Staged Tree Matching for Detecting Code Move across Files

Akira Fujimoto, Yoshiki Higo, Junnosuke Matsumoto, and Shinji Kusumoto
Osaka University
Suita, Osaka, Japan
{a-fujimt,higo,j-matumt,kusumoto}@ist.osaka-u.ac.jp

ABSTRACT

In software development, developers often need to understand source code differences in their activities. GumTree is a tool that detects tree-based source code differences. GumTree constructs abstract syntax trees from the source code before and after a given change, and then, it identifies inserted/deleted/moved subtrees and updated nodes. Source code differences are detected based on the four kinds of information in GumTree. However, GumTree calculates the difference for each file individually, so that it cannot detect moves of code fragments across files. In this research, we propose (1) to construct a single abstract syntax tree from all source files included in a project and (2) to perform a staged tree matching to detect across-file code moves efficiently and accurately. We have already conducted a pilot experiment on open source projects with our technique. As a result, we were able to detect code moves across files in all the projects, and the number of such code moves was 76,600 in total.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**; **Maintaining software**; **Software version control**; *Software creation and management*.

KEYWORDS

Code differences, Abstract Syntax Tree, GumTree

ACM Reference Format:
Akira Fujimoto, Yoshiki Higo, Junnosuke Matsumoto, and Shinji Kusumoto. 2020. Staged Tree Matching for Detecting Code Move across Files. In *28th International Conference on Program Comprehension (ICPC '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3387904.3389289

1 INTRODUCTION

GumTree is one of the state-of-the-art techniques that detect source code differences in the level of abstract syntax tree (in short, AST) [2]. GumTree generates an AST for each of the given two source files of different versions as input, and it outputs the differences of AST subtrees/nodes by comparing them. AST-based differencing techniques have two advantages against text-based differencing ones such as Unix diff command.

ICPC '20, October 5–6, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7958-8/20/05…$15.00
https://doi.org/10.1145/3387904.3389289

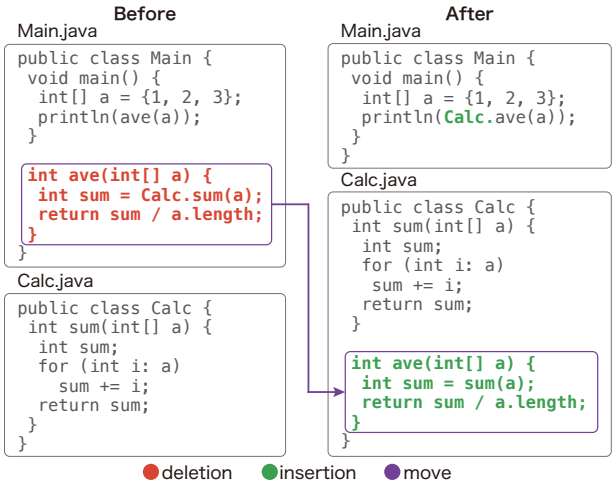

Figure 1: An example of code move across files and detecting code differences of GumTree

- In text-based techniques such as diff and cregit [4], differences are represented by two kinds of information, *deletion*, and *insertion*. On the other hand, in AST-based techniques, differences can be represented by four kinds of information, *deletion*, *insertion*, *move*, and *update*.
- AST-based techniques such as GumTree and ChangeDistiller [3] identify code differences in a more appropriate range than text-based ones. For example, if a token in a line has been deleted, diff regards the change as a deletion of the whole line and an insertion of a new line while GumTree identifies that the subtree for the token has been deleted.

GumTree has been used in many studies. For example, it has been used to analyze Maven build files [6], detect bugfix patterns [7], generate commit messages [1], and suggest API code [9].

However, GumTree (and other AST-based differencing techniques) still has a problem. GumTree performs detecting code differences for each file individually, which means that code moves across files are not captured as they are, but they are captured as code deletion in a file and code insertion in another file. The authors consider that code moves across files happen frequently, and detecting them will be helpful for developers to understand code changes. In this research, the authors are trying to detect across-file code moves. At this moment, our technique includes two new technical contributions.

- Our technique constructs a single huge AST from all the (compilable) source files in a project while GumTree constructs an AST for each file.

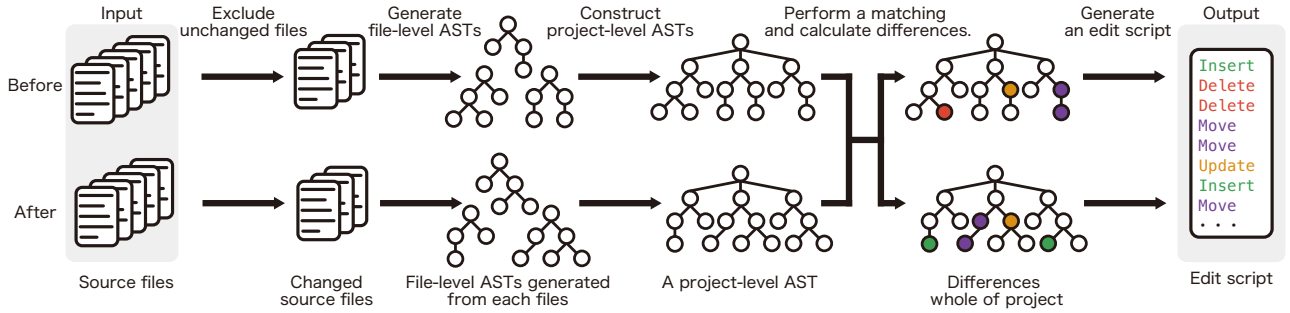

Figure 2: Overview of proposed technique

- Our technique performs a staged tree matching. In the first stage, our technique tries to find matching subtrees within each file. In the second stage, it tries to find matching subtrees that have not been matched in the first stage.

The combination of the above two contributions can realize to detect across-file code moves efficiently and accurately.

We have applied our technique to eight open source projects so far. As a result, we detected 76,600 across-file code moves in total. We also found that there are some common features in across-file code moves.

2 GUMTREE

GumTree receives a pair of source files (before and after a change) as inputs, and it outputs an edit script, which is a sequence of editing operations applied to the source code before the change to obtain the source code after the change. GumTree outputs, as an edit script, the operations of *deletion*/*insertion*/*move*/*update* and the information of the AST nodes where the operations were performed.

GumTree constructs two ASTs for a pair of source files and performs a matching to calculate differences between the tree structures. Matching is the process of associating AST nodes before and after the change. The associated nodes are treated as the same nodes before and after the change. GumTree identifies editing operations by referring to the matching results.

- Nodes existing only in the AST before the change are treated as *deletion*.
- Nodes existing only in the AST after the change are treated as *insertion*.
- Nodes having different parent nodes before and after the change are treated as *move*.
- Nodes including different values before and after the change are treated as *update*.

3 RESEARCH MOTIVATION

GumTree can detect what operations have been performed on changes in a single file. However, changes can be made to multiple files at once. In such cases, GumTree has no capability of detecting code moves across files.

Figure 1 shows a change where files `Main.java` and `Calc.java` have been changed at the same time. In the change, method `ave()` that existed in `Main.java` has been moved to `Calc.java`. If we apply GumTree to this change, we obtain an edit script that `ave()` was deleted from `Main.java` and another edit script that `ave()` was inserted to `Calc.java`. However, in this case, an edit script that `ave()` was moved to `Calc.java` from `Main.java` is more appropriate for expressing the operation actually performed. Consequently, if a developer applies GumTree to this change, the developer may miss the fact that `ave()` was moved across files. Due to this difference, developers may misunderstand the code change.

In this study, we propose a new technique of generating edit scripts that has the capability of detecting code moves across files. We consider that edit scripts generated by our technique are more helpful in understanding changes in source code.

4 PROPOSED TECHNIQUE

If we simply construct a huge AST for all the source files in a project and simply perform a matching on it, the following two problems will happen.

- It will take a very long time to calculate differences because a project-level AST is so huge.
- The accuracy of node matching in a project-level AST will be decreased compared to a file-level AST because there are so many matching candidates in a project-level AST.

To avoid the above problems, we propose to (1) use only changed files to construct a project-level AST and (2) perform a staged tree matching, each of which is described in detail in Subsections 4.1 and 4.2, respectively.

An outline of our technique is shown in Figure 2. The input of our technique is all source files included in a project before and after a change, and it outputs an edit script. Our technique firstly constructs an AST for each of the changed source files, and then, those ASTs are connected with the common root node, which is newly introduced in our technique. That is, a single AST is constructed for a set of changed source files. Then, our technique performs a staged tree matching to calculate the differences.

4.1 Constructing a Project-level AST without Unchanged Files

Figure 3 shows a simple example of constructing a project-level AST from four source files. First, a file-level AST is constructed for each of the files included in the project. Then, a common root node is introduced, and it is connected with each of the root nodes of the file-level ASTs. The order in which the generated ASTs are added as child nodes is the alphabetical order of the source file names.

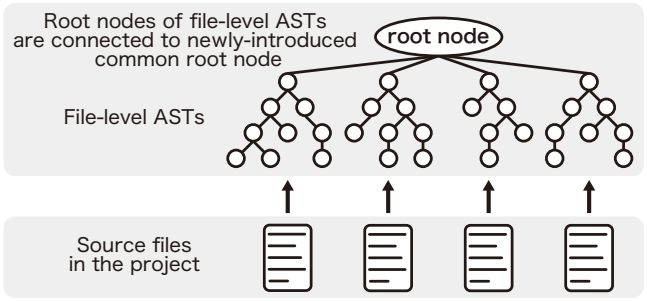

Figure 3: Construction of project-level AST

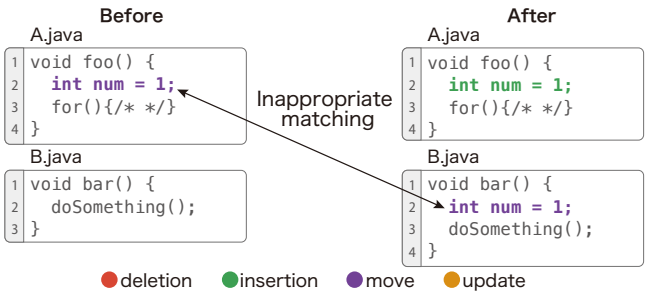

Figure 4: An example of false move detection

If a project-level AST is constructed from all the files in the project, the number of nodes in the project-level AST becomes huge, and the matching process on the AST will take so long time. To avoid this problem, we proposed not to add the ASTs of unchanged files to a project-level AST. In most of the commits, the number of files in them is very small, which means that project-level ASTs for such commits do not become so huge.

4.2 Staged Tree Matching

When the difference detection target is expanded from a single file to the entire project, false positive of code moves across files will be detected. This is due to GumTree not being able to match nodes properly as the AST of the entire project. An example of false move detection is shown in Figure 4. In this example, '`int num = 1;`' is included only in method `A.java` before the change. In the change, '`int num = 1;`' is inserted to method `B.java`. If we apply GumTree to a project-level AST on the change, GumTree regards

- '`int num = 1;`' in `A.java` has been moved to `B.java`, and
- another '`int num = 1`' has been inserted to `A.java`.

The reason why the matching is not performed properly is that the matching target is expanded to the entire project and the number of candidates gets increased, so that an appropriate node does not get matched. More concretely, in this case, '`int num = 1;`' in `A.java` of the before-change AST is coincidentally matched with '`int num = 1;`' in '`B.java`' of the after-change AST.

To avoid this problem, we propose a staged tree matching. An overview of the staged tree matching is shown in Figure 5. In the first stage, matching is performed within the subtrees of each file. After that, as the second stage, another matching is performed for

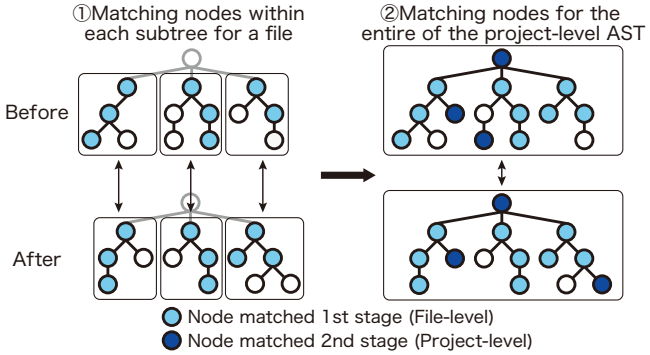

Figure 5: Staged tree matching

the entire of the project-level AST. In the second stage, the matching targets are only the nodes that failed to be matched in the first stage. An AST matching within file-level subtrees is performed first in our technique. This aims to reduce the number of matching candidates in the second stage to realize an appropriate matching across files.

If the staged tree matching is applied to the change of Figure 4, the '`int num = 1;`' in `A.java` before the change is matched with '`int num = 1;`' in `A.java` after the change at the time of the matching within file-level subtrees. On the other hand, '`int num = 1;`' in `B.java` after the change is not matched with any other node at the first-stage and second-stage matchings. As a result, '`int num = 1;`' in `B.java` is regarded as inserted by the change, which can avoid detecting false positives of across-file code move.

5 EXPERIMENT

We conducted an experiment on open source projects with Graftast[1], which is our implementation based on the proposed technique. The purposes of this experiment are answering the following questions.

(1) Can our technique detect code moves across files?
(2) What are the features of code moves across files?

Our experimental targets are the projects included in CVS-Vintage [8] that were used in the experiment of GumTree [2]. Among the projects included in CVS-Vintage, we used the projects that have been migrated to Git as our experimental targets. The

Table 1: Target projects

Project Name	# of commits	Latest Commit Date
ArgoUML	16,144	Jan. 11, 2015
dnsjava	1,771	Oct. 27, 2019
Eclipse[1]	29,811	Nov. 11, 2019
JHotDraw	763	Aug. 27, 2018
JUnit 4	2,418	Nov. 2, 2019
Apache Log4j 2	10,752	Nov. 1, 2019
Apache Struts	5,697	Nov. 4, 2019
Apache Tomcat	21,492	Nov. 6, 2019

[1]Only `eclipse.ui.workbench` was investigated.
[1]https://github.com/kusumotolab/Graftast

target projects are shown in Table 1. All the target projects are written in Java.

In the experiment, source files before and after each commit in the target projects were given as input to our technique, and we counted the number of across-file code moves that were detected by our technique. We also sampled and visually checked the detected code moves across files in order to grasp the common features in across-file code moves.

5.1 Results

Table 2 shows the number of the detected across-file code moves and all the detected code moves. The parentheses next to the number of across-file code moves indicate the percentage of across-file code moves to all detected code moves. The results show that our technique detected across-file code moves from all the target projects, but the ratio of them against all the detected code moves are different (0.5∼10.9%).

We classified the detected code moves based on their node types. Table 3 shows the number of across-file code moves of *method declarations*, *while-statements*, *for-statements*, and *if-statements*. We can see that the number of *method* moves is the largest number, but the developers also moved smaller code fragments such as conditional blocks across files.

We also compared the detection results of our technique with the detection results of ReffactoringMiner [10] on all the target projects except Eclipse because ReffactoringMiner cannot specify subdirectories in a given target program. ReffactoringMiner is a tool that detects refactorings from a given commit history, and it has a capability of detecting 40 refactoring patterns such as Pull Up Method or Push Down Method. The comparison results showed that only our technique was able to detect code moves of methods that were declared in anonymous classes.

We manually investigated across-file code moves for nodes of method declarations and conditional blocks. The remainder of this section describes some features of code moves that we detected.

- When a large class was split into multiple classes, methods in the class were moved to new classes.
- There are many cases that methods were moved to utility classes whose names include `Util` or `Helper`.
- We also found that several methods were moved to non-utility classes. In such cases, the names of classes where the methods existed before and after the moves are similar to each other.
- Nodes of *while-statement*, *for-statement*, *if-statement* are moved to other classes and they are encapsulated in new method declarations. Such code moves were conducted where the original methods were long, and refactorings were performed to cut out the processing as new methods.
- When the inheritance relationship between classes changed, across-file code moves occurred. Most of such cases are code moves from concrete classes to abstract classes. Code moves from abstract classes to concrete classes are minority cases.

Table 2: Number of detected code moves

Project Name	Across-file Moves		All Moves
ArgoUML	9,514	(10.5%)	90,811
dnsjava	1,589	(0.5%)	325,362
Eclipse	25,177	(10.9%)	231,814
JHotDraw	5,711	(0.7%)	847,946
JUnit 4	2,830	(9.7%)	29,262
Apache Log4j 2	6,852	(5.8%)	117,102
Apache Struts	3,708	(7.2%)	51,768
Apache Tomcat	21,258	(8.9%)	259,447

6 CONCLUSION

In this research, we are trying to detect code moves across files with AST-based differencing techniques. We proposed a staged tree matching in this paper to realize high-performance and high-accuracy detection of across-file code moves. We also conducted a pilot experiment and found that the proposed technique had the capability of detecting across-file code moves from large-scale projects.

This research is still in the early stage, and we have many future works. The followings are some of them.

- We found that the detection results included false positives when we checked the detected across-file code moves shown in Table 3. Our first future work is to measure the detection accuracy of the proposed technique.
- We need to conduct a detailed comparison with ReffactoringMiner because at this moment we only have compared the number of detected code moves.
- We plan to evaluate the performance of the staged tree matching.
- Providing raw text of edit script that includes across-file code moves is still not enough to understand what changes happened. We are going to develop a GUI-frontend by extending the visualization tool included in the GumTree package.
- We are going to integrate the functionality of detecting copy-and-paste operations [5] to our proposed technique.

ACKNOWLEDGMENT

This work was supported by MEXT/JSPS KAKENHI 17H01725.

Table 3: Number of across-file code moves on some specific node types

Project Name	method	while	for	if
ArgoUML	186	13	8	138
dnsjava	88	0	1	25
Eclipse	978	18	23	368
JHotDraw	204	1	7	48
JUnit 4	166	1	2	11
Apache Log4j 2	365	3	4	44
Apache Struts	211	1	2	43
Apache Tomcat	974	15	16	364

REFERENCES

[1] Md Salman Ahmed and Anika Tabassum. 2018. Automatic Contextual Commit Message Generation : A Two-phase Conversion Approach.

[2] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *International Conference on Automated Software Engineering, Vasteras, Sweden - 19, 2014*. 313–324.

[3] Beat Fluri, Michael Wuersch, Martin PInzger, and Harald Gall. 2007. Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Transactions on Software Engineering* 33, 11 (2007), 725–743.

[4] Daniel M. German, Bram Adams, and Kate Stewart. 2019. Cregit: Token-Level Blame Information in Git Version Control Repositories. *Empirical Software Engineering* 24, 4 (2019), 2725–2763.

[5] Yoshiki Higo, Akio Ohtani, and Shinji Kusumoto. 2017. Generating Simpler AST Edit Scripts by Considering Copy-and-Paste. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. 532–542.

[6] Christian Macho, Shane McIntosh, and Martin Pinzger. 2017. Extracting build changes with builddiff. In *The 14th International Conference on Mining Software Repositories*. 368–378.

[7] Fernanda Madeiral, Thomas Durieux, Victor Sobreira, and Marcelo Maia. 2018. Towards an automated approach for bug fix pattern detection.

[8] Martin Monperrus and Matias Martinez. [n.d.]. CVS-Vintage: A Dataset of 14 CVS Repositories of Java Software.

[9] Anh Tuan Nguyen, Michael Hilton, Mihai Codoban, Hoan Anh Nguyen, Lily Mast, Eli Rademacher, Tien N Nguyen, and Danny Dig. 2016. API code recommendation using statistical learning from fine-grained changes. In *International Symposium on Foundations of Software Engineering*. ACM, 511–522.

[10] Nikolaos Tsantalis, Matin Mansouri, Laleh M. Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and Efficient Refactoring Detection in Commit History. In *Proceedings of the 40th International Conference on Software Engineering*. 483–494.