Accelerating AST-Based Code Differencing

Optimizing ChangeDistiller's Bottom-Up Matching Strategy with HyperAST

Leo Mangold

2025

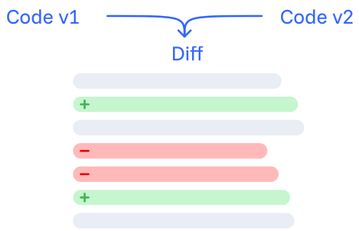
1. Introduction

Traditional code comparison tools like Unix' diff operate on plain text, often missing structural changes in software. Tools such as ChangeDistiller and GumTree improve on this by analyzing Abstract Syntax Trees (ASTs), but they struggle to scale with large codebases.

This research aims to improve ChangeDistiller's runtime performance by adapting it to leverage optimizations provided by HyperAST—a data representation framework optimized for large scale AST Differencing

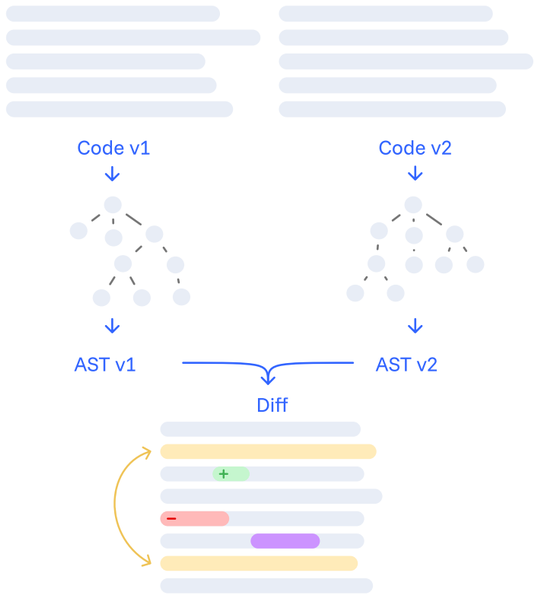
Text Differencing

Textual differencing compares text files line by line, identifying insertions & deletions.



Differencing on ASTs

AST-based differencing compares two code files by converting them to AST representations and analyzing the differences between them.



2. Research Questions

1. Which HyperAST optimization techniques can be adapted to the ChangeDistiller algorithm?
2. How does the adapted ChangeDistiller's runtime performance compare to the original algorithm?

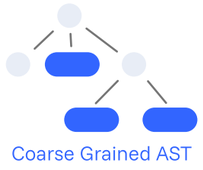
3. Background

ChangeDistiller is a tree differencing algorithm with a bottom-up matching approach on coarse-grained ASTs, focusing on statement-level analysis distinct from the top-down fine-grained approach the GumTree algorithm uses.

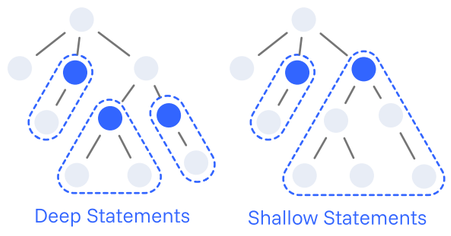
HyperAST is a data structure framework optimized for large scale code analysis, with deduplication, fast data access and precomputed metadata.

4. Statement iteration

ChangeDistiller's AST leaves represent statements.



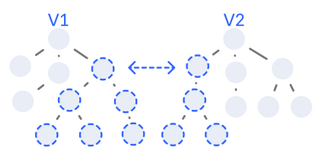
Since HyperAST uses fine-grained ASTs, we built a custom iterator that treats statement nodes as logical leaves. This iterator has two variants: one for the deepest statements and one for the largest (possibly containing statements).



5. Range Based Similarity

HyperAST stores nodes in post-order arrays, which allow us to efficiently get all descendants by taking a range from the left-most leaf descendant to the node itself. This enables efficient range-based similarity computation in the bottom up phase, allowing direct access to a node's descendants without tree iteration.

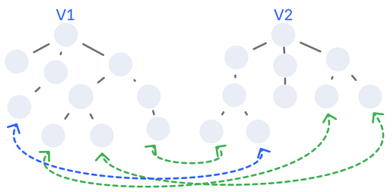
```
subtreeSimilarity = diceSim(lld(A)..id(A),lld(B)..id(B))
```



-91% Subtree similarity computation time

6. Hash Preliminary Matching

When matching leaves we use hash based preliminary matching to reduce most of the O(n+m) string similarity computations to simple O(1) hash comparisons.




```
nodeSimilarity =
  if hash(A) == hash(B) then 1          O(1)
  else stringSim(A,B)                   O(n+m)
```

99.9% String Similarity Computations Eliminated

7. Caching for String Similarity

In the leaves matcher, we implemented caching for leaf node string representations, lazily computing n-grams to optimize performance and reduce redundant calculations.

No Cache

```
diceSim(
  makeNgrams(A),
  makeNgrams(B))
```
for all

N-Gram Cache

```
makeNgrams(A),makeNgrams(B)   once
        Cache  ngramsA  ngramsB
diceSim(ngramsA,ngramsB)      for all
```

For deep statements, n-gram caching offered a 4% improvement over the optimized version without caching. For shallow processing, caching reduced the median runtime by 79% compared to no caching.

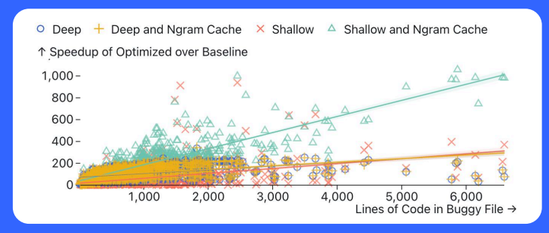
117.9x Median Speedup Compared to Baseline with Shallow Statements

8. Results

We evaluated our optimized version against the baseline on the Defect4J dataset, containing 1046 real-world file pairs of Java bug fixes.

Variant	Total Runtime	Median per File	Median Speedup
Deep Statements			
Baseline	4.5 hours	3,149 ms	
Optimized	2.3 min	31.6 ms	94×
Opt. + Cache	2.3 min	29.9 ms	98×
Shallow Statements			
Baseline	39 min	457 ms	
Optimized	1.5 min	15.6 ms	24×
Opt. + Cache	8.8 sec	3.8 ms	118×

All variants show significant improvements and positive scaling behavior. Shallow with N-gram caching is the fastest and shows a significantly stronger scaling behavior.



HyperAST's optimizations make ChangeDistiller significantly faster and scalable, enabling more efficient analysis of large codebases.

9. Future Work

Generalize to Other Tools: Apply HyperAST optimizations to different algorithms (e.g., MTDiff, RefactoringMiner).

Top-Down Matching: Integrate a preliminary top-down phase to further reduce computation and improve lazy decompression effectiveness.

Native Coarse-Grained ASTs: Extend HyperAST to natively support statement-level ASTs for even faster processing.

Broader Evaluation: Test on more languages, analyze full histories, and assess memory usage.