# Extraction of Evolution History from Software Source Code Using Linear Counting

Liu Shuchang[1,†1,a)] Takashi Ishio[2,b)] Tetsuya Kanda[1,c)] Katsuro Inoue[1,d)]

**Abstract:** A lot of software products might have evolved from an original version. Such kind of evolution history is considered as an important role in software re-engineering activity. However, history would always be lost and there might be only source code in the worst case. In this research, we proposed to extract an Evolution Tree to simulate the evolution history from the source code of product variants. We defined the product similarity using Jaccard Index, and we believed a pair of derived products shared the highest similarity, which turned to be an edge in the Evolution Tree. Instead of calculating the actual similarity from thousands of source files, Linear Counting became a choice to reach an efficient result. With empirical studies, we discussed the influence of parameters on the experiment result which was compared with the actual evolution history.

## 1. Introduction

During our daily software development, most of us are always looking for functionally similar code, and copy or edit it to build ours. It happens so frequently that people name it clone-and-own approach[1] whereby a new variant of a software product is usually built by coping and adapting existing variants. As a result, a lot of software products may have evolved from one original release.

With the analysis of evolution history, it is much more convenient for developers to deal with software re-engineering tasks, such as identifying bug-introducing changes[2], automatic fixing bugs[3], and discovering code clones[4]. Developers always wish to understand and examine evolution history for a wide variety of purposes. Of the 217 developers surveyed in Codoban's work[5], 85% found software history important to their development activities and 61% need to refer to history at least several times a day.

While evolution history supports developers for numerous tasks, in terms of many legacy systems, history is not available and developers are left to rely solely on their knowledge of the system to uncover the hidden history of modifications[6]. Furthermore, there may be only source code in the worst case, because management and maintenance are often scarcely taken care of in the initial phase[7].

In this research, we followed the intuition that two derived products were the most similar pair in the whole products. Similar software products must have similar source code and we defined product similarity based on it using Jaccard Index. Instead of calculating the actual similarity from thousands of source files, we chose the Linear Counting algorithm to estimate an approximate result. Depending on the similarities, we extracted an Evolution Tree to simulate the evolution history. After that, we applied our approaches to different 9 datasets to find out the optimization of various factors. Finally, we worked out the best configuration of them.

This research was also an extension of a previous study by Kanda et al.[8]. It focused on calculating the similarity by counting the number of similar source files between different product variants, which took plenty of time. Our approach depended on estimating instead. We regarded all the source files of one product variant as an entirety, which reached much more efficient. The result of the best configuration showed that 64.3% to 100% (86.5% on average) of edges in the extracted trees were consistent with the actual evolution history, at the speed of 7.15 MB/s to 25.78 MB/s (15.92 MB/s on average).

Our contributions were summarized as follows:

- We proposed an efficient approach to extract an ideal Evolution Tree from product variants
- We performed plenty of experiments to find out the influence of various factors
- After empirical studies, we worked out the best configuration that reached the best results
- Compared to the previous study, our approach was quite faster and showed better accuracy

This paper is organized as follows. Section 2 describes the related work and the previous study. Section 3 introduces our research approaches. The empirical study on a dataset will be shown in Section 4. Section 5 describes the discussion on experiment results. Conclusion and future work will be stated in Section 6.

[1] Osaka University, Suita, Osaka 565-0871, Japan
[2] Nara Institute of Science and Technology, Ikoma, Nara 630-0192, Japan
[†1] Presently with Osaka University
[a)] liu-sc@ist.osaka-u.ac.jp
[b)] ishio@is.naist.jp
[c)] t-kanda@ist.osaka-u.ac.jp
[d)] inoue@ist.osaka-u.ac.jp

## 2. Related Work

### 2.1 Code-history Analysis

In terms of software history analysis, multiple techniques had been proposed to model and analyze the evolution of source code at the line-of-code level of granularity. Reiss[9] proposed a group of line mapping techniques, some of which considered adjacent lines. Asaduzzaman et al.[10] proposed a language-independent line-mapping technique that detects lines which evolve into multiple others. Canfora[11] and Servant[12] further analyzed the results to disambiguate each line in the prior revision to each line in the subsequent revision.

However, the existing techniques presented potential limitations, in terms of modeling too many false positives (low precision) or too many false negatives (low recall), when compared with true code history. Moreover, such errors typically were compounded for analyses performed on multiple revisions, which could lead to substantially inaccurate results[13].

Furthermore, whether these techniques could capture those complex changes, such as movements of code between files, was unknown since they were based on textual differencing. Also, when the size of code increased, the time and space complexity would become exponentially growing.

### 2.2 Software Categorization

Instead of focusing on software history, some tools tended to automatically categorize software based on their functionality. Javier[14] proposed a novel approach by using semantic information recovered from bytecode and an unsupervised algorithm to assign categories to software systems. Catal[15] investigate the use of an ensemble of classifiers approach to solve the automatic software categorization problem when the source code is not available.

While these tools were able to detect similar or related applications from large repositories, our approach focused on those similar product variants derived from the same release, and they might be categorized into the same category by these tools. That was to say, the results of these tools could tell us that some product variants might be categorized into the same group while some other variants might be categorized into another one, which would help us to work out the evolution history.

However, product variants that derived from the same original product could be categorized into different groups as well, if developers changed them for different purposes. Besides, a product variant could even be categorized into a different group from what group the original product was categorized into, for their function could be totally different.

### 2.3 Previous Study

We already stated that this research was also an extension of the previous study by Kanda et al.[8], which also extracted an Evolution Tree based on similarities of product variants. The previous algorithm counted the number of similar files and cared about how much the files were changed as well. It treated both the file pair with no changes and the file pair with small changes as similar files.

Although the accuracy of the previous study was not too bad, because it calculated file-to-file similarities for all pairs of source files of all product variants, it took plenty of time. In the worst case, the time that it took to generate the result from a 1.03GB dataset of product variants could be about 38 hours. Thus we were looking forward to a different way to reach a more efficient result without reducing the accuracy.

By the way, the previous study proposed a method to calculate evolution direction. We would discuss it in detail in Section 5.3.

## 3. Study Approaches

### 3.1 Initialization

Firstly we applied initialization to input product variants. We stated that the source files were regarded as processing objects we would like to deal with. Since each line of code was something like text or sentences in the language, we selected n-gram modeling to do our initialization.

#### 3.1.1 N-gram Modeling

We determined to apply n-gram modeling to each line of code. For example, if the line of code was “int i = 0;” the result generated by trigram modeling (when n=3) should be like {int, nt , t i, i , i =, . . . }. To find out what n we should use, we also did empirical experiments to seek the influence of the number of n on our experiment results.

However, in our cases, the lines of code were not real text or sentences in writings, so there was an issue that whether we should apply n-gram modeling or just regard the whole line as processing objects. Thus we decided to do both of them to find out the difference. In terms of the analysis on n-gram modeling, there would be a detailed description in Section 4.3.

#### 3.1.2 Redundancy

It was easy to understand that there could be duplicate elements after n-gram modeling, so the next questions became whether we should remove it and if so, how we could remove it. Finally, we determined to mark the number of occurrences that an element had occurred during n-gram modeling. For example, in terms of the line of code: “int i = 0;” the result generated by unigram modeling (when n=1) should be like {i, n, t, , i, , . . . }. Then we held the number of times that each element occurred, and the result could become something like {i_1, n_1, t_1, _1, i_2, _2, =_1, _3, 0_1, ;_1}.

By marking the number of occurrences that an element had occurred, we removed most of the redundancy and saved the information of it that might have an influence on our results as well. After this, we also did extra experiments to compare the results that we removed redundancy (the distinguish mode) to the results that we did not remove redundancy (the ignore mode). The comparison would also be described in Section 4.3.

### 3.2 Product Similarity

Since we followed the intuition that two derived products were the most similar pair in the whole product variants, the question was how to describe the word “Similar”. We chose the Jaccard Index as our final choice.

The Jaccard Index, also known as Intersection over Union and the Jaccard similarity coefficient measured similarity between finite sample sets and was defined as the size of the intersection divided by the size of the union of the sample sets as below.

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{1}$$

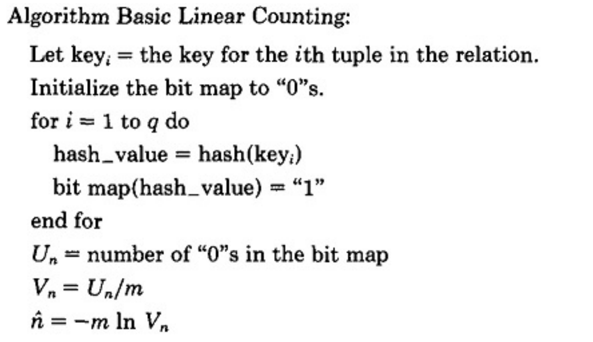
Here A and B meant different sample sets. Based on the Jaccard Index, we would like to count the cardinality number of the intersection from two different product variants as the size of the intersection, and the cardinality number of the union from those two as the size of the union.

However, after initialization, the processing objects became multisets of String. To generate an intersection or a union from two multi-sets of String was extremely difficult especially when the sizes of the multisets were not too small. Thus instead of calculating the actual cardinality number of the intersection and the union, we chose the Linear Counting algorithm to estimate an approximate result.

### 3.3 Linear Counting Algorithm

Various algorithms had been proposed to estimate the cardinality of multisets. The Linear Counting algorithm, as one of those popular estimating algorithms, was particularly good at a small number of cardinalities. In terms of why we selected the Linear Counting algorithm and the difference between it and the others, there was a detailed description in Section 5.1.

The Linear Counting algorithm was presented by Whang[16] and was based on hashing. Consider these was a hash function H, whose hashing result space has m values (minimum value 0 and maximum value m-1). Besides, the hash results were uniformly distributed. Use a bitmap B of length m where each bit was a bucket. Values of all the bits were initialized to 0. Consider a multiset S whose cardinality number was n. Apply H to all the elements of S to the bitmap B, and the algorithm could be described in Figure 1[16].



```
Algorithm Basic Linear Counting:
  Let $key_i$ = the key for the $i$th tuple in the relation.
  Initialize the bit map to "0"s.
  for $i = 1$ to $q$ do
    hash_value = hash($key_i$)
    bit map(hash_value) = "1"
  end for
  $U_n$ = number of "0"s in the bit map
  $V_n = U_n/m$
  $\hat{n} = -m \ln V_n$
```

**Fig. 1** The description of the Linear Counting algorithm

During hashing, if an element was hashed to k bits and the kth bit was 0, set it to 1. When all the elements of S were hashed, if there were Un bits that were 0 in B, here came an estimation of cardinality n as shown in Figure 1.

The estimation was maximum likelihood estimation (MLE). Since Whang had given a complete mathematical proof when he presented it, we would not give it again, but we wished to share an example from his presenting in Figure 2[16].

As shown in Figure 2, the column C, which we could treat as a multiset C, was hashed into a bitmap. Before hashing (scan), all

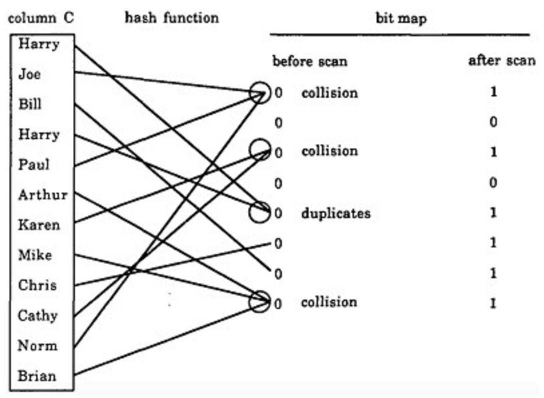


**Fig. 2** An example of the Linear Counting algorithm

the bits in the bitmap were 0 and after that, some elements turned into 1. When all the elements of C were hashed, we calculated the number of bits that were 0 in the bitmap and in this example, it was 2. At the same time, the size of the bitmap was 8. Thus we could calculate Vn like Vn = 2/8 = 1/4 and we could get an estimation of n like -8 * ln(1/4) = 11.0903. Besides, the actual cardinality number of multiset (column) C was 11.

In addition, after hashing the multisets of String became bitmaps. To calculate the intersection and union from those bitmaps was much easier and faster than to calculate them from multisets of String. In fact, it was just for computers to consider the basic logical operators. However, there could be danger when we applied the algorithm not to estimate the cardinality of one multiset but to estimate the cardinality of the intersection and union of multisets. To explain this in detail, there would be a discussion in Section 5.5.

Besides, although it looked like a good estimation, it was also easy to see that there could be duplicates and collision in the hashing process. To find out the influence of different factors on experiment results, Whang developed plenty of experiments and we also performed ours. After these empirical studies, we found two most important factors that mattered. They were the hashing function we applied to the multisets, and the size of the bitmap we set up. Both of them would be described in detail in Section 5.2.

### 3.4 Evolution Tree

After estimating, we had all the similarities between different product variants. Since our key idea was that two derived products should be the most similar pair in the whole products, there should be an edge between those pairs in the Evolution Tree. Besides, if we regarded the similarity as the weight for each possible edge because the similarity itself was undirected, to extract an Evolution Tree became to extract a minimum spanning tree of graph theory. Both of them meant that we founded a subset of edges that formed a tree that included every vertex (each product variant), where the total weight of all the edges in the tree was minimized (maximized actually in our cases while they were telling the same). Considering this, we decided to follow Prim's algorithm to extract the Evolution Tree.

Prim's algorithm was a greedy algorithm that finds a minimum spanning tree for a weighted undirected graph. The algorithm operates by building this tree one vertex at a time, from an arbitrary starting vertex, at each step adding the cheapest possible connection from the tree to another vertex.

In our cases, the starting vertex (product variant) was already known from existed evolution history. Actually to find out which product variant was the original version was too difficult especially when we had the only source code of those variants. In addition, the similarities were undirected, so we were not able to figure out the starter. Finally, we determined to treat the original version known from the existing evolution history as the starter vertex.

Following Prim's algorithm, our extraction could be described as performing these 4 steps.

(1) Input: a vertex set V included all the vertexes where each vertex meant a product variant; an edge set E included all the edges where each edge meant a possible derived pair of product variants and the similarity between any pair turned to be the weight of each edge;
(2) Initialization: Vnex = {x}, where x means the starter vertex in the tree which was the original version from existing evolution history; Enew = {};
(3) Repeat the following steps until all the elements in V were included in Vnew:
    (a) Find an edge (u, v) from E whose weight was maximum, which meant product variant u and product variant v shared the highest similarity, where u was from Vnew and v was from V with not being included in Vnew;
    (b) Add v to Vnew and add (u, v) to Enew;
(4) Output: use Vnew and Enew to describe the generated spanning tree.

In addition, although the similarities were undirected, in terms of an actual evolution history, there should still be directions, so we talked about this in Section 5.3.

### 3.5 Large-scale Oriented Optimization

Since it was very difficult for the previous study to deal with large-scale datasets, our approach would like to solve it. During our empirical experiments, we found that n-gram modeling requires most of the memories and time to generate the initial multisets. Thus we tried to save these multisets after n-gram modeling by putting them into the cache.

However, if the size of a dataset was not too big, we might be able to store these multisets. Once the size of a dataset became much larger, the out of memory errors kept coming. Besides, the bigger n we selected to do n-gram modeling, the more memory we needed to store these multisets. Thus we decided to change into another solution.

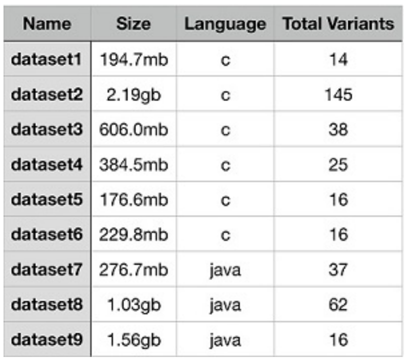
As we explained before, we applied a hashing function to each element of initial multisets after n-gram modeling. Any multiset would turn to be a bitmap after hashing. To store a bitmap was rather easier and faster than to store multisets of string. Besides, for one product variant, there was only one bitmap corresponding to it after all the elements of multisets were hashed. After that, once we saved the bitmaps, for any product variant, there should be only once n-gram modeling and hashing.

On the other hand, after we saved all the bitmaps, the remaining work was to calculate the intersection and union of those bitmaps. It would become much more convenient if we saved the bitmaps already.

Thus we determined to save every bitmap after all the elements of multisets were hashed. The optimization could avoid repeated calculating and reached an efficient result when we dealt with large-scale datasets.

## 4. Empirical Study

As we stated before, we applied our approaches to nine different datasets, which were shown in Figure 3.



| Name | Size | Language | Total Variants |
|---|---|---|---|
| dataset1 | 194.7mb | c | 14 |
| dataset2 | 2.19gb | c | 145 |
| dataset3 | 606.0mb | c | 38 |
| dataset4 | 384.5mb | c | 25 |
| dataset5 | 176.6mb | c | 16 |
| dataset6 | 229.8mb | c | 16 |
| dataset7 | 276.7mb | java | 37 |
| dataset8 | 1.03gb | java | 62 |
| dataset9 | 1.56gb | java | 16 |

**Fig. 3** From dataset1 to dataset9

All the datasets already had existing evolution history, and we would compare our results to it. To list all of them was meaningless, we selected dataset6 to further explain the approaches and to show what or how we thought during the study.

### 4.1 Dataset6

Dataset6 had 16 different product variants and its size was 229.8 MB and the programming language was C. It had two starter vertexes in the existing evolution history as shown in Figure 4.

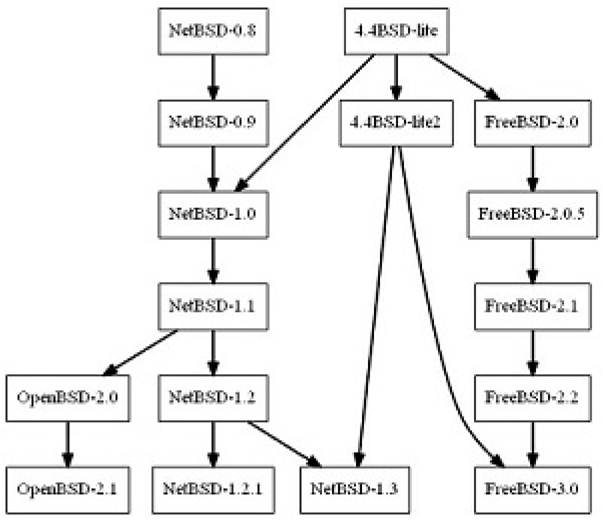


**Fig. 4** The actual evolution history of dataset6

In Figure 4, it was easy to find that both NetBSD-0.8 and 4.4BSD-lite were starter vertexes. Since we extracted the Evolution Tree by Prim's algorithm, we could only begin with one starter vertex. Furthermore, once we selected one of these two vertexes as a starter vertex, the edge that included the other vertex most likely turned to be wrong. Thus we decided to mark that edge as a special one.

## 4.2 Results

Since to list all of the results was too difficult, we just picked part of them in some conditions. In Figure 5, the first column described which product variant was selected as the starter vertex, which was from the actual evolution history in Figure 4. The second column presented the number of n of n-gram modeling. If it was no, it meant we did not apply n-gram modeling and we treated each line as a processing object during initialization, which was the best configuration and would be introduced in detail in Section 5.4. We also stated that since the similarity was shared by a pair of product variants and undirected, we could not give the direction for each edge. Thus we regarded all the reverse edges as proper edges, and we just recorded the number of them. For special edges, which was introduced in Section 4.1, we would not make it count when we calculated the accuracy.

Figure 5 also focused on the speed of the whole experiments. It described the speed which was like xx MB/s, and the speed was calculated by diving the time into the size of the whole dataset. We recorded the time of n-gram modeling together with hashing, because actually the hashing processing took very little time and it was very difficult to record it in seconds.

In terms of the time between different starter vertexes, they were the same number in Figure 5. Besides, to extract the Evolution Tree from product similarities that had been estimated did not take much time, which might be less than 1 seconds. Thus we did not record the time, either.

## 4.3 Analysis on N-gram Modeling

It was easy to find that in Figure 5, not to apply the n-gram modeling turned to be the best choice, because it reached both highest accuracy and the highest speed.

In terms of speed, the hashing itself took very little time and the hashing algorithm we selected was MurmurHash3 whose best benefit was exactly the speed. MurmurHash3 was non-cryptographic, so it could be much faster than traditional hash algorithms. Thus the time in total was almost the time that n-gram modeling took.

Well, how about the accuracy? Before we performed formal experiments, we made lots of tests to find out the influence of some parameters on experiment results. Finally, we found that the number of n did not affect the error between the number of cardinalities estimated and the actual number. In fact, it only determined how many distinct elements there were in the initial multisets after n-gram modeling. That meant, a bigger n of n-gram modeling made a product variant "Bigger" or more complex.

Obviously, the more distinct elements in the initial multisets, the more cardinality estimated by the Linear Counting algorithm. Besides, if any product became Bigger or more complex, as a result, the Jaccard Index, which was the Intersection over Union, must become smaller. Plenty of experiment results showed it, but there was still lack of a mathematical proof. More or less, the product similarities and the number of n were negatively correlated.

However, the truth was that to distinguish different product variants was not corresponding to the number of n of n-gram modeling. In other words, even though the similarity itself became on a lower level when n became larger, in terms of one product variant, the most similar pair that included this product variant would not significantly change. It might be sure that the lower similarity we estimated, the more exactly we could figure out whether these two product variants were similar or not. Nevertheless, in terms of extracting an Evolution Tree from all the product variants, it was not so sure that we needed to know how exactly any pair of product variants were similar.

As a result, we might extract the same Evolution Tree from different levels of product similarities. In Figure 5, although the n became larger after n=10, which was also equivalent to that the product similarities became lower, the accuracy still stayed the same, which meant we would extract the same Evolution Tree.

Well, how about the situation that we did not apply n-gram modeling? Other experiment results showed that we got lower product similarities than n-gram modeling. As we stated before, the product similarities and the number of n were negatively correlated. That meant the results generated from no n-gram modeling (up) was a kind of n-gram modeling where $n \to \infty$. Unfortunately, we just thought so, and we did not give a complete mathematical proof, which we considered as the future work. At the present time, since we regarded not applying n-gram modeling as a kind of n-gram modeling where $n \to \infty$, it would give a better result than applying any number of n of n-gram modeling in theory, while the experiment results showed it as well.

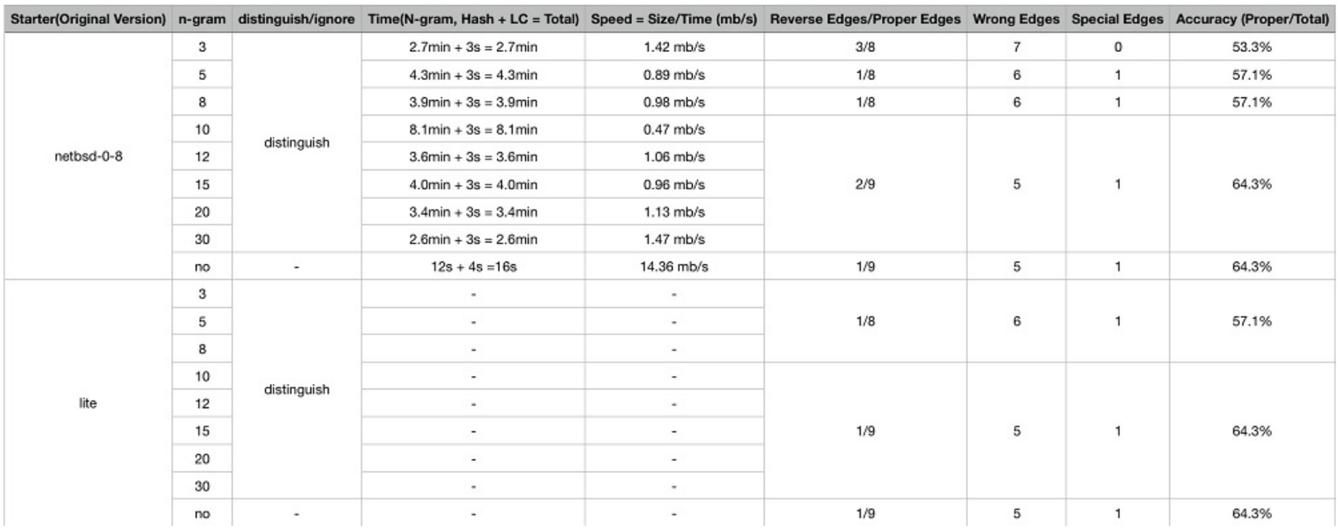
Because not to apply n-gram modeling did give a better result at a higher speed than to apply n-gram modeling, we made not to apply n-gram modeling into the best configuration. There would be a detailed summary of the best configuration in Section 5.4.

## 4.4 Analysis on Starter Vertex

As shown in Figure 5, the best results of beginning with different starter vertexes were almost the same. To expand on it, we checked the Evolution Trees that were extracted.

We found that there existing special edges. It was because we extracted the Evolution Tree by Prim's algorithm, and we could only begin with one starter vertex. Once we selected one of these two vertexes as a starter vertex, the edge that included the other vertex most likely turned to be wrong. Thus we decided to mark that edge as a special one if it turned to be wrong after extracting the trees. Moreover, the reverse edges should be treated as proper edges when calculating the accuracy.

We also found that not all the wrong edges in one Evolution Tree were equivalent to those in the other one. Furthermore, the Evolution Trees were not the same between different starter vertexes. This could be taken as the evidence to explain why we could not figure out the direction easily. We would talk about this more in Section 5.3.



| Starter(Original Version) | n-gram | distinguish/ignore | Time(N-gram, Hash + LC = Total) | Speed = Size/Time (mb/s) | Reverse Edges/Proper Edges | Wrong Edges | Special Edges | Accuracy (Proper/Total) |
|---|---|---|---|---|---|---|---|---|
| netbsd-0-8 | 3 | distinguish | 2.7min + 3s = 2.7min | 1.42 mb/s | 3/8 | 7 | 0 | 53.3% |
| | 5 | | 4.3min + 3s = 4.3min | 0.89 mb/s | 1/8 | 6 | 1 | 57.1% |
| | 8 | | 3.9min + 3s = 3.9min | 0.98 mb/s | 1/8 | 6 | 1 | 57.1% |
| | 10 | | 8.1min + 3s = 8.1min | 0.47 mb/s | 2/9 | 5 | 1 | 64.3% |
| | 12 | | 3.6min + 3s = 3.6min | 1.06 mb/s | | | | |
| | 15 | | 4.0min + 3s = 4.0min | 0.96 mb/s | | | | |
| | 20 | | 3.4min + 3s = 3.4min | 1.13 mb/s | | | | |
| | 30 | | 2.6min + 3s = 2.6min | 1.47 mb/s | | | | |
| | no | - | 12s + 4s =16s | 14.36 mb/s | 1/9 | 5 | 1 | 64.3% |
| lite | 3 | distinguish | - | - | 1/8 | 6 | 1 | 57.1% |
| | 5 | | - | - | | | | |
| | 8 | | - | - | | | | |
| | 10 | | - | - | 1/9 | 5 | 1 | 64.3% |
| | 12 | | - | - | | | | |
| | 15 | | - | - | | | | |
| | 20 | | - | - | | | | |
| | 30 | | - | - | | | | |
| | no | - | - | - | 1/9 | 5 | 1 | 64.3% |

**Fig. 5** Part of the experiment results of dataset6

## 5. Discussion

### 5.1 Cardinality Estimation

There existed not only the Linear Counting algorithm but also some others in the field of cardinality estimation. Cardinality estimation, also known as the count-distinct problem, was the problem of finding the number of distinct elements in a data stream with repeated elements. It had a wide range of applications and was of particular importance in database systems[18]. Various algorithms had been proposed in the past. Most of these algorithms were to solve the memory requirement problem by estimating the result rather than directly calculating it.

Many experiments had been developed to seek the difference between these algorithms. Zhang[17] analyzed five kinds of popular algorithms that were Linear Counting, LogLog Counting, Adaptive Counting, HyperLogLog Counting and HyperLogLog++ Counting. All the algorithms were implemented from an open source software library called CCARD-lib by Alibaba Group. He found that Linear Counting should be the best choice for those input multisets whose cardinality numbers were not too large.

Another experiment developed by Heule[18] also showed the evidence that the Linear Counting algorithm had a smaller error than the other algorithms for the small number of cardinalities. Heule proposed a series of improvements to HyperLogLog Counting algorithm and he implemented it for a system at Google to evaluate the result by comparing it to existing algorithms. Although he believed that his improvements made HyperLogLog Counting a state of the art cardinality estimation algorithm, the comparison he made still proved that for small cardinalities, the Linear Counting algorithm was still better than the improved HyperLogLog Counting algorithm.

In our cases, the cardinality number we would like to deal with was based on the lines of code that existed in the product variants. Since the product variants that were processed in the previous study were not so big and the sizes, in fact, were from 194.7MB to 2.19GB, we chose the Linear Counting algorithm to develop our experiments.

### 5.2 Main Factors

There were various factors that affected experiment results during empirical studies. To explain all of them was meaningless, and we determined to introduce the most important ones.

#### 5.2.1 N-gram Modeling

We already analyzed this comprehensively in Section 4.3. As a conclusion, to apply n-gram modeling would take much more time and more memory than not to apply it. Besides, the accuracy of applying a larger n of n-gram modeling and not applying n-gram modeling was near the same, while not applying n-gram modeling could reach the best results. We regarded the whole line as a processing object for hashing instead. Moreover, the results generated from no n-gram modeling was a kind of n-gram modeling where n .

#### 5.2.2 Hashing Algorithm

Because the Linear Counting algorithm was based on hashing, the hashing function we applied to the multisets was quite important. The algorithm assumed that after all the elements of multisets were hashed, the hash values should be uniformly distributed. Besides, we would apply the hashing function to thousands of lines of source code. Thus we had to learn about the differences between different hashing algorithms.

Earlz[19] asked a similar question on the website StackExchange to wonder which hashing algorithm was best for uniqueness and speed. Many people contributed to answers and most of them voted for MurmurHash3 which was also selected by Redis, Memcached, Cassandra, HBase, and Lucene. Because traditional hashing algorithms, such as MD5, SHA1 and SHA256 were designed to be secure, which usually meant they were slower than algorithms that were less unique. It was until 2008 that MurmurHash was created by Austin Appleby. MurmurHash was non-cryptographic, so it could be much faster than traditional hashing algorithms. Besides, it constructed random decomposition of all elements from input to keep results uniform, which was just what we needed.

The current version was MurmurHash3. It existed in a number of variants, all of which had been released into the public domain. We selected one java port authored by Yonik Seeley[20] that produced exactly the same hash values as the official final C++ version.

#### 5.2.3 Bitmap Size

Another important factor was the bitmap size that we set up. From advanced analysis by Whang[16], we could learn that there was a direct relationship between bitmap size m and cardinality number n. The bigger cardinality number we would like to estimate, the larger bitmap size we needed, which was also the same as the error precision. That was to say, we would better make the size of the bitmap as big as possible.

However, the error precision here meant we could get more exact cardinality numbers of intersection and union. Since our goal was not to calculate the similarity between different product variants but to extract an Evolution Tree to simulate the evolution history. It might be sure that the more exact similarity we calculated, the more exactly we could figure out whether these two product variants were similar or not. Nevertheless, in terms of extracting an Evolution Tree from all the product variants, it was not so sure that we needed to know how similar any pair of product variants were. In other words, we might extract the same Evolution Tree from different levels of product similarities.

On the other hand, it was difficult to confirm whether the bitmap size that we set up was big enough, because the size of initial multisets was always varying. Moreover, although there was a relationship between bitmap size and the number of cardinalities, we could not make the size adaptive. To make it adaptive meant to count the number of cardinality in the initial multisets, which was exactly what we were requesting.

Finally, we set up a bitmap whose size was 128,000,000 bits. The error between the cardinality estimated and actual cardinality was less than 0.001 when the size of the input dataset was 1 GB.

### 5.3 Direction

We extracted the Evolution Tree based on product similarities between different pairs of product variants, and the product similarities were undirected because each of them was shared by any pair of variants. Thus we could not generate any direction based on the similarities.

Since the previous study[8] figured out the directions, we tried to learn about how it did. However, the approach it used to calculate the evolution direction was based on a hypothesis that every modification was doing an adding. In fact, the modification between a derived pair of product variants could be various. We could either add something to the original version or delete something from it. If we assumed that there was only adding, most of the directions might be wrong.

However, if we were not aware of the directions, we could make an approximation to evolution history only if we already knew the starter vertex (the original version). The analysis in Section 4.4 showed that the Evolution Tree extracted from different starter vertexes could be much different. Although the results of the accuracy were similar to each other, we could not select any one of them to declare what was the exact Evolution Tree.

As a conclusion, we could not calculate the directions right now. We could extract the Evolution Tree only if we know the starter vertex (the original version). We had to treat the reverse edges as proper edges when calculating the accuracy.

### 5.4 Best Configuration

The best configuration was still under the approach described in Section 3. The biggest difference was that we would not apply n-gram modeling to any line of code from product variants. The best configuration was summarized as below:

(1) No n-gram modeling
  (a) The element of initial multisets was each line of code.
  (b) The hashing function was applied to each line of code to generate bitmaps.

(2) The Linear Counting algorithm
  (a) MurmurHash3: authored by Yonik Seeley[20] on GitHub.
  (b) The bitmap size: 128,000,000 bits.
  (c) Product similarities: based on the Jaccard Index.

(3) The Evolution Tree
  (a) The minimum spanning tree: Prim's algorithm.
  (b) Starter vertexes: known from actual history.
  (c) Directions: undirected.

In terms of the results, the speed was from 7.15 MB/s to 25.78 MB/s (15.92 MB/s on average) and there were from 64.3% to 100% (86.5% on average) of edges in the extracted trees were consistent with the actual evolution history.

### 5.5 Threats to Validity

In this research, we applied the Linear Counting algorithm to estimate product similarities and we considered n-gram modeling to generate the initial multisets, which turned to be not a good choice at last. Thus we treated each line of code as the element in initial multisets. However, what exactly should be the element was still unknown. The results we generated were based on the idea that the most similar product variants had the most similar lines of code. It might be true but there was not any complete mathematical proof. For the reason that we only had limited datasets that existed actual history, we could not answer this question demonstratively, which might be a threat to validity.

On the other hand, we defined the product similarity based on Jaccard Index. Although Jaccard Index was widely used for calculating similarities between sets, the objects that we dealt with this time were multisets. There might be errors during this processing, but whether there existed another better choice was mysterious. Since this could be an area of NLP, maybe we could work out a better solution after we learned about some knowledge of NLP, which also might be a threat to validity.

Another problem was about the directions. As we described in Section 5.3, we could not calculate the directions right now. Once there was a way to solve the problem of directions, there would be threats.

Furthermore, we should consider the Linear Counting algorithm as well. The Linear Counting algorithm was defined to estimate the cardinality of multisets. However, we applied it to estimate the intersection and union of multisets by counting the number of 0 bits in the bitmap of the intersection and union. To estimate the union of bitmaps should be safe because the difference of elements did not make sense. However, if different elements might turn into the same bit, to estimate the intersection of bitmaps would become dangerous, although in terms of different elements, the possibility of turning into the same bit after applying hashing should be quite low when the size of bitmap was large enough. Thus we made another experiments by calculating like $(|A| + |B| - |A \cup B|)/|A \cup B|$ instead. We found that there was no influence on extracting the Evolution Tree, and we worked out the same trees as before. Nevertheless, although calculating like $(|A| + |B| - |A \cup B|)/|A \cup B|$ might decrease the influence of the possibility that different elements would turn into the same bit, it would still not be safe because we did estimating four times, which might introduce some other danger. As a result, this would be a threat to validity.

Finally, the Linear Counting algorithm did not work very well when the size of datasets became too large. Besides, to use the Linear Counting algorithm meant that we needed to prepare a uniformly distributed hashing algorithm. When the size of dataset became much larger, the MurmurHash3 could be no longer useful. At that time, we had to change the whole approaches into other ones.

## 6. Conclusion and Future Work

In this research, we proposed an efficient approach to extract an ideal Evolution Tree from product variants. We performed plenty of experiments to find out the influence of various factors, and we summarized the best configuration which worked out the best result. Compared to the previous study, we reached a much faster speed and higher accuracy. The result of the best configuration showed that 64.3% to 100% (86.5% on average) of edges in the extracted trees were consistent with the actual evolution history, at the speed of 7.15 MB/s to 25.78 MB/s (15.92 MB/s on average).

During the research, we tried to perform n-gram modeling at first, which turned to be not a good choice in the end. There might be other threats like this that affected the experiment results. However, for the reason that we only had limited conditions, we were not able to give a better result than the best configuration.

For future work, we will apply our approaches to larger datasets to find out the boundary of the Linear Counting algorithm. We will deal with other kinds of programming language as well. To learn about the knowledge of NLP is also important, and we will work out whether there exist other methods to define product similarities better than the Jaccard Index. Furthermore, we will consider how to solve the problem of directions indeed as well.

**Acknowledgments** We would like to thank all the participants of our study.

### References

[1] Rubin, Julia and Kirshin, Andrei and Botterweck, Goetz and Chechik, Marsha: Managing forked product variants, *Proceedings of the 16th International Software Product Line Conference-Volume 1*, ACM, pp. 156-160 (2012).

[2] Śliwerski, Jacek and Zimmermann, Thomas and Zeller, Andreas: When do changes induce fixes? *ACM sigsoft software engineering notes-volume 30*, ACM, PP. 1-5 (2005).

[3] Servant, Francisco and Jones, James A: WhoseFault: automatic developer-to-fault assignment through fault localization, *Software Engineering (ICSE), 2012 34th International Conference on*, IEEE, pp. 36-46 (2012).

[4] Bakota, Tibor: Tracking the evolution of code clones, *International Conference on Current Trends in Theory and Practice of Computer Science*, Springer, pp. 86-98 (2011).

[5] Codoban, Mihai and Ragavan, Sruti Srinivasa and Dig, Danny and Bailey, Brian: Software history under the lens: a study on why and how developers examine it, *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, IEEE, pp. 1-10 (2015).

[6] Lavoie, Thierry and Khomh, Foutse and Merlo, Ettore and Zou, Ying: Inferring repository file structure modifications using nearest-neighbor clone detection, *Reverse Engineering (WCRE), 2012 19th Working Conference on*, IEEE, pp. 325-334 (2012).

[7] Parnas, David Lorge: Software aging, *Software Engineering, 1994. Proceedings. ICSE-16., 16th International Conference on*, IEEE, pp. 279-287 (1994).

[8] Kanda, Tetsuya and Ishio, Takashi and Inoue, Katsuro: Approximating the Evolution History of Software from Source Code, *IEICE TRANSACTIONS on Information and Systems-volume 98*, The Institute of Electronics, Information and Communication Engineers, pp. 1185-1193 (2015).

[9] Reiss, Steven P: Tracking source locations, *Proceedings of the 30th international conference on Software engineering*, ACM, pp. 11-20 (2008).

[10] Asaduzzaman, Muhammad and Roy, Chanchal K and Schneider, Kevin A and Di Penta, Massimiliano: LHDiff: A language-independent hybrid approach for tracking source code lines, *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, IEEE, pp. 230-239 (2013).

[11] Canfora, Gerardo and Cerulo, Luigi and Di Penta, Massimiliano: Identifying Changed Source Code Lines from Version Repositories, *MSR-volume 7*, pp. 14 (2007).

[12] Servant, Francisco and Jones, James A: History slicing: assisting code-evolution tasks, *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ACM, pp. 43 (2012).

[13] Servant, Francisco and Jones, James A: Fuzzy fine-grained code-history analysis, *Proceedings of the 39th International Conference on Software Engineering*, IEEE Press, pp. 746-757 (2017).

[14] Escobar-Avila, Javier and Linares-Vásquez, Mario and Haiduc, Sonia: Unsupervised software categorization using bytecode, *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, IEEE Press, pp. 229-239 (2015).

[15] Catal, Cagatay and Tugul, Serkan and Akpinar, Basar: Automatic Software Categorization Using Ensemble Methods and Bytecode Analysis, *International Journal of Software Engineering and Knowledge Engineering-volume 27*, World Scientific, pp. 1129-1144 (2017).

[16] Whang, Kyu-Young and Vander-Zanden, Brad T and Taylor, Howard M: A linear-time probabilistic counting algorithm for database applications, *ACM Transactions on Database Systems (TODS)-volume 15*, ACM, pp. 208-229 (1990).

[17] Zhang Yang: Effect Experiments and Practical Proposals of Five Common Cardinality Estimation Algorithms, *http://blog.codinglabs.org/articles/cardinality-estimate-exper.html*, CodingLabs (2013).

[18] Heule, Stefan and Nunkesser, Marc and Hall, Alexander: HyperLogLog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm, *Proceedings of the 16th International Conference on Extending Database Technology*, ACM, pp. 683-692 (2013).

[19] Earlz: Which hashing algorithm is best for uniqueness and speed, `https://softwareengineering.stackexchange.com/questions/49550/which-hashing-algorithm-is-best-for-uniqueness-and-speed`, StackExchange (2011).

[20] Yonik Seeley: MurmurHash3, `https://github.com/yonik/java_util/blob/master/src/util/hash/MurmurHash3.java`, Yonik Seeley (2015).