Analyzing Iterative Improvements to Structural Code Diffs

Maciej Mejer[1]

Supervisor(s): Carolin Brandt[1]**, Quentin Le Dilavrec** [1]

[1]**EEMCS, Delft University of Technology, The Netherlands**

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

Abstract

Version control systems rely on code differencing algorithms to track changes and support key development tasks such as merging, code search, and code reviews. Traditional differencing techniques operate on plain-text representations of source code, which sometimes fail to convey the original intent behind code modifications. To address this, modern algorithms operate on abstract syntax trees (ASTs), enabling more accurate and structurally meaningful edit scripts. However, computing differences between ASTs poses new challenges, especially in balancing edit script quality with runtime performance.

This paper investigates the evolution of AST differencing algorithms by analyzing a sequence of key refinements built on top of Xy - a foundational algorithm originally designed for XML. We evaluate three influential enhancements: GumTree's optimal recovery strategy, the simplified recovery heuristic introduced in GumTree Simple, and HyperDiff's use of a compressed AST representation. We evaluate each refinement independently using a shared benchmarking framework and a dataset of real-world Java code changes. Our results show how each refinement incrementally improves scalability, runtime stability, and script quality. These findings offer a deeper understanding of the design trade-offs in AST differencing and provide guidance for developing efficient and interpretable structural diff tools at scale.

1 Introduction

Version control systems (VCS) are foundational to modern software development. They serve not only as repositories of code but also as detailed records of the evolution of software. By tracking changes over time and supporting operations such as branching, merging, and reviewing, version control has become a necessity in virtually all software projects, from small-scale applications to large-scale industrial systems.

A central component of version control is the ability to represent versions of code with so-called *edit scripts*. Edit scripts describe how one version of a file can be transformed into another through a sequence of simple operations. The most commonly used techniques for computing edit scripts interpret the source code as plain text lines that can be added or deleted. In that case, finding the shortest edit script corresponds to the Edit Distance (ED) problem [1] which is efficiently solved with Myer's algorithm [2]. While this approach provides good scalability due to its simplicity, it has limited accuracy when inferring changes in code because it fails to account for factors like hierarchies and recursive structures, which are particularly important in the context of programming languages. One consequence of this is that small edits (such as variable renames) are confusingly described as a deletion and addition of entire lines. Another problem of this approach is that it is unable to identify moves of chunks of code which are a very common operation in any software projects.

To overcome these limitations, we use algorithms that operate on abstract syntax trees (ASTs) instead of plain text. ASTs capture the semantics of code, allowing us to detect a richer set of changes, including moves and renames in addition to insertions and deletions, which significantly improves the interpretability of the resulting edit scripts.

However, with ASTs, the initial Edit Distance (ED) optimization problem becomes a Tree Edit Distance (TED) problem, defined as the minimum-cost sequence of node edit operations that transform one tree into another [3]. Unfortunately, general TED is an NP-hard problem [4], and optimal algorithms for it do not scale to real-world inputs. Therefore, practical systems must trade off some accuracy in favor of improved performance. Early work in this area introduced XY-diff [4], a simple and effective heuristic approach originally designed for detecting changes in XML documents. Although XY-diff made simplifying assumptions about tree structures, it introduced a flexible algorithm framework which encouraged simple incremental improvements to employed heuristics over time.

Building upon XY-diff, more recent algorithms, in particular GumTree [5], GumTree Simple [6], and HyperDiff [7], have refined the tradeoff between accuracy and performance. In this paper, we revisit three incremental improvements to AST differencing algorithms that we believe were the most influential to the evolution of structural code differencing. The main research question we address is formulated as follows:

RQ: How have key heuristic refinements in AST differencing algorithms improved the balance between performance, scalability, and edit script quality?

To explore this, we investigate the following sub research questions each aimed at understanding one of the three main refinements:

- **RQ1: How does the refinement of Xy's recovery strategy in GumTree affect the trade-off between script quality and performance when applied to complex ASTs?**
- **RQ2: To what extent does the simple recovery improve upon GumTree in terms of scalability, runtime stability, and edit script quality for large ASTs?**
- **RQ3: How does the use of the HyperAST data structure in HyperDiff impact performance and scalability compared to GumTree, particularly in cases with a varying number of changes between ASTs?**

To answer these questions, we investigate in detail the impact of every refinement. We identify the main goal of every refinement, describe the measures taken to achieve the goal, and analyse the impact on the efficiency, quality, and overall usability by thoroughly benchmarking the achieved performance.

2 Motivation

The quality of edit scripts is critical for two primary reasons. First, compact and semantically accurate edit scripts enable version control systems to accurately perform operations such as Automatic Program Repair [8; 9; 10], Semantic Code Search [11], or Code Smell Mining [12] on large repositories, helping developers quickly locate relevant code, apply accurate fixes, and improve overall software quality. Second, high-quality edit scripts provide developers with clearer insights into code evolution, improving code reviews, helping in understanding refactorings, and, most importantly, simplifying merging of changes, which enables concurrent collaborative development. Inaccurate or unnecessarily verbose scripts, on the other hand, can obscure the intent of changes, reducing developer productivity.

At the same time, to match the practicality of conventional text-based methods, like Myer's algorithm, AST differening algorithms need to achieve sufficient scalability in terms of runtime. For this reason, these algorithms use heuristics that allow them to improve the runtime at minimal cost in quality. However, to accurately assess their usability, those heuristics need to be thouroughly evaluated and compared in a practical setting, which is a complex task.

Motivated by the need for a comprehensive understanding of the challenges of the quality vs runtime tradeoff of AST diff, this research evaluates AST differencing algorithms to understand how successive refinements to the solution addressed those challenges. We believe that detailed insight into motivation and consequences of past refinements are essential to understand the capabilities and limitations of the general diff approach and that it will help find direction for the future enhancements to the solutions.

3 Background

Understanding how to effectively compute differences between versions of source code requires a solid understanding of the structures and concepts underlying program representation. This section introduces the fundamental definitions used throughout the paper, including abstract syntax trees (ASTs) and edit scripts, and outlines a general framework for how AST differencing algorithms are typically structured. We also provide an overview of prior work in the area, highlighting the evolution of practical, performance-oriented approaches to tree differencing.

3.1 Definitions

An *abstract syntax tree (AST)* is a labeled, ordered, rooted tree that represents the hierarchical syntactic structure of source code according to a formal grammar [5]. Each node in the tree is associated with a label indicating the name of the production rule it instantiates (e.g., `IfStatement`, `Expression`, `Identifier`), and may also hold an optional value, such as a specific token from the source code (e.g., a variable name or literal). The structure of an AST is inherently recursive: each node may have zero or more child nodes, each of which defines a *subtree* that is itself a valid AST corresponding to a syntactic construct in the program.

Following the definition of an AST, we define an *edit script* as a sequence of atomic actions applied to a source AST in order to transform it into a target AST. These atomic actions model fundamental structural changes and are designed to closely mirror how developers reason about code modifications. The supported actions are as defined in [5]:

- $updateValue(t, v_n)$ replaces the value of node t with a new value v_n.
- $add(t, t_p, i, l, v)$ inserts a new node t into the AST. If the parent node t_p and the child index i are specified, t becomes the ith child of t_p. The node t is assigned the label l and value v.
- $delete(t)$ removes the node t from the AST. This operation is restricted to leaf nodes to preserve tree integrity.
- $move(t, t_p, i)$ relocates the subtree rooted at t, making it the ith child of node t_p. All descendants of t are preserved, effectively moving the entire subtree. This behavior mimics how blocks of code such as loops, conditionals, or functions are often moved as coherent units during software evolution.

While multiple edit scripts can transform a given source AST into the same target AST, they can vary significantly in length and complexity. For this reason, the *quality* of an edit script is typically measured by its length, the fewer actions it contains, the more concise and interpretable the transformation. Identifying an edit script of minimal length corresponds precisely to solving the *tree edit distance* [3] problem.

As it turns out, an edit script does not have to be found directly from the trees. Instead, some methods, in particular the methods analysed here, derive the edit script from an initially computed set of node mappings. A *mapping* (also referred to as a *match*) is a pair of nodes from source and target trees. We only consider mapping sets for which every node belongs to at most one pair. An additional constraint on the mapping is that nodes in every pair need to have the same label (in practice, this constraint ensures that only identical code rules can be matched (eg. an `IfStatement` will never be matched to an `Expression`). Having obtained such a set of mappings, we can then derive an edit script using the Chawathe Algorithm [13]. In this paper, we only focus on algorithms that infer the mappings, as Chawathe Algorithm already solves the second step in quadratic time.

3.2 Algorithm framework

At the core, the algorithms analysed in this paper decompose the problem of finding mappings between two ASTs into three distinct phases. These phases are foundational to the overall matching strategy and will be referred to as *Subtree*, *Bottom-up*, and *Recovery* phases throughout the rest of the paper.

1. **Subtree phase** (also refered to as *Top-down phase*): In this phase, the algorithm searches for large, identical subtrees between the two input trees, starting from the roots and proceeding downward. The goal is to find high-level structural similarities as early as possible. The result of this phase is a set of initial *"anchor"*

mappings, which form the foundation for further matching.

2. **Bottom-up phase**: This phase augments the anchor mappings by identifying *"container" mappings* between nodes whose subtrees already contain many matching descendants. Nodes are traversed in post-order, and mappings are determined based on a similarity score (e.g., Dice coefficient) between the sets of their mapped descendants. This process identifies higher-level structural correspondences that may not have been captured in the top-down phase.
3. **Recovery phase**: Once a mapping is established in the bottom-up phase, this phase applies a fine-grained matching algorithm within the descendants of those nodes. The goal is to identify additional *"recovery" mappings*, capturing smaller-scale edits and refinements missed in the earlier phases.

3.3 Existing algorithms

The problem of computing differences between trees has a long and diverse history with numerous variants each addressing different aspects of the problem. Most of the variants differ by the set of allowed actions. For instance, MH-DIFF [14] prioritizes meaningfulness of the edit scripts and therefore allows a richer set of actions. On the other hand, some variants like RTED [15] focus on the edit scripts being optimal which in turn enforces a very basic set of actions. Other approaches focus on differencing unordered trees, where the order of sibling nodes is not semantically important [16]. Our work focuses on a particular branch of tree differencing algorithms that have specific constraints on the structure and prioritize performance over optimality, making them more practical for Abstract Syntax Tree differencing.

To our knowledge, the first influential algorithm in this line of work was the Xy algorithm, originally introduced as BULD algorithm by Cobéna et al. [4] and designed for XML differencing. Despite its simplicity, Xy introduced a powerful and flexible multi-stage structure which was the main inspiration for the generalized three-stage framework. Xy's architectural foundation enabled subsequent algorithms to achieve incremental improvements through targeted enhancements to specific phases, while its simplicity and generalizability made it easily adaptable to diverse programming language syntaxes.

Indeed, successive approaches build upon Xy by refining different parts of the algorithms to improve different aspects like performance, quality, or memory consumption. In 2014, Falleri et al., the creators of GumTree algorithm [5] made an influential refinement to the algorithm and published a widely available tool for code change detection which was a significant milestone. In the case of GumTree, the big change was made to the recovery phase. First of all, GumTree uses an optimal algorithm to find recovery mappings. To limit the performance overhead caused by the algorithm, GumTree only uses the algorithm if the size of the matched subtrees is smaller than a predefined threshold `MAX_SIZE`.

A recent approach, HyperDiff [7], further improves the GumTree algorithm. It leverages a compressed representation of the AST, called HyperAST [17], to simplify the process of identifying identical subtrees.

Parallel to HyperDiff, Falleri et al. introduced an improved version of GumTree, called GumTree Simple [6]. The authors identified a critical limitation in the original GumTree algorithm: the use of an optimal algorithm for the recovery phase, which introduces significant performance overhead and requires the manual tuning of a `MAX_SIZE` threshold to remain tractable. To address this, they propose a new heuristic-based recovery strategy called *simple recovery*. This revised method prioritizes practicality, aiming to significantly reduce computational cost while maintaining comparable result quality.

4 Methodology

To provide reproducible, and statistically sound results, our methodology standardizes the benchmarking environment, uses a well-established dataset, and applies rigorous statistical analysis. We eliminate implementation bias by reusing a common framework and aligning with practices from prior work, such as HyperDiff [7]. The evaluation relies on datasets widely adopted in the literature and employs the Criterion.rs [18] benchmarking suite. The following subsections detail our benchmarking setup, describe the dataset, and finally outline the statistical relevance of our results.

4.1 Environment

To minimize bias in results related to differences in programming languages and testing setups, we followed an approach similar to HyperDiff. First, we ported an existing Java implementation of the algorithms from GumTree repository to the HyperAST repository. Then, we replicated the setup used by HyperDiff, only replacing the matching algorithm with a ported implementation. That way, all benchmarked algorithms use the same framework with exactly the same underlying data structure.

4.2 Dataset

The evaluation re-used the dataset employed in the GumTree paper [19; 6]. More specifically, we evaluated each algorithm on a collection of commits from the following two large open-source real-world Java projects:

- *GitHub Java* is a dataset containing 1000 commits from 10 popular projects.
- *Defects4J* is a dataset of bug fixes used in the program repair community.

In total, the datasets contain 2045 pairs of files. The `before` folders contain the files before modification, and the `after` folders contain the files after. Inside the `before` and `after` folders, there is one folder per project that contains one folder per commit. Note that the commit names are the same in the `before` and `after` folders.

4.3 Evaluation protocol

Because the benchmarks are targeted at multiple algorithms, and focus on particular phases, they were designed with flexibility in mind. To make the benchmarks easily rerunnable in different configurations, we decided to split them into three stages that can be run independently. To avoid running computation-heavy stages multiple times, output from every stage is saved to the filesystem so that intermediate results can be reused.

1. Benchmark: In the first stage, we isolate one phase (or a set of phases) of a given algorithm and measure the runtime using Criterion, as described in more detail in 4.4 Statistical relevance section. For every file pair in the dataset, we collect an estimate for mean runtime together with its 95% confidence interval.
2. Metadata: This stage is aimed at obtaining additional data from the algorithm's execution. To compute , we run all phases and collect the number of matches and the length of the edit script after each phase. We later use these values to compute the number of matched nodes per phase and the deltas of edit script lengths for each phase.
3. Summarize: The final stage combines and organizes all data obtained in previous stages into one JSON file per algorithm. The data from the JSON file is then visualized using `Matplotlib v3.10.3` and `Seaborn v0.13.2` Python libraries.

4.4 Statistical relevance

To support the reproducibility and reliability of our performance measurements, we employ the Criterion.rs [18] benchmarking framework, which is specifically designed to produce statistically sound results. Criterion structures benchmarking into three distinct phases: *warm-up*, *measurement*, and *analysis*, each contributing to the reliability of the final outcome.

During the *warm-up phase*, the code is executed repeatedly to mitigate the impact of transient runtime effects such as CPU throttling or just-in-time optimizations. This helps stabilize the system before any measurements are taken. In the *measurement* phase, Criterion adaptively determines the appropriate number of iterations to run, ensuring that enough data is gathered to support statistically meaningful outputs. Finally, in the *analysis* phase, Criterion uses bootstrap resampling to estimate confidence intervals and detect statistically significant differences. It also performs outlier detection and removal, further improving the quality of the results.

This methodology allows us to report not only mean execution times but also the variance and confidence bounds, providing a more reliable and informative comparison of runtimes.

5 Evaluation

Before we dive into the analysis of results, we first explain the benchmarks used to answer our research questions, including the key methodological decisions made to ensure the clarity and interpretability of our results. Our evaluation is structured around the three core research questions, each corresponding to a specific refinement in AST differencing algorithms. At the start of each subsection, we briefly reintroduce the corresponding research questions to guide the interpretation of the experiments.

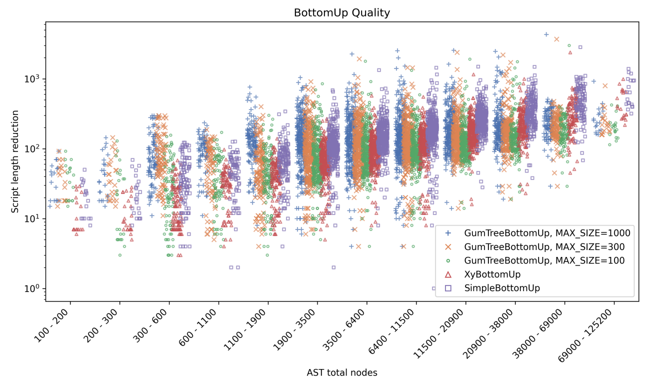

Figure 1: Bottom-up phase edit script improvement by algorithm

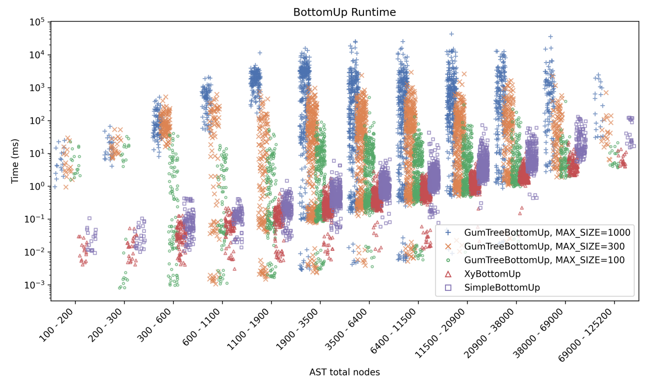

Figure 2: Bottom-up phase runtime by algorithm

5.1 RQ1, RQ2: Bottom-up phase analysis

The first two research questions relate to successive refinements of the bottom-up phase:

RQ1 asks *how the refinement of Xy's recovery strategy in GumTree affects the trade-off between script quality and performance when applied to complex ASTs.*

RQ2 asks *to what extent the simple recovery improves upon GumTree in terms of scalability, runtime stability, and edit script quality for large ASTs.*

Since Both questions are focused on analysing a refinement to the recovery (which is a part of the bottom-up phase), we will isolate the bottom-up phases of the three algorithms addressed in these questions (XyBottomUp, GumTreeBottomUp, SimpleBottomUp) and compare them all together. We made the choice to focus solely on the bottom-up phase (which includes the recovery), as the subtree phase has a smaller impact on overall runtime. Moreover, the subtree phase is identical for both algorithms, making it less relevant for comparative analysis.

To support the analysis of the runtime-quality tradeoff, we provide plots of runtime and quality. The quality is measured as *script length reduction*, which we define as the difference between the lengths of edit scripts before and after the bottom-up phase. In other words, script length reduction tells how much shorter the scripts became thanks to the matches found in the bottom-up phase (Figure 1). This way, we can

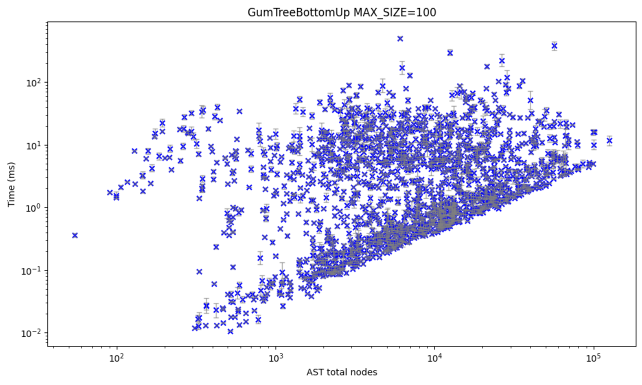

Figure 3: GumTree Bottom-up phase runtime w.r.t AST size

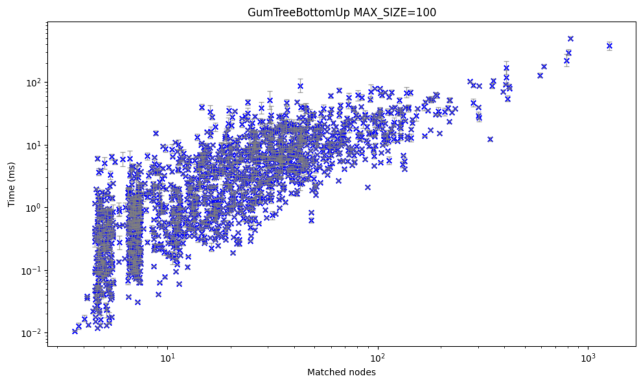

Figure 4: GumTree Bottom-up phase runtime w.r.t. #matches

isolate the impact of the refinement on that particular phase. For the runtime, we show an estimate of the mean execution time in milliseconds w.r.t. the total number of nodes in source and destination ASTs (Figure 2).

As it turns out, GumTree's runtime cannot be easily explained w.r.t. the number of nodes. (Figure 3). This stems from the fact that runtime is not only dependent on the size of the trees but also strongly influenced by other structural properties. To better understand which properties impact runtime the most, we measure the Pearson correlation coefficient between runtime and several basic properties across three algorithm variants (Table 1):

- *AST total nodes* - total number of nodes in both trees
- *Unmatched (subtree)* - number of unmatched nodes in the input to the bottom-up phase
- *Unmatched (bottom-up)* - number of unmatched nodes remaining after the bottom-up phase
- *Matched nodes* - number of nodes matched during bottom-up phase

Metric	**GumTree 500**	**GumTree 100**	**Xy**
AST total nodes	0.014	0.144	0.710
Unmatched (subtree)	0.257	0.507	0.556
Unmatched (bottom-up)	0.062	0.404	0.526
Matched nodes	0.625	0.769	0.622

Table 1: Pearson correlation between runtime and selected metrics

In case of the GumTreeMatcher, we observe the highest correlation with the number of matched nodes. This makes sense as more matched nodes imply that more effort has been done by the expensive optimal recovery algorithm (Figure 4). Based on these findings, we provide one more visualization of the runtime w.r.t. number of matched nodes (given by baseline GumTreeMatcher with `MAX_SIZE=1000`) to be the independent variable on the x-axis (Figure 5).

As we will see later, the `MAX_SIZE` parameter of the GumTreeMatcher also has a significant impact on both performance and result quality. To investigate how this param-

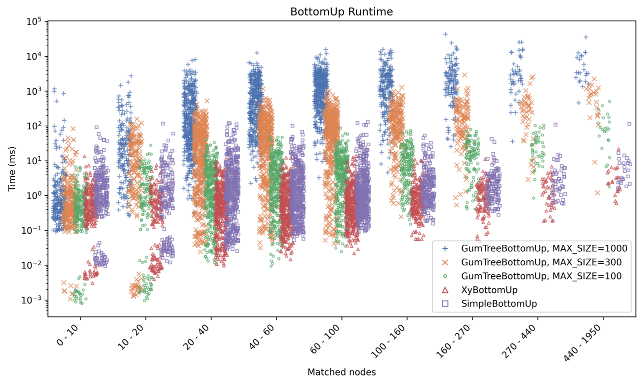

Figure 5: Bottom-up phase runtime by algorithm (w.r.t. #matches)

eter affects outcomes, we repeat the benchmarks using three different values of `MAX_SIZE`: 100, 300, and 1000.

5.2 RQ3: Subtree phase analysis

RQ3 asks *how the use of the HyperAST data structure in HyperDiff impacts performance and scalability compared to GumTree, particularly in cases with a varying number of changes between ASTs.*

For this research question, we chose to isolate the subtree phase. The main reason is that the refinement in HyperDiff has the greatest impact on this particular phase. This is due to HyperDiff's use of a compressed representation of the AST, which drastically simplifies node matching during the subtree phase.

As in RQ1 and RQ2, we compare the runtimes of HyperDiff and GumTree w.r.t the total number of nodes (Figure 6). On top of that, we conduct the same correlation analysis as with GumTree in the previous section (Table 2)

In the case of subtree matchers, the largest correlation can be observed with nodes unmatched after the subtree phase. Therefore, we will provide an additional visualization of the runtime w.r.t. that metric (Figure 7).

In the analysis for RQ3, we will skip the quality comparison as the same underlying procedure is used in both algorithms. HyperDiff utilizes a compressed data structure and introduces several language-specific optimizations, but it does

Metric	GumTree Subtree	HyperDiff Subtree
AST total nodes	0.329	0.038
Unmatched (subtree)	0.653	0.582
Unmatched (bottom-up)	0.588	0.520
Matched nodes	0.426	0.388

Table 2: Pearson correlation between runtime and selected metrics

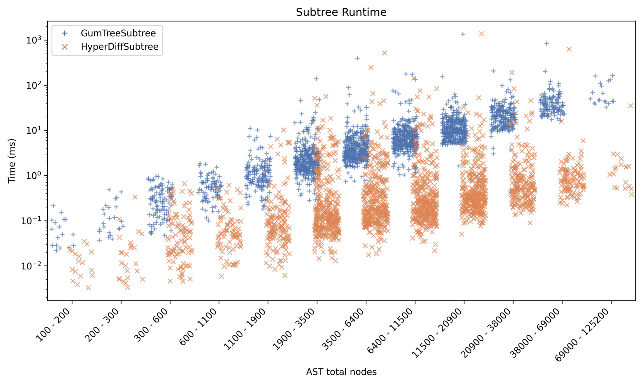

Figure 6: Subtree phase runtime by algorithm

not change the core algorithm, and therefore produces the same set of mappings as GumTree.

5.3 Additional considerations

To improve the readability of the visualizations, the data has been divided into vertical bins. Within each bin, datapoints corresponding to different algorithms have been grouped and plotted using different colors. Moreover, a small horizontal jitter has been applied to the datapoints to reduce overlap. As a result, the x-coordinates of the datapoints are not perfectly exact; we only guarantee that each point falls within the bounds of its respective bin.

Additionally, a logarithmic scale is used in all of our plots to emphasize asymptotic trends and to spread datapoints corresponding to smaller inputs more evenly along the x-axis.

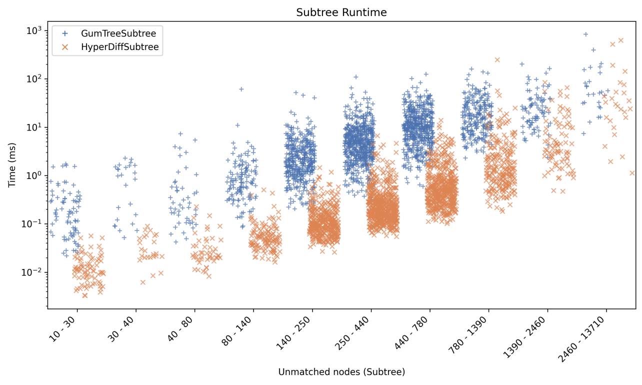

Figure 7: Subtree phase runtime by algorithm (w.r.t. #unmatched nodes)

6 Results and discussion

The primary goal of this paper is not to assess the performance of algorithms in isolation, but rather to examine them together and analyze the impact of successive enhancements made to the algorithms throughout the years. Our findings are supported by a series of comparative graphs and plots, which illustrate the quantitative effects of each enhancement across a set of benchmarks.

We focus on examining how each revision affected behavior, performance, and trade-offs. For each pair, we justify the design decisions involved and provide a detailed evaluation of their consequences, both positive and negative.

6.1 RQ1: GumTree as Xy's adaptation for code

In terms of edit script quality, GumTree significantly improves upon Xy, especially for smaller ASTs. (Figure 1) This is mainly due to the difference in how the recovery phase is handled: Xy employs a naive recovery strategy that only considers immediate descendants of already matched nodes when searching for additional matches. While this approach is effective for small, local edits (eg. renames or variable replacements) it fails to capture more sophisticated changes involving structural or logical transformations in the code.

This limitation comes from Xy's original design, which targeted XML documents that are generally simpler and do not express program logic. Although the adaptation used in our experiments supports source code, the core recovery strategy is, in most cases, insufficient for more complex structures in code. GumTree addresses this issue by performing a deeper inspection of subtrees during the recovery phase. This enables it to detect more complex logical modifications, achieving a significantly higher number of mappings.

It is important to note that the gap in the script length narrows as tree sizes increase. This is because, for efficiency reasons, GumTree disables the recovery phase on trees larger than the configured `MAX_SIZE`. As a result, with increasing tree size, more recoveries are skipped, leading to diminishing improvements in edit script length. In extreme cases, with trees of more than 50000 nodes, Xy may even produce slightly better results.

The gains in quality are substantial, but they come at a considerable cost. GumTree not only traverses all nodes in the subtrees, but it also runs a $\mathcal{O}(n^3)$ algorithm every time a recovery phase is triggered. This causes a huge overhead for the overall runtime compared to the naive Xy approach. As shown in figure 2, even with relatively low `MAX_SIZE`=100, GumTree can take up to 100 times longer to execute than Xy. For large `MAX_SIZE`=1000, execution time can even increase by a factor of 10000.

As a consequence, `MAX_SIZE` should be chosen with great care. Depending on quality expectations and available resources, different values will be suitable. On the one hand, this is an advantage as `MAX_SIZE` allows us to tweak the algorithm for a particular use case. On the other hand, choosing the right value might be an additional burden on the user, which was addressed in the papers on GumTree [5; 20].

RQ1 Summary: The refinement of Xy's recovery strategy in GumTree Greedy leads to notable improvements in the quality of the edit script, particularly for small and medium ASTs. By performing a more thorough recovery process, GumTree captures more complex structural and logical edits that Xy's naive local-only strategy often misses.

However, these quality gains come at a significant performance cost. GumTree's recovery involves a computationally expensive algorithm, resulting in runtime overheads of up to 10,000× compared to Xy in extreme cases.

Despite the cost, GumTree's refinement enhances the practical usability of the algorithm by enabling a sophisticated matching approach to handle more complex, real-world ASTs.

6.2 RQ2: Scalable recovery for GumTree

In terms of performance, SimpleBottomUp drastically improves the runtime of GumTreeBottomUp by 1-2 orders of magnitude across all input sizes (Figure 2). This is expected, as SimpleBottomUp avoids launching an expensive algorithm on large subtrees. Although GumTree's asymptotic runtime growth slows down for trees exceeding 1000 nodes thanks to the `MAX_SIZE` threshold, it remains two orders of magnitude worse on the largest inputs.

Another aspect where Simple proves better than greedy is the runtime stability. Similar to Xy, Simple's runtime increases consistently with input size. In contrast, as shown in figure 5 GumTree's runtime is highly sensitive to factors such as tree structure and the number of changes in the input. While GumTree tends to slow down with more differences in the ASTs, Simple remains consistent and is therefore more reliable for diverse and change-heavy inputs.

Because the recovery phase is completely skipped on larger subtrees, GumTree hits a limit in script length once the tree size crosses a certain value (Figure 1). While this value depends on the `MAX_SIZE` setting, trees of size about 2600 and above reach a ceiling effect in script length for all tested `MAX_SIZE`. The Simple heuristic doesn't have this limitation, and its edit scripts continue to improve even for larger tree size. As a result, for trees larger than 5,600 nodes, Simple begins to outperform GumTree in edit script quality. As shown in figure 5, this quality gap widens with increasing input size.

Overall, while Simple matcher yields slightly lower-quality results than GumTree on small trees, its performance does not fall back on larger trees, which is a significant scalability improvement. Simple provides stable and predictable growth in both runtime and script quality, which allows it to surpass GumTree not only in efficiency but also in result quality for large datasets, making it a more practical choice for real-world repositories containing thousands of commits and millions of nodes in the trees.

RQ2 Summary: The Simple matcher outperforms the GumTree matcher in scalability, runtime stability, and script quality on large and complex ASTs. It is orders of magnitude faster across all input sizes and maintains consistent performance, unlike GumTree, which is sensitive to structural differences and slows down on trees with more edits.

While GumTree's script quality slows down on large trees due to its skipped recovery phase, Simple continues to improve and surpasses GumTree for trees over 5,600 nodes. Overall, Simple's refinements significantly improved the reliability and efficiency for real-world scenarios involving large codebases and thousands of commits.

6.3 RQ3: Impact of HyperAST on GumTree

Looking at figure 6 we observe a significant difference in the runtime of the subtree matchers. Even on the smallest inputs, HyperDiff outperforms GumTree by an order of magnitude. This performance gap consistently increases with larger datasets, reaching up to two orders of magnitude on the largest datasets.

However, while HyperDiff shows strong average performance, the algorithm behaves similarly to GumTree in the worst case, as indicated by a number of outliers. As shown in figure 7, HyperDiff's runtime approaches that of GumTree as the number of unmatched nodes increases. The outliers correspond to ASTs with the highest number of unmatched nodes, likely caused by a large number of changes in the input files.

This effect can be explained by the characteristics of the HyperAST data structure which HyperDiff conveniently exploits. HyperAST is a compressed representation of the trees in which identical subtrees share the same memory reference to reduce redundancy. This property allows HyperDiff to recognize identical subtrees without having to traverse them. Naturally, the runtime benefits depend on the number of subtrees that can be matched and therefore skipped. However, in the worst case, when the number of differences between the trees is large, HyperDiff must traverse most of the tree, just like GumTree, which diminishes the performance gain in such case.

RQ3 Summary: Our results show that HyperDiff significantly improves GumTree in most practical scenarios. Specifically, on 80% of the tested inputs, HyperDiff is at least 5.8x faster, and on 60% of inputs, it achieves a speedup of 12.6x or more. Most importantly, HyperDiff achieves these improvements without consequences on the accuracy, thus substantially improving the runtime-accuracy tradeoff.

7 Responsible Research

Even though this research does not involve human subjects, personal data, or sensitive content, it is still important to address reproducibility, ethical use of resources, and potential societal impact.

7.1 Reproducibility and Availability

To ensure reproducibility, all evaluated algorithms were implemented or ported into a shared and publicly available benchmarking environment. All code and data used in this study are made publicly available (see Appendix B). All source code is based on open source repositories, and the original datasets are also open and accessible. These datasets are widely adopted and documented, which allows easy validation and replication in future research.

7.2 Ethical Use of Computational Resources

The main goal of this research was to improve the understanding of AST-based differencing algorithms by examining their computational requirements and limitations. By providing insights into these factors, the study contributes to future improvements in the efficiency of such algorithms. The knowledge obtained in this research has the potential to reduce the energy footprint of code versioning systems, thereby promoting more sustainable use of computational resources in software development workflows.

7.3 Societal Impact

The techniques explored in this paper have the potential to improve tools used in software development, such as version control systems, automated code review tools, and bug detection frameworks. Through more accurate and efficient structural change detection, these tools may contribute to better software quality, reliability, and maintainability.

8 Conclusion

This research analyzed the evolution of structural AST differencing algorithms by isolating and benchmarking three key refinements: GumTree's optimal recovery, GumTree Simple's recovery heuristic, and HyperDiff's use of a compressed AST representation. Through systematic evaluation, we demonstrated how each refinement advances the trade-off between runtime and edit script quality. GumTree improved script accuracy at a significant performance cost, while Simple achieved better scalability and runtime stability without compromising quality on large ASTs. HyperDiff further optimized subtree matching by leveraging redundancy in real-world trees, offering substantial performance gains without impacting accuracy. Together, these findings highlight the practical impact of targeted heuristic refinements and offer insights into future improvements for scalable and interpretable code differencing tools.

9 Acknowledgements

This research was carried out as part of a group project with four other students. Some aspects of the implementation were conducted collaboratively and shared among group members. We would like to thank our supervisor, Dr. Carolin Brandt, and our course coordinator, Dr. Quentin Le Dilavrec, for their valuable guidance and support throughout the project.

A Use of generative AI

Generative AI tools were used solely to support the writing process of this report. Specifically, a language model was used to improve the clarity and structure of sentences, and to improve the organization of sections. No AI assistance was used during the design or execution of experiments, nor in the analysis or interpretation of results. All findings and conclusions presented in this report are the result of independent work. The use of AI adhered strictly to course guidelines and did not replace any critical thinking or analytical work required for the project.

B Reproducibility Details

The full implementation and all benchmarking and plotting scripts used in this project are available at: https://github.com/HyperAST/HyperAST/pull/73

For reproducibility, refer to commit `cdf5cffaeaf32c8b9ee2577d63f4fcc8c3b69cdf`, available at: https://github.com/HyperAST/HyperAST/pull/73/commits/cdf5cffaeaf32c8b9ee2577d63f4fcc8c3b69cdf

References

[1] E.S. Ristad and P.N. Yianilos. Learning string-edit distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(5):522–532, May 1998.

[2] Eugene W. Myers. AnO(ND) difference algorithm and its variations. *Algorithmica*, 1(1):251–266, November 1986. Company: Springer Distributor: Springer Institution: Springer Label: Springer Number: 1 Publisher: Springer-Verlag.

[3] 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.

[4] 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.

[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] 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.

[7] 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.

[8] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. Towards practical program repair with on-demand candidate generation. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 12–23, New York, NY, USA, May 2018. Association for Computing Machinery.

[9] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 802–811, San Francisco, CA, USA, May 2013. IEEE Press.

[10] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bisyandé. AVATAR : Fixing Semantic Bugs with Fix Patterns of Static Analysis Violations, February 2019. arXiv:1812.07270 [cs].

[11] Kathryn T. Stolee, Sebastian Elbaum, and Daniel Dobos. Solving the Search for Source Code. *ACM Trans. Softw. Eng. Methodol.*, 23(3):26:1–26:45, June 2014.

[12] Ghulam Rasool and Zeeshan Arshad. A review of code smell mining techniques. *Journal of Software: Evolution and Process*, 27(11):867–895, 2015. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.1737.

[13] 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.

[14] Sudarshan S. Chawathe and Hector Garcia-Molina. Meaningful change detection in structured data. In *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, SIGMOD '97, pages 26–37, New York, NY, USA, June 1997. Association for Computing Machinery.

[15] Mateusz Pawlik and Nikolaus Augsten. RTED: A Robust Algorithm for the Tree Edit Distance, December 2011. arXiv:1201.0230 [cs].

[16] Kaizhong Zhang, Rick Statman, and Dennis Shasha. On the editing distance between unordered labeled trees. *Information Processing Letters*, 42(3):133–139, May 1992.

[17] 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.

[18] Jorge Aparicio and Brook Heisler. criterion.rs: Statistics-driven micro-benchmarking library. https://github.com/bheisler/criterion.rs. Accessed: 2025-06-02.

[19] Jean-Rémy Falleri et al. Gumtree datasets. https://github.com/GumTreeDiff/datasets, 2024. Accessed: 2025-06-02.

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