Scalable Structural Code Diffs

Comparing Gumtree Greedy and Gumtree Simple adapted for scaling

Ruben van Seventer
Supervisor(s): Carolin Brandt, Quentin Le Dilavrec
EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 22, 2025

Name of the student: Ruben van Seventer
Final project course: CSE3000 Research Project
Thesis committee: Carolin Brandt, Quentin Le Dilavrec, Jesper Cockx

Abstract

As software evolves, understanding the differences between versions of code becomes more important. While text-based differencing is practical and widespread, it does not capture the structure of code. AST-based differencing solves this by using the structure of the code. Gumtree is a well known reference implementation of multiple structural diff heuristics. Gumtree Greedy is the original algorithm, while Gumtree Simple is a later version that was designed to scale better by making stronger assumptions.

In this paper, we compare ported versions of Gumtree Greedy, Gumtree Simple, and their lazified variants. They were implemented in the Rust-based HyperAST framework and tested on large-scale Java datasets. Our results show that Gumtree Simple uses significantly fewer CPU cycles compared to Gumtree Greedy. Due to suspected bugs in the implementation, we cannot yet conclusively measure the benefits of lazification. However, our implementation experience suggests that Gumtree Simple is easier to adapt and optimize for scalability.

1 Introduction

Whether new features are introduced, bug fixes are applied or refactoring takes place, software is constantly evolving. This makes understanding the differences between different versions of a program essential. These differences are often expressed in terms of actions required to transform one version of the program into the other, this sequence of actions is called the *edit script*. Traditionally, edit scripts are computed with text-based techniques. One advantage of using text-based techniques is that very efficient algorithms [2] exist that guarantee to generate a *minimal edit script*, i.e., the shortest possible sequence of actions. However, it also has downsides, since code is not a collection of unstructured text, but rather structured text that adheres to syntax rules. Text-based differencing doesn't capture these semantics. Moreover, it typically expresses differences using only **insert** and **delete** actions, whereas many code changes can be more naturally expressed when we include **update** and **move** actions. Edit scripts that include these actions are generally considered easier to interpret and tend to better reflect the essence of the changes [5, 4].

Abstract Syntax Tree (AST) based differencing captures the structure and semantics of code. One of the most popular tools to generate edit scripts based on ASTs is Gumtree [5]. As software projects grow, their ASTs become larger, making scalability increasingly important. The original version of Gumtree, known as *Gumtree Greedy*, was later followed by *Gumtree Simple* [4]. Gumtree Simple has stricter assumptions, enabling it to use a faster algorithm and to scale better. Recent developments, like the *HyperAST* [8], a data structure that aims to reduce redundancy by reusing overlapping parts of ASTs, and later the *Hyperdiff* [9] approach, which proposes new methods to compute differences across multiple versions more efficiently, raise new questions about how differencing algorithms can be optimized further.

This paper investigates whether Gumtree Simple enables additional adaptations helping with scalability compared to Gumtree Greedy. To explore this question we investigated the empirical performance of these algorithms, using a Rust-based implementation built on the HyperAST project. The sub-questions we have identified to help answer the main research question are:

SRQ1. *Which characteristics does Gumtree Simple have that enable easier or more effective scalability adaptations than Gumtree Greedy?*

In this sub-question it will be investigated in what ways Gumtree Simple is easier to adapt, or can be more effectively adapted to improve scaling. This will be answered based on implementation experience and the baseline benchmarks.

SRQ2. *How do Gumtree Greedy, Gumtree Simple and adaptations of Gumtree Simple compare at scale?*

In this sub-question the different versions of Gumtree will be empirically compared based on the number of CPU-cycles, and runtime.

The contribution of this paper is giving a comparative performance evaluation of Gumtree Greedy, Gumtree Simple and adaptations of Gumtree Simple. In addition to confirming and complementing earlier findings [4], we ported the original Gumtree Simple algorithm to the Rust based HyperAST codebase.

The rest of the paper is structured as follows. Section 2 motivates the use of AST-based differencing and provides a high-level overview of the approach. In Section 3 related work is discussed. Section 4 outlines the methodology, and introduces the datasets. Section 5 lays out the the results. Section 6 discusses the results, and the threads to validity. In section 7 we evaluate our responsible research practices. Finally, section 8 concludes this paper.

2 Motivating Example and Background

In this section we will first go over a motivating example, this will explore a small example to motivate AST-based differencing. After which we will provide relevant background information.

(a) Original Code Snippet

(b) Code Snippet After Modification

Figure 1: An example of how AST-based differencing captures semantic changes. Orange highlights indicate updates, green highlights indicate insertions, and purple highlights indicate moves.

2.1 Motivating Example

A text-based differencing tool would interpret the modifications shown in Figure 1 as one line deletion, and two insertions (green). It will not capture the semantics of the changes: that the *print("hello")* statement was capitalized (an update, shown in orange), and that the *func()* call was moved (highlighted in purple) rather than deleted and re-added. In contrast, an AST-based differencing algorithm like Gumtree will produce a more meaningful edit script that better reflects the intentions of the developer (as can be seen in the example).

While AST-based differencing is more fine-grained, it becomes computationally expensive when comparing many versions, in for example a long commit histories. This is where the HyperAST, along with the proposed HyperDiff approach [9] offer significant advantages.

2.2 Background

Comparing two Abstract Syntax Trees (ASTs) to generate edit scripts addresses the downsides of text-based differencing. A widely used tool to do this is Gumtree[1]. ASTs represent code as tree structures where each node corresponds to a part of the code, and may carry some additional metadata. Each node has a type label (e.g. 'Body' or 'Call') and optionally a value (e.g. a variable name or literal). Additionally, nodes may store data such as the size and depth of their subtree. Falleri et al. proposed [5] and implemented the original version of Gumtree, *Gumtree Greedy*. The pipeline for Gumtree can be seen in Figure 2, where red arrows correspond to mappings found in that phase and the orange box indicate that nodes were mapped in a previous stage. Here we see that finding the mappings between the two AST's consists of three phases:

1. **Top-down phase**: a top-down traversal where it greedily matches the largest isomorphic subtrees (i.e. subtrees that have the same structure and labels) between the original and modified ASTs.
2. **Bottom-up phase**: matches found in the first phase are propagated upward to their parent nodes.
3. **Recovery phase**: runs every time a mapping is found in the bottom-up phase, and uses the Tree Edit Distance (TED) to determine if nodes should be mapped. It ensures that no extra mappings can be found in the descendants of mapped nodes.

The recovery phase is an important phase, in our example it was responsible for 6 out of the 10 mappings. Greedy uses a variant of the Tree Edit Distance (TED) algorithm [11], which computes the sequence of operations with a minimum-cost required to transform one tree into another. This algorithm has a third order polynomial complexity relative to the number of nodes in both trees and thus doesn't scale very well. In an effort to make Gumtree scale better *Gumtree Simple* [4] was proposed. It replaces the TED with a faster algorithm which has a second order polynomial complexity relative to the number of nodes in both trees.

Le Dilavrec et al. introduced the *HyperAST* [8], a data structure designed to reduce redundancy by reusing overlapping parts of ASTs. Instead of storing all the ASTs for different versions of a project separately, a Directed Acyclic Graph (DAG) is constructed that reuses duplicated nodes. The HyperAST is particularly effective for history-aware analysis, such as tracking the evolution of a code block over time. In the HyperAST part of Figure 3 (the rest of the figure will be discussed in Section 4.2) a simplified visualization of such a DAG is shown. The AST build from the "before" code snippet is shown in black. When the AST build from the "after" snippet is added, many nodes are reused, while the new or modified nodes (highlighted in green) are appended. This clearly shows the append only nature of the data structure, because of this fact many intermediate calculations, like hashes and other metadata can also be reused. This explains why the HyperAST is so effective for history aware analysis; code needs to be followed through multiple versions of a program, which means we need to look at a lot of ASTs. Furthermore, commits often have small in-

[1] https://github.com/GumTreeDiff/gumtree/

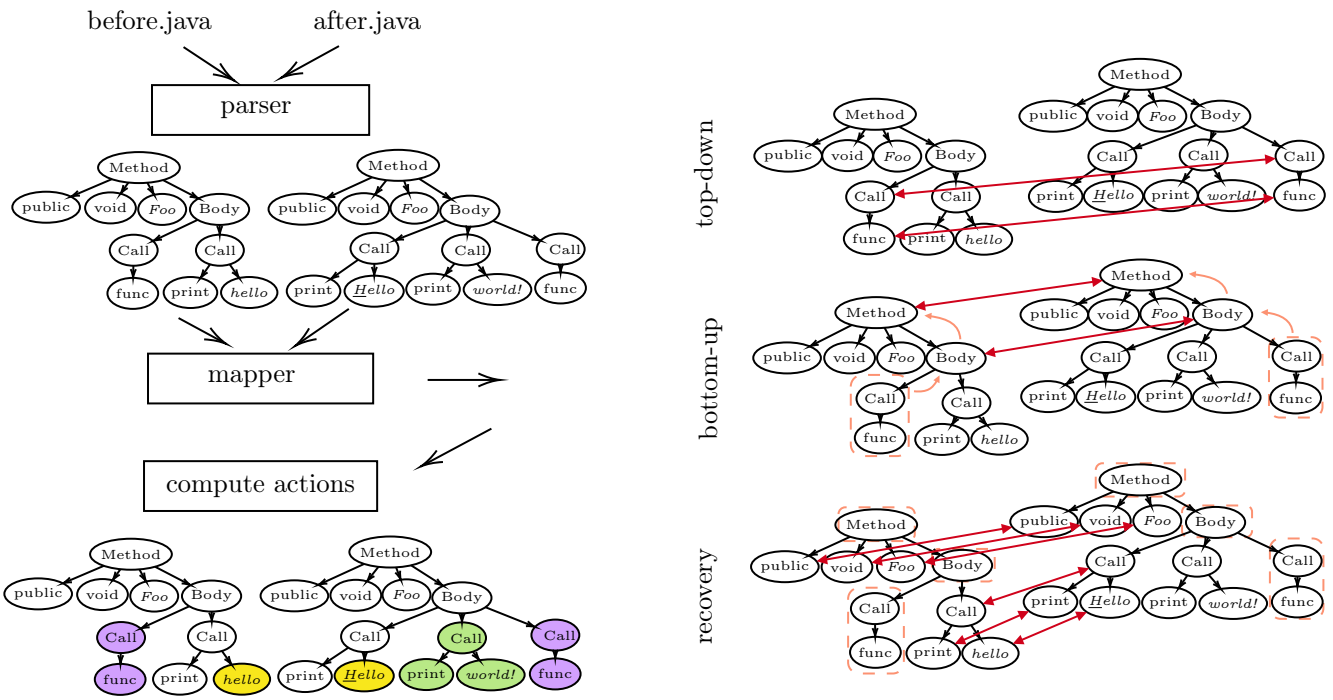

Figure 2: Workflow of the Gumtree algorithm on the code in Figure 1. The left shows the parsed ASTs and the resulting edit actions. The right shows the three mapping phases. Red arrows indicate mappings found in that phase, while previously matched nodes are in an orange box.

cremental changes, meaning that a lot of the AST stays unchanged. With the HyperAST all of these versions are represented in one graph, where unchanged parts are stored only once.

Both the lazified and non-lazified versions of *Gumtree Greedy* and *Gumtree Simple* have hyper-parameters that influence their matchings. The *Gumtree Greedy* implementations are configured with two parameters: "Size Threshold" and "Similarity Threshold". The "Size Threshold" defines the maximum size of a sub-tree that will be used in the (expensive) recovery phase. The "Similarity Threshold" dictates the minimum similarity required for two nodes to be considered a match. The Gumtree Simple implementations rely on a single hyper-parameter, the "Similarity Threshold", which serves the same purpose as in the *Gumtree Greedy* implementations. Martinez et al. [10] discussed how to optimize these parameters, but this is outside the scope of this paper.

3 Related Work

Gumtree [5] pioneered AST-based code differencing. It has been used as a base for more advanced tools like *RefactorMiner* [14, 13], a tool that can detect refactorings between two versions of a program. It was later improved by Alikhanifard et al. [1], who improved performance and mapping accuracy, among other things.

The original Gumtree algorithm, *Gumtree Greedy*, does not scale well for large trees. *Gumtree Simple* [4] was proposed as a faster alternative, it replaces the expensive recovery-phase with a more efficient strategy. In both algorithms the hyperparameter(s) have a big impact on the performance, Martinez et al. [10] studied how they can be optimized.

ChangeDistiller [6] is another AST-based differencing tool that focuses on Java and uses a custom set of heuristics. *ChangeDistiller* works on a coarse-grained AST where leaf nodes are code statements. Gumtree Greedy was compared to *ChangeDistiller* in its original paper, and was found to both produce more mappings and more concise edit scripts.

To improve scalability across large commit histories, Le Dilavrec et al. introduced the *HyperAST* [8], a DAG-based structure that stores multiple ASTs efficiently by reusing unchanged nodes. The *Gumtree Greedy* heuristic was optimized using the Hyper AST in HyperDiff [9].

Our work builds on these approaches by benchmarking lazy and non-lazy versions of both Gumtree Greedy and Gumtree Simple within the HyperAST framework. We focus on evaluating the trade-offs in runtime and accuracy across these combinations.

4 Methodology

In this section we discuss our methodology. We first explain the recovery phase of the Simple heuristic. Then we talk about lazification and how we ported the *Gumtree Simple* algorithm from the Java based Gumtree tool to the Rust based HyperAST repository. After which we introduce the datasets used in our benchmarks, and we end by explaining our evaluation protocol.

4.1 Recovery phase in Gumtree Simple

As shortly stated in Section 2.2, the recovery phase is where the Simple heuristic truly distinguishes it self from the Greedy heuristic. It brings the asymptotic complexity from the whole pipeline down from cubic to quadratic. It accomplishes this speedup by sacrificing general optimality, *it is still optimal but only under some assumptions* which often do not hold the real-world. The two main assumptions are: unique subtree labels, meaning no two subtrees in the same tree share the same label, and unambiguous matchings, meaning each node in one tree can be mapped to at most one node in the other. Switching to this faster recovery phase enables us to remove the "Size Threshold" that was introduced in the Greedy heuristic to limit the size of the trees explored in the recovery phase.

Falleri et al. [4] describe the recovery phase of the Simple heuristic as consisting of three sub-phases. Remember that the recovery phase gets called every time a mapping is found in the bottom-up phase, meaning we always have two 'parent' nodes that were just mapped.

1. **Exact Isomorphism**: The first phase searches the unmatched children of the nodes that were just mapped, for subtrees that are identical both structurally and label-wise. If these matched subtrees belong to the longest common subsequence and consist only of unmapped nodes, they are added to the mapping. This step is very similar to the top-down phase.

2. **Structural Isomorphism**: If no matches are found, we search for structural isomorphic subtrees instead. This is done by ignoring the labels of leaf nodes. Because subtrees where only the identifier (e.g. the visibility of a function) was changed now will also be considered isomorphic. This sub-phase proves to be especially useful to detect update actions.

3. **Type Matching**: As a last resort, type-based matching is attempted. This step is inspired by the XYDiff algorithm [3], and searches for node types that appear only once among the children of both recently matched nodes. If such unique type exist, then the corresponding nodes will be mapped. Unlike in earlier steps, this step can not guarantee isomorphism on these subtrees. Therefore, the recovery phase is recursively applied on these newly mapped nodes.

4.2 Lazification

In order to fairly benchmark and compare the different algorithms, we made sure they are all implemented in one common codebase. This was done by porting the original *Gumtree Simple* implementation from the Gumtree repository[2] to a fork of the HyperAST repository [3]. This port served as a baseline and does not include any HyperAST specific optimizations. To optimize the Simple heuristic we followed the *HyperDiff* [9] approach, and *lazified* the algorithm. This means that nodes are not decompressed until strictly necessary, avoiding necessary computation. For example, if both the "before" and "after" ASTs contain a subtree with the same hash, then the subtrees can be matched without fully decompressing them. Likewise, if two nodes differ significantly, we won't try to match them, thus their subtrees can stay compressed. Leaving subtrees compressed speeds up the mapping process, but when calculating the actions some of the still compressed subtrees will still be decompressed.

In Figure 3 the difference between the original and lazy version is visible. Although our toy example does not have big subtrees that will stay compressed the principle stays the same. In the figure the mappings found during the top-down phase are represented with solid red arrows, the mappings found during the bottom-up phase with dashed blue arrows, and the mappings found during the recovery phase are represented with dotted green arrows.

During implementation we were able to reuse large parts of code already present in the HyperAST repository for the Greedy and lazified Greedy heuristic. The top-down phase is the same for both Greedy and Simple, and thus we were able to reuse both the normal and lazy version of this. Furthermore, since the top-down phase uses isomorphism, we were able to reuse this logic for the Simple recovery phase.

Since optimization should not alter an algorithm, we ensure that the output of the lazified algorithm is the same as the output of the original version. Our full implementation, including both the ported Simple heuristic and the lazified version, is available on GitHub[4].

[2] See footnote 1

[3] `https://github.com/LeaderSupreme/HyperAST`

[4] See footnode 3

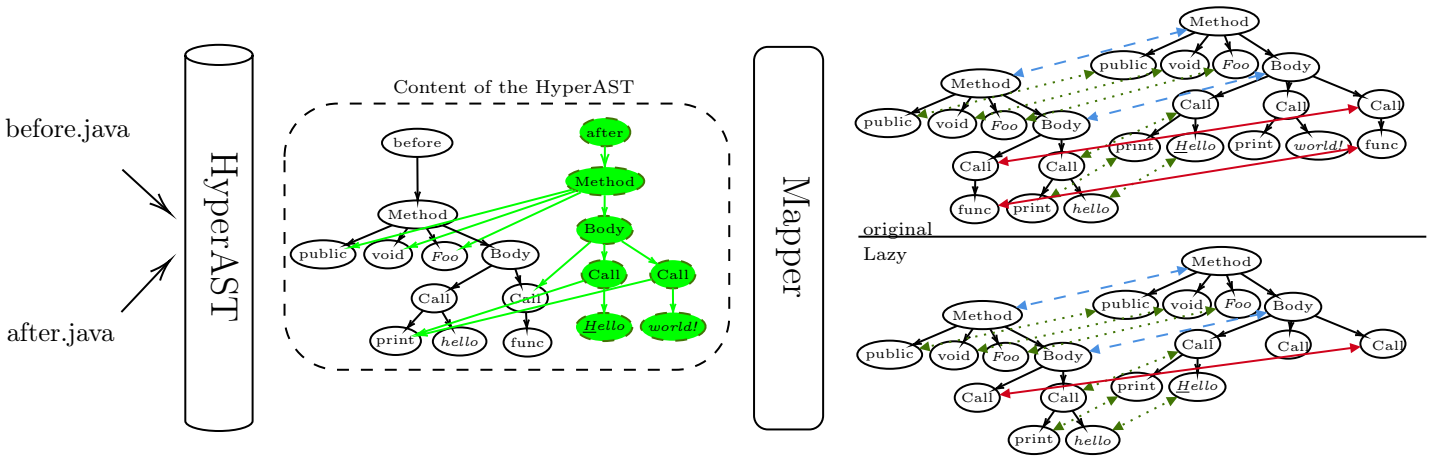

Figure 3: Comparing the workflow of original vs lazy algorithms.

4.3 Dataset

The dataset we used to compare the different heuristics and their implementations is publicly available on GitHub [5]. It consists of a collection of preprocessed diffs, and is divided into two subsets:

- **GitHub Java**: contains the changed files of 1000 commits from 10 popular, well maintained Java projects on GitHub.
- **Defects4J**: is derived from the Defects4J[6] [7, 12] repository and contains bug fixes from different big Java projects.

All the datasets are divided into two directories, `before` and `after`, in which we can find files before and after modification, respectively. Both these directories are organized by project and commit. Both datasets only contain files that were changed in a commit. This dataset is well-suited for our experiments for several reasons. First, it is a subset of the dataset that was used in prior work, including during the benchmarking of the Gumtree Simple [4] algorithm, making our results easily comparable. Second, we did not create this dataset ourselves, this reduces the risk of selection bias. Additionally, the clear structure makes it easy to work with.

4.4 Benchmarking tools

To assess the performance of the *Gumtree Simple* and *Lazified Gumtree Simple* implementations we measure their performance on our benchmark, and compare them to the results of the *Gumtree Greedy* and the *Lazy Gumtree Greedy* implementation already present in the HyperAST codebase. Two well known tools are used to construct the benchmark:

- **Criterion.rs**[7]: A statistics-driven benchmarking library for Rust. It provides detailed insights in runtimes, including confidence intervals and outliers.
- **Criterion-perf-events**[8]: A plug-in for Criterion that uses perfcnt[9] to measure low-level metrics like CPU-cycles.

4.5 Evaluation Protocol

The benchmark is designed to compare the performance of both the lazy and non-lazy versions of *Gumtree Greedy* and *Gumtree Simple*. For every project in the subdatasets, the top-down phase is executed twice, once using the lazy version and once using the non-lazy version. Then, the bottom-up phase is run with all four versions on every commit, thus we only measure phase two (bottom-up) and three (recovery) of the total process. Both the number of CPU-cycles and the runtime where benchmarked. The number of CPU-cycles reflect the computational resources needed to compute the mappings, while the runtime is used to get a more intuitive comparison. During all benchmarks the number of mappings are recorded. The number of mappings is often used as a proxy for the quality of the diff, if there are more mappings found typically a more fine-grained diff can be made. Criterion was set to take 15 samples per benchmark. For each version, it reported a 95% confidence interval around the mean CPU-cycles. We observed that the median runtime always fell within this CI.

[5] `https://github.com/GumTreeDiff/datasets`
[6] `https://github.com/rjust/defects4j`
[7] `https://github.com/bheisler/criterion.rs`
[8] `https://crates.io/crates/criterion-perf-events`
[9] `https://crates.io/crates/perfcnt`

For most hyper-parameters the default values were used. Specifically, the "Similarity Threshold" was set at 0.5 for both Greedy and Simple. We used two values for the "Size Threshold" parameter from Greedy, 1000 (the default suggested in the original paper) and 200. From now on when it is relevant to specify which Size threshold was used Greedy-1000 and Greedy-200 will be used to distinguish them. We have included Greedy-200 so we can compare the results of using the optimal recovery phase on only smaller subtrees.

5 Results

In this section the results from the benchmarks are presented. All benchmarks were run on the following hardware: *Intel© Core™ i5-8250U CPU @ 1.60GHz × 4; 16GiB RAM; 264 GB Hard Drive* running *Linux Mint 22.1 Cinnamon (Cinnamon version 6.4.8, and Linux Kernel version 6.8.0-59-generic).* We will first look at how the six variants (greedy-1000, greedy-200, simple, lazy greedy-1000, lazy greedy-200 and lazy simple) compare at scale. We do this based on the empirical measurements of the CPU cycles, and the number of mappings each version found. The raw data for both the number of mappings and the CPU-cycles can be found in C, Table 1 can be used to get a feeling for the results. Below the results will be discussed further.

Table 1: Comparison of total mappings found and average CPU-cycles spent relative to the (non-lazy) Greedy heuristic, using the default 1000 as the max size threshold.

		GitHub Java	Defects4J
Total mappings	Greedy-200	-0.472	-0.183
	Simple	-0.317	0.772
	Lazy Greedy-200	-0.470	-0.183
	Lazy Simple	-0.517	0.726
CPU-cycles	Greedy-200	-95.633	-96.828
	Simple	-99.900	-99.732
	Lazy Greedy-1000	0.282	3.348
	Lazy Greedy-200	-95.526	-96.682
	Lazy Simple	-99.779	-99.510

5.1 Mappings In the upper part of Table 1 the relative change of total mappings compared to Greedy-1000 can be observed, lazy Greedy-1000 has been left out as it produced the same number of mappings as the non-lazy version. We see that all four versions shown in the table find fewer mappings than Gumtree-1000 when run on the GitHub Java dataset. On the Defects4J dataset Greedy-200 and its lazy variant also find fewer mappings than Greedy-1000, but Simple and its lazy counterpart find more mappings than Greedy-1000. That Greedy-200 never finds more mappings than Greedy-1000 is expected, it searches for mappings with the same algorithm but performs the recovery phase on a subset of subtrees that Greedy-1000 does. Why Simple performed worse on the GitHub Java dataset, but better on the Defects4J dataset is less obvious. A likely explanation is that the GitHub Java contains arbitrary changes, where the Defects4J only contains small localized bug patches. We see however that the differences are small, none of the relative differences are more than one percent off from the number of mappings Greedy-1000 found. Something that is interesting to notice is that the number of mappings of Greedy-1000 and lazy Greedy-1000 stays the same, this is expected as explained in Section 4.2, but the number of mappings found by both Greedy-200 and Simple change when we apply the lazified version of that algorithm. This is something that will be addressed in Section 6.2.1.

5.2 CPU-cycles In the lower part of Table 1 the relative change of CPU-cycles spent compared to Greedy-1000 can be observed. Among the non-lazy variants we see that both Greedy-200 and Simple use significantly fewer CPU-cycles then Greedy-1000. This trend continues when we move to the lazified variants, lazy Greedy-200 and lazy Simple use significantly less CPU-cycles compared to lazy Greedy-1000. However, when we compare the lazy and non-lazy variants an surprising result emerges. All the lazy variants used on average more CPU-cycles than their lazy counterpart.

5.3 Runtime In Figure 4 the runtime in milliseconds is plotted against the absolute change in the number of nodes of a project. Both axis are in log-scale. The plot is missing the line of Greedy-1000, but reflects what we expected to see from Table 1. The results of the lazy variants are close to their normal variants, the trend lazy Greedy-200 of is almost on top of the one from Greedy-200.

6 Discussion

In this section we will reflect on the results we have found earlier and revisit the sub-questions posted in the introduction. After which we will address the threads to validity.

6.1 Research Questions

6.1.1 How do Gumtree Greedy, Gumtree Simple and adaptations of Gumtree Simple compare at scale? As seen in section 5, there are significant differences between the different variants. Across the datasets we saw that Simple uses approximately 99% fewer CPU-cycles than Greedy-1000. This trend continued when we compare

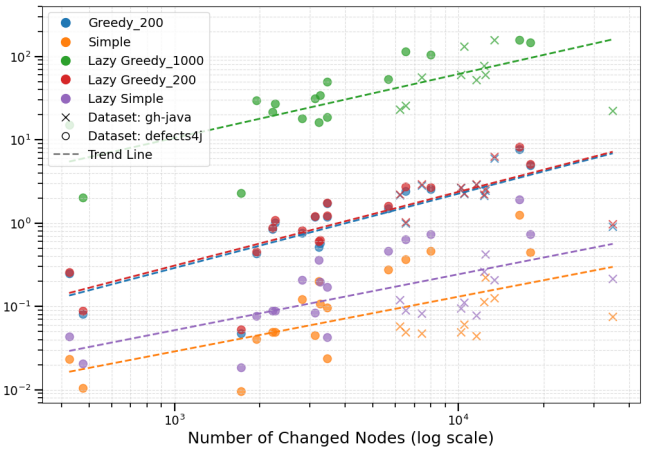

Figure 4: Comparison of median runtime (ms) and number of new nodes (absolute difference between nodes in the buggy files and fixed files).

Lazy Simple with Lazy Greedy, where we see that Lazy Simple also uses 99% fewer CPU-cycles. The same results are reflected in the runtime of the algorithms. We did not get exactly the results we expected, that Simple is so much faster than Greedy aligns with the results found by Falleri et al. [4]. However, that the lazy variants are often slower than their non-lazy counterparts is in stark contrast with the results found by Le Dilavrec et al.[9]. Nevertheless, we in the results that the "Max Size" threshold has a big impact on the CPU-cycles for the Greedy variant. This means that when choosing a heuristic there is a trade-off. Do you want the strong optimality guarantees Greedy offers, and if so, where do you put the "Max size" threshold?

6.1.2 Which characteristics does Gumtree Simple have that enable easier or more effective scalability adaptations than Gumtree Greedy? The difference between Greedy and Simple lies in the recovery phase, here lies also the reason Simple can be easier adapted and optimized. In the recovery phase Greedy uses a complex generally optimal tree differencing algorithm, which while giving a lot of guarantees is also computationally expensive and hard to reason about. In contrast, Simple uses a recovery phase with three clear sub-phases. This structure makes it easy to reason about, and it can be optimized in parts.

6.2 Threats to Validity

6.2.1 Internal Validity We observed that the number of mappings produced by Greedy-200 and Simple are not always equal to their lazy counterpart. In theory these should be the same, as stated earlier in Section 4.2. Although the differences were often small, it suggests that there is a bug in the code. Furthermore, the results of lazy Greedy and Greedy don't resemble the results Le Dilavrec et al. [9] found, this suggests that there are shortcomings in the benchmarking setup. Due to time constraints we were not able to figure out what the cause was of these discrepancies, and thus it is left as future work.

All benchmarks were executed using the Criterion.rs benchmarking framework. Although benchmarks were run on a single machine, and we allowed Criterion to collect sufficient samples to reduce noise, but background processes or noise may still have caused variability in timing measurements.

6.2.2 External Validity Our use of an established dataset that has been used in the evaluation of the: Gumtree Simple [4], HyperAST [8], and HyperDiff [9] algorithms supports our external validity. It improves the comparability of our findings with prior work. However, the dataset only includes Java and Python code and focuses on specific types of commits and repositories. Therefore, our results may not generalize to other programming languages and codebases.

Additionally, our experiments were performed on selected commits and project samples. Although they are representative of a real-world development history, further experiments on larger datasets could make our conclusion more generalizable.

7 Responsible Research

The context of our work is tree differencing. We did not interact with sensitive data and our work do not have direct criminal or harmful applications. However, there are still a lot of things to consider. We find *Reproducibility* and *Transparency* very important in research. Our benchmarks were designed to be reproducible: we used publicly available tools, and both our datasets and code are publicly accessible. Additionally, we clearly documented the benchmarking setup and methodology to ensure others can reproduce or build up on our results. While our research investigated performance gains and adaptability, it was closely correlated with *Sustainable computing practices*. By leveraging the HyperAST and lazifying the algorithm, we reduce unnecessary computations and memory overhead. This significantly reduces computational resources when applied at scale. The dataset used in this research consists of publicly available open-source code and complies with the respective licenses of the included projects.

8 Conclusions and Future Work

In this paper, we compared the performance and adaptability of two AST differencing algorithms: Gumtree

Greedy and Gumtree Simple. We extended the HyperAST repository with lazified variants of both algorithms and evaluated their performance on large-scale Java datasets.

Our results show that Gumtree Simple uses significantly less CPU-cycles than Gumtree Greedy, making it better suited for calculating diffs on large code bases. However, we encountered an unexpected result when comparing the lazy variants to their non-lazy variants, where lazy variants would use more CPU-cycles than their non-lazy counterparts. Nevertheless, our implementation experience suggests that Gumtree Simple has a structure that makes it easier to adapt for scaling.

For future work, the implementation of both the standard and lazy versions of Simple should be checked for correctness. Once validated, the benchmark can be run again for more reliable results, and possibly reproducing the results of the HyperDiff paper [9]. Researchers familiar with the HyperAST codebase, and experienced in optimizing rust code, could look to further optimize the lazy Simple implementation.

Acknowledgments

I would like to thank my peers Elias Hoste, Leo Mangold, Maciej Mejer, and Alexander Nitters for their insightful discussions and helpful feedback throughout the project. I also want to thank my supervisor Quentin Le Dilavrec and responsible professor Carolin Brandt for their guidance and feedback.

References

[1] Pouria Alikhanifard and Nikolaos Tsantalis. A Novel Refactoring and Semantic Aware Abstract Syntax Tree Differencing Tool and a Benchmark for Evaluating the Accuracy of Diff Tools. *ACM Trans. Softw. Eng. Methodol.*, 34(2):40:1–40:63, January 2025.

[2] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. *SIGMOD Rec.*, 25(2):493–504, June 1996.

[3] G. Cobena, S. Abiteboul, and A. Marian. Detecting changes in XML documents. In *Proceedings 18th International Conference on Data Engineering*, pages 41–52, February 2002. ISSN: 1063-6382.

[4] Jean-Remy Falleri and Matias Martinez. Fine-grained, accurate and scalable source differencing. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, pages 1–12, New York, NY, USA, April 2024. Association for Computing Machinery.

[5] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 313–324, New York, NY, USA, September 2014. Association for Computing Machinery.

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

[7] René Just, Darioush Jalali, and Michael D. Ernst. 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, pages 437–440, New York, NY, USA, July 2014. Association for Computing Machinery.

[8] Quentin Le Dilavrec, Djamel Eddine Khelladi, Arnaud Blouin, and Jean-Marc Jézéquel. HyperAST: Enabling Efficient Analysis of Software Histories at Scale. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, pages 1–12, New York, NY, USA, January 2023. Association for Computing Machinery.

[9] Quentin Le Dilavrec, Djamel Eddine Khelladi, Arnaud Blouin, and Jean-Marc Jézéquel. 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 2023, pages 288–299, New York, NY, USA, November 2023. Association for Computing Machinery.

[10] Matias Martinez, Jean-Rémy Falleri, and Martin Monperrus. Hyperparameter Optimization for AST Differencing. *IEEE Transactions on Software Engineering*, 49(10):4814–4828, October 2023. arXiv:2011.10268 [cs].

[11] Stefan Schwarz, Mateusz Pawlik, and Nikolaus Augsten. A New Perspective on the Tree Edit Distance. In *Similarity Search and Applications*, pages 156–170. Springer, Cham, 2017. ISSN: 1611-3349.

[12] Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, and Marcelo A. Maia. Dissection of a bug dataset: Anatomy of 395 patches from defects4j. In *Proceedings of SANER*, 2018.

[13] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. RefactoringMiner 2.0. *IEEE Transactions on Software Engineering*, 48(3):930–950, March 2022.

[14] Nikolaos Tsantalis, Matin Mansouri, Laleh M. Eshkevari, Davood Mazinanian, and Danny Dig. Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th International Conference on Software Engineering*, pages 483–494, Gothenburg Sweden, May 2018. ACM.

A Reproducibility

The benchmarks and implementation of the heuristics can be found in our fork of HyperAST at `https://github.com/LeaderSupreme/HyperAST` (also linked in footnode 3). The implementation of the bottom-up matchers can be found in `crates/hyper_diff/src/matchers/heuristic/gt`, where the "simple_bottom_up_matcher.rs" and "lazy_simple_bottom_up_matcher.rs" files contain the Simple and lazy Simple heuristic respectively. The benchmarks can be found in `benchmark_diffs/benches`. To run the benchmark measuring CPU-cycles checkout `https://github.com/HyperAST/HyperAST/commit/a283fb7f2b52f0ecbfd62fe5e313de7a3fb6be05`, if you instead want to run the benchmark measuring runtime checkout `https://github.com/HyperAST/HyperAST/commit/70157242a6c767a51229abfcc68daa8e1566de7c`. In both cases the benchmark is in the "bench_final.rs" file, and can be ran using: `cargo bench -bench bench_final`.

B Use of LLMs

I have used a LLM in the process of writing this paper. It was used as a tool, and not to replace critical thinking. I used it for creating the latex skeleton of some figures and tables, including Figure 1, and Table 1. I also asked it to pe

Prompts to the LLM were:

- "I have 2 pictures I want to display in my latex file which is two columned. The pictures should be shown next to each other and span both columns. I want them to have their own caption, and one big caption for the whole figure. The latex file is a research paper."
- "can you write a sekeleton for a small table for me? it want have two columns x and y, and have 4 rows, a middle stripe, and then again 4 rows. It is like a matrix, I want along the axis of the rows both sections to have their own name. this name should be vertical and be next to the row names, this name should also have wrap back when the text overlaps."
- "Can you peer review <(partial) section> for me. Make sure everything is correct, the sentences are well formed and there are no spelling mistakes."

C Tables

(a) GitHub Java Dataset

Project	Source Nodes	Target Nodes	Top Down	Greedy_1000	Lazy Greedy_1000	Greedy_200	Lazy Greedy_200	Simple	Lazy Simple
apache-commons-cli	178545	185073	130490	133699	133699	132903	132903	132418	132242
drool	141149	152709	104362	107808	107808	106886	106888	107224	106884
elastic-search	183481	193944	146815	150552	150552	149585	149585	149892	149620
google-guava	253863	260088	203353	204303	204303	204152	204152	204876	204735
h2	694494	706893	571415	573892	573892	573090	573090	574998	574814
jabref	555839	590774	469080	470228	470228	470007	470007	470172	470092
killbill	135830	143245	103889	109305	109305	108104	108110	108550	107850
ok-http	560558	572878	454230	457150	457150	456456	456456	456977	456757
signal-server	187088	197284	149397	153401	153402	152576	152576	152626	151980
zaproxy	297304	310678	233423	239208	239208	237822	237848	238121	237753

(b) Defects4J Dataset

Project	Source Nodes	Target Nodes	Top Down	Greedy_1000	Lazy Greedy_1000	Greedy_200	Lazy Greedy_200	Simple	Lazy Simple
Chart	326328	329149	261660	262527	262527	262396	262396	264557	264518
Cli	157170	159436	120862	122409	122409	121903	121905	123776	123707
Closure	2459361	2475749	1963406	1972313	1972313	1970483	1970499	1981949	1981341
Codec	139530	141474	110492	111252	111252	110926	110926	111970	111925
Collections	42339	42815	33821	33932	33932	33908	33908	34153	34153
Compress	338963	342214	264550	265873	265873	265674	265674	268024	267917
Csv	83576	84002	65690	66044	66044	65950	65950	66715	66703
Gson	159699	161913	122275	122811	122811	122719	122719	123399	123386
JacksonCore	619320	622549	485404	486361	486361	486226	486226	488791	488737
JacksonDatabind	1086790	1104660	856823	862592	862592	861119	861119	870579	869753
JacksonXml	38174	39887	29369	29504	29504	29479	29479	30061	30019
Jsoup	996501	1004463	809064	813207	813207	812268	812268	818838	817409
JxPath	157274	160403	123266	124301	124301	123917	123917	124307	124269
Lang	722814	728466	559120	560815	560815	560383	560383	567604	567522
Math	913932	920430	716363	720552	720552	719308	719308	725717	725499
Mockito	76337	79779	57880	59508	59508	58999	58999	59624	59593
Time	284806	288258	223639	224564	224564	224369	224369	226828	226800

Table 2: Number of mappings found in total. The 1000 and 200 in the Greedy_1000 and Greedy_200 refer the the "Max Size" threshold used.

(a) GitHub Java Dataset

Project	Greedy_1000	Greedy_200	Simple	Lazy Greedy_1000	Lazy Greedy_200	Lazy Simple
apache	266.11	9.98	0.33	266.32	10.15	0.61
drool	541.75	29.67	0.29	541.94	30.05	0.54
elastic	1401.96	22.76	0.38	1402.30	23.34	0.72
google	239.88	22.79	0.33	240.04	23.02	0.75
h2	619.24	24.40	1.28	620.03	25.13	2.55
jabref	232.27	8.92	0.49	232.73	9.40	1.45
killbill	578.64	28.71	0.31	578.63	29.63	0.56
ok	823.32	21.26	0.71	823.92	21.77	1.66
signal	624.55	26.61	0.33	638.30	26.81	0.64
zaproxy	1648.25	61.58	0.78	1648.57	65.15	1.33

(b) Defects4J Dataset

Project	Greedy_1000	Greedy_200	Simple	Lazy Greedy_1000	Lazy Greedy_200	Lazy Simple
Chart	189.40	7.35	0.82	189.77	7.70	1.43
Cli	287.08	10.08	0.34	287.26	10.85	0.62
Closure	1636.14	71.65	6.25	1639.04	75.55	10.81
Codec	314.19	4.32	0.28	314.36	4.47	0.55
Collections	13.31	0.81	0.08	20.55	0.85	0.15
Compress	355.71	5.34	0.66	356.12	5.70	1.28
Csv	159.86	2.44	0.17	159.96	2.52	0.32
Gson	228.80	8.74	0.34	228.99	8.91	0.62
JacksonCore	164.72	4.36	1.30	165.38	5.03	2.45
JacksonDatabind	1493.86	48.96	2.51	1504.09	50.03	4.47
JacksonXml	23.76	0.46	0.07	23.81	0.50	0.14
Jsoup	1077.72	23.63	2.77	1078.99	24.79	4.69
JxPath	328.65	12.13	0.31	328.85	12.30	0.60
Lang	565.66	14.66	1.91	566.43	15.39	3.20
Math	1155.29	23.22	2.22	1156.30	24.20	3.89
Mockito	522.26	18.09	0.16	522.18	18.18	0.29
Time	196.94	11.90	0.67	197.26	12.19	1.19

Table 3: Median (from 15 samples) CPU Cycle Performance for different heuristics (in billions of cycles) (All with a confidence level of 95%). The 1000 and 200 in the Greedy_1000 and Greedy_200 refer the the "Max Size" threshold used.