Evaluating Stable Tree Differencing with Gumtree and HyperDiff

Elias Hoste

Supervisors: 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: Elias Hoste
Final project course: CSE3000 Research Project
Thesis committee: Jesper Cockx, Carolin Brandt, Quentin le Dilavrec

An electronic version of this thesis is available at http://repository.tudelft.nl/.

Abstract

Structural code differencing algorithms are used in software engineering tasks such as version control, code review, and change classification. While the Gumtree algorithm is a popular choice due to its performance and accuracy, it is inherently unstable: the output of a diff may change depending on the direction in which it is computed. A stable variant of Gumtree has been proposed to enforce directional symmetry in mappings. In this paper, we empirically evaluate the performance overhead of Gumtree Stable compared to the original Greedy variant. We also assess the impact of lazy evaluation, enabled by the HyperDiff framework, as a general optimization technique applicable to both variants. Our experiments on real-world code changes show that while stability introduces slight overhead in most cases, some cases see relatively large performance gains, averaging out to a runtime improvement when run on larger datasets. Additionally, lazy evaluation significantly reduces runtime for both Stable and Greedy variants. These findings clarify the trade-offs involved in adopting stable differencing and demonstrate how HyperDiff can be used to optimize structural diff computations independently of algorithmic stability.

1 Introduction

Code differencing is a valuable way to gain insight into the evolution of a codebase over time. Having correct, efficient, and robust algorithms to compute diffs is thus equally valuable. The most widely adopted algorithms, like Myers' [9], use text-based code differencing, which compares code on a character level. However, text-based approaches are limited in their ability to capture syntactic or structural changes, often resulting in noisy or misleading diffs.

To address these issues, structural diff algorithms have been proposed, which instead compare Abstract Syntax Trees (ASTs) of the code. Notable examples include Gumtree [4], ChangeDistiller [6] and MTDIFF [3]. These algorithms offer a much clearer way of representing changes between files for both developers and researchers, as they can avoid structural noise and more accurately reflect the developer's actual changes in the file.

One desirable property of diff algorithms is stability: whether a diff of a file is the same when the two input versions are reversed. Most existing algorithms are not stable, which can cause inconsistencies, for example in git blame attribution or merge operations. Stability provides reversibility, determinism, and reduces noise, which are features particularly valuable in tools like Git. Despite its practical relevance, stability has received little explicit attention in the literature. Most prior work on structural differencing focuses on accuracy or classification of edits, rather than on the consistency of mappings across input orderings. Notably, while Gumtree does have a stable variant (Gumtree Stable), it has not been widely referenced in the literature, and its usage in practical applications has been limited, most likely due to its slight performance overhead. As such, the performance implications of enforcing stability remain poorly understood. Improving the performance of stable algorithms could make them more viable to use and help fill this gap.

Recent advances in performance optimization have leveraged redundancy across versions. HyperAST [7] is a novel way of representing ASTs that deduplicates repeated code across files and versions, using the fact that most code remains unchanged between versions. It underlies the HyperDiff framework [8], which provides a collection of diff algorithms optimized to use HyperASTs to compute diffs over many versions at once, allowing for efficient large-scale code analysis.

In this paper, we consider HyperDiffs effects on Gumtree, a family of structural diff algorithms. Two relevant Gumtree variants are GumTree Greedy (an efficient but unstable baseline), and GumTree Stable (a stable adaptation of Gumtree Greedy). While HyperDiff includes an implementation of GumTree Greedy, it currently lacks support for the stable variant. As such, the main contributions of this paper are as follows:

- An implementation of Gumtree Stable integrated into the HyperDiff framework, filling the gap in the HyperDiff implementation which currently only supports Gumtree Greedy.
- An analysis of the trade-offs between performance and stability when choosing to use Gumtree Stable, with a detailed performance comparison against Gumtree Greedy.

These contributions directly address the knowledge gap regarding stability in structural code differencing algorithms, which has been underexplored in existing research. To this end, we ask the following research questions:

- *How do optimizations made possible by HyperDiff affect the performance of Gumtree Stable and Greedy?*
- *What are the trade-offs between stability and performance when using Gumtree Stable?*

The remainder of the paper is structured as follows. Section 2 provides the necessary context and technical foundations. Section 3 presents our contribution. Section 4 reports on the results of our research. Section 5 reflects on ethical implications, and section 6 concludes and summarizes the paper.

2 Background

To better understand stability in Gumtree, this section first introduces abstract syntax trees (ASTs), then describes how the Greedy and Stable variants of Gumtree differ algorithmically. It concludes with an explanation of how HyperDiff enables performance optimizations.

2.1 Abstract Syntax Trees

An abstract syntax tree (AST) is a labeled, ordered, and rooted tree parsed from source code. Each node has a label, which represents structural parts of the code like

StringLiteral or ReturnStatement, and an optional value that corresponds to the tokens in the code. For example, a node with label StringLiteral may have a value of "foo". ReturnStatement could have a value of return, but as this does not encode any new information, it is discarded. Which data is stored in the label or the value of a node can differ per implementation of AST parsing. For example, return "foo" could also be parsed as a single node with label Statement and value return "foo". Generally, more granular representations like the former are preferred [4, Sec. 2].

2.2 Gumtree Greedy and Stable

A structural diff algorithm can be modeled as a function diff$(T_1, T_2) \rightarrow \mathcal{M}$, producing a set of mappings $\mathcal{M}$ between two input ASTs T_1 and T_2. These mappings can be transformed into an edit script using Chawathe's algorithm [1].

Both Gumtree Greedy and Stable consist of two phases: top-down and bottom-up. In the top-down matching phase, exact subtree matches are found between T_1 and T_2. The bottom-up phase proceeds by visiting nodes in T_1 in post-order and mapping them to the best candidate in T_2, defined by the similarity function dice(t_1, t_2) [4, Sec. 3.1]. This function computes the ratio of common descendants to total descendants, and is symmetric: dice$(x, y) =$ dice(y, x).

To formalize the main difference, let bestCand(n) return the node with highest similarity to n in the other tree, provided it exceeds a similarity threshold (usually 0.5). Greedy maps each node n to bestCand(n) if it exists. Stable adds a constraint: a mapping is only made if $n =$ bestCand(m), where $m =$ bestCand(n), making the matching criteria symmetric.

After each bottom-up match, both variants invoke a recovery phase, which runs a cubic-time algorithm on the matched subtrees to match nodes that were missed. To limit performance cost, this phase is restricted to small subtrees (bounded by a max size). Because Stable performs more conditional checks and may invoke recovery less often, its runtime impact can vary and is not strictly higher than Greedy.

Algorithm 1 outlines the bottom-up phase of Gumtree Stable in pseudo-code. Note that the Greedy algorithm is mostly the same, except for the $bestCand(t_2, minDice) = t_1$ check. Gumtree Stable does feature an implementation-level optimization not used in Gumtree Greedy, but we do not focus on that in this paper.

An example of two input ASTs that cause instability in Gumtree Greedy can be found in fig. 1. In the top-down phase, leaf nodes which exist in both trees are mapped to each other. In the bottom-up phase, Gumtree Greedy maps x to z when mapping from t_1 to t_2, but maps y to x (leaving z unmapped) when mapping from t_2 to t_1. In contrast to this, Gumtree Stable maps x to z in both ways.

2.3 Lazification with HyperDiff

The main performance improvement evaluated in this paper comes from lazifying the original Gumtree variants using HyperDiff [8]. Lazification refers to the technique of deferring expensive tree decompression operations until they are strictly necessary, thus avoiding unnecessary computation and memory usage.

Algorithm 1: Gumtree Stable bottom-up phase with recovery

```
Data: Two trees $T_1$ and $T_2$, a set of mappings $\mathcal{M}$
      (resulting from the top-down phase), a
      similarity threshold $minDice$ and a maximum
      tree size $maxSize$
Result: The set of mappings $\mathcal{M}$
for $t_1 \in T_1 \mid t_1$ is not mapped, in post-order do
    $t_2 \leftarrow bestCand(t_1, minDice)$;
    if $t_2 \neq null \wedge bestCand(t_2, minDice) = t_1$ then
        $\mathcal{M} \leftarrow \mathcal{M} \cup (t_1, t_2)$;
        if $max(size(t_1), size(t_2)) < maxSize$ then
            for $(t_a, t_b) \in zs(t_1, t_2)$ do
                if $t_a, t_b$ not mapped $\wedge$
                  $label(t_a) = label(t_b)$ then
                    $\mathcal{M} \leftarrow \mathcal{M} \cup (t_a, t_b)$;
```

HyperDiff internally represents input ASTs as a compressed DAG structure called a HyperAST, where shared subtrees are stored only once. This enables deduplication, but it comes at the cost of losing some global structural information such as parent links or child orderings. To recover this information when needed, HyperDiff incrementally constructs a decompressed tree: a linearized view of the tree in post-order, backed by data structures like arrays of parent pointers and leftmost descendants.

The key innovation is that this decompressed tree can be constructed lazily, meaning that tree nodes and their structural relationships are only computed when required by the diff algorithm. For example, if two subtrees are matched early in the algorithm based on metadata or referential equality, then their descendants never need to be decompressed at all.

3 Contribution

This section highlights the main contributions of the research: first discussing the definition of stability, then how Gumtree Stable was implemented in HyperDiff, and finally how stability was verified using tests.

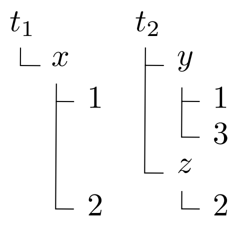
3.1 Formalizing Stability

Section 2 modeled a code differencing algorithm as diff$(T_1, T_2) \rightarrow \mathcal{M}$, where T_1 and T_2 are syntax trees of the code, and $\mathcal{M}$ represents mappings between nodes of these two trees. Using these definitions, we can subsequently formalize the definition of stability as follows: diff$(T_1, T_2) =$ diff(T_2, T_1) for every input pair T_1, T_2.

3.2 Implementing Gumtree Stable

To implement Gumtree Stable and Lazy Stable in HyperDiff, we referenced the existing Greedy and Lazy Greedy implementations in HyperDiff, and the implementation of Gumtree Stable in Java[1]. While Gumtree Greedy uses Zs [11] as its re-

[1]https://github.com/GumTreeDiff/gumtree/tree/0076da2



(a) Two simplified ASTs t_1 and t_2, representing two different versions of a file.

	x	y	z	`bestCand`
x	1	1/2	2/3	z
y	1/2	1	0	x
z	2/3	0	1	x

(b) Pairwise `dice` scores and `bestCand` selection between nodes in t_1 and t_2.

Figure 1: Example of inputs that cause instability in Gumtree Greedy. (a) shows two input ASTs with shared substructure. (b) gives the similarity values and best candidate matches used to determine mappings during the diff.

covery algorithm, which is optimal but generally quite slow, the Java implementation of Gumtree Stable features an alternative faster recovery phase using histograms, which is based on heuristics instead of optimality. Since we aim to only measure the effect of enforcing stability and not the effect of recovery algorithms, we use the Zs algorithm for our implementation of Gumtree Stable. Pseudo-code of our implementation can be found in algorithm 1, and our final implementation can be seen in our public HyperDiff repository[2].

3.3 Testing Stability

To verify the stability of our implementation of Gumtree Stable, we specifically designed test cases which result in instability in Gumtree Greedy, while ensuring that Gumtree Stable remains stable. One of these test cases can be found in fig. 1. We also tested examples which would not result in instability in Gumtree Greedy. For each example, we ran the specified algorithm (Greedy or Stable) in both ways (diff(T_1, T_2) and diff(T_2, T_1)) and compared the mappings on equality. All tests and test cases used can be found in the same repository as our implementation.

4 Evaluation

This section describes our benchmarks by first discussing the dataset, and then the statistics used to evaluate our benchmarks. We present our results as an answer to our research questions, and finally discuss potential threats to the validity of this paper.

4.1 Dataset

In order to benchmark the different Gumtree variants, we used the Defects4J dataset from the Gumtree datasets repository[3]. This dataset consists of 1046 file pairs containing bugs and their respective fixes, sourced from 17 different open-source Java projects. Files used range from 14 to 6591 lines of code. The Gumtree datasets are structured to be simple to process and contain both Java and Python datasets, but as HyperDiff currently does not support Python AST parsing, we only used Java.

[2]https://github.com/Pomegranate123/HyperAST/tree/33ded64

[3]https://github.com/GumTreeDiff/datasets/tree/33024da

4.2 Benchmark Setup

The benchmarks were run on an Intel Core i5-6200 CPU with 16GB of RAM using rustc 1.85.1 (2025-03-15). Each variant (Greedy, Stable, Lazy Greedy, Lazy Stable) was executed 12 times on each file pair in the dataset, varying from 1 to 300. We discard the first two runs as warm-up and calculate the median runtime per file pair from the remaining 10 runs. This entire procedure was repeated for various values of `maxSize` (1, 10, 100, 200, 300) to further assess if Gumtree Stable is able to avoid doing recovery by making more efficient mappings.

To compare the paired runtimes between algorithm variants, we used the rank-biserial correlation [2] (RBC), which quantifies effect size on a scale from –1 (variant A consistently faster) to +1 (variant B consistently faster). For statistical significance, we use the Wilcoxon signed-rank test [10]: a non-parametric alternative to the paired t-test which does not assume normally distributed data. To further quantify performance differences, we report the median and mean runtime deltas (B - A) across all file pairs, and the relative speedup based on the mean log-ratio of runtimes, defined as the mean of log(B/A). The log-ratio is symmetric and reflects the geometric mean performance change, which is preferred over arithmetic means in benchmarking[5].

4.3 Results

We now present the results of running our benchmarks and answer both our research questions. All results were deemed statistically significant with p $\ll 10^{-16}$ using the Wilcoxon signed-rank test.

RQ1: How do optimizations made possible by HyperDiff affect the performance of Gumtree Stable and Greedy?

As can be seen in fig. 2, applying HyperDiff-based lazy optimizations results in a roughly $20\% - 45\%$ speedup for both Greedy and Stable variants, depending on the `maxSize` used. The mean and median deltas also indicate clear runtime improvements both on average and for most files. The rank-biserial correlations near 1.0 indicate that these improvements are consistent across almost all files, which is also visible in fig. 3. The performance difference between different values of `maxSize` can be explained by the recovery phase being executed less often when run with a lower `maxSize`, resulting in fewer nodes to decompress. The non-lazy variant, in contrast, does not benefit from this as it always decompresses both trees. These results suggest that incorporating HyperDiff's optimization techniques improve performance without obvious drawbacks, making it highly beneficial for both Greedy and Stable variants.

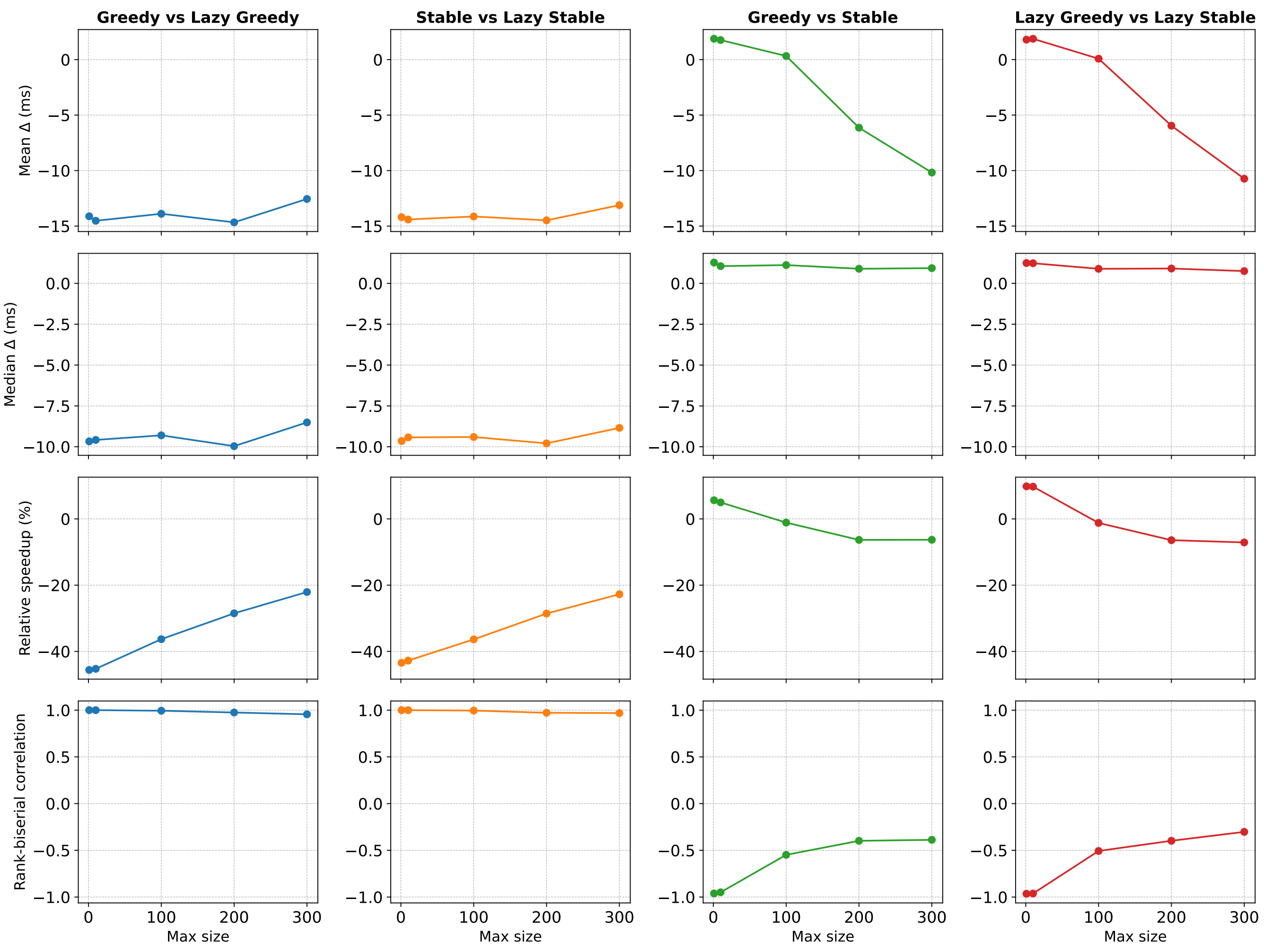


Figure 2: Pairwise comparison of Gumtree variants with various `maxSize` values, showing mean and median runtime deltas over all file pairs, and the relative speedup percentages based on the mean log-ratio (where positive values indicate the first variant is faster), as well as the rank-biserial correlation (where positive values indicate the second variant is faster)

RQ2: What are the trade-offs between stability and performance when using Gumtree Stable?

In the comparison of both Greedy vs Stable and Lazy Greedy vs Lazy Stable, the performance statistics change significantly depending on the `maxSize` parameter. When `maxSize` is set to 1, Greedy consistently outperforms Stable by a small margin, as suggested by a positive mean delta, relative speedup, and an RBC near -1. However, as `maxSize` increases, causing the slow recovery algorithm to be run more frequently, the mean delta and mean log-ratio shift in favor of Stable: when averaged, Stable now outperforms Greedy. Interestingly, the median delta stays consistently positive, and while the RBC trends towards 0, it still generally favors Greedy. This reveals an important nuance: while Greedy is typically faster on most diffs, Stable achieves larger gains on a smaller subset of files by reducing the recovery overhead. This can also be seen in the right two columns of fig. 3: while most runtime deltas are in favor of Greedy, deltas in favor of Stable are up to 10 times larger for higher values of `maxSize`. From a practical standpoint, that makes Stable an attractive option for computing diffs over entire codebases, where total runtime is more sensitive to outliers. In contrast, Greedy is still more efficient for the majority of file pairs when performance per file is the primary concern.

In addition to the performance trade-offs, Stable offers other benefits not quantified in this benchmark: it ensures reversibility, determinism and reduces noise in diffs. For example, Gumtree Stable allows skipping the computation of both diff directions in symmetric applications, effectively halving the runtime and storage cost. This is especially useful in scenarios such as version control analysis, where changes are not strictly directional. These benefits make Stable a more reliable choice in workflows where consistency is critical.

4.4 Threats to validity

We discuss potential threats to the validity of our experimental results, grouped into internal and external threats.

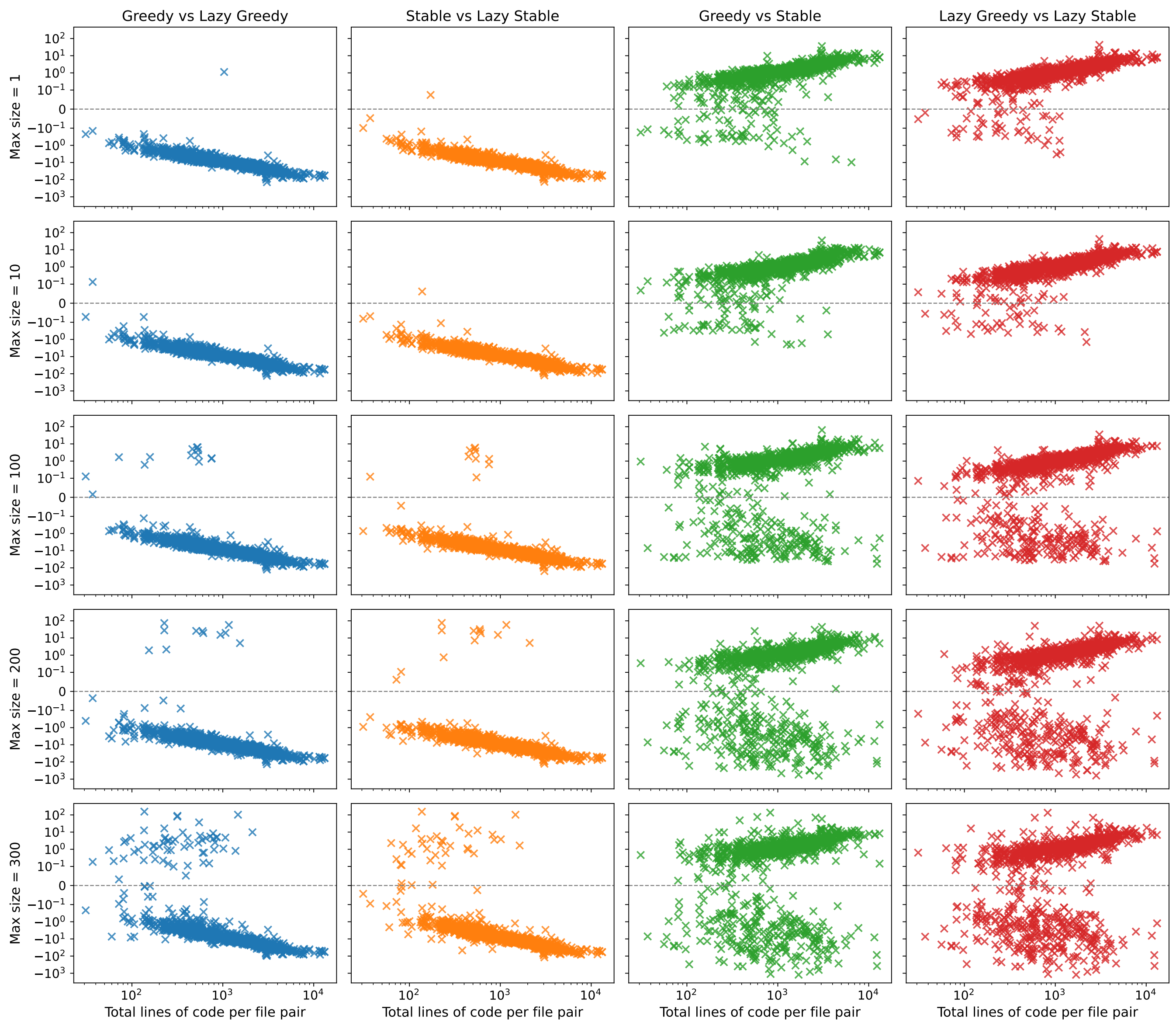


Figure 3: Runtime difference between Gumtree variants (variant B - variant A) compared to the total lines of code (LoC) in each file pair, for various `maxSize` values. The x-axis represents the total LoC, plotted on a log scale, while the y-axis shows the symmetric log of the runtime delta between the variants. Positive values indicate the first variant is faster, while negative values show the second is faster.

Internal Validity

These threats concern the correctness and reliability of our experimental procedure and measurements.

- **Measurement Noise:** Although each algorithm variant was executed 12 times per file pair, and the first two runs were discarded as warm-up, runtime measurements may still be affected by system-level noise like background processes. We mitigate this by taking the median of the remaining 10 runs, which reduces the impact of outliers.
- **Implementation Bias:** All algorithm variants were implemented within the same Rust codebase to ensure consistency. However, subtle implementation differences or unintentional optimizations may benefit certain variants. We aimed to isolate variant-specific behavior and share as much code as possible to reduce this risk.
- **Choice of Statistisc:** We report a range of performance statistics, including mean and median runtime deltas, mean log-ratios, and rank-biserial correlation. Each of these highlights different aspects of performance, and overreliance on any single metric could bias interpretation.

External Validity

These threats concern the generalizability of our findings to other contexts and usage scenarios.

- **Dataset Representativeness:** The benchmark dataset comprises file pairs from 17 real-world Java codebases, covering file sizes ranging from 14 to 6591 lines of code. However, it does not capture the full diversity of software changes encountered in practice (such as merge conflicts, large-scale refactorings, or other programming languages). Further studies are needed to confirm the generality of our results.

- **Hardware and Compilation Environment:** Benchmarks were run on an Intel Core i5-6200 CPU with 16GB RAM, using rustc version 1.85.1 (2025-03-15). Performance characteristics may differ on other hardware platforms (e.g., ARM, high-core-count systems) or under different compiler configurations.
- **Narrow Performance Focus:** Our evaluation focuses solely on runtime performance. Other relevant dimensions, such as the quality of the resulting edit script, ease of human interpretation, or integration with version control tools, are not addressed in this study, but may impact practical adoption.

5 Responsible Research

All experiments are conducted on publicly available code repositories. No personal or sensitive data is used at any point during our research. To ensure reproducibility, all Gumtree variants are implemented in a shared infrastructure, HyperDiff. The benchmarking dataset[4], framework, and tests are all publicly available[5].

We are aware that algorithmic benchmarking can be sensitive to dataset composition bias. Our dataset includes 1046 file pairs of bug fixes from 17 different Java projects, with file sizes ranging from 14 to 6591 lines of code. However, bias could be further reduced by adding other languages and edit types to the dataset. Results are reported as-is, including any cases where stable variants do not show clear advantages.

The aim of this research is to quantify the practical cost of enforcing stability in Gumtree. While not contributing new algorithmic techniques, this evaluation informs future tool builders and researchers about the trade-offs involved in choosing whether to use stable diff algorithms. More predictable diff behavior can benefit downstream applications such as blame tools, refactoring detectors, and code review interfaces. The potential impact is an improvement in the reliability of these tools without sacrificing performance. No foreseeable harm is expected from this research, and care is taken to ensure that performance measurements are contextualized and not used to promote misleading conclusions.

6 Conclusions and Future Work

In this paper, we evaluated our own implementation of Gumtree Stable in the HyperDiff framework, applied lazy optimizations, and compared it to existing implementations of Gumtree Greedy and its lazy counterpart using benchmarks. To conclude our work, we briefly answer our research questions and discuss possible future work.

[4]https://github.com/GumTreeDiff/datasets/tree/33024da

[5]https://github.com/Pomegranate123/HyperAST/tree/33ded64

6.1 Research Questions

In section 1, we introduced our two research questions:

How do optimizations made possible by HyperDiff affect the performance of Gumtree Stable and Greedy? The results show that applying Hyperdiff-based lazy optimizations leads to significant and consistent (with a rank-biserial correlation near 1) runtime performance improvements across both Greedy and Stable variants. Both variants show an equal relative speedup of roughly $20\% - 45\%$ compared to their non-lazy variants, implying HyperDiff optimizations are a viable way to scale these algorithms.

What are the trade-offs between stability and performance when using Gumtree Stable? The results show that Gumtree Greedy generally has a lower median runtime, but Stable achieves better average performance when the `maxSize` parameter rises above 100. While Greedy is faster on most files, Stable avoids costly recovery passes in some cases and can outperform Greedy when applied on a codebase-scale. Stable is preferred for larger scale code differencing or in cases where symmetry and consistency are valued, while Greedy may be better when performance per file is more important.

6.2 Future work

Some suggestions for future work are discussed in the following paragraphs.

Evaluate trade-off between `maxSize` and performance: While we observed interesting performance shifts related to the `maxSize` parameter, a more focused study could model and predict its effect on the performance of Greedy and Stable.

Broaden benchmark dataset: To increase generalizability, future benchmarks could include additional programming languages, like Python or Rust, and more diverse edit types, such as merge commits or refactorings, to reduce bias in our results.

Evaluate diff quality: Our work focuses solely on performance analysis of different Gumtree variants. An important next step is to evaluate whether match quality differs significantly between both Greedy/Stable and lazy/non-lazy, especially with varying values of `maxSize`. Measuring the diff quality could further clarify the trade-offs between these Gumtree variants.

Assess usability of stable diffs: While stable diffs are hypothesized to be more accurate and usable to developers because of its stricter matching criteria, this remains to be verified. Empirical validation via user studies could investigate how stable diffs affect developer experience or code review efficiency.

A Disclaimer on use of LLMs

Large Language Models (LLMs), specifically ChatGPT, were used as a productivity aid throughout this research. The model was employed to help summarize findings and improve the structure and fluency of the written text. No parts of the paper were written entirely by the model, and all scientific claims, analyses, and conclusions were authored and verified.

LLMs were also used to help generate python scripts that provide statistical analysis or plots based on benchmark results. All scripts and their outputs were carefully reviewed for validity and adjusted as needed.

No verbatim output from the model is included in the final paper. The author acknowledges the use of ChatGPT in the development of this work as described above. All final content is the responsibility of the author.

Below is a sample of representative prompts used to interact with the LLM during the writing and research process:

- "How can I compare algorithm runtimes using paired data? Which statistics are best suited?"
- "How can I change my introduction to more clearly address the knowledge gap in existing research?"
- "How do I force a two-column table to appear exactly where I place it in LaTeX?"
- "Generate a Python script that reads benchmark CSV files and plots median/mean runtime deltas by combined file size."
- "What are some things I should mention in a 'threats to validity' section for a paper benchmarking code differencing algorithms?"

Additional prompts were used iteratively to refine writing, generate explanations, and format LaTeX content. All final decisions and edits were made by the author.

References

[1] 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.

[2] Edward E. Cureton. Rank-biserial correlation. *Psychometrika*, 21(3):287–290, 1956.

[3] Georg Dotzler and Michael Philippsen. Move-optimized source code tree differencing. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 660–671, 2016.

[4] 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, page 313–324, New York, NY, USA, 2014. Association for Computing Machinery.

[5] Philip J. Fleming and John J. Wallace. How not to lie with statistics: the correct way to summarize benchmark results. *Commun. ACM*, 29(3):218–221, March 1986.

[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, 2007.

[7] 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, New York, NY, USA, 2023. Association for Computing Machinery.

[8] 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, page 288–299, New York, NY, USA, 2023. Association for Computing Machinery.

[9] Eugene W Myers. An o(nd) difference algorithm and its variations. *Algorithmica*, 1(1):251–266, 1986.

[10] Frank Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, 1945.

[11] Kaizhong Zhang and Dennis Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal on Computing*, 18(6):1245–1262, dec. 1989.