HyperDiff: Computing Source Code Diffs at Scale

Quentin Le Dilavrec, Djamel Eddine Khelladi, Arnaud Blouin, Jean-Marc Jézéquel

HyperDiff: Computing Source Code Diffs at Scale

Quentin Le Dilavrec
Univ Rennes, IRISA, Inria, France
firstname.last-name@irisa.fr

Djamel Eddine Khelladi
CNRS, Univ Rennes, IRISA, Inria, France
first-name.lastname@irisa.fr

Arnaud Blouin
INSA, Univ Rennes, IRISA, Inria, France
firstname.lastname@irisa.fr

Jean-Marc Jézéquel
Univ Rennes, IRISA, Inria, France
firstname.lastname@irisa.fr

ABSTRACT

With the advent of fast software evolution and multistage releases, temporal code analysis is becoming useful for various purposes. Temporal code analyses can consist in analyzing multiple Abstract Syntax Trees (ASTs) extracted from code evolutions, e.g. one AST for each commit or release. Core feature to temporal analysis is *code differencing*: the computation of the so-called Diff or edit script between two given versions of the code. However, jointly analyzing and computing the difference on thousands versions of code faces scalability issues. Mainly because of the cost of: 1) parsing the original and evolved code in two source and target ASTs; 2) wasting resources by not reusing intermediate computation results that can be shared between versions. This paper details a novel approach based on time-oriented data structures that makes code differencing scale up to large software codebases. In particular, we leverage on the *HyperAST* [25], a novel representation of code histories, to propose an incremental and memory efficient approach by lazifying the well known GumTree diffing algorithms [11], a mainstream code differencing algorithm and tool. We evaluated our approach on a curated list of 19 large software projects and compared it to GumTree. Our approach outperforms it in scalability both in time and memory. We observed an order-of-magnitude difference: 1) in CPU time from ×1.2 to ×12.7 for the total time of diff computation and up to ×226 in intermediate phases of the diff computation, and 2) in memory footprint of ×4.5 per AST node. The approach produced 99.3 % of identical diffs with respect to GumTree.

CCS CONCEPTS

• **Software and its engineering → Software version control**; **Maintaining software**.

KEYWORDS

Diff, Edit script, Code history mining, Temporal code analysis

ACM Reference Format:
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 (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3611643.3616312

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

1 INTRODUCTION

Code histories are increasingly used during software development as an important source of information for various software engineering tasks, such as analyzing the origin of issues and bugs, learning from code to build recommendation and bug prediction systems [8], recovery of traceability links [3, 4], refactorings [32], duplicate code [28], bad smells and their origins [33], mining of fixes for program repair [23], or co-evolution mining [24, 26]. These kinds of analyses are examples of *temporal code analyses* [1, 25] as they rely on code history (*e.g.,* Git) and their underlying data (*i.e.,* commits, tags, etc.).

Temporal code analyses also consist in analyzing multiple Abstract Syntax Trees (ASTs) extracted from code evolutions, e.g. one AST for each commit or release. A core feature to temporal analysis is *code differencing*, *i.e.,* the computation of the so-called *Diff* or *edit script*[1] between two given versions of the code from two commits or tags [15], which is more complex than to diff two files that does not face scalability issue. Especially that identifying those files is the results of the diff itself and not known a priori in the commits. However, jointly analyzing and computing the difference on thousands versions of code faces scalability issues. Mainly because of the cost of: 1) parsing the original and evolved code to produce two ASTs, and hence, doing so on thousands of commits; 2) wasting resources by not reusing intermediate computation results that could be shared among versions; 3) unsuitable memory layout of compared trees by allocating nodes in the global heap (*i.e.,* not contiguous, indexed by generic pointers, etc.). Many existing works proposed to compute diffs [2, 5, 6, 12, 16, 19, 29, 31] but with scalability issues within a single evolution. Falleri et al. [11] later addressed the scalability issue within a single evolution by reducing the algorithmic complexity to $O(n^2)$ compared to stat-of-the-art techniques. However, they still suffer from a long computation time, and hence, do not scale at commit level on very large software projects and on code histories. To the best of our knowledge, we are the first to propose an approach that addresses the scalability issue of computing diffs on large code histories.

In this paper, we propose a novel code differencing approach that enables the production of diffs/edit scripts at scale on large software codebases. We combine concepts of GumTree [11], a mainstream code differencing algorithm, and *HyperAST* [25], a novel representation of code histories, to propose an incremental and memory efficient approach. In particular, rather that having an AST for each version to analyze, the *HyperAST* leverages code redundancy in space and time using a Direct Acyclic Graph (DAG) to provide a single temporal AST containing all versions at once. On top of the *HyperAST*, we take on the challenge of proposing novel AST matching algorithms, inspired by GumTree, on this DAG representation

[1] In the rest of the paper we only use the term *Diff*.

instead of the classical and inefficient analyses of a pair of full ASTs. Essentially, we make the original greedy GumTree algorithms lazy. Our approach pre-computes metadata and lazily decompresses the DAG to decorrelates the cost of diffing from the size of the code base. Thus, enabling code differencing at scale.

We evaluated our approach on a curated list of 19 large software projects. Compared to GumTree as a baseline, the proposed approach outperforms it in scalability both in time and memory. We observed an order-of-magnitude difference in CPU time: 1) from ×1.2 to ×12.7 for the total time of diff computation, 2) from ×1 to ×226 for the top-down and bottom-up phases total time, and 3) from ×3.2 to ×233 for the top-down phase total time. We also observed an order-of-magnitude difference in memory footprint of ×4.5 per AST node. Finally, we gain all the time while having a 99.3 % of identical diffs with respect to GumTree and 99.999 75 % of identical mappings and actions in the remaining 0.7 % diffs. Finally, we also outperform GumTree-Spoon when including the parsing cost. We are faster by 14.52 times in median and when excluding extreme cases of gains, we are faster on average by 13.68 times.

This paper follows the *Engineering Research Method* as we propose a novel scalable code differencing approach and evaluated through case studies supplemented with a replication package.

This paper makes the following contributions:

- A novel approach for diffing commits that scales for the analyses of large code histories.
- An evaluation of the proposal composed of benchmarking studies that demonstrates the ability of the approach to scale compared to a state-of-the-art code differencing approach.
- Open Science: All the code of the approach and the artifacts of the evaluation are freely available as a replication package. https://zenodo.org/record/8270267

The rest of the paper is structured as follows. Section 2 introduces background concepts for understanding the proposed approach. Section 3 presents the novel approach. Section 4 details the evaluation and threats to validity. Section 5 discusses related work. Section 6 concludes the paper and outlines future work.

2 BACKGROUND

This section introduces the two main concepts we rely on: computation of diffs and code history representation.

2.1 Computing diffs

As software evolves quickly, one way to study its evolution is through computing source code *diffs*. A diff is usually computed using a source and a target AST. It is composed of actions (*i.e.,* changes) that represent the applied changes from an original version to an evolved version of the code. Actions are either atomic or composed. An atomic action can be either an *insert (add), delete (remove), or update* of AST nodes. A composed action is an action composed of other actions. For example a *move* action is composed of a delete and an insert (possibly also an update), which moves a given node (deletion) to another place in the AST (insertion).

A mainstream and efficient source code differencing approach is GumTree [11] that uses two dedicated algorithms to map original and evolved ASTs (as depicted by the left part of Figure 1). GumTree then produces diffs using the Chawathe algorithm [6] with the obtained mappings as input. The first mapping phase is the **top-down matching**. It is a top-down traversal (breadth-first) that consists in matching subtrees in rounds (level by level starting from the root). Each round, unmatched subtrees are opened, *i.e.,* the root of the subtree is skipped and its children are made available to be matched on the next round. The second mapping phase is the **bottom-up matching**, a bottom-up traversal (post-order) that matches previously unmatched tree nodes using an optimal matching algorithm and the previously matched subtrees. In post-order, unmatched nodes are compared to candidates (*i.e.,* other nodes that share mapped descendants) in the other version. Then, the most similar candidate is selected to apply an optimal matching algorithm such as RTED [30] or Zs [35]. GumTree uses Zs by default. Note that GumTree uses and computes the following metadata:

- the *height* of a subtree (in number of nodes);
- the *size* of a subtree (in number of nodes or characters);
- the *structural hash*: the hash of a subtree that depends on the type and order of nodes (ignore labels);
- the *label hash*: the hash of a subtree that depends on the type, label, and order of nodes.

While the original GumTree focuses on diffing files, we diff at commit level, which is more complex as it has to scale. GumTree-Spoon further integrates the Spoon parser into GumTree in order to diff commits. Diffing commits is useful to analyze the applied changes, doing so at scale can open new possibilities to track changes [14, 15, 17, 18, 22] to a code block, an attribute, or a method on thousands of commits. Cregit [14] improves further the tracking Git 'blame', but works at the token level and not at the AST level. However, such scalable tracking would be relevant, for example, to fully study the origin of fixed security issues in very large code base [21]. Other scenarios can be about identification of distant co-evolutions or step by step introduction of security breaches.

2.2 Efficient representations of histories

Code is usually analyzed as a tree and augmented with additional edges, such as referential relations. However, studying it on multiple versions may face scalability issue. Two approaches handling scalability representation of code histories emerged in the literature [1, 25]. With LISA, Alexandru *et al.* [1] represent code as a graph, where nodes of consecutive versions can be merged with a decimation algorithm. With the *HyperAST* [25], Le Dilavrec *et al.* represent code as a Direct Acyclic Graph (DAG) where subtrees are deduplicated. It takes inspiration for the Merkle DAG of Git but does not limit itself to files as leafs of the DAG. For each subtree, the *HyperAST* allows a simple and efficient reuse of computation that only depends on code in the subtree, in contrast to Lisa.

3 CONTRIBUTION

Considering a code history represented using the *HyperAST*, we introduce a lazy code differencing approach inspired by the GumTree approach [11] with a faster and more efficient mapping of pairs of trees (that characterize commits). The contribution addresses the scalability issues as the following:

1/ the use of the *HyperAST* data structure overcomes the wasting of resources by not reusing intermediate computation results that could be shared among versions;

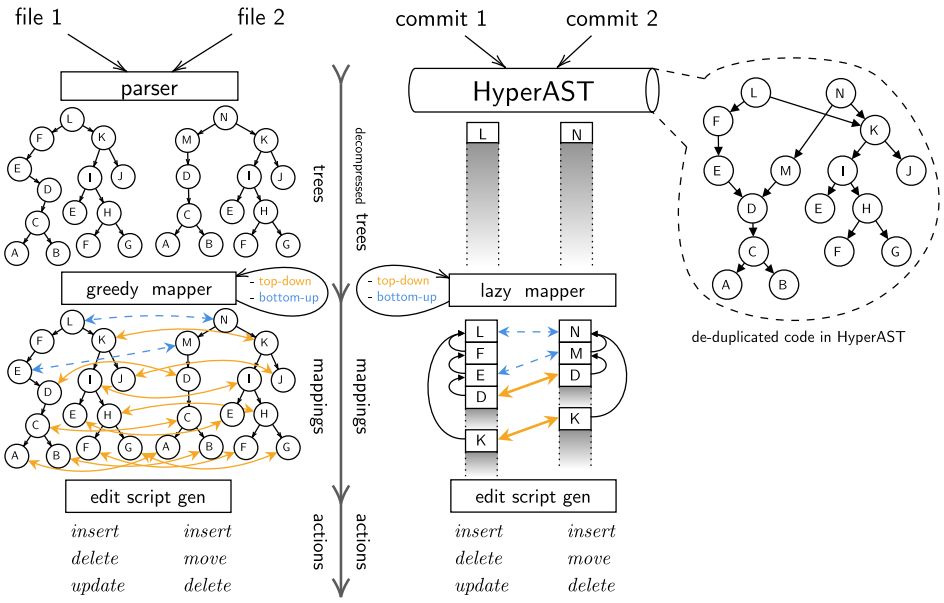

Figure 1: GumTree pipeline (left) ; our pipeline with content of *HyperAST* (right)

2/ the proposed lazy algorithm fixes the unadapted memory layout of compared trees by allocating nodes in the global heap;
3/ combining the *HyperAST* with the lazy diff algorithm reduces unneeded memory accesses to the ASTs during the diff algorithm.

The scaling capabilities of our approach benefit from the same hypothesis as the *HyperAST*: given a large code base, the amount of changes (*i.e.,* new subtrees) brought by each commit is usually tiny compared to the size of the code base. If both *HyperAST* and GumTree handle subtrees, *HyperAST* de-duplicates identical subtrees (*i.e.,* stores unique subtrees only). Thus, providing potential benefits if combined with specific algorithms for matching identical subtrees. Compared to the GumTree approach, the one we propose leverages the structure of the *HyperAST* to significantly reduce memory accesses and cache misses, while speeding up the diff.

Figure 1 shows the main differences between the GumTree approach and ours: The GumTree approach starts by parsing a pair of code files, then process resulting trees to compute metadata and finally produce diffs. It follows a top-down and bottom-up phases to compute the mappings between the original and evolved versions of the ASTs. After that, it computes the diff. Instead, our approach relies on the *HyperAST* to efficiently process a code history, *i.e.,* a Git repository. For each version (*i.e.,* commit), our approach incrementally parses and computes metadata to persist. In addition to persisting metadata, the *HyperAST* also precomputes the structural equality using a reference equality, since identical subtree are de-duplicated. Then, pairs of trees from the *HyperAST* are lazily decompressed and mapped also in the same two phases (top-down and bottom-up) and finally used to produce diffs. Thus, the lazy decompression allows us to only focus on the changed parts of the code for diffing.

In this section, we first detail how the approach leverages *HyperAST* to provide structured code with metadata, and then how those structured data fit the requirements of each matching algorithm we propose. We then present the two specific matching algorithm, namely the lazy top-down matching and the lazy bottom-up matching. Our contribution herein is to adapt the original greedy algorithms of GumTree for the decompressed trees, leveraging on the lazy decompression. While optimizing the performances, our approach produces the same results as the original algorithms.

3.1 GumTree to *HyperAST* metadata

To efficiently compare code elements, GumTree uses several precomputed metadata (see Section 2). As our approach leverages on the *HyperAST* that needs to compute and expose these metadata. The size and a hash (similar to the label hash) were already present in the *HyperAST*. Thus, we only extended the construction of the *HyperAST* (*i.e.,* parsing commits) to compute the label and structural hashes. In addition to these metadata, GumTree needs to test whether two subtrees are identical, *i.e.,* an *isomorphic* function. However, due to the DAG nature of the *HyperAST*, identical subtrees are de-duplicated. Thus, we know that referentially identical subtrees are isomorphic without having to recursively compare their content as GumTree does in its implementation.

3.2 Obtaining a tree from the *HyperAST*

We now present the compressed tree and its lazy decompression.

3.2.1 Decompressed tree. The *HyperAST* is a DAG where subtrees are unaware of their parents, as such, algorithms requiring global information on nodes (*i.e.,* subtrees) need additional structures. Global information refers to any information on parents of a node. It can be the global position of a node, its path, declaring class, file or offset in characters. In the remainder of this paper – as opposed to the compressed tree (the DAG) in *HyperAST*– we call such a structure that holds global information on nodes a **decompressed**

tree. The process of extracting a decompressed tree from the *HyperAST* is named a **decompression**. To exploit spacial locality, a decompressed tree is represented by a contiguous array using a post-order layout. Actually, the bottom-up step mainly traverses trees in post-order, process subtrees (descendants) and other post-order properties, such as contiguous descendants, key roots and leftmost tree descendants. It has almost no downsides, other than having a $O(n)$ access time to access the n-th children (precomputing leftmost tree descendants is mandatory to obtain this complexity).

A decompressed tree has the following structure (inspired by the Zs algorithm [35], see Section 2):

ids: an array of Ids that indexes subtrees in the *HyperAST*
llds: an array of integers that indexes leftmost tree descendants
parents: an array of integers that indexes parents

The decompressed tree is column-oriented, *i.e.,* is a struct of arrays[2], while nodes are indexed by their position in post-order. In addition, considering that a decompressed tree is contiguous we are able to replace the uses of hash sets by bit sets[3]. Indeed, the original GumTree algorithm [11] uses existential quantifier (∃) in various places and implemented by hash sets.

The downside of this decompressed tree would lie in the upfront cost of decompressing two entire versions before being able to compare them. Nonetheless, it is countered by the fact that it is lazily decompressed.

3.2.2 Lazy decompression. The upfront decompression of the *HyperAST* is actually not mandatory. Reducing and deferring the decompression effectively makes the decompression lazy. Indeed, considering the original GumTree matching algorithms and depending on the amount of changes and where they are located, entire subtrees might remain unchanged and will be matched early (using metadata and referential equality) without ever needing to access their descendants. In those cases, there is no need to decompress these subtrees. In Figure 1, a subtree with compressed descendants is materialized by dotted cells (fat red arrows means all descendants are matched uniformly). To control the decompression process while computing the diff, we provide three different methods to decompress a tree T:

***decompress_children*(T, t):** decompresses in T the children of the node located at position t.

***decompress_to*(T, t):** decompresses in T the node located at position t. It also decompresses all its parents with the method *decompress_children*.

***decompress_descendants*(T, t):** decompresses in T the descendants located at position t. It offers optimization opportunities when considering the layout of the decompressed tree (*e.g.,* post-order). Indeed, for the post-order layout, it is possible to reduce the number of accesses to the *HyperAST* with a stack that allows to defer the insertion of children when decompressing a node. Thus, there is no need to access the size (metadata in the *HyperAST*) of those children before actually decompressing them.

[2]Structs of arrays (SOA) reduce memory wasted by padding, while helping with cache misses when only a subset of fields is needed.

[3]Using a bit sets to implement a set, is more memory efficient than a hash set (a single bit per element), considering modern virtual memory, where zeroed pages are not physically allocated.

Each decompression method is incremental. Thus, making the overall decompression incremental. To check whether a node is decompressed, we check if its parent is 0 (*i.e.,* its initial value). This property always holds, except for the edge case where $|T| <= 1$ *i.e.,* the tree is a single node. These three methods are used as follows in the proposed matching algorithms: Methods *decompress_children* and *decompress_descendants* replace every call to the original (non decompressing) children accessor. The method *decompress_descendants* is specifically used when most or all descendants need to be decompressed. The method *decompress_to* should be used when a specific node needs to be decompressed.

3.3 Lazyfied Top-down Mapping Phase

Algorithm 1[4] summarizes the top-down phase that matches the largest isomorphic subtrees between the source T_1 and target T_2. The underlined expressions in Algorithm 1 represents our optimizations for lazifying the GumTree top-down phase.

The following constructs are required to understand Algorithm 1:

- $root(T)$ is the root node of T.
- $s(t)$ returns the list of descendants in post-order *i.e.,* from $lld(t)$ to t (see Section 3.2).
- $BitSet(n)$ is an array of bits of size n.
- $PList(t)$ creates a height-indexed priority tree list L containing the subtree t.
- $peekMax(L)$ returns the greatest height of the list.
- $pop(L)$ takes the list of greatest height subtrees from L.
- $open(L, t)$ inserts all the children of t into L. Algorithm 2 presents this function. Compared to GumTree we changed the access to children into a decompression of said children (using $decompress_{children}$). Then, instead of accessing a precomputed height directly on the node (as GumTree does), we need to access the subtree corresponding to *HyperAST*, first recovering the identifier (with *original*, see Section 3.2) to the *HyperAST*, then accessing the height metadata (with *height*, see Section 2)
- $sim(t_1, t_2)$ computes a similarity distance between t_1 and t_2. It ranges from 0 to 1, where a value of 1 indicates that the descendants of T_1 are the same as those of T_2.

$\mathcal{MM}$ represents a structure containing multi-mappings *i.e.,* mappings where nodes can be part of multiple mappings as opposed to $\mathcal{M}$ that only contains distinct mappings.

- $allSrcs(\mathcal{MM})$ returns all mapped sources in $\mathcal{MM}$.
- $srcs(\mathcal{MM}, t_2)$ returns all source nodes mapped to t_2 in $\mathcal{MM}$.
- $dsts(\mathcal{MM}, t_1)$ returns all destination nodes mapped to t_1 in $\mathcal{MM}$.
- $link(\mathcal{MM}, t_1, t_2)$ maps t_1 and t_2 in $\mathcal{MM}$.

Algorithm 1 follows three steps. The first step (lines 1-19) calculates the multi-mappings (MM) between the largest isomorphic subtrees. It maps isomorphic subtrees (Lines 14-15), while iteratively opening unmapped subtrees. More specifically, when the

[4]Note that we use in Algorithms 1 to 3 the same hyperparameters as GumTree [11], namely, $minHeight = 2$, $maxSize = 100$, and $minDice = 0.5$.

Algorithm 1: Lazy subtree matching

Data: A source tree T_1 and a destination tree T_2, a multi map $\mathcal{MM}$, an empty list $\mathcal{A}$ of candidate mappings, and an empty set of mappings $\mathcal{M}$, the minimum height for a matched subtree *minHeight*

Result: The set of mappings $\mathcal{M}$

1 $L_1 \leftarrow PList(root(T_1))$;
2 $L_2 \leftarrow PList(root(T_2))$;
3 **while** $min(peekMax(L_1), peekMax(L_2)) > minHeight$ **do**
4 **if** $peekMax(L_1) \neq peekMax(L_2)$ **then**
5 **if** $peekMax(L_1) > peekMax(L_2)$ **then**
6 **foreach** $t \in$ pop (L_1) **do** open(t, L_1);
7 **else**
8 **foreach** $t \in$ pop (L_2) **do** open(t, L_2);
9 **else**
10 $H_1 \leftarrow pop(L_1)$; $H_2 \leftarrow pop(L_2)$;
11 $b_1 \leftarrow BitSet(|H_1|)$;
12 $b_2 \leftarrow BitSet(|H_2|)$;
13 **foreach** $(i_1, i_2) \in 0..|H_1| \times 0..|H_2|$ **do**
14 **if** $isomorphic(H_1[i_1], H_2[i_2])$ **then**
15 $link(\mathcal{MM}, H_1[i_1], H_2[i_2])$;
16 $b_1[i_1] \leftarrow 1$;
17 $b_2[i_2] \leftarrow 1$;
18 **foreach** $i_1 \in 0..|H_1|$ ***if*** $\neg b_1[i_1]$ **do** open$(H_1[i_1], L_1)$;
19 **foreach** $i_2 \in 0..|H_2|$ ***if*** $\neg b_2[i_2]$ **do** open$(H_2[i_2], L_2)$;
20 $ignored \leftarrow BitSet(|T_1|)$;
21 **foreach** $t_1 \in allSrcs(\mathcal{MM})$ **do**
22 $uniq \leftarrow \bot$;
23 **if** $|dsts(\mathcal{MM}, t_1)| == 1$ **then**
24 $t_2 \leftarrow dsts(\mathcal{MM}, t_1)[0]$;
25 **if** $|srcs(\mathcal{MM}, t_2)| == 1$ **then**
26 $uniq \leftarrow \top$;
27 add all pairs of isomorphic nodes of $s(t_1)$ and $s(t_2)$ to $\mathcal{M}$;
28 **if** $ignored[t_1] \vee uniq$ **then** continue;
29 **foreach** $t_1 \in srcs(\mathcal{MM}, dsts(\mathcal{MM}, t_1)[0])$ **do**
30 $ignored[t_1] \leftarrow 1$;
31 **foreach** $t_2 \in dsts(\mathcal{MM}, t_1)$ **do**
32 $add(\mathcal{A}, (t_1, t_2))$;
33 $sort\ (t_1, t_2) \in \mathcal{A}$ using $sim(t_1, t_2, \mathcal{M})$
34 $ignored\_src \leftarrow BitSet(|T_1|)$;
35 $ignored\_dst \leftarrow BitSet(|T_2|)$;
36 **foreach** $(t_1, t_2) \in \mathcal{A}$ **do**
37 **if** $\neg ignored\_src[t_1] \wedge \neg ignored\_dst[t_2]$ **then**
38 add all pairs of isomorphic nodes of $s(t_1)$ and $s(t_2)$ to $\mathcal{M}$;
39 $ignored\_src[t_1] \leftarrow 1$; $ignored\_dst[t_2] \leftarrow 1$;
40 **foreach** $t \in s(t_1)$ **do** $ignored\_src[t] \leftarrow 1$;
41 **foreach** $t \in s(t_2)$ **do** $ignored\_dst[t] \leftarrow 1$;

Algorithm 2: Open a subtree in priority list

Data: a subtree $t \in T$, T being layouted in post-order, and a height-indexed priority list L containing subtrees of T

Result: The range of descendants in post-order

1 **foreach** $t' \in decompress\_children(t)$ **do**
2 $h \leftarrow height(original(t')) - 1$;
3 **if** $h > minHeight$ **then**
4 $level \leftarrow maxHeight - h$;
5 $L[level] += t$;

heights are not equals for the subtrees (Lines 4-8) or when they are not mapped in Line 15 with $b_i = 0$ (Lines 18-19).

The second step of Algorithm 1 (Lines 20-33) moves multi-mappings stored in $\mathcal{MM}$ to a list of mappings $\mathcal{A}$ (Line 33), while directly moving mono-mappings (*i.e.,* mappings between exactly 2 nodes) to $\mathcal{M}$ (Line 27). It mainly serves as a preprocessing phase before sorting and filtering multi-mappings. It also removes mono-mappings, and hence, reduces the number of mapping to sort and filter. It uses a bit set to mark nodes that are already mapped (Line 20). Then, Algorithm 1 sorts the list mappings in $\mathcal{A}$ using a similarity distance (Line 33), such as the *dice* distance used in GumTree [11].

The last step (Lines 36-41) extracts mappings from $\mathcal{A}$ to $\mathcal{M}$ (Line 39), while filtering overlapping mappings with an already accepted one, *i.e.,* sharing nodes. Nodes accepted in $\mathcal{M}$ are marked by *ignored_src* and *ignored_dst* (Line 40-41) that forbids them from being accepted later (Line 38).

3.4 Lazyfied Bottom-up Mapping Phase

The bottom-up phase, as shown in Algorithm 3[4], complements the top-down phase, leveraging the previously mapped subtrees in $\mathcal{M}$ to further map the remaining nodes. The underlined expressions represent our optimizations for lazifying the GumTree Bottom-up phase. Using the subtrees mapped in previous phase, Algorithm 3 is able to map slightly different nodes (*i.e.,* not isomorphic nodes). The matcher first compares the number of shared descendants, to then match subtrees smaller than *minHeight* (leveraging existing optimal mapping algorithms).

With the bottom-up phase, in post-order, we aim to map remaining (unmapped) source nodes (Line 1) to destination nodes. We only decompress the unmapped source nodes (Line 2) before skipping the nodes with no matched children (Line 3). Then, the auxiliary *candidate* function is used to find a candidate destination node t_2 (Line 4) most similar to t_1. Using the Dice distance (compared to *minDice* GumTree parameter), if t_1 and t_2 are similar enough (Line 5), they are matched in $\mathcal{M}$ (Line 6). If t_1 and t_2 have a small number of descendants (compared to *maxSize* GumTree parameter), their descendants are then decompressed and provided to *opt* (Lines 7-10). *opt* is an optimal matching algorithm (such as Zs) that matches nodes of u_1 and u_2 while minimizing the edit distance. Finally, only mappings with unmatched nodes and same types are kept and added in $\mathcal{M}$. To generate the diff, we use the same algorithm of Chawathe et al. [6] without lazifying it in this paper.

Algorithm 3: Lazy bottom-up matching

Data: A source tree T_1 and a destination tree T_2, and an empty set of mappings $\mathcal{M}$, a threshold *minDice* and a maximum tree size *maxSize*
Result: The set of mappings $\mathcal{M}$

```
foreach s1 ∈ T1 | s1 is not matched, in post-order do
    t1 ← decompress_to(T1, s1);
    if t1 has no matched children then continue;
    t2 ← candidate(t1, M);
    if t2 ≠ null ∧ dice(t1, t2, M) > minDice then
        M ← M ∪ (t1, t2);
        if |s(t1)| < maxSize ∨ |s(t2)| < maxSize then
            u1 ← decompress_descendants(t1);
            u2 ← decompress_descendants(t2);
            R ← opt(u1, u2);
            foreach (ta, tb) ∈ R do
                if ta, tb not already mapped ∧
                  type(ta) = type(tb) then
                    M ← M ∪ (ta, tb);
```

4 EVALUATION

This section presents the evaluation of our code differencing approach. First, we present the research questions. We then present the data set and evaluation process. After that, we present our evaluation protocol and the obtained results. We finally discuss the threats to validity, limitations, and the scope of the approach. **All the material of this section and a replication package are available on our companion web page**[5]. We ran the implementation of our approach on the following hardware configuration: *2 x Intel(R) Xeon(R) Gold 6238 CPU @ 2.10GHz; 187Gb ram; 1 T SSD*, running *Ubuntu 18.04.6*.

4.1 Research Questions

We formulate the research questions as follows:

RQ1 **To what extent can our approach produce identical results as the state-of-the-art technique?** This aims to investigate the soundness of the produced mappings and diff compared to a ground truth, namely GumTree.

RQ2 **To what extent does our approach perform and scale on the memory footprint** of computing diffs compared to a state-of-the-art approach? This aims to position the scalability performance on memory consumption of our diffing algorithm over a long evolution history with an established state-of-the-art solution. In particular, we measure the memory heap allocated per node.

RQ3 **To what extent does our approach perform and scale on the time performance** of computing diffs compared to a state-of-the-art approach? This aims to position the scalability performance on execution time of our diffing algorithm over a long evolution history with an established state-of-the-art solution. In particular, we measure the total time to compute a diff and the time for the two phases of top-down and bottom-up of the mappings.

RQ4 **To what extent does our approach perform compared to a state-of-the-art approach on a practical use case of parsing and diffing commits?** The previous RQs evaluated the performance of our commit diffing algorithm. Thus, ignoring the AST preparation (extraction and construction) from a history. RQ4 measures the cost of parsing the commits along with the cost of computing the diffs. It works on a concrete use case as experienced by a developer that computes commit diffs on a code history.

[5] https://zenodo.org/record/8270267

Table 1: Data set characteristics, from [25].

Projects	# LoC	# files	Commits	Contributors	Stars
Apache Hadoop	1.63M	10.2k	25,749	435	12k
Apache Flink	1.5M	13.2k	30,587	1,037	1.8k
Quarkus	614k	10.5k	29,635	616	9.7k
Google Guava	509k	3.16k	5,794	273	44k
Netty	317k	2.78k	10,789	569	29k
Apache Dubbo	197k	2.81k	5,437	393	37k
Alibaba fastjson	188k	3.12k	3,946	176	24k
Apache Log4j2	183k	2.32k	12,031	132	2.8k
Jenkins	181k	1.69k	32,252	701	19k
Javaparser	179k	1.67k	8,031	166	4.1k
Inria Spoon	154k	2.06k	3,891	106	1.3k
AWS Toolkit Eclipse	93.9k	1.08k	111	21	27k
Apache Maven	92.5k	1.05k	11,567	150	3.1k
Apache Spark	85.6k	1.06k	32,821	1,805	33k
Apache SkyWalking	84.7k	1.58k	7,022	397	1.9k
Jackson Core	52.3k	283	2,025	59	200
Alibaba Arthas	44.2k	586	1,726	155	29k
Google gson	25.8k	212	1,650	124	21k
SLF4J	13.5k	256	1,956	61	1.9k

4.2 Dataset

Table 1 details the final list of software projects we used in the following evaluation. The evaluation re-used the dataset employed in the *HyperAST* paper [25]. It contains large open-source real-world Java projects with a large number of commits per history. Thus, serving as a representative code histories in our evaluation.

4.3 Evaluation Protocol

We now present the experimental protocol we followed for the evaluation. As GumTree is the most advanced state-of-the-art differencing tool, we select it as a baseline. The evaluation protocol is divided into three parts: one protocol for RQ1, another one for RQ2 and RQ3, and one specific protocol for RQ4. These four RQs use several of the following five objects:

GumTree. The original version of GumTree in Java without its Java parser. Used in RQ1 to RQ3.

Lazy. The current proposed approach (in Rust), relying on the *HyperAST* version with lazy top-down and bottom-up phases. Used in RQ1 to RQ4.

Not lazy. The closest equivalent of GumTree Java but in Rust and relying on the *HyperAST*. This object is useful to mitigate comparison between Java and Rust program executions. This version still benefits from the *HyperAST* but no lazy phases. Used in RQ3.

Partial lazy. Similarly to *Not lazy* but lazy on the top-down phase. Useful to measure the effect of not lazifying during the bottom-up phase. Used in RQ3.
GumTree-Spoon[6]. The original version of GumTree in Java backed with its official Java parser (Spoon). Used in RQ4.

We now detail the protocol for each RQ:
⇒ **RQ1.** It compares the results of our approach (object *Lazy*) to the baseline (*i.e.,* object *GumTree*). To do so, we check whether each diff *Lazy* and *GumTree* produced are identical. The independent variables are the mappings found in a diff and the actions a diff contains. Thus, We compare the mappings and the diff's actions for our approach against GumTree.
⇒ **RQ2.** It focuses on the following two objects: *GumTree*, *Lazy*. *Partial Lazy* and *Not Lazy* are not discussed here as they only differ from *Lazy* about when memory is used, not the total amount consumed. The independent variables for RQ2 are: the measured memory heap allocated using the objects *Lazy* and *GumTree*; the number of nodes used in both objects (both *Lazy* and *GumTree* use the same number of nodes). We then divided the measured heap size by the number of nodes to obtain a result in byte per node.
⇒ **RQ3.** It studies the four following objects: *GumTree*, *Lazy*, *Not Lazy*, *Partial Lazy*. The independent variable in RQ3 is the execution time: first, the total time spent (top-down, bottom-up and computing the diff actions from the mappings with Chawathe algorithm [6]); then, only the two matching phases (top-down and bottom-up); finally, only the top-down phase.
⇒ **RQ1-3.** Regarding RQ1 to RQ3, the first step of the protocol consists in parsing the various commits to provide to GumTree and to construct the *HyperAST*. This is a preprocessing step before computing the diffs. Then, we provide the resulting parsed trees to our approach and to GumTree. To precisely evaluate the commit diffing algorithm, we provide *HyperAST* and GumTree with the same ASTs for these three RQs. Only after that, we start measuring the performance of the different phases and algorithms for computing the diffs. So, the parsing time for GumTree and construction time of the *HyperAST* are not considered in RQ1, RQ2, and RQ3. Thus, ensuring a more controlled measurements of the algorithmic performances and an unbiased comparison.
⇒ **RQ4.** It studies the two following objects: *GumTree-Spoon* and *Lazy*. The independent variable in RQ4 is the execution time. We compare the spent time at computing diffs of the latest 100 commits for each project while including the ASTs parsing time for GumTree-Spoon and construction time of the *HyperAST*. Indeed, the combination of Spoon with GumTree allows us to feed entire commits as ASTs to the GumTree algorithm with the Spoon parser.

4.4 Results

We now present and discuss the observed results.

4.4.1 RQ1. To answer RQ1, we compare the mappings and diffs produced by our approach to the GumTree baseline. In total, we calculated 18 092 diffs implying 919 132 mappings[7]. 17 972 (99.3 %) of these diffs were identical by matching the GumTree results identically. 99.999 % of the 919 132 mappings of the diffs are also identical.

We manually scrutinized and checked the other 120 (0.7 %) diff and found out the following edge cases. 1) for 64 cases, an error occurred during the diff generation, and hence, not computing fully the final diff actions. However, when comparing their mappings they were identical. 2) for 2 cases our diff had 2 more actions than the GumTree diff. 3) for the 54 other cases, our diff had less actions than the GumTree diff. This last 54 cases are explained by the fact that we could calculate more mappings between the AST nodes than what GumTree did. In fact, these cases could highlight better diffs since they do not contain additional add and delete actions due to unmapped AST nodes. Overall, even in these 120 diffs, 99.999 % of mappings and 99.943 % of the diff actions are identical. Only, few mappings and diff actions cause comparison issues. Therefore, we consider those marginal cases as outliers due to implementation issues in our prototype or our execution environment in comparison to the more than 99 % of correct diffs and mappings.

> ***RQ*$_1$ insights:** The results show that 99.3 % of the diffs and 99.999 % of the mappings our approach produced are identical to the GumTree outputs. 120 diffs where not identical to GumTree. Still, they remain similar at 99.943 % of diff actions and at 99.999 % of mappings. Thus, our approach produces identical results that GumTree on the involved data set.

4.4.2 RQ2. To answer RQ2, we measure the memory heap allocated given the same number of nodes for our approach and for GumTree. On average, GumTree needs 74.4 bytes per node. Our implementation needs 0.278 byte per node in the DAG (over 1000 commits) and 16.13 additional bytes per decompressed node. Our approach decompresses nodes lazily. Considering the allocation of a zeroed (0) and contiguous piece of memory, on modern operating systems such an allocation is deferred at the granularity of a virtual memory page. Thus, with our approach physical memory is only allocated when a node is decompressed (turned from zeros to ones). The peak (transient) memory footprint of our approach occurs during the diff generation at the exact same stage as the original GumTree implementation. Indeed, we did not make the Chawathe *et al.* [6] algorithm lazy as it is not a focus of the core contribution on computing the mappings, both in our approach and GumTree. Moreover, the Chawathe algorithm (diff generation) uses a third tree that starts as a copy of the source tree and is mutated for each change until it structurally becomes the destination tree.

> ***RQ*$_2$ insights:** Results show out low memory footprint compared to GumTree with, on average, 0.278 + 16.13 bytes per node in our approach versus 74.4 bytes per node in GumTree. Representing an order-of-magnitude difference of ×4.5.

4.4.3 RQ3. Herein, we measure time at the three different stages of the GumTree algorithm, namely the total time spent; the two matching phases (top-down and bottom-up phases); and the top-down phase only. Figures 2 to 4 shows respectively the results for the three aforementioned stages. Each figure uses the same plotting scheme: the time taken is displayed as vertical box plots on a logarithmic scale where the *mean* is a thick horizontal black line and the *median* is a thin colored horizontal line. The box delimits the first and third quantile. The vertical bar delimits the 95 % confidence interval. Extreme points are displayed outside this interval.

[6] https://github.com/SpoonLabs/gumtree-spoon-ast-diff
[7] Distribution of change size per diff is given in our companion web page in *size-plot.png*, varying from small changes to very large changes several commits.

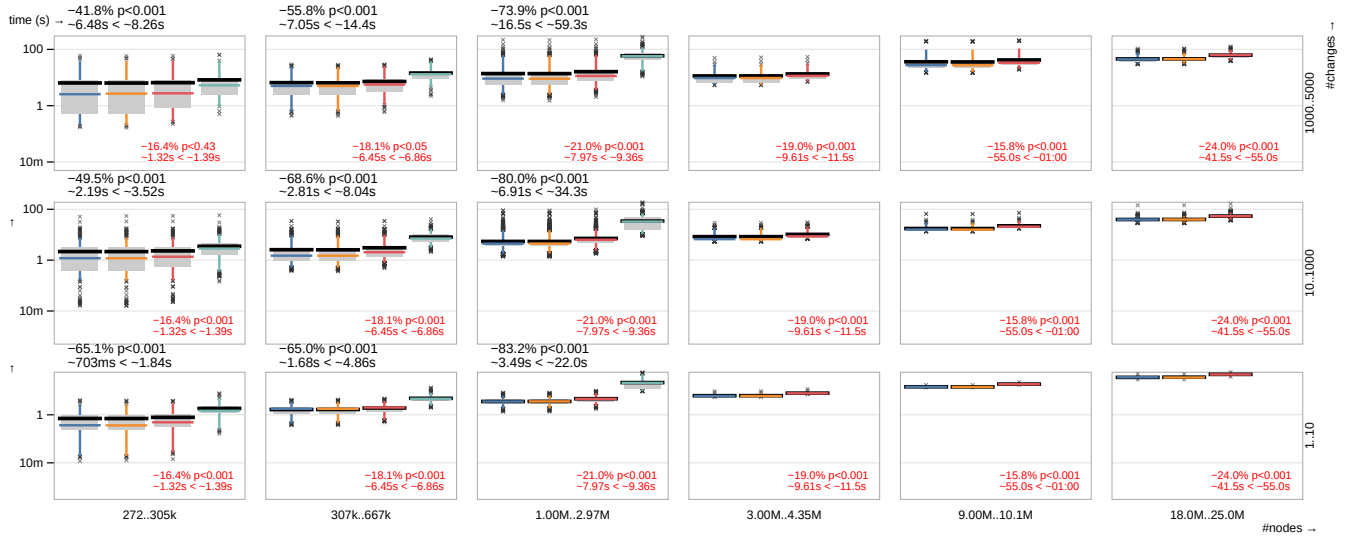

Figure 2: Comparing overall diff time;
Facet Legend 1 in black: [relative gain in %, *p*-value (Mann-Whitney); avg. for our approach, avg. for GumTree]
Facet Legend 2 in red: [relative gain in %, *p*-value (Mann-Whitney); avg. for our lazy approach, avg. for our non-lazy variant]
Color Legend: [lazy: blue, partial lazy: orange, not lazy: red, GumTree: green]

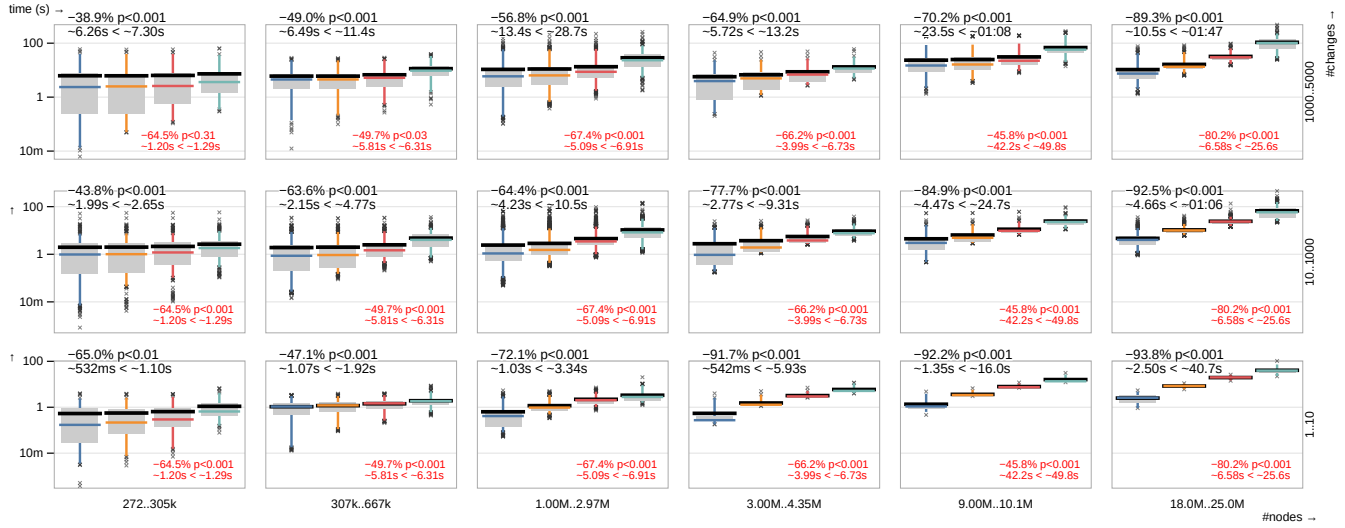

Figure 3: Comparing the top-down and bottom-up phases time;
Facet Legend: [relative gain in %, *p*-value (Mann-Whitney) ; avg. for our approach, avg. for GumTree]
Facet Legend 2 in red: [relative gain in %, *p*-value (Mann-Whitney); avg. for our lazy approach, avg. for our non-lazy variant]
Color Legend: [lazy: blue, partial lazy: orange, not lazy: red, GumTree: green]

Figures 2 to 4 shows groups of four box plots corresponding to the four objects we compare (see Section 4.3), respectively: *Lazy* (our approach, in blue), *Partial lazy*, *Not lazy* (in red), and *GumTree* (in green). We faceted (*i.e.,* grouped) the figure in both axis, due to the correlation of computation time both with: 1) the size of the output (*i.e.,* the diff) horizontally with three groups (rows): 1 to 10, 10 to 1000, and 1000 to 5000 changes. The number of changes is the size of the diff in terms of number of modifications needed to transform one AST version to the other. 2) the size of each version vertically in terms of number of nodes, from hundred thousands to millions in six groups (columns). On top of each facet (*i.e.,* groups of box plots) in black, we put the relative gain in %, the *p*-value, and the average time for our approach and GumTree. In red, we display the same information for our lazy and the non-lazy variant.

Figure 2 presents the total time taken to compute the diff including the top-down and bottom-up mapping phases with the diff generation. We can first observe that our approach outperforms the original GumTree each time, gaining between 42 % (avg. 6.48 s vs 8.26 s) and 83 % (avg. 4.49 s vs 22 s) of reduced overall computation time. The gains are more important with a lower number of

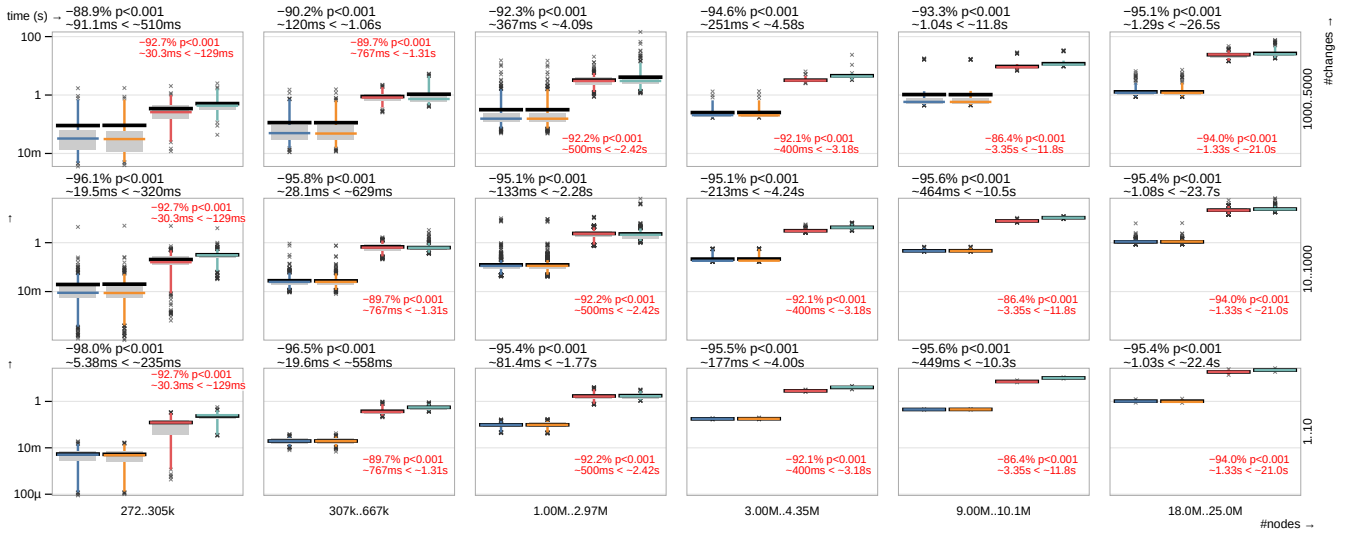

Figure 4: Comparing top-down phase time;
Facet Legend: [relative gain in %, p-value (Mann-Whitney) ; avg. for our approach, avg. for GumTree]
Facet Legend 2 in red: [relative gain in %, p-value (Mann-Whitney); avg. for our lazy approach, avg. for our non-lazy variant]
Color Legend: [lazy: blue, partial lazy: orange, not lazy: red, GumTree: green]

changes relative to the size of the codebase. In particular, between 1 and 10 changes our gains increase up to 83 % of execution time compared to the baseline. Then, between 10 and 1000 changes, our gains increase up to 80 %. Finally, for more than 1000 changes our gains improve up to 74 %. We observe that our approach with the lazy implementation also outperforms the non-lazy variant all the time, regardless of the size of the code and the size of the diff. The gain varies from 16 % (avg. 55 s vs 60 s) to 24 % (41.5 s vs 55 s).

Moreover, for the four largest projects with more than 300KLOC and 3M nodes, we disabled the generation of the diff for the GumTree implementation. Indeed, we observed extreme slowdowns of GumTree when enabled, leading almost every time to out of memory errors. As we did not attempt to lazify the generation of the diff, we further measured the time performance without the generation, *i.e.,* the top-down and bottom-up phases only and the time performance of the top-down phase only.

Figure 3 presents the time taken to compute the mappings during the top-down and bottom-up matching phases. Herein, our approach outperforms the original GumTree top-down and bottom-up phases each time, gaining between 39 % (avg. 6.26 s vs 7.30 s) and 94 % (avg. 2.50 s vs 40.7 s) of reduced overall computation time of the mappings. Between 1 and 10 changes, our gains increase from 47 % to 94 %. Then, between 10 and 1000 changes, our gains increase from 44 % to 93 %. Finally, for more than 1000 changes our gains improve from 39 % to 90 %. Similarly, our lazy variant outperforms the non-lazy variant all the time. The gain varies from 46 % to 80 %.

Figure 4 presents the time taken to compute only the mappings with the top-down matching phase. Herein, our lazy top-down phase outperforms the original GumTree top-down each time, gaining between 89 % (avg. 91.1 ms vs 510 ms) and 98 % (avg. 5.38 ms vs 235 ms) of reduced overall computation time of the top-down mappings. Between 1 and 10 changes our gains increase from 95 % to 98 %. Between 10 and 1000 changes, they increase from 95 % to 96 %. Finally, for more than 1000 changes they increase from 88 % to 95 %. Similarly, our lazy variant largely outperforms the non-lazy variant all the time. The gain varies from 87 % to 94 %.

These three figures highlight our lazy approach significantly gains during the top-down phase (in percentage) and the bottom-up phase (in absolute time). While our diff generation is faster than the GumTree one and scales up to thousands of actions on large software projects (*e.g.,* Hadoop), further gain could be achieved. Indeed, we did not lazify the diff generation, meaning that a full decompression of the trees is mandatory before generating the diff. Lazifying the generation is future work. It is worth noting that for the results of the lazy variant, a full decompression is done before computing the diff, taking 26.6 % of the generation time while only 12.0 % for the non-lazy variant. Furthermore, in Figure 3 we observe that the bigger the projects, the better our lazy approach performs compared to the two variants *Partial lazy* and *Not lazy*. In Figure 4, we also observe the same gain compared to the *Not lazy* variant.

> ***RQ*$_3$ insights:** Results show systematic and significant gains of 83 % on average and up to 99 % of our lazy approach compared to GumTree in all phases, from the top-down and bottom-up matching phases to the generation of the diff. Representing an order-of-magnitude difference in total time: 1) from ×1.2 to ×12.7 for diff computation, 2) from ×1 to ×226 for the top-down and bottom-up phases, and 3) from ×3.2 to ×233 for the top-down phase.

4.4.4 RQ4. To answer this RQ, we measured the execution time for parsing and diffing 100 commits for each project. Hence, we can compare the overall performance in a practical use case similarly as a developer would go through to compute diffs on given number of commits. To do so, we use the *GumTree-Spoon* version that parses a commit and then calls the *GumTree* diffing algorithm.

Figure 5 depicts the time of our approach including the construction of the *HyperAST* in orange versus the computation time for *GumTree-Spoon* including the parsing of the commits. We observe that our approach outperforms *GumTree-Spoon* on all the projects. For our largest projects *Hadoop* and *Flink*, *GumTree-Spoon* crashed in most of the time during the generation of the diffs, due to using more than 32 GB of memory heap. We also had other cases of crashes in *Jenkins* and in *Gson*. Those cases are in red rather than orange in our results. In *SkyWalker*, *GumTree-Spoon* could not parse completely most of the commits due to an issue with the multi-modules of Java. *GumTree-Spoon* could only parse some modules, sometimes one module only. This explains why in many commits it was faster than our approach that did diff all modules. Overall, when computing diff successfully, our approach was on 14.52 times faster than *GumTree-Spoon* in half of the cases (median). We had extreme cases of improvement in the six last small projects (from *Jackson* to *slf4j*) where we were thousands and hundred thousands times faster than *GumTree-Spoon*. Excluding these projects, we reach an average of 13.68 times where we are faster.

We also investigated the case of diffing only two commits by looking at the first two commits in our projects, since we must construct the *HyperAST* entirely in the first commit and incrementally update it in the second commit, before computing the diff. We see that we could compute the diff faster than *GumTree*, as shown by *t[0]* in Figure 5 (note that it is in two formats min:sec or 0.millisec). For example, in maximum in *Hadoop* and *Flink*, our approach took, respectively 1 min 24 s and 1 min 10 s while *GumTree* took 21 min 57 s and 24 min 3 s. In minimum in *Slf4j*, we took 300 ms while *GumTree* took 36 s. In other medium-sized projects as *netty*, *dubbo*, *log4j*, *jenkins*, *javaparser*, *spoon*, *maven* and *spark*, we respectively took from 3 s to 27 s, while *GumTree* took from 19 s to 3 min 35 s. On the smallest four projects, our approach varied from 300 ms (0.3 s) to 675 ms, whereas *GumTree-Spoon* varied from 9 s to 36 s. Therefore, even on two commits where we must build the *HyperAST* from scratch, a developer benefits of our lazified approach based to diff two commits or more. We could reach a gain up to 99 % and an order-of-magnitude of ×122 when considering the two first commits only.

> ***RQ*$_4$ insights:** Our lazy approach outperforms *GumTree-Spoon*. We are faster by 14.52 times in half of the cases (median) and when excluding extreme cases of gains, we are faster on average by 13.86 times. We also outperform *GumTree-Spoon* on the basic use case of diffing two commits only, with a gain up to 99 % and an order-of-magnitude of ×122.

4.5 Threats to validity

This sections discusses threats to validity w.r.t. [34].

Internal Validity. Considering the computation of diffs, we first had to evaluate its ability to produce identical outputs (mappings and diffs) as GumTree. To make sure we have unbiased measurements, we used the *HyperAST* to construct the code history and we retrieved the trees of each commit from the *HyperAST*. Thus, we had a uniform representation of the node elements (*i.e.,* same parser and same grammar for the ASTs) for comparing our approach and GumTree. Moreover, our implementation and the *HyperAST* are implemented in Rust while GumTree is developed in Java. To mitigate this difference of language while comparing execution time and memory usage, we provide and compared our approach with two other objects developed in Rust using the *HyperAST*: *Partial lazy* and *Not lazy*. *Not lazy* is the closest Rust version of GumTree while still benefiting of the *HyperAST*. Compared to these two variant objects, our evaluation still shows significant benefits for our approach with all lazy phases.

External Validity. The evaluation implied 19 projects. We carefully follow a clear protocol to select relevant and significant Java projects. Our curated list of projects represents real-world complex software systems with very large histories.

We implemented our approach on top of the *HyperAST* by lazifying the GumTree algorithms. We then and evaluated our approach for Java with Maven build system. Our conclusions in theory could generalize to other programming languages with similar features as Java (*e.g.,* strong static nominal typing). Nonetheless, further experimentation remains necessary on other languages to generalize our results. We also cannot generalize the diffing results to other diffing algorithms, such as [12]. Note that as the *HyperAST* supports only Java so far, we could only evaluated and compared to GumTree on Java projects. However, the goal of this paper was not to support multiple languages but to show the scalability of computing diffs on large code history. Besides, GumTree performance are not language dependent [11].

Construct Validity. Our evaluation shows that our approach scales the computation of diffs on large code history and large projects representing real-world complex software with very large histories. Compared to GumTree, we outperformed it by an order-of-magnitude difference in CPU time from ×1.2 to ×12.7 for the total time of diff computation and up to ×226 in intermediate phases of the diff computation, and an order-of-magnitude difference in memory footprint of ×4.5 per AST node. Further evaluation remains necessary for more insights and statistical evidence.

5 RELATED WORK

This section focuses on approaches computing diffs of source code.

A first category of approaches compute the diffs on the textual format of the code. Asaduzzaman *et al.* [2], Canfora *et al.* [5] and Reiss *et al.* [31] proposed a language-independent techniques for diffing. However, as they work on the textual representation of the code, they are unable to compute fine-grained changes, such as a change in a parameter or a condition guard. This thus hinders any automatic code analysis based on them.

A second category of approaches compute the diffs on the structured tree representation of the code. Pawlik *et al.* [30] compute the diff but without the move action. Chawathe *et al.* [6] are the first to compute a diff including move actions, thus, shortening the diff. Duley *et al.* [10] generate a diff for Verilog HDL files. Hashimoto *et al.* [16] are able to work on raw ASTs by producing a diff. Nguyen *et al.* [29] also propose to compute a diff but they focus more on finding clones rather than a complete diff.

Fluri *et al.* [12] propose *ChangeDistiller*, a tool inspired by [6] for computing diffs. Falleri *et al.* [11] also proposed *GumTree*, a tool inspired by [6] and also inspired from the algorithm of Cobena *et al.* [7] for compression purposes. Matsumoto *et al.* [27] propose an

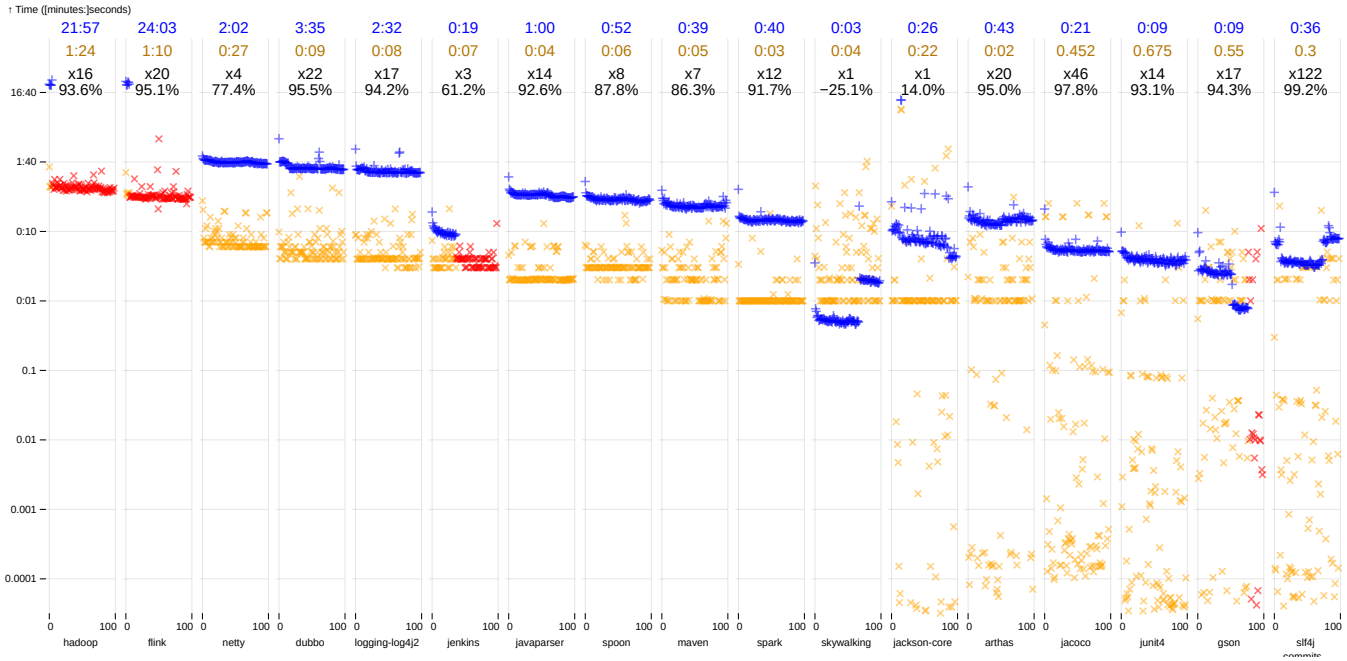

Figure 5: Comparing overall execution time (parsing + diff) on 100 commits;
Facet Legend: [*t[0]* time to compute first diff between first and second commit, lazy, GumTree, format min:sec or 0.millisec]
Color Legend: [lazy: orange times, lazy while Gumtree-Spoon failed: red times, GumTree-Spoon: blue plus]

extension of GumTree by incorporating information of line differences in addition to the AST to propose a diff easier to understand for developers. Higo *et al.* [19] also propose to integrate a new action of copy-and-paste to make the diff easier to understand too.

All these existing approaches focus on efficiently computing a diff between two files. Our approach focuses on a different yet complementary concern: diffing commits picked from a source code history (Git). Since a Git commit can contain numerous files, diffing commits directly faces the scalability issue we overcome in this paper. Other approaches aim at improving the diffs that GumTree produces [9, 13, 20]. This is out of our scope, however, they can still process our diff output similarly as they would do with GumTree diffs. Tsantalis *et al.* [32] propose an approach for detecting refactorings. Refactorings are high level changes that are not the goal of this paper and of the approaches of computing diffs. Moreover, RefactoringMiner mostly focuses on Java, while some other recent extensions try to make it work on Python and Kotlin, it seems like a complex process.

To the best of our knowledge, our approach is the unique one targeting scaling up the computation of diffs to thousands of commits and in large software projects.

6 CONCLUSION

This paper proposes a novel code differencing approach that scales on large code histories. We leverage on the *HyperAST* novel representation of code histories and lazify the *GumTree* algorithms to scale on large histories. The evaluation shows that our approach outperforms the mainstream code diffing approach. In particular, we observed an order-of-magnitude difference in CPU time from ×1.2 to ×12.7 for the total time of diff computation and up to ×226 in intermediate phases of the diff computation. We also observed an order-of-magnitude difference in memory footprint of ×4.5 per AST node. Finally, we gain all the time while having 99.3 % of identical diffs with respect to *GumTree* and 99.999 75 % of identical mappings in the remaining 0.7 % diffs. When including the parsing cost along with the diff, we still outperform *GumTree-Spoon*. We are faster by 14.52 times in half of the cases (median) and when excluding extreme cases of gains, we are faster on average by 13.68 times.

In the future, we would like to further apply the lazification to the diff generator (*i.e.*, the Chawathe algorithm). This may imply functional changes in the produced diffs and on how it should be reapplied, thus possibly exposing structurally independent changes.

Our approach also opens new research opportunities to build temporal code analyses on top of our scalable differing approach. For example, it should be easier to compute more sophisticated metrics on subtrees to be then used in the top-down phase of the *GumTree* algorithm. Second, considering the first part of the top-down phase of *GumTree* it should also be possible to devise levels of cloning/extraction detection. Third, for the industry of software forges, given the efficiency of the *HyperAST* at persisting versions as a fine-grained DAG, it should be possible to efficiently cache mappings and diffs. This would permit software forges to serve related services at very low amortized latency.

Finally, multi-mappings are currently intermediate and internal data not exported as output in the diffs. Our approach may export such data to better track cloned, duplicated, or merged code nodes.

ACKNOWLEDGMENT

The research leading to these results has received funding from the *ANR* agency under grant *ANR JCJC MC-EVO² 204687*.

REFERENCES

[1] Carol V Alexandru, Sebastiano Panichella, Sebastian Proksch, and Harald C Gall. 2019. Redundancy-free analysis of multi-revision software artifacts. *Empirical Software Engineering* 24, 1 (2019), 332–380.

[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 *2013 IEEE International Conference on Software Maintenance*. IEEE, 230–239.

[3] Thazin Win Win Aung, Huan Huo, and Yulei Sui. 2020. A literature review of automatic traceability links recovery for software change impact analysis. In *Proceedings of the 28th International Conference on Program Comprehension*. IEEE/ACM, 14–24.

[4] Gabriele Bavota, Luigi Colangelo, Andrea De Lucia, Sabato Fusco, Rocco Oliveto, and Annibale Panichella. 2012. TraceME: traceability management in eclipse. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 642–645.

[5] Gerardo Canfora, Luigi Cerulo, and Massimiliano Di Penta. 2008. Tracking your changes: A language-independent approach. *IEEE software* 26, 1 (2008), 50–57.

[6] Sudarshan S Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. 1996. Change detection in hierarchically structured information. *Acm Sigmod Record* 25, 2 (1996), 493–504.

[7] Gregory Cobena, Serge Abiteboul, and Amelie Marian. 2002. Detecting changes in XML documents. In *Proceedings 18th International Conference on Data Engineering*. IEEE, 41–52.

[8] Rafael de Mello, Roberto Oliveira, Anderson Uchôa, Willian Oizumi, Alessandro Garcia, Baldoino Fonseca, and Fernanda de Mello. 2022. Recommendations for Developers Identifying Code Smells. *IEEE Software* 40, 2 (2022), 90–98.

[9] 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*. IEEE/ACM, 660–671.

[10] Adam Duley, Chris Spandikow, and Miryung Kim. 2012. Vdiff: a program differencing algorithm for Verilog hardware description language. *Automated Software Engineering* 19, 4 (2012), 459–490.

[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*. ACM/IEEE, 313–324.

[12] Beat Fluri, Michael Wursch, Martin PInzger, and Harald Gall. 2007. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on software engineering* 33, 11 (2007), 725–743.

[13] 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, 264–274.

[14] Daniel M German, Bram Adams, and Kate Stewart. 2019. cregit: Token-level blame information in git version control repositories. *Empirical Software Engineering* 24 (2019), 2725–2763.

[15] Felix Grund, Shaiful Alam Chowdhury, Nick C Bradley, Braxton Hall, and Reid Holmes. 2021. CodeShovel: Constructing method-level source code histories. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1510–1522.

[16] Masatomo Hashimoto and Akira Mori. 2008. Diff/TS: A tool for fine-grained structural change analysis. In *2008 15th working conference on reverse engineering*. IEEE, 279–288.

[17] Hideaki Hata, Osamu Mizuno, and Tohru Kikuno. 2011. Historage: fine-grained version control system for java. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution*. 96–100.

[18] Yoshiki Higo, Shinpei Hayashi, and Shinji Kusumoto. 2020. On tracking Java methods with Git mechanisms. *Journal of Systems and Software* 165 (2020), 110571.

[19] Yoshiki Higo, Akio Ohtani, and Shinji Kusumoto. 2017. Generating simpler ast edit scripts by considering copy-and-paste. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 532–542.

[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*. IEEE/ACM, 679–690.

[21] Emanuele Iannone, Roberta Guadagni, Filomena Ferrucci, Andrea De Lucia, and Fabio Palomba. 2022. The secret life of software vulnerabilities: A large-scale empirical study. *IEEE Transactions on Software Engineering* 49, 1 (2022), 44–63.

[22] Mehran Jodavi and Nikolaos Tsantalis. 2022. Accurate method and variable tracking in commit history. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 183–195.

[23] 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, 3 (2020), 1980–2024.

[24] Quentin Le Dilavrec, Djamel Eddine Khelladi, Arnaud Blouin, and Jean-Marc Jézéquel. 2021. Untangling Spaghetti of Evolutions in Software Histories to Identify Code and Test Co-evolutions. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 206–216.

[25] Quentin Le Dilavrec, Djamel Eddine Khelladi, Arnaud Blouin, and Jean-Marc Jézéquel. 2022. HyperAST: Enabling Efficient Analysis of Software Histories at Scale. In *37th IEEE/ACM International Conference on Automated Software Engineering*. IEEE/ACM, 1–12.

[26] Stanislav Levin and Amiram Yehudai. 2017. The co-evolution of test maintenance and code maintenance through the lens of fine-grained semantic changes. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 35–46.

[27] 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, 550–554.

[28] Davood Mazinanian, Nikolaos Tsantalis, Raphael Stein, and Zackary Valenta. 2016. JDeodorant: clone refactoring. In *Proceedings of the 38th international conference on software engineering companion*. IEEE/ACM, 613–616.

[29] Hoan Anh Nguyen, Tung Thanh Nguyen, Nam H Pham, Jafar Al-Kofahi, and Tien N Nguyen. 2011. Clone management for evolving software. *IEEE transactions on software engineering* 38, 5 (2011), 1008–1026.

[30] 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.

[31] Steven P Reiss. 2008. Tracking source locations. In *Proceedings of the 30th international conference on Software engineering*. IEEE, 11–20.

[32] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. 2020. RefactoringMiner 2.0. *IEEE Transactions on Software Engineering* 1, 1 (2020), 1.

[33] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. 2015. When and why your code starts to smell bad. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 403–414.

[34] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering*. Springer Science & Business Media.

[35] Kaizhong Zhang and Dennis Shasha. 1989. Simple fast algorithms for the editing distance between trees and related problems. *SIAM journal on computing* 18, 6 (1989), 1245–1262.