PAPER

Approximating the Evolution History of Software from Source Code

Tetsuya KANDA[†a)], ***Nonmember***, **Takashi ISHIO**[†b)], ***Member***, ***and*** **Katsuro INOUE**[†c)], ***Fellow***

SUMMARY Once a software product has been released, a large number of software products may be derived from an original single product. Management and maintenance of product variants are important, but those are hardly cared because developers do not make efforts for the further maintainability in the initial phase of software development. However, history of products would be lost in typical cases and developers have only source code of products in the worst case. In this paper, we approximate the evolution history of software products using source code of them. Our key idea is that two successive products are the most similar pair of products in evolution history, and have many similar source files. We did an experiment to compare the analysis result with actual evolution history. The result shows 78% (on average) of edges in the extracted trees are consistent with the actual evolution history of the products.

key words: *software evolution, software product line, visualization*

1. Introduction

When developing a software product, clone-and-own approach is one of the major and easy ways to realize software reuse [1]. Developers copy existing code or the whole of the product and then add features, fix bugs, and so on. A software product contains source files, images, documents, and the other resources. In this paper, we define "a source file" as a source code in the single file and "a software product" as a set of source files.

The new version of the first product is released with slightly different features, so it will have very similar files with the first one. Management of such similar software products is a very important task. They might have the same problems or bugs, or developers can apply same improvement in them. However, developers often copy and modify the software product without using version control systems (VCS) or other management techniques [2] since no one knows whether the product would be successful enough to apply many extensions and derive many variants. Using #IFDEF macro in C language to describe product specific features is one of the solutions, but it is believed to decrease code readability. Clone-and-own approach also gives developer freedom of making changes, without considering making an impact to existing projects.

Many re-engineering methods for existing software products have been proposed [3]–[5]. Since analyzing a large number of software products is a difficult task, Krueger *et al.* suggested that developers should start their analysis from a small number of software products [6]. Koschke *et al.* proposed an extension of reflexion method to construct a product line by incrementally analyzing products [4]. To follow these reasonable approaches, developers must choose representative software products as a starting point. If the history of software evolution is available, developers could recognize the relationships among the products and choose representatives for their analysis. For example, compare products between branches to extract common features and product specific features. In the point of view of re-engineering, understanding the evolution history of software is also an important thing.

However, the history of software products is often not available [7]. Software products are not always managed under the VCS. If the software has branched and managed independently, relationships between branches are not recorded. Some of experts know the whole of the software products, but their knowledge is often incomplete [8]. In the worst case, developers only have access to source code of each product, they cannot get version numbers nor release date for some of the products.

We assume that two successive products are the most similar pair in the products. Similar software products must have similar source files so we analyze the source files and count the number of similar source files between products. We connect the most similar products and construct a tree. This tree is an approximation of the evolution history of software products and two successive products will be connected. Our approach depends only on source files, so we can analyze products whose evolution history is lost; no version numbers, names or release dates.

This paper is an extension to our previous research [9]. The previous algorithm used the number of similar files only and did not care how much the files are changed; both the file pair with no changes and the file pair with small changes are treated as similar files. The new contributions of this paper are follows:

- We have introduced a weighted function between two software products to reflect the effect of small changes.
- We extend an experiment target to programs written in C and Java.
- We did a case study with two variants of Linux kernel

Manuscript received August 19, 2014.
Manuscript revised January 14, 2015.
Manuscript publicized March 17, 2015.

[†]The authors are with the Graduate School of Information Science and Technology, Osaka University, Suita-shi, 565–0871 Japan.
a) E-mail: t-kanda@ist.osaka-u.ac.jp
b) E-mail: ishio@ist.osaka-u.ac.jp
c) E-mail: inoue@ist.osaka-u.ac.jp

DOI: 10.1587/transinf.2014EDP7286

and found out their origin.

2. Related Work

2.1 File Similarity

When comparing software products, similarity between source files is a very important metric. To find out the same or similar source code fragments, many code clone detection tools have been proposed [10], [11]. Using large-scale code clone detection techniques, Hemel and Koschke compared Linux kernel and its vendor variants [12]. They found vendor variants included various patches, but the patches are rarely submitted to the upstream. Another application of code clone detection is detecting file moves occurred between released versions of a software system [7].

Yoshimura *et al.* visualized cloned files in industrial products [13]. They have used an edit distance function as a source file similarity to find out cloned files whose contents are almost the same. Inoue *et al.* [14] proposed a tool named Ichi Tracker to investigate a history of a code fragment with source code search engines. It visualizes how related files are similar to the original code fragment and when they are released. With the visualization, developers can identify the origin of the source code fragment or a more improved version. Our approach enables similar analysis on software products instead of source files.

We have assumed that two successive products are very similar to each other. This observation is shown by Godfrey *et al.* [15]. They detected merging and splitting of functions between two versions of a software system. Their analysis shows that a small number of software entities such as functions, classes or files are changed between two successive versions. Lucia *et al.* reported that most of bug fixes are implemented in a small number of lines of code [16]. Since these analysis reported that two successive versions are very similar, we infer that the most similar pairs of products are likely two successive versions.

2.2 Software Evolution

Yamamoto *et al.* proposed SMAT tool that calculates similarity of software systems by counting similar lines of source code [17]. They identify corresponding source files between two software systems using CCFinder [10], and then compute differences between file pairs. They applied their tool to a case study of software clustering, and extracted a dendrogram of BSD family. The dendrogram reported which OSs are similar to each other. Tenev *et al.* introduced bioinformatics concepts into software variants analysis [18]. One of them is phylogenetic trees, which visualizes the similarity relations. They constructed dendrogram and cladogram from six of BSD family for example of phylogenetic trees.

They can show the relationship that which product is most similar to another and which products were forked from the release. Although their approaches and goals are similar to our idea, our approach visualizes more concrete relationships among products which are not shown in those related works; which product was first released, their evolution direction, and so on.

2.3 Software Categorization

Several tools have been proposed to automatically categorize a large number of software based on their domains such as compiler, database, and so on. MUDABlue [19] classifies software based on similarity of identifiers in source code. MUDABlue employed latent semantic analysis which extracts the contextual-usage meaning of words by statistical computations. LACT [20] uses latent dirichlet allocation in which software can be viewed as a mixture of topics. LACT used identifiers and code comments, but excluded literals and programming language keywords, to improve categorization. CLAN [21] focused on API calls. Its basic idea is that similar software uses the same API set.

While all of these tools are able to detect similar or related applications from a large set of software products, our approach focuses on very similar products derived from the same product, that are likely categorized into the same category by these tools.

3. Approach

We define the "Product Evolution Tree" as a spanning tree of complete graph which includes all input products and connects most similar product pairs first. If many files are similar between two products, it means that those products are similar. A simple example of the tree is shown in Fig. 1. Each node represents a software product. Each edge indicates that a product is likely derived from another product and the direction of derivation: which product is an ancestor and which product is a successor. A label of an edge explains the number of similar files between products. In Fig. 1, the product branched and there are more similar files between A2 and A3 than A2 and B1.

We construct a Product Evolution Tree from source code of products through four steps as follows.

1. We calculate file-to-file similarity for all pairs of source files of all products.
2. We count the number of similar files between two products.
3. We construct a tree of products by connecting most similar product pairs.
4. We calculate evolution direction based on the number of modified lines between two products.

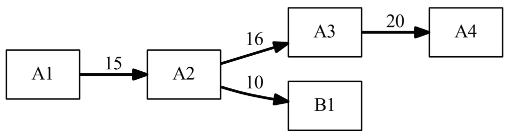


Fig. 1 An example of a product evolution tree.

3.1 File Similarity

We calculate similarity for all pairs of files across different products. We do not consider file names because a file may be renamed. To calculate the similarity of two source files, we first normalize each of source files into a sequence of tokens. In a normalized file f_n, which is a sequence of tokens of file f, each line has only a single token. We remove blanks and comments since they do not affect the behavior of products. All other tokens including keywords, macros and identifiers are kept as is. Given a pair of files (a, b), their file similarity $sim(a, b)$ is calculated as follows:

$$sim(a,b) = \frac{|LCS(a_n, b_n)|}{|a_n| + |b_n| - |LCS(a_n, b_n)|}$$

where $|LCS(a_n, b_n)|$ is the number of tokens in the Longest Common Subsequence between a_n and b_n. This is the deformed expression of $sim(a, b)$ in [9] using $|ADD(a_n, b_n)| = |a_n| - |LCS(a_n, b_n)|$ and $|DEL(a_n, b_n)| = |b_n| - |LCS(a_n, b_n)|$.

We have used a file similarity based on LCS, since we could optimize the calculation as described in Sect. 3.6. Another reason is that LCS-based technique like UNIX diff is one of the most popular choices in comparing source code. There are famous metrics for measuring similarity of documents such as TF-IDF, jaccard similarity, and so on. Of course, those metrics can be applied to the source files (we are using jaccard similarity in optimization), but they are based on the term frequency and do not consider the order of elements. The following computation steps did not depend on the definition of file similarity function; hence, other methods such as code clone detection are also applicable to compute file similarity.

3.2 Count the Number of Similar File Pairs

When the file pair has a higher similarity than a threshold, it is a similar file pair. The set of all possible similar file pairs S is defined as:

$$\begin{aligned} &S(P_A, P_B, th) \\ &\quad = \{(a,b) \mid a \in P_A, b \in P_B, sim(a,b) \geq th\}. \end{aligned}$$

and the number of similar file pairs N between software products P_A and P_B are defined as:

$$N(P_A, P_B, th) = |S(P_A, P_B, th)|.$$

3.3 Construction of the Tree

In this step, we construct a spanning tree of products. We first construct a complete undirected graph $G = (P, E)$, P denotes that software product and E denotes set of edges that connects all those products. From this graph, we pick edges with maximum number of similar files and add to the tree, without making a loop, until all nodes are connected. This is the same operation of the well-known algorithm of the minimum spanning tree. As a result, we get a spanning tree $S = (P, E')$ of the graph G. $E' \subseteq E$ is a set of edges which have the largest number of similar file pairs as follows:

$$\sum_{(P_i, P_j) \in E'} N(P_i, P_j, th).$$

If two or more edges have the same weight values, one of them can be arbitrary selected. In our implementation, it depends on the input order.

3.4 Evolution Direction

After a spanning tree is constructed, we set the direction on each edge which explains the direction of evolution. Our hypothesis is that source code is likely added, so we count the amount of added code in two software products as follows:

$$ADD(P_A, P_B) = \sum_{(a,b) \in S(P_A, P_B, th)} |b_n| - |LCS(a_n, b_n)|$$

where a_n and b_n are the normalized source files. Evolution direction is defined as follows:

$$\begin{cases} ADD(P_A, P_B) > ADD(P_B, P_A) \Rightarrow P_A \rightarrow P_B \\ ADD(P_A, P_B) = ADD(P_B, P_A) \Rightarrow P_A - P_B \\ ADD(P_A, P_B) < ADD(P_B, P_A) \Rightarrow P_A \leftarrow P_B. \end{cases}$$

Direction "–" means no direction detected.

We put directions and labels which denote the number of similar files on each edge of the tree. The Product Evolution Tree is completed through these four steps.

3.5 Weighted Function

The function N explains the number of similar source files. When the software product series goes to maintenance phase, there would be no drastic changes so that changes will not decrease file similarity below the threshold. This means that N cannot explain how much the source code is changed. To reflect the amount of changes to the function, we define another function N_w that weighting the function N with sim:

$$N_w(P_A, P_B, th) = \sum_{(a,b) \in S(P_A, P_B, th)} sim(a,b).$$

sim is already computed in the Step 1 so that we can get N_w without vast amounts of calculating cost. We compare these two functions in the experiment.

3.6 Optimization

To reduce the computation time, we introduced an implementation technique that calculates sim value only if it seems greater than the similarity threshold. The technique is based on the jaccard similarity of two documents. We introduce the term frequency $tf(f, t)$ which represents how many times term t appears in file f. For example, suppose two tokenized files a_n = AAABB and b_n = ABBBB, where A and B are terms in the files. The term frequencies are $tf(a_n, A) = 3$, $tf(a_n, B) = 2$, $tf(b_n, A) = 1$, and $tf(b_n, B) = 4$. Since $LCS(a_n, b_n)$ can include at most one A and two Bs shared by the sequences, the maximum length of $LCS(a_n, b_n)$ is 3.

The maximum length of $LCS(a_n, b_n)$ is calculated as:

$$\sum_{t \in T} min(tf(a_n, t), tf(b_n, t))$$

and we can get maximum similarity

$$msim(a, b) = \frac{\sum_{t \in T} min(tf(a_n, t), tf(b_n, t))}{\sum_{t \in T} max(tf(a_n, t), tf(b_n, t))}$$

of each file pair (a, b) using term frequency. T represents the set of terms appeared in all source files. The value of $sim(a, b)$ equals to $msim(a, b)$ if all the common tokens appear in the same order in two sequences. If the order of tokens is different from another sequence, then $sim(a, b)$ is smaller than $msim(a, b)$. A fomula $msim(a, b) \geq sim(a, b)$ is always true, hence we compute $sim(a, b)$ only if $msim(a, b)$ is greater than the similarity threshold.

3.7 Simple Example

Here is a simple example of the algorithm. In this section, we use two products shown in Fig. 2. We shorten "Product 1" to P_1 and "File A of Product 1" to P_1-A.

File Similarity We calculate all file pairs among P_1 and P_2. The similarity value among those producs are follows:

	P_1-A	P_1-B	P_1-C
P_2-A	0	0	0
P_2-B	0	0.33	0
P_2-C	0	0	0.66

Count the Number of Similar File Pairs When we set the similariy threshold $th = 0.5$, only (P_1-C, P_2-C) is the similar file pair. The cost is $N(P_1, P_2, 0.5) = 1$ and $N_w(P_1, P_2, 0.5) = 0.66$.

Construction of the Tree In this exaple, we have only two products so we just connect them.

Evolution Direction In the similar file pair (P_1-C, P_2-C), P_2-C has one more token "lemon" than P_1-C and no unique token in P_1-C. Please note that P_1-B and P_2-B shares some code but those files are "not similar" so the algorithm does not consider the changes between them.

As a result, $ADD(P_1, P_2) = 1$, $ADD(P_2, P_1) = 0$ so the evolution direction is "$P_1 \rightarrow P_2$".

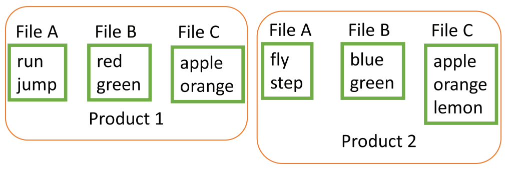


Fig. 2 An example input.

4. Experiment

We have implemented our approach as a tool and conducted an experiment. The goal of the experiment is to evaluate how accurately the Product Evolution Tree recovers the actual evolution history. We have used similarity threshold $th = 0.9$ in this experiment, which is experimentally determined.

4.1 Dataset

We have prepared nine datasets using open source projects, six of them are implemented in C and the other three of them are implemented in Java. The complete list of the dataset is on Table A· 1 and the input order for the tool is same as the table.

PostgreSQL. It is a database management system. In the evolution history of PostgreSQL, each major version was released from the master branch after developing beta and RC releases. After a major version had been released, a STABLE branch was created for minor releases and the master branch was used for developing the next beta version. While each release archive contains a large amount of files, we used only source files under "src" directory in this experiment.

The evolution history of PostgreSQL is simple and well-formed so we select four datasets from PostgreSQL to evaluate some kind of situation.

Dataset 1: Pgsql-major is a dataset whose evolution history is straight, *i.e.*, it has no project forks. *Dataset 2: Pgsql8-all* is a dataset whose evolution history is a tree of a single project with a large number of variants. *Dataset 3: Pgsql8-latest* is a dataset that includes only recent products. If a product family has a long history, older products may be no longer available for developers. *Dataset 4: Pgsql8-annually* is another dataset that a full collection of products is not available. Dataset 4 contains releases which have been released around September from 2005 to 2012.

FFmpeg and Libav. They are libraries and related programs for processing multimedia data. Libav is forked from FFmpeg and is developed by a group of FFmpeg developers. They are independently developed, but similar changes have been applied to both products.

Dataset 5: FFmpeg is a dataset whose project has been forked to two projects. This dataset is created to evaluate whether our approach can recover the evolution history of forked projects or not.

4.4BSD, FreeBSD, NetBSD and OpenBSD. These operating systems are derived from BSD, but they are now independent projects. Figure 4 shows a part of the family-tree for the versions selected for our dataset. According to the tree, NetBSD-1.0 is not only derived from NetBSD-0.9 but also from 4.4BSD Lite. FreeBSD-2.0 is also based on 4.4BSD Lite. OpenBSD is the forked project of NetBSD. 4.4BSD Lite2 affects other BSD operating systems. For each version, we used source files under "src/sys" directory.

Dataset 6: BSD is a dataset whose project has been forked to more than three projects. The evolution history is the most complex in our datasets and there are releases created by merging source code from more than one product. Since our approach extracts only a tree, our approach must miss such merged edges.

Groovy. This is an agile and dynamic language for Java Virtual Machine. In the evolution history of Groovy, each release has own branch. Since they all branched just before the release and there are no changes in source files comparing with original branch, we can say that the evolution history of Groovy is very similar to that of PostgreSQL. We used only source files under "src" directory.

Dataset 7: Groovy is a small dataset of Java application. In the VCS, each release has branched from the main branch, but it has completely same source code so we did not consider such small branches.

Apache hibernate. This is an object relationship mapping library for Java. This evolution history is also similar to PostgreSQL and Groovy. Each major version is developed on their own branches. We used only source files under "hibernate-core" directory.

Dataset 8: hibernate is a large dataset of Java application. This dataset contains 3 branches and 61 versions. Some of them has special version names like "4.2.7SP" and they makes the evolution history bit complex.

OpenJDK. This is an open-source implementation of Java. The OpenJDK project firstly released OpenJDK7, and implement OpenJDK6 from it. We analyze files under "src/share/classes" directory.

Dataset 9: OpenJDK6 is a dataset which represents unusual evolution history. This dataset contains initial OpenJDK6 (the copy of OpenJDK7) and its children. The product starts with OpenJDK7 and modified to implement "old" Java6 standard. So this dataset considered not to follow the standard evolution; implementing new and rich features into later version.

4.2 Results Overview

The correctness of the edges and labels is shown in Table 1 and Table 2. Column "#" denotes the dataset number. Column "H. (History)" denotes the number of edges in the evolution history and "O. (Output)" denotes number of edges in the Product Evolution Tree. Column "Matched Edges" shows how many edges are matched with the actual history without considering direction. In other words, we only checked the shape of the tree. Column "Matched Labels" shows how many correct edges have correct direction. Column "Recall" indicates the proportion of correctly identified edges to edges in an actual evolution history. N is the metrics that we have adopted in [9] but we improved and removed bugs in the implementation so the result is different from the previous paper.

We did not calculate precision in this experiment, since the precision is higher than or the same as the recall. This is because the number of edges in the Product Evolution Tree is the same as or less than the number of edges in the actual evolution history. If the dataset which consist of N products does not contain the loop, the number of edges in the dataset is $N-1$ and the number of edges in our tree is also $N-1$. So the number of false positive edges is always the same number of false negative edges and the precision is the same value as the recall. Only the Dataset 6 contains the loop so the number of false positive edges is smaller than the number of false negative edges and the precision is smaller than the recall.

Table 1 Result with N.

#	H.	O.	Matched Edges		/Labels		Recall
1	13	13	13	(100%)	13	(100%)	100%
2	143	143	106	(74.1%)	104	(98.1%)	72.7%
3	37	37	24	(64.9%)	24	(100%)	64.9%
4	24	24	20	(83.3%)	20	(100%)	83.3%
5	15	15	1	(6.7%)	1	(100%)	6.7%
6	17	15	11	(64.7%)	11	(100%)	64.7%
7	36	36	28	(77.8%)	22	(78.6%)	61.1%
8	61	61	52	(85.2%)	46	(88.5%)	75.4%
9	15	15	8	(53.3%)	5	(62.5%)	33.3%

Table 2 Result with N_w.

#	H.	O.	Matched Edges		/Labels		Recall
1	13	13	13	(100%)	13	(100%)	100%
2	143	143	137	(95.8%)	132	(96.4%)	92.3%
3	37	37	30	(81.1%)	30	(100%)	81.1%
4	24	24	20	(83.3%)	20	(100%)	83.3%
5	15	15	14	(93.3%)	14	(100%)	93.3%
6	17	15	11	(64.7%)	11	(100%)	64.7%
7	36	36	30	(83.3%)	24	(80.0%)	66.7%
8	61	61	53	(86.9%)	47	(88.7%)	77.0%
9	15	15	13	(86.7%)	7	(53.8%)	46.7%

Comparing the result with N and N_w, N_w performed better and Dataset 5 is a case that weighted function has worked most effectively. When the project forks, it has already been in the maintenance phase and few changes are adopted to the forked releases. As a result, all file pairs exceeds the similarity threshold 0.9 and the number of similar files between any two of the dataset are the same value ($N = 618$) so almost all edges showed wrong evolution. Using weighted function N_w, we can reflect the effect of small changes and the tree well approximates the evolution history so we discuss the result with N_w below.

4.3 Patterns of Incorrect Edges

Even though our approach connects most likely similar products, some edges are mismatched with the actual evolution history. To analyze mismatches, we have categorized incorrect edges in Product Evolution Trees into 5 patterns as follows. In Fig. 3, each left graph shows an actual evolution history and each right graph shows an extracted Product Evolution Tree. Thin edges are the connections that exist in the actual history. Thick, dashed edges are extracted by our approach, but they do not exist in the actual history.

P1: Version Skip. This pattern is found in three successive versions; two edges v_1 to v_3 and v_2 to v_3 are detected instead of a path from v_1 to v_3 via v_2. Figure 3 (a) shows an example. This pattern happens when v_2 and v_3 have the

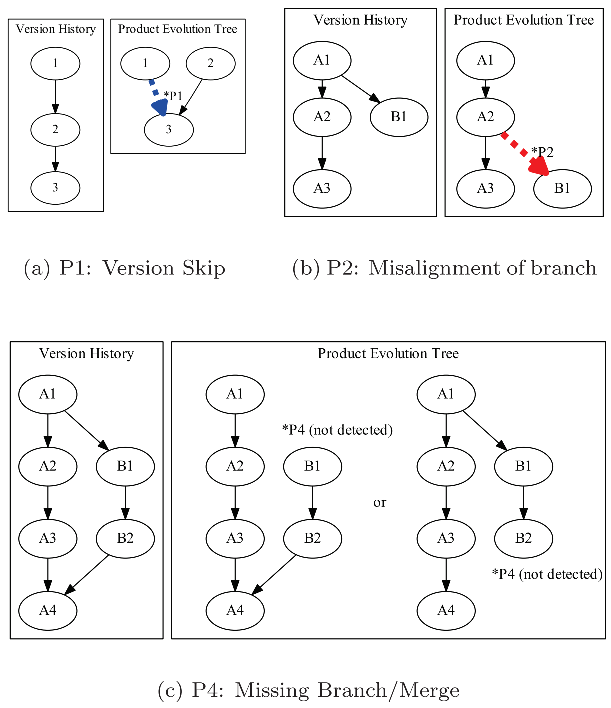


Fig. 3 Patterns of incorrect edges.

same N_w value from v_1 or the N_w between v_1 and v_3 is large. In addition, we classify edges into this category only when the edge skips one version. If the edge skips two or more versions, it classified into P5: Out of Place.

In Dataset 9 for example, tags "b13" and "b15" are connected in the tree and "b14" is skipped. One developer said in his blog that "b15" is tagged just for mark as switching VCS to mercurial. There are no difference in any files between "b14" and "b15" so that $N_w(b13, b14, 0.9)$ and $N_w(b13, b15, 0.9)$ are the same value.

P2: Misalignment of Branch. An edge connects two branches but does not connect actually branched products. In Fig. 3 (b), there are two branches A and B. While B1 was actually forked from A1, the origin of branch B was recognized as A2. In this pattern, A2 actually has more similar files, comparing with B1 than A1.

In Dataset 2, almost all edges connecting branches are not matched. We found that this is because branched products share the same changes. For example, 8.2BETA1 is developed on the master branch as the next version of 8.1.0, but extracted tree says this is the next version of 8.1.5. We examined git repository and found that version 8.1.5 is released right after 8.2BETA1. The master branch developing 8.2BETA1 and STABLE branch for 8.1 received 225 commits that are submitted on the same date with the same log message, but there are only 28 commits unique to the master branch. This fact also means that the actual evolution history does not always show functional differences of products.

P3: Misdirection. An edge connects accurate products, but its label shows the reverse direction. It happens when the size of source code or the number of source files decreased by several activities such as refactoring and deletion of dead code. In the other case, if two versions have the same source files, our approach cannot define the evolution direction.

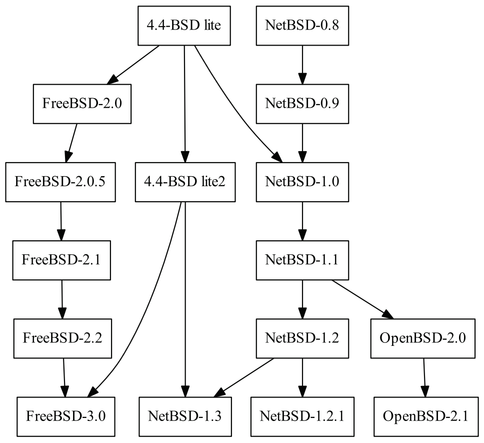


Fig. 4 A family-tree of Dataset 6.

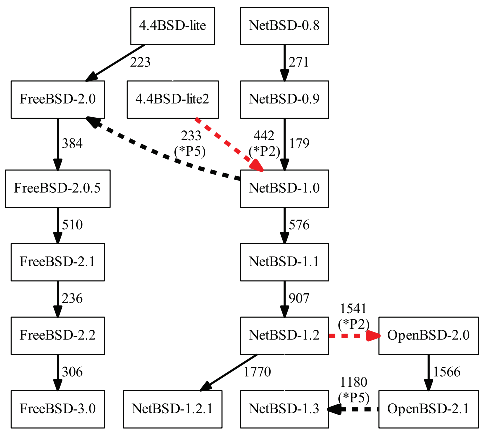


Fig. 5 A Product Evolution Tree of Dataset 6.

Many of this pattern show reversed direction, but other edges around thems show accurate direction, so it is easy to recognize that those edges connects exact products but the direction is reversed. In the case of Dataset 8, two of misdirection patterns, 4.1.2–4.1.2.Final and 4.3.3Final–4.3.4Final, have no direction. A comment in VCS says that there are no changes but the developer tagged them again.

P4: Missing Branch/Merge. Our Product Evolution Tree cannot detect a branch or a merge of two products derived from a single product. In Fig. 3 (c), we can see that the Product Evolution Tree misses branching from version A1 to version A2 and B1 or merging from version B2 to A4. In this pattern, one edge is missing but no wrong edges are output. If an actual evolution history includes a merge (e.g. Dataset 6), 100% recall is not achievable.

This pattern appears in Dataset 6. Figure 4 shows the family-tree and Fig. 5 output of our approach. The Product Evolution Tree included a merge relationship for NetBSD-1.0. It is the next release of NetBSD-0.9 and includs many source files from 4.4-BSD Lite. On the other hand, an edge from 4.4BSD Lite2 to FreeBSD-3.0 is not detected because the Product Evolution Tree does not allow closed paths. In addition, N_w (4.4BSD Lite 2, FreeBSD-3.0, 0.9) = 40 in-

Table 3 Release date of BSD family.

BSD	date
NetBSD 1.2	1996-10-04
OpenBSD 2.0	1996-10-18
OpenBSD 2.1	1997-06-01
NetBSD 1.3	1998-01-04

dicated that all except for 40 files are different between two versions. The relationship from 4.4BSD Lite2 to FreeBSD-3.0 in the family tree may not be captured by the source code difference.

P5: Out of Place. This pattern is a falsely detected edge which is not classified into previous patterns. There are no relationship between the wrong edge and the actual history.

4.4 Discussion

The result shows that 65% to 100% of edges without labels and 47% to 100% of edges with labels are consistent with the actual evolution history.

From the shape of the Product Evolution Tree, developers can learn where the starting point of the evolution is and where they branched. Almost all of the latest products of each branch are represented as leaf nodes, except Dataset 6. Value of the function N_w also provides hints to understand an evolution history. If a vertex has three edges and one of them has a small number of similar files, it may indicate branching and others may indicate the mainline.

Take a look at Fig. 5, FreeBSD-2.0, NetBSD-1.0, and NetBSD-1.2 will get attention because they have more than two edges. Leaf nodes 4.4BSD Lite, 4.4BSDLite2, FreeBSD-3.0, NetBSD-0.8, NetBSD-1.3, and NetBSD-1.2.1 also seem important. The tree suggests that OpenBSD-2.1 is not a characteristic release. It is hard to find out that they are important releases in this dataset.

If the time had passed from previous releases, they would apply the same changes. In Dateset 6 for example, OpenBSD Project is forked from NetBSD 1.1 but its first official release is in October 1996. NetBSD 1.2 is released just before OpenBSD 2.0 was released so we can imagine that there are same changes in NetBSD and OpenBSD. The same things can be said in OpenBSD 2.1 and NetBSD 1.3, showed in Table 3.

Major error P3 is a counterexample for our hypothesis that "source code is likely added". One reason is that refactoring such as class splitting and merging have been applied. Techniques for detecting refactoring [22] may be helpful to remove incorrect labels caused by this reason. Another reason is non-essential changes [23] such as deleting dead code affect a large number of lines of code, while they are less important than other modification tasks such as feature enhancement. We can conjecture some cases that source code is decreased, but P3 was at most 17% (6 of 36 in Dataset 7) of extracted edges in our experiment. Hence, our method for determining the direction still worked effectively. We did not use release dates since they are not always available, but if release dates are available, all evolution direction would be correctly extracted if edges connect successive products.

Table 4 Incorrect edge patterns with N_w.

Dataset	P1	P2	P3	P4	P5	Total
1						0
2		4	5		2	11
3		5			2	7
4		4				4
5		1				1
6		2		4	2	8
7	1	5	6		1	12
8	4	3	6			14
9	2	6				8

Releases with no changes invoke error pattern P1 and P3. Developers easily notice this is an error, since it is hard to think that some files are fixed but total amount of deleted and added code are the same amount.

The optimization reduces the execution time greatly. Dataset 1 for example, we need 10 minutes for analysis using optimization. On the other hand, without optimization, our tool runs over an hour for analyzing first four products.

5. Case Study

The result of experiment shows that our method well approximates an evolution history of software product from their source code with high precision. In the case study, we simulate the situation that finding out the origin of the variants. We continued using similarity threshold $th = 0.9$.

The target is the Linux kernel and two of their variants. One variant is in the kernel repository, labeled "latest", and another variant is kernel files from F-05D Android smartphone[†]. We analyze those two variants with releases of the Linux kernel and check the result with the version number denoted in the Makefile.

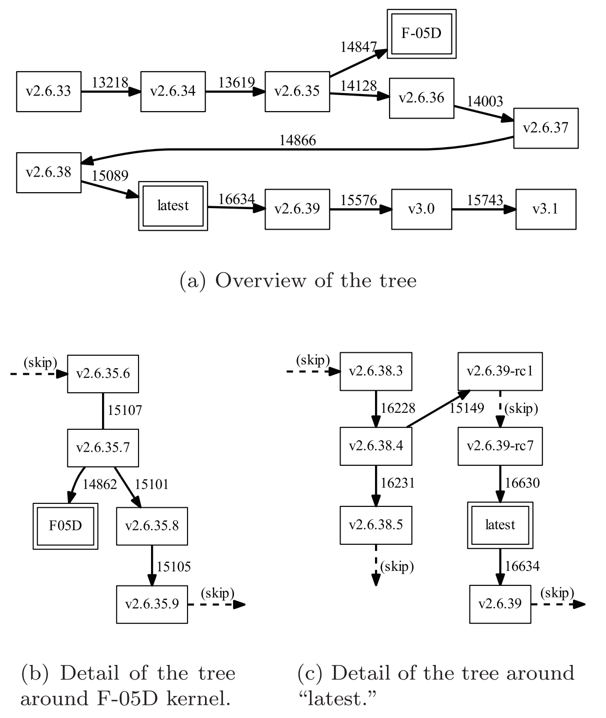
Figure 6 (a) shows the overview of the Product Evolution Tree and Fig. 6 (b) and Fig. 6 (c) shows the detail of the tree around target variants. Those figures show that the F-05D kernel was branched from 2.6.35.7 and latest tag is attached just before 2.6.39 is released. We can see that those two variants have different history. F-05D kernel was branched and they have had some changes. "latest" tag is assigned for development of 2.6.39 but there are still some changes before 2.6.39 is released.

This result matches the version number denoted in the Makefile and its product history. Makefile of F-05D says that this is 2.6.25.7, and "latest" is tagged between 2.6.39-RC7 and 2.6.39 in the repository. The result of the case study shows that our approach is useful for detecting origin of the variants. With the Product Evolution Tree, we can see that which product is the origin and whether the product is branched or not.

6. Threats to Validity

Targets of our experiment are restricted in the OSS with ver-

[†]http://spf.fmworld.net/oss/oss/f-05d/



(a) Overview of the tree

(b) Detail of the tree around F-05D kernel.

(c) Detail of the tree around "latest."

Fig. 6 A case study with Linux kernel and two variants.

sion control system and they have reliable their evolution history. In other words, those projects are considered well maintained. However, our Product Evolution Tree well reflects the development history compared with actual history in some cases. For example, branched timing in the tree follows functional changes in Dataset 2, and we could find completely same versions with different tags in Dataset 7, 8, and 9.

We have used a single threshold 0.9 in the case study, which is determined by a small preliminary experiment. While it works for 9 datasets, a different threshold may be better for a different dataset.

7. Conclusions

To help developers understand the evolution history of products, we proposed a method to extract an approximation of the evolution history from source code. It is defined as a tree that connects most similar file pairs. Specifically, we count the number of similar files with Longest Common Subsequence based source similarity and we construct a spanning tree of complete graph which connects all input products.

As a result, 47% to 100% of edges are correctly recovered. We can identify branches and the latest versions of products using our approach, even if the result included incorrect edges. Our methodology and techniques used are simple, but shows promising result in experiments.

Acknowledgements

This work is supported by Kakenhi kiban (S), No.25220003, and Osaka University Program for Promoting International Joint Research, "Software License Evolution Analysis".

References

[1] J. Rubin, A. Kirshin, G. Botterweck, and M. Chechik, "Managing forked product variants," Proc. SPLC, pp.156–160, 2012.
[2] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki, "An exploratory study of cloning in industrial software product lines," Proc. CSMR, pp.25–34, 2013.
[3] D. Faust and C. Verhoef, "Software product line migration and deployment," Softw. Pract. Exp., vol.33, pp.933–955, 2003.
[4] R. Koschke, P. Frenzel, A. Breu, and K. Angstmann, "Extending the reflexion method for consolidating software variants into product lines," Softw. Quality J., vol.17, pp.331–366, 2009.
[5] K. Yoshimura, F. Narisawa, K. Hashimoto, and T. Kikuno, "FAVE: Factor analysis based approach for detecting product line variability from change history," Proc. MSR, pp.11–18, 2008.
[6] C.W. Krueger, "Easing the transition to software mass customization," Revised Papers from PFE, pp.282–293, 2001.
[7] T. Lavoie, F. Khomh, E. Merlo, and Y. Zou, "Inferring repository file structure modifications using nearest-neighbor clone detection," Proc. WCRE, pp.325–334, 2012.
[8] D.L. Parnas, "Software aging," Proc. ICSE, pp.279–287, 1994.
[9] T. Kanda, T. Ishio, and K. Inoue, "Extraction of product evolution tree from source code of product variants," Proc. SPLC, pp.141–150, 2013.
[10] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code," IEEE Trans. Softw. Eng., vol.28, no.7, pp.654–670, 2002.
[11] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: Finding copy-paste and related bugs in large-scale software code," IEEE Trans. Softw. Eng., vol.32, pp.176–192, 2006.
[12] A. Hemel and R. Koschke, "Reverse engineering variability in source code using clone detection: A case study for linux variants of consumer electronic devices," Proc. WCRE, pp.357–366, 2012.
[13] K. Yoshimura and R. Mibe, "Visualizing code clone outbreak: An industrial case study," Proc. IWSC, pp.96–97, 2012.
[14] K. Inoue, Y. Sasaki, P. Xia, and Y. Manabe, "Where does this code come from and where does it go? – Integrated code history tracker for open source systems –," Proc. ICSE, pp.331–341, 2012.
[15] M. Godfrey and L. Zou, "Using origin analysis to detect merging and splitting of source code entities," IEEE Trans. Softw. Eng., vol.31, no.2, pp.166–181, 2005.
[16] Lucia, F. Thung, D. Lo, and L. Jiang, "Are faults localizable?," Proc. MSR, pp.74–77, 2012.
[17] T. Yamamoto, M. Matsushita, T. Kamiya, and K. Inoue, "Measuring similarity of large software systems based on source code correspondence," Proc. PROFES, pp.530–544, 2005.
[18] V. Tenev and S. Duszynski, "Applying bioinformatics in the analysis of software variants," Proc. ICPC, pp.259–260, 2012.
[19] S. Kawaguchi, P.K. Garg, M. Matsushita, and K. Inoue, "MUDABlue: An automatic categorization system for open source repositories," J. Syst. Softw., vol.79, no.7, pp.939–953, 2006.
[20] K. Tian, M. Revelle, and D. Poshyvanyk, "Using latent dirichlet allocation for automatic categorization of software," Proc. MSR, pp.163–166, 2009.
[21] C. McMillan, M. Grechanik, and D. Poshyvanyk, "Detecting similar software applications," Proc. ICSE, pp.364–374, 2012.
[22] P. Weißgerber and S. Diehl, "Identifying refactorings from source-code changes," Proc. ASE, pp.231–240, 2006.
[23] D. Kawrykow and M.P. Robillard, "Non-essential changes in version histories," Proc. ICSE, pp.351–360, 2011.

Appendix A: Dataset

Table A· 1 Datasets.

#	Name	Language	Included versions/tags	#product	#file	#LOC
1	Pgsql-major	C	PostgreSQL: 7.0, 7.1, 7.2, 7.3, 7.4, 8.0.0, 8.1.0, 8.2.0, 8.3.0, 8.4.0, 9.0.0, 9.1.0, 9.2.0, 9.3.0	14	9,451	4,680,600
2	Pgsql8-all	C	PostgreSQL: 8.0BETA1 – 8.0BETA5, 8.0RC1 – 8.0RC5, 8.0.0 – 8.0.26, 8.1BETA1 – 8.1BETA4, 8.1RC1, 8.1.0 – 8.1.23, 8.2BETA1 – 8.2BETA3, 8.2RC1, 8.2.0 – 8.0.23, 8.3BETA1 – 8.3BETA4, 8.3RC1 – 8.3RC2, 8.3.0 – 8.3.21, 8.4BETA1 – 8.4BETA2, 8.4RC1 – 8.4RC2, 8.4.0 – 8.4.14, 8.5ALPHA1 – 8.5ALPHA3	144	96,448	48,478,395
3	Pgsql8-latest	C	PostgreSQL: 8.0.20 – 8.0.26, 8.1.17 – 8.1.23, 8.2.17 – 8.2.23, 8.3.15 – 8.3.21, 8.4.8 – 8.4.14, 8.5ALPHA1 – 8.5ALPHA3	38	26,232	13,401,899
4	Pgsql8-annually	C	PostgreSQL: 8.0.4, 8.0.9, 8.0.14, 8.0.18, 8.0.22, 8.0.26, 8.1.5, 8.1.10, 8.1.14, 8.1.18, 8.1.22, 8.2.5, 8.2.10, 8.2.14, 8.2.18, 8.2.22, 8.3.4, 8.3.8, 8.3.12, 8.3.16, 8.3.21, 8.4.1, 8.4.5, 8.4.9, 8.4.14	25	16,816	8,488,128
5	FFmpeg	C	FFmpeg (before fork): v0.5 – v0.5.3 FFmpeg (after fork): n0.5.5 – n0.5.10 LibAV: v0.5.4 – v0.5.9	16	9,872	3,952,273
6	*-BSD	C	BSD: 4.4BSD Lite, 4.4BSD Lite2 FreeBSD: 2.0, 2.0.5, 2.1, 2.2, 2.3 NetBSD: 0.8, 0.9, 1.0, 1.1, 1.2, 1.2.1, 1.3 OpenBSD: 2.0, 2.1	16	16,204	6,050,462
7	Groovy	Java	Groovy: 2.0.0BETA2 – 2.0.0BETA3, 2.0.0RC1 – 2.0.0RC4, 2.0.0 – 2.0.8, 2.1.0BETA1, 2.1.0RC1 – 2.1.0RC3, 2.1.0 – 2.1.9, 2.0.0BETA1, 2.2.0RC1 – 2.2.0RC3, 2.2.0 – 2.2.2, 2.3.0BETA1 – 2.3.0BETA2	37	34,797	3,962,603
8	Hibernate	Java	Hibernate: 4.0.0Alpha1 – 4.0.0Alpha3, 4.0.0Beta1 – 4.0.0Beta5, 4.0.0CR1 – 4.0.0CR7, 4.0.0Final, 4.0.1, 4.1.0Final, 4.1.1 –4.1.2, 4.1.2Final – 4.1.12Final, 4.1.5SP1, 4.2.0CR1 – 4.2.0CR2, 4.2.0Final – 4.2.12Final, 4.2.0SP1, 4.2.7SP1, 4.3.0Beta1 – 4.3.0Beta5, 4.3.0CR1 – 4.3.0CR2, 4.3.0Final – 4.3.5Final	62	271,372	19,767,324
9	OpenJDK6	Java	OpenJDK6: b00 – b15	16	112,922	19,858,640

Tetsuya Kanda received the master's degree in information science and technology from Osaka University in 2013. He is a Ph.D. student at Osaka University. His research interests are software evolution and source code analysis. He is a member of the IPSJ and IEEE.

Takashi Ishio received the Ph.D. degree in information science and technology from Osaka University in 2006. He was a JSPS Research Fellow from 2006–2007. He is now an assistant professor of computer science at Osaka University. His research interests include program analysis and program comprehension. He is a member of the IEICE, IPSJ, JSSST, IEEE, and ACM.

Katsuro Inoue received the B.E., M.E., and D.E. degrees in information and computer sciences from Osaka University, Japan, in 1979, 1981, and 1984, respectively. He was an assistant professor at the University of Hawaii at Manoa from 1984–1986. He was a research associate at Osaka University from 1984–1989, an assistant professor from 1989–1995, and a professor beginning in 1995. His interests are in various topics of software engineering such as software process modeling, program analysis, and software development environment. He is a member of the IEEE, the IEEE Computer Society, and the ACM.