A Tool For Line-Mapping And Visualization Between File Versions

Asfi Hassan
School of Computer Science
University of Windsor
Ontario,Canada
hassan5c@uwindsor.ca

Kavish Palan
School of Computer Science
University of Windsor
Ontario,Canada
palan@uwindsor.ca

Anas Mohammad
School of Computer Science
University of Windsor
Ontario,Canada
moham721@uwindsor.ca

Lara Shreim
School of Computer Science
University of Windsor
Ontario,Canada
shreiml@uwindsor.ca

Abstract—**In this project, we built a tool that compares two versions of the same code file and shows all the differences between them. The end goal of this project is to figure out which lines were unchanged, which ones were removed, which ones were edited, and which ones were newly added. So, we tested this tool by collecting 28 files pairs from different code languages and resources, then tested each pair by using our tool. The tool performed well on our dataset and the dataset the professor provided. Lastly, we designed a simple, easy-to-read GUI visualization that shows how each line is mapped between the old and new version of the file.**

Keywords—Line Mapping, File Comparison, Git Commits, Version Control, XML output comparer, Visualization.

I. Introduction

In the world of software development, code is always changing, whether debugging code or editing existing code. With the constant change in code, developers often need a clear method on how one version of the file differs from the other version as a way to track their work progress. That's where the line-mapping comes in, a tool that compares between two versions of a file and identifies which lines were added, edited, or removed. This project's goal is to build the line-mapping tool which aims to provide a reliable mappings across different programming languages.

In this process, we evaluated our built tool by testing it on dataset of 28 pair files that we collected as well as the dataset provided by the professor. Then, we also built a simple output comparer code to check whether our mapping results matched with the expected results. The results show that the tool performs well across different simple program languages, demonstrating a structured line-map that helps developers observe the change of file versions and understand tasks like debugging and version control analysis.

II. Data Collection

To test our line-mapping tool performance, we gathered 28 pair files dataset, with each pair demonstrating two different versions of the same file. These dataset files' purpose is to evaluate the tool across the different kinds of source codes.

A. *Choosing the File Pairs*

The 28 pairs of files were collected from various sources which are:

- Previous Coding Projects: Theses files contain languages like Java, C, and Python with alterations made from previous submissions.
- Open-Source Codes from GitHub repositories: We chose some of our files from repositories with different version history, with commits that are focused on actual code alterations such as insertion and removal.
- New Coding Scripts: We created new pairs of code files from various languages like Java, C, JavaScript, and other languages.

While collecting dataset, we searched particularly for pairs with huge differences in their versions. We also deleted pairs whose versions were only updating whitespace or formatting, since it wouldn't boost the line-mapping tool's performance.

Each dataset versions is chosen based on this criteria:

- Inserted and removal parts of the code.
- Modified parameters and variable variation.
- Altered loops, conditions, and outputs.

B. *Determining the Line Mappings*

Once the file pairs are collected, we determine the correct line-mapping by two different approaches:

1- For the file pairs we created and collected, we manually scan each file's versions side by side. For every line, we determine the state of it by these five categories:

- Unchanged- same line in both versions.
- Edited- same line in both versions but modified.
- Deleted- line was removed in the newer version.
- Added- line was inserted in the newer version.

By the end of this process, we had created verified line-mapped codes, forming a ground-truth dataset and evidence that we used to test our line-mapping python code accurately and across all the pair files.

2- For the files provided by the professor, instead of scanning through each pair manually, we modified our line-mapper python code to produce an XML files that demonstrates the actual mapping between the old version program lines and the new version program lines, then we built a comparison program that takes both the XML file generated by our tool and the XML file provided by the professor and compares them line by line, then calculates the accuracy percentage and shows how many lines are match and not matched. Which makes the evaluation faster and more consistent.

By combining both the manual checking for our own dataset and XML comparison for the professor's dataset, we determine that the evaluation was consistent and accurate on a high scale.

III. Technique Description and Evaluation

The purpose of this evaluation is to determine how each line from the old version of the program corresponds to the new version of the same program. Which is accomplished by using the python's text-matching comparison algorithm combined with few rules that decides whether the line has been modified, changed, shifted to another location, or other.

A. *Overview of the line-mapping method*

Our method is to evaluate how each line from the old version of the program corresponds to the new version of the same program. Which is accomplished by using the python's text-matching comparison algorithm combined with few rules that decides whether the line has been modified, changed, shifted to another location, or other.

Our method works in three steps:

- **Preprocessing:** where both versions of the file are read and split into two different lines, note that whitespace differences are ignored and only focuses on actual line changes.
- **Matching Unchanged Lines:** Using our python file linemapper.py, we prioritize finding exact matches (unchanged lines) between our two version of the program.
- **Detect all the changes:** After we are done matching the unchanged lines, we perform our second similarity check with our new version of the file where most of the cases are:
    1. If the line if still exists but with modifications, then we mark it as "line edited"
    2. If there is now trace of our line, then mark as "line deleted".
    3. If there is a new line in the program, then mark it as "line added".
    4. If it's the same line in both versions, then mark it as "line unchanged".
- XML output evaluation: Our tool gives an XML output file so that it can be used for comparisons with other referenced files.

With these steps, our tool is able to map most of the lines relationships even if the changes are a lot to handle.

B. *Evaluation Method*

To evaluate our tool performance, we tested it into two categories:

1- The 28 file pairs that our team manually created and collected : we determine the mapping of this dataset by manually scanning the old and new versions of the file side by side. We traced each old line with the corresponding new line, then labeled all the changes that happened such as unchanged, added, deleted, or edited. Which produced 500 verified mapped-lines.

2- The dataset files provided by the professor: where he also provided the XML outputs files for the dataset, so we added a feature in our line-mapper code where it produces an XML file output. Then we built a program where it compares both the XML file generated by the line-mapper code and the XML file give by the professor (both are for the same file pair topic).This code program counts all the matched lines and the mismatched lines, and it gives the overall accuracy percentage of the two XML output files.

C. *Accuracy Results*

To understand the algorithm of the accuracy scale in the XML file comparer code, we used this metric:

$$\text{Accuracy} = \frac{Number\ of\ the\ correct\ mappings}{Total\ Number\ of\ Mappings} \times 100\%$$

So Assume we identified 400 mapping line to be correct out of our 500 mapping lines from our dataset, then the overall accuracy will be:

$$\text{Accuracy} \approx \frac{400}{500} \times 100\% = 80\%$$

This result will change based on the amount of lines are mapped and the whether the line mapper code gets modified.

In this case, the professor's XML files only shows the effected lines while our XML outputs shows every single line that was mapped, which explains why the accuracy scale tend to be a bit lower than expected.

Factor	Effect on Accuracy	Reason
Format change	Decreases	Line comparisons detect diff even if logic is unchanged
Renaming Variable	Decreases	Line matching fail because text no longer matches
Reordered lines/ functions	Significant decrease	Treats reordered content as a mismatch
Added or removed lines	Decreases	Reduces the matched line ratio
Structural code changes	Decreases	Drops when control flow changes
Logic changes	Major Decrease	Semantic comparisons means a mismatch as well

Comments added/ removed	Decreases	Affects in the accuracy scale
Identical code with diff layout	Slight Decrease	Depends on the metric

Fig. 1. Demonstration of how the accuracy scale works.

D. *Discussion and Improvements*

Overall, our tool performs well for basic line change detection. Especially with the unchanged lines and added/deleted lines. However, our tool becomes less accurate when it comes to complex line mappings, such as lines that were moved in the program. In this case, the function *difflib* in our python code treats the line as line replacement rather than a line modification, which makes map complex.

Despite these limitations, we can improve our tool by refining the way our tool detects the line changes between the versions of the file or add a token-based matching to make the tool language-aware which improves the edit changes in the lines we map. This improvement will help reduce the error rate of the line-mapping and increase the accuracy rate of the line-maps.

IV. Presentation of the Line Mapping Information

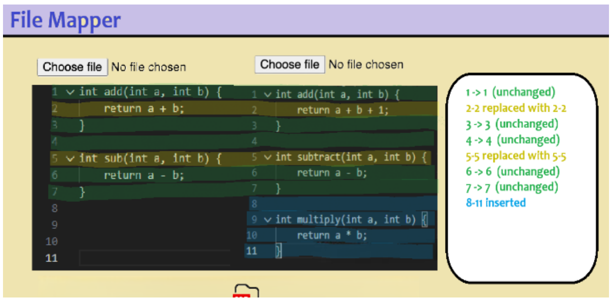
After discussing the functionality of our project, this section describes how the line-mapping results are displayed in GUI.

A. *GUI Design*

The GUI follows a style that's similar to version control, where the left panel shows the old version of the file while the right panel shows the new version of the file. Each line from old file correspond to the line from the new file using different colors based on each case. We used colors to make it easier for the user to read and understand the line-mapping concept.

B. *GUI example*

Below are simple GUIs for simple codes example from our dataset representing our line-mapping algorithm:

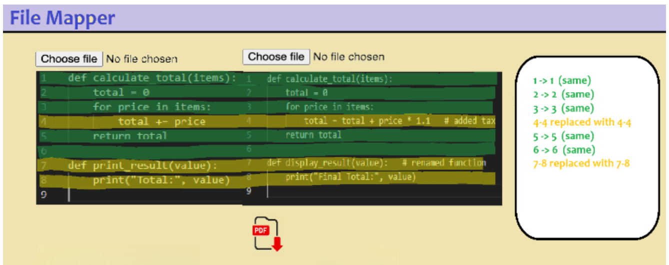


```
1  int add(int a, int b) {
2      return a + b;
3  }
4
5  int sub(int a, int b) {
6      return a - b;
7  }
8
9
10
11
```

```
1  int add(int a, int b) {
2      return a + b + 1;
3  }
4
5  int subtract(int a, int b) {
6      return a - b;
7  }
8
9  int multiply(int a, int b) {
10     return a * b;
11 }
```

```
1 -> 1 (unchanged)
2-2 replaced with 2-2
3 -> 3 (unchanged)
4 -> 4 (unchanged)
5-5 replaced with 5-5
6 -> 6 (unchanged)
7 -> 7 (unchanged)
8-11 inserted
```

Fig. 2. GUI for code example 1.



```
1  def calculate_total(items):
2      total = 0
3      for price in items:
4          total +– price
5      return total
6
7  def print_result(value):
8      print("Total:", value)
9
```

```
1  def calculate_total(items):
2      total = 0
3      for price in items:
4          total - total + price * 1.1   # added tax
5      return total
6
7  def display_result(value):    # renamed function
8      print("Final Total:", value)
9
```

```
1 -> 1 (same)
2 -> 2 (same)
3 -> 3 (same)
4-4 replaced with 4-4
5 -> 5 (same)
6 -> 6 (same)
7-8 replaced with 7-8
```

Fig. 3. GUI for code example 2.

C. *How to Intrepret the GUI*

Users can read this visualization example by reading and looking at the color scale we added on the side of the panel, where each color represents a case from the line-mapping method:

- Green – means code line was unchanged from the old version to the new version of the file.
- Blue – means code line was inserted in the new version of the file.
- Yellow – means the code line has been edited in the new version of the file.
- Red – means the code line has been deleted from the old version of the file.

Using colors as a guide to read this visualization is a good advantage since it makes it easier for the user to read and understand the functionality of this project.

References

[1] Python Software Foundation, "os – Miscellaneous operating system interfaces." https://docs.python.org/3/library/os.html

[2] Python Software Foundation, "difflib – Helpers for computing deltas." https://docs.python.org/3/library/difflib.html

[3] I. Sommerville, *Software Engineering*, 9th ed. Boston, MA: Addison-Wesley, 2010.

[4] Kaggle, "Dataset – Machine Learning & Data Science." https://www.kaggle.com/datasets

[5] Git Documentation, "Git Diff and Merge Algorithms," https://git-scm.com/docs

[6] C. Collberg et al., "Measuring Software Evolution Using File-Level Similarity", IPCP 2014

[7] Spinellis, D., "Version Control Systems", IEEE Software, 2005.

[8] "Python difflib Module Explained with Examples," Youtube Video, 2019.
Available: https://www.youtube.com/watch?v=FxdXEufiZoI

[9] "How to Create a Simple but Effective Diff-Tool in Python," Youtube video, 2018.
Available:https://www.youtube.com/watch?v=akYZmyHDkO8

[10] Microsoft, "XML Diff and Patch Tool," Microsoft Docs, 2022.
Available: https://learn.microsoft.com

[11] IBM, "XML Diff and Merge Tool," IBM Developer Documentation, 2017. Available: https://developer.ibm.com

[12] Fireship, "UI Design Fundementals," Youtube, 2021.
Available: https://www.youtube.com/watch?v=Qn7FAwCvb64

[13] Programming with Mosh, Python Tutorial – Python Full Course for Beginners, Youtube, 2022.
Available: https://www.youtube.com/watch?v=_uQrJ0TkZlc