Accelerating AST-Based Code Differencing

Optimizing ChangeDistiller's Bottom-Up Matching Strategy with HyperAST

Leo Mangold[1]

Supervisors: Carolin Brandt[1]**, Quentin Le Dilavrec**[1]

[1]**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: Leo Mangold
Final project course: CSE3000 Research Project
Thesis committee: Carolin Brandt, Quentin Le Dilavrec, Jesper Cockx

Abstract

Traditional AST-based code differencing tools like *ChangeDistiller* struggle to scale on large codebases. *HyperAST* is a framework that models versioned code as a Directed Acyclic Graph (DAG) of Abstract Syntax Trees (ASTs), with deduplication of unchanged nodes and precomputed metadata. This approach has demonstrated effectiveness in improving the performance of the *GumTree* algorithm. However, its applicability to algorithms with fundamentally different matching strategies, like *ChangeDistiller*'s bottom-up approach, was unclear. We ported *ChangeDistiller* to Rust and adapted it to leverage *HyperAST*'s optimizations. Experiments on 1,046 real-world code change pairs from the Defects4J dataset demonstrate a 99.13% reduction in total runtime (~4.5 hours to 2.3 minutes) and a median per-file reduction of 98.97% (3149.25 to 31.55 milliseconds), all without altering the core algorithm's behavior. Our research demonstrates that *HyperAST*'s techniques can be applied beyond *GumTree*, significantly improving *ChangeDistiller*'s runtime performance.

1 Introduction

Understanding how software changes over time is fundamental to computer science. Developers make thousands of modifications daily—fixing bugs, adding features, refactoring, and optimizing performance. Tracking and analyzing these changes is crucial for software engineering tasks such as code review, debugging, and version control.

Traditional tools like Unix `diff`[1] compare files line-by-line as plain text, missing crucial structural information. For example, moving a function appears as a complete deletion and insertion rather than a relocation. To address these limitations, *structural differencing* tools operating on Abstract Syntax Trees (ASTs) were developed. Diff tools like *ChangeDistiller* [1] and *GumTree* [2] operate on ASTs to preserve the syntactic structure and facilitate a semantic understanding of code changes, thereby enabling more accurate code analysis.

However, these structural differencing tools face a critical scalability challenge when used to analyze entire codebases. Their focus mainly lies on differencing between two files, but when we want to understand more about the evolution of codebases over time, the computational cost of comparing ASTs becomes prohibitively expensive. Processing large codebases can take hours or even become computationally infeasible, limiting the practical applicability of these otherwise powerful tools.

Efforts to address these challenges have taken different directions. Some approaches, such as pruning ASTs based on textual diff-hunks [3], aim to improve performance by reducing the input size. While effective, pruning methods focus on pre-filtering rather than enhancing the efficiency of the underlying data structures. In contrast, Quentin et al. in *HyperDiff* [4] adapted *GumTree* to utilize *HyperAST* [5], a framework designed for optimized AST representation and processing at scale. By incorporating techniques such as deduplication, lazy processing, and precomputed metadata, this adaptation achieved substantial performance gains, demonstrating the value of data structure optimizations for AST differencing.

Other research has focused on improving the *quality* of edit scripts generated by AST differencing tools. For instance, tools like *IJM* [6], *MTDiff* [7], and hybrid methods that combine AST and textual data [8] aim to produce more concise and intuitive scripts that align with developer intent. Hyperparameter optimization of *GumTree* has also been demonstrated to be effective in improving quality [9]. However, these efforts primarily refine matching algorithms and output formats rather than addressing performance.

Similarly, refactoring detection tools, such as *RefactoringMiner* [10] and Alikhanifard et al.'s approach [11], leverage AST analysis to identify code transformations. While improving the performance of their algorithms to some extent, their focus remains on detecting refactoring types rather than optimizing AST differencing performance at scale.

Building on these advancements, this research investigates whether *HyperAST*'s innovations can be transferred to *ChangeDistiller*, which employs a bottom-up matching approach distinct from *GumTree*'s hybrid strategy. Specifically, we address two main research questions: (1) **Which *HyperAST* optimization techniques can be effectively adapted to the *ChangeDistiller* algorithm?** (2) **How does the adapted *ChangeDistiller's* runtime performance compare to the original algorithm?**

This paper outlines our strategy for adapting *ChangeDistiller* to take advantage of *HyperAST*'s optimizations. We ported *ChangeDistiller* to Rust and integrated it with the optimized data structures and techniques from *HyperAST*. Our approach maintains the core bottom-up matching strategy and coarse-grained processing of *ChangeDistiller*, while achieving significant performance improvements.

We conducted a comprehensive empirical evaluation across 1,046 real-world code change pairs from the Defects4J dataset [12], demonstrating substantial runtime improvements. Our *HyperAST*-adapted *ChangeDistiller* achieved a median speedup of 94.2× while preserving the essential algorithmic behavior, demonstrating that *HyperAST*'s optimizations are effective even for algorithms with distinct characteristics.

The remainder of this paper is structured as follows: Section 2 provides background on AST-based differencing and the *HyperAST* framework. Section 3 details our adaptation methodology and technical innovations, including configurable processing strategies and caching optimizations. Section 4 presents our comprehensive evaluation and its analysis. Section 5 concludes and outlines future work. Finally, Section 6 addresses responsible research aspects.

[1] https://www.man7.org/linux/man-pages/man1/diff.1.html

2 Background

Structural differencing techniques operating on ASTs enable semantically meaningful change detection by analyzing the syntactic structure of code. These techniques compute edit scripts that capture insertions, deletions, updates, and structural moves by mapping corresponding nodes between two ASTs–often done in multiple phases–and generating a sequence of operations to transform one AST into another.

Two prominent AST-based differencing tools, *ChangeDistiller* [1] and *GumTree* [2], employ distinct strategies to compute edit scripts. *ChangeDistiller* uses a bottom-up matching approach inspired by Chawathe et al.'s algorithm [13]. It begins with a leaf-matching phase, where noncompound statement nodes are matched using string similarity based on bi-grams and the *Dice Coefficient* [14]. This is followed by an inner-node matching phase, where subtree similarity is computed based on previously matched children. *ChangeDistiller* operates on coarse-grained ASTs, meaning that the leaves of the ASTs represent statement level nodes as their string representation.

In contrast, *GumTree* employs a hybrid strategy combining top-down greedy matching, bottom-up matching for remaining nodes, and a recovery phase. Operating on fine-grained ASTs, *GumTree* produces shorter edit scripts and detects structural moves more effectively [2].

To address the scalability challenges of differencing large-scale software systems, frameworks like *HyperAST* [5] have been developed. *HyperAST* optimizes AST representation and processing by modeling software history as a Directed Acyclic Graph (DAG) of ASTs. Identical subtrees are deduplicated across versions and within a single version, reducing memory usage and computational overhead. Techniques such as lazy decompression—where DAG nodes are materialized only on demand—and precomputed node metadata further enhance efficiency.

3 Approach

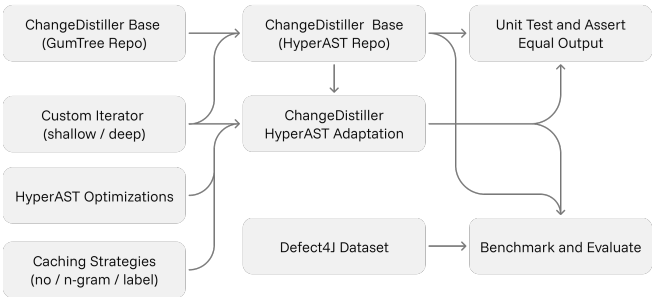
This section describes our methodological approach to adapting *ChangeDistiller* to leverage *HyperAST*'s optimized data structures and presents the technical innovations that enable improved performance while preserving the original algorithmic approach.

3.1 Adaptation Strategy

The primary challenge in adapting *ChangeDistiller* to *HyperAST* lies in reconciling fundamentally different design philosophies. *ChangeDistiller* employs a bottom-up matching approach on coarse-grained ASTs, treating statement-level nodes as atomic units, whereas *HyperAST* stores complete fine-grained AST structures. Our adaptation strategy preserves *ChangeDistiller*'s original algorithmic logic while leveraging *HyperAST*'s performance optimizations.

We ported *ChangeDistiller* from *GumTree*'s Java repository[2] to Rust to mitigate any differences in performance caused by language differences. We preserved its default hyperparameters and core two-phase algorithm: leaves matching using Dice bi-gram string similarity, followed by inner node matching based on subtree similarity. This baseline implementation serves as the foundation for evaluating our optimizations. Figure 1 illustrates the general outline of our work.

Figure 1: General Outline of the Process of our Approach



To assess the effectiveness of different optimizations, we implemented eight variants: two baseline configurations (shallow and deep statement processing) and six optimized configurations that combine both statement processing approaches with caching strategies (no cache, n-gram cache, and label cache). This design isolates the impact of individual techniques and their interactions.

3.2 Technical Implementation

The following three elements form the core of our implementation: a custom iterator for coarse-grained processing on fine-grained ASTs, integration of *HyperAST*'s optimizations, and caching mechanisms for string similarity calculations.

3.2.1 Coarse-Grained Statement Processing on Fine-Grained ASTs

The original *ChangeDistiller* algorithm operates on coarse-grained ASTs where "leaves in the tree are noncompound statements" [1], while *HyperAST* stores complete fine-grained ASTs containing fully parsed expressions.

We introduced a custom post-order iterator with configurable leaf predicates, allowing us to specify which nodes to treat as logical leaves. This enables *ChangeDistiller* to operate at the statement level without altering *HyperAST*'s internal structure. With that, we developed two processing approaches:

Deep Statement Processing treats the deepest noncompound statement-level nodes as logical leaves, closely matching *ChangeDistiller*'s original definition.

Shallow Statement Processing treats statement nodes closest to the root as logical leaves, grouping compound statements as single units.

To illustrate these differences, consider the simple function in Listing 1 and its AST representation in

[2]https://github.com/GumTreeDiff/gumtree/tree/5b939f8

Listing 1: Simple Example Java Function corresponding to Figure 2

```
function foo(int i) {
  if (i < 0) {
    return false;
  }
  return i + 10;
}
```

Figure 2: AST representation of Listing 1 highlighting the logical leaf nodes of different iteration modes. **Legend:** *Blue fill* = Fine-grained terminal nodes; *Orange border* = HyperAST shallow logical leaves; *Light blue fill* = HyperAST deep logical leaves; *Dashed border* = Coarse-grained logical leaves. Nodes with combined styles are processed by multiple approaches.

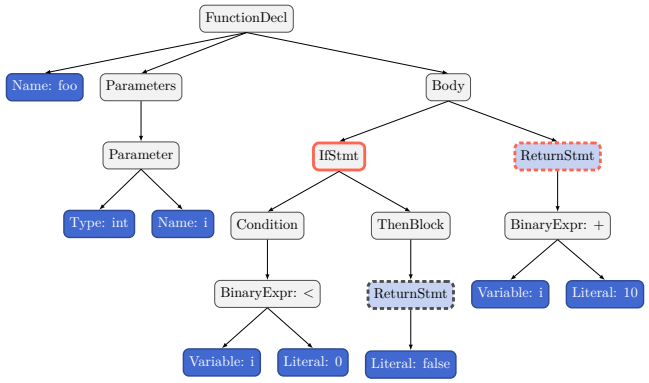


Figure 2, which demonstrates how *HyperAST* stores the complete fine-grained AST structure. Our custom iterator treats different nodes as logical leaves based on the configured processing approach.

Fine-grained processing (blue fill) identifies all actual terminal nodes in the AST - the atomic elements like identifiers (`Name: foo`, `Variable: i`) and literals (`Literal: 0`, `Literal: false`, `Literal: 10`) that contain no child nodes. This is the default behavior of *HyperAST* and is equivalent to our custom iterator with a leaf predicate that returns true only if the node has no children.

Our **shallow statement iterator** (orange border) treats the outermost statement-level constructs as logical leaves: the entire `IfStmt` compound statement and the outer `ReturnStmt`. This approach prioritizes higher-level structural units over individual statements within compound blocks. This behavior is achieved by setting a leaf predicate that returns true only if the node is of type `Statement` and setting the iterator to shallow mode, which means it will not descend into child nodes.

Our **deep statement iterator** (light blue fill) identifies the deepest non-compound statement-level nodes as logical leaves: both `ReturnStmt` nodes (`return false;` and `return i + 10;`), but not the compound `IfStmt` that contains one of them. This more closely matches the original *ChangeDistiller* algorithm's focus on individual statement units. This behavior is achieved by setting a leaf predicate that returns true only if the node is of type `Statement` and has no descendant node of type `Statement`, ie. setting the iterator to deep mode.

Iterating over a **Coarse-grained AST** (dashed border) yields the same leaves as a deep iterator. However, it would iterate over ASTs where the statement nodes are terminal nodes with string representations of the entire statement rather than ASTs containing their fine-grained parsed structure. We mimic this behavior by treating statement-type nodes as logical leaves during traversal and using their serialized string representation (a string containing the entire statement) rather than a node's label for string similarity.

Custom iterators enable us to utilize *HyperAST*'s data representation without modification while maintaining *ChangeDistiller*'s coarse-grained analysis strategy. Additionally, the iterator provides traversal modes to return all nodes (up to the logical leaves), only the logical leaf nodes, or only the inner nodes, enabling efficient access for leaves matching and bottom-up matching.

3.2.2 Core HyperAST Optimizations Integration

We successfully integrated three fundamental *HyperAST* optimization techniques into all of the optimized variants of our *ChangeDistiller* implementation:

Optimized Data Structures HyperAST stores AST nodes in post-order traversal within contiguous arrays, where each node's identifier corresponds directly to its array index, and all descendants of a node occupy positions from the node's leftmost leaf descendant up to the node itself. We adapted *ChangeDistiller* to leverage this structure for efficient data retrieval and range-based operations.

Lazy Decompression The framework implements lazy decompression of *HyperAST*'s DAG nodes, materializing post-order arrays only on demand rather than computing all traversals upfront. We modified *ChangeDistiller*'s tree access patterns to defer computational costs until nodes are needed during algorithm execution.

Hash-Based Preliminary Matching HyperAST stores precomputed label hashes as metadata for all nodes. We integrated these hashes into *ChangeDistiller*'s matching phases as a preliminary filtering step. When two nodes share identical label hashes, they represent semantically equivalent subtrees and can be immediately matched without computing expensive similarity scores. When label hashes differ, the algorithm falls back to the original n-gram similarity computation, ensuring no potential matches are missed while significantly reducing computational overhead.

Algorithm 1 presents our optimized leaves matching phase. First, it collects all logical leaf nodes from T_1 and T_2 through our custom post-order iterator. Then, for each unmatched leaf in T_1, it seeks a corresponding unmatched leaf in T_2. Line 11 shows the preliminary matching–nodes

Algorithm 1 Optimized Leaves Matcher Phase. Changes to the Original Highlighted in Blue.

```
1: L1 ← collectLeaves(T1)
2: L2 ← collectLeaves(T2)
3: for all t1 ∈ L1 do
4:     if alreadyMatched(t1) then
5:         continue
6:     end if
7:     for all t2 ∈ L2 do
8:         if alreadyMatched(t2) then
9:             continue
10:        end if
11:        if labelHash(t1) = labelHash(t2) then
12:            addMapping(t1, t2)
13:            continue to next t1
14:        end if
15:        text1 ← serializedText(t1)
16:        text2 ← serializedText(t2)
17:        sim ← diceSimilarity(text1, text2)
18:        if sim ≥ τ then
19:            candidates.add(t2, sim)
20:        end if
21:    end for
22:    if candidates not empty then
23:        best ← arg max_(t2,sim) candidates
24:        addMapping(t1, best)
25:    end if
26: end for
```

with identical label hashes are immediately mapped, bypassing string similarity calculations (line 17).

Algorithm 2 details the bottom-up matching phase for inner nodes. Once leaf nodes have been matched, the algorithm traverses all inner nodes in post-order, as provided by our custom iterator. The main optimization in this phase is primarily due to *HyperAST*'s range-based data structures. When evaluating the similarity between inner nodes (line 13), the algorithm can directly access all descendant mappings via contiguous array ranges, thereby eliminating the need for additional tree traversal.

3.2.3 Caching Strategies

We anticipated string similarity computation to be a major runtime bottleneck in *ChangeDistiller*. To try and mitigate this, we implemented two caching strategies:

Label Caching Lazily computes and stores each node's serialized representation (the complete statement text) upon first encounter during similarity calculations. Subsequent accesses reuse the cached value, avoiding redundant serialization.

N-gram Caching Lazily computes and stores n-gram hash-sets during the leaves matching phase, only for nodes requiring similarity computation. This avoids redundant string splitting and processing across iterations.

These caching strategies are integrated at lines 15 and 16 of Algorithm 1, where they first check for cached values before computing or serializing.

Algorithm 2 Optimized Bottom-Up Matcher Phase. Changes to the Original Highlighted in Blue.

```
1: I1 ← collectInnerNodes(T1)
2: I2 ← collectInnerNodes(T2)
3: for all t1 ∈ I1 do
4:     if alreadyMatched(t1) then
5:         continue
6:     end if
7:     for all t2 ∈ I2 do
8:         if alreadyMatched(t2) then
9:             continue
10:        end if
11:        range1 ← t1.descendantsRange
12:        range2 ← t2.descendantsRange
13:        sim ← diceSimilarity(range1, range2)
14:        if sim ≥ τ then
15:            addMapping(t1, t2)
16:        end if
17:    end for
18: end for
```

4 Evaluation

This section presents the empirical evaluation of our *HyperAST*-adapted *ChangeDistiller* implementation. We detail the experimental setup, summarize the results, and analyze the findings to address our two primary research questions: (1) Which *HyperAST* optimization techniques can be effectively adapted to the *ChangeDistiller* algorithm? and (2) How does the adapted *ChangeDistiller*'s runtime performance compare to the original algorithm?

4.1 Experimental Setup

We evaluated our implementation using the Defects4J dataset[3], which contains 1,046 real-world Java file pairs from 17 open-source projects. Eight algorithm variants were tested: two baseline configurations (shallow and deep statement processing) and six optimized configurations combining statement processing approaches with caching strategies (no cache, n-gram cache, and label cache). All experiments were conducted on a MacBook Pro with an M1 Pro processor and 16GB of RAM.

To focus the runtime comparison on the matching phases—the primary target of our optimizations—, we excluded *HyperAST* parsing overhead and edit script generation from the evaluation. Ensuring that the reported runtime improvements accurately reflect the impact of *HyperAST*'s optimizations on the matching process.

4.2 Results

The performance evaluation demonstrates substantial improvements across all tested variants. Our optimized implementations consistently outperformed their respective baselines, with runtime improvements of over 96% compared to the baseline for all variants.

Table 1: Runtime performance showing total time across 1,046 file pairs and median per-file runtime, with percentages showing reductions compared to the baseline.

Variant	Total (s)	Median (ms)
Deep		
Baseline	16,140.61	3,149.25
Optimized	141.04 (-99.13%)	31.55 (-98.99%)
+ Ngram	137.29 (-99.15%)	29.86 (-99.05%)
+ Label	140.36 (-99.13%)	31.01 (-99.01%)
Shallow		
Baseline	2,342.95	457.14
Optimized	90.40 (-96.14%)	15.56 (-96.59%)
+ Ngram	8.78 (-99.62%)	3.76 (-99.18%)
+ Label	86.52 (-96.30%)	14.99 (-96.72%)

4.2.1 Runtime Performance

As detailed in Table 1, the optimized deep statement variant—most comparable to the original *ChangeDistiller*—achieved a 99.13% reduction in total runtime (from 16,140.61 seconds to 141.04 seconds) and a 98.99% reduction in median runtime per file (from 3,149.25ms to 31.55ms). The shallow variant with n-gram caching was the fastest overall, completing the dataset in 8.78 seconds total and achieving a median runtime of 3.76ms per file. All optimized variants reduced runtime by at least 96%, with statistically significant improvements (Mann-Whitney U test, $p < 0.01$).

Figure 3 shows the runtime of each file pair for both deep and shallow processing, along with their optimized versions (plotted on log axes). Regression lines are included for each variant, with R^2 values indicating the strength of the correlation between file size and runtime. The high R^2 values for the baseline variants indicate that runtime is very strongly correlated with file size. The more moderate R^2 values for the optimized variants suggest that the correlation is still strong, but not as strong as for the baseline variants.

4.2.2 Speedup Ratios

Table 2 summarizes the speedup ratios of the optimized versions compared to their respective baselines. Deep statement processing achieved a median speedup of 94.2×, with the middle 50% of files showing speedups between 67.9× and 143.8×. Shallow processing achieved faster absolute runtime but more minor relative speedups, with a median speedup of 24.4×. The shallow variant with n-gram caching demonstrated the highest speedup ratios overall, achieving a median speedup of 117.9× and a maximum speedup of 218.8× for the largest files.

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

Figure 3: Runtime of each file pair for deep and shallow processing and their optimized versions (log axes). Regression lines and confidence areas are shown for each variant, with R^2 values: baseline shallow = 0.96, optimized shallow = 0.59, baseline deep = 0.94, optimized deep = 0.85.

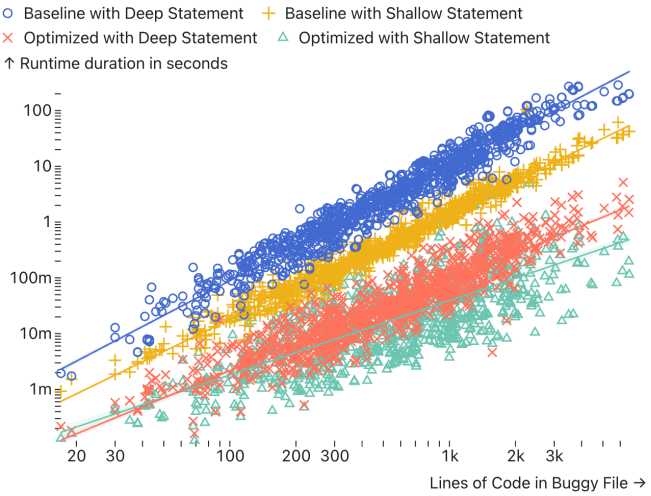


Table 2: Speedup ratios of optimized versions and their variants compared to their respective baselines, showing median and quartile ranges.

Variant	Median	Q1	Q3
Deep	94.2×	67.9×	143.8×
Deep + Ngram	97.7×	61.4×	146.0×
Deep + Label	94.8×	58.6×	145.7×
Shallow	24.4×	10.1×	67.6×
Shallow + Ngram	117.9×	61.9×	218.8×
Shallow + Label	26.1×	10.8×	71.5×

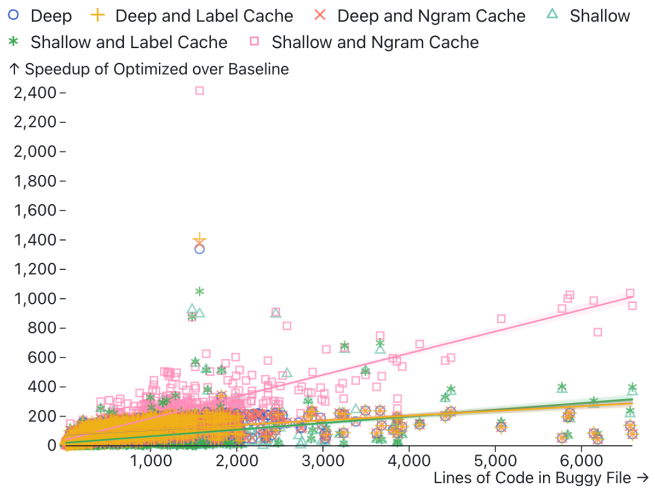
4.2.3 Reduction in String Similarity Computations

The median ratio of runtime contributed to string similarity computation, a major runtime bottleneck, was reduced from 45.3% to 2.8% for deep processing and from 89.1% to 70.5% for shallow processing. Enabling n-gram caching further reduced similarity checks to 1.0% for deep processing and to 6.1% for shallow processing.

Hash-based preliminary matching eliminated over 99.9% of similarity computations across all variants. For shallow processing, similarity checks were reduced from 543,492,106 to 127,906 across the dataset, and for deep processing, from 7,537,025,163 to 4,852,744.

All reductions in runtime ratio and similarity checks are significant (Mann-Whitney U test, $p < 0.01$).

Figure 4: Speedup ratios with regression lines and confidence areas of optimized variants compared to their respective baselines showing positive scaling with AST size. Optimized with shallow processing and n-gram caching shows a significantly stronger scaling behavior compared to other variants, which show a moderate scaling behavior.



4.2.4 Scaling Behavior

Figure 4 illustrates the speedup ratios across file sizes. Linear regression analysis reveals that the shallow variant with n-gram caching exhibits the strongest scaling behavior, with a regression slope of 0.148 ($R^2 = 0.64$). This indicates a moderate-to-strong correlation between file size and performance improvement. The regression line slopes from 39× for the smallest files to 1,014× for the largest files, demonstrating increasing performance gains for larger files. All other variants exhibit consistent but more modest scaling, with slopes around 0.033 ($R^2 \approx 0.18$–0.20).

4.3 Analysis

The experimental results provide strong empirical evidence for the effectiveness of adapting *HyperAST* optimization techniques to *ChangeDistiller*. The following analysis addresses our research questions and examines the implications of the observed performance improvements.

4.3.1 Adaptation of *HyperAST* Techniques (RQ1)

The successful integration of *HyperAST* optimization techniques into *ChangeDistiller* highlights their broader applicability beyond *GumTree*'s hybrid matching strategy. Optimized data structures, hash-based preliminary matching, and caching mechanisms were highly effective in addressing runtime bottlenecks. However, lazy decompression—while impactful for *GumTree*—likely had a limited effect in *ChangeDistiller* due to its bottom-up matching approach, as discussed in Section 4.4.

4.3.2 Runtime Performance Improvements (RQ2)

The optimized deep statement variant achieved a 99.13% reduction in total runtime and a 98.99% reduction in median runtime per file, transforming *ChangeDistiller* into a tool capable of efficiently analyzing large-scale codebases. The primary driver of these improvements was hash-based preliminary matching, which eliminated over 99.9% of string similarity computations by leveraging *HyperAST*'s precomputed label hashes. This optimization significantly reduced the computational overhead of the matching phases, addressing the core bottleneck of the original algorithm.

4.3.3 Variant-Specific Observations

Deep statement processing operates on many small statement units, resulting in numerous comparisons. While this increases the baseline runtime, it also makes hash-based filtering highly effective, achieving a median speedup of 94.2×. Shallow processing, which treats larger compound statements as single units, reduces the number of comparisons but increases the cost of each comparison due to longer statement strings. This explains why n-gram caching provides very noticeable benefits for shallow processing, increasing the median speedup from 24.4× to 117.9× by avoiding redundant string splitting and processing of longer strings.

4.3.4 Scalability Analysis

The shallow variant with n-gram caching not only achieved the best overall performance for our dataset but also exhibited the strongest scaling behavior, showing increasing performance gains with larger files. This variant's ability to speed up processing by over 1,000 times for the largest files underscores its potential for efficiently analyzing large-scale codebases. All other variants displayed consistent scaling behavior with increased input sizes. Although their gains were more modest, they still indicate that *HyperAST*'s optimizations are particularly beneficial for larger inputs.

The R^2 values for the regression lines in Figure 3 quantify how well file size predicts runtime for each variant. The high R^2 values (> 0.9) for the baseline variants indicate a very strong relationship between file size and runtime—larger files almost always take proportionally longer to process. In contrast, the optimized variants show somewhat lower R^2 values (0.59 for shallow and 0.85 for deep), suggesting that while file size remains an important factor, the optimizations have reduced the direct dependency of runtime on file size. This is especially notable for the optimized shallow variant, where the lower R^2 reflects more variable, but generally much lower, runtimes across different file sizes. This pattern indicates that the optimizations not only reduce overall runtime but also show that specific optimizations have a higher impact on certain types of files.

4.4 Limitations

While the evaluation highlights notable performance improvements, some limitations remain. These include differences from *GumTree*'s original implementation, the focus on Java-specific datasets, and the lack of evaluation of edit script quality. Nonetheless, the consistent runtime reductions across all file pairs suggest that the optimizations are widely applicable and scalable.

Internal Validity We verified the correctness of our implementation through unit tests for individual components and by comparing the output of the baseline and optimized versions of *ChangeDistiller*. These tests ensured that the optimizations preserved the core behavior of the algorithm. However, we did not compare our implementation to *GumTree*'s original Java-based *ChangeDistiller*[4], which could introduce subtle differences in behavior due to variations in AST representation or processing.

External Validity Our evaluation focused on Java bug fixes from the Defects4J dataset, which may limit the generalizability of our findings to other programming languages or more general commits. Despite these limitations, consistent improvements across all 1,046 file pairs suggest the broad applicability of the optimizations.

Construct Validity The metrics used to evaluate performance focus on runtime efficiency but do not account for memory usage or the quality of the generated edit scripts. While runtime improvements were the primary goal, the impact of the optimizations on memory usage and edit script quality remains unexplored.

Scalability Our results demonstrate strong scalability for larger files, particularly for the shallow processing variant with n-gram cache, which showed increasing performance gains as file sizes grew. However, the evaluation was limited to individual file pairs rather than full repository histories.

Limited Impact of Lazy Decompression Lazy decompression, a key optimization in *HyperAST*, likely had minimal impact on *ChangeDistiller* due to its bottom-up nature requiring all leaves. In *ChangeDistiller*'s first phase, all leaf nodes must be processed, which inherently forces the decompression of their parent nodes (i.e., all inner nodes), requiring full AST materialization regardless of lazy loading. In contrast, *GumTree*'s top-down phase can bypass entire subtrees when parent nodes match, making lazy decompression a valuable optimization in that context. This constraint presents an opportunity for future adaptations, such as incorporating a preliminary top-down phase to improve efficiency, as described in Section 5.4.

5 Conclusion and Future Work

This research demonstrates that *HyperAST* optimization techniques can be effectively applied beyond their original *GumTree* context to distinct AST-differencing algorithms, confirming both the technical feasibility and broader applicability of these performance enhancements.

5.1 Summary of Contributions

Our investigation addressed two fundamental research questions: (1) Which *HyperAST* optimization techniques can be effectively adapted to *ChangeDistiller*? and (2) How does the integration improve *ChangeDistiller*'s performance compared to the original implementation? These questions guided our systematic approach to evaluating the transferability and impact of *HyperAST*'s optimizations.

We made several key technical contributions. First, we successfully ported *ChangeDistiller* from Java to Rust, preserving the original algorithmic logic while leveraging *HyperAST*'s optimizations. Our approach includes a coarse-grained statement processing system that operates on fine-grained ASTs through custom post-order iterators, enabling *ChangeDistiller* to maintain its statement-level analysis without modifying *HyperAST*'s structure.

Second, we integrated three core *HyperAST* optimization techniques: optimized data structures, lazy decompression, and hash-based preliminary matching. We also implemented caching strategies, including label caching and n-gram caching, to further enhance performance. Finally, we conducted a systematic evaluation across eight algorithm variants to isolate the impact of individual optimization techniques.

5.2 Key Findings

Our empirical evaluation across 1,046 real-world code change pairs demonstrated substantial performance improvements. The primary optimized implementation achieved a 99.13% reduction in total runtime (from 16,140.61s to 141.04s) and a 98.99% reduction in median runtime per file (from 3,149.25ms to 31.55ms), with a median speedup of 94.2×, while maintaining core algorithmic behavior.

The systematic variant analysis showed important characteristics: deep statement processing–using non-compound statements as leaves–achieved superior improvement ratios (median 94.2× speedup), while shallow statement processing–using the largest compounded statements as leaves–provided better absolute runtime performance (as low as 3.76ms median with n-gram caching). Caching of n-grams proved highly effective for shallow statements, providing a 4.8× additional speedup over no caching. Hash-based preliminary matching proved highly effective, achieving a reduction of over 99.9% in similarity calculations for all variants. The moderate to strong positive correlations between AST size and performance improvement demonstrate that our optimizations become increasingly valuable for larger codebases, especially when combined with caching strategies.

5.3 Broader Implications

Our results provide strong empirical evidence that *HyperAST*'s optimization techniques have broader applicability

[4]https://github.com/GumTreeDiff/gumtree/tree/5b939f8

beyond their original *GumTree* implementation. The successful adaptation to *ChangeDistiller*'s fundamentally different bottom-up matching approach suggests these optimizations could benefit other AST-differencing algorithms regardless of their design principles.

The substantial performance improvements make *ChangeDistiller* viable for large-scale software analysis tasks that were previously computationally prohibitive. These advancements could enable new applications in continuous integration, automated code review, and software evolution studies.

Furthermore, the achieved speedups facilitate practical integration into workflows where rapid feedback is critical. For example, in continuous integration pipelines, our improvements allow for near real-time structural analysis of code changes, providing developers with immediate insights during code review. This capability enhances overall software development efficiency by supporting more informed coding decisions and quicker turnaround times.

5.4 Future Work

Our research opens several avenues for extending *HyperAST*'s optimization techniques to the broader AST-differencing ecosystem.

Generalized Adaptation Framework Our methodology could be extended to other AST-differencing algorithms, such as IJM [6], MTDiff [7], or RefactoringMiner [10], establishing a systematic framework for applying *HyperAST* optimizations across diverse matching strategies.

Preliminary Top-Down Processing Adding a preliminary top-down phase to *ChangeDistiller* could significantly improve performance. This phase would traverse from root to file, class, or method level, using label-hash matching to identify unchanged structures early. By marking entire subtrees as matched before the leaves phase begins, we could skip processing most nodes and focus only on subtrees containing changes. This optimization would also enhance the effectiveness of lazy decompression, as matched subtrees would not need materialization.

Framework-Level Improvements **Native Coarse-Grained Storage**: *HyperAST* could directly store statement-level representations instead of or alongside fine-grained ASTs, eliminating serialization overhead during similarity computation.

Enhanced Iterator Design: Leveraging *HyperAST*'s metadata during AST construction to pre-mark statement boundaries would enable more efficient traversal patterns and eliminate runtime predicate evaluation.

Evaluation Extensions Future evaluations could include cross-language validation, memory profiling to quantify space-time trade-offs, a complete repository history analysis rather than individual file pairs, and formal verification of algorithmic equivalence with the original implementation.

These directions could position *HyperAST* as an essential infrastructure for scalable code analysis, facilitating new applications in real-time change impact analysis and repository-wide software evolution studies.

6 Responsible Research

This section addresses the ethical considerations and reproducibility aspects of our research, emphasizing transparency and integrity and enabling independent verification of our findings.

6.1 Ethical Considerations

This research does not involve human subjects or sensitive personal data. All experimental data consists of publicly available source code from open-source software projects, which are already subject to public scrutiny and distributed under permissive licenses. Specifically, the Defects4J dataset used in this research is distributed under the MIT License, and the *GumTree* repository is distributed under the GNU Lesser General Public License (LGPL) Version 3, ensuring compliance with open-source licensing standards.

While our work focuses on technical improvements to code differencing algorithms, we acknowledge the broader impacts of enhanced code analysis tools. Improved AST-differencing capabilities can contribute positively to software engineering practices by enabling better code review processes, more accurate change impact analysis, and enhanced software maintenance workflows. However, such tools could theoretically be misused for unauthorized code analysis or raise intellectual property concerns. To mitigate these risks, we emphasize that our research adheres to ethical standards and aims to advance legitimate software engineering practices while contributing to the open-source community's toolkit for code analysis.

6.2 Reproducibility

Reproducibility is an important aspect of our research, and we have made efforts to ensure that all components necessary to verify our findings are publicly accessible:

Implementation: Our *HyperAST*-adapted *ChangeDistiller* implementation, including all algorithm variants and optimization strategies described in the paper, is available in the code repository[5].

Dataset: We used the Defects4J dataset from the *GumTree* datasets repository[6], which contains 1,046 real-world Java file pairs from 17 open-source projects, as described in our evaluation.

Baseline Implementation: Our baseline comparisons use the original *ChangeDistiller* implementation from the *GumTree* Java repository[7] but ported and adapted to Rust and *HyperAST* without any additional optimizations.

[5] https://github.com/leo-mangold/HyperAST/tree/9a1ab796
[6] https://github.com/GumTreeDiff/datasets/tree/33024da
[7] https://github.com/GumTreeDiff/gumtree/tree/5b939f8

All benchmark configurations, experimental parameters, and analysis scripts are included in the code repository[8], ensuring transparency and enabling independent verification of our results.

6.3 Disclaimer on the usage of LLMs

This report was produced as part of the CSE3000 Research Project course at Delft University of Technology. In accordance with the course policy on the use of Large Language Models (LLMs), these tools were utilized to support the research and writing process.

Specifically, LLMs were used for purposes such as gathering information and assisting in the writing of texts. Importantly, **all content, ideas, and information presented in this report have been written or thoroughly verified by the author**. The use of LLMs does not absolve the author of responsibility for the content presented. LLMs were treated strictly as a supportive tool, complementing but not replacing critical thinking and analysis.

LLMs were used to: (1) help with broader understanding before reading papers in detail. Prompt: "Summarize the main points of the attached paper." (2) get feedback on the clarity and coherence of the text. Prompt: "Give me feedback on the clarity and coherence of the following text." (3) fix typos and grammatical errors. Prompt: "Fix spelling and grammar errors in the following text."

7 References

[1] B. Fluri, M. Wursch, M. PInzger, and H. Gall, "Change distilling:tree differencing for fine-grained source code change extraction," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 725–743, 2007, doi: 10.1109/TSE.2007.70731.

[2] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and Accurate Source Code Differencing," in *Proceedings of the International Conference on Automated Software Engineering*, Västeras, Sweden, 2014, pp. 313–324. doi: 10.1145/2642937.2642982.

[3] C. Yang and E. J. Whitehead, "Pruning the AST with hunks to speed up tree differencing," in *2019 IEEE 26th international conference on software analysis, evolution and reengineering (SANER)*, 2019, pp. 15–25. doi: 10.1109/SANER.2019.8668032.

[4] Q. Le Dilavrec, D. E. Khelladi, A. Blouin, and J.-M. Jézéquel, "HyperDiff: Computing Source Code Diffs at Scale," in *ESEC/FSE 2023 - 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, San Francisco (CA, USA), United States: ACM, Dec. 2023, pp. 1–12. doi: 10.1145/3611643.3616312.

[5] Q. Le Dilavrec, D. E. Khelladi, A. Blouin, and J.-M. Jézéquel, "HyperAST: Enabling Efficient Analysis of Software Histories at Scale," in *ASE 2022 - 37th IEEE/ACM International Conference on Automated Software Engineering*, Oakland, United States: IEEE, Oct. 2022, pp. 1–12. Available: https://inria.hal.science/hal-03764541

[6] V. Frick, T. Grassauer, F. Beck, and M. Pinzger, "Generating accurate and compact edit scripts using tree differencing," in *International conference on software maintenance and evolution (ICSME)*, Sep. 2018, pp. 264–274. doi: 10.1109/ICSME.2018.00036.

[7] G. Dotzler and M. Philippsen, "Move-optimized source code tree differencing," in *2016 31st IEEE/ACM international conference on automated software engineering (ASE)*, 2016, pp. 660–671. doi: 10.1109/ASE.2016.92.

[8] J. Matsumoto, Y. Higo, and S. Kusumoto, "Beyond GumTree: A hybrid approach to generate edit scripts," in *Proceedings of the 16th international conference on mining software repositories*, in MSR '19. Montreal, Quebec, Canada: IEEE Press, 2019, pp. 550–554. doi: 10.1109/MSR.2019.00082.

[9] M. Martinez, J.-R. Falleri, and M. Monperrus, "Hyperparameter optimization for AST differencing," *IEEE Transactions on Software Engineering*, vol. 49, no. 10, pp. 4814–4828, 2023, doi: 10.1109/TSE.2023.3315935.

[10] N. Tsantalis, A. Ketkar, and D. Dig, "RefactoringMiner 2.0," *IEEE Transactions on Software Engineering*, vol. 48, no. 3, pp. 930–950, 2022, doi: 10.1109/TSE.2020.3007722.

[11] P. Alikhanifard and N. 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.*, vol. 34, no. 2, Jan. 2025, doi: 10.1145/3696002.

[12] R. Just, D. Jalali, and M. 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*, in ISSTA 2014. New York, NY, USA: Association for Computing Machinery, 2014, pp. 437–440. doi: 10.1145/2610384.2628055.

[13] S. S. Chawathe, A. Rajaraman, H. García-Molina, and J. Widom, "Change detection in hierarchically structured information," *ACM SIGMOD Record*, vol. 25, pp. 493–504, 1996, doi: 10.1145/235968.233366.

[14] L. R. Dice, "Measures of the amount of ecologic association between species," *Ecology*, vol. 26, no. 3, pp. 297–302, 1945, Accessed: Jun. 19, 2025. [Online]. Available: http://www.jstor.org/stable/1932409

[8] https://github.com/leo-mangold/HyperAST/tree/9a1ab796