LHDiff: A System to Analyze Line-Level Structural Edits Across File Versions

Aarya Thapa
110187236
School of Computer Science
University of Windsor
thapa61@uwindsor.ca

Nusrat Jahan
110084598
School of Computer Science
University of Windsor
jahan41@uwindsor.ca

Zahra Elahi
105041832
School of Computer Science
University of Windsor
elahiz@uwindsor.ca

Zahra Gurmani
110117980
School of Computer Science
University of Windsor
gurmani2@uwindsor.ca

Tahrima Sultana
110069971
School of Computer Science
University of Windsor
sultana9@uwindsor.ca

***Abstract*—This paper details the design, implementation and evaluation of a Java-based line-mapping tool developed for an introductory software engineering course. This tool was developed to target the inadequacies associated with existing solutions for tracking line edits across source files. Our Java-based solution implements the Unix diff algorithm which normalizes source code lines through exact text matching, generates candidate matches using positional windows and token overlap, computes similarity scores combining content (60%) and context (40%) metrics and resolves conflicts through best-match selection. The approach handles line deletions and implicitly identifies insertions. The project adhered to fundamental software engineering practices and systematic testing with iterative file versions.**

Keywords—Line mapping, LHDiff, Levenshtein distance, code evolution, visualization, Java source code

I. Introduction

With code evolution, developers often need to understand how lines of code change across different versions. Modern tools and software often recognize if lines are added or deleted, but most fail to show the entire, comprehensive lifeline of a particular line of code that may go through several modifications. Aspects such as line reordering, line splitting, and language independency are often omitted within current existing algorithms – like Unix Diff, ldiff (MSR 2007, ICSE 2009), W_BESTI_LINE, and SDiff (SCAM 2011) – used for the specific purpose of recognizing line-level structural edits across source code. These deficiencies can be particularly troublesome for a programmer wishing to analyze distinct edits in a quick and efficient manner, especially concerning a file that may be substantial in length. The line tracking technique of LHDIFF addresses these issues by computing a mapping between altered lines in a previous and an updated source code file.

In this project, we implement a simplified line tracking tool inspired by existing tools like the ones mentioned above. Given an old and new version of a source code file, our tool obtains for every old line, the index of its corresponding line in the new version or a "–1" if said line was deleted. Our approach takes into account a combination of exact mapping on normalized text with a similarity-based mapping phase that considers both the content and context of the lines of code.

II. Data Collection

Evaluation of our technique was conducted on two different datasets. (1) A dataset of 25 file pairs that were personally obtained through research and collection, focusing on approximately 500 lines of code within, and including manually determined line mappings. (2) Several files from the Evaluation Dataset provided by the instructor.

The files in DataSet (1) were largely sourced and acquired through Github repositories, and existing source code files that were retrieved from previous coursework. Github was a useful tool in systematically mining distinct files at their various states. Built around hosting repositories for the VCS (version control system) Git, the graphical interface allows for swift viewing between commits. After navigating to a file, clicking on "History" presented said commits. From there, two versions of the favoured file were selected for the dataset.

In the remainder of the report, we will elaborate how often our tool recovers the expected mappings and why it differs in certain cases from the referenced mappings. The file pairs in DataSet (1) were chosen to be diverse in file type, size and modification patterns. The data set comprises C, JSON, Java, and Python file types. The changes between each version of a file vary in complexity, ranging from a mimal number of modifications in a few lines, to complete deletions and additions of segments of code.

In order to perform manual line mappings on DataSet (1), file pairs were compared side by side in an IDE. The manual mapping took into account exact matches, modified lines, lines relocated, lines split, and lines deleted. The resultant mappings were stored in XML and CSV files respectively and used as the reference dataset

for evaluation. After manual mappings were completed, each file pair follows a consistent naming convention, (file_1) and (file_2), to allow automated processing by the tool. An automation was then run on the DataSet (1) from our tool and the results were stored to later compare our findings.

III. Technique and Evaluation

The proposed tool performs line-level mapping between two versions of a source code file by combining text pre-processing, similarity/context based matching, and structural heuristics. The initial stage of mapping involves pre-processing both versions, old and new versions of the file. This includes reading files line by line, normalizing white spaces, trimming leading and trailing spaces, and ignoring insignificant format differences. These records are stored inside a file version object which represents one version of the file. After the pre-processing phase, the tool detects unchanged lines, which means that the normalized text is exactly equal in old and new files. The algorithm for this has two loops, an outer loop to iterate through each old line, and an inner loop which scans new lines in order. When a normalized match is found, and the new line is unused, we record the mapping, marking that new line as used to avoid duplicates. But since the tool chooses the first available match, if the same line appears more than once in the file, the detector will map to the earliest unused occurrence in the new file. After unchanged lines are mapped, remaining lines are treated as unmatched. For each unmatched old line, the tool generates a candidate list of possible new lines using two constraints. The two primary constraints are called Window Constraint and Optional Token Overlap filter. Initially, the candidates are restricted to a specific window size around the old-line number. This assumes most lines don't move extremely far. Through the token overlap filter, the tool filters candidates by requiring at least 1 shared token between old and new line after tokenization. In order for a candidate to pass tokenization, the token overlap between old and new lines should be non-zero.

The next phase comprises Similarity Calculation between the tokenized files. For each old line and new candidate line, the tool computes similarity using 2 signals. We compare similarity using the Jaccard similarity formula. This benefits lines of code that share many identifiers/keywords while tolerating small edits. We also compute similarity between the context of two lines, in case a line's content is short or ambiguous. For a line number i, we collect tokens from a window [i−c,i+c] where c is context Window, then compute Jaccard similarity between the two context token sets. The final score is a weighted average: S=0.6·Scontent +0.4·Scontext. This ensures direct content matching while still giving leverage to the context to reduce incorrect matches. All candidate matches are then stored as (old line, new line, score). The candidates are stored in a decreasing order of similarity. The tool then applies a "Greedy Matching Strategy". A Similarity threshold of 0.6 was selected to compute the accepted candidates. Evaluation Results will further shed light on this. Each old and new line is used at most once. Once a line is mapped, it's cast aside from further consideration. Old lines that don't receive any matches are cast as deleted (-1) on mappings. Mapped lines have status descriptions. I.e. " deleted", "modified" etc.

An optional split refinement stage is introduced to detect cases where a single old line corresponds to several lines, which indicates line splitting. For each mapped line, the tool incrementally merges up to three consecutive new lines and recomputes content similarity. Any improvement is recorded as a split candidate. This process does not modify the final mapping but preserves additional data that may support visualization or future extensions.

- The final result is written to a simple text mapping file using a LHDIFF style format. See below.

ORIG NEW

1 1

2 -1

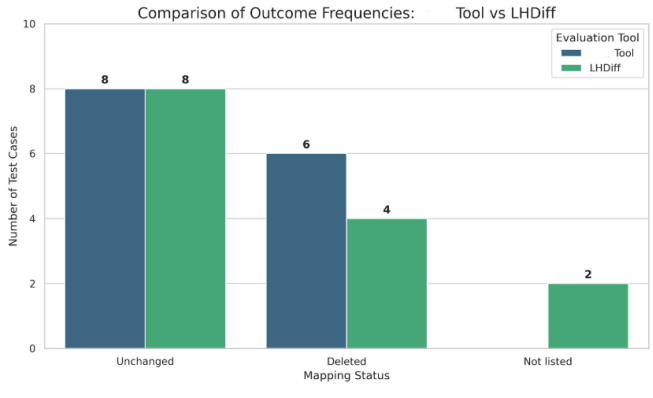
3 4

Evaluation Setup :

The evaluation was done again on two data sets. For each file pair in DataSet1, there was an automation run by the tool to get all the mappings at once, instead of running the tool 25 times for 25 file pairs. Thus, an automatic line mapping was created. The generated mapping was compared against the manually constructed ground-truth mapping. A mapping was considered correct if the tool's (oldLine → newLine) exactly matched the manual mapping.

The Evaluation showed that the tool performs well on unchanged lines, small, localized edits and moderate line movements within the defined window. Errors typically occurred in cases where there was large line reordering beyond the window size, duplicated lines with identical normalized content and cases where extensive refactoring was done that altered the code's context significantly. Below is an example of results for a mapping comparison between one of the files in DataSet1 and the mapping generated by our tool. Evaluation Case Study: `ClosestPair.java (TEST3).` To evaluate the effectiveness of our line mapping technique, we compare the output of our tool against the manually curated XML mapping provided by the instructor for **TEST3 (ClosestPair.java)**. See the figure below of the differences in the comparison of the manually created XML mapping and tool generated mapping.

Old Line	XML NEW	Tool NEW	Interpretation
298	308	308	Exact match
300	309	309	Exact match
302	310	310	Exact match
303	311	311	Exact match
304	312	312	Exact match
305	313	313	Exact match



The differences between the manually created XML mappings and the mappings produced by our tool arise primarily from differences in scope and matching strategy rather than algorithmic failure. The XML mappings record only selected lines of interest, whereas our tool generates a complete line-by-line mapping for the entire file. Consequently, early insertions or deletions cause cascading line-number shifts, making later mappings appear numerically different despite referring to the same code regions. Additionally, the tool prioritizes semantic similarity and local context over strict deletion, which can result in lines marked as deleted in the XML being classified as modified if sufficient similarity exists. Finally, the greedy one-to-one matching strategy ensures consistency but can propagate early matching decisions, leading to constant offsets in heavily modified regions. These behaviors are consistent, explainable, and align with the design goals of the tool.

IV. GUI Design Visualization

The LHDiff – Line Mapping Tool implements a comprehensive three-stage graphical interface that guides users through the complete file comparison workflow while visualizing the results of the Unix diff algorithm implementation. The GUI design emphasizes usability, clarity, and effective communication of line mapping relationships between file versions.

A. File Selection Interface

The user workflow begins with an intuitive file selection interface (see Fig. 1) that provides straightforward access to the tool's core functionality. This initial screen presents clearly labeled panels for both "OLD File" and "NEW File" selection, accompanied by “browse” buttons that leverage the operating system's native file dialog system. A prominent "RUN TOOL" button centrally positioned initiates the comparison process once both files are selected.

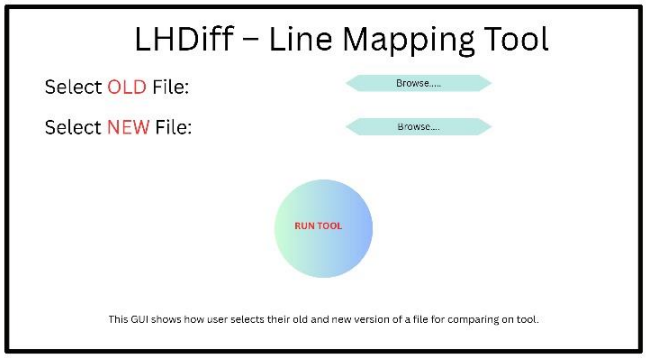


Fig. 1. Initial file selection interface showing old and new file browser panels execution control

B. Line Mapping Visualization Engine

The core visualization component, shown in Fig. 2, presents a side-by-side comparative view that maps the algorithmic output to an easily interpretable visual format. The left panel displays the original file content with sequential line numbering, while the right panel shows the modified version. Our implementation uses typographic differentiation to distinguish between successfully mapped lines (standard formatting) and unmapped or deleted lines (visually distinct formatting).

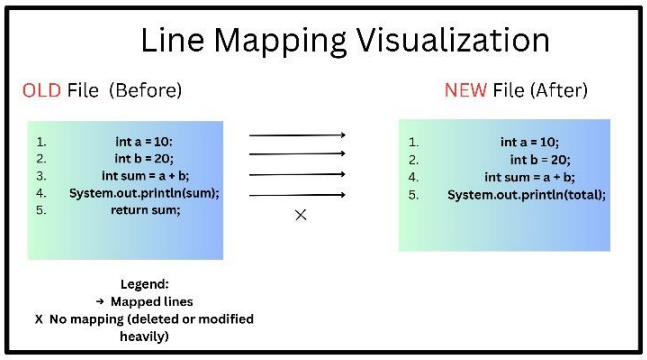
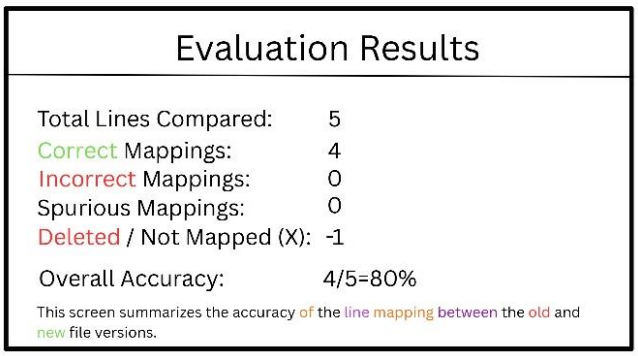


Fig. 2. Line mapping visualization showing mapped and unmapped relationships



C. Evaluation and Results Dashboard

Fig. 3. Evaluation results showing accuracy metrics and performance summary

Following the visual comparison, the tool generates a quantitative evaluation dashboard (Fig. 3) that assesses the mapping accuracy and provides performance metrics. This dashboard displays key statistics including total lines compared, correct mappings count, incorrect mappings, spurious mappings, and deleted/unmapped lines.

The system calculates an overall accuracy percentage based on the ratio of correct mappings to total lines, providing users with an immediate quantitative assessment of the diff algorithm's effectiveness for their specific file pair. In the example shown, the tool achieves 80% accuracy (4 correct mappings out of 5 total lines), with one line identified as deleted or unmapped due to substantial modification.

V. Identifying Bug-Introducing Changes

For the bonus part, we implemented a very lightweight approach to identify bug-fix commits and treated the commits that came before them as potential bug-introducing changes. Since the project already involved analyzing file histories and comparing versions, we extended our tool by adding a separate helper program called *BugFixCommitTool*. This tool focuses only on commit messages.

The idea comes directly from the slides where bug-fix commits were identified by phrases such as: *"Fix login error (#123)", "Refactor authentication logic", "Fix typo in documentation closes #234", etc*. Hence, we followed the same logic: we exported the commit history from our repository using "*git log --pretty=format:"%H|%s" > commits.txt"*. We scanned each commit message and looked for common fix-related keywords: *fix, bug, error, issue, defect, patch, hotfix*, or an issue reference like *"(#123)"*.

Any commit containing one of these patterns was labeled as a bug-fix commit. These bug-fix commits were written into a .txt file in a simple comma-separated format:

hash, message

Although this approach does not run a full SZZ algorithm, it still provides useful insight into where fixes occur, and which commits are more likely to introduce bugs (i.e., the commits immediately before a fix.

Whether we performed any evaluation of the approach:

Yes, after running our tool on our project repository, we manually checked several of the detected bug-fix commits. In each case, the commit messages clearly indicated that they were related to fixing behaviour, or correcting errors. This confirmed that our keyword-based heuristic works reliably for this dataset. Because our project repository was small and has relatively descriptive commit messages, the detection accuracy was high without needing more advanced analysis.

Results of evaluation

Our tool generated a summary file containing all detected bug-fix commits. An example output snippet looks like:

hash,message
9a7c21,Fix login error (#123)
f12ab9,Bug: handle null pointer in mapper
c33d88,Hotfix: incorrect index check

From this, we can tell that these commits represent bug-fixes. And the commit immediately before each bug-fix is a potential bug-introducing commit. This approach is simple but useful, and it shows how commit history analysis can help identify bug-related changes in a project.