# Fine-grained, Accurate, and Scalable Source Code Differencing

Jean-Rémy Falleri
falleri@labri.fr
Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR 5800
Talence, France
Institut Universitaire de France
Paris, France

Matias Martinez
matias.martinez@upc.edu
Universitat Politècnica de Catalunya
Barcelona, Spain

## ABSTRACT

Understanding code changes is of crucial importance in a wide range of software evolution activities. The traditional approach is to use textual differencing, as done with success since the 1970s with the ubiquitous `diff` tool. However, textual differencing has the important limitation of not aligning the changes to the syntax of the source code. To overcome these issues, structural (i.e. syntactic) differencing has been proposed in the literature, notably GumTree which was one of the pioneering approaches. The main drawback of GumTree's algorithm is the use of an optimal, but expensive tree-edit distance algorithm that makes it difficult to diff large ASTs. In this article, we describe a less expensive heuristic that enables GumTree to scale to large ASTs while yielding results of better quality than the original GumTree. We validate this new heuristic against 4 datasets of changes in two different languages, where we generate edit-scripts with a median size 50% smaller and a total speedup of the matching time between 50x and 281x.

## CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**; *Software configuration management and version control systems.*

## KEYWORDS

Software evolution, Code differencing

**ACM Reference Format:**
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 '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597503.3639148

## 1 INTRODUCTION

Understanding the changes between two source code versions is a daily concern for developers, for instance, when performing a code review, debugging a regression, or when they simply want to catch up with an updated code base. Historically, analyzing the changes between two source code versions has been done using *textual evolution inference*. Simply put, this technique represents the source code as a sequence of text lines and aims at finding the shortest sequence of insertions or deletions of text lines that would transform the *original* source code into the *modified* source code. This technique has the enormous advantage of having very efficient algorithms [33] that can find an optimal solution to the problem of finding a minimal sequence of such actions allowing this technique to scale to huge code bases.

Textual evolution inference, however, also bears some limitations. Firstly, a source code is not an unstructured text but obeys syntactic rules. Textual evolution inference does not consider this syntax and occasionally outputs changes that do not make sense, such as inserted code lines spanning two different functions. Another pain point is that beyond inserting or deleting code, developers also use other edit actions, such as changing the value of tokens (renaming a variable) or moving code around (extracting a new function). Since these actions are not taken into account by textual evolution inference, they are represented by insertions or deletions, making the sequence of actions harder to understand.

To solve the previously described limitations, *syntactic evolution inference* has been proposed. It uses a richer model of source code: rooted, ordered and labeled trees that can adequately represent abstract syntactic trees (ASTs). It also usually considers a richer set of edit actions, such as renaming node labels or moving subtrees. Using these richer models of code and actions, syntactic evolution inference may produce results easier to understand, especially in the presence of a small amount of changes [9, 11]. Another advantage of syntactic evolution inference is that it produces actions that operate on syntactic elements of the code rather than on text lines, which are easier to process automatically to perform empirical studies on software evolution.

Syntactic evolution inference also comes with a great limitation: with such rich models of code and actions, the problem of finding a minimal sequence of actions is NP-hard [3]. Therefore, syntactic evolution inference is performed using heuristics that must reach a tradeoff between the runtime complexity and the quality of the results.

GumTree [11] is one of the most popular heuristics to perform syntactic evolution inference. It relies on a three-phased process to find mappings between the nodes of two ASTs. One of the main limitations of GumTree lies in the last phase, called *recovery*, where an optimal tree-edit algorithm is used as a "last chance" to uncover relevant mappings of AST nodes. This algorithm has a $O(n^3)$ time complexity, limiting its ability to scale on large ASTs. For this reason, GumTree's algorithm introduces a hyperparameter, called *maximum size threshold*, that limits the size of the subtrees where the recovery phase is applied. Setting a low value speeds up GumTree at the expense of losing many relevant mappings and setting a high value will drastically increase the running time. In this article, we describe a new recovery phase where the optimal

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04
https://doi.org/10.1145/3597503.3639148

tree-edit algorithm is removed and thus the maximum size threshold. We proceed to an extensive validation of our new heuristic on four different datasets in two popular programming languages: two bug-fixes datasets, BugsInPy [39] and Defects4J [22], as GumTree is widely used in the software repair domain (e.g., [2, 4, 17, 18, 21, 23–25, 38, 41]) and against two datasets of 1000 commits coming from 10 popular projects in Java and Python, to gauge its effectiveness on arbitrary changes. We generate edit-scripts with a median size 50% smaller on all datasets with a drastic total speedup of the matching time between 50x and 281x.

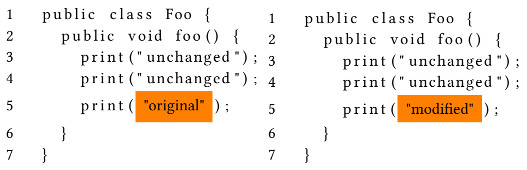
## 2 MOTIVATION

GumTree is a heuristic that maps nodes from two ASTs, and from these mappings, outputs an edit-script i.e., a sequence of actions that transform the first AST to the second.

In this section, we discuss how GumTree works in order to understand how its limitations affect the consumers of GumTree edit-scripts. We identify two main consumers of these edit-scripts: a) *developers* via a visualization tool for changes comprehension (e.g., [16]), or b) *analysts*: engineers or researchers that carry out studies in software evolution and maintenance (e.g., [38]) or in machine learning on source code (e.g. [27]), which require analyzing changes from a large number of commits, using an algorithm that consumes GumTree edit-scripts (e.g., [24, 30]).

### 2.1 How GumTree Works

To illustrate how GumTree works, we use the example shown in Fig. 1 where there is a single modification in a literal string (line 5).



```
1  public class Foo {
2    public void foo() {
3      print("unchanged");
4      print("unchanged");
5      print("original");
6    }
7  }
```

(a) Original file

```
1  public class Foo {
2    public void foo() {
3      print("unchanged");
4      print("unchanged");
5      print("modified");
6    }
7  }
```

(b) Modified file

**Figure 1: A sample modification with its ideal visual syntactic diff.**

GumTree maps nodes from the two ASTs corresponding to the original and modified files in a three-phased process, as follows:

(1) **Top-down phase:** greedily searches for the biggest isomorphic subtrees among the two ASTs to establish mappings. Note that subtrees smaller than a given size (controlled by a `min_size` threshold) are not considered.
(2) **Bottom-up phase:** uses the previously established mappings to propagate mappings to their parent nodes.
(3) **Recovery phase:** is performed each time a mapping is added during the bottom-up phase, to ensure that no additional mappings can be found among the unmapped descendants of the mapped nodes.

In the first top-down matching phase, GumTree searches for the biggest isomorphic subtrees, resulting in the mappings depicted with a green background in fig. 2. These mappings correspond to the two first invocations of the `print()` method in fig. 1. At the end of this phase, there are no other mappings. GumTree leverages these initial mappings during the second bottom-up matching phase where the initial mappings are propagated to the parent nodes. For instance, in this phase, GumTree will match first the bottom-right `Body` node (depicted with a blue background) since there are two already mapped `MethodCall` subtrees in the descendants. This is at this precise moment when the recovery phase comes into action. Note that just after having established the mapping for the `Body` node, this node has still three unmapped descendants, corresponding to the modified `print()` invocation of fig. 1. These descendants have not been mapped during the top-down matching phase since their size is smaller or equal to the `min_size` threshold (by default set to 1). Note that it would be very hazardous to decrease the `min_size` threshold to zero to match such nodes since at this point, it is very likely to make a mistake (for instance there are usually a lot of possible mappings for an unmatched `public` visibility). It is exactly for this reason that this threshold exists in the first place. However, these unmapped descendants should be mapped together to have an accurate diff, since the developer, in this case, expects a single update action from the string literal *original* to *modified*, as shown in fig. 1. This is the objective of the recovery phase that is expected to map the three descendants with a yellow background fig. 2. This process will go on among the remaining `MethDef`, `Body`, and `ClassDef` containers (depicted with a blue background), each time yielding a recovery phase finding additional mappings depicted with a yellow background. Finally, the mappings shown in fig. 2 yield the edit-script shown in fig. 3 from which the visualization of fig. 1 (highlighting the update action in orange) is derived.

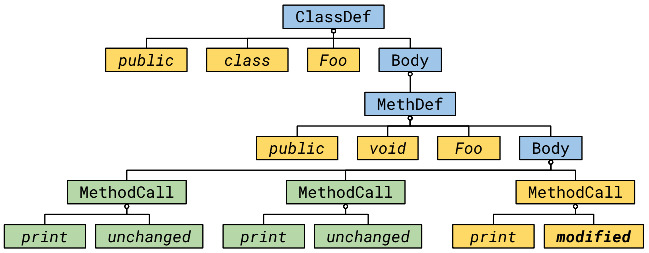


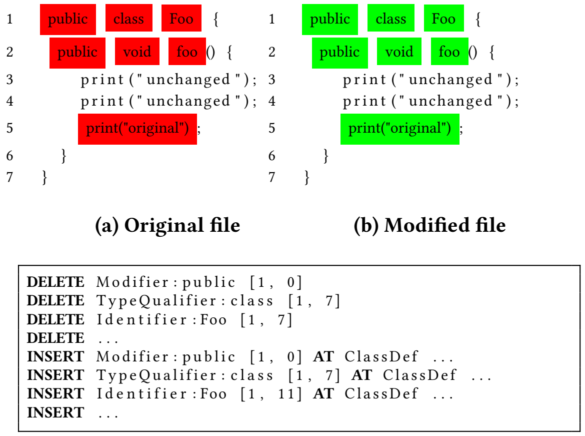
**Figure 2: AST generated from the modified file of fig. 1. The leaf node with labels are in italics and their types are omitted for the sake of readability. The only modified node is in bold. The nodes found in the top-down matching phase have a green background. The ones found in the bottom-up matching phase have a blue background. Finally, the nodes found during the recovery phases have a yellow background.**

```
UPDATE "original" [5, 8] BY "modified"
```

**Figure 3: Edit-script with a recovery phase.**

### 2.2 Achilles Heel of GumTree: Recovery Phase

With this example in mind, we note the crucial importance of the recovery phase which accounts for 9 out of the total 19 mappings.



```
public class Foo {
  public void foo() {
      print("unchanged");
      print("unchanged");
      print("original");
  }
}
```

(a) Original file

```
public class Foo {
  public void foo() {
      print("unchanged");
      print("unchanged");
      print("original");
  }
}
```

(b) Modified file

```
DELETE Modifier:public [1, 0]
DELETE TypeQualifier:class [1, 7]
DELETE Identifier:Foo [1, 7]
DELETE ...
INSERT Modifier:public [1, 0] AT ClassDef ...
INSERT TypeQualifier:class [1, 7] AT ClassDef ...
INSERT Identifier:Foo [1, 11] AT ClassDef ...
INSERT ...
```

(c) Extract of the edit-script without a recovery phase

**Figure 4: Visual syntactic diff without a recovery phase. Code with delete (resp. insert) actions is highlighted in red (resp. green).**

Without the mappings established in the recovery phases, the resulting syntactic diff has, as fig. 4 shows, insert and delete actions for all nodes depicted with a yellow background in fig. 2 instead of update actions. Unfortunately, the recovery phase of GumTree makes use of an expensive optimal algorithm to compute the mappings between the two subtrees rooted at the container nodes [34]. This algorithm has a $O(n^3)$ complexity that induces long running times for big subtrees. For this reason, the authors of GumTree introduced a `max_size` threshold that controls the application of this algorithm: it is not applied as soon as the biggest of the two subtrees compared in the recovery phase has more nodes than `max_size`. Even if the default value for this threshold, 1000, could seem high enough, it is often not enough in practice. In our example, the container nodes close to the root (such as `ClassDef`) often have a large number of descendants. This phenomenon is even worse with languages where code files contain a lot of source code.

Therefore the users of GumTree are left with a tough dilemma: either setting a high value of `max_size` and suffering high running time in exchange for good accuracy or setting a low value and obtaining edit-scripts with a lot fewer update actions.

*Impact of Recovery on Developers.* When developers compare two large pieces of code, the visualization of the edit-scripts could show insert and delete actions for elements that are not modified (we call them *spurious* and discuss them in section 6.2), similar to those presented in fig. 4. Both situations occur due to the abortion of the recovery phase and affect the usability of GumTree.

*Impact of Recovery on Analysts.* The dilemma of `max_size` at the recovery phase also affects *analysts*, which analyze large amounts of file-pairs. In addition to introducing irrelevant actions in the output, it also inflates the running time. To illustrate the impact it has on them, we executed GumTree on Defects4J (D4J), a dataset with 832 bug fixes. We chose D4J because previous work (e.g., [26, 28, 41]) has already applied GumTree on it. For each bug fix, we run GumTree to compare the buggy version of a file with the corresponding fixed version. We tried four `max_size` thresholds: *a)* 100, *b)* 500, *c)* 1000 (default value on GumTree), *d)* 1500. In table 1, we observe that decreasing `max_size` from 1000 (default value) to 100 or 500 impacts both the output (GumTree produces larger edit-scripts) and the execution time (GumTree runs faster, as it executes fewer times the *expensive* recovery phase, on smaller subtrees). In contrast, increasing `max_size` from 1000 to 1500 leads to an increase in execution time (the recovery phase is invoked more times with bigger subtrees) and produces shorter edit-scripts (the new executions of the recovery phase manage to find additional mappings, which avoid the generation of spurious actions caused by the abortion of the recovery phase). Even using a big value such as 1500, there are still 1032 subtrees where recovery was aborted while the running time is doubled compared to the default value of 1000, and the impact on the reduction of the edit-script size is minimal (from 26 to 25).

**Table 1: Execution of GumTree on Defects4J [22] using different `max_size` thresholds. `recovery applied` (resp. `aborted`) is the number of times the recovery phase is executed (resp. aborted). `median size` is the median size of edit-scripts while `time` os the total time required to process the dataset.**

| | recovery | | | |
|---|---|---|---|---|
| `max_size` | applied | aborted | median size | time (ms) |
| 100 | 3062 | 3430 | 33 | 6897 |
| 500 | 4758 | 1728 | 28 | 58803 |
| 1000 (default) | 5228 | 1258 | 26 | 187699 |
| 1500 | 5453 | 1032 | 25 | 391652 |

In this article, we present a brand new recovery phase that is much faster than the original recovery phase enabling us to remove the `max_size` threshold. Consumers get the best of both worlds: drastically reduced running times and good accuracy of the edit-scripts.

## 3 BACKGROUND

In this section, we first introduce the definitions of the tree structure we use as well as the edit actions we consider. Secondly, we describe the original GumTree heuristic that operates in four phases.

### 3.1 Definitions

Similarly to how it is done in GumTree [11], we define an abstract syntactic tree (AST) $t$ as a rooted, ordered, and labeled tree. Each node $n \in \mathcal{N} \cup \bot_t$ of an AST have:

- A parent: $\text{parent}(n) \in \mathcal{N} \cup \{\bot_n\} \setminus \{n\}$. The root is the only node that has parent $\bot_n$ (which in reality represents the fact that the root has no parent)
- A possibly empty ordered sequence of children: $\text{children}(n)$ where each child $c \in \mathcal{N} \setminus \{n\}$.
- A type: $\text{type}(n) \in \mathcal{T}$.
- A label: $\text{label}(n) \in \mathcal{L} \cup \{\bot_l\}$. Nodes with no label have $\text{label}(n) = \bot_l$. Only leaves can have a label $l \neq \bot_l$.

Diffing two ASTs $T_1$ and $T_2$ aims at finding a sequence $E$ of actions that when applied to $T_1$ yields $T_2$. This sequence is called the edit-script, and the possible actions are:

- insert_node$(n, p, i)$: insert a new node $n$ as a child of node $p$ at position $i$.
- insert_tree$(s, p, i)$: insert a new subtree $s$ as a child of node $p$ at position $i$.
- delete_node$(n)$: delete leaf node $n$.
- delete_tree$(s)$ delete subtree $s$.
- move_tree$(s, p, i)$: move subtree $s$ into node $p$ at position $i$.
- update_node$(n, l)$: change the label of node $n$ to $l$.

In addition to the actions defined in GumTree, we introduce two new actions insert_tree and delete_tree which conveniently decrease the size of the edit-script when whole subtrees of code get inserted and deleted at once, which is frequent in code evolution (for instance when a complete statement is inserted or deleted). These actions facilitate the understanding of the edit-script.

## 3.2 Overview of GumTree

GumTree maps nodes from two AST $T_1$ and $T_2$, in three phases. First, the *top-down* phase applies a greedy search of isomorphic subtrees between $T_1$ and $T_2$ and adds them to a set of mappings $\mathcal{M}$. Secondly, the *bottom-up* phase, takes $\mathcal{M}$ and propagates the mappings to their parent nodes, as soon as two parent nodes have a significant number of matched descendants. Third, the *recovery* phase takes each new mapping from the previous phase and looks for recovery mappings, that are searched among the still unmatched descendants of the mapping's nodes. Since we present a brand new recovery phase in this article, we now discuss the original recovery phase from GumTree. More details about the other two phases (which we use without applying any change) can be found in [11].

## 3.3 Recovery Phase

To search for additional mappings between the descendants of a pair of matched nodes during the bottom-up phase, GumTree applies a last recovery phase. In the original version of GumTree, an optimal (without move actions) tree-edit distance algorithm [34] is applied and every mapping it finds is added to $\mathcal{M}$ as long as they involve previously unmatched nodes. Since this algorithm is expensive ($O(n^3)$ time complexity), as it was shown in the motivation example from section 2, it is only applied to pairs of nodes that have a bounded number of descendants, configured by a threshold called `max_size`. This algorithm is shown in algorithm 1.

## 3.4 Generation of the Edit-script

After the computation of mappings, the edit-script is generated using the algorithm of [5]. The original version of GumTree produces four actions: insert_node, delete_node, update_node (all these three applied to a single node) and move_tree, applied to a complete subtree. In addition, to detect the insert_tree and delete_tree actions that are not considered in GumTree, we post-process the edit-script, looking for a sequence of insert_node (resp. delete_node) actions that target a complete subtree, as described in section 3. Whenever we find such a sequence of actions, we replace it with an insert_tree (resp. a delete_tree) action at the same position in the script. Note that these actions were not part of the original GumTree output.

# 4 APPROACH

Our experience with GumTree led us to reconsider the optimal recovery phase defined in the previous section as we explained in section 2. In this section, we describe our new recovery heuristic, called `simple`, that has been designed to be much faster than the optimal recovery while achieving good results. It allowed us to completely remove the `max_size` threshold, freeing the users of a difficult choice to make. Based on this new recovery heuristic, we also define a `hybrid` recovery that combines `optimal` and `simple`.

## 4.1 Simple Recovery

Our new recovery phase is called *simple* and is shown in algorithm 2. It operates in three main steps. In the first step (line 1), it searches first for isomorphic subtrees inside the unmapped children of both nodes using a longest common subsequence algorithm. Whenever such isomorphic subtrees belong to the longest common subsequence and contain only unmapped descendants, they are added with all their descendants inside the recovery mappings. This part is very similar to what is done during the top-down phase (and also uses the hash function of [6] to check the isomorphism), but can still be useful for cases where the subtrees were not considered during the top-down matching phase due to a size under the `min_size` threshold. The second step (line 2) is similar to the first one but searches for isomorphic subtrees when labels are removed from the leaf nodes (structure isomorphism). We also use the hash function of [6] but adapted to discard the label of the nodes to compute the hash. Therefore subtrees where the only change is an identifier will

**Algorithm 1:** The optimal recovery algorithm.

```
Data: Two nodes $t_1$ and $t_2$, a set $\mathcal{M}$ of mappings (under
      construction inside the bottom-up phase), a threshold
      max_size
if $\max(|s(t_1)|, |s(t_2)|) <$ max_size then
    $C \leftarrow \text{opt}(t_1, t_2)$;
    foreach $(t_a, t_b) \in C$ do
        if $t_a, t_b$ not already mapped $\wedge$ type$(t_a)$ = type$(t_b)$
         then
            $\mathcal{M} \leftarrow \mathcal{M} \cup (t_a, t_b)$;
```

**Algorithm 2:** The simple recovery algorithm.

```
Data: Two nodes $t_1$ and $t_2$, a set $\mathcal{M}$ of mappings (under
      construction inside the bottom-up phase)
1 add to $\mathcal{M}$ all pair of descendants of pairs of nodes from
    $\text{lcs}_e(\text{children}(t_1) \mid (t_1, t_x) \notin \mathcal{M}, \text{children}(t_2) \mid (t_y, t_2) \notin \mathcal{M})$;
2 add to $\mathcal{M}$ all pair of descendants of pairs of nodes from
    $\text{lcs}_s(\text{children}(t_1) \mid (t_1, t_x) \notin \mathcal{M}, \text{children}(t_2) \mid (t_y, t_2) \notin \mathcal{M})$;
3 foreach $(t_a, t_b) \in \text{uniqueType}(\text{children}(t_1) \mid (t_1, t_x) \notin \mathcal{M}, \text{children}(t_2) \mid (t_y, t_2) \notin \mathcal{M})$ do
4     $\mathcal{M} \leftarrow \mathcal{M} \cup (t_a, t_b)$;
5     simple_recovery$(t_a, t_b, \mathcal{M})$;
```

**Algorithm 3:** The hybrid recovery algorithm.

```
Data: Two nodes $t_1$ and $t_2$, a set $\mathcal{M}$ of mappings (under
      construction inside the bottom-up phase), a threshold
      max_size
1 if $\max(|s(t_1)|, |s(t_2)|) <$ max_size then
2     optimal_recovery($t_1, t_2, \mathcal{M}$);
3 else
4     simple_recovery($t_1, t_2, \mathcal{M}$);
```

be isomorphic. This step is thus particularly useful to detect update actions. For the last step, we seek pairs of nodes whose types appear only once in both children's lists (line 3). This step is inspired by the XYDiff algorithm [7]. The intuition behind this step is that a given node has only one child of a given type (for instance one visibility), the odds are very high that it should be mapped to its counterpart. In contrast with the two previous lines, here we only map the two nodes and not their descendants (line 4), since we do not have the guarantee that the subtrees are isomorphic. To look for additional mappings among their descendants we recursively apply this simple recovery algorithm to each such mapped node (line 5).

As an example, let us consider how the simple recovery would work on the sample tree of fig. 2. The first recovery phase would be launched on the bottom-right `Body` node. The first step searching for isomorphic subtrees would not find anything different since the two unchanged `MethodCall` have already been found during the top-down phase. The second step searching for isomorphic subtrees discarding labels would find three mappings for the three nodes contained in the modified `MethodCall` subtree since when discarding the labels, these subtrees are isomorphic. The second recovery phase would be launched on the `MethDef` node. The first step that searches for isomorphic subtrees would find mappings for the `public`, `Foo` and `Void` nodes as they are isomorphic and in the same order as their counterparts in the modified AST. Note that this is typically a case where the mappings were not found in the top-down phase because of a size below the `min_size` threshold. The other step would find nothing more as all descendants of the `MethDef` nodes are mapped at this point. The recovery phase would not be launched for the top-right `Body` node since there are no unmapped descendants for this node. Finally the last recovery phase, launched for the `ClassDef` node would find mappings for the `public` and `Foo` nodes, similarly to how it behaved for the `MethDef` node. All the nodes mapped during the recovery phases are depicted in yellow in fig. 2.

## 4.2 Hybrid Recovery

In the previous sections, we presented two recovery algorithms: 1) an expensive one that is supposed to recover a lot of new mappings but cannot be applied on too big subtrees and 2) a fast one that is expected to recover fewer mappings. As a last strategy, we introduce a hybrid strategy that aims at combining the strengths of the aforementioned ones by applying the expensive recovery on small subtrees and the fast recovery on big subtrees, as shown in algorithm 3. In this strategy, we still use the `max_size` threshold to decide if we launch the optimal or the simple recovery (line 1).

# 5 EVALUATION

Our evaluation aims at answering the two following research questions: *RQ1: how well do our heuristics scale on edit-scripts?* and *RQ2: what is the quality of the output of our heuristics?* First, we describe the datasets we used in our experiments, and then we describe our experimental protocol. All the code and data used in the experiments are available in our replication package [12].

## 5.1 Datasets

To validate our heuristics we use four datasets. The first two are Defects4J [22] (D4J) and BugsInPy [39] (BIP). These two datasets contain collections of file pairs, where each pair corresponds to a bug fix (within the pair, we say the left part contains the buggy version of a file, the right one the fixed version of that file). These datasets have been curated so that they contain only changes related to the bug fix, while irrelevant changes have been discarded. We chose these two datasets for several reasons. First, they have been independently created by other research teams, therefore reducing bias. Second, they use two languages that are well supported by GumTree and sufficiently different from each other (one has static typing, the other dynamic typing) to make it interesting to investigate both to strengthen the external validity. Third, these two datasets contain bug-fixes, and automated software repair [32] is one of the domains where syntactic differencing has been used the most (for instance in [2, 24, 25] among many others), therefore the ability to perform well on these benchmarks is important.

To also evaluate our heuristics on arbitrary diffs, we create two additional datasets namely GhPython (GHP) and GhJava (GHJ). To select the projects, we used the GitHub explore feature by browsing projects with the Java and Python topics ordered by stars. We then browsed the list to select 10 projects for each language by taking care that the projects corresponded to software projects (as opposed to courses, books, or student assignments). We also took care to select projects developed by different communities (as opposed to selecting only Apache projects). Using PyDriller [36], we selected 100 file pairs for each project according to the procedure shown in algorithm 4. We went through the non-merge commits of the project from the oldest to the newest, and for each commit, we gathered the edited files written in the targeted language (Java or Python) until we reached 100 file pairs for the project. By traversing the commits from older to newer, we postulate that we bias our datasets towards changes usually applied in an initial development period, where feature additions and refactorings are arguably more frequent than single bug-fixes. We therefore hypothesize that these two datasets will contain more complex changes than the bug-fixes datasets, making them complementary.

We eliminated in the four datasets the files that could not be parsed with the JDT parser for Java and tree-sitter parser for Python (2 for BIP, 3 for D4J, 1 for GHJ and 9 for GHP). Finally, the number of remaining file pairs is 643 for BIP, 1046 for D4J, 991 (from 530 commits) for GHP, and 999 (from 186 commits) for GHJ.

To investigate our hypothesis about the increased size of changes between the bug-fixes datasets and the arbitrary changes datasets, we compute the textual diff size of all file pairs using the `diff -u` and `diffstat` commands. We define the diff size as the sum of added, deleted, and modified lines of code as detected by `diff`. The

**Algorithm 4:** The procedure to select the file pairs for each project from GHJ and GHP.

```
Data: A project $P$
Data: A targeted language $L$
Result: The set of changed files $\mathcal{F}$ initially empty
foreach $c \in$ commits($P$) (browse non-merge commits on the main branch in chronological order) do
    foreach $(f_1, f_2) \in$ changedFiles(c) do
        if extension($f_1$) == $L \wedge$ extension($f_2$) == $L$ then
            $\mathcal{F} \leftarrow \mathcal{F} \cup (f_1, f_2)$;
            if size($\mathcal{F}$) >= 100 then
                return $\mathcal{F}$;
```

distribution of the diff sizes is shown in fig. 5. The distribution of the diff sizes is fairly similar in both bug-fixes datasets (first two violin plots), with a median diff size of around 5 lines and a third quartile of around 10 lines. It indicates that bug fixes have a similar size across Python and Java projects. However, we can note the presence of a few very large diffs in the BIP dataset, with more than one thousand changed lines. A manual inspection of the underlying data indicates that these outliers come from the pandas project where some files are changed drastically for several bug instances. The distribution of the diff sizes of the two arbitrary changes datasets (last two violin plots) is once again similar, the Python diffs being slightly smaller The median diff size is around 10 for both datasets and the third quartile is between 20 and 30. Once again we can notice the presence of outliers in the dataset in Python (GHP). Finally, we can notice that arbitrary diffs (the two violin plots on the right) have a doubled median length compared to bug fixes (the two violin plots on the left), supporting our hypothesis that they contain more complex changes than the bug-fixes datasets.

## 5.2 Experimental Protocol

To answer our questions, we will resort to a quantitative and a qualitative experiment. W.r.t. to the quantitative experiment we will apply four heuristics to our four datasets. As baselines, we will apply the `opt-1000` (optimal recovery with a `max_size` threshold of 1000) heuristic which is current GumTree's default. We choose GumTree as a baseline for three reasons. First, we designed our approach to follow the footsteps of GumTree by not requiring language-specific information in the diff algorithm, ruling out language-specific baselines such as [9, 15]. Second, a study [13] indicates that GumTree is a relevant baseline, achieving the smallest ratio of revisions with inaccurate mappings among the studied approaches. Finally, our heuristics are tangled with GumTree since they reuse two of the three phases, making it the most obvious baseline. We will also apply `opt-100` (optimal recovery with a `max_size` threshold of 100) to have a baseline efficiency since it will apply the optimal recovery only on small subtrees. We will finally apply our two candidate heuristics: `simple` and `hybrid-100` (hybrid recovery with a `max_size` threshold of 100). For each file pair and heuristic, we will perform five runs and gather the time spent in the matching step (the parsing and script generation steps being the same for all heuristics). We also collect the produced edit-scripts and use them to compute the following metrics for each file pair of each dataset:

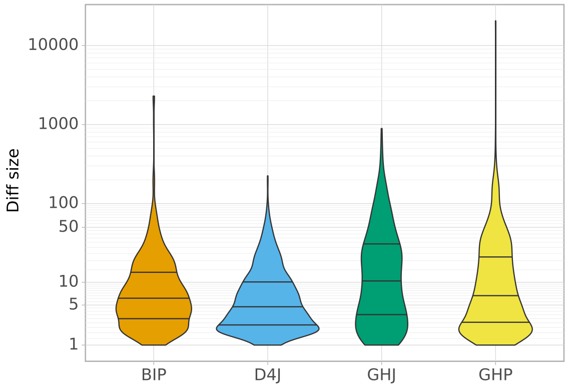


**Figure 5: Distributions of the datasets' text diff sizes.**

- **median-time**: the median runtime among five measures on an Intel Xeon W-2125 4GHz CPU
- **size**: the total number of actions,
- **nunm**: the number of move and update actions.

We will assess the runtime performance using the **median-time** metric, while the **size** and **nunm** metrics will be used as a proxy to evaluate the quality of the edit-script. Indeed, the size of an edit-script is a widely used proxy to measure its quality, based on the hypothesis that a short edit-script is easier to understand than a long one [7, 11]. We also measure the number of move and edit actions since these actions are particular to syntactic differencing and since they can improve the understandability of an edit-script.

To complement this quantitative experiment, we also proceed to a manual analysis. Since this analysis requires experts and is very time-consuming, we will limit ourselves to manually analyzing 100 cases, using the best-performing heuristic in the first experiment and comparing it to `opt-1000`, GumTree's default. We adopt a pessimistic stance where we consider that this heuristic can only perform better than the baseline when its edit-script is shorter. Our goal is to investigate if this hypothesis is true. To that extent, we draw at random 25 file pairs from each dataset where the edit-script is smaller for the candidate heuristic than for `opt-1000`. Then the two authors, that are experts in code differencing, will look at the two edit-scripts produced by the two heuristics, using a graphical user interface showing a side-by-side diff, similarly to what is shown in fig. 10, and will rate the edit-script produced by the candidate heuristic into one of these three categories:

- **better**: all relevant actions contained in the edit-script of `opt-1000` are in the one of `simple` and not conversely
- **worse**: all relevant actions contained in the edit-script of `simple` are in the one of `opt-1000` and not conversely
- **mixed**: any other case

The relevance of actions is of course subjective especially for the move and update actions because it is sometimes hard to judge if the mapped nodes are conceptually equivalent in the old and new versions. Also, insert and delete actions can sometimes be irrelevant when they target nodes that are in fact present in both versions. To counteract this subjectivity, the two authors will rate independently the file pairs and finalize the rating during a discussion session where each divergent rating is discussed and resolved to a common rating or discarded from the experiment.

The two authors involved in this qualitative experiment are aware of which heuristic generated the edit-scripts, possibly inducing a confirmation bias. Additionally, they could interpret edit-scripts differently than arbitrary developers. To complement the opinion of the authors, we seek the opinion of external participants on the results. Seven participants were recruited in the laboratories of the authors. They have a computer science background, with a Bachelor's (1), a Master's (5), or a PhD's degree (1) and between 3 to 10 years of programming experience. They are presented with the same 100 cases, but with pseudonymized heuristics, and an alternated order of presentation of the results to avoid a serial-position effect. The participants are provided a guide explaining the graphical user interface displaying the edit-scripts. Then for each file pair, we ask the following question: *in a development situation where you need to understand how a code file changed, which one of the following diffs would you prefer?* The rating is sigma (resp. omega) when the preferred output comes from the candidate (resp. baseline) heuristic, or none when it is impossible to decide between both outputs. To resolve the divergence between the participant's ratings, we compute a *majority* rating by assigning for each case sigma (resp. omega) if it has the absolute majority of opinions on the case, and none if not.

# 6 RESULTS

## 6.1 Quantitative Analysis

*How well do our heuristics scale on edit-scripts?* Figure 6 show the running times of the heuristics in our four datasets. These distributions exhibit a similar trend in all our datasets. Firstly, we can notice that the least performant heuristic is opt-1000 (GumTree's default) as expected since it launches the expensive optimal recovery on the biggest subtrees. The median runtime is around 0.1 seconds in all datasets, but we can note some spikes that can reach up to 10 seconds, which is a considerable time to compute a single diff. Decreasing the `max_size` threshold to 100 has the effect of improving the median runtime above 0.01 seconds. This runtime is on par with the runtime of the `hybrid` heuristic, indicating that the running time of the simple recovery is negligible. This is confirmed by the distributions of the runtimes of the `simple` heuristic that are by far the best runtimes, with a median of around 0.001 seconds. Therefore, the results indicate that the best-performing heuristic is the `simple` one, with an improvement of around 99% compared to the `opt-1000` heuristic which is the current GumTree default.

To zoom into the results of the `simple` and `hybrid-100` heuristics compared to the default `opt-1000` heuristic, we show in table 2 the ratio of file pairs where the edit-script resulting from `simple` is computed faster (resp. slower) that the one resulting from `opt-1000`, for the four datasets. For `hybrid-100`, the results on all datasets are very similar, with a ratio of below 80% of faster runtimes for `simple` and above 20% of slower runtimes. For `simple`, the results on all datasets are very similar and the results are even better than for `hybrid-100`, with a ratio of about 90% of faster runtimes for `simple` and 10% of slower runtimes.

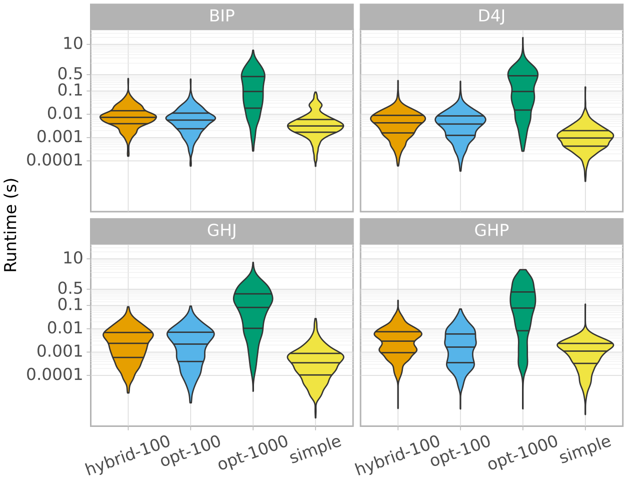


**Figure 6: Distributions of the datasets' runtimes.**

**Table 2: Ratio of file pairs were the runtime of `simple` (resp. `hybrid-100`) is faster (resp. slower) than `opt-1000`.**

| | simple | | hybrid-100 | |
|---|---|---|---|---|
| | faster | slower | faster | slower |
| D4J | 0.94 | 0.06 | 0.78 | 0.22 |
| BIP | 0.90 | 0.10 | 0.78 | 0.22 |
| GHJ | 0.93 | 0.07 | 0.79 | 0.21 |
| GHP | 0.89 | 0.11 | 0.74 | 0.26 |

Finally, table 3 shows the total runtimes of `opt-1000`, `opt-100`, `simple` and `hybrid-100` defined as the sum of median runtime for each file pair. It also shows the speedup, defined as the total runtime of `opt-1000` divided by the total runtime of `opt-100` (resp. `simple` and `hybrid-100`) on all datasets. We note that there is a considerable speedup in all cases. First, increasing the `max_size` threshold from 100 to 1000 (`opt-100` vs `opt-1000`) is very expensive, with a total runtime multiplied from 33 to 52. The runtimes and speedups of `opt-100` and `hybrid-100` are very similar. Finally, the best speedup is achieved by `simple`: between 50x to 281x.

**Table 3: Comparison of the total runtimes (in seconds) for all heuristics and speedup compared to `opt-1000`.**

| | | D4J | BIP | GHJ | GHP |
|---|---|---|---|---|---|
| Runtime (seconds) | opt-1000 | 367.26 | 205.65 | 253.55 | 338.69 |
| | opt-100 | 7.09 | 6.21 | 5.55 | 4.92 |
| | simple | 2.01 | 4.07 | 0.90 | 1.69 |
| | hybrid-100 | 7.31 | 7.83 | 5.50 | 5.86 |
| Speedup vs. `opt-1000` | opt-100 | 51.82 | 33.10 | 45.65 | 68.87 |
| | simple | 183.17 | 50.50 | 281.34 | 200.92 |
| | hybrid-100 | 50.21 | 26.25 | 46.09 | 57.80 |

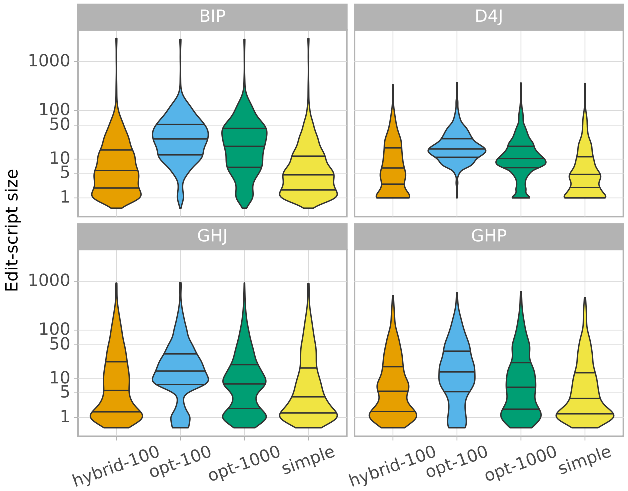


**Figure 7: Distributions of the datasets' edit-scripts sizes.**

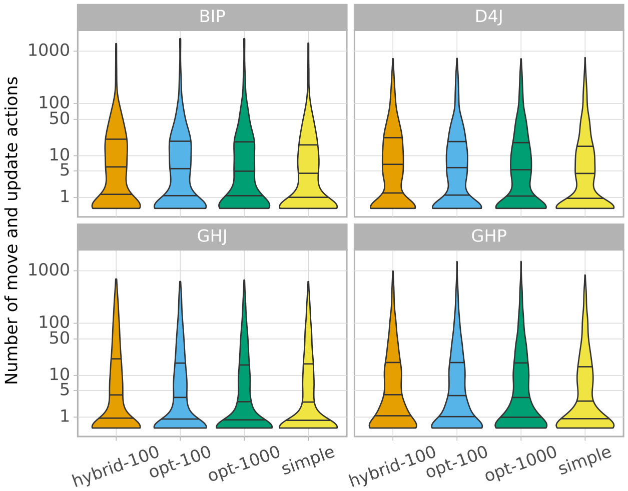


**Figure 8: Distribution of the datasets' number of move and update actions.**

*What is the quality of the output of our heuristics?* Figure 7 shows the distributions of the edit-script sizes of the four datasets. These distributions are very similar. Firstly, we can notice that the longest edit-scripts are produced by the `opt-100` heuristic as expected since it only launches the optimal recovery phases only on small subtrees, and therefore lose many opportunities to find additional mappings. The median size is around 20 actions in all datasets. Increasing the `max_size` threshold to 1000 (`opt-1000`, GumTree's default) has the effect to decrease the median edit-script size to around 10 actions. Interestingly, the `simple` heuristic produces notably smaller edit-scripts than both optimal recoveries with a median of around 5 actions. Moreover, even the `hybrid-100` heuristic does not manage to produce smaller edit-scripts than the `simple` one. Therefore, the results indicate that the smallest edit-scripts are produced by the `simple` and `hybrid-100`, with an improvement of around 50% compared to the `opt-1000` heuristic (GumTree's default).

To ensure that these good results are not at the expense of the move and update actions which are important to reduce the complexity of edit-scripts, we show in fig. 8 the distributions of the move and update actions in the four datasets. The distributions are very similar for the four heuristics, indicating that most move and update actions uncovered by the optimal recoveries are also uncovered by the simple and hybrid recoveries.

To zoom into the results of the `simple` and `hybrid-100` heuristics compared to the default `opt-1000` heuristic, we show in table 4 the ratio of file pairs where the edit-script of `simple` is shorter (resp. equal and bigger) than the one of `opt-1000`, for the four datasets. The results on the bug-fixes datasets (D4J and BIP) are very similar, with a ratio slightly below 75% of shorter edit-scripts for `simple` and around 65% for `hybrid-100`. About 20% of the edit-scripts have the same size for both our candidate heuristics while only 5% of the edit-scripts are bigger for `simple` and 10% for `hybrid-100`. We note a difference with the results on the arbitrary change datasets (GHJ and GHP). The ratio of smaller edit-scripts is smaller (about 40% for `simple` and 30% for `hybrid-100`) while the ratio of edit-scripts with the same size is bigger (about 50% for both heuristics). Finally, the ratio of bigger edit-scripts is slightly increased (about 10% for `simple` and 20% for `hybrid-100`). These results confirm the trend observed in the size distributions: the `simple` heuristic usually produces shorter or similar edit-scripts than `opt-1000`, and very rarely longer edit-scripts. The `hybrid-100` heuristic shows a similar tendency but is less marked.

*Conclusion.* The `simple` heuristic usually produces the shortest edit-scripts with the best runtime. It achieves a median size 50% smaller on all datasets, and a total speedup between 50x and 281x compared to the default `opt-1000`. On the bug fixes (resp. arbitrary changes) datasets, `simple` finds a shorter edit-script with a faster runtime in about 65% (resp. 30%) of the cases.

## 6.2 Qualitative Analysis

In this section, we focus on `simple` which is the best-performing heuristic in the quantitative experiment, both for the runtimes and for the edit-script sizes. Table 5 shows the results of the manual analysis of the 100 file pairs by the authors and external participants. Overall, there were 20 disagreements between the authors out of 100, confirming the need for an independent review. However, for these twenty cases, the authors were able to reach an agreement on the final value. This table confirms the validity of the `size` metric

**Table 4: Ratio of file pairs where the edit-script sizes of `simple` (resp. `hybrid-100`) is smaller, same or bigger than `opt-1000`.**

| | simple | | | hybrid-100 | | |
|---|---|---|---|---|---|---|
| | smaller | same | bigger | smaller | same | bigger |
| D4J | 0.73 | 0.22 | 0.05 | 0.65 | 0.23 | 0.12 |
| BIP | 0.74 | 0.21 | 0.05 | 0.67 | 0.25 | 0.09 |
| GHJ | 0.37 | 0.50 | 0.14 | 0.28 | 0.51 | 0.21 |
| GHP | 0.41 | 0.51 | 0.07 | 0.33 | 0.51 | 0.16 |

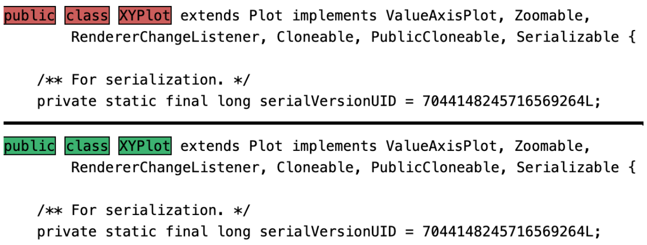
as 83% of the edit-scripts produced by simple were classified as better by the authors than their longer counterparts produced by opt-1000. The opinion of the external participants corroborates the opinion of the authors. Interestingly, the number of cases where simple is preferred to opt-1000 is even greater (91%). This phenomenon mostly comes from the fact that many cases rated mixed by the authors were rated better by the participants. In these cases, the number of unique relevant actions in opt-1000 was lower than the ones of simple.

We note an interesting difference between the bug-fixes and arbitrary changes datasets since the shorter edit-scripts in these datasets are almost always better while this phenomenon is less marked for arbitrary changes datasets. We postulate that this is because bug fixes are usually smaller than arbitrary changes, reducing the odds of irrelevant mappings. These results indicate that the simple heuristic is particularly relevant for automated software repair approaches. In the remainder of the section, we describe three (non-exclusive) representative situations that have been uncovered during the manual analysis of the 100 cases by the authors where the opt-1000 and simple heuristics behave differently.

*Spurious insert-deletes (86 cases).* This situation was very frequently encountered and was one of the design goals of the simple recovery. A prototypical example of this situation is shown in fig. 9 that shows the relevant part of a bug-fix from D4J applied in the JFreeChart project and obtained using the opt-1000 recovery. In this situation, the optimal recovery is not launched due to a subtree size exceeding the max_size threshold. It happens mostly to parent nodes close to the root, here on a ClassDef node. In this case, the ClassDef was mapped to its counterpart during the bottom-up phase, but since the recovery phase was not applied, the children of the class definition (modifiers, class keyword, identifier of the class, ...), which are usually leaves that are not mapped during the top-down phase because they are not considered by default. It leads to some spurious add/remove actions that are confusing since the elements are present in both versions. These spurious actions are not produced by the simple recovery, resulting in much easier-to-understand edit-scripts.

*Aggressive recovery (15 cases).* A more occasional situation that leads the opt-1000 recovery to produce erroneous edit-scripts is called *aggressive recovery*. It comes from the fact that this recovery is sometimes too aggressive, especially in case of major changes inside a method code, because it reuses as many nodes as possible to produce shorter edit-scripts, due to its optimal nature. It can therefore produce some update actions (displayed in yellow or orange in the figures) that are confusing to understand, as we can see in fig. 10. Here, the opt-1000 mapped Util to names, which is not a renaming that a developer would expect. In this situation, simple often produces more readable edit-scripts since it is less prone to reuse nodes abusively. In the same example, simple only produces an update action between the get and getStringArray nodes, which is the only sensible one.

*Missing moves (15 cases).* A third and last occasional situation we encountered where this time the opt-1000 usually yields better results than simple is the *missing moves* one. This situation happens when a leaf node or very small subtree has a different nesting and has not been mapped during the top-down phase. In this situation, simple recovery is not able to discover the mapping since it does not consider changes of nesting, yielding spurious inserts and deletes instead of a move action. Such a situation is shown in fig. 11 where the opt variables are not seen as moved. In the same example opt-1000 correctly mapped the opt variables to their counterparts in the opt.getOpt() expressions, yielding relevant move actions. We tried the hybrid-100 heuristic on the few file pairs falling in this category and it managed to find these move actions. Therefore, the hybrid-100 could perform better than simple in this situation.



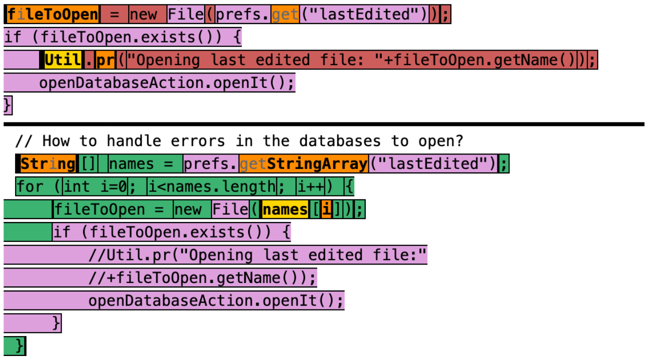
```
public class XYPlot extends Plot implements ValueAxisPlot, Zoomable,
        RendererChangeListener, Cloneable, PublicCloneable, Serializable {

    /** For serialization. */
    private static final long serialVersionUID = 7044148245716569264L;
```

```
public class XYPlot extends Plot implements ValueAxisPlot, Zoomable,
        RendererChangeListener, Cloneable, PublicCloneable, Serializable {

    /** For serialization. */
    private static final long serialVersionUID = 7044148245716569264L;
```

**Figure 9: A *spurious insert-deletes* in JFreeChart (D4J, opt-1000). Inserted (resp. removed) nodes are in green (resp. red). The top (resp. bottom) corresponds to the original (resp. modified) version.**



```
fileToOpen = new File(prefs.get("lastEdited"));
if (fileToOpen.exists()) {
    Util.pr("Opening last edited file: "+fileToOpen.getName());
    openDatabaseAction.openIt();
}
```

```
 // How to handle errors in the databases to open?
 String[] names = prefs.getStringArray("lastEdited");
 for (int i=0; i<names.length; i++) {
      fileToOpen = new File(names[i]);
      if (fileToOpen.exists()) {
          //Util.pr("Opening last edited file:"
          //+fileToOpen.getName());
          openDatabaseAction.openIt();
      }
 }
```

**Figure 10: An *aggressive recovery* in JabRef (GHJ, opt-1000). Inserted (resp deleted, updated and moved) nodes are in green (resp. red, orange, and violet). The top (resp. bottom) corresponds to the original (resp. modified) version.**

**Table 5: Manual comparison of the edit-script quality between opt-1000 and simple.**

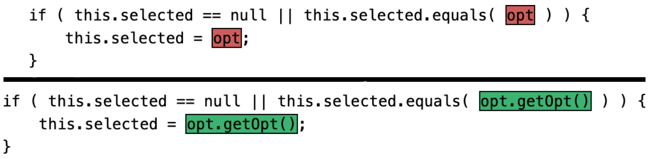
| | authors | | | participants | | |
|---|---|---|---|---|---|---|
| | better | mixed | worse | better | mixed | worse |
| BIP | 23 | 1 | 1 | 24 | 1 | 0 |
| D4J | 22 | 3 | 0 | 23 | 1 | 1 |
| GHJ | 20 | 2 | 3 | 22 | 1 | 2 |
| GHP | 18 | 4 | 3 | 22 | 0 | 3 |
| total | 83 | 10 | 7 | 91 | 3 | 6 |

## 6.3 Threats to Validity

*Construct validity.* The edit-script size is used as a proxy for the quality of an edit-script. However, a short edit-script may be more difficult to understand than a longer one, or contain actions that would be considered irrelevant by a software developer. To complement the use of this metric, we also proceeded to a manual experiment where we assessed the quality of diffs where our candidate heuristic yields a shorter edit-script than the baseline.

*Internal validity.* Measuring a running time is a complex endeavor because it is a very volatile measure that depends on many uncontrollable factors, such as the system load. Our efforts to have a realistic measure include using a dedicated machine with a minimal system performing no other tasks, and the use of five measurements. The authors are involved in the qualitative experiment. Therefore they could be subject to a confirmation bias. To reduce this threat, they used an objective rating scale based on edit-scripts inclusion. We also confirmed the results obtained by the authors by running a second experiment with external participants. Another threat with this manual experiment is the subjectivity of rating the edit-scripts, which could affect both authors and external participants. To mitigate this threat, we resorted to two independent ratings for the two authors with a final harmonization session. For the external participants, we used seven independent ratings that were aggregated using the absolute majority. Another threat w.r.t. to this experiment is that the edit-scripts are visualized using a side-by-side diff through a graphical user interface. Complex diffs could be difficult to analyze using our interface, which could result in participants making arbitrary choices. To reduce this threat, we provided the participants with a *none* rating preventing them from making a choice when they have trouble analyzing the diff. A final threat w.r.t. this experiment is that there is a tiny bug in the Python parser, resulting in an off-by-one visual highlighting of the edit-script. However, the participants were warned about this problem. Finally, to allow scrutiny of our results, we make all edit-scripts used in the experiment available in HTML.

*External validity.* Our datasets contain bug fixes as well as arbitrary changes in Java and Python. However, we cannot claim that these changes are representative of all Java and Python bug-fixes or arbitrary changes respectively. We have therefore no guarantee that our results would generalize to all bug-fixes and arbitrary changes in Java and Python. Additionally, there is no guarantee that the results would generalize to other languages, although the trends in the results do not differ much between the two languages.



```
   if ( this.selected == null || this.selected.equals( opt ) ) {
       this.selected = opt;
   }
```

```
if ( this.selected == null || this.selected.equals( opt.getOpt() ) ) {
    this.selected = opt.getOpt();
}
```

**Figure 11: A *missing moves* in Apache Commons CLI (GHJ, `simple`). Inserted (resp. removed) nodes are in green (resp. red). The top (resp. bottom) corresponds to the original (resp. modified) version.**

# 7 RELATED WORK

Source code differencing has first been investigated using a textual approach, with the ubiquitous *diff* tool [33] that detects inserted and deleted lines of code. Several approaches have tried to extend textual differencing to take into account the possibility of moving lines around, such as [1, 35]. However, the biggest issue of textual differencing algorithms is the impossibility to align the changes to the syntax of the program, which is a significant problem both for change comprehension and for the ability to process changes automatically. Syntactic differencing overcomes this issue and has been developed in foundational articles such as [14, 19] by leveraging approaches designed to tackle tree structures or tree-based documents such as XML [5, 42]. However, the problem of finding an optimal edit-script on a tree structure including move and update actions is NP-hard [3], therefore a lot of work has been dedicated to finding smart heuristics to solve this problem. One foundational heuristic, inspired by XyDiff [7], is GumTree [11], which is the first heuristic, implemented in an open-source tool and able to work on full-fledged abstract syntax trees. Several other heuristics have been proposed over the years by trying to improve on some aspects of GumTree. Dotzler et al. [10] introduced MTDIFF that refines the heuristics of GumTree to find more move actions. Decker et al. [9] and Frick et al. [15] use language-specific information to improve the edit-scripts in respectively IJM and srcDiff. Yang and Whitehead [40] and Matsumoto et al. [31] introduced hybrid approaches that combine textual and syntactic differencing to reduce the size of trees to be matched, thus improving the edit-script size and the time required to compute it. Recently, Fan et al. [13] used a differential testing approach to evaluate the accuracy of GumTree, MTDIFF, and IJM where they show that the GumTree (resp. MTDIFF and IJM) generates inaccurate mappings in 20%-29% (resp. 25%-36% and 21%-30%) of the file revisions, indicating that there is room to improve the existing heuristics. De la Torre et al. [8] defined four categories of accuracy problems found in edit-scripts generated by GumTree on C# programs. Huang et al. [20] and Tsantalis et al. [37] introduced approaches that produce high-level edit-scripts (cluster of actions or refactorings) to improve the understandability of the edit-scripts. Martinez et al. [29] proposed DAT, a hyperparameter optimization approach of AST differencing algorithms. DAT includes types of hyper-optimizations: a) *Global* which aim at finding the optimal hyperparameter values for a particular programming language or language parser, b) *Local*: hyper-optimizes an diff algorithm on just a single case. The evaluation of DAT conducted on GumTree shows that optimizing its hyper-parameters allows GumTree to produce shorter edit-scripts.

In our work, we build on GumTree because we wanted to conserve a language-independent approach that does not need to be adapted to each programming language and because it has been shown by Fan et al. [13] to be a relevant baseline. We introduce a new recovery heuristic that drastically improves the runtime performances, allowing the removal of a tough-to-set threshold and the reduction of the median size of edit-scripts. In contrast to [31, 40], we do not use a hybrid textual and syntactic approach but rather optimize directly the algorithm, although the hybrid approach could be leveraged to improve the speed even further. Finally, hyperparameter optimization as defined in DAT could be applied to gumtree-simple as it has two hyperparameters that could be optimized to, for instance, find shorter edit-scripts. Moreover, DAT could also be used to decide which matcher (between simple, opt, and hybrid) to use in a particular case.

## 8 CONCLUSION

In this article, we presented an improvement of the GumTree heuristic, called gumtree-simple that has two main advantages. It not only produces smaller and easier-to-understand edit-scripts than the original GumTree, but it is also dramatically faster. We validated our new heuristic on two bug-fixes datasets and two arbitrary changes datasets. Our heuristic achieved a consistent 50% decrease in the median edit-script sizes on all datasets, with 40% to 75% of file-pairs having a decreased edit size. Our qualitative experiment confirmed that when our new heuristic manages to reduce the edit-script size, its output is preferred in 91% of the cases by external participants. Our heuristic also consistently outperformed the original GumTree with speed-ups of the matching time ranging from 50x to 281x.

## REFERENCES

[1] 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 *2013 IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, September 22-28, 2013*. IEEE Computer Society, New York, NY, 230–239.

[2] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: learning to fix bugs automatically. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 159:1–159:27. https://doi.org/10.1145/3360585

[3] Philip Bille. 2005. A survey on tree edit distance and related problems. *Theoretical Computer Science* 337, 1 (2005), 217–239. https://doi.org/10.1016/j.tcs.2004.12.030

[4] Saikat Chakraborty, Yangruibo Ding, Miltiadis Allamanis, and Baishakhi Ray. 2022. CODIT: Code Editing With Tree-Based Neural Models. *IEEE Transactions on Software Engineering* 48, 4 (2022), 1385–1399. https://doi.org/10.1109/TSE.2020.3020502

[5] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. 1996. Change detection in hierarchically structured information. *ACM SIGMOD Record* 25, 2 (June 1996), 493–504. https://doi.org/10.1145/235968.233366

[6] Michel Chilowicz, Etienne Duris, and Gilles Roussel. 2009. Syntax tree fingerprinting for source code similarity detection. In *The 17th IEEE International Conference on Program Comprehension, ICPC 2009, Vancouver, British Columbia, Canada, May 17-19, 2009*. IEEE Computer Society, New York, NY, 243–247.

[7] Gregory Cobena, Serge Abiteboul, and Amélie Marian. 2002. Detecting Changes in XML Documents. In *Proceedings of the 18th International Conference on Data Engineering, San Jose, CA, USA, February 26 - March 1, 2002*, Rakesh Agrawal and Klaus R. Dittrich (Eds.). IEEE Computer Society, New York, NY, 41–52.

[8] Guillermo de la Torre, Romain Robbes, and Alexandre Bergel. 2018. Imprecisions Diagnostic in Source Code Deltas. In *Proceedings of the 15th International Conference on Mining Software Repositories* (Gothenburg, Sweden) *(MSR '18)*. ACM, New York, NY, 492–502. https://doi.org/10.1145/3196398.3196404

[9] Michael John Decker, Michael L Collard, L Gwenn Volkert, and Jonathan I Maletic. 2020. srcDiff: A syntactic differencing approach to improve the understandability of deltas. *Journal of Software: Evolution and Process* 32, 4 (2020), e2226. Publisher: Wiley Online Library.

[10] Georg Dotzler and Michael Philippsen. 2016. Move-optimized source code tree differencing. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, David Lo, Sven Apel, and Sarfraz Khurshid (Eds.). ACM, New York, NY, 660–671. https://doi.org/10.1145/2970276.2970315

[11] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, Ivica Crnkovic, Marsha Chechik, and Paul Grünbacher (Eds.). ACM, New York, NY, 313–324. https://doi.org/10.1145/2642937.2642982

[12] Jean-Remy Falleri and Matias Martinez. 2024. *Replication package for Fine-grained, accurate and scalable source differencing.* https://doi.org/10.5281/zenodo.10474673

[13] 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 *Proceedings of the 43rd International Conference on Software Engineering (ICSE '21)*. IEEE Press, New York, NY, 1174–1185. https://doi.org/10.1109/ICSE43902.2021.00108 Place: Madrid, Spain.

[14] Beat Fluri, Michael Würsch, Martin Pinzger, and Harald Gall. 2007. Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Trans. Software Eng.* 33, 11 (2007), 725–743. https://doi.org/10.1109/TSE.2007.70731

[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)*. IEEE, New York, NY, 264–274.

[16] Veit Frick, Christoph Wedenig, and Martin Pinzger. 2018. DiffViz: A Diff Algorithm Independent Visualization Tool for Edit Scripts. In *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018*. IEEE Computer Society, New York, NY, 705–709. https://doi.org/10.1109/ICSME.2018.00081

[17] Ali Ghanbari, Samuel Benton, and Lingming Zhang. 2019. Practical Program Repair via Bytecode Mutation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 19–30. https://doi.org/10.1145/3293882.3330559

[18] Quinn Hanam, Fernando S. de M. Brito, and Ali Mesbah. 2016. Discovering Bug Patterns in JavaScript. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 144–156. https://doi.org/10.1145/2950290.2950308

[19] Masatomo Hashimoto and Akira Mori. 2008. Diff/TS: A Tool for Fine-Grained Structural Change Analysis. In *WCRE 2008, Proceedings of the 15th Working Conference on Reverse Engineering, Antwerp, Belgium, October 15-18, 2008*, Ahmed E. Hassan, Andy Zaidman, and Massimiliano Di Penta (Eds.). IEEE, New York, NY, 279–288. https://doi.org/10.1109/WCRE.2008.44

[20] Kaifeng Huang, Bihuan Chen, Xin Peng, Daihong Zhou, Ying Wang, Yang Liu, and Wenyun Zhao. 2018. ClDiff: generating concise linked code differences. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, Marianne Huchard, Christian Kästner, and Gordon Fraser (Eds.). ACM, New York, NY, 679–690. https://doi.org/10.1145/3238147.3238219

[21] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping Program Repair Space with Existing Patches and Similar Code. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 298–309. https://doi.org/10.1145/3213846.3213871

[22] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 437–440. https://doi.org/10.1145/2610384.2628055 event-place: San Jose, CA, USA.

[23] Serkan Kirbas, Etienne Windels, Olayori McBello, Kevin Kells, Matthew Pagano, Rafal Szalanski, Vesna Nowack, Emily Rowan Winter, Steve Counsell, David Bowes, Tracy Hall, Saemundur Haraldsson, and John Woodward. 2021. On The Introduction of Automatic Program Repair in Bloomberg. *IEEE Software* 38, 4 (2021), 43–51. https://doi.org/10.1109/MS.2021.3071086

[24] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2020. Fixminer: Mining relevant fix patterns for automated program repair. *Empirical Software Engineering* 25 (2020), 1980–2024. https://doi.org/10.1007/s10664-019-09780-z Publisher: Springer.

[25] Xuan-Bach Dinh Le, David Lo, and Claire Le Goues. 2016. History Driven Program Repair. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1*. IEEE Computer Society, New York, NY, 213–224. https://doi.org/10.1109/SANER.2016.76

[26] Kui Liu, Li Li, Anil Koyuncu, Dongsun Kim, Zhe Liu, Jacques Klein, and Tegawendé F. Bissyandé. 2021. A critical review on the evaluation of automated program repair systems. *Journal of Systems and Software* 171 (2021), 110817. https://doi.org/10.1016/j.jss.2020.110817

[27] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. arXiv:2102.04664 [cs.SE]

[28] Fernanda Madeiral, Thomas Durieux, Victor Sobreira, and Marcelo Maia. 2018. Towards an automated approach for bug fix pattern detection. arXiv:1807.11286 [cs.SE]

[29] Matias Martinez, Jean-Rémy Falleri, and Martin Monperrus. 2023. Hyperparameter Optimization for AST Differencing. *IEEE Transactions on Software Engineering* 49, 10 (2023), 4814–4828. https://doi.org/10.1109/TSE.2023.3315935

[30] Matias Martinez and Martin Monperrus. 2019. Coming: a tool for mining change pattern instances from git commits. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, New York, NY, 79–82. https://doi.org/10.1109/ICSE-COMPANION.2019.00043

[31] Junnosuke Matsumoto, Yoshiki 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)*. IEEE, New York, NY, 550–554.

[32] Martin Monperrus. 2018. Automatic software repair: a bibliography. *ACM Computing Surveys (CSUR)* 51, 1 (2018), 1–24. Publisher: ACM New York, NY, USA.

[33] Eugene W. Myers. 1986. An O(ND) Difference Algorithm and Its Variations. *Algorithmica* 1, 2 (1986), 251–266. https://doi.org/10.1007/BF01840446

[34] Mateusz Pawlik and Nikolaus Augsten. 2011. RTED: a robust algorithm for the tree edit distance. *Proceedings of the VLDB Endowment* 5, 4 (2011), 334–345. http://dl.acm.org/citation.cfm?id=2095692

[35] Steven P. Reiss. 2008. Tracking source locations. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn (Eds.). ACM, New York, NY, 11–20. https://doi.org/10.1145/1368088.1368091

[36] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. 2018. PyDriller: Python framework for mining software repositories. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*. ACM Press, New York, New York, USA, 908–911. https://doi.org/10.1145/3236024.3264598

[37] Nikolaos Tsantalis, Matin Mansouri, Laleh Mousavi 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, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, New York, NY, 483–494. https://doi.org/10.1145/3180155.3180206

[38] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation. *ACM Trans. Softw. Eng. Methodol.* 28, 4, Article 19 (sep 2019), 29 pages. https://doi.org/10.1145/3340544

[39] Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, Brian Goh, Ferdian Thung, Hong Jin Kang, Thong Hoang, David Lo, and Eng Lieh Ouh. 2020. BugsInPy: A Database of Existing Bugs in Python Programs to Enable Controlled Testing and Debugging Studies. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1556–1560. https://doi.org/10.1145/3368089.3417943 event-place: Virtual Event, USA.

[40] Chunhua Yang and E. James Whitehead Jr. 2019. Pruning the AST with Hunks to Speed up Tree Differencing. In *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*, Xinyu Wang, David Lo, and Emad Shihab (Eds.). IEEE, New York, NY, 15–25. https://doi.org/10.1109/SANER.2019.8668032

[41] He Ye, Jian Gu, Matias Martinez, Thomas Durieux, and Martin Monperrus. 2022. Automated Classification of Overfitting Patches With Statically Extracted Code Features. *IEEE Transactions on Software Engineering* 48, 8 (2022), 2920–2938. https://doi.org/10.1109/TSE.2021.3071750

[42] K. Zhang and D. Shasha. 1989. Simple Fast Algorithms for the Editing Distance Between Trees and Related Problems. *SIAM J. Comput.* 18, 6 (Dec. 1989), 1245–1262. https://doi.org/10.1137/0218082