Ldiff: an Enhanced Line Differencing Tool

Gerardo Canfora, Luigi Cerulo, Massimiliano Di Penta
RCOST – Dept. of Engineering, University of Sannio
Via Traiano, 82100 Benevento, Italy
canfora@unisannio.it, lcerulo@unisannio.it, dipenta@unisannio,it

Abstract

Differencing tools are highly relevant for a series of software engineering tasks, including analyzing developers' activities, assessing the changeability of software artifacts, and monitoring the maintenance of critical assets such as source clones and vulnerable instructions.

This tool demonstration shows the features of ldiff, an enhanced, language-independent line differencing tool. Ldiff builds upon the Unix diff and overcomes its limitations in determining whether an artifact line has been changed or is the result of additions and removals, and in tracking artifact fragments that have been moved upward or downward within the file. The paper describes the tool and shows its capability of analyzing changes on different kinds of software artifacts, including use cases, code developed with different programming languages, and test cases.

Keywords: differencing algorithm, software evolution, mining software repositories

1 Introduction

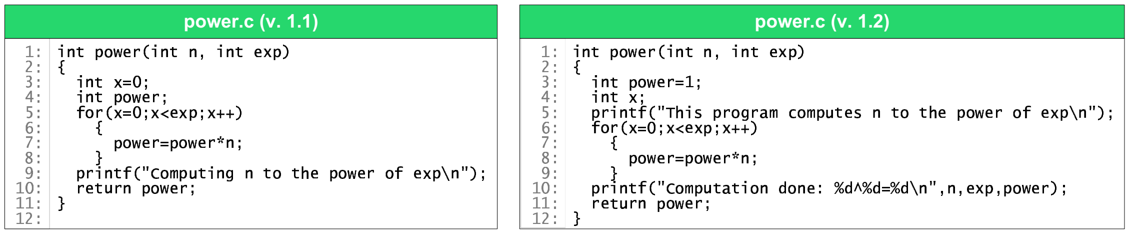
Tasks such as keeping track of developers' activities, analyzing changes occurring on software artifacts across releases, monitoring changes occurring on critical assets such as code clones and vulnerable instructions are crucial to support software development and maintenance activities. The availability of techniques [3, 8, 10] to integrate data from different kinds of software repositories, ranging from bug tracking systems to versioning systems, certainly represents an important baseline for the development of tools able to support developers with the above tracking and monitoring activities.

Most analyses of software repositories strongly rely on the performance of differencing tools that identify changes occurred between an artifact revision and the subsequent one. The most widely adopted differencing tool—on which also versioning systems rely—is the Unix *diff*, which compares two textual files and determines the minimum number of line additions and removal that produces the second file from the first one. From a developer's point of view, *diff* has two serious limitations that reduce the kinds of analyses that can be performed: (i) it makes difficult to determine whether a source code fragment was just changed in some points, or instead it was completely replaced; (ii) in case a code fragment is moved upward or downward in a file, it is not always possible to keep track of it.

Indeed, there exist enhanced differencing tools—for example the Change Distiller by Fluri *et al.* [4]—that are able to precisely identify the kind of change performed, e.g., the addition of a method, a parameter removal, etc. Although being very precise, this kind of tools need to build the Abstract Syntax Tree (AST) of the source code to be differenced, thus rely on the availability of a parser for a specific language. Reiss [7] empirically found that source code can be tracked through multiple versions of a file by using relatively simple techniques, such as line matching based on the Levensthein distance.

This paper describes the demonstration of the *ldiff* (line differencing) tool, which overcomes the limitations of *diff* in the identification of changed textual fragments, and is also able to identify code fragment moving. The tool builds upon the analysis produced by *diff*, and adds iterations of similarity computations at fragment level—by means of text similarity measures—and at line level—by means of line distance measures. According to the taxonomy proposed by Kim and Notkin [5]—which classified code differencing algorithms into algorithms working on a structured representation of the program (e.g., AST), and algorithms working on a flat representation (e.g., sequence of lines)—*ldiff* falls in the second category, thus it is language independent. This means that not only it can work on source code developed with different programming languages, but it can also be used to analyze differences in other artifacts such as use cases, test cases, or even design documents (although for the last purpose there are more appropriate and specific tools, e.g., UMLDiff [9]).

The differencing algorithm and its evaluation have been thoroughly described in separate papers [1, 2]. This paper



```
 1: int power(int n, int exp)
 2: {
 3:   int x=0;
 4:   int power;
 5:   for(x=0;x<exp;x++)
 6:     {
 7:       power=power*n;
 8:     }
 9:   printf("Computing n to the power of exp\n");
10:   return power;
11: }
12:
```

```
 1: int power(int n, int exp)
 2: {
 3:   int power=1;
 4:   int x;
 5:   printf("This program computes n to the power of exp\n");
 6:   for(x=0;x<exp;x++)
 7:     {
 8:       power=power*n;
 9:     }
10:   printf("Computation done: %d^%d=%d\n",n,exp,power);
11:   return power;
12: }
```

Figure 1. Two versions of a simple C function.

briefly summarizes the *ldiff* algorithm, describes the tool syntax and shows several application examples, comparing *ldiff* results with those of the Unix *diff*.

The remainder of the paper is organized as follows. Section 2 highlights the limitations of the Unix *diff* by means of a motivating example. Section 3 briefly recalls the differencing approach; then, it summarizes the approach performances and describes the tool options. Section 4 shows, with the example of Section 2, how *ldiff* overcomes the limitations of the Unix *ldiff*. Section 5 concludes the paper.

2 Motivating Example

Let us consider the two versions of a C function—shown in Figure 1—that computes n to the power of *exp*, with $exp > 0$. If we execute the Unix *diff* on these two file versions, this is what we obtain:

```
diff power-v1.c power-v2.c

3,4c3,5
<   int x=0;
<   int power;
---
>   int power=1;
>   int x;
>   printf("This program computes n to the power
of exp\n");
9c10
<   printf("Computing n to the power of exp\n");
---
>   printf("Computation done: %d^%d=%d\n",n,exp,power);
```

Diff indicates that lines 3–4 on the left side have been changed into lines 3–5 on the right side, and that line 9 has been changed to line 10. However, it appears likely that the programmer swapped lines 3,4, modified them, moved line 9 to line 5, and then added line 10. This shows the limited ability of the Unix *diff* in tracking lines moved away from their original position, and in distinguishing changes from additions and removals.

3 Approach and Tool

Our differencing algorithm comprises three steps, visually represented in the example of Figure 2[1].

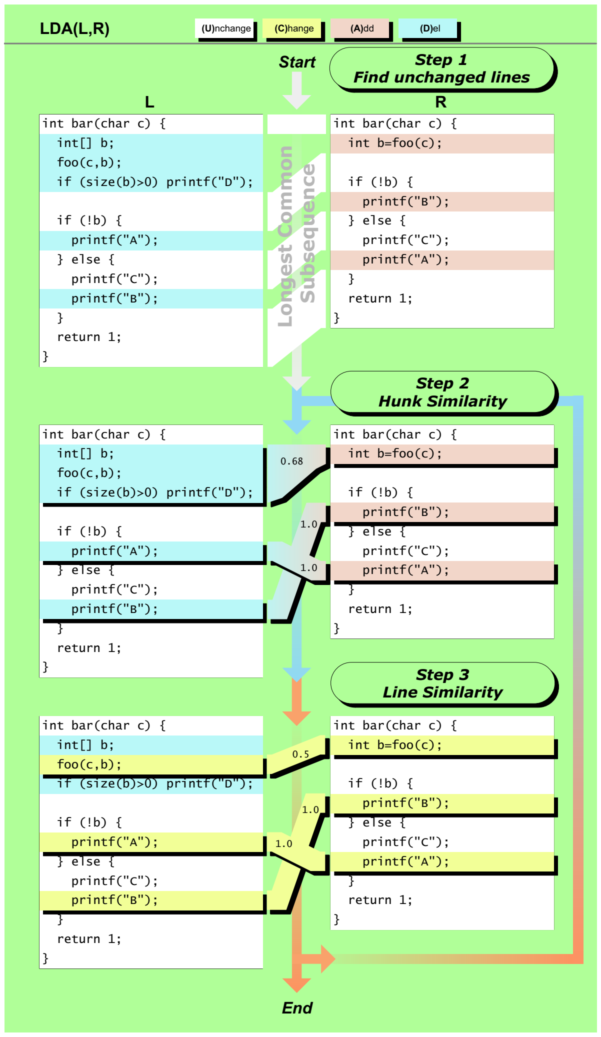
- **Step 1** aims at determining the set of unchanged lines. For this step, *ldiff* relies on the Unix *diff*, which adopts a longest common subsequence algorithm.
- **Step 2** compares all combinations of f_1 and f_2 fragments—not classified as unchanged at the previous step using a textual similarity measure, e.g., the Vector Space Model cosine similarity.
- **Step 3** considers the most similar distinct *HT* hunk pairs, and for each pair performs a line-by line comparison using a line differencing algorithm (e.g., the Levenhstein edit distance [6]). The pairs having a similarity higher than a given threshold *LT* are traced together and classified as changed. The remaining ones will be considered in the subsequent iterations of Steps 2 and 3. In particular, the user can calibrate the tool to perform i iterations, such that the recall of changed fragments is increased.

3.1 Performances

This section briefly summarizes the performances of the differencing approach. Details on this assessment can be found in separate papers [1, 2].

1. *the ability of ldiff to identify moved line hunks* has been assessed by randomly generating new releases of 100 source code files from PostgreSQL[2] and openSSH[3], where source code fragments composed of 1 to 10 source code lines were randomly moved. We found that *ldiff* had a precision of about 92% in detecting code movements, and a recall between 62% and 73%, increasing with the number of iterations i;

[1]The figure is taken from our paper [2]; *f1* and *f2* are the files to be compared.
[2]http://www.postgresql.org/
[3]http://www.openssh.com/



```
int bar(char c) {
  int[] b;
  foo(c,b);
  if (size(b)>0) printf("D");

  if (!b) {
    printf("A");
  } else {
    printf("C");
    printf("B");
  }
  return 1;
}
```

```
int bar(char c) {
  int b=foo(c);

  if (!b) {
    printf("B");
  } else {
    printf("C");
    printf("A");
  }
  return 1;
}
```

```
int bar(char c) {
  int[] b;
  foo(c,b);
  if (size(b)>0) printf("D");

  if (!b) {
    printf("A");
  } else {
    printf("C");
    printf("B");
  }
  return 1;
}
```

```
int bar(char c) {
  int b=foo(c);

  if (!b) {
    printf("B");
  } else {
    printf("C");
    printf("A");
  }
  return 1;
}
```

```
int bar(char c) {
  int[] b;
  foo(c,b);
  if (size(b)>0) printf("D");

  if (!b) {
    printf("A");
  } else {
    printf("C");
    printf("B");
  }
  return 1;
}
```

```
int bar(char c) {
  int b=foo(c);

  if (!b) {
    printf("B");
  } else {
    printf("C");
    printf("A");
  }
  return 1;
}
```

Figure 2. Differencing algorithm steps (from [2]).

2. *the precision of ldiff, compared with the Unix diff, in identifying changed, added, deleted, and unchanged lines* has been assessed on 11 change sets[4] from the ArgoUML[5] CVS repository, each one composed of a number of files ranging from 11 and 72 and a number of lines between 32 and 401. Results indicated that:

    - there is no difference in the identification of unchanged lines—precision=99% in both cases—since *ldiff* relies on the Unix *diff* for this purpose (see Step 1 of the approach);

[4]The sequence of file revisions that share the same author, branch, and commit notes, and such that the difference between the timestamps of two subsequent commits is less or equal than 200 seconds [10].

[5]http://argouml.tigris.org/

Table 1. Ldiff command line options

`-i n`	number of iterations
`-HT CUT:N`	hunk similarity threshold specified with a cut level ($0 \leq$ `CUT` ≤ 1) and the number of s to consider (`N` ≥ 1)
`-LT THR`	line similarity threshold ($0 \leq$ `THR` ≤ 1)
`-lm metric`	line distance (e.g., Levenshtein)
`-hm metric`	hunk similarity metric (e.g., cosine, Jaccard, and overlap)
`-lt type`	line tokenizer (e.g., char, word, ngram, cpp)
`-ht type`	hunk tokenizer (e.g., char, word, ngram, cpp)

    - *ldiff* outperforms the Unix *diff* in the identification of changed lines (precision=87% vs. 19%);
    - instead, *diff* performs better in the identification of added (precision=99% vs 77%) and deleted lines (precision=99% vs. 68%);

3. *time needed to perform the analyses: ldiff* has a quadratic complexity, and on a 2 GHz Intel Centrino^{TM} laptop with 1 GB of RAM, it takes (for i=1) 2 seconds to classify 34 line pairs and 54 seconds to classify 171 line pairs;

4. *cases in which ldiff fails: ldiff* detects changes between distinct line pairs. It fails when a line is split into more lines or, conversely, groups of lines are merged into one line. In such cases, the change is detected for only one pair of lines, while it is missed for the remaining ones, where the normalized Levenshtein Distance is lower than the *LT* threshold.

3.2 Tool Features

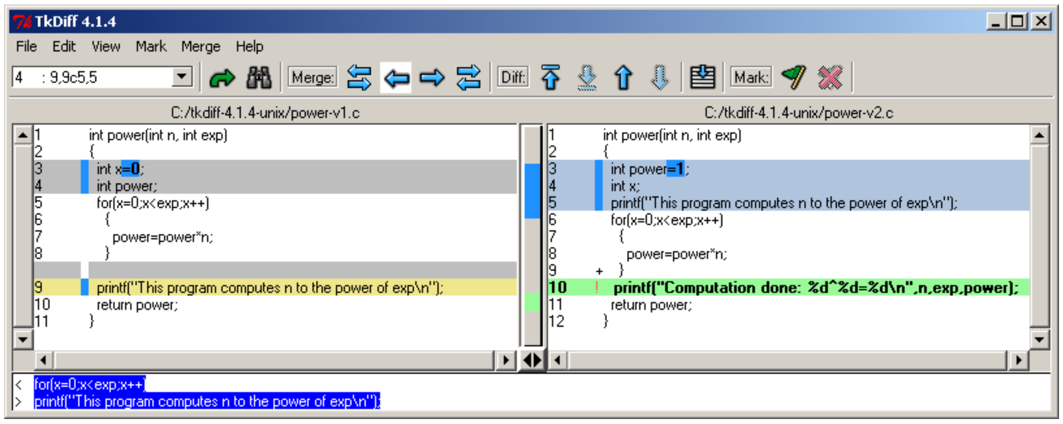
A Perl implementation of *ldiff* is available for downloading at the URL: *http://rcost.unisannio.it/cerulo/tools.html*. *Ldiff* supports a variety of hunk similarity metrics (Cosine, Jaccard, Dice, and Overlap), and different text item extraction techniques (chars, words, n–grams, of C/C++ language tokens). The tool parameters can be configured as described in Table 3.2.

4 Motivating Example Revisited

Let us analyze again the example of Figure 1, this time using *ldiff* instead of the Unix *diff*. This is what we obtain:

```
./ldiff.pl -i=3 power-v1.c power-v2.c

3,3c4,4
<   int x=0;
---
>   int x;
4,4c3,3
<   int power;
---
>   int power=1;
8a10,10
>   printf("Computation done: %d^%d=%d\n",n,exp,power);
9,9c5,5
<   printf("Computing n to the power of exp\n");
```



```
C:/tkdiff-4.1.4-unix/power-v1.c
1   int power(int n, int exp)
2   {
3     int x=0;
4     int power;
5     for(x=0;x<exp;x++)
6       {
7         power=power*n;
8       }

9     printf("This program computes n to the power of exp\n");
10    return power;
11  }
```

```
C:/tkdiff-4.1.4-unix/power-v2.c
1   int power(int n, int exp)
2   {
3     int power=1;
4     int x;
5     printf("This program computes n to the power of exp\n");
6     for(x=0;x<exp;x++)
7       {
8         power=power*n;
9  +    }
10 !  printf("Computation done: %d^%d=%d\n",n,exp,power);
11    return power;
12  }
```

```
< for(x=0;x<exp;x++)
> printf("This program computes n to the power of exp\n");
```

Figure 3. Visualizing ldiff results with TKDiff.

```
---
>   printf("This program computes n to the power
of exp\n");
```

Ldiff indicates that line 3 was changed and moved to line 4, and vice versa line 4 was changed and moved to line 3. Also, it detects the movement of line 9 to line 5, and the addition of line 10.

Besides the usage from command line, *ldiff* can be used with any *diff* front end that allows to modify the *diff* command line and its parameters. For example, as shown in Figure 3, we can use the TKDiff front end[6]. Changed lines appear highlighted in grey in the first version (left side), and in cyan in the second version (right side), while added lines appear highlighted in green. The top-left combo-box shows that the selected line (highlighted in yellow) has been moved from line 9 to line 5.

5 Conclusions

This paper described *ldiff*, a line differencing tool that overcomes the limitations of the Unix *diff* in distinguishing likely changed lines from added and removed lines, and is capable of tracking line moving. The demonstration will show—comparing results obtained with the Unix *diff* and with *ldiff*—how *ldiff* is able to analyze any kind of source file or software artifact, and how it can be used to monitor vulnerable instructions.

References

[1] G. Canfora, L. Cerulo, and M. Di Penta. Identifying changed source code lines from version repositories. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 14. IEEE CS, 2007.

[2] G. Canfora, L. Cerulo, and M. Di Penta. Tracking your changes: a language-independent approach. *IEEE Software*, 27(1):50–57, 2009.

[3] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Proceedings of the International Conference on Software Maintenance*, pages 23–32, Amsterdam Netherlands, September 2003.

[4] B. Fluri, M. Würsch, M. Pinzger, and H. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Software Eng.*, 33(11):725–743, 2007.

[5] M. Kim and D. Notkin. Program element matching for multi-version program analyses. In *Proceedings of the 2006 International Workshop on Mining Software Repositories, MSR 2006, Shanghai, China, May 22-23, 2006*, pages 58–64, 2006.

[6] V. I. Levenshtein. Binary codes capable of correcting deletions,insertions, and reversals. *Cybernetics and Control Theory*, (10):707–710, 1966.

[7] S. P. Reiss. Tracking source locations. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, pages 11–20, 2008.

[8] J. Sliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *Proceedings of the 2005 International Workshop on Mining Software Repositories MSR 2005 Saint Louis Missouri USA*, May 17 2005.

[9] Z. Xing and E. Stroulia. Differencing logical UML models. *Autom. Softw. Eng.*, 14(2):215–259, 2007.

[10] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 563–572. IEEE CS, 2004.

[6] *http://tkdiff.sourceforge.net/*