# 修士学位論文

題目

# Extraction of Evolution Tree from Product Variants Using Linear Counting Algorithm

指導教員
井上 克郎 教授

報告者
Liu Shuchang

平成 30 年 2 月 7 日

大阪大学 大学院情報科学研究科
コンピュータサイエンス専攻 ソフトウェア工学講座

平成 29 年度 修士学位論文

Extraction of Evolution Tree from Product Variants Using Linear Counting Algorithm

Liu Shuchang

## 内容梗概

A lot of software products might have evolved from one original release. Such kind of evolution history was considered as an important role in software re-engineering activity. However, management and maintenance were often scarcely taken care of in the initial phase. As a result, history would always be lost and there may be only source code in the worst case.

In this research, we proposed an efficient approach to extract an ideal Evolution Tree from product variants. We defined product similarities using the Jaccard Index, and we believed that a pair of derived products shares the highest similarity, which turns 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 estimate an approximate result.

With empirical studies, we discussed the influence of various factors on the experiments which were compared with the actual evolution history. The Best Configuration showed that 86.5% (on average) of edges in the extracted trees were consistent with the actual one, at the speed of 15.92 MB/s (on average).

## 主な用語

Software Evolution History
Linear Counting
Software Product Similarity

## 目 次


**1 Introduction** **4**

**2 Related Work** **6**

2.1 Code-history Analysis . . . . . . . . . . . . . . . . . . . . . . . 6

2.2 Software Categorization . . . . . . . . . . . . . . . . . . . . . . 6

2.3 Previous Study . . . . . . . . . . . . . . . . . . . . . . . . . . 7

**3 Study Approaches** **8**

3.1 Initialization . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

3.1.1 N-gram Modeling . . . . . . . . . . . . . . . . . . . . . . 8

3.1.2 Redundancy . . . . . . . . . . . . . . . . . . . . . . . . 9

3.2 Product Similarity . . . . . . . . . . . . . . . . . . . . . . . . 9

3.3 Linear Counting Algorithm . . . . . . . . . . . . . . . . . . . . 10

3.4 Evolution Tree . . . . . . . . . . . . . . . . . . . . . . . . . . 12

3.5 Large-scale Oriented Optimization . . . . . . . . . . . . . . . . 13

3.6 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14

3.7 Example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15

**4 Empirical Studies** **19**

4.1 Empirical Study on Dataset6 . . . . . . . . . . . . . . . . . . . 19

4.1.1 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . 20

4.1.2 Analysis on N-gram Modeling . . . . . . . . . . . . . . . 23

4.1.3 Analysis on Starter Vertex . . . . . . . . . . . . . . . . . 25

4.2 Empirical Study on Dataset7 . . . . . . . . . . . . . . . . . . . 26

4.2.1 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . 27

4.2.2 Supplementary Analysis on N-gram Modeling . . . . . . . 27

4.2.3 Analysis on Complex Vertex . . . . . . . . . . . . . . . . 28

**5 Discussion** **31**

5.1 Cardinality Estimation . . . . . . . . . . . . . . . . . . . . . . 31

5.2 Main Factors . . . . . . . . . . . . . . . . . . . . . . . . . . . 33

5.2.1 N-gram Modeling . . . . . . . . . . . . . . . . . . . . . . 33

5.2.2 Hashing Algorithm . . . . . . . . . . . . . . . . . . . . . 33

5.2.3 Bitmap Size . . . . . . . . . . . . . . . . . . . . . . . . 34

5.3 Direction . . . . . . . . . . 35
5.4 Best Configuration . . . . . . . . . . 36
5.5 Threats to Validity . . . . . . . . . . 37

**6 Conclusion and Future Work** **39**

**謝辞** **40**

**参考文献** **41**

## 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[15] 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.

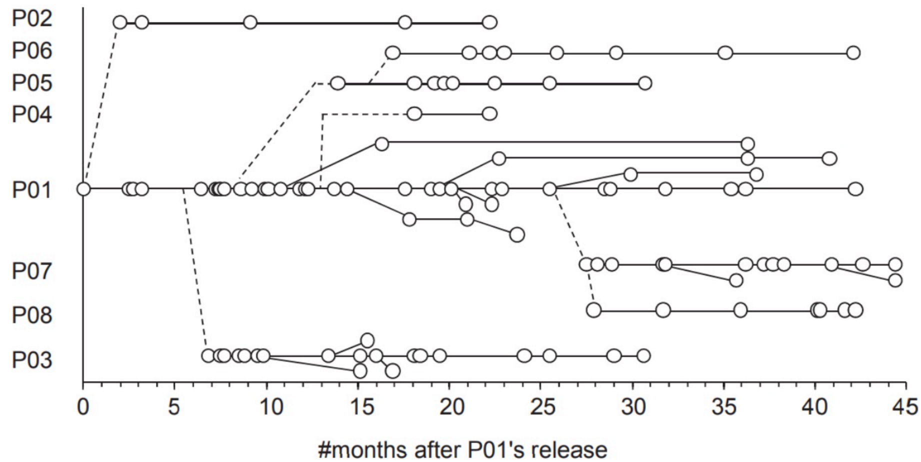


図 1: An example of how product variants derived from a single one

Figure 1[12] shows an example about such kind of evolution history. 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. 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. In Figure 1 we may find only 8 major product series and each series of products has 2 to 42 versions.

With this analysis, it is much more convenient for developers to deal with software re-engineering tasks, such as identifying bug-introducing changes[20], automatic fixing bugs[18], and discovering code clones[2]. Of the 217 developers surveyed in Codoban's work[6], 85% found software history important to their development activities and 61% need to refer to history at least several times a day. That is to say, developers always wish to understand and examine evolution history for a wide variety of purposes. For example, during a software re-engineering, development teams rely on the revision history of the software system to recover its design and transfer its functionalities.

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[11]. 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[13].

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.[10]. 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 thesis is organized as follows. Section 2 describes the related work and the previous study. Section 3 introduces our research approaches. Empirical studies on two datasets will be shown in Section 4. Section 5 describes the discussion on experiment results. Conclusion and future work will be stated in Section 6.

## 2 Related Work

This section described the related work and previous study. Before we performed our experiments, we had looked for the work that dealt with software evolution history as well. However, there were not many articles that were exact to talk about how to work out the evolution history. Thus we introduced two fields that were analogous to our work to some extent. One focused on analyzing the code to do history analysis, and the other was to categorize different software.

### 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[14] proposed a group of line mapping techniques, some of which considered adjacent lines. Asaduzzaman et al.[1] proposed a language-independent line-mapping technique that detects lines which evolve into multiple others. Canfora[4] and Servant[17] 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[19].

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[8] proposed a novel approach by using semantic information recovered from bytecode and an unsupervised algorithm to assign categories to software systems. Catal[5] 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.[10], 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 based on the number of modified lines between two products. It calculated the direction by counting the amount of modified code in pairs of software products. However, its hypothesis was based on that source code was likely added. That was to say, if the modification was replacing or deleting, it could not give a correct answer. Actually, there could be any kind of modification during developing software. Therefore, we did not think it was a feasible idea and we wished to find a new way if possible.

# 3 Study Approaches

This section described all the study approaches we took during the research. It was constructed in the sequence of the process flow. We would explain the profile of each approach and introduced how we applied it to perform the experiments. Most of the approaches contained various factors that had an influence on experiment results, which we would talk about in detail in Section 5.

## 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

N-gram modeling was a type of probabilistic language model for predicting the next item in such a sequence in the form of an (n-1)-order Markov model. Here an n-gram was a contiguous sequence of n items from a given sample of sentences[3]. It was widely used in probability, communication theory, computational linguistics (for instance, statistical natural language processing), computational biology (for instance, biological sequence analysis), and data compression. Two benefits of n-gram modeling (and algorithms that use them) were simplicity and scalability – with larger n, a model could store more context with a well-understood space-time tradeoff, enabling small experiments to scale up efficiently.

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 4.1.2 and Section 4.2.2.

### 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. In terms of programming languages, there could be lots of general words such as This, String, Long, New, and so on. However, we could not easily remove these words to do initializations because they were also part of the code. Furthermore, even if we remove these words there would still be redundancy from plenty of other situations. For example, the names that developers used to describe the parameters and methods were usually near the same.

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 4.1.2 and Section 4.2.2.

## 3.2 Product Similarity

Now we had appropriate processing objects as a start point and to extract the Evolution Tree of product variants, another important point was to define the product similarity which the Evolution Tree was based on. 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 was a statistic used for comparing the similarity and diversity of sample sets. The Jaccard 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, while in our cases, the input objects were not sets but multisets. It meant that there could be repeated elements, even though we might have removed most of the redundancy. Thus there could be some other index or the coefficient that was able to be used to define the product similarity, which might generate a better result. Because we had not found one more suitable than Jaccard Index, we finally defined our product similarity based on it.

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. After that, we could calculate the result of the division.

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 the Linear Counting algorithm and the others, there was a detailed description in Section 5.1.

The Linear Counting algorithm was presented by Whang[21] 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 2[21].

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 2.

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 3[21].

**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
$V_n = U_n/m$
$\hat{n} = -m \ln V_n$

図 2: The description of the Linear Counting algorithm

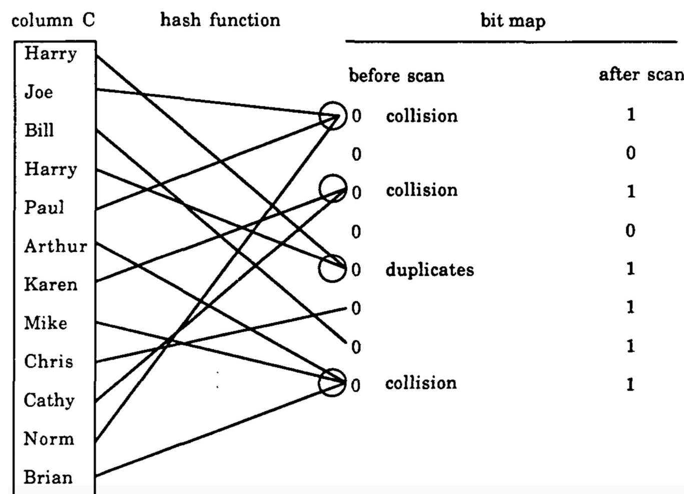


図 3: An example of the Linear Counting algorithm

As shown in Figure 3, the column C, which we could treat as a multiset C, was hashed into a bitmap. Before hashing (scan), all 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. After calculating the intersection and union, we followed the Linear Counting algorithm to estimate the cardinality number of them and calculated the product

similarity. 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. Since to save a bitmap would not need too much memory, the errors would decrease as well.

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.

### 3.6 Summary

As a summary, our approach was divided into 4 steps.

1. **Initialization**. For all source files of product variants, we did two kinds of initialization.

   (a) We would apply different n of n-gram modeling to each line of code and we treated the results as processing objects; during n-gram modeling, we considered an ignore mode to keep the redundancy and a distinguish mode to mark the number of occurrences that an element had occurred to remove the redundancy.

   (b) We would not apply any n-gram modeling and we just regarded the whole line as a processing object.

2. **Estimating Product Similarities**. After initialization, we estimated the product similarities using the Linear Counting algorithm.

   (a) **Hashing**. Since each product variant turned to be a multiset after initialization, we applied the hash function to every element in the multisets to generate bitmaps; if a multiset was initialized by a.1), the element should be the n-gram sequence generated from each line of code and if a multiset was initialized by a.2), the element was the whole line; we also saved the bitmaps in memory to avoid repeated calculating.

   (b) **Calculating the Intersection and Union**. Since each bitmap was corresponding to a product variant, we calculated the intersection and union from different pairs of bitmaps, which was also equivalent to calculating the intersection and union from different pairs of product variants; besides, to calculate an intersection or union from bitmaps was very easy, because, in fact, it was just for computers to consider the basic logical operators.

(c) **Calculating the Jaccard Index**. Once we got an intersection or union from bitmaps, we counted the number of 0 bits and estimated the cardinality number by the Linear Counting algorithm, and we calculated the Jaccard Index (Intersection over Union), which turned to be the product similarity.

3. **Extracting the Evolution Tree**. After we estimated all the similarities, we extracted the Evolution Tree which was also a minimum spanning tree and we developed it by Prim's algorithm, which was exhaustively described in Section 3.4.

4. **Verification**. We compared our Evolution Tree with existing actual evolution history to verify whether our approximation was correct or not.

### 3.7 Example

We considered an example to describe the steps in Section 3.6 more exhaustively. There was a dataset whose name was dataset9 and it had 16 product variants. Its size was 1.56 GB and the programming language was Java. The existing actual evolution history was shown in Figure 4.

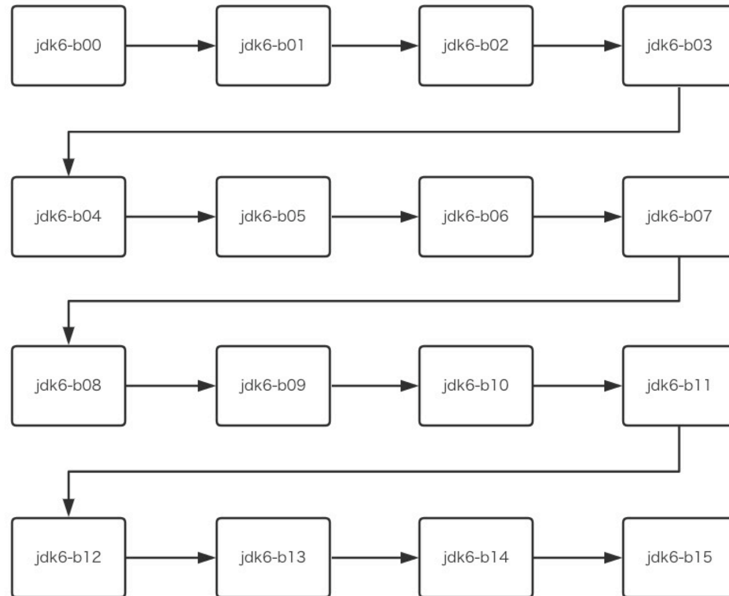


図 4: The actual evolution history of dataset9

In figure 4, each rounded rectangle corresponded to a product variant and the label was the name of each product. Involuntarily, the naming scheme was exactly telling the actual

sequence of the evolution. Nevertheless, we would not make the names count during our experiments and the processing object was only the source code of source files.

To extract our Evolution Tree to approximate the actual evolution history, we followed the steps in Section 3.6.

1. Initialization. We did two kinds of initialization, for all source files of product variants from jdk6-b01 to jdk6b15.

   (a) We would apply different n of n-gram modeling to each line of code; during n-gram modeling, we considered an ignore mode to keep the redundancy and a distinguish mode to remove the redundancy.

   (b) We would not apply any n-gram modeling and we just regarded the whole line as a processing object.

2. Estimating product similarities. After initialization, we estimated the product similarities using the Linear Counting algorithm.

   (a) Hashing. We applied the hashing function to every element in the multisets to generate bitmaps; we also saved the bitmaps in memory to avoid repeated calculating.

   (b) Calculating the intersection and union. We calculated the intersection and union from different pairs of bitmaps.

   (c) Calculating the Jaccard Index. We counted the number of 0 bits from bitmaps of the intersection and union; then we estimated the cardinality number by the Linear Counting algorithm; finally, we calculated the Jaccard Index, which turned to be the product similarity.

   After that, we could get a table that described product similarities from different pairs of product variants. There was part of the table as shown in Figure 5.

   In Figure 5, the horizontal axis and the vertical axis represented each product variant from dataset9, and the number in the middle was the similarity between them. Coincidentally all the similarities were very high, which was not a common fact in other datasets.

3. Extracting the Evolution Tree. After we got a table like the one in Figure 5 for all the similarities, we could extract an Evolution Tree which was also a minimum spanning tree; we developed it by Prim's algorithm.

| | jdk6-b00 | jdk6-b01 | jdk6-b02 | jdk6-b03 | jdk6-b04 | jdk6-b05 |
|---|---|---|---|---|---|---|
| **jdk6-b00** | 1 | 1 | 0.999718415329949 | 0.9993116090683708 | 0.990043289631914 | 0.9894859466686343 |
| **jdk6-b01** | - | 1 | 0.999718415329949 | 0.9993116090683708 | 0.990043289631914 | 0.9894859466686343 |
| **jdk6-b02** | - | - | 1 | 0.9995927581695581 | 0.9903218428434175 | 0.9897643437951325 |
| **jdk6-b03** | - | - | - | 1 | 0.9907257184931767 | 0.9901674035254002 |
| **jdk6-b04** | - | - | - | - | 1 | 0.9994352175296378 |
| **jdk6-b05** | - | - | - | - | - | 1 |

図 5: Part of results (dataset9, n=8, the distinguish mode) of similarity estimating

To show an example, we could consider part of the results in Figure 5; at first, we knew that the start vertex should be jdk6-b00 from existing evolution history, and we picked it out as our starter; then we found the highest similarity between jdk6-b00 and the other product variant, which was jdkb-01; therefore we could draw an edge between jdk6-b00 and jdk6-b01; after that, we found the next highest similarity between jdk6-b00 or jdkb-01 and the other product variant; we repeated this until we completed the Evolution Tree.

4. Verification. We compared the Evolution Tree with existing actual evolution history to verify whether our approximation was correct or not.

| StartVertex | EndVertex | Sim |
|---|---|---|
| **jdk6-b00** | jdk6-b01 | 1 |
| **jdk6-b00** | jdk6-b02 | 0.999718415329949 |
| **jdk6-b02** | jdk6-b03 | 0.9995927581695581 |
| **jdk6-b03** | jdk6-b04 | 0.9907257184931767 |
| **jdk6-b04** | jdk6-b05 | 0.9994352175296378 |
| **jdk6-b05** | jdk6-b06 | 0.9967804952883383 |
| **jdk6-b06** | jdk6-b07 | 0.9999814348222185 |
| **jdk6-b07** | jdk6-b08 | 0.9999965963512913 |
| **jdk6-b08** | jdk6-b09 | 0.99999489454362 |
| **jdk6-b09** | jdk6-b10 | 0.9917724880915659 |
| **jdk6-b10** | jdk6-b11 | 0.9999717511759275 |
| **jdk6-b11** | jdk6-b12 | 0.9990934204767752 |
| **jdk6-b12** | jdk6-b13 | 0.999845534087624 |
| **jdk6-b13** | jdk6-b15 | 0.999399005066143 |
| **jdk6-b15** | jdk6-b14 | 1 |

図 6: The result table that described the Evolution Tree (datset9, n=8, the distinguish mode)

As we stated before, the final result would be a table for describing the Evolution Tree; in Figure 6, we recorded each edge by StartVertex, EndVertex, and its weight – in other words, the product similarity between the pair of product variants; we compared this with the actual evolution history in Figure 4; we colored wrong edges yellow and reverse edges blue; because similarities were shared by the pair of product variants and there was no direction about it, we could not show what direction it should be between a pair of product variants; as a result, we counted the reverse edges as proper edges to calculate the accuracy.

For the result of dataset9 shown in Figure 6, which was generated by 8-gram modeling and in a distinguish mode, we could calculate the accuracy that was $13/15 = 86.7\%$; besides, the total time was about 45 minutes.

# 4 Empirical Studies

In this section, we showed empirical studies on datasets for analyses. As we stated before, we applied our approaches to nine different datasets, which were shown in Figure 7.

| 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 |

図 7: 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 and dataset7 to further explain the approaches described in Section 3 and to show what or how we thought during the study. We analyzed some factors that had an influence on the experiment results as well. Each study was presented as the following subsections.

## 4.1 Empirical Study on Dataset6

We applied all the approaches described in Section 3 to dataset6. Since to list all of the results was too difficult, we just picked part of them in some particular conditions.

Dataset6 had 16 different product variants and its size was 229.8 MB and the programming language was C. It was different from other datasets that it had two starter vertexes in the existing evolution history as shown in Figure 8.

In Figure 8, it was easy to find that both NetBSD-0.8 and 4.4BSD-lite were starter vertexes in evolution history. 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 a wrong edge. On the other hand, we had no idea about how to solve this problem.

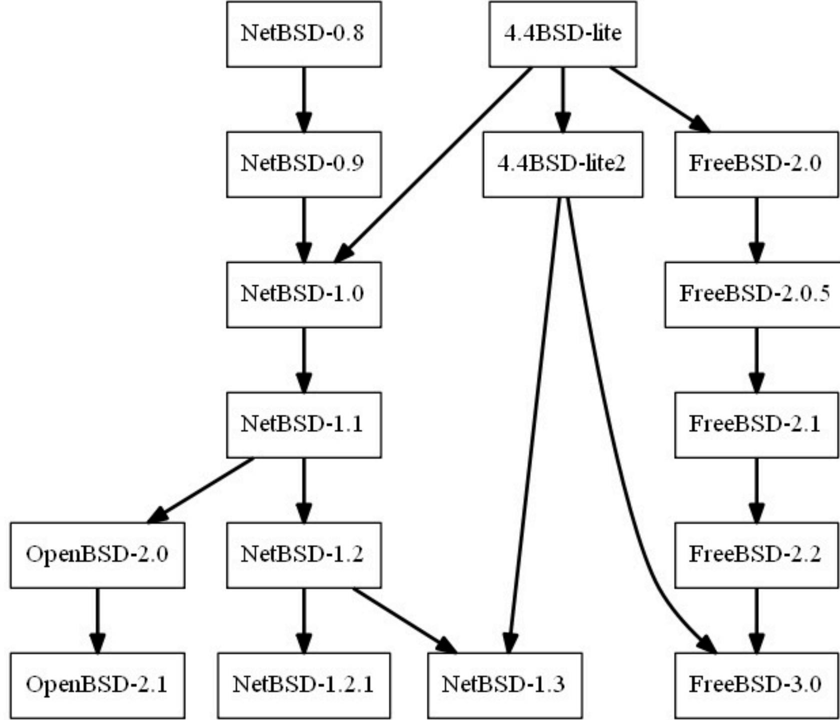


図 8: The actual evolution history of dataset6

Thus we decided to mark that edge as a special one if it turned to be wrong after extracting.

### 4.1.1 Results

As we stated before, since to list all of the results was too difficult, we just picked part of them in some conditions. In Figure 9, the second column described which product variant was selected as the starter vertex, which was from the actual evolution history in Figure 8. The third 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.

Instead of caring about the accuracy, Figure 10 focused on the speed of the whole experiments. The last column described the speed which was like xx MB/s, and the speed

| Name | Starter(Original Version) | n-gram | distinguish/ignore | Reverse Edges/Proper Edges | Wrong Edges | Special Edges | Accuracy (Proper/Total) |
|---|---|---|---|---|---|---|---|
| dataset6 | netbsd-0-8 | 3 | distinguish | 3/8 | 7 | 0 | 53.3% |
| | | 5 | | 1/8 | 6 | 1 | 57.1% |
| | | 8 | | 1/8 | 6 | 1 | 57.1% |
| | | 10 | | 2/9 | 5 | 1 | 64.3% |
| | | 12 | | | | | |
| | | 15 | | | | | |
| | | 20 | | | | | |
| | | 30 | | | | | |
| | | no | - | 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% |

図 9: Part1 of the experiment results of dataset6

| Name | Size | Language | Total Variants | Starter(Original Version) | n-gram | distinguish/ignore | Time(N-gram, Hash + LC = Total) | Speed = Size/Time (mb/s) |
|---|---|---|---|---|---|---|---|---|
| dataset6 | 229.8mb | c | 16 | netbsd-0-8 | 3 | distinguish | 2.7min + 3s = 2.7min | 1.42 mb/s |
| | | | | | 5 | | 4.3min + 3s = 4.3min | 0.89 mb/s |
| | | | | | 8 | | 3.9min + 3s = 3.9min | 0.98 mb/s |
| | | | | | 10 | | 8.1min + 3s = 8.1min | 0.47 mb/s |
| | | | | | 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 |
| | | | | lite | 3 | distinguish | - | - |
| | | | | | 5 | | - | - |
| | | | | | 8 | | - | - |
| | | | | | 10 | | - | - |
| | | | | | 12 | | - | - |
| | | | | | 15 | | - | - |
| | | | | | 20 | | - | - |
| | | | | | 30 | | - | - |
| | | | | | no | - | - | - |

図 10: Part2 of the experiment results of dataset6

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 as described in Section 3.5, we did large-scale oriented optimization by saving bitmaps generated by hashing in memory. Furthermore, actually the hashing itself took very little time and it was very difficult to record it in seconds. Besides, it was simple to find that the time Linear Counting took was also very little.

In terms of the time between different starter vertexes, because we only recorded the time of n-gram modeling, hashing and Linear Counting and there was no difference between these operations when starter vertexes changed, they were the same number in Figure 10. 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.

The same as before, we just listed part of the experiment results in some conditions. Besides, we would like to show the best results of the Evolution Tree we extracted as below.

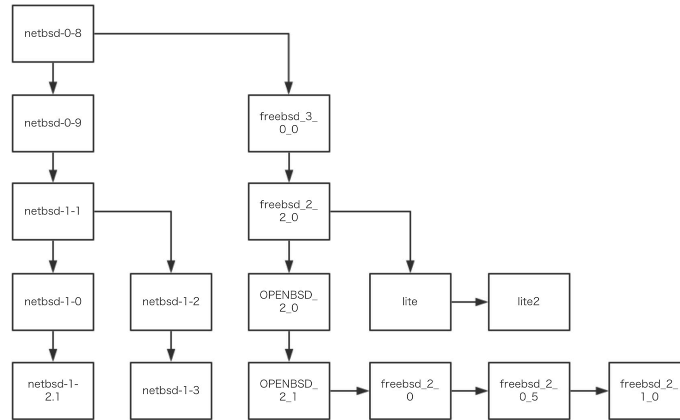


図 11: The Evolution Tree we extracted when starting with NETBSD-0-8

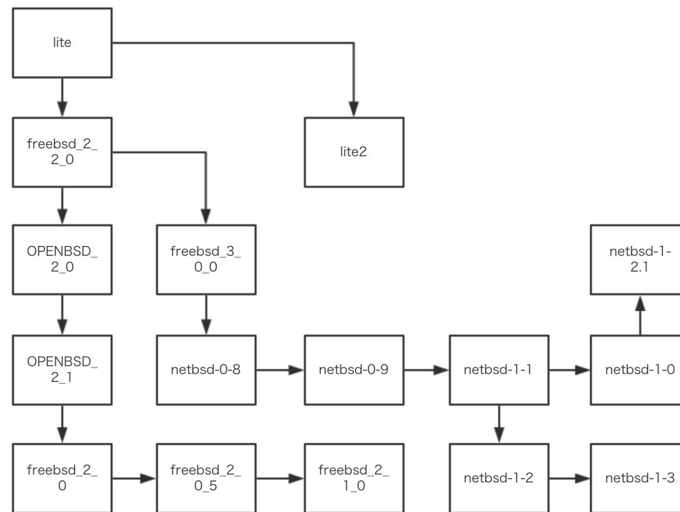


図 12: The Evolution Tree we extracted when starting with LITE

Since there were two starter vertexes and we could only begin with one starter vertex, we had to extract the Evolution Tree based on both of them, which was shown in Figure 11 and Figure 12. There was a detailed description about this problem in Section 4.1.3. To talk about it from the figures was not convenient, we would also show the result tables that described the Evolution Tree in Section 4.1.3 as well.

#### 4.1.2 Analysis on N-gram Modeling

It was easy to find that both in Figure 9 and Figure 10, 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, it was simple to learn from the column of Time in Figure 10. We also explained it before, the hashing itself took very little time and it was very difficult to record it in seconds. The hashing algorithm we selected was MurmurHash3, and its best benefit was exactly the speed. MurmurHash3 was non-cryptographic, so it could be much faster than traditional hash algorithms. Thus the time of n-gram modeling and hashing from the column of time was almost the time that n-gram modeling took, which was most of the total time.

On the other hand, the time of Linear Counting, which included calculating intersection and union, counting the number of 0 bits in bitmaps, calculating Jaccard Index, was also very little in seconds. Most of the calculating of Linear Counting was to deal with the bits, which was very familiar to the computer, so did the hashing. However, n-gram modeling was to deal with multisets of String, which was not familiar to the computer. As a result, n-gram modeling took most of the total time.

Well, how about the accuracy? Before we performed formal experiments, we made lots of tests to find out the influence of some parameters which included the number of n of n-gram modeling on experiment results.

In Figure 13, there was a description of the influence of the number of n of n-gram modeling on cardinality estimating by Linear Counting. After some testing experiments, we found that the most important factors which affected the accuracy of cardinality estimating were hashing algorithm and the size of bitmaps we set up. There would be additional discussion on this problem in Section 5.2, so we just skipped it here.

From Figure 13, it was clear 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, which was shown in Figure 13. Besides, if any product became " Bigger " or more complex, the intersection between a pair of product variants might become smaller and the union between a pair of product variants should

| Ngram | Actual Cardinality A | Estimated Cardinality A | Error A |
|---|---|---|---|
| 1 | 1680 | 1680.0110250897121 | 6.562553400071703E-6 |
| 2 | 13315 | 13313.692376009243 | 9.820683370310461E-5 |
| 3 | 72325 | 72324.42905460631 | 7.894163756474216E-6 |
| 4 | 207974 | 207965.8530563335 | 3.917289500854222E-5 |
| 5 | 410989 | 410978.07168395934 | 2.6590288403492668E-5 |
| 6 | 657506 | 657509.859059507 | 5.869238466306419E-6 |
| 7 | 916688 | 916714.8575354577 | 2.929844773543912E-5 |
| 8 | 1164125 | 1164101.4744668228 | 2.020876896999153E-5 |
| 9 | 1383782 | 1383845.6961313684 | 4.6030466770344264E-5 |
| 10 | 1564845 | 1564804.0310890744 | 2.6180810831459968E-5 |
| 11 | 1707162 | 1707269.361989865 | 6.28891633395539E-5 |
| 12 | 1813572 | 1813497.311077958 | 4.1183323321063135E-5 |
| 13 | 1887701 | 1887651.6926730585 | 2.6120305568264313E-5 |
| 14 | 1933693 | 1933866.4536708381 | 8.970072852213318E-5 |
| 15 | 1956871 | 1956909.014488023 | 1.9426159426455442E-5 |
| 16 | 1961612 | 1961581.9973538248 | 1.5294893269011995E-5 |
| 17 | 1951678 | 1951552.860872203 | 6.411873669579791E-5 |
| 18 | 1930898 | 1931016.7543211118 | 6.15021203149166E-5 |
| 19 | 1901417 | 1901271.7922725766 | 7.636816512285491E-5 |
| 20 | 1865330 | 1865245.6210829152 | 4.523538306078496E-5 |
| 21 | 1824140 | 1824057.2990250522 | 4.533696698048007E-5 |
| 22 | 1778890 | 1778944.8231667483 | 3.081875031523927E-5 |
| 23 | 1730685 | 1730511.388902498 | 1.0031351603674644E-4 |
| 24 | 1680158 | 1680021.199912026 | 8.142096634598717E-5 |
| 25 | 1627932 | 1628134.0026623344 | 1.2408544234916342E-4 |

図 13: The influence of the number of n of n-gram modeling on cardinality estimating

become larger. As a result, the Jaccard Index, which was defined as the product similarity, 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 9, although the n became larger after n=10, which was also equiv-

alent 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? The fact was we got lower product similarities than n-gram modeling, which was shown in Figure 14.

| | OPENBSD_2_0_BASE | OPENBSD_2_1_BASE | freebsdRELEASE_2_0 | freebsdRELENG_2_0_5_RELEASE | freebsdRELENG_2_1_0_RELEASE |
|---|---|---|---|---|---|
| OPENBSD_2_0_BASE | 1 | 0.6818987264228135 | 0.6201918002189669 | 0.36076564188580373 | 0.19015949899670917 |
| OPENBSD_2_1_BASE | - | 1 | 0.87941215946281 | 0.48610874444670055 | 0.25163881702165336 |
| freebsdRELEASE_2_0 | - | - | 1 | 0.5422380350007516 | 0.2737222041673952 |
| freebsdRELENG_2_0_5_RELEASE | - | - | - | 1 | 0.4383832230653385 |
| freebsdRELENG_2_1_0_RELEASE | - | - | - | - | 1 |

| | OPENBSD_2_0_BASE | OPENBSD_2_1_BASE | freebsdRELEASE_2_0 | freebsdRELENG_2_0_5_RELEASE | freebsdRELENG_2_1_0_RELEASE |
|---|---|---|---|---|---|
| OPENBSD_2_0_BASE | 1 | 0.6835087306799571 | 0.6144655332623513 | 0.35141639071611575 | 0.18473397236977482 |
| OPENBSD_2_1_BASE | - | 1 | 0.8670625358898247 | 0.4794831946380419 | 0.24597428992401696 |
| freebsdRELEASE_2_0 | - | - | 1 | 0.5467509581226984 | 0.27029000133988734 |
| freebsdRELENG_2_0_5_RELEASE | - | - | - | 1 | 0.4258794124386262 |
| freebsdRELENG_2_1_0_RELEASE | - | - | - | - | 1 |

図 14: Part of product similarities generated from no n-gram modeling (up) and n-gram (n=30) modeling (down)

As we stated before, the product similarities and the number n were negatively correlated. That meant the results generated from no n-gram modeling (up) was a kind of n-gram modeling where n $\rightarrow \infty$. 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 $\rightarrow \infty$, 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.

#### 4.1.3 Analysis on Starter Vertex

As shown in Figure 9 and Figure 10, the best results of beginning with different starter vertexes were almost the same. To expand on it, the result tables that described the Evolution Tree were shown in Figure 15.

The yellow was still for the wrong edges while the blue for the reverse edges. The red was for the special edges described in Section 4.1. Existing special edges 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 a wrong edge. Thus we decided

| StartVertex | EndVertex | StartVertex | EndVertex |
|---|---|---|---|
| netbsd-0-8 | netbsd-0-9-RELEASE | lite | lite2 |
| netbsd-0-9-RELEASE | netbsd-1-1-RELEASE | lite | freebsdRELENG_2_2_0_RELEASE |
| netbsd-1-1-RELEASE | netbsd-1-0-RELEASE | freebsdRELENG_2_2_0_RELEASE | OPENBSD_2_0_BASE |
| netbsd-1-1-RELEASE | netbsd-1-2-RELEASE | OPENBSD_2_0_BASE | OPENBSD_2_1_BASE |
| netbsd-1-2-RELEASE | netbsd-1-3-RELEASE | OPENBSD_2_1_BASE | freebsdRELEASE_2_0 |
| netbsd-1-0-RELEASE | netbsd-1-2-PATCH001 | freebsdRELEASE_2_0 | freebsdRELENG_2_0_5_RELEASE |
| netbsd-0-8 | freebsdRELENG_3_0_0_RELEASE | freebsdRELENG_2_0_5_RELEASE | freebsdRELENG_2_1_0_RELEASE |
| freebsdRELENG_3_0_0_RELEASE | freebsdRELENG_2_2_0_RELEASE | freebsdRELENG_2_2_0_RELEASE | freebsdRELENG_3_0_0_RELEASE |
| freebsdRELENG_2_2_0_RELEASE | OPENBSD_2_0_BASE | freebsdRELENG_3_0_0_RELEASE | netbsd-0-8 |
| OPENBSD_2_0_BASE | OPENBSD_2_1_BASE | netbsd-0-8 | netbsd-0-9-RELEASE |
| OPENBSD_2_1_BASE | freebsdRELEASE_2_0 | netbsd-0-9-RELEASE | netbsd-1-1-RELEASE |
| freebsdRELEASE_2_0 | freebsdRELENG_2_0_5_RELEASE | netbsd-1-1-RELEASE | netbsd-1-0-RELEASE |
| freebsdRELENG_2_0_5_RELEASE | freebsdRELENG_2_1_0_RELEASE | netbsd-1-1-RELEASE | netbsd-1-2-RELEASE |
| freebsdRELENG_2_2_0_RELEASE | lite | netbsd-1-2-RELEASE | netbsd-1-3-RELEASE |
| lite | lite2 | netbsd-1-0-RELEASE | netbsd-1-2-PATCH001 |

図 15: The result tables of different start vertexes that described the Evolution Tree (dataset6, the best configuration)

to mark that edge as a special one if it turned to be wrong after extracting the trees. Moreover, as we stated before, the reverse edges should be treated as proper edges when calculating the accuracy.

It was very simple to find that not all the wrong edges in one table were equivalent to those in the other table. 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.

### 4.2 Empirical Study on Dataset7

The same as before, we applied all the approaches described in Section 3 to dataset7. Since to list all of the results was too difficult, we just picked part of them in some particular conditions.

Dataset7 had 37 different product variants and its size was 276.7 MB and the programing language was Java. The existing actual evolution history was as shown in Figure 16.

In Figure 16, we could find several vertexes that introduced a new raw of vertexes, such as GROOVY210BETA1, GROOVY220BETA1, GROOVY201, and so on. Actually to find edges that included these vertexes were more likely difficult to extract than others because each of these vertexes had several edges, which meant there could be similar pairs that might be extracted.

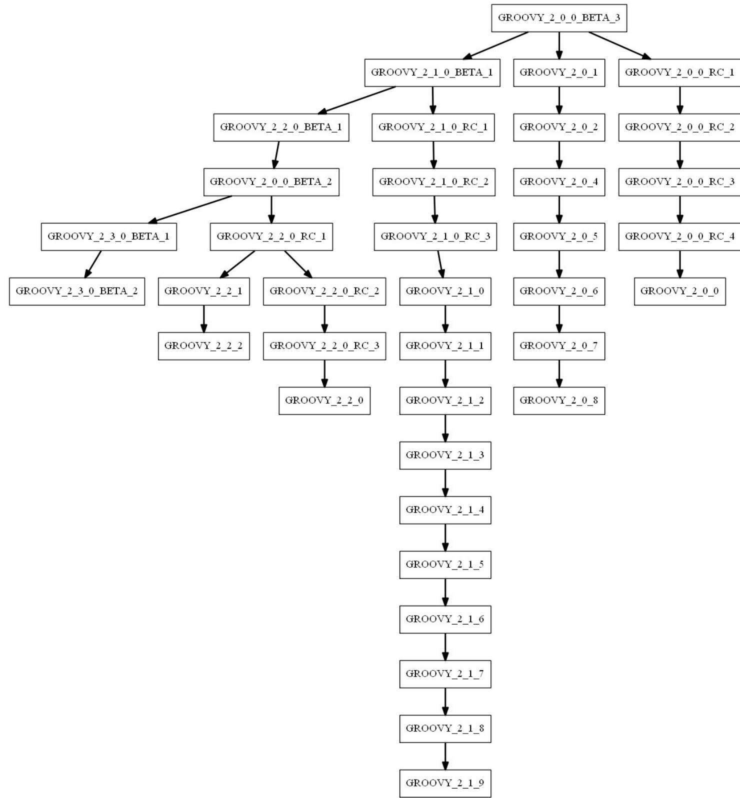


図 16: The actual evolution history of dataset7

### 4.2.1 Results

Part of the results was as shown in Figure 17. The way we recorded these results were the same as dataset6. Thus we would not explain it here again.

### 4.2.2 Supplementary Analysis on N-gram Modeling

We thought if we did not apply n-gram modeling, the result might be the same as unigram (n=1). However, the experiment results were totally different. Until that time, we started to analyze n-gram modeling and in reality, this supplementary was the beginning of the analysis.

What if we applied unigram modeling to any line of code? The result should be like word by word in a multiset. However, if we did not apply n-gram modeling to the line of code, the result should be like sentence by sentence in a multiset. That was it! That was why we believed that the results generated from no n-gram modeling were a kind of n-gram modeling where n $\rightarrow \infty$.

| Name | 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) |
|---|---|---|---|---|---|---|---|---|
| dataset7 | 1 | distinguish | 75s + 16s = 91s | 3.04 mb/s | 4/12 | 24 | 0 | 33.3% |
| | 3 | distinguish | 150s + 16s = 166s | 1.67 mb/s | 2/27 | 9 | 0 | 75% |
| | 3 | ignore | 58s + 17s = 75s | 3.69 mb/s | 2/27 | 9 | 0 | 75% |
| | 5 | distinguish | 201s + 17s = 218s | 1.27 mb/s | 2/28 | 8 | 0 | 77.8% |
| | 5 | ignore | 130s + 16s = 146s | 1.90 mb/s | 2/28 | 8 | 0 | 77.8% |
| | 8 | distinguish | 340s + 16s = 356s | 0.78 mb/s | 2/28 | 8 | 0 | 77.8% |
| | 10 | | 210s + 16s = 226s | 1.22 mb/s | | | | |
| | 12 | | 245s + 16s = 261s | 1.06 mb/s | | | | |
| | 15 | | 252s + 15s = 267s | 1.04 mb/s | | | | |
| | 20 | | 348s + 15s = 363s | 0.76 mb/s | | | | |
| | 30 | | 329s + 16s = 345s | 0.80 mb/s | | | | |
| | no | - | 5s + 16s = 21s | 13.18 mb/s | 2/28 | 8 | 0 | 77.8% |

図 17: Part of the experiment results of dataset7

It could also be learned from Figure 17 that the results from the distinguish mode and those from the ignore mode were quite the same. In fact, we also checked the tables described the Evolution Tree and they were also the same as each other. This could be the evidence to the description in Section 4.1.2 that to distinguish different product variants was not corresponding to n-gram modeling. In other words, we might extract the same Evolution Tree from different kinds of n-gram modeling.

It was also able to find that the speed increased when we applied the ignore mode of n-gram modeling, compared to the distinguish mode of the same number of n of n-gram modeling. Although there were not plenty of experiments to be taken as the evidence

We might explain it in theory as well. Because when we applied the distinguish mode of n-gram modeling, the initial multisets after initialization became “ bigger ” or more complex. Obviously, the more distinct elements in the initial multisets, the more cardinality estimated by the Linear Counting algorithm. That meant, we needed to do more calculating in the distinguish mode than we did in the ignore mode. Thus the speed slowed down and the time increased. On the other hand, this could be also the evidence to the description in Section 4.1.2 as well.

### 4.2.3 Analysis on Complex Vertex

As we stated in Section 4.2, there were several vertexes that were more likely difficult to deal with than the others, because each of them introduced a new raw of vertexes. On the other hand, to find out edges that included these vertexes were complex as well because each of these vertexes had several edges, which meant there could be similar pairs that might be extracted.

Part of the result table generated by the best configuration that described the Evolution Tree was shown in Figure 18.

Since there were 37 product variants in dataset7, to list all of them was too hard,

| StartVertex | EndVertex | Sim |
|---|---|---|
| **GROOVY_2_0_0_RC_1** | GROOVY_2_0_0_RC_3 | 0.9973322014670906 |
| **GROOVY_2_0_0_RC_3** | GROOVY_2_0_0 | 0.9948901980121734 |
| **GROOVY_2_0_0** | GROOVY_2_0_1 | 0.982759281100505 |
| **GROOVY_2_0_6** | GROOVY_2_1_0_BETA_1 | 0.9568273373675054 |
| **GROOVY_2_1_6** | GROOVY_2_2_0_BETA_1 | 0.9853784874168485 |
| **GROOVY_2_2_0_RC_2** | GROOVY_2_2_0 | 0.9984306593921771 |
| **GROOVY_2_2_0** | GROOVY_2_2_1 | 0.9973087604171575 |
| **GROOVY_2_2_2** | GROOVY_2_3_0_BETA_1 | 0.9423697577436071 |

図 18: Part of the result table that described the Evolution Tree (dataset7, the best configuration)

and we selected part of the result table that described the Evolution Tree by the best configuration. All the edges described in Figure 18 were yellow, which meant they were wrong edges.

Comparing this table to the actual evolution history in Figure 16, we could find only one edge did not include the complex vertexes which were (GROOVY200RC3, GROOVY200), and all the others either began with one complex vertex or ended at one. It confirmed the hypothesis, and after extra experiments, we found other datasets were almost the same. The Evolution Tree extracted from those datasets without these complex vertexes could have much higher accuracy. The more complex vertexes the actual history had, the more wrong edges there were in the Evolution Tree extracted.

We considered the reason in theory as well. In our approaches described in Section 3, the edges that we found were the most similar pair in the whole product variants. We followed the intuition that similar software products must have similar source code and we estimated the product similarity.

However, were the edges in the existing actual evolution history exactly the most similar pair in the whole product variants? It was sure that we could believe most of the situations were near the same, but it was not sure that there were not particular situations. In fact, the reasons that one software product variant was similar to another one could be various. Maybe these two product variants were not a consequent pair but a pair of a sequence raw in the actual evolution history, which was the same as the edge (GROOVY200RC3, GROOVY200). The modification from GROOVY200RC4 to GROOVY200 might be near a reverse modification from GROOVY200RC3 to GROOVY200RC4, which made the error

occurred.

On the other hand, there could be much other relation between the complex vertexes and the edges that included these vertexes. Thus the more complex vertexes the actual history had, the more wrong edges there were in the Evolution Tree extracted.

However, since our approaches were only based on similarities between pairs of product variants, we could not solve this problem. Only if we changed the idea into another method, we could find out why these errors occurred around the complex vertexes. They might be the area of NLP we thought. By the way, the Evolution Tree extracted itself was an approximation, and we naturally could not get a hundred percent result.

## 5 Discussion

In this section, we would discuss study approaches we took in detail and some problems we met during the research. We would begin with the field of cardinality estimation where there existed not only the Linear Counting algorithm but also some others, and we would like to introduce the difference between them. Then we would discuss the influence of major factors that affected the experiment results which were compared with the actual evolution history. The discussion was based on all the empirical studies while some of them were not involved in Section 4. Next, we would like to talk about the problem of directions. After that, we summarized the best configuration that could reach the best result. Finally, we talked about the threats to validity as well.

### 5.1 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[9]. Various algorithms had been proposed in the past, such as Linear Counting, LogLog Counting, HyperLogLog Counting, and so on. Most of these algorithms were to solve the memory requirement problem by estimating the result rather than directly calculating it.

Since there were plenty of algorithms that were proposed, many experiments had been developed to seek the difference between these algorithms. Zhang[22] 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. Zhang made a study of the results generated by five algorithms from different cardinality numbers of initial input multisets. He found that Linear Counting should be the best choice for those input multisets whose cardinality numbers were not too large and LogLog Counting, on the other hand, could be the choice for the other initial multisets whose cardinality numbers were not too small.

Another experiment developed by Heule[9] 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 shown in Figure 19 still proved that for small cardinalities, the Linear Counting algorithm (the green line) was still better than the improved HyperLogLog Counting algorithm (the blue line). Finally, he determined the intersection of the error curves of the bias-corrected raw estimate and the Linear Counting algorithm to be at some point, and use Linear Counting to the left of that threshold.

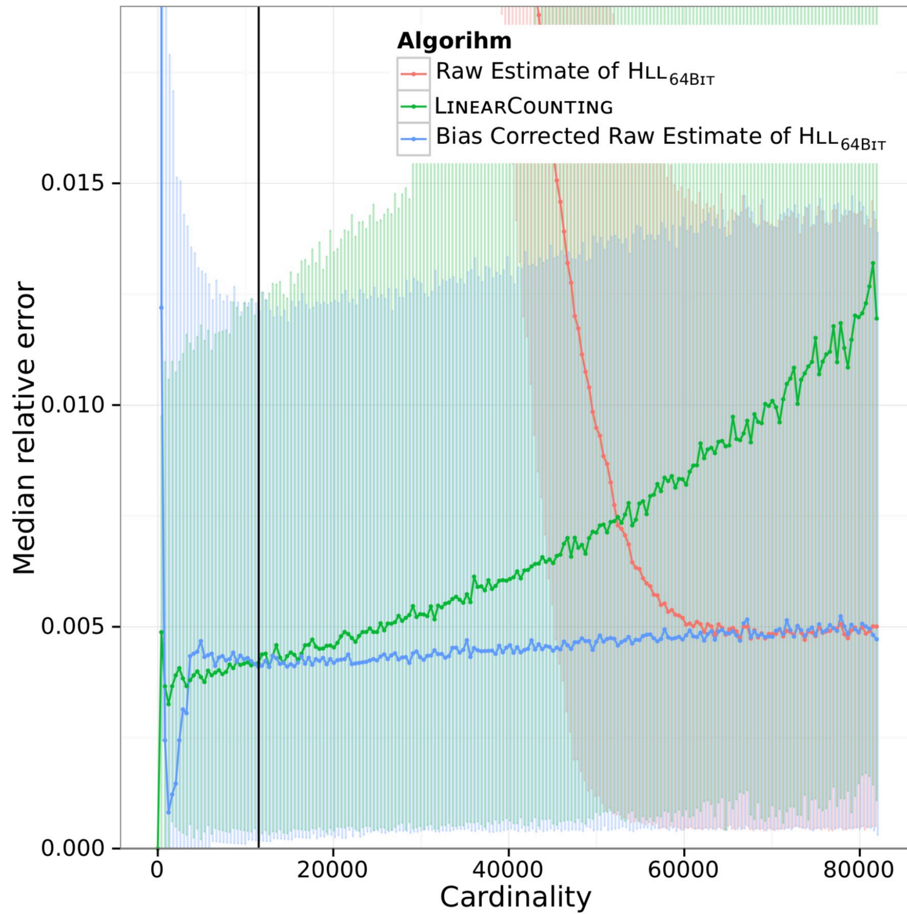


図 19: The median error of the raw estimate, the bias-corrected raw estimate, as well as Linear Counting

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. For future work, if the sizes become much larger, maybe we would better change into another algorithm to reach a better result.

### 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. 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. Because to apply a larger n of n-gram modeling meant more calculating and it would take more time and memory, we determined not to apply n-gram modeling to any line of code. 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 $n \to \infty$.

#### 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 select a hashing algorithm which was best to generate a uniform result and had a high speed.

At first, the hashing algorithm we selected was Java.hashcode(). Actually, it worked not so bad that we had to change it. However, once we learned that Java.hashcode() could only return 32 bits value, we started to consider that maybe we would deal with some situations of 64 bits in the future. Thus we decided to find another possible hashing algorithm. Before that, we had to learn about the differences between different hashing algorithms.

Earlz[7] 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[16] that produced exactly the same hash values as the official final C++ version.

By the way, Java.hashcode() was based on a prim number 31, which could change multiplication into shift operation and subtraction. After some other optimizations, the performance could be quite good. So why did we reject it?

There were two reasons. The first one was about the performance. Many tests[23] still showed that the capability of MurmurHash was still better than Java.hashcode() especially when dealing with large-scale datasets. The second reason was that the difference between results from different elements was not intense enough by Java.hashcode(). On the other hand, Murmurhash showed the best answer to that difference. Thus finally we chose MurmurHash3.

### 5.2.3 Bitmap Size

Another important factor was the bitmap size that we set up. From advanced analysis by Whang[21], we could learn that there was a direct relationship between bitmap size m and cardinality number n. In Figure 20, n meant the cardinality number and m meant the size of the bitmap that we set up. “ .01 ” or “ .10 ” meant the precision of the error. It was easy to see that 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

| | Map size $m$ epsilon | | | Map size $m$ epsilon | |
|---|---|---|---|---|---|
| $n$ | .01 | .10 | $n$ | .01 | .10 |
| 100 | 5034 | 80 | 80000 | 23029 | 10458 |
| 200 | 5067 | 106 | 90000 | 24897 | 11608 |
| 300 | 5100 | 129 | 100000 | 26729 | 12744 |
| 400 | 5133 | 151 | 200000 | 43710 | 23633 |
| 500 | 5166 | 172 | 300000 | 59264 | 33992 |
| 600 | 5199 | 192 | 400000 | 73999 | 44032 |
| 700 | 5231 | 212 | 500000 | 88175 | 53848 |
| 800 | 5264 | 231 | 600000 | 101932 | 63492 |
| 900 | 5296 | 249 | 700000 | 115359 | 72997 |
| 1000 | 5329 | 268 | 800000 | 128514 | 82387 |
| 2000 | 5647 | 441 | 900000 | 141441 | 91677 |
| 3000 | 5957 | 618 | 1000000 | 154171 | 100880 |
| 4000 | 6260 | 786 | 2000000 | 274328 | 189682 |
| 5000 | 6556 | 948 | 3000000 | 386798 | 274857 |
| 6000 | 6847 | 1106 | 4000000 | 494794 | 357829 |
| 7000 | 7132 | 1261 | 5000000 | 599692 | 439233 |
| 8000 | 7412 | 1412 | 6000000 | 702246 | 519429 |
| 9000 | 7688 | 1562 | 7000000 | 802931 | 598645 |
| 10000 | 7960 | 1709 | 8000000 | 902069 | 677040 |
| 20000 | 10506 | 3105 | 9000000 | 999894 | 754732 |
| 30000 | 12839 | 4417 | 10000000 | 1096582 | 831809 |
| 40000 | 15036 | 5680 | 50000000 | 4584297 | 3699768 |
| 50000 | 17134 | 6909 | 100000000 | 8571013 | 7061760 |
| 60000 | 19156 | 8112 | 120000000 | 10112529 | 8373376 |
| 70000 | 21117 | 9294 | | | |

図 20: The relationship between bitmap size m and cardinality number n

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[10] 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 4.1.3 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[16] 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. The lines of code were formed by the programming language. There might be different choices in different kinds of programing language. This time we applied our approach to datasets of Java and C and we got a good result. Next time if the programming language changed, would the result still be good? 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. We were able to extract the Evolution Tree only if we knew the starter vertex (the original version). We had to treat the reverse edges as proper edges when calculating the accuracy. Once there was a way to solve the problem of directions, all of these became 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. Although we could use a larger bitmap to decrease this defect, the memory was limited. 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, and there might be no hashing algorithms that could generate uniformly distributed results. At that time, we had to change the whole approaches into other ones. Moreover, we also had to change the approaches when we found out other better methods to define similarities between pairs of product variants.

# 6 Conclusion and Future Work

In this research, we proposed an efficient approach to estimate the similarities between pairs of software product variants, and we extracted the Evolution Tree by connecting the pairs that shared the highest similarity. With the proposed approaches we did plenty of empirical experiments, and we summarized the best configuration which worked out the best result. We also discussed the influence of various parameters on the experiments based on the empirical studies. 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 if it existed. We will deal with other kinds of programming language to confirm whether the language matters or not. To learn about the knowledge of NLP (Natural Language Processing) 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.

## 謝辞

There is no denying that it is a tremendous challenge for me, a foreign student, to study and do research in an unfamiliar land. However, thanks to numerous people’s tireless efforts, my overseas life has eventually come to a happy end.

My first acknowledgements should go to my supervisor Prof. Katsuro Inoue, who has ever taught me throughout the two years ’overseas life. Without his careful and responsible teaching, I would not have acquired the knowledge needed for this thesis.

My heartfelt gratitude also goes to the members of my research group: Prof. Takashi Ishio and Prof. Tetsuya Kanda, who have taught me, in detail, how to prepare and develop the research, and whose preciseness and passion in study and research impressed me a lot. Owing to their insightful guidance and comment on my thesis as well as patient revising, the completion of this thesis is made possible.

I am particularly indebted to Mrs. Mizuho Karube, who is always so kind to me, and she has given me a lot of guidance, assistance, and concern.

No less gratitude should be given to all the members of my laboratory for their two years ’ constant support, without which I would fail in getting used to the life in Japan.

Finally, I would like to express my sincere thanks to my dear friends: Mr. Wu Yuhao, Mr. Qiu Shi, and Mr. Boris Genadiev Todorov, who have accompanied me throughout the two years, sharing sadness and happiness with me and teaching me a lot.

## 参考文献


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

[2] Tibor Bakota. Tracking the evolution of code clones. In *International Conference on Current Trends in Theory and Practice of Computer Science*, pages 86–98. Springer, 2011.

[3] Andrei Z Broder, Steven C Glassman, Mark S Manasse, and Geoffrey Zweig. Syntactic clustering of the web. *Computer Networks and ISDN Systems*, 29(8-13):1157–1166, 1997.

[4] Gerardo Canfora, Luigi Cerulo, and Massimiliano Di Penta. Identifying changed source code lines from version repositories. In *MSR*, volume 7, page 14, 2007.

[5] Cagatay Catal, Serkan Tugul, and Basar Akpinar. Automatic software categorization using ensemble methods and bytecode analysis. *International Journal of Software Engineering and Knowledge Engineering*, 27(07):1129–1144, 2017.

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

[7] 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*, 2011.

[8] Javier Escobar-Avila, Mario Linares-Vásquez, and Sonia Haiduc. Unsupervised software categorization using bytecode. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, pages 229–239. IEEE Press, 2015.

[9] Stefan Heule, Marc Nunkesser, and Alexander Hall. Hyperloglog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm. In *Proceedings of the 16th International Conference on Extending Database Technology*, pages 683–692. ACM, 2013.

[10] Tetsuya Kanda, Takashi Ishio, and Katsuro Inoue. Approximating the evolution history of software from source code. *IEICE TRANSACTIONS on Information and Systems*, 98(6):1185–1193, 2015.

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

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

[13] David Lorge Parnas. Software aging. In *Software Engineering, 1994. Proceedings. ICSE-16., 16th International Conference on*, pages 279–287. IEEE, 1994.

[14] Steven P Reiss. Tracking source locations. In *Proceedings of the 30th international conference on Software engineering*, pages 11–20. ACM, 2008.

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

[16] Yonik Seeley. Murmurhash3. *https://github.com/yonik/java_util/blob/master/src/util/hash/MurmurHash3.java*, 2015.

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

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

[19] Francisco Servant and James A Jones. Fuzzy fine-grained code-history analysis. In *Proceedings of the 39th International Conference on Software Engineering*, pages 746–757. IEEE Press, 2017.

[20] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *ACM sigsoft software engineering notes*, volume 30, pages 1–5. ACM, 2005.

[21] Kyu-Young Whang, Brad T Vander-Zanden, and Howard M Taylor. A linear-time probabilistic counting algorithm for database applications. *ACM Transactions on Database Systems (TODS)*, 15(2):208–229, 1990.

[22] Zhang Yang. Effect experiments and practical proposals of five common cardinality estimation algorithms. *http://blog.codinglabs.org/articles/cardinality-estimate-exper.html*, 2013.

[23] Zhang Yang. Interpret cardinality estimation algorithm (part 2: Linear counting). *http://blog.codinglabs.org/articles/algorithms-for-cardinality-estimation-part-ii.html*, 2013.