LHDiff: Tracking Source Code Lines To Support Software Maintenance Activities

Muhammad Asaduzzaman Chanchal K. Roy Kevin A. Schneider Massimiliano Di Penta†
Department of Computer Science, University of Saskatchewan, Canada
†Department of Engineering, University of Sannio, Italy
{md.asad, chanchal.roy, kevin.schneider}@usask.ca, dipenta@unisannio.it

***Abstract*—Tracking lines across versions of a file is a necessary step for solving a number of problems during software development and maintenance. Examples include, but are not limited to, locating bug-inducing changes, tracking code fragments or vulnerable instructions across versions, co-change analysis, merging file versions, reviewing source code changes, and software evolution analysis. In this tool demonstration, we present a language-independent line-level location tracker, named LHDiff, that can be used to track lines and analyze changes in various kinds of software artifacts, ranging from source code to arbitrary text files. The tool can effectively detect changed or moved lines across versions of a file, has the ability to detect line splits, and can easily be integrated with existing version control systems. It overcomes the limitations of existing language-independent techniques and is even comparable to tools that are language dependent. In addition to describing the tool, we also describe its effectiveness in analyzing source code artifacts.**

***Index Terms*—differencing tools; line tracking; language-independent differencing tool**

I. Introduction

Software maintenance activities often require tracking source code lines across versions of a software system. One reason may be to separate changed lines from those that are deleted from an old version of a file or added to its new version during code review. If a bug has been identified in a software system, tracking lines containing the bug can assist us in locating the bug in subsequent versions. Mining version archives for co-changed lines can recommend line locations after a change. Source location tracking techniques can help in this regard to obtain an accurate estimation of changed lines. Results obtained from line tracking techniques can be used to track higher level language constructs. For example, the evolution of functions, methods or classes can be tracked across versions using location mapping data of changed lines. In addition, a line tracking tool can be used to track the evolution of other kinds of software artifacts, such as requirements, design documents or configuration files.

Versioning systems (such as CVS or Subversion) and many development environments rely on *Unix diff* or its variants to track lines between two versions of a text file. *Diff* reports the minimum number of added, deleted or changed lines between two file revisions. However, it has limitations in detecting reordered or changed lines that prevents it from becoming an ideal line tracking tool [3]. To address the problems, a number of line tracking techniques and tools have been developed [1]. For example, Canfora *et al.* developed the *ldiff* tool that uses a combination of information retrieval techniques and Levenshtein distance to track lines across versions of a file independent of source code language [2], [4]. In addition to describing various source location tracking techniques, Reiss recommends W_BESTI_LINE to track lines which uses context and content similarity to track lines [9]. Spacco and Williams developed another tool, SDiff, that accesses the syntactic structure of Java source files to track lines across versions [10]. All these techniques are sensitive to the degree and kind of source code changes. While techniques that use an Abstract Syntax Tree (AST) can provide fine grain change information, they require the source file to be parsed which may not always feasible. For example, developers may issue a file differencing command in the middle of editing a file, which may not be possible to parse at that point in time.

In this paper, we describe the *LHDiff* tool that can track source code lines across versions of a software system. It does not consider the AST of source files, and thus can be applied to arbitrary text files. The tool takes two versions of a file as input and utilizes *Unix diff* to track lines other than those that are reported as either deleted from the old file or added to the new file by *diff*. To determine mapping between the set of added and deleted lines reported by *diff*, the tool uses both context and content of a line. While the line itself represents the content, the context is computed by combining its top and bottom four lines. However, to make the mapping process faster, it leverages the simhash technique [5], [8] to compute mapping candidates for each deleted line of the old file. Next, it computes the context and content similarity using source code lines to select one from each set of mapping candidates. Since the algorithm and its evaluation with other state-of-the-art line tracking techniques have been discussed in a separate paper [1], we briefly summarize the algorithm in this tool paper and explain tool syntax.

The remainder of the paper is organized as follows. Section II provides examples that motivate us developing the tool. Section III briefly describes the *LHDiff* tool. Section IV briefly summarizes the tool effectiveness, while Section V describes its syntax and usage. Finally, Section VI concludes the paper.

II. Motivating Examples

Fig. 1 shows four different examples, each one showing two different versions of a file. Due to space limitations, we only show code fragments from the files (except Fig. 1-a).

Fig. 1: Line tracking examples.

The highlighted lines are the ones we are interested in and the correct mapping of those lines are marked with arrows. Fig. 1-a shows an example of a line split. In the new version of the file in Fig. 1-b, the highlighted line is moved outside the for loop. Fig. 1–c shows an example of a method split, where lines from the *readFile* method are moved to another method in the new version. In the last example (Fig. 1-d), the variable *arrayTb* is replaced with *arrayType* and the condition part of *if* block is updated.

Unix diff fails to provide a correct mapping in all four examples. Lines that are changed, or moved are reported as either added or deleted by *diff*. Although existing state-of-the-art line tracking techniques provide better results than diff, they have their own limitations. While *SDiff* was able to detect line splits, it fails to detect lines that are moved to another method (see Fig. 1-c). *Ldiff* fails to detect line splits or mapping of lines in the last example (Fig. 1-d). The technique recommended by Reiss is also not an exception. This shows the limitation of existing source location tracking tools in detecting changed or moved lines and motivates us to develop an enhanced line tracking tool.

III. LHDiff: Tracking Source Code Lines

Unlike *SDiff* that requires parsing source files, *LHDiff* is purely textual in nature and can be applied to arbitrary text files. It is a hybrid technique because it leverages findings collected through analyzing incorrect mappings of existing state-of-the-art location tracking techniques. The tool works in five different phases and they are briefly summarized below (further details about the LHDiff approach/algorithms can be found in a related research paper [1]):

- **Step 1** involves normalization of the input files where the objective is to remove editing differences that are only cosmetic in nature. For example, multiple spaces are replaced with only one.
- **Step 2** applies *Unix diff* to determine the set of unchanged lines. *Diff* reports list of lines that are deleted from the old file and added to the new file. We refer to them as the left and right lists. Since *diff* often fails in detecting changed and moved lines, we need to find the missing link between the lines of these two lists.
- **Step 3** acts as an intermediate step to speed up the mapping process without sacrificing accuracy. For each line in the left list we need to find a line from the right list, if there exists any mapping for that line. We leverage content and context of lines to determine the correct mapping. The line itself represents the content and context is calculated by concatenating its top and bottom four lines (see Fig. 3). To speed up the mapping process, we add an intermediate step instead of acting on source code lines in the first place. For each line in the left list we determine a small set of lines from the right list that are the probable mapping candidates. Instead of working on raw lines, we first apply a hash function to determine a 64 bit binary simhash value for both context and content. A simhash is nothing but a short binary representation of a much longer string with an important property that simhash values of two similar strings have a small Hamming distance. The content and context similarity scores are calculated by calculating the Hamming distance between their corresponding simhash values. If a and b are two binary strings, the Hamming distance is the number of 1s in $a \oplus b$ (the bitwise exclusive

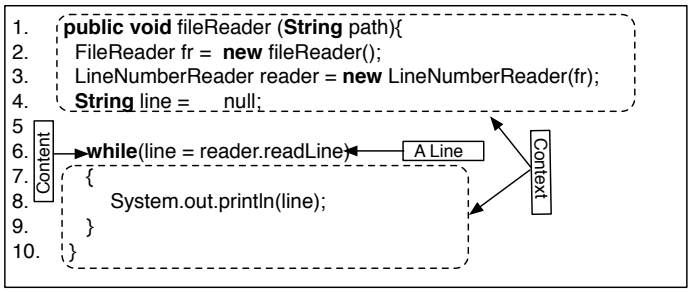

Fig. 3: An example of finding context and content of a line.

Fig. 2: Working steps of LHDiff.

OR between a and b). We now calculate the combine similarity score by taking 0.6 times the content similarity and 0.4 times the context similarity for all pairs of lines between left and right lists. For each line in the left list, we then determine a set of lines that constitute the possible mapping candidates using a predefined threshold value.

- **Step 4** selects one line from each set of mapping candidates. For each line from the left list and its mapping candidates, we select the line pair that gives the highest similarity score, but this time the similarity scores are calculated on the raw lines instead of on the simhash values. The content similarity between a pair of lines is calculated by applying the Levenshtein edit distance, and the context similarity is calculated using cosine similarity. A combined similarity score is then calculated by combining both context and content similarity.
- **Step 5** deals with detecting line splits. This step can be enabled/disabled through a command line option or programatically. To detect line splits, *LHDiff* does the following. For each unmapped lines of the old file, it repeatedly concatenates lines of the new file, one at a time and starting from the top unmapped line of the old file, and calculates the Levenshtein edit distance similarity. We repeat the concatenating operation until the similarity value starts dropping. If such a similarity value crosses a predefined threshold value, we map all the concatenated lines of the new file to that line of the old file.

Fig. 2 shows an example that explains working steps of *LHDiff*. We omit the preprocessing step from the figure for the sake of simplicity.

IV. Performance

This section briefly summarizes the performance of *LHDiff*. Details about evaluation procedure and detailed results can be found in a separate paper [1]. In the following, we also highlight the benefits of using the simhash technique, and also warn users about the limitation of the tool.

A. Correctness of tracking lines and time requirements

We compared *LHDiff* with other state-of-the-art line tracking tools using three different benchmarks (Reiss, Eclipse and NetBeans) where the lines are collected from real-world applications. Our evaluation results reveals that *LHDiff* outperforms all language dependent techniques and even a language dependent technique, *SDiff*. For example, while the tool correctly maps 82.8% of target lines in Eclipse benchmark,

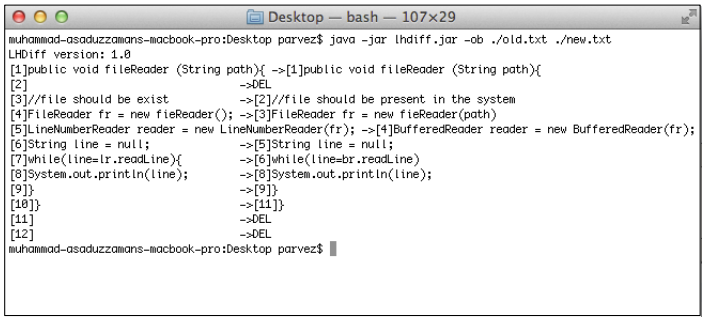

Fig. 4: Tool output example.

where the files underwent changes to a higher degree than other benchmarks, the closest score to ours is 74.1% by *SDiff*.

In terms of execution time, *LHDiff* is comparable with other state-of-the-art techniques. Although *LHDiff* may require slightly more time than some of the techniques, this small additional time gives the tool the ability to achieve higher accuracy.

B. Performance gain from the simhash technique

The tool leverages the simhash technique to filter mapping candidates for lines from the left list because calculating similarity scores at the line level between all pairs of lines of left and right lists is computationally expensive, and this is particularly true for what concerns the Levenshtein edit distance. To better understand the effect of simhash, we ran *LHDiff* without enabling step-3 on the files in Reiss benchmark [9], which consists of 25 revisions of `ClideDataManagerManager.java` file. While the original *LHDiff* tool took 5.22 sec. to complete tracking line locations, the modified version took 18.06 sec. to complete running. This indicates more than three times improvement of running time from using simhashing, and it can be even more when differencing a large number of source code files.

C. Limitations

LHDiff fails to track lines when both context and content of a line changes a lot. The combined similarity score gives more weight on the content similarity and this sometimes leads to an incorrect mapping for short lines. For example, even if the context suggests that there is no connection between line 13 of the old file to line 26 of the new file, content similarity gives a very high score, because of the matching keywords and the length of the keywords dominating the total line length which results in an incorrect mapping (see Fig. 5).

V. Tool Implementation

An implementation of the *LHDiff* tool is available in Java as a command line tool [7]. The tool supports the use of various similarity/distance metrics (Levenshtein, Cosine, Jaccard, and Dice) and Table I summarizes the configuration options *LHDiff* has available. Fig. 4 shows an example of running the tool from the command line, where the input files are the same ones we used in Fig. 2.

TABLE I: LHDiff configuration options.

Option	Description	Default Settings
-i	Ignore case differences	disabled
-k	The size of mapping candidate set	15
-p	Context weight ($0 \leq CXW \leq 1$) and the threshold value ($0 \leq TH \leq 1$) for combine similarity score used in Step 4. Content weight will be automatically set to $1 - CXW$	0.4 and 0.6
-cnm	Line content similarity metric	Levenshtein
-cxm	Line context similarity metric	Cosine
-cxs	Context size	4
-ls	detect line split	disabled
-ob	Output both line number and content	display only line number

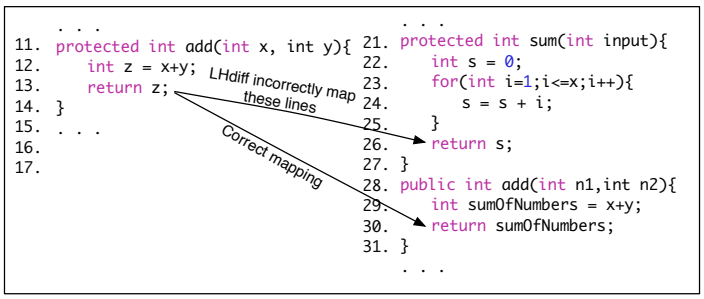

Fig. 5: An example of incorrect mapping.

VI. Conclusion

This paper describes *LHDiff*, a line tracking tool that can be applied to any text files, can easily be integrated with existing version control systems, and can even detect line splits. The demonstration will show how *LHDiff* can be used to analyze source code or other text files, track buggy lines, and examples from real world applications where *LHDiff* succeeds in mapping lines but *Unix Diff* or other state-of-the-art line tracking techniques fail(s). As a work-in-progress, we are investigating how to reduce the degree of false mappings, and developing visualizations to aid comprehending the results from the tool.

References

[1] M. Asaduzzaman, C. K. Roy, K. A. Schneider, M. Di Penta, "LHDiff: A Language-Independent Hybrid Approach for Tracking Source Code Lines", accepted to be published in ICSM, 2013
[2] G. Canfora, L. Cerulo, and M. Di Penta, "Tracking Your Changes: A Language-Independent Approach", in IEEE Softw., pp. 50-57, 2009.
[3] G. Canfora, L. Cerulo, M. Di Penta, "Identifying Changed Source Code Lines from Version Repositories", in MSR, pp.14, 2007.
[4] G. Canfora, L. Cerulo and M. Di Penta, "Ldiff: An enhanced line differencing tool", in ICSE, pp. 595 -598, 2009.
[5] M. S. Charikar, "Similarity estimation techniques from rounding algorithms", in STOC, pp. 380-388, 2002.
[6] M. Kim and D. Notkin, "Program element matching for multi-version program analyses", in Proceedings of the 2006 international workshop on Mining software repositories, pp. 58-64, 2006.
[7] "The LHDiff Tool" http://asaduzzamanparvez.wordpress.com/Research
[8] G. S. Manku, A. Jain and A. D. Sarma, "Detecting Near Duplicates for Web Crawling, in WWW, pp. 141-150, 2007.
[9] S. P. Reiss, "Tracking source locations", in ICSE, pp. 11-20, 2008.
[10] J. Spacco and C. Williams, "Lightweight Techniques for Tracking Unique Program Statements", in SCAM, pp. 99–108, 2009.