

BDiff: Block-aware and Accurate Text-based Code Differencing

YAO LU, WANWEI LIU, TANGHAORAN ZHANG, KANG YANG, YANG ZHANG, WENYU XU, LONGFEI SUN, and XINJUN MAO, National University of Defense Technology, China
SHUZHENG GAO and MICHAEL R. LYU, The Chinese University of Hong Kong, China

Code differencing is a fundamental technique in software engineering practice and research. The most widely used differencing technology today remains text-based differencing methods, which operate at the textual line level. While researchers have proposed text-based differencing techniques capable of identifying line changes over the past decade, existing methods exhibit a notable limitation in identifying edit actions (*EAs*) that operate on text blocks spanning multiple lines. Such *EAs* are common in developers' practice, such as moving a code block for conditional branching or duplicating a method definition block for overloading. Existing tools represent such block-level operations as discrete sequences of line-level *EAs*, compelling developers to manually correlate them and thereby substantially impeding the efficiency of change comprehension. To address this issue, we propose BDiff, a text-based differencing algorithm capable of identifying two types of block-level *EAs* and five types of line-level *EAs*. Building on traditional differencing algorithms, we first construct a candidate set containing all possible line mappings and block mappings. Leveraging the Kuhn-Munkres algorithm, we then compute the optimal mapping set that can minimize the size of the edit script (*ES*) while closely aligning with the original developer's intent. To validate the effectiveness of BDiff, we selected five state-of-the-art tools, including large language models (LLMs), as baselines and adopted a combined qualitative and quantitative approach to evaluate their performance in terms of *ES* size, result quality, and running time. Experimental results show that BDiff produces higher-quality differencing results than baseline tools while maintaining competitive runtime performance. Our experiments also show the unreliability of LLMs in code differencing tasks regarding result quality and their infeasibility in terms of runtime efficiency. Based on the proposed algorithm, we have implemented a web-based visual differencing tool, which can be integrated into Git. We have open-sourced our tool at <https://github.com/bdiff/bdiff>.

CCS Concepts: • **Software and its engineering** → **Software configuration management and version control systems**; **Software maintenance tools**; **Maintaining software**.

Additional Key Words and Phrases: Code differencing, Code review, Large language model, Code block

1 Introduction

Code differencing, a technique that computes the differences between two program versions, is fundamental and essential for both software engineering (SE) practice and research. Developers use code differencing tools to review code changes, debug defects, and track code evolution. Computer-aided software engineering tools rely on code differencing techniques as foundational support for tasks such as code merging. Furthermore, researchers employ these techniques in studies aimed at identifying bug introductions [6, 23, 40], mining change patterns [33, 48], measuring developer contributions [19, 31], etc.

Existing code-differencing algorithms can be classified into two categories based on the representation of code [25]: (1) text-based algorithms working on textual lines and (2) tree-based algorithms working on an abstract syntax tree (AST). Text-based methods, such as the traditional Myers algorithm [36], compare raw source code at the character or line level. They typically use techniques like the longest common subsequence (LCS) to detect basic changes of lines and can be applied to

Authors' Contact Information: Yao Lu, luyao08@nudt.edu.cn; Wanwei Liu, wwliu@nudt.edu.cn; Tanghaoran Zhang, zhangthr@nudt.edu.cn; Kang Yang, yangkang@nudt.edu.cn; Yang Zhang, yangzhang15@nudt.edu.cn; Wenyu Xu, xuwenyu@nudt.edu.cn; Longfei Sun, lsun@nudt.edu.cn; Xinjun Mao, xjmao@nudt.edu.cn, National University of Defense Technology, Chang Sha, Hunan, China; Shuzheng Gao, szgao23@cse.cuhk.edu.hk; Michael R. Lyu, lyu@cse.cuhk.edu.hk, The Chinese University of Hong Kong, Hong Kong, China.

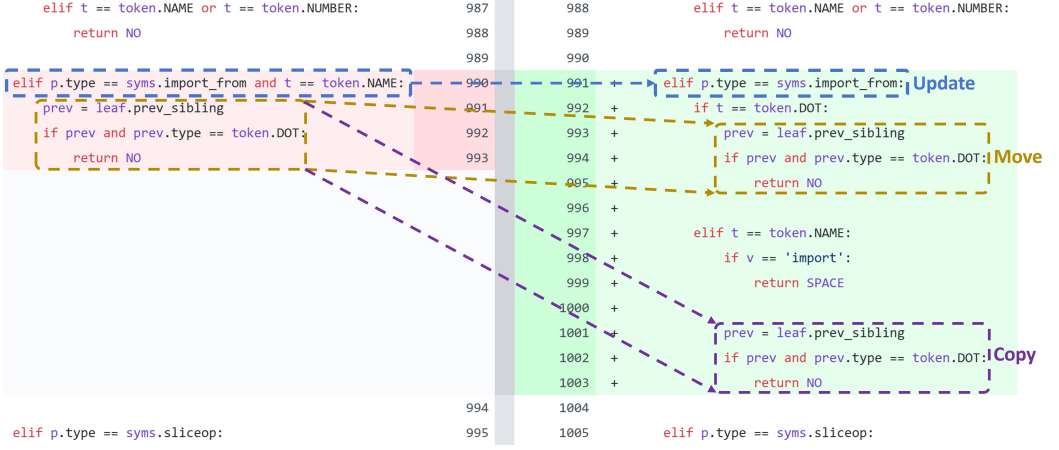


Fig. 1. A real case of line updating, block moving, and block copying (psf/black, e1e8909, black.py)

any text file. In contrast, tree-based methods, such as ChangeDistiller [13] and GumTree [11], parse code into ASTs and compare these ASTs to detect differences. These approaches capture structural and semantic changes more accurately, offering more fine-grained changes. A major limitation of these methods is that they are language-dependent and require adapting different parsers for various programming languages and language versions. Currently, owing to their universality, efficiency, and robustness, developers generally use text-based differencing tools in practice [14, 37], and we focus on this technical approach.

Text-based differencing has been a long-standing research topic, and a suite of notable algorithms has been proposed and refined over time. The Hunt-McIlroy algorithm [24], proposed in 1976, serves as a foundational method for computing the LCS between two sequences. The Myers algorithm [36], adopting a greedy strategy, is a highly efficient and influential method for computing the minimum edit script (ES) between two sequences, *i.e.*, the minimum number of edit actions (EAs) required to transform one sequence into the other. A limitation of these traditional algorithms is that the computed EAs are only represented as deletions and insertions. These primitive EAs often fail to capture developers' actual changes such as line updates. To address this limitation, Canfora et al. [3, 4] proposed ldiff, a method that takes the output of the Myers algorithm as input and can identify line updates and line moves. Furthermore, other related works [2, 9, 18, 42] have developed algorithms capable of tracking the evolution of source code lines, *e.g.*, line splitting and line merging in LHDiff [2].

While existing works have made progress in identifying line-level changes, they exhibit a key limitation in detecting block-level changes, which poses substantial challenges for developers in effectively understanding actual change intentions. In practice, developers frequently perform edit operations on multi-line code blocks, *e.g.*, moving a code block into an *if*-condition structure for defensive programming, or duplicating a function block with subsequent modifications to implement function overloading [1]. Figure 1 presents the Git diff view of a historical commit from the GitHub project “psf/black”. The differencing results consist of 4 consecutive deleted lines and 13 consecutive added lines. Through a detailed analysis, however, we can infer that the developer actually moved a code block (spanning lines 991–993) into an *if*-condition and copied the same block into an *elif*-condition. This example illustrates that a single block-level EA corresponds to multiple line-level EAs. Identifying such block-level EAs can therefore substantially reduce

the *ES* size and help developers more efficiently comprehend edit intent. Moreover, in real-world development scenarios, the two blocks involved in block-level *EAs* may be far apart, encompassing both changed code (for block movement) and unchanged code (for block duplication) from the original version, and the identified block-level *EAs* can therefore assist developers in evaluating the impact of code changes. However, identifying both line-level and block-level *EAs* that can minimize the size of the *ES* and are close to the original developer's intent is challenging. Code files often contain multiple instances of identical or highly similar lines and blocks-this ambiguity can lead to scenarios where a single line or block maps to multiple counterparts in the other version. Computing an optimal mapping set that establishes mappings between lines and blocks across the original and modified code versions remains a non-trivial challenge.

To address the challenges, in this paper, we present BDiff, a novel text-based differencing algorithm which can effectively identify two types of block-level *EAs* and five types of line-level *EAs*. BDiff is composed of three successive phases. First, building on the results of traditional differencing algorithms, we construct the candidate set containing all possible line mappings and block mappings. Then, we model the mappings as a weighted bipartite graph and iteratively use the Kuhn-Munkres algorithm [35] to compute the optimal mapping set, which can minimize the edit-script size and are close to the original developer intent. Finally, based on the optimal mapping set, we deduce the *ES*. To validate BDiff's effectiveness, we constructed an evaluation dataset consisting of 2,997 real-world code-change cases, covering three popular languages: Python, Java, and XML. We selected five baseline tools representing three distinct technical paradigms (text-based, AST-based, and large language model (LLM)-based approaches) and quantitatively compared BDiff's performance against these baselines in *ES* size and running time. To investigate the quality of BDiff's results, we further conducted two experiments: a qualitative manual evaluation experiment involving 10 raters and a mutation-based evaluation experiment. The experimental results show that: (1) BDiff outperforms all baseline tools in differencing result quality; (2) BDiff outperforms text-based baseline tools in *ES* size, reducing the average *ES* size by at least 28%; (3) BDiff outperforms all baseline differencing tools in running time (0.085s on average), with the only exception being Git diff (Myers); and (4) the accuracy of BDiff in identifying ground-truth *EAs* and *ES* are 95.3% and 82.4%, respectively. Based on our algorithm, we have implemented an open-source and web-based diff visualization tool. To summarize, our work makes the following contributions:

- To the best of our knowledge, we are the first to explore the identification of block-level *EAs* which are common in developers' edit operations.
- We propose BDiff, a systematic approach to identify both line-level *EAs* and block-level *EAs*, which consists of three phases: establishing cross-version mappings, computing an optimal mapping set, and deducing the *ES*.
- We conduct an extensive experiment to evaluate the performance of BDiff and baseline tools. Experimental results show that BDiff produces higher-quality differencing results than baseline tools while maintaining competitive runtime performance.
- We have implemented an open-source visual differencing tool, which can be integrated into Git.

Our dataset, scripts, and experimental results are available in the GitHub repository¹.

2 Related Work

2.1 Text-based Code Differencing

Traditional text-based differencing techniques typically compare two files at the line granularity, with outputs consisting of line insertions and deletions. Hunt and McIlroy introduced a foundational

¹<https://github.com/BDiff/BDiff-Evaluation-Experiment>

method for computing text file differences—the seminal *Hunt-McIlroy* algorithm [24]. This algorithm employs dynamic programming to efficiently identify the LCS between files. It was employed as the early differencing algorithm in Unix *diff*. Another prominent line-differencing algorithm is Myers [36], which uses dynamic programming to find a LCS with a time complexity of $O(N+D^2)$. It superseded the *Hunt-McIlroy* algorithm as the differencing algorithm for Unix *diff* and has been widely adopted in version control systems such as Git. Besides the default Myers, Git offers three alternative *diff* algorithms: *Minimal*, *Patience*, and *Histogram*. Notably, *Minimal* and *Histogram* are improved variants of Myers and *Patience*, respectively. Through a systematic mapping study, Nugroho et al. [37] observed that Git’s three alternative *diff* algorithms have seen limited adoption in existing works. They further investigated the impact of Myers and *Histogram* on three major applications: code churn metrics, the *SZZ* algorithm [44], and patch extraction. Their findings revealed that *Histogram* exhibits superior performance in describing code changes.

Over the past decade, numerous studies have explored the use of differencing algorithms for tracking code locations. Godfrey and Zou [18] introduced a method to detect merging and splitting of source code entities. Reiss [42] conducted a comprehensive evaluation of methods for maintaining source locations, finding that *W_BESTI_LINE*, which uses Levenshtein distance for string comparison, incorporates context lines, and ignores indentation, matches the effectiveness of other methods while offering speed and low storage requirements. Canfora et al. [3, 4] proposed *ldiff*, a differencing tool that can track line updates and movements. Based on the results of Unix *diff*, it compares similarity across all possible hunk pairs between original and new versions. For the most similar hunks, it further analyzes individual lines to detect changes. Asaduzzaman et al. [2] proposed *LHDiff*, which employs locality-sensitive hashing and normalized Levenshtein edit distance to compute content similarity by considering both content and context.

Despite these advancements, current text-based differencing techniques still suffer from a significant limitation: they are incapable of identifying block-level *EAs*, such as block moves and copies, which are routine in developers’ coding practices. In this paper, we propose a text-based differencing algorithm capable of identifying block-level *EAs*, as well as five types of line-level *EAs*.

2.2 Tree-based Code Differencing

Since source code can be represented as ASTs, tree differencing techniques can be used to compute AST differences and derive fine-grained code changes. Hashimoto and Mori [22] proposed *Diff/TS*, a tool that visualizes *EAs* through detailed structural analysis of source code trees. Spacco et al. [45] introduced *SDiff*, a hybrid differencing technique that combines line-based and AST-based approaches. Building on an existing tree differencing algorithm [5], Fluri et al. [13] developed *ChangeDistiller* to extract fine-grained source code changes.

Among various AST-based code differencing algorithms, *GumTree* [11] is an influential one. Its node-mapping process consists of two phases: 1) a greedy top-down phase to find isomorphic subtrees, and 2) a bottom-up phase to match tree nodes. Evaluation results demonstrate that *GumTree*’s differencing outputs are often more comprehensible than those of Unix *diff*. To address the limitation of *GumTree* in handling large code files, Falleri et al. [10] proposed *gumtree-simple*, an improved heuristic of *GumTree* that is not only significantly faster but also produces smaller and more understandable *ESs*. In recent years, many studies on AST-based differencing have been carried out based on *GumTree*. Based on *GumTree*, Matsumoto et al. [34] leveraged both AST and line differences to generate more interpretable *ESs*. Building on *GumTree*, Frick et al. [15] introduced *IJM*, a differencing approach that can generate more accurate and compact *ESs* capturing developers’ intent. Dotzler and Philippsen [8] proposed *MTDIFF* that refines the heuristics of *GumTree* to find more move actions. Building on *GumTree*, Dilavrec et al. [7, 29] proposed *HyperDiff*, an approach that addresses the scalability issue of computing diffs over large code histories. Huang



Fig. 2. Examples of block-level EAs

et al. [49] proposed CDiff that can generate concise, linked, and understandable code differences with a granularity between code differencing and code change summarization methods. Fan et al. [12] proposed a hierarchical approach to automatically compare the similarity of mapped statements and tokens across different AST mapping algorithms. Their experimental results reveal that state-of-the-art AST mapping algorithms still produce a considerable number of inaccurate mappings.

As evident from the preceding review of relevant studies, recent research on code differencing has primarily focused on AST-based approaches. Compared with text-based differencing algorithms, AST-based methods can extract fine-grained, syntax-aware changes, and scholars often leverage such change information for research purposes [14]. In the code review scenario, however, developers typically need to first grasp the information of textual change at the surface-level. Furthermore, tree-based differencing approaches rely on language-specific parsers [11]; yet modern software projects often involve multiple programming languages, and these languages undergo continuous evolution. This introduces practical barriers in real-world software development contexts and may even lead to analysis failures. For these reasons, traditional text-based differencing tools continue to be widely used in real-world development practices [16].

3 The BDiff Algorithm

3.1 Approach Overview

Before developing an effective code differencing algorithm, we first need to figure out what constitutes a high-quality code differencing result. Regarding this question, there is a well-established consensus in prior research [26, 37]: (1) from an internal perspective, the result should have a minimal *ES* to reduce the time developers spend reading changed code; and (2) from an external perspective, it should systematically present changes to help developers understand the actual changes made and their impacts on the code. This leads us to a second question that we need to clarify: What types of *EAs* do developers perform during code editing? From the perspective of *EA*

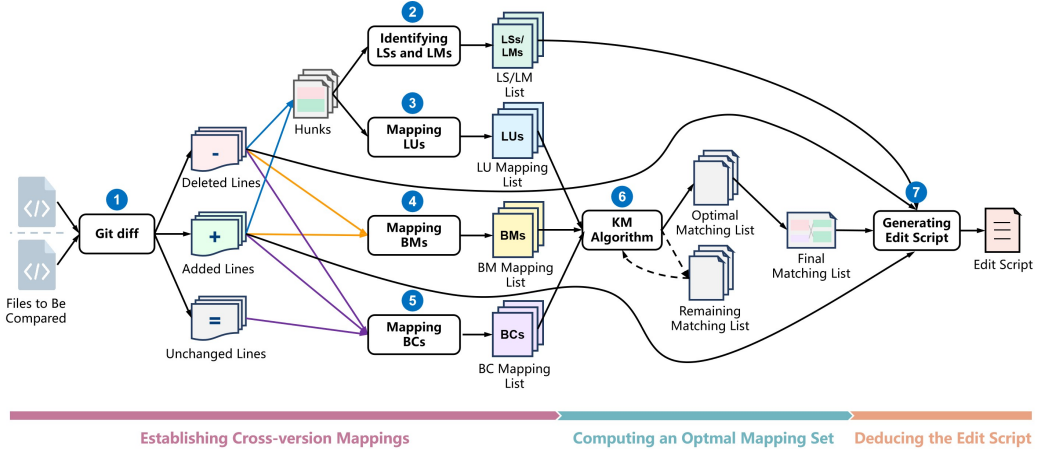


Fig. 3. The BDiff algorithm framework

granularity, we classify these *EAs* into two categories: line-level *EAs* and block-level *EAs*. Line-level *EAs* operate on individual lines of code (LOC) and include the following types:

- **Line Deleting (LD)**: removing an entire line.
- **Line Adding (LA)**: inserting a new line.
- **Line Updating (LU)**: modifying a portion of a single line, including three specific scenarios: deleting a segment of the line, inserting a segment into the line, or replacing one segment with another.
- **Line Splitting (LS)**: dividing a single line of code into multiple lines.
- **Line Merging (LM)**: combining multiple LOCs into a single line.

Block-level EAs refer to the *EAs* that operate on a set of consecutive lines as a whole, with the relative indentation between lines within the block remaining unchanged. They include two types:

- **Block Moving (BM)**: relocating a contiguous code block to another position in the same file with optional indentation adjustments or only adjusting its indentation level, e.g., moving a code block into a if-condition block for defensive programming.
- **Block Copying (BC)**: duplicating a contiguous code block to a new location in the file with optional indentation adjustments. e.g., duplicating and modifying a method definition block for overloading. Prior research [1] has shown that 64% of copy-and-paste operations in software development occur within a single source file.

As illustrated in Figure 2, the lines 4–9 in *Block 1* are all indented right by one level (equivalent to four spaces), which constitutes a *BM*. In contrast, the lines in *Block 2* and *Block 3* are also indented right, but with different indentation levels, which results in their classification as two separate *BMs*. The minimum number of non-blank lines required to form a code block is configurable, with a default value of two. Notably, unlike prior research on source location tracking, we do not identify one-line moves as *EAs*, because they are prone to false positives. Additionally, block-level *EAs* are frequently accompanied by *LUs*. For example, when duplicating a method block to implement overloading, developers typically modify parameters or return types within the copied block. To address this common scenario, BDiff supports the identification of composite block-level *EAs*, i.e., block-level *EAs* combined with intra-block *LUs*.

Figure 3 presents an overview of BDiff. The inputs to BDiff are the two versions of a source code file to be compared. From now on, we refer to the original and modified versions as the left and

right versions, respectively. Overall, BDiff is composed of three phases. First, we establish line-level and block-level mappings between the left and right versions. We apply the Git diff command to extract the deleted, added, and unchanged lines ①, as prior studies [2, 42] have demonstrated its high accuracy and efficiency for this task. The deleted and added lines are in the left and right versions, respectively. Since Git diff can be configured to use four different diff algorithms (as introduced in Section 2.1), we evaluate the performance of BDiff on our dataset using each of these algorithms to identify the optimal one. Within each hunk (i.e., a maximal sequence of consecutive deleted and/or added lines describing closely related changes) generated by Git diff, we identify *LSs* and *LMs* and remove the corresponding lines from the original deleted/added line lists and their associated hunks ②. For the resulting hunks, we then compare the remaining deleted and added lines to produce a *LU* mapping list ③. By comparing all deleted and added lines across the entire file, we obtain a *BM* mapping list ④. Since copy actions can originate from either deleted lines (i.e., copying followed by deletion) or unchanged lines, we compare all lines in the left version with the added lines in the right version to obtain a *BC* mapping list ⑤. The mappings within and across the three lists (*LU*, *BM*, and *BC*) may overlap. In the second phase, we therefore synthesize these lists and iteratively apply the Kuhn-Munkres (KM) algorithm to find the optimal final matching set, which includes *LAs*, *BMs*, and *BCs* ⑥. Finally, using the identified *LSs* and *LMs*, the final matching set, and the remaining *LAs* and *LDs* as inputs, we deduce the final *ES* ⑦.

3.2 *LSs* and *LMs* Identification

LSs and *LMs* occur within individual hunks. Since these two *EA* types are reciprocal and symmetric, we only detail the detection process for *LSs*: a single line in the left version is exactly equal to the concatenation of several consecutive lines in the right version. Unlike *LHDiff* [2], which relies on text similarity and a threshold to identify "possible" *LSs* that may include line modifications, our approach identifies *LSs* and *LMs* that are "exact" splits or merges, with no additional changes, to ensure high identification accuracy. To achieve this, we repeatedly check whether the current left line starts with the content of the current right line. If it does, and the two lines are not identical, we update the left line by removing the matched prefix and continue checking against the next right line in the hunk. If, at any point, the processed left line exactly matches the current right line, we have found a *LS*, i.e., the original left line is mapped to all the right lines involved in this process. We set a maximum number of detection attempts for *LSs* and *LMs*, with a default value of 8 [2], which is configurable by the user. Because *LSs* and *LMs* exhibit a strong correlation between the left and right lines, with a high probability of representing the actual *EAs*, we directly identify them as the final *EAs*. Therefore, once a *LS* or *LM* is found, the corresponding lines are removed from the deleted/added line lists and their associated hunks, so they are not considered in subsequent steps. If a right line is blank, we simply move to the next right line without incrementing the detection attempt counter.

3.3 Max Intersection Removal-based *LU* mapping

Like *LSs* and *LMs*, *LUs* occur within hunks. For each hunk remaining after the *LS/LM* identification phase, we compare each deleted line with each added line to identify all potential *LU* mappings. We adopt the *W_BEST_LINE* method [42] to determine whether a deleted line and an added line form a *LU* mapping (see Algorithm 1). This technique combines content and context similarity to track source lines, with prior studies [2, 42] validating its effectiveness and efficiency. In our implementation, content similarity is measured using Levenshtein ratio, while context similarity is computed as the proportion of matched context lines relative to all context lines in a window. The context window includes 4 lines before and after the target line. The combined similarity score for each line pair is calculated as $0.6 \times \text{content similarity} + 0.4 \times \text{context similarity}$ [2, 42];

Algorithm 1: The LU mapping algorithm

Data: A list of lines in the left version $\mathcal{L}$, a list of lines in the right version $\mathcal{R}$, a list of successive deleted lines $\mathcal{D}$ and a list of successive added lines $\mathcal{A}$ within a hunk, context length $ctxLen$, line similarity weight $lineWgt$, combined similarity threshold $simThres$, and an empty LU-mapping dictionary $\mathcal{M}$

Result: The LU-mapping dictionary $\mathcal{M}$

```

1  if  $\mathcal{D} \neq \emptyset$  &&  $\mathcal{A} \neq \emptyset$  then
2      foreach  $r \in \mathcal{A}$  do
3          foreach  $l \in \mathcal{D}$  do
4               $s \leftarrow W\_BEST\_LINE(l, r, \mathcal{L}, \mathcal{R}, ctxLen, lineWgt)$ ;
5              if  $s \geq simThres$  then
6                   $\mathcal{M}[(l, r, s)] \leftarrow \emptyset$ ;
7              end if
8          end foreach
9      end foreach
10     foreach  $(l_1, r_1, s_1) \in \mathcal{M}$  do
11         foreach  $(l_2, r_2, s_2) \in \mathcal{M}$  do
12             if  $(l_1 - l_2) * (r_1 - r_2) < 0$  then
13                 add( $\mathcal{M}[(l_1, r_1, s_1)], (l_2, r_2, s_2)$ );
14             end if
15         end foreach
16     end foreach
17     /* Sort  $\mathcal{M}$  in ascending order, where the primary key is the number of intersections, and
18        the secondary key is the combined similarity score. */
19      $\mathcal{M}.sort(sortKey = (size(\mathcal{M}.value), \mathcal{M}.key[2]))$ ;
20     while  $\mathcal{M} \neq \emptyset$  do
21         if  $\mathcal{M}[-1].value \neq \emptyset$  then
22              $lastLU \leftarrow pop(\mathcal{M}[-1])$ ;
23             foreach  $m \in \mathcal{M}$  do
24                 if  $lastLU.key \in \mathcal{M}[m]$  then
25                      $\mathcal{M}[m].remove(lastLU.key)$ ;
26                 end if
27             end foreach
28         end if
29         if  $\mathcal{M}$  does not contain intersections then
30             break;
31         end if
32          $\mathcal{M}.sort(sortKey = (size(\mathcal{M}.value), \mathcal{M}.key[2]))$ ;
33     end while
34 end if
  
```

line pairs with a combined score of at least 0.5 are added to the *LU* mapping list. All parameters mentioned, including context window size, content similarity weight, and combined score threshold, are configurable. Through this approach, candidate *LU* mappings may intersect within a single hunk, which contains line movements and thus violates our constraints. To resolve this issue while maximizing the number of valid *LUs* extracted, we calculate the intersection count for each mapping and sort mappings in descending order of their intersection counts. We then iteratively remove the mapping with the highest intersection count, update the intersection counts of remaining mappings, and repeat until no intersecting mappings remain. The resulting list constitutes the final *LU* mapping list. In the mapping list, the mappings may contain conflicts where multiple mappings share the same line from either the left or the right version. These conflicts will be systematically resolved in the subsequent *KM* step ⑥.

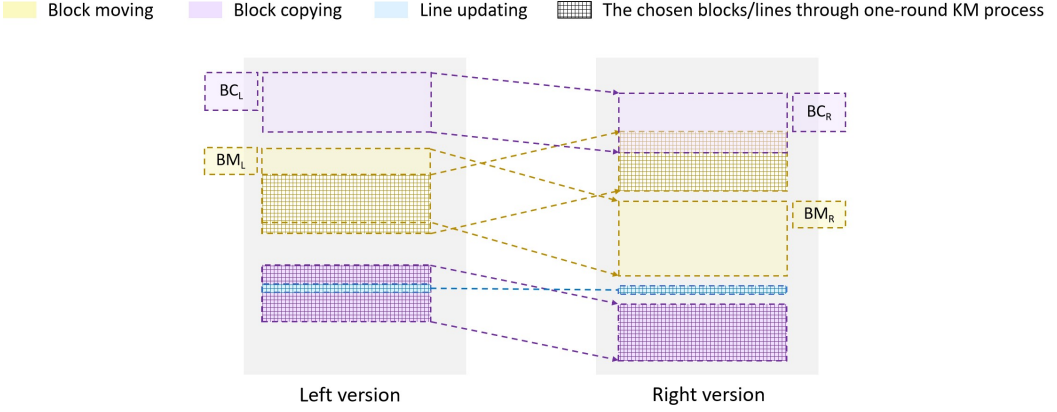


Fig. 4. The multiple mapping problem

3.4 Mapping BMs and BCs

We identify the *BM* mapping list by comparing consecutive added lines with consecutive deleted lines. Since developers may first copy a block and then delete the original lines, we derive the *BC* mapping list by comparing consecutive added lines with all consecutive lines in the left version. We determine whether a “left” line and a “right” line matches when they are identical or their Levenshtein ratio exceeds 0.6 [2, 42]. We record line pairs that are not identical, i.e., the *LU*s associated with *BM*s or *BC*s. We allow that a block starts with a blank line and contain blank lines, which aligns with actual developer practice and can reduce the *ES*. Because developers typically perform *BM*s and *BC*s on meaningful code, we exclude lines containing only stop words (e.g., ‘{’) from the block-size calculation by default. Each time a *BC* or *BM* is found, we compute a weight for it; the details are described in Section 3.5.

3.5 KM-based Optimal Mapping Computing

Through the previous two procedures, we obtain three lists of candidate mappings: *LU*s, *BM*s, and *BC*s. These mappings may overlap: both internally within a list, and externally, among the three lists (as illustrated in Figure 4). However, to form a valid *ES*, each deleted line in the left version except for *BC* and each added line in the right version must be assigned to exactly one source and one target, respectively. To resolve these conflicts, we merge the three mapping lists into a single list, and iteratively apply the KM algorithm [27, 35] to find the optimal matching set. The KM algorithm is an efficient combinatorial optimization method for finding the maximum or minimum weight matching in a bipartite graph, ensuring an optimal one-to-one correspondence between two disjoint vertex sets based on edge weights. Aligning with the characteristics of good differencing results outlined in Section 3.1, our objective is to minimize the *ES* size while preserving the original edit intent as much as possible.

We model the block and line mappings as a weighted bipartite graph $G = (V, E)$, where V is the set of vertices (i.e., the mapped blocks or lines) and E is the set of edges (i.e., the mapping relations). The vertex set V is partitioned into two disjoint subsets, V_1 and V_2 , where V_1 contains the mapped blocks or lines from the left version and V_2 contains those from the right version. Each edge $e \in E$ is defined according to Equation (1).

$$e = (u, v, w) \quad (1)$$

Algorithm 2: Handling multiple mappings based on the KM algorithm

Data: A *LU* mapping list $\mathcal{LU}$, A *BM* mapping list $\mathcal{BM}$, A *BC* mapping list $\mathcal{BC}$, a list of lines in the left version $\mathcal{L}$, a list of lines in the right version $\mathcal{R}$, minimal *BM* block length minBM , and minimal *BC* block length minBC

Result: The final optimal matching list $\mathcal{M}$

```

1  $M_a \leftarrow \mathcal{LU} \cup \mathcal{BM} \cup \mathcal{BC};$ 
2 if  $M_a \neq \emptyset$  then
3    $M_k, M_r \leftarrow km(M_a, \mathcal{L}, \mathcal{R}, \text{minBM}, \text{minBC});$ 
4    $\mathcal{M} \leftarrow \mathcal{M} \cup M_k;$ 
5   while  $M_r \neq \emptyset$  do
6      $M_k, M_r \leftarrow km(M_r, \mathcal{L}, \mathcal{R}, \text{minBM}, \text{minBC});$ 
7      $\mathcal{M} \leftarrow \mathcal{M} \cup M_k;$ 
8   end while
9 end if

```

where $u \in V_1, v \in V_2$, and w is the weight assigned to the vertex pair (u, v) . Using this weighted bipartite graph model, our problem reduces to finding a minimum weight matching, which can minimize the *ES* size while maximally reflect the original edit intent. To this end, we calculate the weight $w_e^{(b)}$ for *BM* or *BC* using Equation (2).

$$w_e^{(b)} = \frac{\text{EditTimes}(e)}{\text{Len}(u)} + \frac{1 - \text{CtxSim}(u, v)}{10} + \frac{\text{Dist}(u, v)}{100} \quad (2)$$

where:

- $\text{EditTimes}(e)$ computes the number of edits required for mapping e . When other factors are equal, mappings with fewer edits are prioritized during the conflict resolution in the KM process, because they are associated with shorter *ES* and are more likely the actual *EAs*. We initialize the values for *BM* and *BC* mappings to 2 and 3, respectively, as the former requires fewer edits than the latter. Since *BMs* or *BCs* may be accompanied by *LUs* and indentation changes, the value is incremented by 1 if they involve indentation changes or are associated with a *LU*.
- $\text{Len}(u)$ calculates the LOC of u (or v) in the mapped block. When resolving overlapping vertices with otherwise equivalent weights, priority is given to the vertex with greater block length.
- $\text{CtxSim}(u, v)$ measures the context similarity between u and v . For overlapping vertices, higher context similarity indicates a stronger alignment with the original edit intent. We measure the context similarity by calculating the Levenshtein ratio of the 4 lines above and below both u and v .
- $\text{Dist}(u, v)$ quantifies the relative line distance between u and v . Based on the principle of locality, where programs tend to reuse data and instructions near recently accessed ones, closer mappings are more likely to reflect the original edit intent. This metric is derived from Git diff results, calculated as the sum of unchanged lines plus the maximum of added and deleted lines between the two start lines of the u and v .

Overall, we designed the weight calculation method in this way to reflect the priority of the three components, where the order is: block length > context similarity > relative line distance. This is because, in most cases, the orders of magnitude of the first, second, and third components in Equation (2) are 0.1, 0.01, and 0.001, respectively. Similarly, we calculate the weight $w_e^{(l)}$ for *LU* using Equation (3). Since *LUs* inherently occur within a single hunk, we do not include the distance factor when calculating their weight.

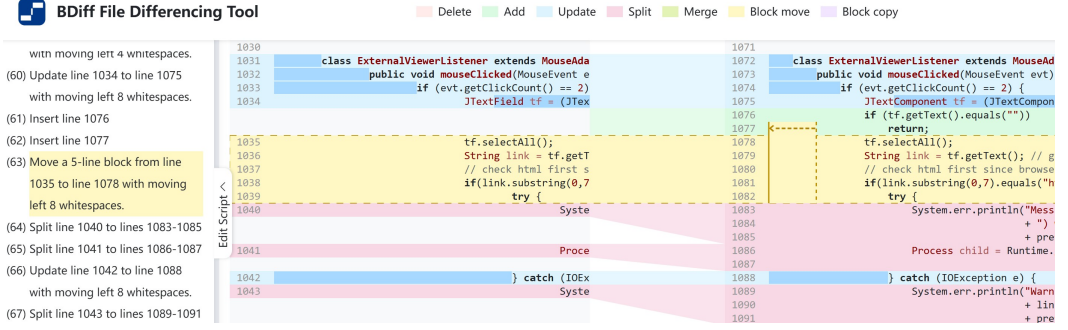


Fig. 5. A screenshot of BDiff webpage

$$w_e^{(l)} = \frac{EditTimes(e)}{Len(u)} + \frac{1 - Sim(u, v)}{10} \quad (3)$$

where:

- $EditTimes(e)$ is set to 1, as LU mappings have an initial value of 1 and involve no additional edits.
- $Len(u)$ equals 1, since the LU consists of a single line of code.
- $Sim(u, v)$ corresponds to the W_BEST_LINE value [42] associated with mapping e .

When constructing the bipartite graph, we treat left overlapping vertices that share deleted lines or right overlapping vertices that share added lines as a single vertex. As illustrated in Figure 4, the two left BM vertices overlap and thus form one left vertex; similarly, a right BM vertex and a right BC vertex that overlap form one vertex. Because BC can be superimposed with other EAs , we treat the left vertices of BCs as additional vertices. For instance, in Figure 4 the left vertex of LU and the left vertex of the BC at the bottom that overlap are two distinct vertices. However, this approach can lead to a fragmentation issue: after one round of the KM process, several non-overlapping “pieces” of the unselected blocks or lines may remain. These pieces still need to be considered for inclusion in the ES , such as the mappings $BC_L \rightarrow BC_R$ and $BM_L \rightarrow BM_R$ in Figure 4. Note that these remaining pieces may still overlap. To address this, we iteratively apply the KM algorithm to the remaining pieces to find an optimal matching in each round, adding the results to the final matching set until no pieces remain (Algorithm 2).

4 The BDiff Tool

Based on the algorithm described in the previous section, we have implemented a web-based, open-source visual differencing tool². Our tool allows developers to specify the EAs to be identified and to configure algorithm-related parameters, e.g., tab size, the synthetic line-similarity threshold, the minimum block lengths for BM and BC , etc. To facilitate developers in reviewing differences, our tool supports bidirectional navigation between differences and the EAs in the ES (see Figure 5). We offer five ways to use the BDiff tool: (1) through the website³, where developers can select two files to compare via a browser and view the results; (2) through Git, by configuring BDiff as the

²<https://github.com/bdiff/bdiff>

³<http://www.bdiff.net/>

*difftool*⁴ and viewing results via Git commands; (3) by exporting the differencing result page using our program⁵; and (4) by directly obtaining the *ES* data through the provided API⁶.

5 Evaluation

5.1 Evaluation Setup

We now present the empirical evaluation of BDiff. Our goal is to assess its performance against existing state-of-the-art code differencing tools. We first focus on *ES* size, a key quality metric; shorter *ES* sizes generally reduce the time developers need to understand the changes [11], aligning with our first criterion for good differencing results (Section 3.1). This leads to our first research question:

RQ1: Does BDiff produce smaller *ES* sizes than existing code differencing tools?

Another evaluation criterion for a differencing tool is whether it helps developers efficiently understand the actual changes made and their impacts on the code, i.e., the perceived quality of the differences. Thus, our second research question is:

RQ2: Is the perceived quality of BDiff's output higher than that of existing code differencing tools?

Through RQ2, we assess the perceived quality of BDiff's results relative to other tools. Additionally, we want to evaluate the correctness of BDiff's results, i.e., the degree to which its output matches the original edits:

RQ3: What is the correctness of the *ES*s computed by BDiff?

The previous research questions focus on functional aspects. Our fourth question evaluates runtime performance:

RQ4: How does BDiff's runtime performance compare with existing code differencing tools?

To answer these questions, we used a mixed-method approach that combined qualitative and quantitative methods. For RQ1 and RQ4, we conducted quantitative analyses of relevant metrics. For RQ2, we performed a manual evaluation with 10 raters to collect qualitative feedback. For RQ3, we implemented an algorithm that generates random changes to code files, producing ground-truth *ES*s, and then we compared the *ES*s computed by BDiff against the ground-truth *ES*s.

We selected five baseline tools: (1) Git diff (2.41.0.windows.1, default Myers algorithm) [17], the most widely-used text-based differencing tool, (2) ldiff (1.0.8) [32], a text-based differencing tool that can identify line updates and moves, (3) GumTree (v4.0.0-beta4, in which *Simple* [10] is the default matcher) [20], the most advanced state-of-the-art tree-based differencing tool, (4) GPT-5-mini, a recent closed-source LLM [38], and (5) Qwen3-32B, an advanced open-source LLM [41]. We prompt the two LLMs to generate *ES*s in the same *EA* types and format with BDiff. LHDiff [2] was excluded as it produces line mappings for the line-tracking scenario rather than an *ES*. In the BDiff algorithm, we use Git diff (2.41.0.windows.1) to obtain the deleted and added lines ①.

5.1.1 Dataset. We evaluated BDiff's performance on a dataset derived from real-world change histories. Although BDiff is text-based and language-independent, we aimed to create a dataset whose programming languages represent the major execution paradigms, as different languages exhibit distinct textual structures and formatting conventions. We therefore selected Java, Python, and XML, which represent compiled, interpreted, and declarative (markup) paradigms, respectively. For Java and Python, we used the *GhPython* and *GhJava* datasets [10], which were constructed via stratified sampling to evaluate GumTree. Each dataset is built from the commit histories of 10 popular GitHub projects, containing approximately 100 file pairs (i.e., 100 cases) per project. For XML, we followed the same methodology to create a comparable dataset, named *GhXML*, from 10

⁴<https://github.com/BDiff/BDiff-Git>

⁵<https://github.com/bdiff/bdiff-Visualization-Exporter>

⁶<https://s.apifox.cn/aa189f38-f9da-4ff2-9e08-bc65663b7708/api-198645561>

Task Description	You are an expert in code differencing. You will be given two pieces of code: ...
Rule Instruction	1. Global Resolution Rules (MUST follow) 1.1 Mutual exclusivity per change. ... 2. Line-level Edit Actions (EAs) 2.1 Line Deleting (LD) - Deleting an entire line. ... 3. Block-level Edit Actions (EAs) 3.1 Block Moving (BM) - Moving a code block or changing indentation. ...
Example	Code A: <Code A example> Code B: <Code B example> <Example edit script>
Input	Code A: <Code A> Code B: <Code B>

Fig. 6. Overview of the prompt template.

popular GitHub projects that use XML. Consequently, we have a total of 2,977 cases with textual differences, containing 991, 999, and 987 cases for Python, Java, and XML, respectively.

For each case, we ran BDiff and all baseline tools to compute the *ES* on a desktop computer with a 3.6GHz Intel Xeon W-2223 CPU with 16GB of RAM. We set the analysis time limit at one hour. Consequently, among the 2,977 cases that have textual differences, Git diff successfully analyzed all the file pairs; ldiff and BDiff could not complete the analysis process in the time limit for one file pair (scikit-learn/scikit-learn, cd076af, pendigits.py), which stores a large number of pure numbers; GumTree encountered 69 analysis errors on XML files and reported no differences in 128 cases, as it cannot detect formatting changes (e.g., the cases in Appendix B.2); GPT-5-mini can successfully analyze 2,797 cases (the remaining 200 cases contain 183 no-difference cases and 17 failure cases); and Qwen3-32B can successfully analyze 2,443 cases (the remaining 554 cases contain 52 no-difference cases and 502 failure cases). To ensure a fair comparison, we retained 2,244 cases, where all tools successfully detected differences, as the dataset for our quantitative analysis.

5.1.2 Base diff algorithm selection. The BDiff algorithm builds upon the deleted and added lines identified by a base diff algorithm. Consequently, different base diff algorithms can yield different results (e.g., different change hunks [37]) for BDiff. To evaluate this influence and select an optimal base algorithm, we computed the *ES* for all file pairs in our dataset using BDiff with each of Git’s four diff algorithms: Myers, *Minimal*, *Patience*, and *Histogram*. We used *ES* size as the quality metric. Consequently, among the 2,976 cases, 144 exhibited discrepancies in *ES* size across algorithms. In these 144 cases, *Histogram* produced the smallest *ES* most frequently (75 times); meanwhile, it also achieved the smallest average *ES* size (16.28) and the shortest average running time (0.12s). Therefore, we selected *Histogram* as default base diff algorithm for BDiff. Our results align with findings in [37] that *Histogram* outperforms Git’s other three algorithms.

5.1.3 Prompt design. We prompted the LLMs to replicate BDiff’s functionality and *ES* format. Following established practices for instructing LLMs [21], we employed a one-shot prompting strategy that integrates the task description, rule instructions, an example, and the input into a single unified prompt. Figure 6 shows the prompt template. The *Task Description* (in the system prompt) defines the LLM’s role as a code differencing expert, tasked with analyzing *Code A* (the left version) and *Code B* (the right version) to output a JSON-like Python list representing the *ES*. The *Rule Instruction* specifies eight global resolution rules (e.g., minimizing *ES* length, enforcing length constraints for the blocks) and detailed formatting requirements for all the seven types of EAs. The *Example* section provides a real-world case with *Code A*, *Code B*, and the corresponding *ES*,

explicitly mapping code changes to compliant *EAs* and illustrating how to apply the prompt's rules and formats. Finally, the *Input* section uses a structured placeholder format that clearly defines the two code versions to be compared.

5.2 Manual Evaluation

To answer *RQ2*, we consider the viewpoint of the developer. For her, what matters is that the computed *ES* helps her efficiently understand the changes [11]. Accordingly, we designed a manual evaluation experiment to compare developers' perceptions of BDiff's results against those of the baseline tools. We recruited 10 raters: 5 experienced industry software engineers and 5 graduate students majoring in Software Engineering. They evaluated a set of cases containing results from different differencing tools. To ensure representativeness, we used stratified sampling [39] to randomly select 100 Python, 100 Java, and 100 XML diff cases from the 2,997 cases in our dataset. For each language, 50 cases included block-level *EAs* and 50 did not. For each case, we generated differencing results using BDiff and all baseline tools. While BDiff and GumTree natively provide web-based visualizations, other tools only output *ESs* in text format. To standardize the evaluation interface, we developed an independent diff-result visualization tool⁷ that accepts BDiff's *ES* format as input and can export visualized differencing results as web pages. We then adapted the results of Git, LLMs, and ldiff⁸ to BDiff's *ES* format and obtained visual diff results of text-based and LLM-based differencing tools in a consistent style. This enables raters to compare all tools' outputs conveniently and intuitively while mitigating bias from inconsistent result presentation. Additionally, we provided commit messages for each case, which describe the change intent and facilitate meaningful assessment of differencing result quality.

The evaluation was conducted centrally in an online meeting. Raters were randomly divided into two groups (A and B) of five. Each group evaluated the complete case set, but each rater received a random subset of 60 cases (30 with block-level *EAs* and 30 without). At the meeting, the first author of this paper introduced the experiment's background, objectives, and requirements, and emphasized that the results would be published to ensure accountability and objectivity. Raters then independently reviewed and scored the differencing results from all six tools for each case. Scores ranged from 1 to 6 (with higher values indicating better quality), representing a ranking of the tools' outputs; ties were allowed. Due to random selection, some cases might not have been successfully analyzed by certain tools (e.g., LLMs); for these, raters were instructed to assign a score of 1 to the corresponding tool. For each case, a comment field collected general feedback. For the 30 cases involving block-level *EAs*, an additional field invited specific comments on the correctness of these block-level *EAs*. After the evaluation, raters discussed the tools' performance and provided valuable suggestions for improving BDiff, offering insights for our qualitative analysis.

5.3 Mutation-based Evaluation

To objectively assess the correctness of the generated *ES* of BDiff (*RQ3*), we employed a mutation-based evaluation method [2, 43, 46]. This method involves randomly injecting changes into a given code file and then analyzing how well BDiff's computed *ES* aligns with the ground-truth *ES*. We implemented an algorithm that takes a file as input, randomly mutates it using the seven *EA* types that BDiff can identify (Section 3.1), and outputs the modified file along with the ground-truth *ES*. To ensure the accuracy of the evaluation results, we configured the parameters of the mutation algorithm to be identical to those used by BDiff.

⁷<https://github.com/bdiff/Textual-Diff-Visualization-Exporter>

⁸The line movements in ldiff are treated as single-line *BMs* in BDiff

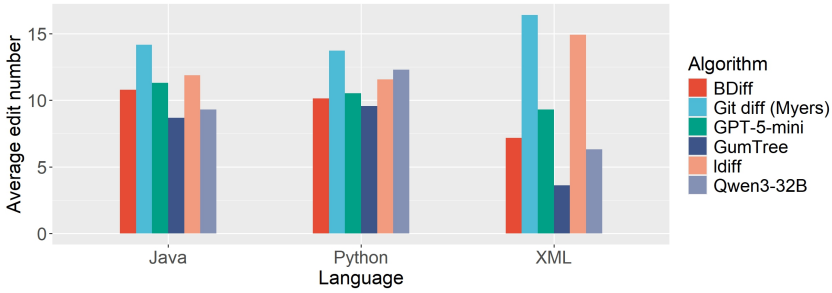


Fig. 7. Average edit-script size of different languages

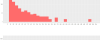
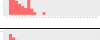
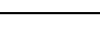
Generally, the algorithm consists of the following three steps: 1) Data preparation. Using the input file (the left version), we create a list called *right list* that contains a sequence of dictionaries, one for each line in the input file. This list represents the current state of the file during the mutation process and corresponds to the right version. The list's indices are used to determine the current line numbers, as these will dynamically change after *EAs* are applied. In addition to the line content, each dictionary stores the corresponding line number from the left version, which is used to locate the correct position for subsequent mutations. 2) Code mutation. We first generate a random upper bound for the number of *EAs*. The process then iteratively generates random changes in a top-to-bottom order, starting from the beginning of the file. In each iteration, we randomly select a position and an *EA*, generate the corresponding change, and locate and modify the target line(s) in the *right list*. The process terminates when the number of *EAs* exceeds the maximum limit or the edit position moves beyond the end of the file. 3) *ES* and right-version generation. Using the final *right list*, we update the line numbers recorded in the *EAs* and output the final *ES* along with the right-version file.

We executed this algorithm to automatically mutate a random left version of the 2,997 cases in our dataset, obtaining the corresponding generated right versions and their ground-truth *ESs*. We then ran BDiff on these left/right version pairs to obtain the computed *ESs*. Finally, we compared the *EAs* in BDiff's computed *ESs* with those in the ground-truth *ESs*. We define two *ESs* as equivalent if they contain exactly the same *EAs*. The order of *EAs* within an *ES* is not considered, as it is not unique (e.g., the order of two *BCs*). To formalize our evaluation criteria, we define two *EAs* as equivalent if they match in the following three aspects: (1) *EA* type; (2) source line number(s) (applicable to non-*LA* *EAs*); and (3) target line number(s) (applicable to non-*LD* *EAs*). To ensure the reliability of our findings, we repeated the entire process for 3,000 mutations. We computed the matching rate, defined as the proportion of ground-truth *EAs* that BDiff can correctly identify, to evaluate BDiff's correctness in *ES* computation. Consequently, among the mutated 3,000 cases, the mean and median sizes of ground-truth *ES* are 14.1 and 5.0, respectively, which are close to the mean and median (16.3 and 4.0) of the *ES* sizes identified by BDiff in real-world dataset.

5.4 RQ1: Edit-script Size

To evaluate BDiff's performance in terms of *ES* size, we first compared the descriptive statistics of the *ES* sizes generated by BDiff and the baseline approaches. The results are shown in Table 1. Note that when calculating *ES* size, we also include *LUs* that occur within *BMs* and *BCs*. In general, BDiff generates the smallest *ES* among the three text-based code differencing algorithms. The average *ES* size of BDiff is 37.9% and 28.5% lower than that of Git diff (Myers) and Idiff, respectively. In particular, for XML files, the average *ES* size of BDiff is 56.3% and 52.0% lower than that of Git diff

Table 1. Descriptive statistics of the *ES* sizes of the differencing algorithms

Approach type	Diff algorithm	25%	50%	Mean	75%	Max	Histogram ¹
Tree-based	GumTree	1.0	2.0	7.1	6.0	460.0	
LLM-based	GPT-5-mini	1.0	3.0	10.3	10.0	490.0	
	Qwen3-32B	1.0	2.0	9.1	7.0	893.0	
Text-based	Git diff (Myers)	2.0	5.0	14.9	14.0	698.0	
	ldiff	1.0	4.0	12.9	11.0	698.0	
	BDiff	1.0	3.0	9.2	10.0	178.0	

¹ Histograms are in log scale.

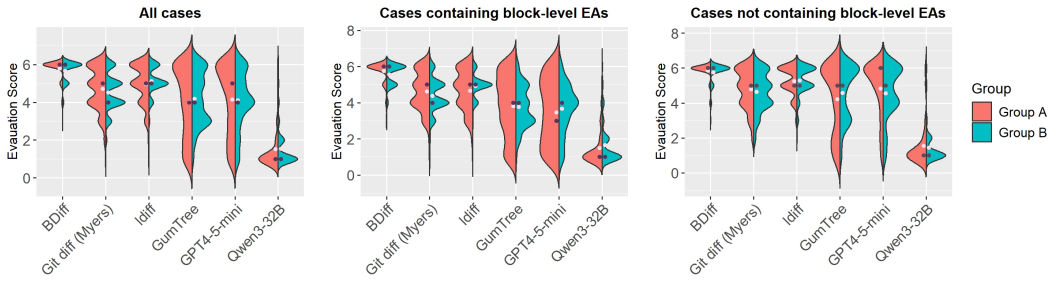
and ldiff, respectively (see Figure 7). We attribute this to the fact that XML elements often share identical tags and similar structures, leading to BDiff's *EAs* that contain more *BMs* and *BCs*. In practice, developers frequently perform such edits when modifying XML files. Specifically, in 1,709 (76.2%) cases, the *ES* from BDiff is shorter than that of Git diff; in 746 (33.2%) cases it is shorter than that of ldiff; and in 1,345 (59.9%) cases, BDiff produces an *ES* of the same size as ldiff. The average *ES* size of GumTree is 22.8% lower than that of BDiff, while the *ESs* generated by the two LLMs are generally comparable to those of BDiff. We further used the paired nonparametric Wilcoxon signed-rank test to examine the significance of the differences in *ES* size between BDiff and each of the baseline tools. The results show that all differences are significant at the 0.0001 level. The effect size is small for Myers (Cliff's δ : -0.196) and negligible for ldiff (Cliff's δ : -0.045), GumTree (Cliff's δ : 0.133), *GPT-5-mini* (Cliff's δ : 0.015), and *Qwen3-32B* (Cliff's δ : 0.110).

We manually analyzed cases where the lengths of *ESs* generated by BDiff and GumTree significantly differ and found that GumTree's shorter *ESs* are mainly attributable to two reasons: (1) GumTree identifies changes to multiple lines belonging to a single AST node as a single *EA*, e.g., updating a multi-line comment (Figure 17 in Appendix B.1) and adding a complete function definition (Figure 18 in Appendix B.1); and (2) GumTree is insensitive to code formatting changes, e.g., it does not identify *LMs* (Figure 19 in Appendix B.2), *LSs*, or changes in code block indentation (Figure 20 in Appendix B.2). Additionally, we observed that for certain *BMs* identified by BDiff, Gumtree would identify the lines within these *BMs* as individual line movements when they are not treated as an integral AST node (Appendix B.3).

Summary: In terms of *ES* size, BDiff outperforms text-based differencing tools, reducing the average *ES* size by at least 28%. It performs roughly on par with LLM-based tools, though its *ES* is 23% longer than that of the AST-based differencing tool GumTree.

5.5 RQ2: Result Quality of the Differencing Tools

To assess the reliability of the ratings, we computed the linearly weighted Cohen's Kappa (κ) coefficient to measure inter-rater agreement between the two groups of raters, who employed a 1–6 point scoring scale. The analysis yielded a weighted κ of 0.61 with a p -value < 0.001 , indicating statistically significant substantial agreement between the two rater groups [28, 47]. We first analyzed the raters' scores on the differencing results from the manual evaluation experiment (see Figure 8). Overall, the average scores given by the 10 raters to the results generated by BDiff, ldiff, Git, GPT-5-mini, GumTree, and Qwen3-32B are 5.73, 5.04, 4.61, 4.13, 4.09, and 1.53, respectively. Among BDiff's 300 results, 63.3% received the highest score of 6 from both raters,



Note: The points ● and ● in the violin plots represent the mean and median, respectively.

Fig. 8. Evaluation results of the result quality of the tools

<pre> 109 110 /** 111 * This method initializes this 112 * 113 * @return void 114 */ 115 private void initialize() { 116 this.setName("ExtensionScanner"); </pre>	<pre> 108 /** 109 * This method initializes this 110 * 111 * @return void 112 */ 113 private void initialize() { 114 this.setName("ExtensionActiveScan"); 115 </pre>
---	--

Fig. 9. A diff snippet generated from GPT-5-mini

and only 5 cases (1.7%) received a score below 5 from both. During the discussion segment, raters generally agreed that BDiff’s results were intuitive and concise. In particular, raters reported that the block-level EAs identified by BDiff are of high readability: “BDiff’s BCs are highly intuitive” [A5] and “BDiff performs best in text block handling and can recognize a variety of complex scenarios” [B5]. We also found that for a considerable number of cases with block-level EAs, BDiff’s results can intuitively reflect developers’ actual editing intentions and facilitate readers in efficiently understanding the changes, e.g., *reordering of assignment statement blocks* (Appendix A.1), *indenting a code block for conditional branching* (Appendix A.2), *reusing an XML block or a function definition* (Appendix A.3, A.4, A.5), and *reformatting code* (Appendix A.6). For all tools except for Qwen3-32B, the average scores of the cases that contain block-level EAs (i.e., the *block cases*) were lower than those of the cases not containing block-level EAs (i.e., the *non-block cases*): 5.65 vs 5.81, 4.79 vs 5.27, 4.5 vs 4.72, 3.56 vs 4.70, and 3.80 vs 4.40, and 1.57 vs 1.49 for BDiff, ldiff, Git, GPT-5-mini, GumTree, and Qwen3-32B, respectively. We believe this is mainly due to the complexity of the block cases: the average ES sizes (computed by Git diff) of the block cases and non-block cases are 75.0 and 11.1, respectively. Raters also reported that in some complex change scenarios, the identified block-level EAs are inaccurate and can hinder comprehension, e.g., “There are too many BCs and BMs, which is confusing and hard to understand” [A5], and “Various EAs are intertwined, making it rather chaotic. It’s not as easy to understand at a glance as Git” [A2, A3, B1].

Due to its ability to identify line changes, ldiff received higher average scores than Git. Raters reported that “Git only identifies LDs and LAs, which makes it difficult to understand in scenarios involving a large number of consecutive EAs” [A4]. Notably, for the same case, ldiff may split a single BM (as identified by BDiff) into multiple line moves, resulting in “Longer ESs that are neither intuitive nor easy to understand” [A3].

GumTree generally received lower scores compared to text-based differencing tools. Our results on GumTree’s performance relative to Git diff are lower than those reported in a prior study [11].

Table 2. Experimental results of the mutation-based evaluation

Language	# Files	Ground-truth ES size		BDiff ES size		Avg. matching rate							
		Mean	Median	Mean	Median	Total	LD	LA	LU	LS	LM	BM	BC
Java	985	16.4	5.0	19.8	5.0	0.948	0.968	0.995	0.914	0.997	0.955	0.810	0.808
Python	1,015	17.0	5.0	19.4	5.0	0.953	0.948	0.995	0.948	0.986	0.937	0.832	0.824
XML	1,000	8.7	4.0	9.5	4.0	0.957	0.967	0.999	0.962	0.992	0.967	0.834	0.887
Total	3,000	14.1	5.0	16.2	5.0	0.953	0.961	0.996	0.940	0.991	0.952	0.825	0.828

We attribute this discrepancy to two possible factors: (1) the cases in the previous experiment contained fewer changes, while our experiment included cases with complex changes; and (2) prior raters were all from research teams, while half of our raters are industry engineers who are relatively unfamiliar with AST-based differencing results, despite relevant introductions before the evaluation. In our experiment, three-fifths of the raters from industry reported that GumTree’s results were difficult to understand, e.g., “*It is challenging to understand Gumtree’s results*” [A3], and “*Gumtree relies on manual correlation for side-by-side line comparison, which is user-unfriendly and lacks proper alignment*” [A5].

Finally, regarding the two LLM-based tools, the average score of the open-source model Qwen3-32B (1.53) is much lower than that of GPT-5-mini (4.13), and three raters felt that Qwen3-32B delivered the worst performance [A3, B4, B5]. The raters reported that both GPT-5-mini and Qwen3-32B exhibited numerous recognition errors and significant hallucinations in the diff generation task, e.g., “missing certain EAs” [B2], “incorrectly identifying line numbers” [B2], and “incorrectly identifying EAs” [A5]. Even in simple diff scenarios, these LLMs can introduce naive errors. As shown in Figure 9, GPT-5-mini cannot correctly identify simple *LDs* and *LAs*.

Summary: BDiff achieved the highest average score, primarily due to the superior readability of its differencing results. In contrast, the outputs from LLM-based tools received relatively low scores and were plagued by hallucinations.

5.6 RQ3: ES Correctness of BDiff

Table 2 presents the results of the mutation-based evaluation. Overall, no significant differences in ground-truth *ES* size are observed across languages. The mean and median sizes of ground-truth *ES* are 14.1 and 5.0, respectively. By comparison, the average *ES* size computed by BDiff across the mutation cases is slightly larger (16.2). Among the 3,000 mutations, the average *ES* matching rate reaches 95.3%, and 2,473 (82.4%) cases achieve a 100% matching rate, demonstrating high identification accuracy of BDiff. Among the seven *EA* types, *LA* and *LS* achieve the highest matching rates (>99.0%), while the block-level *EAs*, *BM* and *BC*, yield relatively lower rates (82.5% and 83.8%, respectively). We therefore conducted a focused analysis of BDiff’s accuracy on *BM* and *BC*. Across all mutation cases, 1,075 cases generated 1,228 *BMs*, and 1,167 cases generated 1,543 *BCs*. BDiff correctly identified 1,012 (82.4%) of the *BMs* and 1,278 (82.8%) of the *BCs*. We analyzed the misidentified cases and identified two primary causes: (1) inaccuracies in the Git diff output, such as misidentifying deleted and added lines, which propagates to BDiff’s identification of *BMs* and *BCs* (and can also lead to misidentifying of other *EAs* such as *LU*); and (2) the presence of multiple similar code blocks within the same file, where BDiff may fail to select the correct block mapping during the optimal matching step 6.

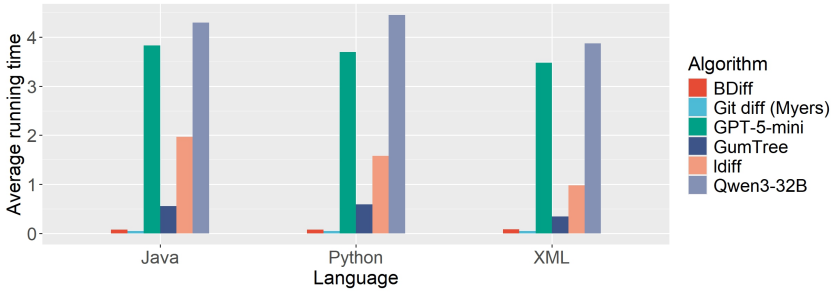


Fig. 10. Average running time of different languages

Table 3. Descriptive statistics of the runtimes (in seconds) of the diff algorithms

Approach type	Diff algorithm	25%	50%	Mean	75%	Max	SD	Histogram ¹
Tree-based	GumTree	0.426	0.637	0.643	0.764	1.950	0.256	
	GPT-5-mini	18.473	29.897	38.076	48.723	874.001	39.633	
	Qwen3-32B	16.413	37.398	66.681	89.805	1480.290	98.497	
Text-based	Git diff (Myers)	0.050	0.051	0.051	0.052	0.057	0.002	
	ldiff	0.467	0.547	3.766	1.472	946.784	27.581	
	BDiff	0.075	0.077	0.085	0.080	2.633	0.084	

¹ Histograms are in log scale.

Summary: BDiff achieves an accuracy of 95.3% in identifying ground-truth *EAs*, with fully correct *ES* identification in 82.4% of cases. Among all *EA* types, the accuracy for block-level *EAs* is relatively lower (approximately 83%), primarily due to inaccuracies in the base diff algorithm’s output and the presence of multiple similar code fragments within the same file.

5.7 RQ4: Runtime Performance

We now analyze the runtime performance of BDiff and the baseline tools on the 2,244 cases. For each case, we ran 10 times and retained the median value. For all tools, we only recorded the time taken to compute the *ES*, excluding the time spent on visualizing differencing results. For GumTree, for instance, we used the “*TextDiff*” mode to obtain the computed *ES* in text format. Overall, the tools show no significant difference in average running time across different programming languages (see Figure 10). The descriptive statistics are summarized in Table 3. As expected, Git diff is the fastest due to its simpler identification capability, with an average running time of 0.051s. BDiff ranks second, with an average running time of approximately 0.085s. BDiff also exhibits good runtime stability: its standard deviation of 0.084s is the second lowest, just above Git’s. The maximum runtime across our cases is 2.633s, which is completely acceptable in practice. The average running time of ldiff is 3.766s, but its running time is more variable, with a standard deviation of 27.581s. In our cases, 136 (6.1%) take longer than 10 seconds, which may hinder interactive use by developers [11]. By contrast, the two LLM-based tools perform extremely poorly in terms of runtime: the average running times of GPT-5-mini and Qwen3-32B are 38.076s and 66.681s,

respectively. Furthermore, only 72 cases (3.2%) for GPT-5-mini and 170 cases (7.6%) for Qwen3-32B complete within 10 seconds, making them impractical for interactive use.

Summary: BDiff exhibits short and stable runtime, with an average running time of less than 0.10 seconds and a standard deviation of around 0.08 seconds—ranking second only to Git. In contrast, the LLM-based differencing tools have average running times exceeding 30 seconds, making them impractical for interactive use.

6 Discussion

6.1 For Developers

6.1.1 Customizing BDiff for Specific Scenarios. In our evaluation experiment, we have observed the phenomenon of “one size does not fit all”: the results of BDiff generally received high ratings; however, for certain cases, BDiff exhibits suboptimal performance under the same algorithmic settings. We believe that this can be primarily attributed to the inherent inaccuracies of the mapping step in a difference computing process, in which edit process information is missing and the left version and right version may have multiple same/similar lines/fragments. In response, developers can adjust the algorithm settings for specific scenarios to obtain better diff results. For example, developers can disable the identification of line updates within block-level *EAs*, or set a larger minimum block size for *BMs* and *BCs* to reduce “overly broad” matches. Furthermore, developers can configure which *EA* types to recognize. For complex cases, they may disable specific *EAs* to simplify results to only include *LDs* and *LAs*, thereby improving results readability.

6.1.2 Using *BMs* and *BCs* to support *SE* activities. The ability to identify block-level *EAs* is a key feature of BDiff. Specifically, a *BM* associates consecutive added lines with consecutive deleted lines, while a *BC* associates consecutive added lines with consecutive lines in the left version. These block-level associations operate at a coarser granularity than the line-based associations used in traditional source tracking techniques [2]. Developers can leverage these associations to support several practical *SE* activities: (1) Defect localization. By tracking code blocks through *BMs* and *BCs*, developers can more effectively pinpoint the source of defects. In particular, *BCs* can help discover additional code blocks that may contain defects by revealing their relationship to existing code. (2) Change analysis. Changes in *BM* block positions, particularly those indicated by indentation changes, can be used to analyze modifications in the scope of related variables. Furthermore, *BCs* can reveal code clones, suggesting potential refactoring opportunities such as abstracting common code blocks into shared functions or classes.

6.2 For Researchers

6.2.1 BDiff’s support for downstream *SE* tasks. Code differencing is a foundational technique in software maintenance and evolution research. To the best of our knowledge, BDiff is the first to propose and implement block-level *EA* recognition, with comprehensive *EA* identification capabilities. This advance can significantly benefit many downstream *SE* research efforts based on differencing technology. We outline four main research directions: (1) Conflict detection. Current approaches primarily include text-based, AST-based, and semistructured methods. BDiff can provide more block-aware, accurate text-based differencing results, which can be used to improve detection accuracy and reduce false positives. (2) Change pattern mining. With BDiff’s more accurate differencing results that align with developers’ intended *EAs*, researchers can conduct more precise studies on defect repair patterns and software evolution patterns. (3) Change summary generation. Researchers can generate more accurate change summaries [30] from BDiff’s results. For example,

changes involving only *BM*, *LS*, and *LM* can produce refactoring-oriented summaries. (4) Contribution measurement. Our experimental results show that BDiff reduces the average *ES* size by 28% compared with existing text-based differencing tools. BDiff's results enable more accurate measurement of developers' code contributions. For instance, calculations involving *BM* more accurately reflect actual contributions compared to traditional metrics based solely on *LDs* and *LAs*.

6.2.2 Intelligent code differencing based on BDiff. Existing code differencing techniques employ model-driven approaches that formalize problem-solving processes through explicitly defined mathematical or logical frameworks, enabling deterministic optimization. While these methods are static and efficient, they struggle to consistently achieve optimal performance across all scenarios, i.e., the "one size does not fit all" phenomenon noted in Section 6.1.1. To address this limitation, researchers could explore integrating data-driven methods (e.g., deep learning and reinforcement learning) with BDiff's approaches and results to intelligently generate more accurate and contextually appropriate outcomes. Furthermore, our design of the weight calculation formula for the KM process is primarily based on empirical insights, lacking rigorous validation. For future work, researchers could leverage data-driven approaches to train more accurate and robust parameters.

7 Threats to Validity

Construct validity. In this paper, we use *ES* size as a metric to measure the quality of differencing results. However, a shorter *ES* may be more difficult to understand than a longer one [11] and may even be incorrect. To complement this metric, we conducted both a qualitative manual experiment and a quantitative mutation-based evaluation. In the mutation-based evaluation, we use the matching rate as a measure of the correctness of *ES* identification. This metric considers only the proportion of matched *EAs* among the ground-truth *EAs*, without accounting for over-identified *EAs*. However, this limitation had minimal impact in our experiments because 82.4% of the cases achieved perfectly matched *ESs*, and the average *ES* size of BDiff (16.2) is slightly larger than that of the actual *ES* (14.1).

Internal validity. In our experiments, we used BDiff's default settings, which incorporate multiple parameter configurations such as a maximum line count of 8 for *LS* and *LM*, and a minimum block size of 2 for *BM* and *BC* (excluding lines containing only stop words). These defaults have not been validated through dedicated experiments to confirm they yield optimal differencing results. To mitigate this threat, we referenced default parameters from prior tools (e.g., the maximum line count for *LS* and *LM* in LHDiff) and randomly selected a subset of results for manual analysis to verify parameter effectiveness. Another threat in the manual evaluation is the subjectivity of rater assessments. Raters with diverse professional backgrounds and experiences may have scored cases based on their prior familiarity with differencing tools, e.g., engineers who frequently use Git diff may be relatively unfamiliar with GumTree's AST-based approach, while graduate students often have research experience with GumTree. To address this, we: (1) introduced all tools during the evaluation briefing and analyzed examples of high-quality differencing results; (2) provided real-time clarification of raters' questions during evaluation; and (3) held post-evaluation discussions allowing raters to revise their scores before final submission. A third threat involves GumTree's XML backend, which is known to have issues. In our experiments, some XML cases triggered parsing errors during GumTree analysis. We mitigated this by excluding error cases from the dataset and validating the correctness of remaining results through random sampling.

External validity. Our dataset comprises 2,244 cases across three languages: Java, Python, and XML. However, we cannot claim these changes are representative of all changes in these languages [10]. Moreover, while BDiff is language-independent, we cannot guarantee our results will generalize to other languages.

8 Conclusion

We have developed BDiff, a novel text-based and block-aware approach for accurate code differencing. BDiff builds upon the results of traditional differencing algorithms and uses the KM algorithm to compute an optimal *ES*. We conducted a comprehensive evaluation, and the results demonstrate the effectiveness and practical utility of our approach. Our algorithm is implemented in an open-source, web-based visual differencing tool. We believe BDiff can advance both software engineering practice and related research.

Acknowledgments

We would like to thank the experiment raters for their time. We gratefully acknowledge the support from the National Natural Science Foundation of China (Grant No.62302515).

References

- [1] Tarek M. Ahmed, Weiyi Shang, and Ahmed E. Hassan. 2015. An Empirical Study of the Copy and Paste Behavior during Development. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. Florence, Italy, 99–110. doi:10.1109/MSR.2015.17
- [2] Muhammad Asaduzzaman, Chanchal K. Roy, Kevin A. Schneider, and Massimiliano Di Penta. 2013. LHDiff: A Language-Independent Hybrid Approach for Tracking Source Code Lines. In *Proceedings of 2013 IEEE International Conference on Software Maintenance*. 230–239. doi:10.1109/ICSM.2013.34
- [3] Gerardo Canfora, Luigi Cerulo, and Massimiliano Di Penta. 2009. Tracking Your Changes: A Language-Independent Approach. *IEEE Software* 26, 1 (2009), 50–57. doi:10.1109/MS.2009.26
- [4] Gerardo Canfora, Luigi Cerulo, and Massimiliano Di Penta. 2009. Ldiff: An enhanced line differencing tool. In *2009 IEEE 31st International Conference on Software Engineering*. 595–598.
- [5] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. 1996. Change detection in hierarchically structured information. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*.
- [6] Daniel Alencar Da Costa, Shane McIntosh, Weiyi Shang, Uirá Kulesza, Roberta Coelho, and Ahmed E Hassan. 2016. A framework for evaluating the results of the szz approach for identifying bug-introducing changes. *IEEE Transactions on Software Engineering* 43, 7 (2016), 641–657.
- [7] Quentin Le Dilavrec, Djamel Eddine Khelladi, Arnaud Blouin, and Jean-Marc Jézéquel. 2022. HyperAST: Enabling Efficient Analysis of Software Histories at Scale. In *ASE '22: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*.
- [8] Georg Dotzler and Michael Philippsen. 2016. Move-optimized source code tree differencing. In *IEEE/ACM International Conference on Automated Software Engineering*. 660–671.
- [9] Ekwa Duala-Ekoko and Martin P. Robillard. 2007. Tracking code clones in evolving software. In *29th International Conference on Software Engineering (ICSE'07)*.
- [10] Jean-Rémy Falleri and Matias Martinez. 2024. Fine-Grained, Accurate, and Scalable Source Code Differencing. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. 2856–2867. doi:10.1145/3597503.3639148
- [11] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (Vasteras, Sweden) (ASE '14)*. Association for Computing Machinery, New York, NY, USA, 313–324. doi:10.1145/2642937.2642982
- [12] Yuanrui Fan, Xin Xia, David Lo, Ahmed E. Hassan, Yuan Wang, and Shanping Li. 2021. A Differential Testing Approach for Evaluating Abstract Syntax Tree Mapping Algorithms. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 1174–1185. doi:10.1109/ICSE43902.2021.00108
- [13] Beat Fluri, Michael Wursch, Martin Pinzger, and Harald C. Gall. 2007. Change Distilling—Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING* (2007). Issue 10.
- [14] Beat Fluri, Michael Wursch, Martin Pinzger, and Harald Gall. 2025. A Retrospective of ChangeDistiller: Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Transactions on Software Engineering* (2025), 1–6. doi:10.1109/TSE.2025.3538326
- [15] Veit Frick, Thomas Grassauer, Fabian Beck, and Martin Pinzger. 2018. Generating Accurate and Compact Edit Scripts using Tree Differencing. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*.
- [16] Daniel M. German, Bram Adams, and Kate Stewart. 2019. cregit: Token-level blame information in git version control repositories. *Empirical Softw. Engg.* 24, 4 (aug 2019), 2725–2763. doi:10.1007/s10664-019-09704-x

- [17] Git. 2025. Git - git-diff Documentation. <https://git-scm.com/docs/git-diff>
- [18] M. W. Godfrey and L. Zou. 2005. Using Origin Analysis to Detect Merging and Splitting of Source Code Entities. *IEEE Transactions on Software Engineering* 31, 2 (2005), 166–181.
- [19] Georgios Gousios, Eirini Kalliamvakou, and Diomidis Spinellis. 2008. Measuring developer contribution from software repository data. In *Proceedings of the 2008 international working conference on Mining software repositories*. 129–132.
- [20] GumTreeDiff. 2025. Release v4.0.0-beta4 - GumTreeDiff/gumtree. <https://github.com/GumTreeDiff/gumtree/releases/tag/v4.0.0-beta4>
- [21] Junwoo Ha, Hyunjun Kim, Sangyoon Yu, Haon Park, Ashkan Yousefpour, Yuna Park, and Suhyun Kim. 2025. One-Shot is Enough: Consolidating Multi-Turn Attacks into Efficient Single-Turn Prompts for LLMs. *arXiv preprint arXiv:2503.04856* (2025).
- [22] Masatomo Hashimoto and Akira Mori. 2008. Diff/TS: A Tool for Fine-Grained Structural Change Analysis. In *2008 15th Working Conference on Reverse Engineering*. 279–288. doi:10.1109/WCRE.2008.44
- [23] Hideaki Hata, Osamu Mizuno, and Tohru Kikuno. 2012. Bug prediction based on fine-grained module histories. In *2012 34th international conference on software engineering (ICSE)*. IEEE, 200–210.
- [24] J. W. Hunt and M. D. McIlroy. 1976. *An Algorithm for Differential File Comparison*. Technical Report. Bell Laboratories Computing Science Technical Report.
- [25] Miryung Kim and David Notkin. 2009. Discovering and representing systematic code changes. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, USA, 309–319. doi:10.1109/ICSE.2009.5070531
- [26] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. 2005. An Empirical Study of Code Clone Genealogies. In *European software engineering conference*.
- [27] Harold W. Kuhn. 1955. The Hungarian method for the assignment problem. *Naval Research Logistics (NRL)* 52 (1955). <https://api.semanticscholar.org/CorpusID:9426884>
- [28] J Richard Landis and Gary G. Koch. 1977. The measurement of observer agreement for categorical data. *Biometrics* 33 1 (1977), 159–74. <https://api.semanticscholar.org/CorpusID:11077516>
- [29] Quentin Le Dilavrec, Djamel Eddine Khelladi, Arnaud Blouin, and Jean-Marc Jézéquel. 2023. HyperDiff: Computing Source Code Diffs at Scale. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (San Francisco, CA, USA) (ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 288–299. doi:10.1145/3611643.3616312
- [30] Jiawei Li, David Faragó, Christian Petrov, and Iftekhar Ahmed. 2024. Only diff is not enough: Generating commit messages leveraging reasoning and action of large language model. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 745–766.
- [31] Yao Lu, Xinjun Mao, Zude Li, Yang Zhang, Tao Wang, and Gang Yin. 2018. Internal quality assurance for external contributions in GitHub: An empirical investigation. *JOURNAL OF SOFTWARE-EVOLUTION AND PROCESS* (2018), e1918. Issue 4.
- [32] mdipenta luice. 2025. ldiff download | SourceForge.net. <https://sourceforge.net/projects/ldiff/>
- [33] Matias Martinez and Martin Monperrus. 2019. Coming: A tool for mining change pattern instances from git commits. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 79–82.
- [34] Junnosuke Matsumoto, Yoshiaki Higo, and Shinji Kusumoto. 2019. Beyond GumTree: a hybrid approach to generate edit scripts. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*.
- [35] James Munkres. 1957. Algorithms for the Assignment and Transportation Problems. *Journal of The Society for Industrial and Applied Mathematics* 10 (1957), 196–210. <https://api.semanticscholar.org/CorpusID:268092561>
- [36] Eugene W. Myers. 1986. An O(ND) Difference Algorithm and Its Variations. *Algorithmica* 1, 1-4 (1986), 251–266.
- [37] Yusuf Nugroho, Hideaki Hata, and Kenichi Matsumoto. 2019. How different are different diff algorithms in Git?: Use -histogram for code changes. *Empirical Software Engineering* 25 (09 2019). doi:10.1007/s10664-019-09772-z
- [38] OpenAI. 2025. GPT-5-mini. <https://platform.openai.com/docs/models/gpt-5-mini>
- [39] A. Podgurski, McCleese Masri, W., Wolff Y., and C. Yang. 1999. Estimation of software reliability by stratified sampling. *ACM Transactions on Software Engineering and Methodology* (1999), 263–283. Issue 3.
- [40] Sophia Quach, Maxime Lamothe, Yasutaka Kamei, and Weiyi Shang. 2021. An empirical study on the use of SZZ for identifying inducing changes of non-functional bugs. *Empirical Software Engineering* 26, 4 (2021), 71.
- [41] Qwen. 2025. Qwen/Qwen3-32B. <https://huggingface.co/Qwen/Qwen3-32B>
- [42] P. Reiss Steven. 2008. Tracking source locations. In *ACM/IEEE International Conference on Software Engineering*. 11.
- [43] Chanchal K. Roy. 2009. Detection and analysis of near-miss software clones. In *2009 IEEE International Conference on Software Maintenance*. 447–450. doi:10.1109/ICSM.2009.5306301
- [44] Jacek Sliwinski, Thomas Zimmermann, and Andreas Zeller. 2005. When do changes induce fixes?. In *Proceedings of the 2005 International Workshop on Mining Software Repositories, MSR 2005, Saint Louis, Missouri, USA, May 17, 2005*.

- [45] Jaime Spacco and Chadd Williams. 2009. Lightweight Techniques for Tracking Unique Program Statements. In *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*.
- [46] Jeffrey Svajlenko and Chanchal K. Roy. 2021. The Mutation and Injection Framework: Evaluating Clone Detection Tools with Mutation Analysis. *IEEE Transactions on Software Engineering* (2021), 1060–1087. Issue 5.
- [47] Anthony J Viera and Joanne M Garrett. 2005. Understanding interobserver agreement: the kappa statistic. *Family Medicine* 37, 5 (2005), 360–3.
- [48] Tanghaoran Zhang, Yao Lu, Yue Yu, Xinjun Mao, Yang Zhang, and Yuxin Zhao. 2024. How do Developers Adapt Code Snippets to Their Contexts? An Empirical Study of Context-Based Code Snippet Adaptations. *IEEE Transactions on Software Engineering* (2024), 1–20. Issue 11.
- [49] Kaifeng Huang;Bihuan Chen;Xin Peng;Daihong Zhou;Ying Wang;Yang Liu;Wenyun Zhao. 2018. ClDiff: Generating Concise Linked Code Differences. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*.

A Typical BDiff Results with Block-level EAs

A.1 Changing the Orders of Initialization Assignment of Class Attributes



Fig. 11. GitHub Project: kitao/pyxel. Commit: 3861523. File: app.py

A.2 Indenting a Block for Conditional Branching

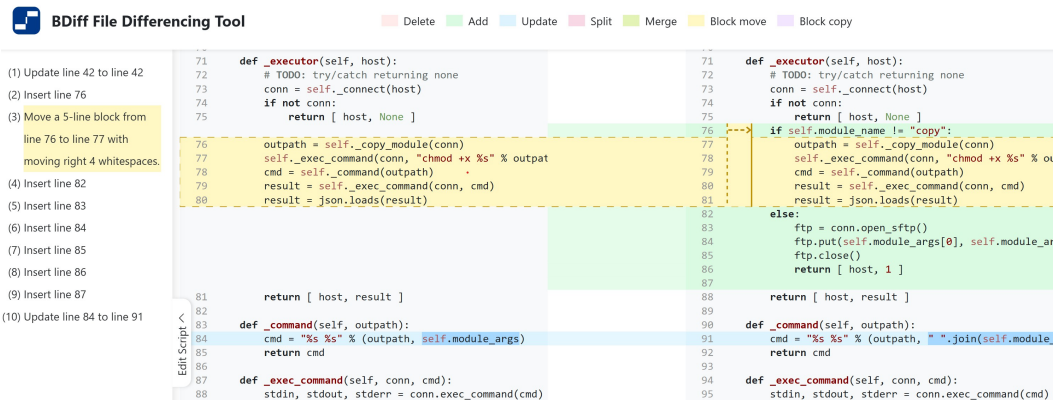


Fig. 12. GitHub Project: ansible/ansible. Commit: 3807824. File: __init__.py

A.5 Duplicating an XML Block into another Block with Indentation

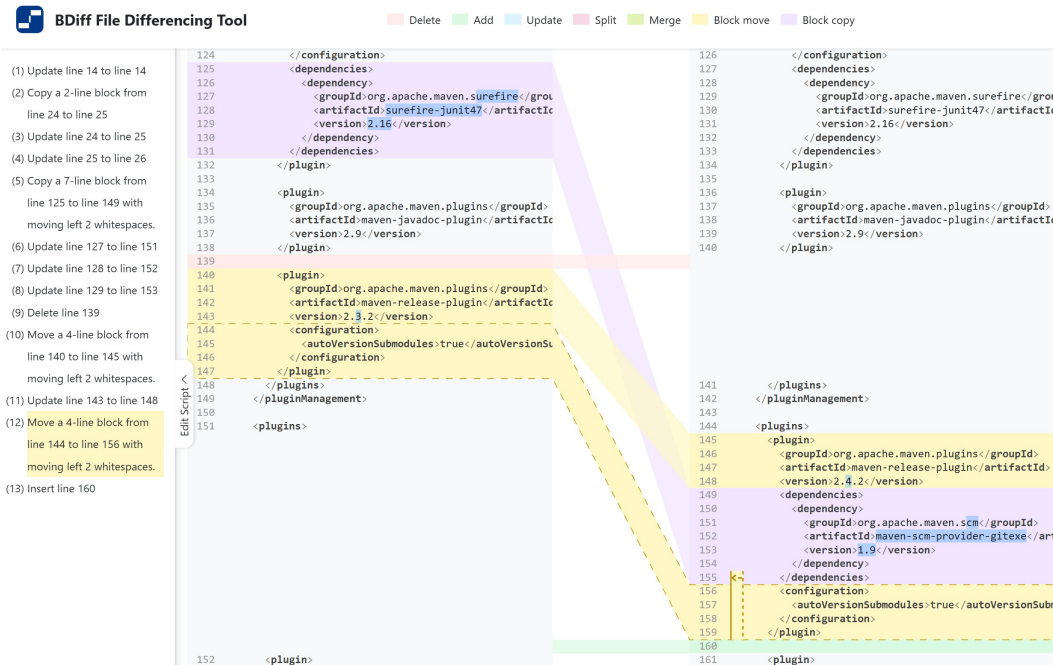


Fig. 15. GitHub Project: square/okhttp. Commit: c863881. File: pom.xml

A.6 Splitting Lines and Indenting Blocks

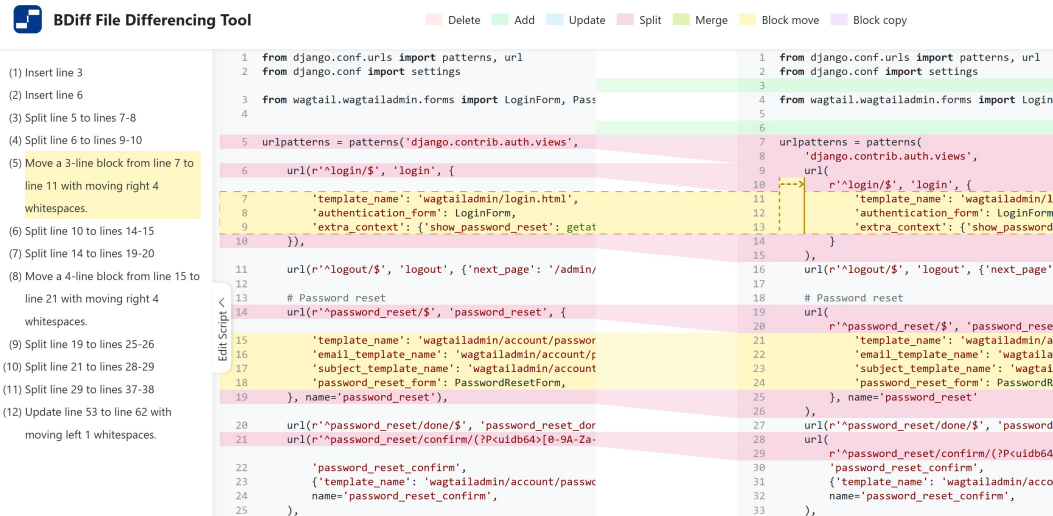


Fig. 16. GitHub Project: wagtail/wagtail. Commit: a2a580f. File: urls.py

B Typical Diff Cases of GumTree and BDiff
B.1 GumTree Identifies an AST Node Change that Contains Multiple Lines as One EA

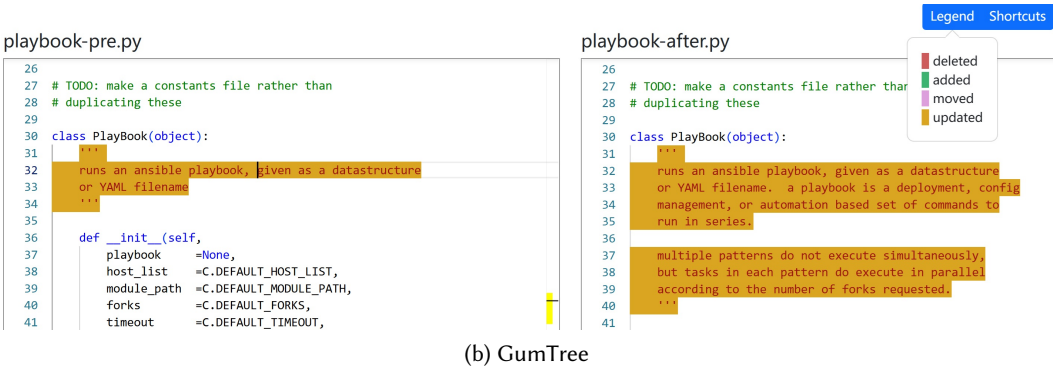
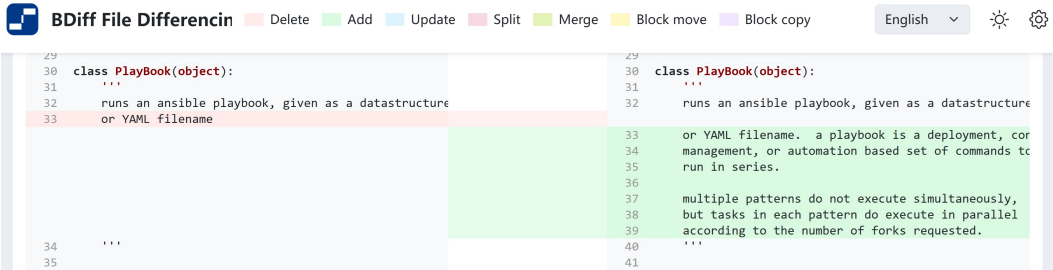
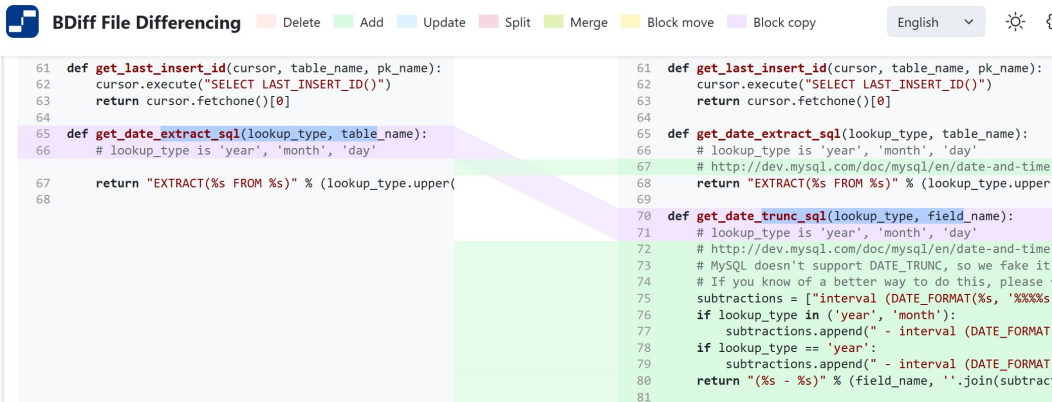
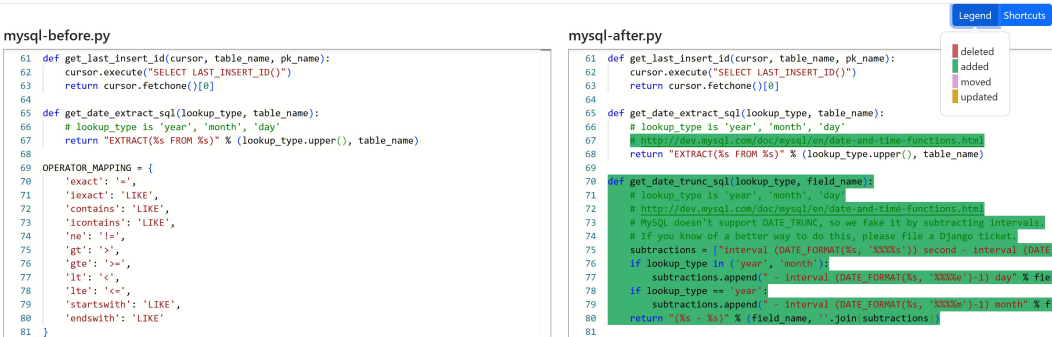


Fig. 17. GitHub Project: ansible/ansible. Commit: 43f7dee. File: playbook.py



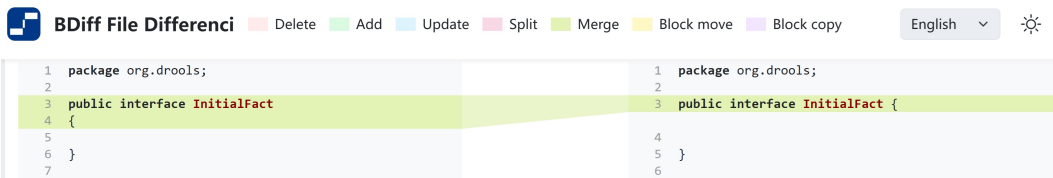
(a) BDiff



(b) GumTree

Fig. 18. GitHub Project: django/django. Commit: b1c543d. File: mysql.py

B.2 GumTree is Insensitive to Formatting Changes



(a) BDiff



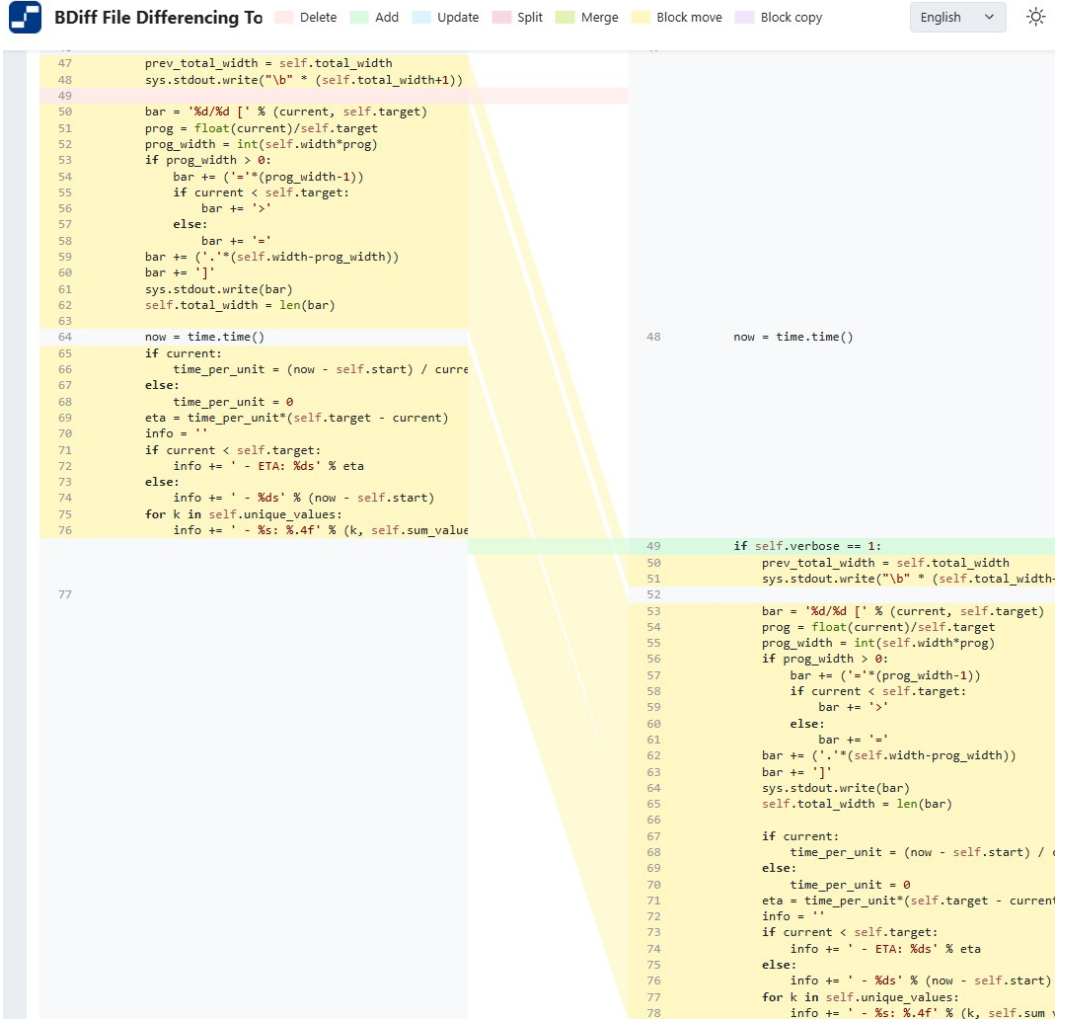
(b) GumTree

Fig. 19. GitHub Project: apache/incubator-kie-drools. Commit: 5bb719f. File: InitialFact.java



Fig. 20. GitHub Project: wagtail/wagtail. Commit: a2a580f. File: urls.py

B.3 GumTree Identifies Consecutive Lines Moving as Individual Line Movements



(a) BDiff

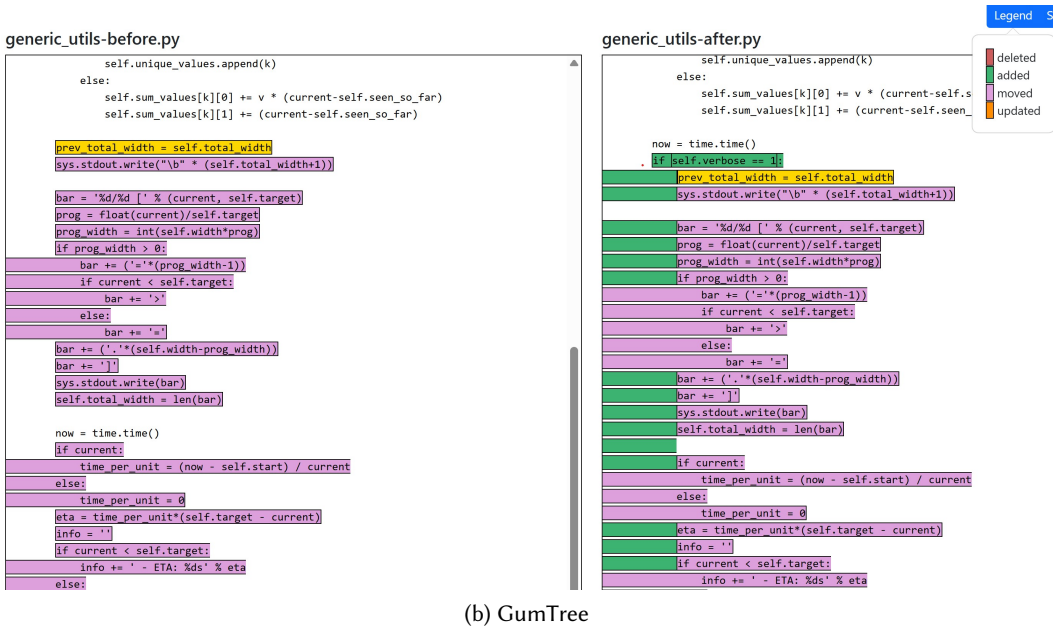


Fig. 20. GitHub Project: keras-team/keras. Commit: c224482. File: generic_utils.py

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009