Extraction of Product Evolution Tree from Source Code of Product Variants

Tetsuya Kanda, Takashi Ishio, Katsuro Inoue
Graduate School of Information Science and Technology, Osaka University
1-5 Yamadaoka, Suita, Osaka 565-0871, Japan
{t-kanda, ishio, inoue}@ist.osaka-u.ac.jp

ABSTRACT

A large number of software products may be derived from an original single product. Although software product line engineering is advocated as an effective approach to maintaining such a family of products, re-engineering existing products requires developers to understand the evolution history of the products. This can be challenging because developers typically only have access to product source code. In this research, we propose to extract a Product Evolution Tree that approximates the evolution history from source code of products. Our key idea is that two successive products are the most similar to one another in the evolution history. We construct a Product Evolution Tree as a minimum spanning tree whose cost function is defined by the number of similar files between products. As an experiment, we extracted Product Evolution Trees from 6 datasets of open-source projects. The result showed that 53% to 92% of edges in the extracted trees were consistent with the actual evolution history of the projects.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms

Experimentation

Keywords

software product line; software evolution; visualization

1. INTRODUCTION

Copying existing code fragments, source files, and an entire project is a common practice when developing a new software product [20]. For example, the Linux kernel is forked into many projects extending beyond Linux distributions including embedded software such as the Android OS. Nonaka *et al.* analyzed corrective maintenance data of industrial embedded software products [17]. A part of the evolution history of the software product family is shown in Figure 1. The horizontal axis represents the number of months from the first release of the original product series (P01), and the vertical axis represents the product series ID in a company. In Figure 1, a circle corresponds to a product. Each dashed edge indicates that the new product series is derived from the original product. A solid edge connecting products indicates that the products are released as different versions of the same product series. Figure shows only 8 major product series and their variations, while the company had 25 series of products. Each series of products has 2 to 42 versions. Although Software Product Line Engineering (SPLE) is a well-known approach to efficient maintenance of a software product family, the industry already maintains a large number of derived software products without applying SPLE. The construction of a software product line from existing products is a major problem and many re-engineering methods have been proposed as solutions [2, 11, 26].

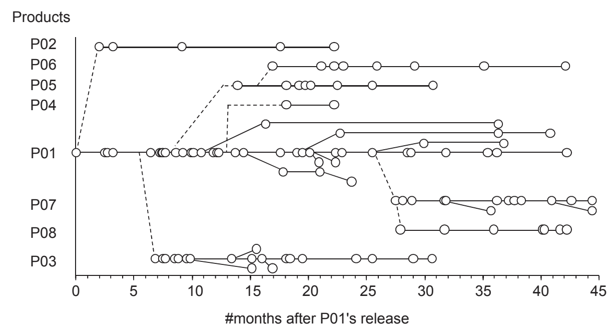


Figure 1: A product family derived from a single product [17].

To construct a software product line, developers must analyze and compare software products to identify commonality and variability between them. Since analyzing a large number of software products is a difficult task for developers, Krueger suggested that developers should begin their analysis with a small number of software products, instead of all products at once [12]. Koschke *et al.* proposed an extension of the reflexion method to construct a product line by incrementally analyzing products [11]. To follow these reasonable approaches, developers must choose representative software products as a starting point. If an evolution history of software products such as Figure 1 were available, devel-

SPLC 2013 August 26 - 30 2013, Tokyo, Japan

opers could recognize the relationships among the products and choose representatives for their analysis. For example, developers could analyze the original product and the latest release on each branch. The selection would enable developers to identify core features and product specific features of the product family. However, such a history of products is often not available for developers [13]. In the worst cases, developers only have access to source code of each product.

In this research, we propose to extract an approximate the evolution history of software products from their source code. Our approach depends on only source code so that we can analyze products without version numbers, names or release dates. We define a *Product Evolution Tree* as a labeled tree whose nodes each represent a product, each edge connects similar products, and each label indicates the similarity of products and the direction of evolution, respectively. A Product Evolution Tree is computed as a minimum spanning tree. Its cost function is defined by the number of similar files between products. Similarity between files is computed by Yoshimura's function, which is based on the longest common subsequence of the files [25].

We implemented our approach as a tool that takes source code as input and visualizes a Product Evolution Tree. Using the tool, we conducted a case study with six datasets based on open-source projects. Whereas our approach is a simple algorithm, the results show that 53% to 92% of edges (79% on average) were consistent with the actual evolution history.

Our contributions are summarized as follows:

- We propose a visualization technique of relationships among software products from their source code.
- Our tool and datasets are publicly available [27] so that other researchers can replicate and improve the approach.

In Section 2, we describe related work. Section 3 details our proposed approach. Section 4 shows the result of a case study. We discuss the results in Section 5. Finally, in Section 6, we present the conclusion and future work.

2. RELATED WORK

2.1 Product Analysis

To apply SPLE to existing similar software products, developers must analyze features in the products. Kastner *et al.* proposed CIDE to simplify software product line development [8]. CIDE requires a single software product and decomposes it into features. Duszynski *et al.* proposed a technique for analyzing multiple software system variants [1]. They extracted system structure models and variability models that represent commonality and variability of system variants from source code. Their technique allows for detailed goal-driven refinement of the analysis results.

To utilize these techniques for a large number of software products, developers must choose one or more software products for their analysis. Our approach enables developers to choose a starting point for their analysis by visualizing relationships among software products.

2.2 Software Categorization

Several tools have been proposed to automatically categorize large volumes of software based on domains such as compiler, database, and editor. MUDABlue [9] classifies software based on the similarity of identifiers in source code. It employs latent semantic analysis which extracts the contextual-usage meaning of words by statistical computations. LACT [22] uses latent Dirichlet allocation in which software can be viewed as a mixture of topics and utilizes identifiers and code comments, but excludes literals and programming language keywords to improve categorization. CLAN [16] focuses on API calls with the basic idea that similar software uses the same set of APIs. While all of these tools are able to detect similar or related applications from a large software set, our approach focuses on very similar products derived from the same product, which are likely to be grouped into the same category by these tools.

2.3 Software Evolution

Yamamoto *et al.* proposed a tool SMAT that calculates the similarity of software systems by counting similar lines of source code [24]. The tool identifies corresponding source files between two software systems using CCFinder [7], and then computes differences between file pairs. They applied the tool to a case study of software clustering, and extracted a dendrogram of BSD-Unix operation systems. The dendrogram reported which operating systems were similar to each other.

Tenev *et al.* introduced bioinformatics concepts into software variants analysis [21]. One such concept is the phylogenetic tree, which visualizes similarity relations. They constructed a dendrogram and a cladogram from six BSD-Unix family systems as an example of phylogenetic trees.

Although their approaches and goals are similar to our idea, our approach visualizes more concrete relationships among products: which product was first released, which products were forked from the release, and so on.

2.4 File-to-File Similarity

The comparison of two or more software systems with one another has been the subject of much research. When comparing software systems, the similarity between source files is a very important metric. Many code clone detection tools have been proposed to identify source code fragments that are identical or similar [7, 14]. Using large-scale code clone detection techniques, Hemel and Koschke compared the Linux kernel with its vendor variants [5]. They found the vendor variants included various patches, but the patches were rarely submitted to the upstream. Another application of code clone detection is detecting file moves that take place between released versions of a software system [13].

Yoshimura *et al.* visualized cloned files in industrial products by using an edit distance function as a source file similarity to identify cloned files with nearly the same contents [25]. We employed their similarity function with an optimization and aggregated file similarity to product similarity.

Inoue *et al.* [6] proposed a tool called Ichi Tracker to investigate a history of a code fragment using source code search engines. The tool takes a code fragment as an input and extracts related code from source code search engines. It visualizes how related files are similar to the original code fragment and when they are released. Using the visualization, developers can identify the origin of the source code fragment or a more improved version of the code fragment. Our approach enables similar analysis on software products instead of source files.

We assume that two successive products are very similar to each other. This observation is shown by Godfrey *et al.* [3], who detected merging and splitting of functions between two versions of a software system. Their analysis showed that a small number of software entities such as functions, classes, or files are changed between two successive versions. Furthermore, Lucia *et al.* reported that most bug fixes were implemented in a small number of lines of code [15]. Since these analyses reported that two successive versions of software were very similar, we infer that the most similar pairs of products are likely to be two successive versions. Although developers may modify a substantial number of lines of code to release a new version, this new version is likely more similar to the original version than future products derived from the new version.

3. PRODUCT EVOLUTION TREE

A Product Evolution Tree is a tree that approximates evolution history. Each node of a tree represents a software product and each edge indicates that a product is likely derived from another product. An edge label explains the cost of software changes between products and the direction of derivation, indicating which product is an ancestor and which product is a successor. We construct a Product Evolution Tree from product source code through following three steps:

1. We calculate file-to-file similarity for all source file pairs of all products.
2. We construct a minimum-spanning tree of products. The cost between two products is based on the number of similar files between the products.
3. We assign labels to edges based on the number of modified tokens between two products.

3.1 File Similarity Calculation

To calculate file similarity, we first normalize each source file into a sequence of tokens. In a normalized file, each line has only a single token. We remove blank space from source files to avoid an impact from coding style. We also remove comments since they do not affect the behavior of products. All other tokens including keywords, macros, and identifiers retained in their original state.

We calculate the similarity for all pairs of files across different products, since a file may be renamed in a different product variant. To calculate similarity among source files, we follow Yoshimura's inter-file similarity analysis [25]. Given a pair of files (a, b), their file similarity $sim(a, b)$ is calculated using the normalized sequences a_t and b_t of the files as follows:

$$sim(a, b) = \frac{LCS(a_t, b_t)}{LCS(a_t, b_t) + ADD(a_t, b_t) + DEL(a_t, b_t)}$$

where $LCS(a_t, b_t)$ is the number of tokens in the longest common subsequence (LCS) between a_t and b_t. A pair of $ADD(a_t, b_t)$ and $DEL(a_t, b_t)$ represents an edit distance from a_t to b_t. $DEL(a_t, b_t)$ is the number of deleted tokens unique to a_t, and $ADD(a_t, b_t)$ is the number of added tokens unique to b_t. ADD and DEL can be represented as follows.

$$ADD(a_t, b_t) = LENGTH(b_t) - LCS(a_t, b_t)$$
$$DEL(a_t, b_t) = LENGTH(a_t) - LCS(a_t, b_t)$$

where $LENGTH(s)$ is the number of tokens in s.

We used a file similarity based on LCS, since we could optimize the calculation as described in Section 3.4. The following computation steps do not depend on the definition of the file similarity function; hence, other methods such as code clone detection are also applicable to compute file similarity.

3.2 Construction of the Minimum Spanning Tree

In this step, we construct a minimum spanning tree of products. To define a cost function for products, we count the number of similar file pairs. When the file pair has a higher similarity than a threshold value, it is considered as a similar file pair. A similarity threshold $th = 0.9$ was experimentally determined and is used in this paper. The number of all possible similar file pairs E_s and cost C between software products P_1 and P_2 are defined as:

$$\begin{aligned} E_s(P_1, P_2, th) &= \{(a, b) \mid a \in P_1, b \in P_2, sim(a, b) \geq th\} \\ C(P_1, P_2, th) &= -|E_s(P_1, P_2, th)|. \end{aligned}$$

It should be noted that cost is a negative value, as the cost decreases if products have more similar file pairs.

After calculating cost, the result is an undirected weighted graph $G = (V, E)$. V denotes the software products and E connects them with the cost function C. We construct a minimum spanning tree $S = (V, E')$ of the graph G. $E' \subset E$ is a set of edges which have the smallest total cost:

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

We use Prim's algorithm [18]. In Prim's algorithm, an initial vertex is picked up at first. The vertex is a tree comprising of a single vertex. Next, the algorithm lists out all edges connecting a vertex in the tree to a vertex outside of the tree, selects the edge with the lowest cost, and includes the edge and its connected vertex in the tree. The process is repeated until all vertices are included in the tree.

Prim's algorithm allows any vertex as a starting point. In addition, if two or more edges have the same lowest cost, one of them can be arbitrarily selected. In our implementation, we select a vertex or an edge depending on the input order.

3.3 Evolution Direction Calculation

After a minimum spanning tree is constructed, we assign labels to the edges. We hypothesize that source code is likely added rather than deleted when software evolves into its next version. To compute the relationship, we first define two functions for two products P_1 and P_2 as follows:

$$ADD_{ALL}(P_1, P_2) = \sum_{(a,b) \in E_s(P_1, P_2, th)} ADD(a, b)$$

$$DEL_{ALL}(P_1, P_2) = \sum_{(a,b) \in E_s(P_1, P_2, th)} DEL(a, b)$$

Both functions approximate the total number of modified tokens in similar files between the products. We determine a direction label for an edge between products using the following conditions:

$$\begin{cases} P_1 \rightarrow P_2, & ADD_{ALL}(P_1, P_2) > DEL_{ALL}(P_1, P_2) \\ P_1 = P_2, & ADD_{ALL}(P_1, P_2) = DEL_{ALL}(P_1, P_2) \\ P_1 \leftarrow P_2, & ADD_{ALL}(P_1, P_2) < DEL_{ALL}(P_1, P_2) \end{cases}$$

It is easy to determine the amount of source code that has been changed because we calculated how many tokens are added or deleted from one source file to another through the file similarity calculation.

3.4 File Similarity Optimization

A naive computation of sim for N files requires the computation of LCS $N(N-1)/2$ times. To reduce the computation time, we introduced an optimization that calculates the sim value only if it is necessary.

The LCS of two files comprises only of tokens included in both files. To estimate the length of LCS between files, we introduce the term frequency $tf(f,t)$ which represents how many times a term t appears in a file f. For example, consider two files S1(AAABB) and S2(ABBBB), where A and B are terms in the files. The term frequencies are $tf(A,S1) = 3$, $tf(B,S1) = 2$, $tf(A,S2) = 1$ and $tf(B,S2) = 4$. Since the LCS can include at most one A and two Bs shared by the sequences, the maximum length of the LCS is 3. Indeed, the actual LCS between S1 and S2 is ABB whose length is 3. Comparing S1(AAABB) with S3(BBBAB) for another example, this pair also shares one A and two Bs, but there is no LCS with a length of 3. AB, BA, and BB are possible LCSs for them.

With term frequency, we can get maximum similarity

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

of each file pair (a,b). T represents the set of terms appearing in all source files. The value of $sim(a,b)$ is equal to $msim(a,b)$ if all the common tokens appear in the same order in two sequences. If the order of tokens in a sequence is different from another sequence, then $sim(a,b)$ is smaller than $msim(a,b)$. A formula $msim(a,b) \geq sim(a,b)$ is always true, hence we need to compute $sim(a,b)$ only if $msim(a,b) \geq th$. Although we have to scan a file to construct a term frequency vector, the vector is used $N-1$ times. In addition, the time complexity of $msim$ is $\mathcal{O}(|T|)$, which is much smaller than the LCS calculation.

4. CASE STUDY

We implemented our approach as a tool and conducted a case study. The goal of the case study is to evaluate how accurately a Product Evolution Tree recovers actual evolution history.

4.1 Datasets

We prepared six datasets using open-source projects. Each dataset comprises a set of products whose evolution history is publicly available. Table 1 shows the lists of products in the datasets. Column "#" indicates the ID of the dataset of each row. The other columns show the name of the dataset, products included in the dataset, the number of products, the total number of files and the total number of lines of code. Due to the limited space, we have omitted the intermediate version numbers in the table. For example, "8.0.0 – 8.0.26" indicates that the dataset included 27 version from 8.0.0 to 8.0.26. All datasets and the results are publicly available on our website [27].

In the datasets, we used the following software.

PostgreSQL. This 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 was 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 the "src" directory in this case study.

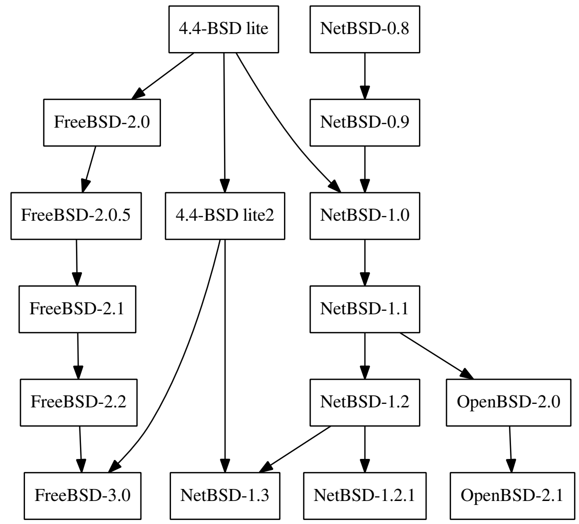


Figure 2: The family-tree of Dataset 6

FFmpeg and Libav. These 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 the source code of both programs.

4.4BSD, FreeBSD, NetBSD, and OpenBSD. These operating systems are derived from 4.3BSD, but are now independent projects. The evolution history of this project family is publicly available as a "family-tree." Figure 2 shows a section of the family-tree for the versions included in our dataset. According to the tree, NetBSD is originally derived from 4.3BSD, but NetBSD-1.0 is derived from 4.4BSD Lite. FreeBSD-2.0 is also based on 4.4BSD Lite. OpenBSD is forked from NetBSD. OpenBSD-2.0 is its first official release. 4.4BSD Lite2 is the last release of 4.4BSD and it affects other BSD operating systems. In each version of distributed files, we used source files under the "src/sys" directory.

Each dataset represents a particular situation of product analysis as follows.

Dataset 1: Pgsql-major.

This dataset has a straight evolution history, *i.e.*, it has no project fork. The dataset contains 13 versions that are the initial versions of each major release. We found that all of these releases were developed in the master branch. Hence, the resultant Product Evolution Tree should form a straight line.

Table 1: Datasets

#	Name	Included versions/tags	#product	#file	#LOC
1	Pgsql-major	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	13	8,533	4,163,127
2	Pgsql8-all	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	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	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	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	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

Dataset 2: Pgsql8-all.

The evolution history of this dataset is a tree of a single project. We created it to emulate a practical case; a large number of products are derived from a single original product. This dataset contains six branches: five STABLE branches for versions 8.0.X to 8.4.X, and the master branch developing into three ALPHA releases for 8.5. It should be noted that these branches were developed in parallel. For example, some products were released from the 8.0.X branch after 8.1.0. The dataset contains 144 versions in total.

Dataset 3: Pgsql8-latest.

This dataset includes only recent products. If a product family has a long history, older products may be too obsolete to be included in a product line. In addition, such older products may be no longer available to developers. The dataset is a subset of Dataset 2. It contains 38 versions including 7 latest versions in each of 5 STABLE branches and 3 releases in the 8.5ALPHA series. This dataset contains no older releases indicating how the STABLE branches have been created. Therefore, it is difficult to extract the relationship among branches.

Dataset 4: Pgsql8-annually.

This is another dataset where a full collection of products is not available. Dataset 4 contains 25 versions that were released around every September from 2005 to 2012. Extracting the relationship among branches is difficult for the dataset, since it does not contain any major releases.

Dataset 5: FFmpeg.

This dataset consists of a project that has been forked to two projects. It was created to evaluate whether our approach could recover the evolution history of forked projects. The dataset contains the FFmpeg 0.5 series from 0.5 to 0.5.3 before the fork and from 0.5.5 to 0.5.10 after the fork. 0.5.4 is not included since the tag was not available in the repository. In addition, Libav 0.5.4 to 0.5.9 are included in the dataset.

Dataset 6: BSD.

This dataset consists of a project that has been forked to more than three projects. It was created to evaluate whether our approach could recover the complex evolution history of the projects or not. The evolution history of BSD operating systems is the most complex among our datasets. In addition, there are releases created by merging source code from more than one product that were derived from their single origin. Since our approach only extracts a tree, our approach must miss such edges.

4.2 Results

Our tool takes source code in a dataset as input and outputs a Product Evolution Tree. For each edge in a Product Evolution Tree, we checked whether the edge connected versions in the parallel with the evolution history of the dataset, and then checked the labeled direction for the matched edges.

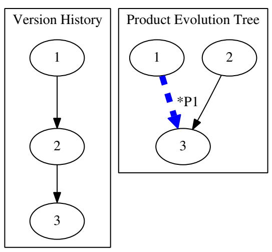
The correctness of the edges and labels is shown in Table 2. The "Matched Edges" column shows how many edges were matched with the actual history of the dataset without taking direction into consideration. In other words, we only checked the shape of the tree. The "Matched Labels" column shows how many correct edges also had correct direction. The "Recall" column indicates the proportion of correctly identified edges to edges in the actual evolution history of each dataset. We did not calculate precision in this experiment, since precision and recall are either the same (Datasets 1–5) or a higher value (Dataset 6). This is because the number of edges in the Product Evolution Tree is the same as or fewer than the number of edges in the actual evolution history.

The results show that 53% to 92% of edges were consistent with the actual evolution history. This is a very promising result. However, one important concern is what types of errors were included in the Product Evolution Tree. Since developers are unaware of a product's actual evolution history, an incorrect edge may lead developers to develop a false understanding of the product family. To analyze errors, we categorized incorrect edges in Product Evolution Trees into five patterns that are described below and shown in Figure 3. In Figure 3, each left graph shows an actual evolution history and each right graph shows an extracted

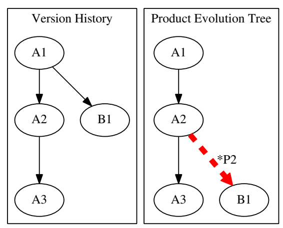
Table 2: Results

Dataset	History	Output	Matched Edges		Matched Labels		Recall
1 †	12	12	12	(100%)	11	(91.7%)	91.7%
2 (Fig. 4)	143	143	136	(95.1%)	128	(94.1%)	89.5%
3 †	37	37	30	(81.1%)	30	(100%)	81.1%
4 (Fig. 5)	24	24	20	(83.3%)	20	(100%)	83.3%
5 (Fig. 6)	15	15	13	(86.7%)	11	(84.6%)	73.3%
6 (Fig. 7)	17	15	12	(70.6%)	9	(75.0%)	52.9%

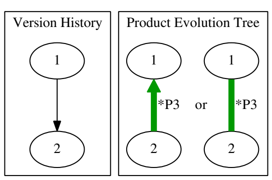
†Figures are available on our website [27].



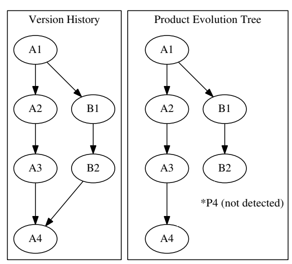
(a) P1: Version Skip



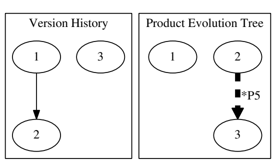
(b) P2: Misalignment of Branch



(c) P3: Misdirection



(d) P4: Missing Merge



(e) P5: Out of Place

Figure 3: Patterns of incorrect edges

Product Evolution Tree. Thin edges are the connections that exist in the actual history, thick edges represent the errors; thick solid edges connect exact products but indicate a wrong direction, and thick dashed edges 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 occurs when v_2 and v_3 have the same change cost from v_1 or the change cost between v_1 and v_3 are very small. For example, a small bug fix between v_2 and v_3 that modifies a few lines of code added for v_2 can cause this pattern to appear.

P2: Misalignment of Branch. An edge connects two branches but does not connect products that are actually branched. In Figure 3(b), there are two branches A and B. While version 1 of branch B (B1) was actually forked from version 1 of branch A (A1) in the evolution history, the origin of branch B was instead recognized as version 2 of branch A (A2). In this pattern, A2 is actually more similar to B1 than A1, since both A2 and B1 include the same changes from A1, such as bug fixes.

P3: Misdirection. An edge connects products accurately, but its label shows the reverse direction as shown in Figure 3(c). This occurs when the size of source code or the number of source files decreases due to several factors such as refactoring and the deletion of dead code.

P4: Missing Merge. Our Product Evolution Tree cannot detect a merge of two products derived from a single product. In Figure 3(d), we can see that the Product Evolution Tree detects branching from version A1 to version A2 and B1, but cannot detect merging from version B2 to A4. When a software product forks into other software products, they are usually very similar. Since the forked products are independently modified, they would exhibit an increased level of differences. Our approach connects similar products first, and a tree cannot have a closed path. Therefore, edges indicating project fork often appear in the tree but merges are hardly detected. 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.

P5: Out of Place. This pattern falsely detects edge that is not classified into previous patterns and there are no relationships between this false edge and the actual history.

In these patterns, P1, P2, and P3 connected related products. P1 could be corrected by analyzing three related products. P2 is not easily corrected, but it still connects two relevant branches. Therefore, developers might misunderstand only information concerning at what point the actual branching occurred. P3 could be corrected by manually comparing the edge direction with other edges around the products, and investigating difference between source code of the connected products. Therefore, these patterns are not too problematic. P4 is a missing edge and is not correctable. Our future work includes detecting merges. P5 is the most problematic error among the patterns, as it connects irrelevant products.

Table 3 shows the number of pattern instances of incorrect edges. It indicates a small number of problematic errors

Table 3: The number of instances of incorrect edge patterns

Dataset	P1	P2	P3	P4	P5	Total
1			1			1
2	1	4	8		2	15
3		5			2	7
4		4				4
5	1	1	2			4
6		2	3	2	1	8

found in the datasets, though the more minor errors of P2 and P3 occur rather frequently.

4.3 Product Evolution Trees

In this section, we show the details of each Product Evolution Tree extracted from the datasets. We compare a Product Evolution Tree with its actual history and then colored incorrect edges as indicated in Figure 3.

Dataset 1: Pgsql-major.

The form of the tree was perfectly matched with the actual evolution history. However, one label indicated the reverse direction on the edge 9.1.0–9.2.0. Among these releases, more source code was added than deleted, since several identifiers were renamed and some refactoring was performed. Despite the existence of the directional error, this Product Evolution Tree is still useful, as it is still capable, of identifying the latest version and the oldest version of the software product.

Dataset 2: Pgsql8-all.

An overview of the extracted Product Evolution Tree for Dataset2 is shown in Figure 4. Since the tree is too large to show in the paper, a sequence of correctly connected versions is indicated by a single node. For example, the top-left node in Figure 4 represents 14 versions: 8.0.0BETA1 to 8.0.0BETA5, 8.0.0RC1 to 8.0.0RC5, and 8.0.0 to 8.0.3. In the figure, we annotated only incorrect edges and labels. The full version of this figure is also available on our website.

The Product Evolution Tree for Dataset 2 recovered its actual evolution history with a high recall percentage. However, almost no edges connecting branches were matched. For example, 8.2BETA1 was developed on the master branch as the next version of 8.1.0. In the extracted tree, 8.2BETA1 is indicated as the next version of 8.1.5. We examined git repository and found that version 8.1.5 was released right after 8.2BETA1. The master branch developing 8.2BETA1 and STABLE branch for 8.1 received the same 225 commits that were submitted on the same date with the same log message. During the same period (8.1.0 to 8.2BETA1), the differences between two branches only consisted of 28 commits unique in comparison to the master branch. This fact means that even the actual evolution history does not always show functional differences of products.

Although the Product Evolution Tree is not perfect, we can identify six branches in the product set, and we can pick up the latest versions for each branch: 8.0.26, 8.1.23, 8.2.23, 8.3.21, 8.4.14 and 8.5ALPHA3. There are several labels in the branches indicating the incorrect direction, but a majority of edges indicated the correct evolution in the branches. Validating the direction of 143 edges of the tree requires a much smaller effort than comparing arbitrary pairs of 144 products ($144 \times 143/2 = 10296$ pairs).

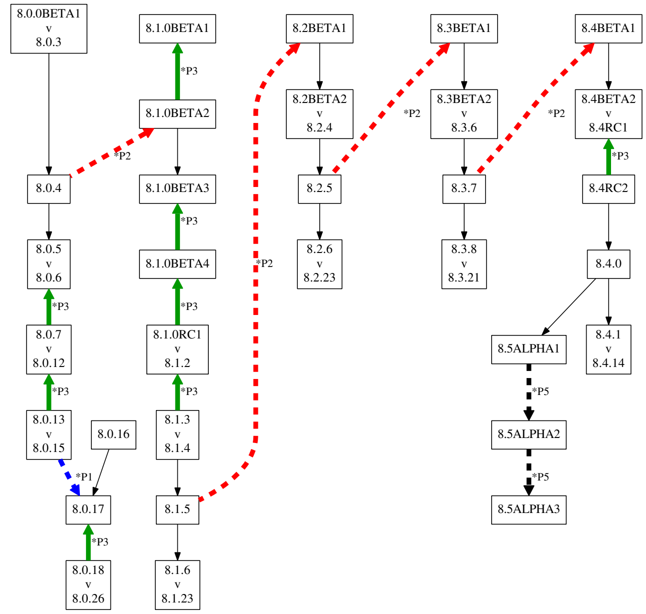


Figure 4: Product Evolution Tree for Dataset 2

Dataset 3: Pgsql8-latest.

The Product Evolution Tree of this dataset is almost the same as that of Dataset 2, with the exception of the edges connecting branches. Two STABLE branches often include the same changes, while only minor changes are unique to one of the branches. Therefore, intermediate versions of two branches were connected to each other (P2 instances). Nevertheless, the Product Evolution Tree shows the initial and latest versions of each branch, since there were a few error edges outside of the ones between branches.

Dataset 4: Pgsql8-annually.

Figure 5 shows the Product Evolution Tree extracted from Dataset 4. Products are arranged horizontally if they are released on the same day. All edges and labels in the same branch are consistent with the actual evolution history. On the other hand, edges connecting between branches are mismatched. Since such inter-branch edges have larger cost values than edges inside a branch, we can identify the initial and latest versions of each branch in the Product Evolution Tree.

Dataset 5: FFmpeg.

Figure 6 shows the Product Evolution Tree extracted from Dataset 5. The Product Evolution Tree does not correctly indicate when the Libav project was branched from the FFmpeg project. FFmpeg 0.5.3, which is the last release of the old FFmpeg project, was branched into Libav 0.5.4 and FFmpeg 0.5.5 in the actual history, but the tree seems to indicate that FFmpeg 0.5.5 was derived from Libav 0.5.5. In those versions, projects shared the same changes despite becoming independent projects.

Figure 6 also includes an incorrect edge between FFmpeg 0.5 and 0.5.2. This edge was connected since the same

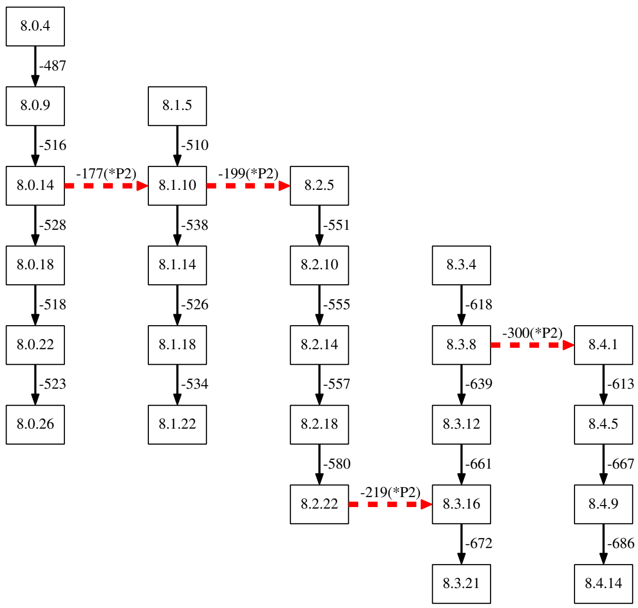


Figure 5: Product Evolution Tree for Dataset 4

files changed in 0.5.1 were changed again in 0.5.2. In other words, both versions had the same number of files changed from FFmpeg 0.5. This is the same phenomenon observed in Dataset 2.

The Product Evolution Tree shows Libav 0.5.5 is likely an origin of three branches. From this perspective, developers may choose the root version Libav 0.5.5 and leaf nodes FFmpeg 0.5, FFmpeg 0.5, FFmpeg 0.5.1, and Libav 0.5.9. The tree is still effective since the selected versions correctly contain the original and the latest versions. The tree reduces also the effort for comparing all 15 versions.

Dataset 6: BSD.

Figure 7 shows the Product Evolution Tree extracted from Dataset 6. The recall percentage of this dataset was the worst in the all datasets, since the dataset included 17 correct edges but a Product Evolution Tree could include at most 15 edges.

The Product Evolution Tree included a merge relationship for NetBSD-1.0, which was the next release of NetBSD-0.9 and included many source files from 4.4-BSD Lite. On the other hand, an edge from 4.4BSD Lite2 to FreeBSD-3.0 was not detected because the Product Evolution Tree does not allow closed paths. In addition, the cost between the two versions, $C(\text{4.4BSD Lite, FreeBSD-3.0, 0.9}) = -11$, indicated that all except for 11 files were dissimilar 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.

Looking at the tree overall, NetBSD-1.0 and NetBSD-1.2 receives the most attention because they process three edges. Leaf nodes FreeBSD-3.0, NetBSD-0.8, NetBSD-1.3, and NetBSD-1.2.1 also appear vital in the evolution history. The Product Evolution Tree suggests that 4.4-BSD Lite and OpenBSD-2.1 are not characteristic releases and it is difficult to identify them as important releases in this dataset.

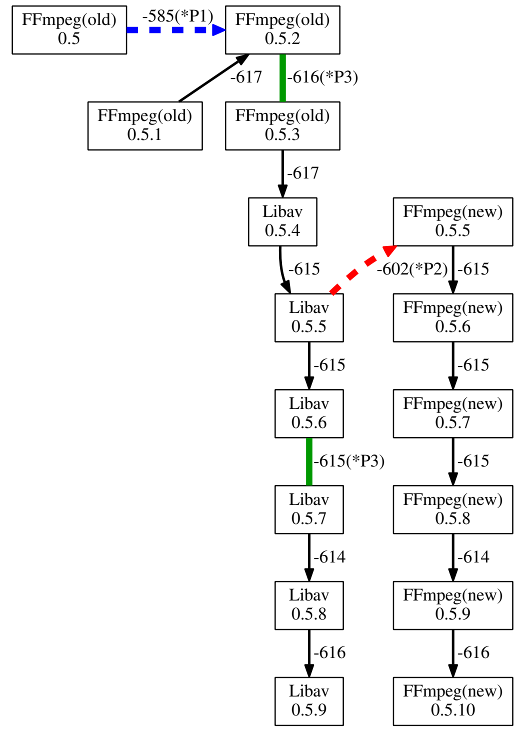


Figure 6: Product Evolution Tree for Dataset 5

5. DISCUSSION

5.1 Effectiveness of Product Evolution Tree

The results show that 70% to 100% of edges without labels and 53% to 92% of edges with labels were consistent with their respective actual evolution history. Furthermore, almost all of the latest products of each branch were represented as leaf nodes, except OpenBSD-2.1 in Dataset 6.

One of the major error patterns identified in the Product Evolution Trees was P2. P2 is not considered a serious problem since edges connecting products between branches usually have a larger cost than edges in a branch. Therefore, developers could easily recognize branches in a tree, even if their connections are not correct.

Another major error pattern that was identified, P3, act as a counterexample for our hypothesis that "source code is likely added." By analyzing source code, we found two major reasons behind the occurrences of P3. The first reason was that refactoring actions including class splitting and merging had been applied. Techniques for detecting refactoring [23] may be helpful to remove incorrect labels caused by this reason. The second reason behind P3 was non-essential changes [10]. Non-essential changes such as deleting dead code affect many lines of code, despite trailing in importance to other modification tasks such as feature enhancement. We can conjecture for some cases where the volume of source code is reduced, but P3 made up 20% (3 of 15 in Dataset 6) of extracted edges in our case study at most. Hence, our method for determining the direction was still effective overall. It should also be noted that we did not include release date information in our approach, as they are not always available. If release dates were available, all evolution directions would have been correctly extracted in cases where the edges were

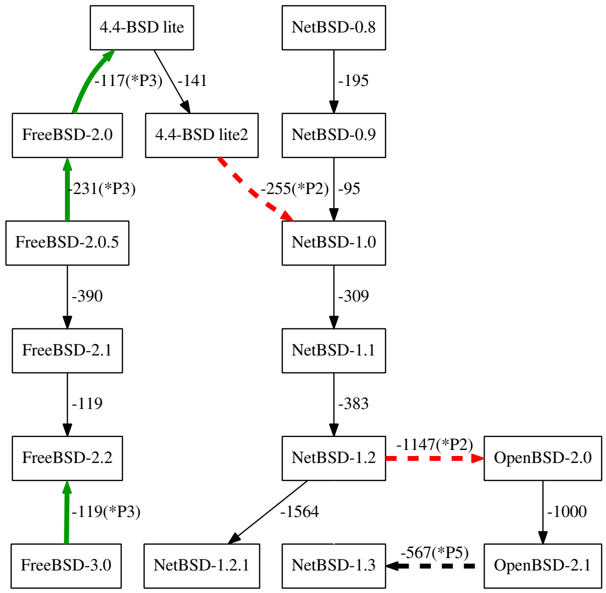


Figure 7: Product Evolution Tree for Dataset 6

correct.

Our approach always extracts $n - 1$ edges for n product variants as guaranteed by the algorithm. This is much smaller than $n(n-1)/2$ comparison required if the relationships among the products have been completely lost. Even if the tree included incorrect edges, majority of these incorrect edges still connect versions that were somewhat related. Therefore, the tree would reduce the number of product pairs developers would have to compare.

A tree cannot have closed paths; as a consequence, we hardly detected software merges as exhibit in Dataset 6. However, merging does not occur as frequently as forking, as shown in Figure 1. Since we could correctly detect forking on numerous occasions in the case study, another analysis technique could be built to detect merging.

Although our approach can be applied to arbitrary source code, intermediate versions are important clues to recovering an evolution history. In Dataset 4, although several incorrect edges between branches (P2 instances) were included, other edges were detected completely. This is because minor releases in a single branch possess small changes. In Dataset 1, numerous lines of code were altered between two major releases; as a consequence, our approach extracted one wrong direction. A full collection of products may also lead to the identification of incorrect edges as described in the P1 pattern, since two successive products are too similar to each other. However, we do not believe this is a problem because developers would be able to recognize the similarity between products through cost labels.

From the shape of the Product Evolution Tree, developers can learn where the starting point of the software evolution is and the points at which they branched. The value of the cost function also provides insight into understanding evolution history. If the cost of an edge is very small in comparison with its surrounding edges, there is evidence of only a slight difference and we can avoid comparing them. If a vertex has three edges and one of them has a higher cost than the others, the high cost edge may indicate branching, while the others may indicate the mainline.

A Product Evolution Tree does not directly provide commonality and variability among products, but it will allow developers to use several analysis techniques. For example, Hashimoto *et al.* proposed tracking source code changes in an evolution history [4]. They searched code clones in branches and mapped nodes of an Abstract Syntax Tree among versions. Using this technique, the origin of source code can be estimated in the Product Evolution Tree. Rubin *et al.* proposed searching product features using the source code differences [19]. Comparing the latest versions selected from a tree would be an effective for developers to search for features in a particular version.

We reviewed our approach with regards to scalability. A machine equipped with two Intel Xeon E5507 processors (2.27 GHz, 4 cores) and 24GB RAM required approximately one day to run a full analysis on Dataset 2, the largest dataset in this study. We believe that this is reasonable cost for developers since it is unlikely that product line analysis would be an urgent task. In addition, file similarity and term frequency vectors can be computed in parallel. Furthermore, they are reusable for future analysis. For example, if new products are added, we can incrementally re-construct a new Product Evolution Tree by comparing only the new products with the existing products.

5.2 Threats to Validity

In our approach, our assumption was that "two successive products are very similar to each other." However, this may not always be the case, especially when developers modify large amounts of code. Though the assumption is not always true, it did occur in many versions in our datasets.

Our algorithm for constructing the Product Evolution Tree is language-independent. However, all of the open-source projects we used in the case study were written in C. Source files in the datasets were well organized according to their functions. As a result, the number of edited files reflects the number of edited features. In other words, the number of similar files correctly reflected the distance between versions. If files are not clearly separated, our approach could be applied to such a program by dividing source code into functional units like subroutines or procedures.

Datasets 1, 2, 3, and 4 contained only PostgreSQL. Since there were some overlapping products, the average accuracy of the results may be affected by source code of PostgreSQL than other projects. This threat can easily be removed by additional experiments, since our tool only requires source code as input.

We used a single threshold 0.9 in the case study, which was determined by a small preliminary experiment. While this threshold value was suitable for the six datasets in this study, a different threshold may be more suitable for a different dataset.

6. CONCLUSION

Constructing SPLs from existing software products is becoming an important activity. In this paper, we proposed an automatic extraction of a Product Evolution Tree to help developers to understand evolution history of products. Our approach is dependent only on product source code and connects similar software products based on the number of similar files. We have applied our tool to 6 datasets including several open-source projects. In our results, 53% to 92%

of edges were correctly recovered. There are promising results, since we can identify branches and the latest versions of products using a Product Evolution Tree, even if the tree included incorrect edges.

In future work, we must investigate whether or not the correctness of this approach is accurate enough for developers to analyze software products or not. In addition, we would like to try automated detection of software merges, so that we can extract a complete evolution history for arbitrary projects.

7. ACKNOWLEDGMENTS

This work is supported by KAKENHI (No.23680001).

8. REFERENCES

[1] S. Duszynski, J. Knodel, and M. Becker. Analyzing the source code of multiple software variants for reuse potential. In *Proc. of WCRE*, pages 303–307, 2011.

[2] D. Faust and C. Verhoef. Software product line migration and deployment. *Software Practice and Experience*, 33:933–955, 2003.

[3] M. Godfrey and L. Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, 31(2):166–181, 2005.

[4] M. Hashimoto and A. Mori. A method for analyzing code homology in genealogy of evolving software. In *Proc. of FASE*, pages 91–106, 2010.

[5] A. Hemel and R. Koschke. Reverse engineering variability in source code using clone detection: A case study for linux variants of consumer electronic devices. In *Proc. of WCRE*, pages 357–366, 2012.

[6] 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 –. In *Proc. of ICSE*, pages 331–341, 2012.

[7] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[8] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in software product lines. In *Proc. of ICSE*, pages 311–320, 2008.

[9] S. Kawaguchi, P. K. Garg, M. Matsushita, and K. Inoue. MUDABlue: an automatic categorization system for open source repositories. *Journal of Systems and Software*, 79(7):939–953, 2006.

[10] D. Kawrykow and M. P. Robillard. Non-essential changes in version histories. In *Proc. of ICSE*, pages 351–360, 2011.

[11] R. Koschke, P. Frenzel, A. Breu, and K. Angstmann. Extending the reflexion method for consolidating software variants into product lines. *Software Quality Journal*, 17:331–366, 2009.

[12] C. W. Krueger. Easing the transition to software mass customization. In *Proc. of PFE*, pages 282–293, 2001.

[13] T. Lavoie, F. Khomh, E. Merlo, and Y. Zou. Inferring repository file structure modifications using nearest-neighbor clone detection. In *Proc. of WCRE*, pages 325–334, 2012.

[14] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32:176–192, 2006.

[15] Lucia, F. Thung, D. Lo, and L. Jiang. Are faults localizable? In *Proc. of MSR*, pages 74–77, 2012.

[16] C. McMillan, M. Grechanik, and D. Poshyvanyk. Detecting similar software applications. In *Proc. of ICSE*, pages 364–374, 2012.

[17] M. Nonaka, K. Sakuraba, and K. Funakoshi. A preliminary analysis on corrective maintenance for an embedded software product family. *IPSJ SIG Technical Report*, 2009-SE-166(13):1–8, 2009.

[18] C. R. Prim. Shortest connection networks and some generalizations. *Bell System Technical Jounal*, 36:1389–1401, 1957.

[19] J. Rubin and M. Chechik. Locating distinguishing features using diff sets. In *Proc. of ASE*, pages 242–245, 2012.

[20] J. Rubin, A. Kirshin, G. Botterweck, and M. Chechik. Managing forked product variants. In *Proc. of SPLC*, pages 156–160, 2012.

[21] V. Tenev and S. Duszynski. Applying bioinformatics in the analysis of software variants. In *Proc. of ICPC*, pages 259–260, 2012.

[22] K. Tian, M. Revelle, and D. Poshyvanyk. Using latent dirichlet allocation for automatic categorization of software. In *Proc. of MSR*, pages 163–166, 2009.

[23] P. Weißgerber and S. Diehl. Identifying refactorings from source-code changes. In *Proc. of ASE*, pages 231–240, 2006.

[24] T. Yamamoto, M. Matsushita, T. Kamiya, and K. Inoue. Measuring similarity of large software systems based on source code correspondence. In *Proc. of PROFES*, pages 530–544, 2005.

[25] K. Yoshimura and R. Mibe. Visualizing code clone outbreak: An industrial case study. In *Proc. of IWSC*, pages 96–97, 2012.

[26] K. Yoshimura, F. Narisawa, K. Hashimoto, and T. Kikuno. FAVE: factor analysis based approach for detecting product line variability from change history. In *Proc. of MSR*, pages 11–18, 2008.

[27] PRET-Extractor: `http://sel.ist.osaka-u.ac.jp/pret/`.