edges in the edit graph are removed. Consequently, a new diff produced differs from the diff shown in Fig. 1a before the feedback is given.

We formulated the process of generating a diff reflecting feedback as a feedback-aware differencer function $diff_{fix}\colon 2^A \to D$. This function receives zero or more feedback actions and returns a path (a diff). The feedback actions passed to the function in Fig. 6 are $\{(2,*),(3,6)\}$. Due to the feedback, the dashed edges on the editing graph are removed.

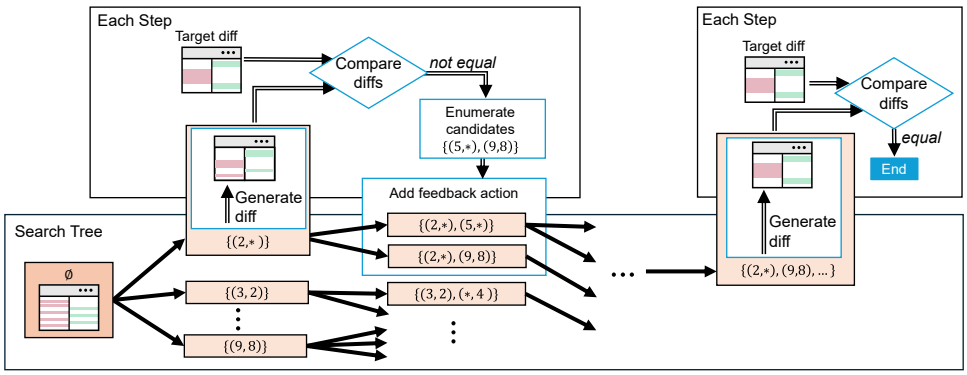


Fig. 5. Overview of the automated simulation.

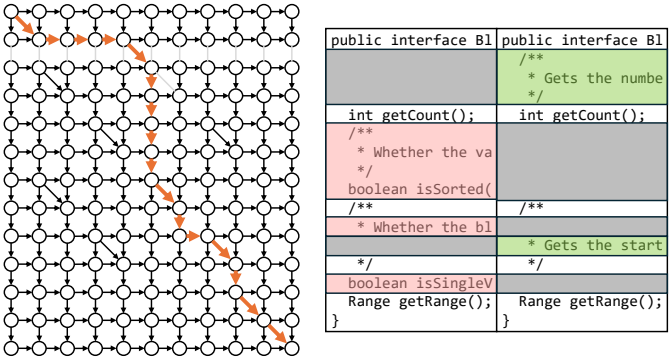


Fig. 6. Diff generation reflecting the feedback.

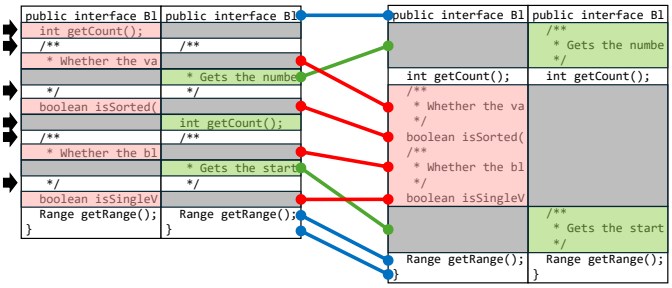


Fig. 7. Generation of feedback action candidates.

A differencer searches for the shortest path on this editing graph, and the path represented by the thick arrow is the value of $diff_{fix}(\{(2,*),(3,6)\})$.

D. Target Diff

In formulating the search problem, it is necessary to define the goal of the search. Therefore, in simulations, a different diff from the original one is predetermined, and the goal is set to generate a diff identical to this through repeated feedback. This is referred to as the target diff, denoted by $d^* \in D$.

Figure 1b represents the target diff. By defining the target diff, it is possible to mechanically imitate the human process of repeatedly obtaining diff that matches Fig. 1b from the diff shown in Fig. 1a, which are without feedback.

E. Generation of Feedback Candidates

To conduct the simulation, we describe a method that mechanically calculates feedback actions to be provided for the generated current diff. Parts needing feedback are where edit scripts differed when comparing the current and target diff. The simulator creates feedback that can correct these points. Because such feedback often exists in multiples, the simulator calculates a set of possible feedback candidates to provide.

Figure 7 shows the comparison results between the diff in Fig. 1a and the target diff in Fig. 1b. Same edit scripts, such as the match between Line 1 of the old version and Line 1 of the new version with `public interface blo...`, are connected by lines in both diff. On the other hand, six edit scripts indicated by arrows in the current diff do not share any edit scripts with the target diff. Therefore, feedback actions that are applied to these six points become the next candidates to be performed.

Let a diff being compared be $d \in D$ and the target diff be $d^* \in D$. We defined feedback candidate function $c\colon D \times D \to 2^A$ that returns a set of feedback to be provided by comparing the two diffs, as follows.

$$c(d; d^*) = \{action(e) \mid e \in d\} \setminus \{action(e) \mid e \in d^*\}.$$

Furthermore, the number of feedback candidates $|c(d; d^*)|$ represents *distance* between the current diff d and the target diff d^*. So, a diff of a shorter distance with the target diff is similar to the target diff. In contrast, a diff of a longer distance with the target diff differs in many parts from the target one.

F. Search Problem

1) Formulation of Search Problem: For the simulation, we described the interactive diff optimization as a search problem. In each problem, an edit graph (V, E), a universal set of possible feedback actions on the edit graph A, and a target diff d^* are given. Based on the conditions provided and the preparations described in this section, we explain definitions of the search state s, the initial state s_0, the goal condition g, and the transition function δ.

Search State. Each state in the search problem is expressed as a set of feedback actions to be given: $s \in S\ (\subseteq 2^A)$. In Fig. 5,

each rectangle within the search tree represents a search state. A search state represents a model of diff based on feedback in interactive diff optimization.

We did not use a diff itself as a search state because a different set of feedback actions can generate the same diffs. Furthermore, when previous feedback actions are different, even if the same feedback is given for the same diff, the resulting diffs may vary. Therefore, we use a set of feedback actions as a search state.

Initial State. The initial state s_0 was defined as the state where the set of feedback actions is empty, specifically, $s_0 = \emptyset$. The initial state models the first diff output in actual use. If the set is empty, there are no changes in the edit graph, and the generated diff is the same. Therefore, it is rational to consider the empty set as the initial state. In Fig. 5, the initial state is positioned at the left as the root of the search tree.

Goal Condition. We defined the goal condition $g\colon S \to \{\top, \bot\}$. It is a predicate that takes a search state and returns true if and only if the feedback-reflected diff matches the target diff. The goal condition was formulated using the feedback-aware differencer function $\mathit{diff}_{\mathit{fix}}$, the search state s, and the target diff d^* as follows:

$$g(s) = \mathit{diff}_{\mathit{fix}}(s) \equiv d^*.$$

If the state s satisfies $g(s) = \top$, then the simulator outputs that search state and terminates the search. This corresponds to a use case where a user can obtain an optimal diff through interactive feedback s.

Transition Function. The actions applicable for each state are adding a feedback action. Therefore, action a can be expressed as feedback action, where $a \in A$. To bring the state s closer to the target diff d^*, we defined a function $\mathit{succ}\colon S \to 2^A$ that returns the set of applicable actions. This can be described using the feedback-aware differencer function $\mathit{diff}_{\mathit{fix}}$ and the feedback candidate function c as follows:

$$\mathit{succ}(s) = c(\mathit{diff}_{\mathit{fix}}(s); d^*).$$

Then, the transition function $\delta\colon S \times A \to S$, given a state s and an action a, generates the next state as follows:

$$\delta(s, a) = s \cup \{a\}.$$

This represents one iteration of the interactive method where users provide feedback on the nonoptimal points of the diff.

2) Formulation of the Search Space: Based on the formulation above, the search space S was defined. First, the initial state is included in the search space, i.e., $s_0 \in S$.

The upper side of Fig. 5 represents the flow where child states of a state $s \in S$ are added to the search space. First, a diff $\mathit{diff}_{\mathit{fix}}(s)$ is generated based on feedback. Then, goal condition $g(s)$ is checked. If true, the search ends there, so there are no child states; if false, the creation of child states continues. Based on the diff, the possible set of actions $\mathit{succ}(s)$ is calculated. Lastly, new states can be created from the state and the action. These are also elements of the search space:

$$s \in S \Rightarrow g(s) \ \vee \ \forall a(a \in \mathit{succ}(s) \Rightarrow \delta(s, a) \in S).$$

Based on the initial state and this addition process, the search space was defined.

The example in Fig. 5 demonstrates the process of adding child states of the state $\{(2, *)\}$. Differencer generates diff $\mathit{diff}_{\mathit{fix}}(\{(2, *)\})$, but it differs from the target diff d^*. Consequently, the child states $\{(2, *), (5, *)\}$ and $\{(2, *), (9, 8)\}$ are added to the search tree.

V. Empirical Study

In the format of the search problems written in the previous section, we simulated interactive diff optimization. Based on the simulation, we empirically investigated the optimization performance of the proposed method. For the empirical study, we considered two types of feedback strategies. One is the case of providing ideal feedback, and the other is the case of providing random feedback. To evaluate the optimization performance for these two feedback strategies, corresponding *RQ*s were established.

A. Data Collection

Manual preparation of target diffs was challenging because this investigation requires many source code changes and corresponding target diffs. Therefore, in the empirical investigation, we created the target diffs by Histogram algorithm. In interactive diff optimization, diffs are generated based on the shortest path search of edit graphs. The concept prioritized in the Histogram algorithm differs from this, so diffs are also dissimilar within a practical range. As such, this allowed for efficient preparation of the required target diffs that differed from ones without feedback.

Previous research [4] conducted a study on diffs generated by the histogram algorithm. They have published source code changes where the diff generated by Histogram and Myers algorithms differ, as artifacts from 24 GitHub projects. In this study, we preliminarily investigated whether diffs generated by Histogram algorithm differed from those generated by our differencer on this dataset and used the changes that created different diffs as the dataset.

Furthermore, to limit the size of the search problem and the computation time, we filtered for problems where lines of code were 3,000 or less in both the new and old versions and the number of feedback candidates in the initial state was 30 or less. Also, the same changes only about a license update existed in more than 700 files, so these files were excluded from the dataset. As a result, we used 9,229 source code changes from 23 projects that met the above criteria as our data set. Table I summarizes this 9,229 dataset for the lines of code in the new and old versions, the number of changed lines, and the number of initial feedback candidates.

Additionally, when conducting simulations, as a preprocessing step for the source code, lines consisting solely of newlines were removed. It is unlikely for users to optimize for blank lines because information about how the blank line is changed is rarely useful to the user in understanding the change. However, states do not satisfy the goal condition until they fully match the target diff in the simulation, so

TABLE I
ATTRIBUTE OF DATASET

	Min	Q1	Q2	Q3	Max	average
LOC in the old version	8	138	285	618	2,929	493.73
LOC in the new version	3	153	310	654	2,993	515.34
# changed lines	3	55	111	223	3,779	182.29
# initial feedback candidates	2	2	5	11	30	7.78

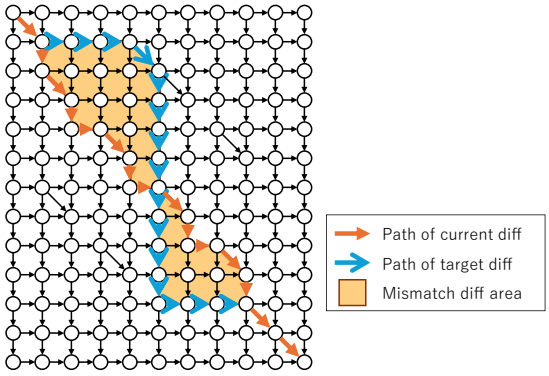


Fig. 8. Mismatch diff area.

there was a lot of feedback on optimizing blank lines. If such cases were included in the study, the results would be impractical. Therefore, eliminating unnecessary feedback related to blank lines was necessary to obtain more practical research outcomes.

B. RQ_1: What is the optimization performance in the ideal case?

1) Motivation: The case in which the proposed method's optimization performance is highest occurs when ideal feedback is provided; that is, when the number of feedback is minimized. This case demonstrates the most significant benefit of interactive diff optimization, and investigating these results is meaningful in considering the effort involved in this interactive method. Therefore, in RQ_1, we investigated the optimization performance of feedback in the ideal case and the factors that influence it.

2) Study Design: The simulator applied the A* algorithm to a search problem that imitated diff optimization and sought a state that satisfied the goal condition with the fewest number of feedback actions. The subject of this study was 9,229 source code changes in the dataset, with a maximum execution time limit of 30 minutes.

The A* search requires a heuristic function, for which we defined mismatch diff areas. These are continuous areas on the edit graph, like the two yellow areas in Fig. 8, enclosed by the current and target diffs. The number of feedback instanceequals required to optimize diff must equal or exceed the number of mismatch diff areas. In the example shown in the figure, users must provide feedback at least twice. This fact can be stated for the following reasons. Edges on different mismatch diff areas are not removed in a feedback action. This is because edges removed in feedback are either single or multiple edges parallel to the vertical or horizontal,

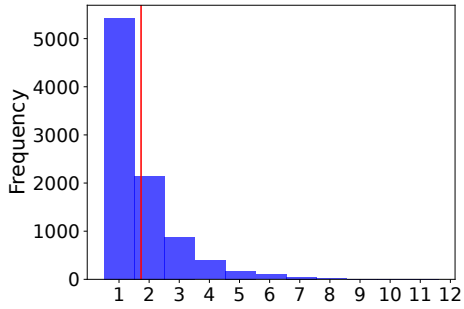


Fig. 9. Distribution of the number of feedback required.

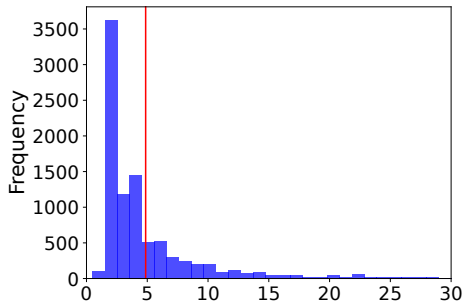


Fig. 10. Distribution of the amount of optimization on average per feedback.

while the mismatched diff areas are arranged diagonally. If part of the current diff and the target diff share the same start and end nodes, the current diff always takes precedence. This is because the differencer always prioritizes when the starting and ending nodes of two paths are the same to prevent variations in the results of each simulation, and the prioritized path is the current diff. Therefore, each area must have its edges removed through different feedback actions to obtain the target diff. The number of areas acts as a heuristic function $h\colon S \to \mathbb{N}$.

$$h(s) = |\{v^i_j \mid 0 \le i, i', i'' \le N \land 0 \le j, j', j'' \le M \land \langle v^{i'}_{j'}, v^i_j \rangle \in \mathit{diff}_{fix}(s) \land \langle v^{i''}_{j''}, v^i_j \rangle \in d^* \land (i' \neq i'' \lor j' \neq j'')\}|$$

3) Results and Discussion: Excluding 44 changes that failed due to exceeding the time limit or insufficient memory, 9,185 entries were used. Figure 9 shows the minimum number of feedback required for diff optimization. Figure 10 represents the *optimized amount per feedback*, which represents the impact of single feedback action, and it is computed as the distance of the initial diff to the target diff divided by the minimum number of feedback actions. The red lines in each graph represent the average values of the data.

In Fig. 9, the average number of feedback actions required for optimization was 1.73. In 59% of the dataset, only one feedback action made the diffs optimal. In 92% of cases, the diffs were optimized with three or fewer feedback actions. Thus, the effort required to optimize diff is considerably tiny in ideal cases.

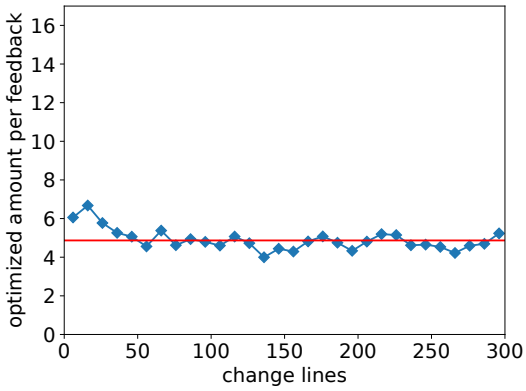


Fig. 11. optimized amount for each feedback against the number of changed lines

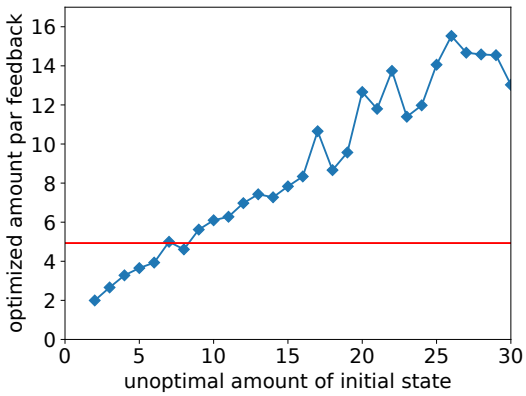


Fig. 12. Optimized amount for each feedback against the number of initial feedback candidates

The average number of optimizations per feedback was approximately 4.87 locations. The mode was 2 in Fig. 10 because the most common pattern among nonoptimal diffs was the case where there were two feedback candidates. This was addressed in a feedback action; hence, the value of 2 is prominently numerous in the graph. Since values greater than one are widely distributed in the graph, it is possible to perform multiple optimizations simultaneously in an ideal case, and interactive methods can efficiently optimize diffs.

As shown in Fig. 11, although the number of lines changed differs, the amount of optimization per feedback does not vary. Therefore, according to feedback, the number of changed lines isn't relevant to the optimization performance.

Figure 12 shows the impact of the number of nonoptimal parts in the initial diff on the optimization performance. If the difference between the initial and target diff is significant, more effort is required to optimize the diff. When there was significant room for optimization in the first diff, the amount corrected simultaneously through feedback was also substantial. Therefore, it was observed that an increase in the number of feedback due to the large number of nonoptimal parts could be suppressed, and such a diff required less effort.

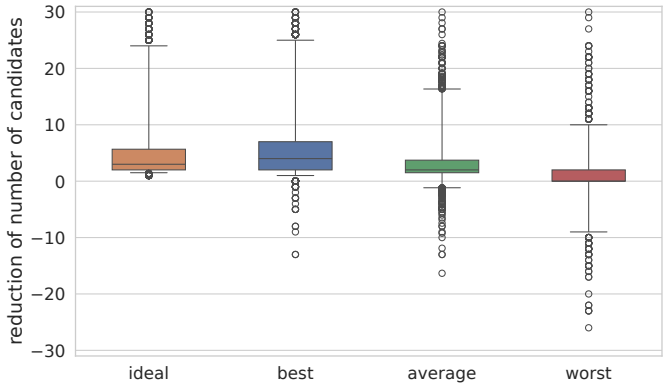


Fig. 13. Optimization amounts for each feedback pattern.

TABLE II
AVERAGE AND RATIO TO IDEAL FOR EACH TYPE OF FEEDBACK IN OPTIMIZATION AMOUNT

	Ideal	Best	Average	Worst
Amount of Optimization	4.87	5.67	3.14	0.71
Ratio to Ideal Feedback	1	1.14	0.68	0.22

The interactive optimization approach has shown its potential to optimize a diff with minimal effort in ideal cases. In 92% of cases, three or less times of feedback actions were required to optimize diffs. Additionally, 4.87 lines were correctly fixed per a feedback action on average.

C. RQ_2: What is the optimization performance in average cases?

1) Motivation: When users optimize diffs, they can only sometimes provide ideal feedback. To evaluate the performance of interactive diff optimization, it is reasonable to consider not only the ideal cases but also the general ones.

Since RQ_1 was an investigation of the case with the minimum number of indications, such typical cases were not included in the results. In contrast, RQ_2 evaluated the diffs when various types of feedback were provided, and the results included multiple cases.

2) Study Design: We investigated how each feedback action in the initial state optimized the diff. BFS was performed on the search problem, and the logs of all states at depth one were produced to measure the diffs. From there, the average, best, and worst values were recorded as representative values for each diff optimization case. Code changes for which results were not obtained in RQ_1 were also excluded from the results of this research question.

3) Results and Discussion: Fig. 13 shows the optimized amount, the reduction in the number of feedback candidates of diffs before and after feedback, for each category and the optimized amount for the ideal case is added for reference. Table II shows the average per category and the average ratio compared to the ideal. The whiskers of the box plot are set to the lower 1% and upper 99%. The ideal is represented as a box plot in Fig. 10.

In the best-case scenario, the distribution is longer above than in the ideal case, and the ratio exceeds 1. This is because when the number of feedback instances is more than one, there is a mixture of feedback that optimizes the diff more and feedback that optimizes them less.

In the best case scenario, the average optimization amount is 5.67, whereas in the average case, it dropped to about 3.14, approximately 55%. The ratio between the average and the ideal cases averaged 68%. Therefore, not all feedback is equally performed, and on average, the optimization performance is inferior to that in ideal scenarios.

The parts of the figure below zero represent cases where the diff obtained by feedback was nonoptimal compared to the previous ones. 15% of the worst-case patterns fall into this category, and it was 2% for the average patterns. Therefore, while the diffs can worsen due to feedback, such situations rarely occur on average.

> The feedback in average cases had lower optimization performance than the ideal case. However, since the optimization amount is over 3, it could reduce the effort on average.

D. Threats to Validity

For filtering the dataset, RQ_1 might show different results from those of an unfiltered dataset. This is because feedback candidates with a diff greater than 30 are filtered out, and including them would increase both the number of feedback instances and the amount of optimization per feedback. However, attempting to include these would result in more simulation failures within the dataset, leading to unreliable research results, which is why the current experimental setup was adopted.

In RQ_2, instead of examining every feedback within the search problem, we focused only on feedback made from the initial state. In interactive diff optimization, the user checks the current diff and feedback; there is no need to worry about the past. Therefore, examining the feedback that can be given in the initial state is sufficient to evaluate the performance.

The target diffs using the Histogram algorithm were mechanically generated in the empirical study. Thus, they included diffs that were nonoptimal. This investigation aims to demonstrate the extent to which feedback affects the diffs quantitatively, and the quality of the diffs is not a concern.

VI. Related Work

A lot of differencing algorithms and tools are proposed. The most basic one is a line-based approach. Also, some of them use the structural information such as XML [17]. JDiff [14] uses the structural information of program source code, whereas Change Distiller [16] uses abstract syntax trees (AST). GumTree [18] is also an efficient differencing algorithm for source code based on the matching on ASTs.

Semantic Diff [19] applies LCS-based differencing to sequences of program statements. This is useful both for efficiency and accuracy. However, of course, it produces incorrect mappings so that the approach of correcting difference is still effective.

Consideration of the correction or adaptation of code diffs is not novel; some researchers have already discussed it [20], [21]. The main difference of ours from them is that our technique focuses on interactive correction using a semi-automated tool that enabling developers to update the diffs without specifying the ideal mapping.

Constrained longest common subsequence (C-LCS) is a sub-category of LCS having constraints. One example of the constraints is that the resulting LCS should include the given particular subsequence [22], [23]. The corrections of an edit graph in the proposed technique can also be regarded as constraints. However, these work do not mention about the application to the computation of source code diffs.

VII. Conclusion

In this study, we proposed an interactive optimization of source code differences, a method for creating optimal differences crucial for understanding changes during code reviews, an essential process in software development. Since the proposed method requires human feedback, it was necessary to investigate the effort to evaluate practicality. To clarify the effort, we investigated the optimization performance of the feedback. In this investigation, the interactive process of the differences was formulated and simulated as a search problem. As a result, in the ideal case, feedback was needed on average 1.73 times. Furthermore, an average of 4.87 editing operations were corrected simultaneously with one feedback instance. The average optimization amount of feedback was about 68% compared to the ideal feedback. Supplemental material, including the experimental results, is available [24].

Future challenges can be listed up as follows:

Simulation of More Diverse Data. In this study, differential generation in the simulation employs dynamic programming. As a result, differential generation was time-consuming, and to address RQ_1 and reduce error-prone data, it was necessary to impose filter conditions. Faster difference generation and improved exploration efficiency could enable the simulation of more complex issues. It is anticipated that using a broader range of survey results will enhance the comprehensiveness of examining correction performance in interactive optimization.

Manual Assessment of Results. Information on the feedback provided and the resulting difference improvements can be obtained through the simulation. In the simulation, humans can check and evaluate ideal or low-performance feedback. It is conceivable to improve the method so that humans can optimize more through natural feedback.

Optimization across Different Granularities. The current method generates and optimizes the differences between source code elements line-by-line. As a future extension of the proposed method, it can optimize differences at granularities other than lines, such as at the token or syntactic levels. Such extensions will enable more appropriate differences that better reflect the reviewer's intentions.

REFERENCES

[1] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "An empirical study of the impact of modern code review practices on software quality," *Empirical Software Engineering*, vol. 21, pp. 2146–2189, 2016.

[2] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *Proceedings of the 35th International Conference on Software Engineering*, 2013, pp. 712–721.

[3] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim, "How do software engineers understand code changes? an exploratory study in industry," in *Proceedings of the 20th International Symposium on the Foundations of Software Engineering*, 2012, pp. 1–11.

[4] Y. S. Nugroho, H. Hata, and K. Matsumoto, "How different are different diff algorithms in Git?: Use `--histogram` for code changes," *Empirical Software Engineering*, vol. 25, pp. 790–823, 2020.

[5] A. Ram, A. A. Sawant, M. Castelluccio, and A. Bacchelli, "What makes a code change easier to review: an empirical investigation on code change reviewability," in *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, p. 201–212.

[6] M. Barnett, C. Bird, J. Brunet, and S. K. Lahiri, "Helping developers help themselves: Automatic decomposition of code review changesets," in *Proceedings of the 37th International Conference on Software Engineering*, vol. 1, 2015, pp. 134–144.

[7] G. Canfora, L. Cerulo, and M. Di Penta, "Ldiff: An enhanced line differencing tool," in *Proceedings of the 31st International Conference on Software Engineering*, 2009, pp. 595–598.

[8] C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli, "Modern code review: A case study at Google," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, 2018, pp. 181–190.

[9] E. W. Myers, "An O(ND) Difference Algorithm and Its Variations," *Algorithmica*, vol. 1, no. 1, pp. 251–266, 1986.

[10] P. C. Rigby, D. M. German, and M.-A. Storey, "Open source software peer review practices: A case study of the apache server," in *Proceedings of the 30th Internatiol Conference on Software Engineering*, 2008, pp. 541–550.

[11] W. Miller and E. W. Myers, "A file comparison program," *Software: Practice and Experience*, vol. 15, no. 11, pp. 1025–1040, 1985.

[12] E. W. Myers, "An O(ND) difference algorithm and its variations," *Algorithmica*, vol. 1, pp. 251–266, 1986.

[13] R. Al-Ekram, A. Adma, and O. Baysal, "diffX: An algorithm to detect changes in multi-version XML documents," in *Proceedings of the 2005 Conference of the Centre for Advanced Studies on Collaborative Research*, 2005, pp. 1–11.

[14] T. Apiwattanapong, A. Orso, and M. J. Harrold, "A differencing algorithm for object-oriented programs," in *Proceedings of the 19th IEEE International Conference on Automated Software Engineering*, 2004, pp. 2–13.

[15] G. Cobena, S. Abiteboul, and A. Marian, "Detecting changes in XML documents," in *Proc. 18th International Conference on Data Engineering*, 2002, pp. 41–52.

[16] B. Fluri, M. Wuersch, M. Pinzger, and H. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Transaction on Software Engineering*, vol. 33, pp. 725–743, 2007.

[17] Y. Wang, D. J.DeWitt, and J.-Y. Cai, "X-Diff: An effective change detection algorithm for XML documents," in *Proc. 19th International Conference on Data Engineering*, 2003, pp. 519–530.

[18] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," pp. 313–324, 2014.

[19] A. Yoshida, S. Yamamoto, and K. Agusa, "Semantic diff," *IPSJ Journal*, vol. 38, no. 6, pp. 1163–1171, 1997.

[20] T. Kehrer, U. Kelter, P. Pietsch, and M. Schmidt, "Adaptability of model comparison tools," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, 2012, pp. 306–309.

[21] K. M uller and B. Rumpe, "User-driven adaptation of model differencing results," in *Proceedings of the International Workshop on Comparison and Versioning of Software Models*, 2014.

[22] P. Bonizzoni, G. D. Vedova, R. Dondi, and Y. Pirola, "Variants of constrained longest common subsequence," *Information Processing Letters*, vol. 110, no. 20, pp. 877–881, 2010.

[23] Y.-T. Tsai, "The constrained longest common subsequence problem," *Information Processing Letters*, vol. 88, no. 4, pp. 173–176, 2003.

[24] Anonymous, "Dataset of Toward interactive optimization of source code differences: An empirical study of its performance," Temporary available at https://www.dropbox.com/scl/fi/5rw915g161f69dak5cmsg/scam2024-dataset.csv.bz2?rlkey=o53gd1f5ifwk1u7ts72pjwqhn&dl=0. will be published to Zenodo after acceptance, 2024.