# Towards Accurate File Tracking Based on AST Differences

Akira Fujimoto*, Yoshiki Higo* and Shinji Kusumoto*
*Graduate School of Information Science and Technology, Osaka University, Japan
{a-fujimt, higo, kusumoto}@ist.osaka-u.ac.jp

***Abstract*—In the field of software development, version control systems such as Git are imperative tools that help software teams manage source code. Git can detect a change history of each file individually. Even if a file was renamed in the past, Git can identify and track the before renamed file based on content similarities, which are calculated as the ratio of lines that match pre- and post-change files to the total number of lines. However, line-based comparison techniques do not consider source code structures and have coarse granularity, which can result in misidentifying pre-change files and tracking interruptions. To resolve these problems, this paper proposes a technique that calculates file content similarities using source code differences based on an abstract syntax tree. In experiments conducted on 197 open source Java-based projects, we found that the number of rename detections increased 3.3%, and that, on average, our technique tracked commits 1.37 times more frequently than previous technique. We also measured accuracy levels and found that the maximum F-measure was 0.943, which is higher than the 0.926 maximum value of the line-based technique.**

***Index Terms*—File tracking, Git, Abstract syntax tree**

## I. Introduction

In the field of software development, version control systems are imperative tools that help software teams manage source code over time [1]. Such systems enable developers to restore changed files to their previous versions, check differences between versions, and develop software in parallel with multiple developers. Additionally, accumulated change histories help developers to better understand software behaviors and the reasons for the changes [2] and change histories are also widely used in various research fields to predict buggy modules [3] and detect change patterns [4].

One such version control system is Git, which can identify a change history that includes not only the entire repository, but also individual files. In a case that a file has been changed in a past commit, Git does different things to track it depending on whether the filename has been changed or not. If the filename has not been changed, the file is marked as M (Modified) in the commit. Thus, the file can be tracked with the same filename. If the filename has been changed, the post-change file is marked with A (Added) and the pre-change file is marked with D (Deleted). The mapping of a post-change file to a pre-change one is performed by calculating the similarity of their contents, which is the ratio of lines that match both files. If the content similarity level exceeds a certain threshold, the file is regarded as renamed versions of the same file.

However, there is room for improvement in the way the similarity of files is calculated because line-based comparisons do not consider source code structures and have coarse granularity, which can result in misidentifying pre-change files and tracking interruptions. For example, even if only a class name, a method name, or a variable name is changed due to the file renaming process, the entire line is treated as changed, which may cause decreasing similarity and tracking losses.

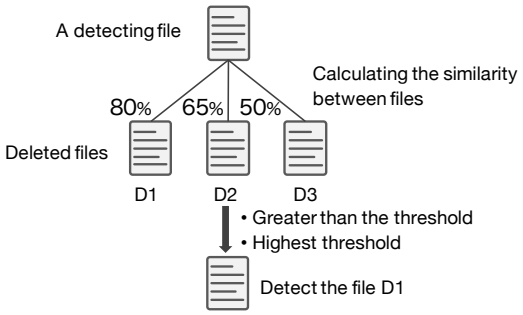


Fig. 1. File rename detection in Git.

To address these problems, this paper proposes a technique that uses source code difference detection based on an abstract syntax tree (AST) to improve file tracking on Git. Since an AST represents the source code structure itself, the authors believe its use could improve file tracking when calculating the similarity levels from their differences.

Although we are still in the early stages of this research, we have already conducted small-scale experiments to determine whether the central idea of our proposed technique has the potential to work well. More specifically, we carried out comparative experiments on 197 open source projects with a line-based technique in which we investigated four items: the number of rename detections, tracking accuracy levels, trackable history lengths, and execution times. These experimental results show that our proposed technique detected 3.3% more filename changes, and that the number of commits in the output increased by a factor of 1.37 Additionally, the tracking accuracy of our proposed technique was better than the line-based technique. However, the proposed technique increased the execution time by 1.71 times.

## II. Preliminaries

### A. File tracking on Git

In Git, developers can use the `--follow` option to track a specific file and view its change history, and they even can perform such tracking if the file has been renamed. If the filename being tracked is changed, Git identifies the file that existed before the change and uses that filename to track earlier commits. Figure 1 shows the Git file tracking mechanism for a commit where a file has been renamed. Git records information related to the deleted files, added files, and modified files for each commit, and then calculates

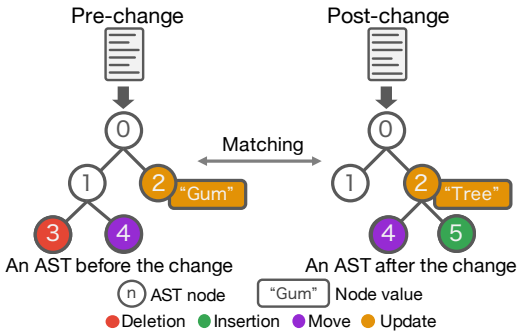


Fig. 2. Overview of differences detection in GumTree.

the file content similarity between the tracked and deleted files to detect the renamed files. File pairs that exceed a certain threshold are regarded as renamed file pairs. If multiple pairs of files exceed the threshold, the pair with the highest similarity level is designated as the renamed file pair.

The similarity $S$ between two files is calculated using the following formula. $length(f)$ is the number of lines of a file $f$ and $common(f_1, f_2)$ is the number of matched lines.

$$S(f_1, f_2) = \frac{common(f_1, f_2)}{max(length(f_1), length(f_2))} \times 100 \quad (1)$$

Algorithms such as Myers [5], Histogram[1], which is the enhanced version of Patience[2], and line-by-line hashing can be used to detect the line granularity differences.

### B. Abstract Syntax Tree

An AST is a tree-shaped data structure obtained by parsing source code that is constructed from a source file. The edge of an AST indicates a direct parent-child relationship between the nodes at its two ends, and each node of an AST is composed of at least four of the following five elements:

- **ID**: Each node includes a unique identifier in the AST.
- **Parent node**: Each node has a reference to its parent node. (Except for the root node, which does not have a parent node.)
- **Child nodes**: Each node has a reference to its child nodes. (Except for leaf nodes, which do not have child nodes.)
- **Label**: Each node has a label that represents its grammatical type, such as an if-statement or a variable declaration.
- **Value**: Some nodes have values that represent other information than the label. For example, the nodes of identifiers include their method names or variable names as values.

### C. GumTree [6]

GumTree is a tool that detects AST differences by generating an AST for each of the two source files of different versions given as input. It then compares those files and outputs the differences of the AST subtrees/nodes as an edit script. An edit script is a sequence of editing operations applied to the pre-change source code in order to obtain the post-change source code. Specifically, the edit scripts output by GumTree contain the delete/insert/move/update operations and the AST nodes where those operations were performed.

Figure 2 shows GumTree in operation. As can be seen in the figure, GumTree matches between the pre- and post-change versions of each AST node and treats such matched nodes as a same node. Next, GumTree refers to the matching results and ASTs and identify nodes where operations were performed using an algorithm created by Chawathe et al. [7]. In this example, Node 3 is regarded as deleted because it only exists before the change, while Node 5 is regarded as inserted because it only exists after the change, and Node 4 is regarded as moved because its parent is changed. Finally, Node 2 is regarded as updated because, while its parent node remained the same, its pre- and post-change values are different.

[1]https://javadoc.io/doc/org.eclipse.jgit/org.eclipse.jgit/latest/org/eclipse/jgit/diff/HistogramDiff.html

[2]https://alfedenzo.livejournal.com/170301.html

```
ebfb0d4753392ad00a5866e368cd0357aec12947 api tidy up
- .../src/main/java/com/beardedhen/androidbootstrap/support/DimenUtils.java
+ .../src/main/java/com/beardedhen/androidbootstrap/utils/DimenUtils.java
 1     - package com.beardedhen.androidbootstrap.support;
     1 + package com.beardedhen.androidbootstrap.utils;
 2   2   import android.content.Context;
 3   3   import android.content.res.Resources;
 4   4   import android.support.annotation.DimenRes;
     5 + /**
     6 +  * Utils class for resolving color resource values.
     7 +  */
 5   8   public class DimenUtils {
 6     -     public static float textSizeFromDimenResource(Context context,
                                                           @DimenRes int sizeRes) {
     9 +     /**
    10 +      * Resolves a dimension resource that uses scaled pixels
    11 +      *
    12 +      * @param context the current context
    13 +      * @param sizeRes the dimension resource holding an SP value
    14 +      * @return the text size in pixels
    15 +      */
    16 +     public static float pixelsFromSpResource(Context context,
                                                     @DimenRes int sizeRes) {
 7  17           final Resources res = context.getResources();
 8  18           return res.getDimension(sizeRes) / res.getDisplayMetrics().density;
 9  19       }
    20 +     /**
    21 +      * Resolves a dimension resource that uses density-independent pixels
    22 +      *
    23 +      * @param context the current context
    24 +      * @param res     the dimension resource holding a DP value
    25 +      * @return the size in pixels
    26 +      */
    27 +     public static float pixelsFromDpResource(Context context,
                                                     @DimenRes int res) {
    28 +         return context.getResources().getDimension(res);
    29 +     }
    30 +
    31 +
10  32   }
```

Fig. 3. An example of a change that interrupt tracking due to the low similarity.

## III. Motivating Example

Figure 3 shows a `DimenUtils.java` modification to a AndroidBootStrap project. In this change, the pathname was modified because of a restructuring project. At the same time, additions were made to the methods and Javadoc, but the pre- and post-change versions are still considered to be the same file. However, since the calculated similarity level of the two files is just 25% based on the formula (1) in section II-A, and since the default threshold value in Git is 50%, these files were regarded as non-identical. As a result, it would be impossible to track the pre-change file history.

The similarity level decreases even though the files are identical because the line granularity comparison does not consider the source code structure. For example, in the first line, only a part of the package name was changed, but the similarity level was calculated based on the assumption that the entire package statement had been deleted and reinserted. Similarly, only the method names were changed in the deletion on line 6 and the insertion on line 16, but those changes were

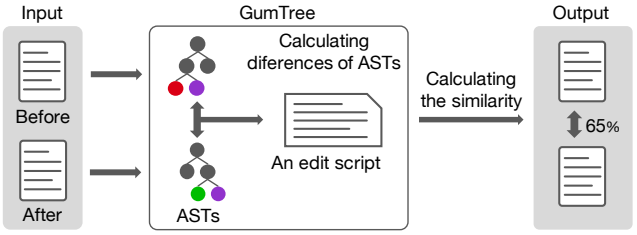


Fig. 4. Overview of the proposed technique.

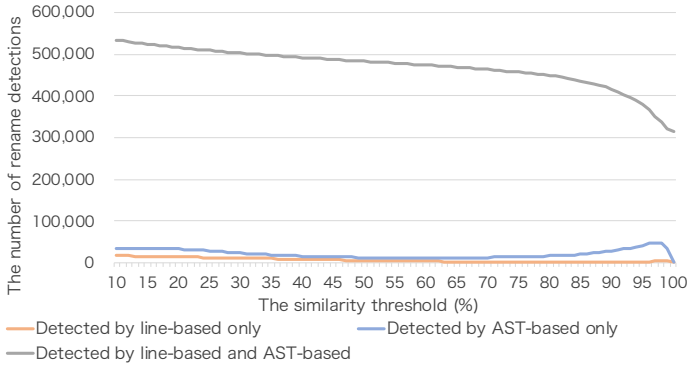


Fig. 5. The number of renames detected by each technique.

treated as entire line changes. As a result, 25% of the entire method was regarded as being changed. Furthermore, the fact that Javadoc occupied 17 of the 24 inserted lines is also a significant factor that reduced the similarity level, even though it did not change the program function.

To resolve this problem, our study proposes a similarity calculation technique that uses ASTs to consider source code structures. The use of ASTs makes it possible to compare fine levels of granularity and reduce the percentage of components, such as Javadoc and comments, which can introduce noise when calculating similarity levels.

## IV. Proposed Technique

An overview of our proposed technique, which improves the file tracking similarity calculation in Git, is shown in Figure 4. As can be seen in the figure, GumTree receives the pre- and post-change files, which are then used to calculate similarity levels. GumTree then generates ASTs from each file and calculates tree differences. Finally, based on the editing script output from GumTree, the similarity level is calculated. We define the similarity $S$ via the following formula. $t_1$ and $t_2$ denote the ASTs generated from the pre- and post-change files, while $treeSize(t)$ denotes the number of nodes contained in $t$. $editScript(t_1, t_2)$ is an edit script that GumTree outputs when it receives $t_1$ and $t_2$.

$$S(t_1, t_2) = \left(1 - \frac{length(editScript(t_1, t_2))}{treeSize(t_1) + treeSize(t_2)}\right) \times 100 \quad (2)$$

The more similar the contents of pre- and post-change files are, the shorter the length of the edit script is, and thus the higher the output similarity level. If the similarity level exceeds a certain threshold, the files are designated and tracked as renamed versions of the same file.

Here, it should be noted that our proposed technique does not consider the operation type in an edit script. In other words, the length value of an edit script increases by one no matter which operation is performed. This is because insertions and deletions operate on a single AST node, and the length of an edit script corresponds to the number of nodes. On the other hand, moves operate on an AST subtree even though the edit script length is one. Therefore, insertions and deletions have longer edit script lengths than moves. We consider the functional differences that result when the source code is moved rather than inserted or deleted to be minor. Therefore, to increase the similarity level when a move occurs, we use an edit script without weighing each operation.

Next, we use our proposed technique to calculate the similarity of the example shown in Figure 3. Here, the number of nodes in the AST generated from the pre- and post-change file was 53 and 103, respectively, and the length of the edit script output by GumTree was 52. As a result, the calculated similarity was 66.7%, and the proposed technique continues tracking because it regards the files as renamed versions of the same file.

If one or more of the files is unable to generate an AST, or if there is a file that contains syntax errors, the line-based technique is used to calculate the similarity level.

## V. Evaluation

In this section, we report on experiments conducted to evaluate our technique, in which we use Git that is implemented in Java, hereafter referred to as "JGit". This implementation was selected because GumTree is also implemented in Java, and thus can be easily integrated into JGit. More specifically, the experiment compared two versions of JGit that were applied to the proposed and line-based techniques in reference to the following four items:

- the number of rename detections,
- tracking accuracy levels,
- trackable change history lengths, and
- execution times.

### A. Target projects

The 197 Java projects used in this experiment were selected for use as targets from the Borges dataset [8], which consists of 2,279 projects that are popular on GitHub. Of these, 202 are Java projects. Two of those projects were excluded because they have not been already published, and three projects were excluded because they failed to run GumTree due to runtime errors.

### B. The number of rename detections

First, we varied the similarity threshold value by 1%, executed the rename detection of all the commits in the project, and then compared the number of rename detections. Figure 5 shows the total number of filename changes for all projects. In this figure, "AST-based" denotes the proposed technique, and "line-based" denotes the prior line-based technique.

Both techniques detected the majority of filename changes, and it seems that both techniques have the same level of

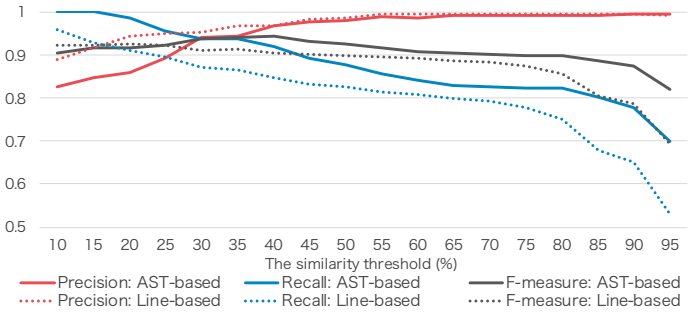


Fig. 6. Precision, recall, and F-measure levels.

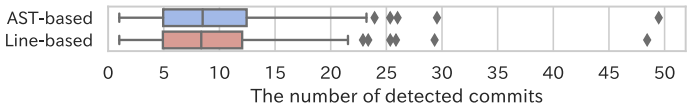


Fig. 7. The length of trackable change histories.

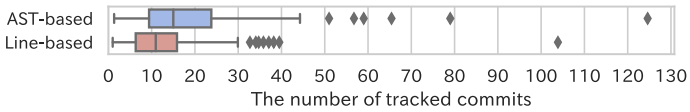


Fig. 8. History lengths when focusing solely on files with different outputs between the proposed and line-based techniques.

detection capability. However, the number of renames detected by the proposed technique exceeded the number of detection by the line-based technique by an average of 3.3% and a maximum of 13.1%. It can be seen that the number of rename detections by the proposed technique increased when the threshold was greater than 90%. Since the line-based technique identifies slight changes, such as variable and method name modifications, as an entire line changes, the number of matched lines decreased. In contrast, the similarity level calculated by the proposed technique tended to be higher because our technique can detect minor changes to only the target AST node. As a result, the number of rename detections identified by the proposed technique increased to a high threshold of more than 90%.

### C. Tracking accuracy

Next, we manually checked whether the change histories output by each technique are correct. Since it is evident that the tracking is correct when there are no filename changes, only filename changes were checked. We also varied the similarity threshold by 5% and then calculated the precision, recall, and F-measure at each threshold.

We randomly selected a single file from each project and created a dataset that consisted of the correct histories of the selected files. However, since it was not practical to check all deleted and added files combinations to create a dataset when files are renamed, our correct renames dataset was created based on the following steps by referring to literature [9]:

1) We execute the following command, manually check change histories, and count the number of correct rename detections.
   `jgit log -p -U 15 -M -Mscore 10 --follow` *path*[3]
2) Two of the authors carried out Step 1 independently.
3) The two authors then collated the results of Step 2 and discussed the change histories that conflict the results to reach a consensus on the number of renames.

The experimental targets were 191 projects that contain Java files in the latest commit. In Step 1, we used JGit, which we partially modified to apply to the proposed technique. In Step 3, the authors identified five projects that did not correspond to the number of filename changes and agreed on their number.

[3] `-Mscore` is an option we implemented to set the similarity threshold.

Next, we compared the number of renames detected by each technique and the dataset in order to calculate precision and recall levels. The following command measures the number of detected renames. When JGit detects renames, it outputs "`rename from XXX.java`" and "`rename to YYY.java`". Thus we can count the number of renames using `grep` and `wc` command.

```
jgit log -p -U 15 -M -Mscore threshold
--follow path
  | grep "^rename from|^copy from"
  | wc -l
```

Figure 6 shows how the precision, recall, and F-measure change levels according to given thresholds. The proposed and line-based techniques show almost the same level of precision when the threshold is greater than 40%. However, the precision level of the proposed technique falls below the level of the line-based technique when the threshold is lower than 40%. Thus, in the case of low threshold cases, the proposed technique may track more than necessary. On the other hand, the recall level of the proposed technique is higher than the line-based technique for all thresholds, thus indicating that the proposed technique can detect longer change histories than the line-based technique. Furthermore, there is only a slight gap in the tracking even if the similarity threshold is set at a high level. The maximum F-measure of the proposed technique is 0.943 at the threshold of 40%, which is higher than the F-measure of the prior technique at all set thresholds. The maximum value of the line-based technique is 0.926 at 40%. From these results, we can conclude that our proposed technique outperforms the line-based technique in terms of tracking accuracy, and the best threshold of the proposed technique is 40%.

### D. The length of trackable change history

Next, we compared the number of commits tracked by each technique using the `--follow` option. The average number of commits in a change history for each project, per file, is shown as a box plot in Figure 7. The target is all the files contained in each project. This includes not only files that exist in the latest commit but also all the previously existing files. The similarity threshold is set at 40%, which is the highest F-measure in the proposed technique based on the tracking accuracy experiment results.

From the results obtained by comparing both techniques, we can see that the proposed technique tracks the history slightly longer than the line-based technique. The median of

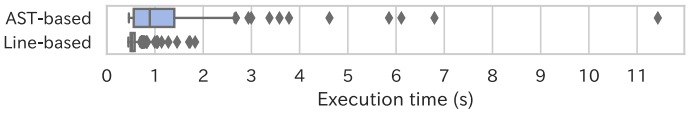


Fig. 9. Execution times.

the proposed technique is 8.49, and that of the line-based technique is 8.37. Thus, we can see that the overall number of trackable commits increased by 1.5%.

Next, we focused solely on files with different output histories between the proposed and line-based techniques. Figure 8 shows the average number of commits in the change history of each file for each project. A total of 150 projects contained files with different outputs when the proposed and line-based techniques were used. The result indicates that the proposed technique tracks the change history much longer than the line-based technique. When comparing the median, we see that the number of trackable commits increased 1.37 times. From these results, we can conclude that the proposed technique makes it possible to track change histories past the point where the line-based technique breaks off.

There were 12 projects that the line-based technique that could track histories longer than the proposed technique. The average difference in the number of commits was 1.27. On the other hand, the proposed technique could track 138 projects longer than the line-based technique and the mean difference in their tracking lengths was 6.56. These results show that, in most cases, our proposed technique can track change histories longer than the line-based technique, and that in projects where the line-based technique can track histories longer than the proposed technique, the difference in the number of commits is insignificant.

### E. Execution times

In this experiment, we select a file from each project to track and measure the execution times. The following commands apply to each technique:

```
jgit log -M -Mscore 40 --follow path
```

The similarity threshold is set to 40%, and the targets are 191 projects that contain Java files in the latest commit.

We used an Ubuntu machine with a Ryzen Threadripper 3960X (24C/48T) CPU and 128 GB RAM. The file tracking and supporting multi-threading were optimized for JGit.

Figure 9 shows the execution times of each technique as a box plot. As can be seen in the figure, the proposed technique takes longer because the computational complexity increases as it constructs the necessary ASTs and detects the differences between them. When the median values were compared, we found the execution time of the proposed technique was 1.71 times longer than the line-based technique.

## VI. Threats to Validity

Although all the experiments were conducted on Java projects, GumTree also supports other programming languages such as Python and JavaScript, so the proposed technique could be applied to those languages as well. This means that future studies involving programming languages other than Java could provide different results.

We created a dataset of correct rename detections using the tracking results of Git for the accuracy portion of this study. Since it is normally necessary to check all the file combinations in a commit where the filename was changed and then decide which combination is most suitable (correct), the dataset may have affected the accuracy evaluation. However, since it would be unrealistic to check all of the combinations, even for a small number of targets, we limited this study to checking as many results as practical with a sufficiently low threshold. Nevertheless, even though this construction process does not ensure a 100% correct dataset, it was sufficiently accurate for use when comparing different techniques.

## VII. Related Works

FinerGit [9] and Historage [10] track change histories in Git at method-level granularity. These techniques split a file for each method and used the Git file tracking system to track those methods. In particular, since Historage extracts methods and saves them in a file, it is possible to apply our technique via an AST method parser. CodeShovel [11], which is also proposed as a method-level tracking technique, matches methods by filename, method signature, and method body to detect operations performed on methods. Code clone detection and text-based similarity levels are used to match methods.

Our proposed technique differs from the abovementioned techniques in that it aims to improve tracking at file-level granularity. Furthermore, while Java was the target of above-mentiond techniques, the proposed technique can be applied to all programming language to which GumTree is applicable.

## VIII. Conclusion

This study proposed a technique for improving the file tracking system in Git. Our approach calculates similarity levels from the output edit scripts by GumTree, which detects AST differences. To confirm the validity of our proposed technique, we surveyed 197 Java projects for data related to the number of rename detections, tracking accuracy levels, trackable change history lengths, and execution times. The obtained results showed that our proposed technique could detect more filename changes as well as longer and more accurate track history changes than the line-based technique. The best threshold of the proposed technique is 40%. However, execution times increased for the proposed technique.

In the future, we will focus on improving our similarity calculations in order to detect more filename changes, increase tracking accuracy, and speed up. We also plan to apply the proposed technique to other works because it is our belief that the proposed technique may have applications in other research fields and thus has the potential to help other tools that use Git change histories to achieve better results. When a bug prediction tool based on development histories is applied to the proposed technique, it is expected to predict higher accuracy.

## Acknowledgments

This work was supported by JSPS Grant Numbers JP20H04166 and JP21K18302.

## REFERENCES


[1] B. De Alwis and J. Sillito, "Why are software projects moving from centralized to decentralized version control systems?" in *ICSE Workshop on Cooperative and Human Aspects on Software Engineering*, 2009, pp. 36–39.

[2] L. Hattori, M. D'Ambros, M. Lanza, and M. Lungu, "Software evolution comprehension: Replay to the rescue," in *IEEE International Conference on Program Comprehension*, 2011, pp. 161–170.

[3] H. Hata, O. Mizuno, and T. Kikuno, "Bug prediction based on fine-grained module histories," in *Proceedings of the international conference on software engineering*, 2012, pp. 200–210.

[4] S. Negara, M. Codoban, D. Dig, and R. E. Johnson, "Mining fine-grained code changes to detect unknown change patterns," in *Proceedings of the International Conference on Software Engineering*, 2014, pp. 803–813.

[5] E. W. Myers, "An O(ND) difference algorithm and its variations," *Algorithmica*, vol. 1, no. 1-4, pp. 251–266, 1986.

[6] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the ACM/IEEE international conference on Automated software engineering*, 2014, pp. 313–324.

[7] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, "Change detection in hierarchically structured information," *Acm Sigmod Record*, vol. 25, no. 2, pp. 493–504, 1996.

[8] H. Borges, A. Hora, and M. T. Valente, "Understanding the factors that impact the popularity of github repositories," in *IEEE International Conference on Software Maintenance and Evolution*, 2016, pp. 334–344.

[9] Y. Higo, S. Hayashi, and S. Kusumoto, "On tracking java methods with git mechanisms," *Journal of Systems and Software*, vol. 165, p. 110571, 2020.

[10] H. Hata, O. Mizuno, and T. Kikuno, "Historage: fine-grained version control system for java," in *Proceedings of the International Workshop on Principles of Software Evolution and the annual ERCIM Workshop on Software Evolution*, 2011, pp. 96–100.

[11] F. Grund, S. Chowdhury, N. C. Bradley, B. Hall, and R. Holmes, "Codeshovel: Constructing method-level source code histories," in *Proceedings of the IEEE/ACM International Conference on Software Engineering*, 2021, pp. 1510–1522.