LHDiff: A Language-Independent Hybrid Approach for Tracking Source Code Lines

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 source code lines between two different versions of a file is a fundamental step for solving a number of important problems in software maintenance such as locating bug introducing changes, tracking code fragments or defects across versions, merging file versions, and software evolution analysis. Although a number of such approaches are available in the literature, their performance is sensitive to the kind and degree of source code changes. There is also a marked lack of study on the effect of change types on source location tracking techniques. In this paper, we propose a language-independent technique, *LHDiff*, for tracking source code lines across versions that leverages simhash technique together with heuristics to improve accuracy. We evaluate our approach against state-of-the-art techniques using benchmarks containing different degrees of changes where files are selected from real world applications. We further evaluate *LHDiff* with other techniques using a mutation based analysis to understand how different types of changes affect their performance. The results reveal that our technique is more effective than language-independent approaches and no worse than some language-dependent techniques. In our study *LHDiff* even shows better performance than a state-of-the-art language-dependent approach. In addition, we also discuss limitations of different line tracking techniques including ours and propose future research directions.**

***Keywords*-diff; line tracking; lightweight; levenshtein;**

I. Introduction

Tracking source code across multiple versions of a program is an essential step for solving a number of problems related with multi-version program analysis. For example, consider the problem of locating bug introducing changes. Existing techniques solve this problem by finding the lines that are affected through bug fixes and then trace back those lines to determine their origin. If a bug has been identified in a software system, tracking lines containing the bug in the subsequent versions can help us determine whether the same problem persists in the next versions and if yes, allows developers to fix the problem at ease. Results obtained from a line tracking technique can be aggregated for fine-grained evolutionary analysis. For example, clone evolution analysis requires tracking clone fragments across multiple versions of a software system. Tracking lines of a clone fragment can help us understand how that fragment evolves over time. Such analysis can also guide us in understanding the nature, effects, and reasons for cloning. To support collaboration in software development, annotation or tagging has been used in source code that can facilitate both navigation and coordination [28], and source location tracking techniques can help manage tags across versions. Software development involves more than just the creation of source code. There are also different kinds of software artifacts that can benefit from mapping lines across versions. Possible applications are, but not limited to, studying when and how requirements are changed or ensuring whether intended changes have been applied to all configuration files.

A number of line tracking techniques can be found in the literature. Reiss [20] listed a number of approaches for tracking lines across versions and found that language-independent approaches often provide good results. Canfora et al. [4] proposed another language-independent line matching technique, called *ldiff*, that uses a combination of information retrieval techniques and Levenshtein distance for mapping lines. Techniques that take into account the syntactic structure of source files can provide change information at a more fine-grain level [9]. An example of a line tracking technique that falls in this group is *SDiff* [27] that can determine changes at method and identifier levels. If fine-grain source code change information is not required, language-independent text-based approaches are suitable for tracking source code lines and can be applied to different kinds of documents besides source code (e.g., test cases or use cases). They also have the potential to be integrated with existing version control systems with little or no modification. However, the performance of these techniques vary depending on the degree of changes applied to the source code. In a recent comparative study of source location tracking techniques, William and Spacco[27] found that techniques that performed well in the Reiss study did not perform well against their benchmark in which files had a higher degree of changes. This motivates us to investigate the reasons behind this discrepancy and to device a robust language-independent solution.

This paper introduces *LHDiff*, a language-independent technique to track the evolution of source code lines across versions of a software system. The technique uses *Unix diff* between two different versions of a file to determine the set of unchanged lines. To track the remaining lines, it uses a combination of context and content similarity. However, to speed up the mapping process, it first leverages *simhash* technique to determine a list of mapping candidates for each deleted line in the old file. Next, the technique computes similarity scores again, but this time on the source code lines instead of the *simhash* values, to select one line from each set of mapping candidates. To validate the effectiveness of our language-independent technique, we compare it against state-of-the-art techniques using different benchmarks where the files are collected from real world applications. We further evaluate our technique with other approaches using different types of changes in a mutation based analysis. The experimental results in both cases suggest that the technique is superior to other language-independent approaches, and even often gives better result than the language-dependent technique *SDiff*.

The remainder of the paper is organized as follows. Section II covers previous work related with our study. Section III describes our hybrid line mapping technique. Section IV presents a quantitative evaluation of line tracking techniques using three different benchmarks. In Section V, we describe results of mutation based analysis. Section VI explains some threats to our study and finally, Section VII concludes the paper with future research directions.

II. Related Work

Tracking source code lines across program versions is crucial to support various maintenance activities and several approaches exist that consider line content, context, abstract syntax tree, edit distance or a combination of these techniques to solve the problem. In general, existing techniques can be divided into two categories: (1) text-based and (2) syntax tree-based. The first group of techniques is purely textual in nature and does not require parsing source files. As a language-independent technique the *Unix diff* algorithm has been widely used in many studies, not only to track lines but also for program differencing. *Diff* is based on solving the problem of longest common subsequence and it reports the minimum number of line changes that can convert one file to another. However, it has its limitations. For example, it cannot detect reordered lines. The addition of comments to a line can cause *diff* to report deletion of the old line and addition of a new line. However, such cosmetic changes are irrelevant to a programmer and should be ignored [13].

Diff reports regions of file lines that differ between a pair of files where each region is called a hunk. Zimmermann et al. [32] addressed this modification changes using an annotation graph where large modifications are considered as combined addition and deletion of lines, otherwise all lines between hunk pairs are connected with each other in a modification. The technique detects origins conservatively, does not consider the issue of reordered lines and is susceptible to errors.

Canfora et al. [5], [4] developed a line differencing technique, called *ldiff*, to track line locations independent of languages. Their technique uses the *Unix diff* algorithm to determine the unchanged lines. After that, set-based and sequence-based metrics are used to complete the mapping of remaining lines. However, they only compared the technique with *Unix diff* algorithm. To the best of our knowledge, Reiss was the first to conduct a study to evaluate the performance of several techniques for tracking source locations [20]. Interestingly, the result reveals that simple techniques like the one mentioned above that do not consider program structure perform better than those that consider syntactic structure of source files (like abstract syntax tree-based techniques). Reiss also recommended the *W_BESTI_LINE* technique to track line locations, which uses a combination of context and content similarity to find the evolution of lines independent of languages. While *LHDiff* is also language-independent, our technique differs from the above approaches in that it uses the *simhash* technique to speed up the mapping process and a set of heuristics to improve the effectiveness of tracking source locations.

Spacco and Williams [27] extended the idea of tracking lines for tracking program statements across multiple revisions of Java code. They developed *SDiff*, an abstract syntax tree based technique that leverages tokenization, *Unix diff* and *Kunhn-Munkres* algorithm [17] to complete tracking line locations. *SDiff* is a great algorithm to determine differences at the line level. However, the technique cannot be applied to arbitrary source code and cannot handle comments. While comments are not executed, their importance cannot be ignored. Tags [28] are usually associated with comments to support asynchronous collaboration and they need to be tracked across versions. Moreover, *SDiff* requires that the source code to be parsed without any error. However, in reality this can not be guaranteed. For example, the source code may be written using an old grammar of a language or developers may issue file differencing commands in the middle of an edit operation. *LHDiff* on the contrary is a language-independent technique, can be applied to arbitrary source code, and can track the evolution of comments/tags across versions.

Line location information can be obtained through techniques that determine fine-grain differences between two versions of a file. However, these techniques require knowledge about language constructs. Among these approaches, most notable is the ChangeDistiller [9], which considers the abstract syntax tree of a Java source file and leverages a tree differencing algorithm to determine fine-grain changes in the source code. The algorithm is not immune to cosmetic changes (changes that do not affect the behaviour of a program like addition of a comment), cannot work on arbitrary text files and is limited to Java files only. Xing and Stroulia [31] presented an algorithm, known as UMLdiff to determine structural changes in object-oriented software. It uses a combination of name similarity and structure similarity measures for recognizing conceptually the same entities. Apiwattanapong [2] presented an algorithm to determine changes between two Java programs. The technique considers program structure and semantics of programming language constructs to determine changes that are difficult to detect with a pure textual differencing technique. While these approaches are language specific and focus on fine-grain change details, *LHDiff* focuses on tracking lines independent of source code languages.

Techniques for tracking program elements across versions are also related with our study. Matching higher level language constructs prior to mapping lines improves mapping quality. A comprehensive survey of various techniques can be found in the work of Kim et al. [16]. Godfrey and Zou [11] developed a semi-automatic technique to detect merging and splitting of source code entities during the evolution of a software system. Kim et al. [15] used a combination of textual similarity and a location overlapping score to track clone fragments across versions. Duala-Ekoko and Robbillard developed a technique to track the evolution of clones [8]. The technique is also available as an Eclipse-plugin, called Clone Tracker. Here, clones are identified using an abstract clone region descriptor (CRD) that is independent of source code line locations. While the above techniques focus on tracking code fragments, we focus on tracking individual lines.

III. LHDiff: A language-independent hybrid line tracking technique

This section introduces *LHDiff*, our language-independent hybrid line tracking technique. Figure 1 summarizes the entire mapping process.

A. Preprocess input files

The algorithm starts with reading lines from two different versions of a file. A large number of changes in source code are only cosmetic in nature and do not change the behaviour of a program. Examples include changes to whitespace/newline characters. They are inserted to change the indentation of a program to improve readability. To ignore such changes each line of the source file is normalized so that multiple spaces are replaced by only one. We also remove all parentheses and punctuation symbols from the text except curly braces because we obtained best results when considering them as part of the line context.

B. Detect unchanged lines

After preprocessing we apply *Unix diff*, which uses the longest common subsequence algorithm to determine the set of unchanged lines. We use *diff* because previous studies report that it can detect the set of unchanged lines with great accuracy [20]. *Diff* reports the sequences of lines that have been deleted or added between the files. We store the list of deleted and added lines into two different lists. From now on, we refer them as the left and right lists correspondingly.

C. Generate Candidate List

Now, our goal is to determine the mapping of a line from the left list to that of the right list. Similar to *W_BESTI_LINE* (recall that *W_BESTI_LINE* is another language-independent source location tracking technique), we use both context and content of a line to determine the correct mapping. The line itself represents the content and the context is created by concatenating four lines before and after the target line. While the content similarity between a pair of lines is calculated using normalized Levenshtein edit distance which considers the order of characters in them, the content similarity is calculated applying cosine similarity that does not consider the order of tokens/characters. While building context we ignore blank lines, but keep the curly braces. Such changes allow us to gather sufficient contextual information for a line. We then determine a combined similarity score by considering both content and context similarity. We need to calculate such scores between all possible pairs of deleted and added lines. After that we map only those lines that provide the highest similarity scores and also exceed a predefined threshold value. However, the complete operation would take a long time to complete because of the complexity associated with computing similarity scores, particularly the normalized Levenshtein edit distance. To improve the running time of the algorithm we follow a different strategy. We first apply a form of locality sensitive hashing and calculate the combined similarity score of the hash values instead of the original lines to determine a small set of possible mapping candidates for each line of the left list. Then we use the original line content and combined similarity score to select a line from each set of mapping candidates. Since the hashing technique reduces large data into a much shorter sequence of bits, the overall mapping time is reduced significantly.

While a cryptographic hash function tries to avoid generating the same key to ignore collisions, in this form of hashing files containing similar content are mapped to identical or very similar binary hash keys. The technique is known as *simhash* [7] and it has been found that the technique is practically useful to determine near-duplicate pages in a large collection of documents [18]. The core of the algorithm uses a hash function to generate *simhash* values. Among various non-cryptographic hash functions we use *Jenkin* hash function since it shows better similarity preserving behaviour compared to other functions and also found effective in detecting near-miss code fragments in other studies [1], [29], [30]. We generate a 64 bit *simhash* value for both context and content using the *simhash* algorithm [25]. Instead of working on the original lines, we are now working on the *simhash* values. We now calculate the context and content similarity for each pair of added and deleted lines by calculating the Hamming distance between their corresponding *simhash* values. While the Hamming distance between two strings is the number of substitutions required to convert one string into another, for binary strings (a and b) Hamming distance is calculated by counting the number of 1 bits in $a \oplus b$ (bitwise exclusive OR operation between a and b). The smaller the Hamming distance is, the closer the two strings are (see Figure 2). The context and content similarity values are normalized between zero and one. A combined similarity score is calculated for each pair of lines using 0.6 times the content similarity and 0.4 times the context similarity (this is determined after experimenting with different combinations of values). For each line on the left list, we then determine k-neighbours from the right list that are the most probable mapping candidates of that line based on the combined score. We refer this set as the matching candidates list. Since we are not comparing raw source code lines for selecting the mapping candidates, this saves significant time. During our study with different values of k we found that $k = 15$ is a good choice to work with.

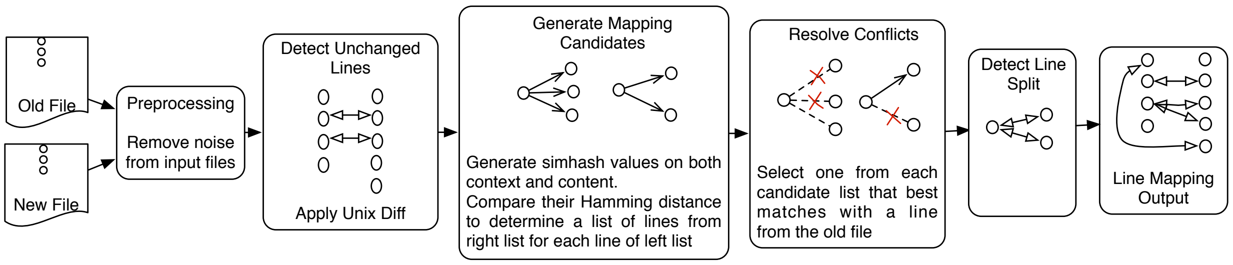
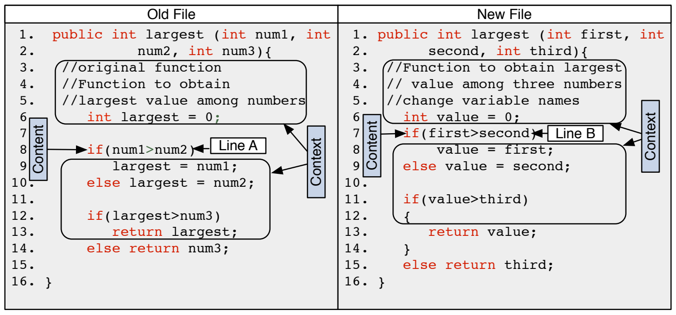

Old File
New File
Preprocessing
Remove noise from input files
Detect Unchanged Lines
Apply Unix Diff
Generate Mapping Candidates
Generate simhash values on both context and content.
Compare their Hamming distance to determine a list of lines from right list for each line of left list
Resolve Conflicts
Select one from each candidate list that best matches with a line from the old file
Detect Line Split
Line Mapping Output

Fig. 1: Summarizing the line mapping process in LHDiff

D. Resolve conflicts

We now have a candidate list for each line of the left list (except those that are detected as deleted/unchanged by the algorithm), but we do not know the exact mapping yet. The objective of this step is to select one line from each candidate list to resolve the conflicts. As an example, consider that we are looking for the mapping of line 20, and from the previous step we determine that the candidate list consists of three lines $(37, 46, 51)$. We now use the original lines to generate both context and content, and the algorithm determines the combined similarity score between each possible mapping pairs $(\{20, 37\}$, $\{20, 46\}$,and $\{20, 51\})$. It selects the one that gives the highest similarity score and also exceeds a predefined threshold value. We now use normalized Levenshtein distance to measure context similarity and cosine similarity to measure content similarity. Both the values are normalized and the combined similarity score is determined using 0.6 times the content similarity and 0.4 times the context similarity. These were the same values used by Reiss. We set the threshold value to 0.45 after experimenting with various other values because at this setting *LHDiff* provides best result.

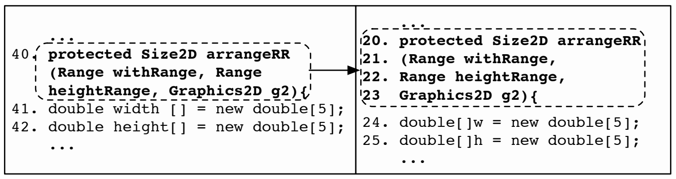


```
public int largest (int num1, int
              num2, int num3){
//original function
//Function to obtain
//largest value among numbers
   int largest = 0;

   if(num1>num2)
      largest = num1;
   else largest = num2;

   if(largest>num3)
      return largest;
   else return num3;

}
```

```
public int largest (int first, int
        second, int third){
//Function to obtain largest
// value among three numbers
//change variable names
  int value = 0;
  if(first>second)
       value = first;
  else value = second;

  if(value>third)
  {
     return value;
  }
  else return third;
}
```

Context	Simhash	Hamming Distance
Line A	10001001101101010000000000001011	9
Line B	11100000101001010110000011001011	

Fig. 2: An example of calculating Hamming distance



```
   ...
40. protected Size2D arrangeRR
   (Range withRange, Range
   heightRange, Graphics2D g2){
41. double width [] = new double[5];
42. double height[] = new double[5];
   ...
```

```
    ...
20. protected Size2D arrangeRR
21. (Range withRange,
22. Range heightRange,
23  Graphics2D g2){
24. double[]w = new double[5];
25. double[]h = new double[5];
    ...
```

Fig. 3: An example of a line splitting

E. Detect line splits

The last part of our technique deals with detecting line splitting. We use the term line splitting instead of statement splitting since *LHDiff* is not aware of the boundary of a statement, and only works at the line-level. An example of line splitting is shown in the Figure 3 where a single line breaks into multiple small lines. The basic *LHDiff* algorithm tries to map each line from the old file to a line in the new file, but fails to map some lines where the textual similarity differs a lot. To track lines affected by the line splitting we use the following approach. For each unmapped line of the new file, we repeatedly concatenate the line to the successive lines, one at a time, and determine normalized Levenshtein distance (LD) with another unmapped line in the old file until the similarity value starts to decrease. Then, if the textual similarity between the concatenated lines and the left side line crosses a predefine threshold value we map them. As an example, we concatenate line 20 with 21 and determine the normalized Levenshtein distance between R_{20+21} and L_{40}. The similarity value is greater than the similarity between R_{20} and L_{40}. Thus we concatenate the next line and determine the similarity again between $R_{20+21+22}$ and L_{40}. Since the similarity is again greater than the previous step we continue the concatenate operation until the similarity decreases. This happens as soon as we add line 24 $(LD(R_{20+21+22+23+24}, L_{40}) < LD(R_{20+21+22+23}, L_{40}))$. This indicates that line 24 cannot be a part of the split lines. Since $LD(R_{20+21+22+23}, L_{40})$ exceeds the threshold value and there are four lines in the concatenated part, *LHDiff* maps all these four lines to line 40 on the left side. To avoid false mapping we set the threshold value to a high value, 0.85, and we also limit the concatenation to a maximum of 8 lines. During our manual analysis we did not find any example where a line splits more than that. This heuristic approach can only compensate line splitting when such an operation does not change the contents of the line or changes only little. It cannot detect other complex line splitting operations, such as those described in Section V-A.

F. Evaluation

We evaluate the effectiveness of *LHDiff* using two different methods. First, we consider three different benchmarks, each of them containing line mapping information of versions of files where the files are collected from real world applications. Second, we also compare *LHDiff* with other state-of-the-art techniques using a mutation based analysis where we consider different types of changes. We describe details about these two evaluation methods in the following two sections.

IV. Evaluation using Benchmarks

This section describes the benchmarks we used to evaluate source location tracking techniques including results of our evaluation.

A. Experiment details

We used two different benchmarks available from previous studies to measure the effectiveness of source location tracking techniques. The first one was developed by Reiss, which contains location information of 53 lines of a file (ClideDatabaseManager.java) in 25 revisions and amounts to 1325 test cases [20]. We also evaluated our technique with another benchmark developed by Reiss that contains locations of 14 lines of the JiveRuntime.java file in 27 different revisions. However, we did not report the result in this paper because the changes are simple and there is no significant differences among source location tracking techniques for those changes. In both cases, lines were selected by looking at every tenth line of the source file discarding those containing blank lines. Additional problematic or interesting changes (such as name and comment changes) were also included. Williams and Spacco developed another benchmark (known as Eclipse benchmark) containing the change information of 232 lines [27]. We manually analyzed all changes. By doing this we not only validated changes but also identified interesting change patterns during the evolution of these lines. It should be noted that during our manual investigation we found a few incorrect mappings in the Eclipse benchmark. We used the benchmark in our evaluation after correcting those mappings. That is why readers will notice a slight difference in our results for Eclipse benchmark than what was reported in the original paper [27].

In addition, we also developed a third benchmark using source code from the NetBeans project [19]. NetBeans is an integrated development environment for developing applications using different languages. We randomly selected a number of lines (including those we found challenging) and then determined the new locations of those lines in another version. Since the decision is subjective in nature, to avoid bias the first two authors of this paper determined the correct mapping of these lines separately. Cases where there was a disagreement between the two authors, we removed that line from our study.

TABLE I: Three forms of incorrect mappings

Label	Meaning
Change	the algorithm finds a mapping of a line but the mapping is not correct
Spurious	the algorithm detects mapping of a line but the line is deleted
Eliminate	the algorithm detects deletion of a line but the line exists in the new file

Although our technique is language-independent, we considered techniques in both categories for comparison. We instructed *ldiff* to ignore all whitespaces and also to ignore changes whose lines are all blank. For all other settings of *ldiff* we used default values except we changed the number of iterations to four different values. When the value is set to 0, *ldiff* considers only line similarity. For all other positive values it considers hunk similarity. In the case of *SDiff*, we evaluated all different nine configurations used in the original experiment. For details of these configurations we refer interested readers to their original paper [27]. Among various techniques evaluated by Reiss we report the result of *W_BESTI_LINE*, with the same similarity thresholds and context length suggested by Reiss, because Reiss recommended this technique for tracking lines. We also include results of *diff*, configured to ignore spacing differences and cases of letters during the mapping process.

To calculate a score for each method, we followed the basic scoring mechanism used in previous studies where the score was calculated by determining the number of correct mappings. To document line mapping information, we used the following approach. If a line in the old file is deleted, the new location of that line is encoded with -1, otherwise each line of an old file is associated with a non-negative integer representing the new position of that line. Incorrect classifications can result from three different types of errors and we use different labels to signify them as shown in Table I. All experiments were conducted on a computer running with Ubuntu Linux that has a 2.93 GHz Intel Core i7 processor and 8 GB of memory.

B. Results

Table II shows the results of our evaluation. For each benchmark and line tracking technique we not only provide the percentage of correct mappings but also show results for each incorrect mapping type (these are: change, spurious and eliminate; see Table I for their definitions). From the table we can see that *LHDiff* performs better than both *SDiff* and *ldiff*. The default settings of *ldiff* uses cosine similarity for mapping hunks and Levenshtein distance for mapping lines. We considered all possible combinations of metrics and tokenizers. While for the Reiss benchmark *ldiff* correctly maps around 96.5% of lines, the performance drops significantly for the Eclipse benchmark. This is similar to the result provided by Spacco and Williams. On the other hand, although *W_BESTI_LINE* performs similar to *LHDiff* for the Reiss benchmark, accuracy drops to around 53% for the Eclipse benchmark. The latter contains a large number of changes as the files are heavily edited in it. The performance of *W_BESTI_LINE* and *ldiff* varies depending on the number of changes occured to the files. However, both *SDiff* and *LHDiff* are stable in nature. Although *SDiff* shows around 87% accuracy for the Reiss benchmark, according to the authors accuracy can be even up to 96% if we ignore curly braces and non executable statements. For the Eclipse benchmark, *SDiff* produces around 74% accurate result. However, *SDiff*

TABLE II: Running location tracking techniques against Reiss, Eclipse, and NetBeans benchmarks

Type	**Method**	**Reiss Benchmark**					**Eclipse Benchmark**					**NetBeans Benchmark**				
			Incorrect Mapping Categories					**Incorrect Mapping Categories**					**Incorrect Mapping Categories**			
		Correct%	**Change%**	**Spurious%**	**Eliminate%**	**Time [s]**	**Correct%**	**Change%**	**Spurious%**	**Eliminate%**	**Time [s]**	**Correct%**	**Change%**	**Spurious%**	**Eliminate%**	**Time [s]**
Language independent Techniques (Text-based approach)	W_BESTI_LINE	96.7	0	29.6	70.5	4.8	52.6	28.2	56.4	15.5	0.7	61.9	41.9	45.2	12.9	2.9
	LHDiff	**97.0**	**42.5**	**0**	**57.5**	**4**	**82.8**	**22.5**	**20**	**57.5**	**3.3**	**85.5**	**18.6**	**13.6**	**67.8**	**6.8**
	diff	92.5	0	0	100	0.2	41.0	0.7	0	99.3	0.1	48.4	1.4	0	98.6	0.2
	ldiff [-i 0]	96.4	0	66.7	33.3	25.7	62.5	9.2	1.2	89.7	58.2	74.0	16.0	7.6	76.4	6.1
	ldiff [-i 1]	96.4	0	25	75	73.9	59.1	9.5	1.1	89.5	209.2	76.4	14.6	5.2	80.2	103.5
	ldiff [-i 3]	96.2	0	29.4	70.6	118.6	72.0	20	4.7	75.4	419.7	80.6	24.1	8.9	67.1	171.5
	ldiff [-i 5]	96.2	0	29.4	70.6	160.7	72.4	20.3	4.7	75	472.4	81.3	25	9.2	65.8	218.2
	Git	92.5	0	0	100	0.8	45.3	0.8	0	99.2	0.3	53.7	1.6	0.5	97.9	0.2
Language Dependent Technique (Requires access to abstract syntax tree)	SDiff	86.2	0	86.3	13.7	1.9	70.7	11.8	80.9	7.4	1.7	71.8	7.0	86.1	7.0	4.2
	SDiff-probable	87.0	3.0	82.4	14.6	1.7	69.9	11.4	75.7	12.9	1.5	73.5	9.3	82.4	8.3	4.3
	SDiff-possible-probable	87.0	3.0	82.4	14.6	1.7	70.7	11.8	75	13.2	1.6	75.4	12	77	11	4.3
	SDiff-min	85.6	0	82.5	17.5	2.6	74.1	13.3	76.7	10	2	72.5	7.1	85.7	7.1	6.8
	SDiff-min-probable	86.4	2.9	78.6	18.5	2.6	73.3	12.9	74.2	12.9	2.1	74.2	10.5	81.0	8.6	6.8
	SDiff-min-possible-probable	86.4	2.9	78.6	18.5	2.6	74.1	13.3	73.3	13.3	2.1	75.2	12.9	74.3	12.9	6.8
	SDiff-token	85.6	0	82.5	17.4	1.0	73.3	11.3	79.0	9.7	1.0	71.0	6.8	87.3	5.9	2.3
	SDiff-token-probable	85.6	2.7	79.8	17.5	1.0	73.3	11.3	75.8	12.9	1.0	73.7	10.3	82.2	7.5	2.3
	SDiff-token-possible-probable	86.4	2.9	78.7	18.5	1.0	74.1	11.7	75	13.3	1.0	74.5	11.5	76.9	11.5	2.3

takes into account the structure of source files, requires the files to be syntactically valid, and is only available for Java. *LHDiff*, on the other hand, is purely textual in nature and can be applied to any files (be it a source file or a plain text file). Without considering structural information, *LHDiff* provides around 97% of correct mappings for the Reiss benchmark and for the Eclipse one the result is around 83%. Among the incorrect mappings we found for the Eclipse benchmark, the highest amounts (more than 57%) were due to elimination.

For the NetBeans benchmark (see Table II), *LHDiff* shows better result than the other techniques. While *LHDiff* detects around 86% correct mapping, *SDiff* detects only around 75%. Although *ldiff* detects more correct mappings (81%) than *SDiff*, it requires more computation time.

C. Discussion

In this section we first explain the reasons behind the poor results of *W_BESTI_LINE* and also *ldiff* on the Eclipse benchmark that originally motivated us to develop another language-independent source location tracking technique. We then compare *LHDiff* with the content tracking technique of *Git* and present another study result where we tried to improve the accuracy of *LHDiff* using tokenization.

1) Why technique recommended by Reiss or ldiff did not perform well with Eclipse benchmark?: The algorithm (*W_BESTI_LINE*) recommended by Reiss works fine as long as the context and content of a line do not change significantly. The technique fails when both of them go through a large amount of changes. Another possible threat to this approach is the addition of blank lines or lines containing stop list or punctuation symbols only. If a developer inserts blank lines before and/or after a line, the line becomes isolated and the context differs remarkably. If the line also changes significantly, the algorithm fails to map the old line to the new one. Reiss did not consider this issue. However, we can eliminate this problem by ignoring blank lines which can help locate proper contextual information. Another downside of this technique is that the cost of running the technique is high. For a line in the old version, the technique compares the line with every other lines in the new version and selects the one that provides the best matching. If the objective is to determine new locations of a few lines of the old file, then the technique may be adequate. However, situations where we need to map every line of an old file to the new file, then the approach may be expensive particularly when the size of the file is large. That is why we use the *simhash* technique in *LHDiff* to faster the mapping process. *W_BESTI_LINE* tries to map a subset of lines of an old file to the new file whose mappings are requested by a user. While this approach lessens the running time of the algorithm, it drops the accuracy of the technique, particularly where files have gone through a large number of changes. A line ($l_i^{old} \in f_{old}$) cannot be aggressively mapped to another line ($l_j^{new} \in f_{new}$) without considering the similarity of l_j^{new} to all other lines of the old file. Their might be another line ($l_k^{old} \in f_{old}$) to which the newly mapped line ($l_j^{new} \in f_{new}$) best matches. For these reasons *W_BESTI_LINE* performed poorly for the Eclipse benchmark.

The running time of *ldiff* improves because of applying *Unix diff* at the beginning to determine unchanged lines. A threat to the technique comes from the fact that before mapping lines *ldiff* tries to map a hunk from the old file to another hunk of the new file. If the hunk similarity exceeds a threshold value then a line mapping process begins. However, there is a possibility that hunk similarity does not exceed the threshold value because of their size difference and only a few

TABLE III: Results of LHDiff before and after applying tokenization

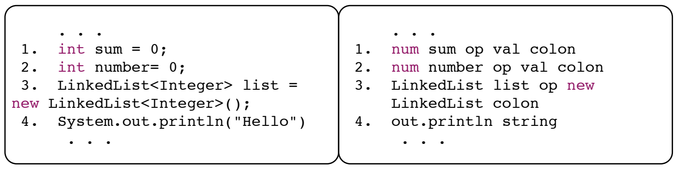
Benchmark	LHDiff	Correct%	Change%	Spurious%	Eliminate%
Reiss	Non-tokenize	96.98	42.50	0	57.50
	Tokenize	93.66	0	48.81	51.19
Eclipse	Non-tokenize	82.76	22.50	20	57.50
	Tokenize	82.76	47.50	32.50	20
NetBeans	Non-tokenize	85.50	18.64	13.56	67.80
	Tokenize	82.56	32.39	9.86	57.75

lines are common between the hunks. In such a situation *ldiff* reports them as deleted and added lines which contributed to its poor performance for the Eclipse benchmark.

2) How is LHDiff comparable to the content tracking technique of Git?: Some version control systems (like CVS or SVN) change code authorship of a line even if the line goes through formatting changes or moves to a different location. In both cases, the line is reported as a new one. Git overcomes such a limitation by tracking the content of a file across versions. Git *blame -C -M* command can track the movement of unchanged lines. Here, -C finds code copies and -M finds code movement. However, if the lines move even with slight changes, Git assigns code authorship to the new author and marks them as newly created, although ideally these lines come from different locations of the previous version. To compare the content tracking technique of Git with *LHDiff*, we first commit each pair (the original one and its new version) of file versions of a benchmark in a local repository and then apply the *git blame -M -n* to determine the origin of line locations of the new file. Here, *-n* tells Git to show the line numbers in the original commit which is by default turned off. The results are summarized in Table II. In general, the mapping accuracy is similar to diff, except it detects more correct mappings for some cases where lines are moved within files (such as encapsulation, function split and reordering categories).

Besides *blame*, Git also offers *pickaxe* to dig deeper into the history. While it can find the commits that change a block of code in a file, it cannot find the list of commits that contain the block of code. In general, Git focuses more on tracking code blocks instead of individual lines and stays out from the advanced diff or line tracking technique, possibly because of eliminating the overhead of running such an algorithm. Thus, the line level content tracking technique of Git is not as powerful as *ldiff* [3] or *LHDiff*.

3) Can tokenization improve the performance of LHDiff?: Source code can go through various cosmetic changes that can affect performance of text-based source location tracking techniques. For example, the order in which parameters appear in the method definition can be changed. In object-oriented programming languages, developers often use *this* keyword to access object fields or methods which does not change the behaviour of the program. The access modifier of a class



```
   . . .
int sum = 0;
int number= 0;
LinkedList<Integer> list =
new LinkedList<Integer>();
System.out.println("Hello")
   . . .
```

```
   . . .
num sum op val colon
num number op val colon
LinkedList list op new
LinkedList colon
out.println string
   . . .
```

Fig. 4: Example of tokenization

can change from private to public to make a class visible to all other classes. Handling these changes requires accessing individual source code elements and in this regard tokenization can assist us.

Tokenization is the process of converting a sequence of characters of a source file into a set of tokens, where each token is a string of characters representing a category of symbols. Tokenization does not require parsing source files and can be implemented using regular expressions. We anticipate that instead of working on raw source files, working on the tokens may help us to ignore the effect of source code changes and thus, improve the accuracy of the algorithm. To verify this, we first tokenize source files of our benchmarks according to the lexical rules of the Java programming language. During this process we keep track of the line locations of each token so that we can construct source files with tokens later. Next, we apply a set of transformation rules to normalize token sequences (an example is shown in Figure 4). This step is necessary to eliminate differences between source code versions. We then reconstruct source files using transformed token sequences and use *LHDiff* to track source code lines across file versions.

Working on the tokens does not improve the performance of the *LHDiff* algorithm. While for the Eclipse benchmark tokenization does not change mapping quality, performance drops by 3% for the NetBeans benchmark. When we examined those lines that were not correctly mapped by *LHDiff*, we found that working on the tokens eliminates differences between file versions. While for some cases it helps to correctly map lines that were earlier detected as deleted (the percentage of incorrect mapping for *eliminate* category drops from 67.8% to 57.75% for the NetBeans benchmark) but for some cases it leads to false mapping too (the percentage of spurious mapping increases from 18.64% to 32.39%). We observe similar picture for other benchmarks also. Though tokenization does not require parsing of source code, it requires knowledge of the tokens of programming languages. Since our objective is to develop a language-independent solution, *LHDiff* does not include tokenization as a part of the detection process. Moreover, our study results reveal that even tokenization can lead to poor result.

Although the above results show the performance of *LHDiff* over other methods, it does not explain how source code changes affect line tracking techniques. To find this out, we use a mutation based analysis that considers the editing taxonomy of source code changes as describe in Section V-A.

V. Evaluation using mutation based analysis

While previous studies evaluate source location tracking techniques against manually verified line mapping data, there

TABLE IV: Editing taxonomy of source code changes

No	Name	Example		Description
		Old File	**New File**	

1	Line Splitting/Merging	*else if (list.get(i)%3 == 0)*	*else*
			    if (list.get(i)%3 == 0)
				An *else if* statement splits into two lines

		if (list.get(i)%3 == 0) sum += list.get(i);	*if (list.get(i)%3 == 0){*
			    System.out.println("i = " + i);
			    sum += list.get(i)
			    }
				A statement in one line splits into multiple lines where each contains one statement and some lines are added in between them.

		list.add(Integer.parseInt(line));	*number = Integer.parseInt(line);*
			System.out.println(number);
			list.add(number);
				Another example of line splitting of the above kind.

2	Function Splitting/Merging	*public int sum(File file){*
		//Read numbers from the file
		//Store them in a list
		  ...
		//Now process the list
		//Calculate sum of the numbers
		  ...
		return sum;
		}
			public ArryaList readFile(Filefile){
			  //Read numbers from the file
			  //Store them in a list
			  ...
			  return numberList
			}
			public int sum(File file){
			  ArrayList list = readFile(file)
			  //Now process the list
			  //Calculate sum of the numbers
			  ...
			  return sum;
			}
				A developer creates a function *readFile* with some lines from *sum* and replace those lines in *sum* by adding a call to that function in the new version of *sum* or vice versa

3	Wrapping/ Unwrapping	*while((line = br.readLine())! = null)*	*try {*
			 while((line = br.readLine())! = null)
			}
			catch (IOException e){
			  e.printStackTrace()
				A line in the old file moves to a try-catch block in the new version of the file or vice versa

4	Change in Data Structure	*ArrayList list = new ArrayList();*	*LinkedList list = new LinkedList();*	*ArrayList* data structure is replaced by *LinkedList* or vice versa

5	Renaming	*sum = sum + list.get(i);*
		return sum;
			total = total + list.get(i);
			return total;
				Variable *sum* is renamed to *total* or vice versa

6	Code Reordering	*public ArrayList readFile (File file) {*
		    ...
		}
		public int sum (File file) {
		    ...
		}
			public int sum (File file) {
			    ...
			}
			public ArrayList readFile (File file) {
			    ...
			}
				Functions *sum* and *readFile* are reordered or vice versa

is not much discussion about the types of changes appearing in them. This opens the question of how stable/vulnerable the techniques are to different change groups and makes it difficult to compare them in an objective fashion. Instead of a random selection of lines from source code and tracking their positions in subsequent versions, we use a taxonomy of source code changes to synthesize new lines and evaluate the techniques objectively. Towards this goal, in this section we first present an editing taxonomy that captures typical actions performed by developers during source code evolution and then use a mutation based analysis to evaluate line tracking techniques based on the taxonomy following Roy and Cordy's [22], [21] mutation analysis framework for evaluating software clone detectors [23].

A. Editing taxonomy of source code changes

We developed an editing taxonomy by studying a large body of published work [10], [11], [16], [14], clone taxonomy [21], our previous study on code clones [24], and through manual investigation of the previous benchmarks [27], [20]. We further refined our taxonomy by analyzing changes of two open source Java systems (NetBeans [19] and iText [12]). We choose Java because of our familiarity with this programming language which helped us to determine correct mapping of lines with less confusion. Presenting the frequency of the changes in software systems is out of the scope of this paper.

Table IV shows our editing taxonomy of source code changes. For each change type we also provide an example that shows the change from the old version of a file to the new version of that file. The edit operations describe in the taxonomy are not mutually exclusive. Instead, they can be applied together to create more complex changes. To save space, simple edits (addition, deletion or modification of lines) are not discussed although they are the building blocks of all kinds of edits.

The first group of change in our taxonomy is line splitting/merging. Use of code formatters can trigger line splitting to improve readability of a source code. Table IV shows three different kinds of line splitting examples. We do not show any example of line merging since it is the opposite action of line splitting. Function merging/splitting is the second change type in our taxonomy. While Merging is done for service

TABLE V: Results of mutation based evaluation

Method	Change in Data Structure	Wrapping	Line Split	Function Split	Renaming	Reordering
W_BESTI_LINE	18.8	86.7	4.3	93.4	17.9	82.7
LHDiff	**56.3**	**90**	**68.1**	**93.4**	**35**	**83.8**
Unix diff	6.3	80	2.1	55.7	0	13.5
Git	6.3	80	2.1	85.3	0	66.2
ldiff [-i 0]	12.3	80	40.1	55.7	42.5	13.5
ldiff [-i 1]	6.3	90	44.7	86.9	35	52.7
ldiff [-i 3]	6.3	90	40.7	95.1	35	78.4
ldiff [-i 5]	6.3	90	40.7	95.1	35	82.4
SDiff	81.3	80	51.1	10	10	82.4
SDiff-probable	93.8	80	51.1	10	10	82.4
SDiff-possible-probable	93.8	80	51.1	15	15	82.4
SDiff-min	87.5	80	51.1	15	15	82.4
SDiff-min-probable	93.8	80	51.1	15	15	82.4
SDiff-min-possible-probable	93.8	80	51.1	15	15	82.4
SDiff-token	87.5	80	51.1	15	15	82.4
SDiff-token-probable	93.8	80	51.1	15	15	82.4
SDiff-token-possible-probable	93.8	80	51.1	15	15	82.4

consolidation or code clone elimination, it can be split also. The term Wrapping refers to a change where an old line is moved inside a block of code and we define the opposite action as unwrapping. Wrapping makes a line difficult to trace even when we try with contextual information since the context may differ and can be worse if the line itself changes significantly. In some cases, one form of wrapping can be changed into another. For example, the line attached with an *if* statement can be moved to the *else* part. We found scenarios where data structures used in previous versions are replaced with other kinds. For example, a HashTable can be replaced by a Map or an ArrayList implementation can be replaced by a LinkedList (see Table IV(4)). Changes of this kind may or may not preserve the textual similarity of lines and pose a threat to line mapping techniques. Changes in identifier names are common in source code evolution, particularly where copy-paste programming is involved. Location tracking techniques in general perform well when identifiers use descriptive names. Other language constructs such as function, method or class name can be changed. The last group of change type in our taxonomy is code reordering where functions or blocks of code can be reordered.

B. Experiment Details

In the generation phase, we mutate source files. First, we select a source file in version v_i. We then make another copy of that source file. The mutation is carried out either by injecting or changing existing code fragments in the copied file in such a way that considers several possible change scenarios listed in our taxonomy of source code changes. To avoid bias in comparison, we did not consider mapping of comments or curly braces in our analysis since *SDiff* ignores mapping them.

C. Results

The results of our mutation based analysis is summarized in Table V. In most of the change groups, *LHDiff* either outperforms other techniques or performs very close to the best technique with an exception in the data structure change category. In that case, accuracy drops to 56.3% whereas *SDiff* correctly detects around 93% mappings. When we investigate the reasons, it reveals that both context and content changes significantly which leads to such poor result.

W_BESTI_LINE suffers from the problem of detecting boundary of language constructs (such as a statement) and thus is immune to line splitting. *SDiff* operates at the statement level and can detect line splitting, but the accuracy is variable. For example, cases where lines are added in between of the split lines (see Table IV(1), second example), *SDiff* cannot determine correct mappings. In our mutation based analysis *LHDiff* performs not too bad in detecting line splitting. Manual investigation reveals that even in the case of line splitting, they do share some degree of similarity with the original line.

While *W_BESTI_LINE* and *ldiff* work at line level, they can track movement of lines because of function splitting/merging. *SDiff* is affected by function or method splitting and cannot detect a block of lines that is moved to other functions. Thus, *SDiff* reports them as deleted. For instance, *SDiff* cannot track lines that are moved from function *sum* to *readFile* (see Table IV(2)) and report them as deleted. The reason is that *SDiff* first tries to map functions and if it finds a mapping, it tries to map the lines. For this example, *SDiff* finds a mapping between sum_{old} and sum_{new} and then maps their lines without considering the fact that lines of the *sum* function can be moved to other functions also.

Renaming of variables, functions or classes also affect line tracking techniques and *ldiff*, *LHDiff* or *W_BESTI_LINE* may or may not map the line containing function declaration depending on the amount of changes occurred in the file. However, *SDiff* uses an origin analysis technique [14] to map functions across versions and thus can determine renaming of functions. In case of reordering, both *LHDiff* and *W_BESTI_LINE* show good performance. As expected *Unix diff* cannot detect any reordering since reordering of lines causes *Unix diff* to report them as addition and deletion of lines. While *SDiff* shows 82.4% accuracy in detecting changes via reordering, *Unix diff* become the last.

VI. Threats to Validity

There are a number of threats to the validity of this study. In this section we discuss them in brief.

First, the mapping of a line is subjective in nature and we cannot guarantee that there is no incorrect mapping in our benchmarks. However, we tried to minimize the number of false mapping as much as possible. The first two authors of this paper manually investigated all mappings including those found in the Reiss and Eclipse benchmarks. Cases where it was difficult to correctly map a line or there was a disagreement, the mapping was removed to avoid ambiguity. Second, considering the small sample size of the benchmarks one can argue that the sample may not represent the population and thus the performance observed in our study does not reflect the original scenario. However, finding changes of lines across versions and determining their correctness is a time consuming task. We carefully validated the correctness of existing benchmarks and developed a new benchmark containing different types of changes. We also used mutation based analysis to evaluate our technique with other approaches. Third, lines can be changed in various ways. In this paper we describe various kinds of changes that can affect the evolution of a line. Although we cannot guarantee that benchmarks contain all change types, we tried to minimize the effect of change types on their evaluation through collecting line change data at random. Finally, we penalize line tracking techniques with the same weight for detecting different false mapping types (spurious, change and eliminate). While one can argue about weighting false mapping types, we want to highlight to the fact that this is the same scoring scheme used in previous studies [20] and we followed the same strategy to make the result comparable with others.

VII. Conclusion

This paper proposes a novel, language-independent line tracking approach called *LHDiff*. Our experiment shows that lines can effectively be tracked across versions with *LHDiff*. We not only evaluate *LHDiff* with benchmarks created from real world applications but also use a mutation based analysis to evaluate it with other line tracking techniques against different types of source code changes. The results reveal that *LHDiff* is more effective than any other language-independent line tracking technique. We also compared *LHDiff* with *SDiff* which is a state-of-the-art language-dependent technique and found that in most cases *LHDiff* provides better result than *SDiff*. The mutation based analysis also enables us to explain the strength and weaknesses of line tracking techniques and can help a user to decide when to use which technique. Although *LHDiff* incorporates some features from *W_BESTI_LINE*, another simple line tracking technique developed by Reiss, but the potential of that technique is not fully explored in the early work, possibly because that work focused on comparing a large number of techniques and the benchmark used in that experiment contains small changes. *LHDiff* is reasonably fast (Comparable in speed to *SDiff* also), requires a small amount of memory, can easily be incorporated into source code control systems, and can be used with arbitrary text files. For future study, we plan to identify the degree of structural knowledge required to reduce incorrect mappings. The current implementation of *LHDiff* only uses information available within files and we would like to explore whether information stored within source code control systems can assists us mapping lines. While this work focuses on tracking lines, visualizing this information poses another challenge that we also want to address. The code of *LHDiff*, data files used in this experiment, and complete evaluation results can be found online [26].

Acknowledgements: We would like to thank Steven P. Reiss for giving us an implementation of *W_BESTI_LINE* and data files used in his experiment. Jaime Spacco provided useful comments on *SDiff* and supported our study by providing both code and data files.

References

[1] F. Al-Omari, I. Keivanloo, C. K. Roy, and J. Rilling, "Detecting Clones across Microsoft .NET Programming Languages", in Proc. WCRE, pp. 405-414, 2012.

[2] T. Apiwattanapong, A. Orso, and M. J. Harrold, "Jdiff: A differencing technique and tool for object-oriented programs", Automated Software Engg., vol. 14, no. 1, pp. 336, 2007.

[3] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu, "The promises and perils of mining git", in Proc. MSR, pp. 1-10, 2009.

[4] G. Canfora, C. Luigi, and M. Di Penta, "Tracking Your Changes: A Language-Independent Approach", in IEEE Softw., pp. 50-57, 2009.

[5] G. Canfora, L. Cerulo, M. Di Penta, "Identifying Changed Source Code Lines from Version Repositories", in Proc. MSR, pp.14, 2007.

[6] G. Canfora, L. Cerulo, and M. Di Penta, "Ldiff: An enhanced line differencing tool", in Proc. ICSE, pp. 595-598, 2009.

[7] M. S. Charikar, "Similarity estimation techniques from rounding algorithms", in Proc. STOC, pp. 380-388, 2002.

[8] E. Duala-Ekoko, M. P. Robbilard, "Tracking Code Clones in Evolving Software", in Proc. ICSE, pp. 158-167, 2007

[9] B. Fluri, M. Wuersch, M. Pinzger, and H. Gall, "Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction", in IEEE Trans. Softw. Eng., pp. 725-743, 2007.

[10] B. Fluri and H. Gall, "Classifying Change Types for Qualifying Change Couplings", in Proc. ICPC, pp. 35-45, 2007.

[11] M. W. Godfrey and L. Zou, "Using Origin Analysis to Detect Merging and Splitting of Source Code Entities", IEEE Transactions on Software Engineering, v.31 n.2, pp.166-181, 2005.

[12] "The iText", http://www.sourceforge.net/projects/itext

[13] D. Kawrykow and M. P. Robillard, "Non-essential changes in version histories", in Proc. ICSE, pp. 351-360, 2011.

[14] S. Kim , K. Pan , E. J. Jr. Whitehead, "When Functions Change Their Names: Automatic Detection of Origin Relationships", in Proc. WCRE, pp. 143-152, 2005.

[15] M. Kim, V. Sazawal, D. Notkin, G. Murphy, "An empirical study of code clone genealogies", in Proc. ESEC/FSE, pp. 187-196, 2005.

[16] M. Kim and D. Notkin, "Program element matching for multi-version program analyses", in Proc. MSR, pp.58-64, 2006.

[17] H. Kuhn, "The Hungarian Method for the assignment problem", Naval Research Logistics Quaterly, pp. 2:83-97, 1955.

[18] G. S. Manku, A. Jain, and A. D. Sarma, "Detecting Near Duplicates for Web Crawling", in Proc. WWW, pp. 141-150, 2007.

[19] "The NetBeans", http://www.netbeans.org

[20] S. P. Reiss, "Tracking source locations", in Proc. ICSE, pp. 11-20, 2008.

[21] C. Roy and J. R. Cordy, "A Mutation/Injection-Based Automatic Framework for Evaluating Code Clone Detection Tools", in Proc. ICSTW, pp. 157-166, 2009.

[22] C. K. Roy and J. R. Cordy, "Towards a Mutation-Based Automatic Framework for Evaluating Code Clone Detection Tools", in Proc. C3S2E, pp. 137-140, 2008.

[23] C. K. Roy, "Detection and Analysis of Near-Miss Software Clones", in Proc. ICSM, pp. 447-450, 2009.

[24] R. K. Saha, M. Asaduzzaman, M. F. Zibran, C. K. Roy, and K. A. Schneider, "Evaluating code clone genealogies at release level: An empirical study", in Proc. SCAM, pp. 87-96, 2010.

[25] "The Simhash Algorithm", http://d3s.mff.cuni.cz/~holub/sw/shash/

[26] "Source code and data", http://asaduzzamanparvez.wordpress.com/Research

[27] J. Spacco and C. Williams, "Lightweight Techniques for Tracking Unique Program Statements", in Proc. SCAM, pp. 99-108, 2009.

[28] M. Storey, L. Cheng, I. Bull, and P. Rigby, "Shared waypoints and social tagging to support collaboration in software development", in Proc. CSCW, pp. 195-198, 2009.

[29] M. S. Uddin, C. K. Roy, K. A. Schneider, and A. Hindle, "On the Effectiveness of Simhash for Detecting Near-Miss Clones in Large Scale Software Systems", in Proc. WCRE, pp. 13-22, 2011.

[30] M. S. Uddin, C. K. Roy, and K. A. Schneider, "SimCad : An Extensible and Faster Clone Detection Tool for Large Scale Software Systems", in Proc. ICPC, pp. 236-238, 2013.

[31] Z. Xing and E. Stroulia, "UMLDiff: an algorithm for object-oriented design differencing", in Proc. ASE, pp. 54-65, 2005.

[32] T. Zimmerman, S. Kim, A. Zeller, and E. J. Jr. Whitehead, "Mining Version Archives for co-changed lines", in Proc. MSR, pp. 72-75, 2006.