Inferring Repository File Structure Modifications Using Nearest-Neighbor Clone Detection

Thierry Lavoie[1], Foutse Khomh[2], Ettore Merlo[1], Ying Zou[2]
[1] Departement de genie informatique et logiciel, Ecole Polytechnique de Montreal, Montreal, QC, Canada,
[2]Department of Electrical and Computer Engineering, Queen's University, Kingston, ON, Canada
thierry-m.lavoie@polymtl.ca, foutse.khomh@queensu.ca, ettore.merlo@polymtl.ca, ying.zou@queensu.ca

***Abstract*—During the re-engineering of legacy software systems, a good knowledge of the history of past modifications on the system is important to recover the design of the system and transfer its functionalities. In the absence of a reliable revision history, development teams often rely on system experts to identify hidden history and recover software design. In this paper, we propose a new technique to infer the history of repository file modifications of a software system using only past released versions of the system. The proposed technique relies on nearest-neighbor clone detection using the Manhattan distance. We performed an empirical evaluation of the technique using Tomcat, JHotDraw and Adempiere SVN information as our oracle of file operations, and obtained an average precision of 97% and an average recall of 98%. Our evaluation also highlighted the phenomena of implicit *Moves*, which are, *Moves* between a system's versions, that are not recorded in the SVN repository. In the absence of revision history and software experts, development teams can make use of the proposed technique during the re-engineering of their legacy systems.**

***Keywords*-Software repository, Software similarity, Software clones, Software evolution, Legacy systems, Nearest-Neighbor**

I. Introduction

Software companies depend on Version Control Systems (VCS) to track and manage modifications on their software systems. The information recorded by VCS is key to the success of many software development activities. For example, during a software reengineering, development teams rely on the revision history of the software system to recover its design and transfer its functionalities. However, for many legacy systems, VCS are not available and developers are left to rely solely on their knowledge of the system to uncover the hidden history of modifications. Developers knowledge of a system is often incomplete [24]. Therefore, relying solely on this knowledge is likely to be ineffective. Nevertheless, since source code based evolution analysis can be performed also at release level (*i.e.*, using the source code of the versions released to customers), in the absence of a reliable VCS, it is possible to infer information about files structure and the history of their modifications. Inspired by clone detection techniques, we present an original technique to recover file *moves* information between released versions of a system. The new technique relies on a nearest-neighbor clone detection approach using the Manhattan distance on frequency vectors of code fragments. To assess the effectiveness of the proposed technique, we perform a case study using the open source software systems Tomcat, JHotDraw and Adempiere. We answer the following research question:

RQ1: Can the proposed technique accurately recover file moves that occur between released versions of a software system ?

Using the proposed technique, we are able to identify *moves* of files across the releases of a software system with an average precision of 97% and an average recall of 98%. The proposed technique also identifies *implicit* files movements that are not recorded by VCS. These *implicit* files movements are generally the results of file clonings. In the absence of a reliable VCS, development teams can make use of the proposed technique to recover important knowledge about the modifications of their software systems. The proposed technique can also be used to enrich the meta-data of existing VCS with information about *implicit moves* that are not recorded currently.

The rest of the paper is divided as follows: Section II summarizes the related literature concerning clone detection and code fragments evolution analysis, Section III details the proposed algorithm, Section IV presents our experimental setup and our methodology, Section V reports the results of our experiments, Section VI discusses the obtained results and potential threats to their validity. Finally, Section VII summarizes our work and outlines avenues for future work.

II. Literature

This section summarizes the related literature on clone detection techniques, provenance, and clone evolution analysis.

A. Clone Detection Techniques

The state of the art on clone detection includes many different techniques. For identical and parametric clones, they range from AST-based detection techniques [7] to metrics-based [21], suffix tree-based [13], and string matching [11] detection techniques. Roy and Cordy [27], Göde and Koschke [13], and Lavoie and Merlo [19] have also proposed detection techniques for near-miss clones. A detailed survey of clone detection techniques is presented in [26]. In this paper, we improve on our own clone detection technique [20], and extend it to track similarities between all code fragments (including non cloned code) using the same approximation of the Levenshtein distance. We do this because we believe that clones are only a special case of similarity in software systems.

B. Provenance and Clone Evolution Analysis

The work presented in this paper is also related to provenance analysis since we track the provenance of code fragments across the versions of a software system. Searching for the provenance of code fragments in a preceding version is comparable to tracking a clone throughout the evolution of a software system. The problem of code provenance analysis is a special case of clone evolution analysis [18].

Code provenance analysis are becoming more and more popular in the software clone community. Recently, Godfrey et al. [14] published a position paper in which they highlight some key issues related to the problem of code provenance. They recommend the development of simple and lightweight techniques capable of reducing the search space of the potential origins of candidates pieces of code. Other relevant literature on provenance analysis include [9], [12]. Kim et al. [17] proposed the first work on software clone evolution. They analyzed clone classes and defined patterns of clone evolution. Using two java systems and the CCFINDER clone detection tool, they performed a case study of the evolution of clones in software systems and concluded that at least half the of clones in a software system are eliminated within eight check-ins after their creation. A systematic review of clone evolution studies is presented in [25]. The work presented in this paper differs from previous work in the way that we track the provenance of all code fragments in a version instead of searching for the provenance of clones only. The next section elaborates more on the details of our technique.

III. PROBLEM, ALGORITHM, AND METHOD

A. Definition of the problem

The goal to achieve is to infer the underlying file structure modifications between two versions of a system. There are three file structure modification primitives:

- *Add*: A new file is added in version N+1
- *Delete*: A file from version N is deleted in version N+1
- *Move*: A file from version N has been moved in version N+1

Conceptually, a *move* can be represented as a succession of a *Delete* of the original file and an *Add* of a file which is a copy of the original file with a different name. Indeed, Version Control Systems represent a *move* using these two primitives. In these systems, raw *Add* and *Delete* are either part of a *Move* or strict *Add* or *Delete*. To identify *Add* and *Delete* operations properly, it is necessary to first identify the *move* operations. Thereafter, the remaining *Add* and *Delete* are the true *Add* and *Delete* operations. Thus, the important task in inferring repository file structure modifications is the detection of the *move* operations.

Assuming the above, the rest of this section discusses only the algorithm to track *move* operations. It follows naturally that the detection accuracy of the *Add* and *Delete* is directly dependent on the accuracy of the *move* identification step. Consequently, the evaluation will also only focus on the *moves*.

As already mentioned in section II, the technique is an extension over [20]. The key element in this extension is the use of a nearest-neighbor search instead of range-query search. Both operations are faster if executed in a metric tree rather than using a brute-force approach. Considering this fact, before introducing our algorithm called Move Inferring Algorithm (MIA), let us review some basic concepts about the metric tree data structure.

B. Metric tree definition and construction

Since the clone detection operation relies on finding a set of fragments with a certain distance property, it follows from [19], [20] that it is worth using a specialized data structure to optimize the search space. The metric tree, originally presented in [28], and then notably used in [8], [10], is well suited for this task. Other structures, such as the kd-tree and the cover-tree might be worth exploring, but since the metric tree supports arbitrary metrics, it is best suited for clone detection as precised in [19], [20].

Using a metric tree restricts the similarity measures allowed to the set of distances satisfying the metric axioms. The following is a brief reminder of those axioms:

$$\delta(x, y) \geq 0 \ (non\ negativity) \tag{1}$$

$$\delta(x, y) = \delta(y, x) \ (symmetry) \tag{2}$$

$$\delta(x, y) = 0 \Leftrightarrow x = y \tag{3}$$

$$\delta(x, y) + \delta(y, z) \geq \delta(x, z) \ (triangle\ inequality) \tag{4}$$

Many common distances like the Jaccard distance, the Levenshtein distance, the Euclidean distance and the l_i norm family satisfy these axioms. Colloquially, they are called distances even though they are metrics. Also, in this paper distance will always be used in the same sense as metric.

Nodes in the metric tree contain one or two elements. For similarity analysis, these elements may be code fragments such as files, classes, methods or any mapping of these fragments to other representation such as frequency vectors. The variables f and f' in the following figures always represent a fragment or a representation of a fragment, according to the context. A node containing only one element is a leaf. A node containing two elements may be a leaf or not. Nodes with two elements, called pivots, split the search space into four regions according to the distance d between the two pivots. These four regions themselves are individually nodes that contain up to two pivots and recursively divide the space. Edges in the tree link nodes to their four children regions.

The metric tree supports three important primitives: $insert(f)$ and $rangeQuery(f, \epsilon)$ and $findnn(f)$. The $insert(f)$ primitive takes a fragment and inserts it in the tree. The primitive $rangeQuery(f, \epsilon)$ takes a fragment f and a real number ϵ as parameters and returns the set of all fragments f' in the tree for which $\delta(f, f') \leq \epsilon$, as follows:

$$rangeQuery(f, radius) = \{f^{'} \mid \delta(f, f^{'}) \leq \epsilon\} \tag{5}$$

The $findnn$ primitive searches for the nearest-neighbor of a fragment f in the tree, i.e.:

$$findnn(f) = f' |\ \delta(f, f') = min(\delta(f, f_i)) \forall f_i \in F \quad (6)$$

An outline of the $insert$ primitive is presented in Figure 1.

```
1: insert (f)
2: target = n_0
3: region = 0
4: while target.d ≠ UNDEFINED do
5:    d_1 = δ(target.x, f)
6:    d_2 = δ(target.y, f)
7:    if d_1 < target.d then
8:       if d_2 < target.d then
9:          region = 0
10:      else
11:         region = 1
12:      end if
13:   else
14:      if d_2 < target.d then
15:         region = 2
16:      else
17:         region = 3
18:      end if
19:   end if
20:   target = target.c[region]
21: end while
22: if target.x ≠ UNDEFINED then
23:   target.x = f
24: else
25:   target.y = f
26:   target.d = δ(target.x, target.y)
27: end if
28: return
```

Fig. 1: Insertion algorithm in metric trees

In the insertion algorithm, $node$ contains fragment x and y and the distance d between x and y. Line 2 of the algorithm starts by initializing a variable $target$ with the node n_0 assumed to be the root of the tree. The variable $target$ represents the node in which we will insert the new fragment. The main loop in the algorithm will assign different nodes to $target$ as the algorithm progresses. The first two steps of the loop at lines 5 and 6 compute the distance of f with the two fragments already assigned to the node, called the pivots. From lines 7 to 21, the distance of f to the two pivots is compared to the distance between the two pivots. A region is selected according to criteria based on those distances. The loop continues to search nodes until it finds a node for which at least one pivot is undefined. Lines 22 to 27 then check which of the pivot is undefined and setup the node accordingly with the new fragment.

C. Building the token frequency vectors

To compute fast matching, it is required to find a proper representation of code fragments to insert in the metric tree. We chose to represent each fragment with a frequency vector of its tokens n-gram. The proposed algorithm combines different known ideas for clone detection along with new ones. Token-based clone detection was proposed by [16]. Other authors [6], [27] have since used tokens with different algorithms and token manipulation is now widely used by many tools at different step of the process of clone detection. Code metric based clone detection was introduced by [21] and developed in [22], [23]. The idea of code metric clone detection is to choose software syntactic features, such as statements, branching instructions, function calls, etc., and to build a vector for which each dimension is a specific feature. The value of each vector component is the frequency of the corresponding feature. Syntactic analysis is first done to compute the frequencies to then build the vectors. These are then compared using a similarity criterion, such as the Euclidean distance, cosine distance, etc. The original technique of [21] used space discretization for clone clustering. The new algorithm presented in this paper combines token analysis and code metric to create a vector: it builds vectors of token frequencies. It is not limited to the use of single tokens, but can also be extended to n-grams which we call windowed-tokens. With the new vectors, similarity is computed according to the Manhattan distance, also known as the l_1 metric.

To build these vectors, the first step of the algorithm is to extract the tokens from the software source code. This is done with a lexical analyzer on a file basis. Using lexical analysis instead of syntactic analysis has some advantages. It is faster, first, since most of the times it relies only on regular expression matching instead of context-free grammar matching, and second, it is also usually easier to write a lexer than a parser.

The second step uses the extracted tokens from the files to build frequency vectors. The base case is to use single tokens or windowed-tokens of length 1. A unique *ID* number corresponding to its corresponding dimension in $\mathbb{R}^n$ is assigned to every different token type. The tokens *ID* are generated dynamically. Each time a new token type is encountered, the next available *ID*, starting with *ID* 0, is assigned to the newly discovered type. After the token type has been identified, the corresponding vector component is incremented by one. Every component in the vector has an initial value of 0. For better memory storage, the frequency vectors are not stored in a vector-array data structure but rather in a hash table. Since code fragments have a frequency of 0 for most token types, the hash table will reduce memory usage. This may not be trivially apparent, but if we allow windowed-tokens of length 2 or more, storing data in vector-arrays is not an option because of storage capacity. For example, in a language with 200 token types, the required array length to store every components explicitly is 200 for window length 1, 19 900 for length 2, and 1 313 400 for length 3. In general, the vector length is described as:

$$\frac{t!}{(t-l)!} \quad (7)$$

where t is the number of token types in the language and l the window length. This equation is of course growing exponentially with respect to the length of the window and thus compels for a better memory storage. Since to any fragment there cannot be more token types associated than its number of tokens, the hash table will use a storage linear in the number of tokens.

To extend the base case to a window length above 1, the same procedure is used but token type identifiers are assigned on an n-gram basis. For example, let the tokens of a language be $\{A, C, G, T\}$, and let s_0 be:

$$s_0 = ATGCGTCGGGTCCCAG \tag{8}$$

a random string. With a window of length 1, *ID* assignment would be:

$$(A, 0)\ (T, 1)\ (G, 2)\ (C, 3) \tag{9}$$

and s_0 frequency vector v_{s_0}:

$$v_{s_0} = (2, 3, 6, 5) \tag{10}$$

With a length 2 window, *ID* assignment would be:

$$(AT, 0)\ (TG, 1)\ (GC, 2)\ (CG, 3)\ (GT, 4) \tag{11}$$
$$(TC, 5)\ (GG, 6)\ (CC, 7)\ (CA, 8)\ (AG, 9) \tag{12}$$

and s_0 frequency vector v_{s_0}:

$$v_{s_0} = (1, 1, 1, 2, 2, 2, 2, 2, 1, 1) \tag{13}$$

and so forth with higher window lengths. The reader should note that the total number of token types in the second example, 10, is less than the theoretic maximum, 12. This is almost always the case with higher window lengths and thus it supports the idea that a hash table will consume much less memory than a vector-array.

The third step, after extracting the tokens from software and building their corresponding frequency vectors, is to build a metric tree [10], [19] with all the resulting vectors under a chosen distance. The l_i metric family is a natural choice. The goal of the algorithm is to be as precise as possible and to have a fine grain adjustable threshold on the similarity criterion. The l_1 metric, or Manhattan distance, is the best choice in the l_i family according to our criterion. The motivation behind this choice is a simple geometric observation. To get the finest grain threshold, the space enclosed by the distance at a specific threshold should be as small as possible. That space is of course a sphere defined by the chosen l_i metric. Using a quick proof, it will be shown that at each step, the l_1-sphere is always smaller than the l_2-sphere. An analogous reasoning can then be applied to show that an l_i-sphere is always smaller than an l_j-sphere if $i < j$ for any integer value, making the l_1-sphere the smallest. Lets define an l_i-sphere $\mathbf{S}_{i,\delta}$ as the set $x \in \mathbb{R}^n | l_i(x - y) \leq \delta$ for a fixed n. In $\mathbb{R}^2$, the l_1-sphere $\mathbf{S}_{1,\delta}$ is thus a $\frac{\pi}{4}$ radians rotated square of side-length $\sqrt{2\delta}$ and the $l_2 - sphere$ $\mathbf{S}_{2,\delta}$ is a circle of radius δ. In $\mathbb{R}^n$, every point x in $\mathbf{S}_{1,\delta}$ has a distance to the origin equal to:

$$l_1(x, 0) = \sum_{i=0}^{d} |x_i| \tag{14}$$

and the distance of every point in $\mathbf{S}_{2,\delta}$ to the origin is:

$$l_2(x, 0) = \sqrt{\sum_{i=0}^{d} x_i^2} \tag{15}$$

Now we have:

$$l_1(x, 0) = \sum_{i=0}^{d} |x_i| > l_2(x, 0) = \sqrt{\sum_{i=0}^{d} x_i^2} \tag{16}$$

$$l_1(x, 0) = \sum_{i=0}^{d} \sqrt{x_i^2} > l_2(x, 0) = \sqrt{\sum_{i=0}^{d} x_i^2} \tag{17}$$

which holds $\forall d \geq 2$ and $x_i \neq 0$. Thus, an l_1-sphere covers less space than an l_2-sphere since there are some points in the l_2-sphere that do not belong to the l_1-sphere, but the l_1-sphere is entirely comprised in the l_2-sphere. It should be obvious that this argument holds for every metric in the l_i family. It follows that the best choice to have a fine control over the sensibility of the algorithm is l_1. One may argue that other spheres could have better semantic properties, but our personal experience suggests that the finer the control, the better the results.

The Manhattan distance, l_1 metric, between two vectors u and v is defined as:

$$l_1(u, v) = \sum_{i=0}^{d} |u_i - v_i| \tag{18}$$

One can recall that higher metrics require root extraction which is an expansive operation. Clearly, l_1 is the fastest to compute in the l_i family. Being the best for precision control and the fastest to compute, it is a natural choice. It can be found in many geometry textbooks that l_1 satisfies the metric axioms (non-negativity, symmetry, identity and triangle inequality). Fulfilling such axioms allows us to use it in a metric tree.

However, to include the impact of the fragments relative size, it is best to normalize the metric. Thereafter, all queries will be specified with a real number in the interval $[0, 1]$. The chosen normalized Manhattan distance is the following:

$$\epsilon(u, v) = \frac{\sum_{i=0}^{d} |u_i - v_i|}{\sum_{i=0}^{d} max(u_i, v_i)} \tag{19}$$

Incidentally, this normalization of the Manhattan distance coincides with the Jaccard distance of the two sets, under certain hypotheses. Other Jaccard distances could be defined over the sets of fragment tokens, but this one suits our needs better since it is derived from the Manhattan distance.

D. Using a Nearest-Neighbor approach

Contrary to the approach presented in [19], this paper relies on finding the nearest-neighbor instead of performing a range-query to find the closest match. For the *move* retrieval problem, we need to search for a file that is likely to have generated the moved file. Between the time the file is originally moved and the time we try to retrieve its generator, the file might be altered in a way that it is no longer very close to the original. Thus, using a range-query to find multiple close candidates could result in missing those far-away *moves*. However, it is very likely that even if a file is deeply modified, the file that is the closest to it in the search space is the more likely to have generated it, no matter the distance. This problem is solved by finding the file's nearest-neighbor. Figure 2 outlines the procedure in a metric tree.

The algorithm is split-up in two core parts. Lines 1 to 16 compute the distance from the query to the pivots of the current node. It then selects the best match as the new nearest-neighbor if that match has a distance smaller than the current known nearest-neighbor. The rest of the algorithm recursively traverses each node that may have better candidates. Each condition tests whether or not it is possible to find a better match in the sub-tree rooted at that node. The criteria to visit each region are summarized in Table I.

TABLE I: Criteria for region selection in the findnn primitive.

Region 1	$\delta(x,f) < \epsilon + d \wedge \delta(y,f) < \epsilon + d$
Region 2	$\delta(x,f) + \epsilon \geq d \wedge \delta(y,f) < \epsilon + d$
Region 3	$\delta(x,f) < \epsilon + d \wedge \delta(y,f) + \epsilon \geq d$
Region 4	$\delta(x,f) + \epsilon \geq d \wedge \delta(y,f) + \epsilon \geq d$

Combining the metric tree, the frequency vectors and the nearest-neighbor search, we can now describe the Move Inferring Algorithm (MIA) shown in Figure 3. The procedure is straightforward. In MIA, a code fragment is a file. MIA is provided with two arguments: the $source$, which contains all files from a system at version N, and the $destination$, which contains all files at version N+1. It is assumed that every file has already been analyzed and transformed into frequency vectors. MIA iterates over all files in $destination$ and queries the metric tree of $source$ to find its nearest-neighbor in it. Then, it checks if the nearest-neighbor path exists in $destination$ and if the path of the query fragment is in $source$. If not, MIA indicates that the file added in $destination$ has a closely related file in $source$, but that this file no longer exists. This strongly indicates the presence of a *move*; the pair of paths is added to the results. At the end, MIA returns the list of all inferred *moves*.

```
1: findnn(node,f)
2: if node == NULL then
3:    return ERROR
4: end if
5: result = ∅
6: ϵ = ∞
7: d1 = δ(node.x, f)
8: d2 = δ(node.y, f)
9: if d1 < ϵ then
10:   result = node.x
11:   ϵ = d1
12: end if
13: if d2 < ϵ then
14:   result = node.y
15:   ϵ = f2
16: end if
17: if δ(node.x, f) < ϵ+node.d ∧ δ(node.y, f) < ϵ+node.d
    then
18:   (candidate, ϵ0) = findnn(node.region1, f)
19:   if ϵ0 < ϵ then
20:      result = candidate
21:   end if
22: end if
23: if δ(node.x, f)+ϵ ≥ node.d ∧ δ(node.y, f) < ϵ+node.d
    then
24:   (candidate, ϵ0) = findnn(node.region2, f)
25:   if ϵ0 < ϵ then
26:      result = candidate
27:   end if
28: end if
29: if δ(node.x, f) < ϵ+node.d ∧ δ(node.y, f)+ϵ ≥ node.d
    then
30:   (candidate, ϵ0) = findnn(node.region3, f)
31:   if ϵ0 < ϵ then
32:      result = candidate
33:   end if
34: end if
35: if δ(node.x, f)+ϵ ≥ node.d ∧ δ(node.y, f)+ϵ ≥ node.d
    then
36:   (candidate, ϵ0) = findnn(node.region4, f)
37:   if ϵ0 < ϵ then
38:      result = candidate
39:   end if
40: end if
41: return (result, ϵ)
```

Fig. 2: Nearest-neighbor algorithm in metric trees

IV. Experimental setup

A. Analyzed systems and computational equipment

All computations were performed on an Intel Core 2 Duo 2.16MHz, with 4GB of RAM, using an SSD hard-drive under Linux Fedora 15 OS. Clone detection executable binaries were compiled using g++ version 4.6.3 with the -O3 flag.

The systems on which MIA was evaluated are presented in Table II. The testbench is comprised of the Java web service manager Tomcat, the drawing application JHotDraw and the enterprise resource management Adempiere. For Tomcat and JHotDraw, we identified the release dates of many consecutive versions and downloaded the corresponding source code from their repository. For Adempiere the same procedure was followed, except for the first version, for which we had to infer a release date for fictitious versions, called 2.x.x and 2.y.y. It was necessary to do so because many *moves* in Adempiere didn't

TABLE II: Systems versions statistics

System	# Version	Date	# Files	LOCs	# Moves	# True Moves	# Ghost Moves
JHotDraw	8	2007-01-10 - 2011-01-09	1457-1665	217 372-281 082	299	186 (62%)	113 (38%)
Tomcat	19	2009-06-03 - 2012-04-05	1526-1681	402 843-433 148	19	2 (11%)	17 (89%)
Adempiere	6	2007-07-26 - 2010-06-14	3623-4217	652 261-1 186 149	847	766 (91%)	81 (9%)

1: **MIA** $(source, destination)$
2: $moves = \varnothing$
3: **for all** $f \in destination.fragments$ **do**
4: $nn = findnn(f, source.tree.root)$
5: **if** $nn \notin destination.file \wedge f \notin source.file$ **then**
6: $moves = moves \cup (nn, f)$
7: **end if**
8: **end for**
9: $return\ moves$

Fig. 3: Move inference algorithm (MIA)

seem to coincide with one of the later releases and the history of early releases is incomplete. Nevertheless, many repository commits were done between 2.x.x and 2.y.y and this should maintain the data validity. All systems have Subversion (SVN) repositories which may be found at [1], [2], [3].

B. Computing the nearest-neighbor

For each system, the nearest-neighbor of every file was computed using the algorithm presented in section III-B. For comparison purposes of our results, we actually computed two nearest-neighbors using a slight variation of the presented approach. We first computed the nearest-neighbor using the token *ID*, and then computed the nearest-neighbor using the token *image*. Although the token *ID* is usually the only representation used for clone detection, textual similarity might be more accurate for the search of *moves*, we decided to include results using token *image*. Both variants have been evaluated and compared.

C. Building the repository oracle using VCSMiner

We use our framework VCSMiner to extract file movements information from the source code repository of each subject system. VCSMiner extracts commit information from source code repositories (*e.g.,* CVS, SVN, and GIT) and stores the information into a database. We query this database to identify *moves* of files across the revisions of our studied systems. For VCSs such as CVS which reports *moves* of files implicitly, VCSMiner compares MD5 hashes of files to identify file movements. Using the extracted *moves*, we built an oracle for our problem. That oracle excludes every *move* outside the trunk of the repository. That experimental design choice was made to avoid interference between the many versions and the $tags$ and $branches$ directories. Since $tags$ and $branches$ are partial clones of the trunk directory, finding the nearest-neighbor in the whole repository could lead to find the nearest-neighbor in a random version, because there is no way for MIA to make the difference between identical files in two different paths. This choice does not alter the claim to identify *move* operations between versions, since the $tags$ and $branches$ directories are not part of the current version. Consequently, their exclusion is a reasonable choice.

D. Categorizing the oracle moves

Since the experiment is version-based instead of commit-based, it might happen that *move*d files cannot be identified as such because of file creation and deletion. We classified the reported repository *moves* in the following two categories:

- *True move* : The original file is in version N, but not in N+1, and the resulting file after the *move* is still in version N+1, but was not in N
- *Ghost move* : Both the original and the resulting file are missing from versions N and N+1

For the purpose of computing precision and recall on *move* identification, we consider only the oracle *moves* tagged as *true move*. The *ghost move* operation is impossible to identify when using source code of released versions only, thus precision and recall for this primitive is not reported. However, the total number of *moves* along both with the number of identified *true move* and *ghost move* is reported for each system. The reported percentage for the two different types of moves are relative to the total number of moves. From one version to another, as it is displayed in table II, the number of *ghost moves* is not generally significant, and there are enough *true move* to conduct a sound experiment.

Also, if the tool reported a move that wasn't reported by the VCS, then we classified it as an *implicit move*. An example of a legitimate implicit move would be a file movement not recorded by a developer in the VCS.

E. Methodology to compute precision and recall

Using the oracle produced by tool VCSMiner and classified according to the scheme introduced in section IV-D, we computed the recall of MIA.

As it will be discussed in section VI, the tool precision couldn't be evaluated with the chosen oracle and it had to be inferred statistically.

For recall, we compared all the possible $destination - source$ pairs, including pairs with the same $source$ parameter.

V. Results

Tables III, IV, and V show the detailed results for each system using the two variants of the algorithm. On average, using token *image* instead of *ID* gives a better recall. It also reports more implicit *moves*. However, after a quick inspection of the reported implicit *moves* in both cases, we concluded that more false-positives were produced by the *ID*-based algorithm. For the *image*-based algorithm, almost all pairs of $destination - source$ in implicit *moves* had the same file name at the end of the path, which was not the case for the *ID*-based version. Thus, qualitatively, the precision of the

TABLE III: JHotDraw recall of inferred moves, with implicit found moves.

Versions	Recall		#Implicit moves	
	Token Id	Token Image	Token Id	Token Image
7.0.8-7.0.9	0.0	0.9286	0	4
7.0.9-7.1.0	0.0	0.8571	0	3
7.1.0-7.2.0	0.0	0.7500	0	8
7.2.0-7.3.0	0.0	1.0000	0	1
7.3.0-7.4.1	0.0	0.9912	0	17
7.4.1-7.5.1	0.0	0.8667	0	6
7.5.1-7.6	0.4587	1.0	2	0
Summary	0.0765	0.9647	2	39

image-based algorithm is greater than the *ID*-based one. For this reason, we did not investigate further the results of the *ID*-based *moves* as they were clearly worse than the one reported by the *image* variant.

Table VI shows that for JHotDraw and Adempiere, on average the similarity of two different *move* types is really close. The averages for Tomcat cannot be compared in a meaningful way because it only contained 2 *true moves*. Despite this small number of *true moves*, results on Tomcat are not excluded because they help evaluate a limit case for the false-positive rate: they provide an answer to the question "how many results are reported when only few are expected ?" Evaluating limit case is as important as evaluating the average one. Figure 4 shows a box-plot of the implicit *moves* distance distribution excluding the *moves* with distance 0.0. As the analysis of their averages already suggested, JHotDraw and Adempiere seem to share some common characteristics regarding their *moves*; it may also suggest that MIA gives coherent results independently of the system. Because verifying all the implicit moves manually was not practical due to their large number, we randomly sampled and checked a representative sample of 100 implicit moves out of all the reported implicit moves and manually verified if they were real *moves* using information from both file names and contents. We tagged each sampled *move* as a "hit" or a "miss" by assigning it a real value of 1.0 for "hit" and 0.0 for "miss". The mean of "hit" and "miss" distribution corresponds to the precision of MIA. The mean of the "hit" and "miss" distribution was found to be 0.9700. The chosen sampling allows us to compute the precision of the implicit *moves* with a confidence level of 99%, and a confidence interval of 0.0422 using the Central Limit Theorem [15]. The theorem proves sample averages to be normally distributed if the sample is random, which is the case here. Therefore, the confidence interval is built using standard equations also provided in [15]. Building this interval does not require a specific test. The lower bound on precision of MIA is 0.92788 (*i.e.*, $0.9700 - 0.0422$). We combined the average recall obtained over all our subject systems (*i.e.*, 0.98) with the lower bound of the precision (*i.e.*, 0.92788) to compute the F-value. We found the F-value to be 0.9533. Hence, we conclude that MIA achieves a very high accuracy. We answer our research question *RQ1* positively. Our proposed technique MIA can successfully recover the revision history of a software system.

Figures 5, 6, and 7 display the running time of the algorithm, including the tokenizing preprocessing step. with respect to

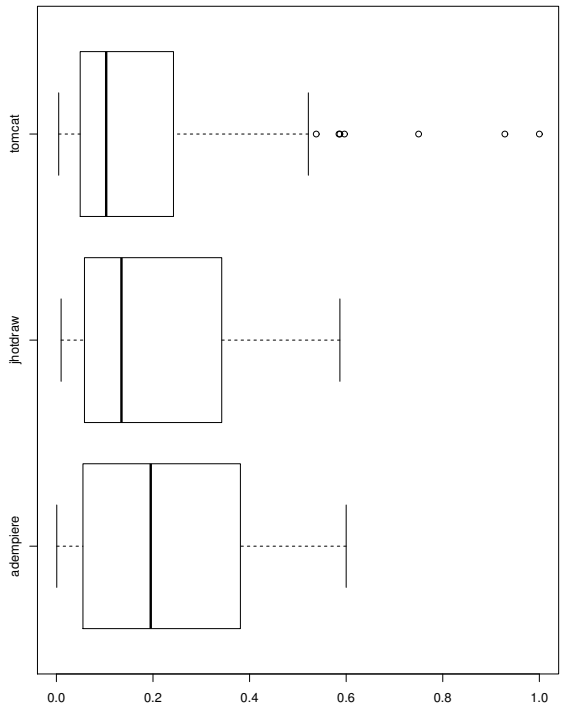

Fig. 4: Box-plot of distance distribution of inferred moves excluding identical matches for all systems

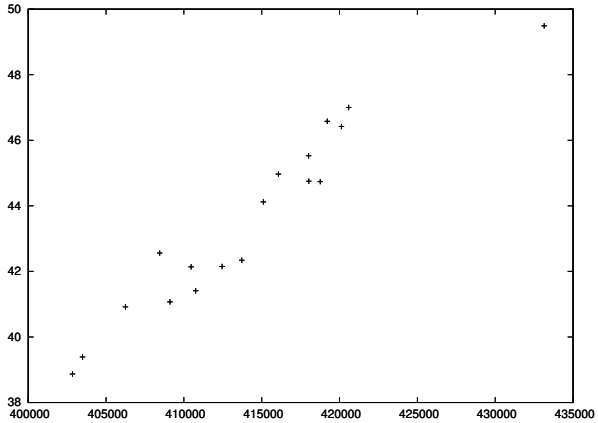

Fig. 5: Execution time (s.) of the Nearest-Neighbor clone detection for Tomcat with respect to the number of lines of codes in each version

TABLE IV: Tomcat recall of inferred moves, with implicit found moves.

Versions	Recall		#Implicit moves	
	Token Id	Token Image	Token Id	Token Image
6.0.20-7.0.0	1.0000	1.0000	0	0
7.0.0-7.0.4	1.0000	1.0000	0	0
7.0.4-7.0.5	1.0000	1.0000	1	1
7.0.5-7.0.6	1.0000	1.0000	0	0
7.0.6-7.0.8	1.0000	1.0000	0	0
7.0.8-7.0.10	1.0000	1.0000	0	0
7.0.10-7.0.11	1.0000	1.0000	0	0
7.0.11-7.0.12	1.0000	1.0000	63	63
7.0.12-7.0.14	1.0000	1.0000	0	0
7.0.14-7.0.16	1.0000	1.0000	3	3
7.0.16-7.0.19	1.0000	1.0000	9	9
7.0.19-7.0.20	1.0000	1.0000	1	1
7.0.20-7.0.21	1.0000	1.0000	0	0
7.0.21-7.0.22	1.0000	1.0000	0	0
7.0.22-7.0.23	1.0000	1.0000	0	0
7.0.23-7.0.25	1.0000	1.0000	0	0
7.0.25-7.0.26	1.0000	1.0000	0	0
7.0.26-7.0.27	1.0000	1.0000	2	2
Summary	1.0000	1.0000	79	79

TABLE V: Adempiere recall of inferred moves, with implicit found moves.

Versions	Recall		#Implicit moves	
	Token Id	Token Image	Token Id	Token Image
2.x.x-2.y.y	0.8571	1.0000	7	5
2.y.y-3.3.0	0.8490	1.0000	250	131
3.3.0-3.3.1	1.0000	0.9972	4	3
3.3.1-3.4.0	1.0000	1.0000	0	0
3.4.0-3.5.3	1.0000	1.0000	1131	970
3.5.3-3.5.4	0.0455	0.9091	31	22
3.5.4-3.6.0	1.0000	1.0000	3	0
Summary	0.8217	0.9844	0	1131

TABLE VI: Average similarity of True Move and Implicit Move for all systems

System	True moves average distance		Implicit moves average distance	
	Token *ID*	Token *image*	Token *ID*	Token *image*
JHotDraw	0.0509	0.1651	0.2336	0.1951
Tomcat	0.1997	0.2546	0.0261	0.0359
Adempiere	0.0687	0.0812	0.0921	0.0930

the lines of code in each version. The longest execution time is 600 seconds from an Adempiere version over 1 MLOC, but this data seems to be an outlier with respect to the other points gathered for Adempiere. After investigation, that point possibly had more cache misses in hard-drive read operations than the others and the occurrence seems to be random and not linked to that particular version.

VI. Discussion

The analysis focused on finding the closest code fragment from one version to what is the closest related fragment in another version. This is the first similarity analysis that uses a nearest-neighbor approach with a continuous distance instead of a binary matching without an underlying explicit distance. Using this proximity query allowed to find the most likely generator in version N of a fragment in version $N + 1$. According to the results presented in section V, it compares well to the *move* information provided by the repository of our system. It also supplies more information about implicit *moves* not recorded in the repository. Such conclusions establish the soundness of the algorithm to infer file structure modifications. As noted in section I, this implies the possibility of recovering repository information for systems that do not have one or to fix existing one.

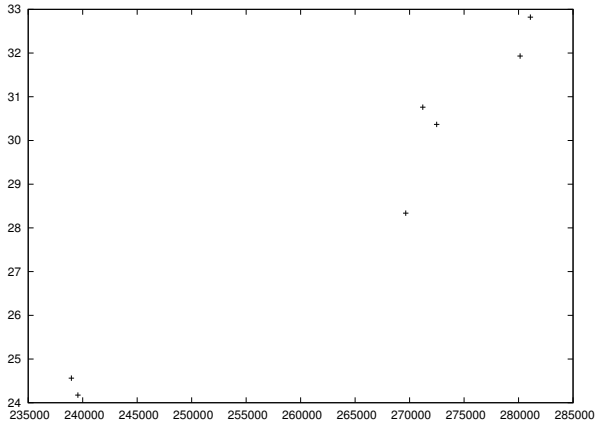

Fig. 6: Execution time (s.) of the Nearest-Neighbor clone detection for JHotDraw with respect to the number of lines of codes in each version

It was unexpected that the implicit *moves* would account for such a large proportion of all the *moves*. This makes the assessment of the precision complicated. By looking closely at our results, we observed that a large proportion of the implicit *moves* has a very small distance. Assuming our distance is accurate, it is more likely that these *moves* were

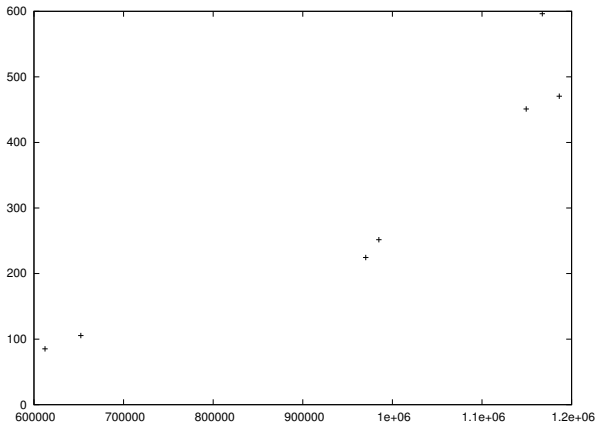

Fig. 7: Execution time (s.) of the Nearest-Neighbor clone detection for Adempiere with respect to the number of lines of codes in each version

done without the developers recording them in the VCS. If that is true, the information provided by the repositories may not be reliable as a complete oracle. This is the principal threat to validity to our experiment. Assessing recall like what was done in section V is necessary but not sufficient to prove true recall to be high because some *moves* seem to be missing from the oracle. Also, the proportion of implicit *moves* seems too large to do a precision analysis, since with reasonable arguments our results seem precise. Therefore, our study indicates the need for more data to compare repositories information to draw more conclusions on the general pattern of *moves* recording. Nevertheless, our experimental results show with strong evidences that MIA achieves results at least as good as the information provided by the existing repositories, if not better.

All experiments were done on systems of size above 100 KLOC, with the biggest of size 1.5 MLOC. Execution time of the clone detection step never exceeded 10 minutes for token *image* based analysis, and the lexical analysis step stayed below 5 minutes. Even if the technique may not be applied in real time, it is reasonable to say that it is scalable, since it runs under 10 minutes for systems of few MLOCs. As this experiment did not have any previous knowledge to enhance the searches, it was sound to use a nearest-neighbor approach even if it is time-expensive. However, as it turns out that the inferred *move* information is very akin to clone detection information, it may be possible to use a much faster range-query to find generator candidates and retain the closest result afterwards.

Finally, the move inference problem has proven to be solvable using techniques from clone detection. Moreover, observing the evaluation methodology proposed here might give a clever insight to challenge the clone detectors evaluation problem. Even if the proposed oracle seems to have flaws in its construction, it provides an interesting set of naturally occurring clones. Testing against the *move* reference might not be necessary and sufficient, but it can be an interesting indicator about tools capacity to detect small-gap and large-gap clones. Despite the evident flaws we already outlined, disregarding it would mean to eliminate a naturally occurring set of clones which might serve as a reference set.

Under the hypothesis that our tool is precise, the many *moves* identified that were not present in the repository information suggest that the technique may be used to suggest to developers to record certain modifications. It might be integrated in a version control system to automate the process of identifying *moves* between commit instead of asking developers to provide the information manually. This might help storing more meaningful information about software evolution in repositories. Automating the process of recording file structure modifications may also help developers to manage local sandboxes with greater ease and avoid file operation mistakes that are generally arduous to fix in VCS.

Moreover, the information contained in version control systems or repositories, such as Subversion and GIT, may also be used to deduce more information, as described in [5]. Repository information may also be used to provide version history editing as suggested by [4].

A. Threats to validity

This section discusses the threats to validity of our study following the guidelines for case study research [29].

Threats to construct validity concern the relation between theory and observation. In this work, the construct validity threats are mainly due to measurement errors. We extracted *moves* information from repositories but discovered that information is probably incomplete. This limits the interpretation of the recall as an upper bound instead of a definitive value and makes it impossible to evaluate the precision against the oracle. The alternative to measure precision may be biased by the opinion of the expert who inspected the implicit moves. To mitigate this bias, we inspected the name of the reported file to verify whether there was a coherence pattern and indeed there was. Thus, it increases the confidence in the reported precision value.

Threats to internal validity concern our selection of subject systems, tools, and analysis method. In [20], we measured the distortion of the Manhattan distance with the Levenshtein distance for clone detection and found the distortion to be very low in general. However, the seldom cases for which the distortion is not very low could affect our findings.

Threats to conclusion validity concern the relation between the treatment and the outcome. We paid attention not to violate assumptions of the performed statistical tests.

Threats to reliability validity concern the possibility of replicating this study. We attempt to provide all the necessary details to replicate our study. The source code repositories of Tomcat, JHotDraw and Adempiere are publicly available to obtain the same data for the same releases. We also provided all the algorithmic versions of our code with all the software experimental parameters and hardware configuration. Moreover, we published on-line[1] the raw data to allow other

[1] https://dl.dropbox.com/u/14931955/wcre12.zip

researchers to replicate our study and to test other hypotheses on our data set.

Threats to external validity concern the possibility to generalize our results. On the two systems with the most *moves*, we obtained similar precision and recall and a small difference between the average distance of the oracle *moves* and the implicit *moves*. Nevertheless, the validation of our technique is limited to three open source software systems written in Java, therefore, we cannot generalize our findings to other programming languages. However, the results are very encouraging and suggest further studies on different systems written in different programming languages to make our findings more generic.

VII. Conclusion

In this paper we have introduced a new technique to recover file structure modifications information and file movement informations in source code repositories. The technique proved to be highly precise (F-value = 0.9533) and scalable. The technique may also be applied to other reverse-engineering tasks that require file structure modification information. Further research could include, comparing the nearest-neighbor with range-query primitive as well as investigating the different proposed applications. In practice, our proposed technique can be integrated in existing VCSs to enrich their metadata with information on *implicit moves*; easing file structure manipulations and preventing mistakes that may require fixes that are hard to handle. The oracle used in the evaluations our proposed approach provides an interesting set of naturally occurring clones that can be reuse to evaluate clone detectors.

VIII. Acknowledgments

We would like to thank the anonymous reviewers of WCRE for their valuable feedback that helped us to improve the quality of the paper. Many thanks also go to Francois Gauthier and Anael Maubant for their many good advices on the writing of this paper. This research has been funded by the Natural Sciences and Engineering Research Council of Canada (NSERC) under the Discovery Grants Program and the Fonds Quebecois Nature et Technologie (FQRNT).

References

[1] Adempiere. http://sourceforge.net/projects/adempiere/.

[2] Jhotdraw. http://sourceforge.net/projects/jhotdraw/.

[3] Tomcat. http://tomcat.apache.org.

[4] D. L. Atkins. Version sensitive editing: Change history as a programming tool. In *ECOOP 98, SCM-8, LNCS 1439*, pages 146–157. Springer-Verlag, 1998.

[5] T. Ball, J. min Kim, A. A. Porter, and H. P. Siy. Abstract if your version control system could talk...

[6] H. Basit, S. Pugliesi, W. Smyth, A. Turpin, and S. Jarzabek. Efficient token based clone detection with flexible tokenization. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2007.

[7] I. Baxter, A. Yahin, l. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance - IEEE Computer Society Press*, pages 368–377, 1998.

[8] S. Brin. Near neighbor search in large metric spaces. In *Proceedings of the 21th International Conference on Very Large Data Bases*, VLDB '95, pages 574–584, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[9] J. Cheney, S. Chong, N. Foster, M. I. Seltzer, and S. Vansummeren. Provenance: a future history. In *OOPSLA Companion*, pages 957–964, 2009.

[10] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Proc. of 23rd International Conference on Very Large Data Bases*, pages 426–435. Morgan Kaufmann Publishers, 1997.

[11] S. Ducasse, O. Nierstrasz, and M. Rieger. On the effectiveness of clone detection by string matching. *International Journal on Software Maintenance and Evolution: Research and Practice - Wiley InterScience*, (18):37–58, 2006.

[12] D. M. German, M. D. Penta, G. Antoniol, and Y. gal Guhneuc. Code siblings: Phenotype evolution. In *In Proc. of the 3rd Intl. Workshop on Detection of Software Clones*, 2009.

[13] N. Göde and R. Koschke. Incremental clone detection. In *Proceedings of the 2009 European Conference on Software Maintenance and Reengineering*, pages 219–228. IEEE Computer Society, 2009.

[14] M. W. Godfrey, D. M. German, J. Davies, and A. Hindle. Determining the provenance of software artifacts. In *Proceedings of the 5th International Workshop on Software Clones*, IWSC '11, pages 65–66, New York, NY, USA, 2011. ACM.

[15] W. Hines. *Probability and Statistics in Engineering*. John Wiley, 2003.

[16] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multi-linguistic token-based code clone detection system for large scale source code. volume 28, pages 654–670. IEEE Computer Society Press, 2002.

[17] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pages 187–196, New York, NY, USA, 2005. ACM.

[18] R. Koschke, I. D. Baxter, M. Conradt, and J. R. Cordy. Software Clone Management Towards Industrial Application (Dagstuhl Seminar 12071). *Dagstuhl Reports*, 2(2):21–57, 2012.

[19] T. Lavoie and E. Merlo. Automated type-3 clone oracle using levenshtein metric. In *Proceedings of the 5th International Workshop on Software Clones*, IWSC '11, pages 34–40, New York, NY, USA, 2011. ACM.

[20] T. Lavoie and E. Merlo. An accurate estimation of the levenshtein distance using metric trees and manhattan distance. In *Proceedings of the 6th International Workshop on Software Clones*, IWSC '12, pages 1–7, New York, NY, USA, 2012. ACM.

[21] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of the International Conference on Software Maintenance - IEEE Computer Society Press*, pages 244–253, Monterey, CA, Nov 1996.

[22] E. Merlo, G. Antoniol, M. D. Penta, and F. Rollo. Linear complexity object-oriented similarity for clone detection and software evolution analysis. In *Proceedings of the International Conference on Software Maintenance - IEEE Computer Society Press*, pages 412–416. IEEE Computer Society Press, 2004.

[23] E. Merlo and T. Lavoie. Detection of structural redundancy in clone relations. Technical Report EPM-RT-2009-05, Ecole Polytechnique of Montreal, 2009.

[24] D. L. Parnas. Software aging. In *Proceedings of the 16th international conference on Software engineering*, ICSE '94, pages 279–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[25] J. Pate, R. Tairas, and N. Kraft. Clone Evolution: A Systematic Review. *Journal of Software Maintenance and Evolution: Research and Practice*, Sept. 2011.

[26] C. Roy and J. Cordy. A survey on software clone detection research. Technical Report Technical Report 2007-541, School of Computing, Queen's University, November 2007.

[27] C. K. Roy and J. R. Cordy. Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*, ICPC '08, pages 172–181, Washington, DC, USA, 2008. IEEE Computer Society.

[28] J. K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Inf. Process. Lett.*, 40(4):175–179, 1991.

[29] R. K. Yin. *Case Study Research: Design and Methods - Third Edition*. SAGE Publications, London, 2002.