Evolution of modern Structural Diff Algorithms

A detailed analysis of heuristic advancements in structural diff algorithms for scalable and accurate code differencing.

TU Delft Delft University of Technology

1 Structural Code Diffs

What are structural diffs?
Unlike traditional line-based diffs, structural code diffs operate on Abstract Syntax Trees (ASTs). This enables them to capture the structure and intent of code changes, such as moving, renaming, or refactoring elements, rather than just tracking insertions and deletions of lines.

Why do we need them?
Line-based diffs are often misleading—renames look like deletions and insertions, and moves are missed entirely. Structural diffs generate concise, semantically meaningful edit scripts that reflect how developers perceive changes. This improves tools like version control systems, branch merging, and automated refactoring.

What challenges do they pose?
Computing diffs over trees requires solving the Tree Edit Distance (TED) problem, which is NP-hard. Exact solutions do not scale. Practical systems must rely on heuristics that balance script quality and runtime, while ensuring results are still accurate

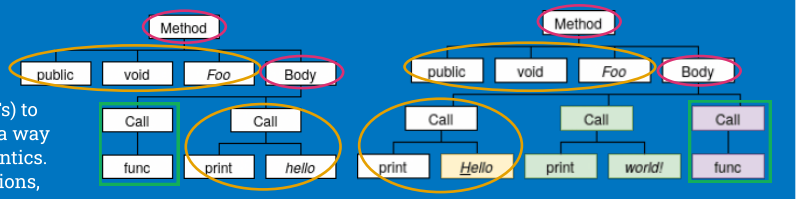
```
1 public void Foo() {
2     func();
3     print("hello");
4 }
```

```
1 public void Foo() {
2     print("Hello");
3     print("world!");
4     func();
5 }
```

2 Solution

Approach
We use abstract syntax trees (ASTs) to represent the structure of code in a way that captures its syntax and semantics. The actual edit operations—insertions, deletions, moves, and updates—are derived from a set of node mappings using the Chawathe algorithm.

What are mappings?
Mappings are one-to-one correspondences between nodes in the source and target ASTs that share the same label and represent equivalent syntactic constructs. A high-quality set of mappings leads directly to a more concise and interpretable edit script.



Three-phase algorithm design
Each phase of the algorithm is responsible for finding a portion of the node mappings. By separating the process into distinct phases, we can improve each part independently.

Subtree phase: Detects large identical subtrees, anchoring identical subtrees.

Bottom-up phase: Builds on these anchors by matching parent nodes based on their already-mapped descendants.

Recovery phase: Performs detailed local matching to capture fine-grained edits that were missed earlier.

3 Diff algorithm evolution

1 Xy Algorithm

- First to adopt the 3-phase model for tree differencing (from XML).
- Very fast, but recovery phase is shallow: only considers direct children.
- Provides a flexible base, but performs poorly on complex code structures.

2 GumTree

- Introduces an optimal recovery algorithm that inspects entire subtrees.
- Captures complex refactorings and nested changes.
- High-quality edit scripts, but costly: runtime grows rapidly with AST size.
- Introduces MAX_SIZE to skip recovery on large subtrees.

3 GumTree Simple

- Replaces optimal recovery with a fast heuristic.
- Trades some precision for dramatic runtime and stability gains.
- Maintains quality on large trees where GumTree's recovery is skipped.
- Becomes more accurate than GumTree on large codebases due to fewer omissions.

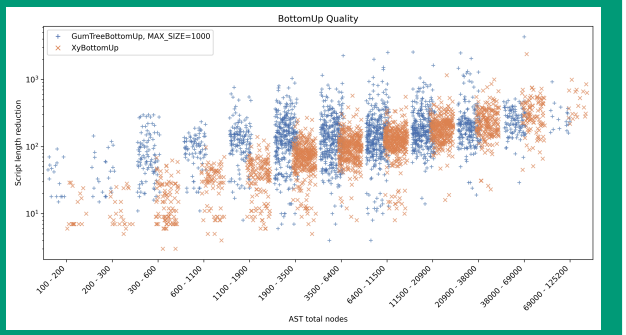
4 HyperDiff

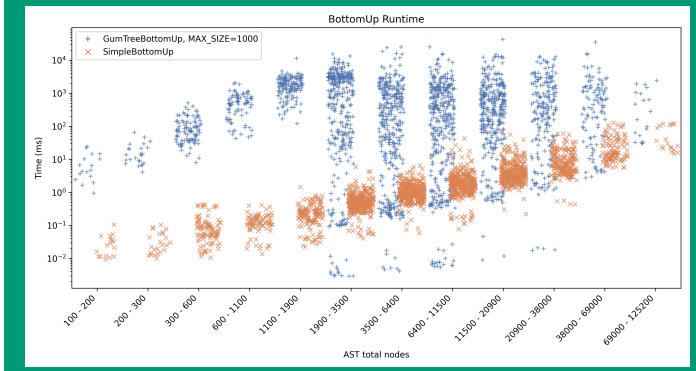
- Leverages HyperAST, a compressed AST structure that shares repeated subtrees.
- Detects identical subtrees efficiently without traversal.
- Matches GumTree in quality but improves performance by up to 12× in real-world benchmarks.
- Especially effective when code has many repeated constructs.

Summary
Over time, structural diff algorithms evolved from a generic, XML-focused design (Xy) toward highly specialized tools for code. GumTree introduced optimal recovery, significantly improving accuracy but at a high computational cost. GumTree Simple responded with a scalable heuristic, reducing runtime while maintaining quality on large trees. HyperDiff further improved performance using a compressed AST representation, maintaining GumTree's quality with up to 12× faster execution.

4 Evaluation

Main Question
How do refinements in AST differencing algorithms improve the trade-off between performance, scalability, and script quality?
To answer this, we define subquestions focused on each major refinement: GumTree's optimal recovery, Simple's heuristic strategy, and HyperDiff's compressed AST. Each is evaluated in isolation to understand its specific impact.





Methodology
We evaluate the algorithms on 2045 Java file pairs. For every algorithm, we isolate all phases and measure runtime and script quality. Runtime is recorded using Criterion.rs to ensure statistically reliable and reproducible results. This setup allows us to attribute improvements to specific refinements in each algorithm.

Author: Maciej Mejer (m.j.mejer@student.tudelft.nl)
Poster style inspired by: Jakub Rutkowski

Supervisor: Quentin Le Dilavrec
Responsible Professor: Carolin Brandt