Estimating Product Evolution Graph using Kolmogorov Complexity

Yasuhiro Hayase
Division of Information Engineering
University of Tsukuba
Tsukuba, Japan
hayase@cs.tsukuba.ac.jp

Tetsuya Kanda
Graduate School of Information Science and Technology
Osaka University
Osaka, Japan
t-kanda@ist.osaka-u.ac.jp

Takashi Ishio
Graduate School of Information Science and Technology
Osaka University
Osaka, Japan
ishio@ist.osaka-u.ac.jp

ABSTRACT

This paper proposes a method of estimating a product evolution graph based on Kolmogorov complexity. The method *EEGL* applies lossless compression to the source code of products, then, presumes a derivation relationship between two products when the increase of information between the two products is small. An evaluation experiment confirms that EEGL and an existing method *PRET* tends to produce different errors when estimating evolution graph results.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering, Version control*; D.2.9 [**Software Engineering**]: Management—*Software configuration management*

General Terms

Algorithms

Keywords

Software evolution, evolution graph, estimation, Kolmogorov complexity, lossless compression

1. INTRODUCTION

When developing a new software product, an existing similar software product can be used as a *base product* – namely, a base product is copied and modified into the new product (i.e. *derived product*). Since a derived product shares source code with a base product, the derivation relationship should be recorded and managed in a consistent manner for the purpose of maintenance or aggregation of shared code. However, records of derivation are sometimes lost thorough software evolution process.

IWPSE'15, August 30, 2015, Bergamo, Italy
ACM. 978-1-4503-3816-5/15/08...$15.00
http://dx.doi.org/10.1145/2804360.2804368

Kanda *et al.* [11] proposed *PRET*, a method to estimate derivation relationships between multiple software products (i.e. evolution graph). PRET employs an original distance measure for two software products that counts pairs of two similar source files whose longest common subsequence (LCS) of tokens is longer than certain ratio, across the two products. PRET builds a spanning tree of products based on number of similar files between products. Then, direction of an edge that means direction of derivation is estimated using the number of the tokens not included in the LCS.

Although PRET achieved high accuracy of estimation, there is still a room for improvement on the following points:

(1) Only tokens in source files are used for calculating product similarity. In other words, neither comments in source files nor other type of files, such as manuals or build scripts, do not contribute to the similarity. These documents also evolve in a similar manner as source files. Therefore, the documents can be used for estimating product derivation.

(2) File similarity is based on LCS of tokens. In the case of a code fragment is moved, LCS does not include tokens in the moved fragments, then file similarity is get lower.

(3) Only number of tokens impact file similarity. Therefore, newness and complexity of tokens do not impact the similarity. For example, a completely new and unique code fragment and a code fragment made by copy-and-paste (code clone) have same impact. Likewise, a frequently appearing reserved word and a first appearance of long and complex string literal have same impact.

(4) If a source file is split into two segment files, both of the segment files are judged dissimilar to the origin file. Since the origin file have many tokens that are not included in each segmented file. Merging of source files involves same problem.

(5) (minor problem) To obtain tokens from source files, PRET need to recognize a programming language used for each source file.

To deal with the above issues, this paper proposes a new method **EEGL** (Estimating Evolution Graph using Lossless compression) to estimate a product evolution graph using increase of information in the sense of Kolmogorov complexity [20, 21, 13, 3]. Kolmogorov complexity is a complexity measure for a string; it is defined as the length of the

shortest program that output the string. Since Kolmogorov complexity is not computable for arbitrary strings, lossless compression methods are used as upper bound (e.g. approximate) function for Kolmogorov complexity. [5, 16, 12, 7] In our approach, divergence from product p to product q is measured as difference between the size of a compressed archive of p and a compressed archive of p with q.

EEGL solves the issues on PRET: 1) documents, Makefile, and comments in a source file is equally used for divergence calculation. 2) When a code fragment is moved, size of compressed archive gets slightly bigger. However, the increase must be small compare to moved fragment. 3) Since code clones are redundant, increase of compressed archive is small if a big code clone is added. New appearance of long complex string literal brings deserved increases to a compressed archive. 4) File split or merging without changing total contents has small impact to compressed size. 5) Proposed approach is language independent.

The rest of this paper is organized as follows. Section 2 describes detailed idea and procedure of our approach. In section 3, preliminary investigation of lossless compression tools is performed. Section 4 shows comparison with PRET using same data sets of evaluation experiment in [11]. Section 5 shows related work. Finally, conclusion and future work are presented in Section 6.

2. APPROACH

Proposed approach EEGL estimates evolution graph by comparing the compressed size of a single product and compressed size of merging of two different products. The approach is based on two basic ideas. First idea is that pair of products that are in direct derivation relationship should share more information than the pairs that are not in direct derivation relationship. This fact is confirmed by Arbuckle[1]. Second idea is that a derived product tends to have more quantity of information than a base product. It is known that values of size or complexity metrics for sequential releases have strong tendency to increase. [14, 15]

Procedure to estimate the evolution graph is described below:

Input Source code set of n products $P = \{p_1, p_2, ...p_n\}$ (p_i is source code of i-th product)

Output Directed Graph $G = (P, E)$. Edge set E means estimation of a product derivation relationship. Node set P is identical to input.

Step 1. Remove binary files (e.g. pictures) from each of $p_i (i = 1..n)$.

Step 2. For each $p_i (i = 1..n)$, make `tar` archive in order of path name, and then apply lossless compression to the tar archives. Define $Z(p_i)$ as compressed data size of p_i.

Step 3. For all pair of products (p_i, p_j) where $p_i, p_j \in P$, make a combined tar ball $p_i \cdot p_j$. Combined tar ball composed from all source files in both p_i and p_j. Order of files in tar ball is ascendant order of the path name except for top directory name. Example of file order is shown in table 1. This order is expected to help lossless compression algorithms to find similar parts in files, since files that have similar path names tend to have similar contents.

Table 1: Example of file orders in single and combined products

a	`a/0/1.c a/1.c a/4.c`
b	`b/0/2.c b/1.c b/3.c`
$a \cdot b$	`a/0/1.c b/0/2.c a/1.c b/1.c b/3.c a/4.c`

Step 4. Compress all combined tar balls. Define $Z(p_i \cdot p_j)$ as compressed data size of $p_i \cdot p_j$.

Step 5. Finally, make product derivation edges E based on $I(p_i, p_j) = Z(p_i \cdot p_j) - Z(p_i)$, i.e. *increase of information from p_i to p_j*, and following assumptions.

ASM1. Quantity of information of a derived product is larger than base product.

ASM2. Increase of information from true base product is small, compared to increase from other products such as ancestor or sibling products.

ASM3. Number of base product of each product is at most one.

Based on these assumptions, directed edge set E of estimated evolution graph $G = (P, E)$ is defined as follows.

$$(p, q) \in E \iff Z(p) < Z(q) \ \wedge \ (\forall r \in P, \quad Z(r) \geq Z(q) \ \vee \ I(r, q) \geq I(p, q))$$

This definition means that for all product q, make a edge from p to q when increase of information from p is smallest in the products whose compressed size is smaller than q.

3. PRELIMINARY INVESTIGATION: COMPARISON OF COMPRESSION TOOLS

To evaluate aptitude of lossless compression algorithms for estimating derivation relationships, we performed preliminary investigation of examining divergence of program versions and increase of compressed data sizes. Specifically, we compressed the all possible combined products of two different versions by several compression tools, and then observed compressed data size. Following three compression tools are compared: gzip (deflate (LZ77[23] + Huffman coding[9])), bzip2 (block sorting [2] + Huffman coding), xz (LZMA (Variant of LZ77 + Range Encoder[18])). All tools are executed with `-9` option, which controls compression ratio. Target software products and versions are 8.0.0 to 8.0.26 (totally 27 releases) of PostgreSQL[1] source code.

First of all, summary compression ratio of each single release is shown in Table 2. Order of average compression ratio is gzip > bzip2 > xz. This result confirms that newer compression tools achieve better compression ratio. On the other hand, this result also shows compression ratio of bzip2 is unstable, since stddev compare to average of compression ratio is bigger than others.

Next, we observed $\frac{Z(p \cdot q) - Z(p)}{Z(p)}$, that is to say, relative increase ratio of compressed data size when release q is added

[1] `http://www.postgresql.org/`

Table 2: Compression ratio for each single release

Tool	Compression Ratio (average ± stddev)	stddev/average
gzip -9	0.231 ± 0.000384	0.00166
bzip2 -9	0.182 ± 0.000368	0.00202
xz -9	0.163 ± 0.000233	0.00143

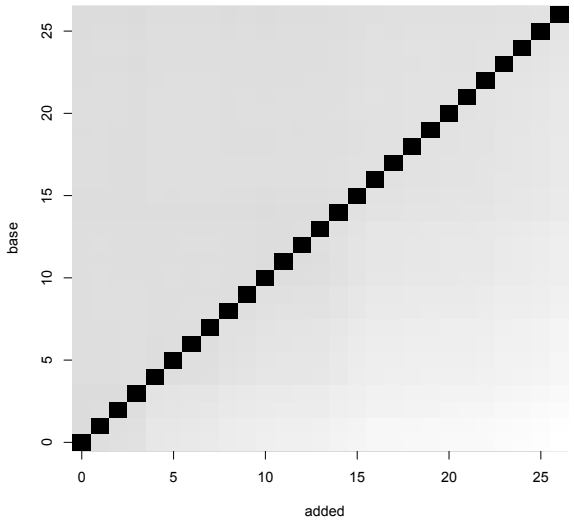

Figure 1: Heatmap of relative increase for gzip -9

to release p. Figure 1 to 4 are a heat map whose cells express the relative increase ratio. Vertical axis means minor release number (x of 8.0.x) of base product p. Horizontal axis means minor release number of added product q. Darker cells means lower relative increase; brighter cells means higher relative increase. Diagonal cells in the heatmaps is black that means very low (almost zeor) increase ratio, since the diagonal cells correspond to the case that two identical products are combined. Figure 1, 2, and 3 are cases of gzip, bzip2, and xz respectively, and employs 0.5 for gray gamma curve. Figure 4 is also the same case of gzip except that gamma value is 0.1, because Figure 1 is difficult to read subtle differences between cells.

If the heat map of a compression tool monotonically gets brighter from diagonal cells to right, that means the compression tool accurately catches increase of information. This is because difference of information contained in the source code of two releases should be bigger if the distance of the releases is far.

Let us consider the monotonicity of the three tools. Total brightness of Figure 1 means difference of relative increase ratio of gzip is smaller than other tools. However, we can see that monotonicity of gzip is fine from Figure 4, even though the difference is small. Dappling in Figure 2 reveals anti-monotonicity of bzip2. Figure 3 shows the best monotonicity of xz.

From the above observation, xz is expected to output the most accurate evolution graph, followed in order by gzip and bzip2.

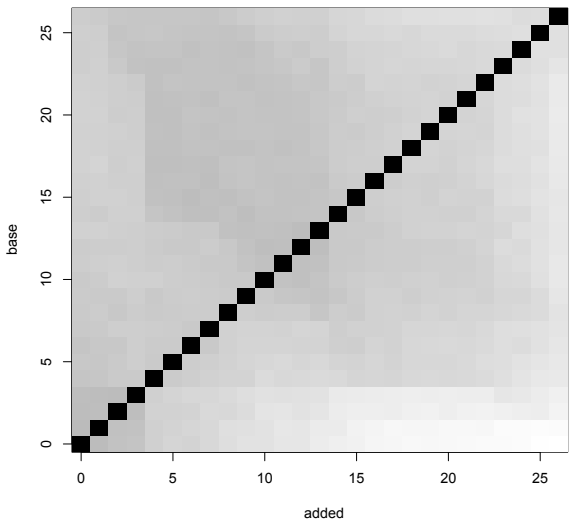

Figure 2: Heatmap of relative increase for bzip2 -9

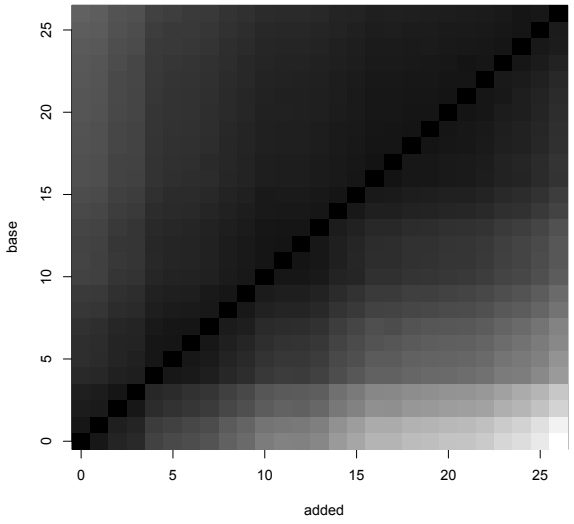

Figure 3: Heatmap of relative increase for xz -9

4. EVALUATION EXPERIMENT: COMPARISON WITH PRET

This section describes the comparison of EEGL with existing approach PRET, using the same data sets used to evaluated PRET. (data sets are available on our website [2]) Evaluation basis is accuracy of estimated evolution graphs. On executing PRET, its threshold parameter of file similarity is set to 0.9 where the best accuracy was achieved in the data sets. All the target projects are implemented in C.

Six data sets are designed to cover the following situation.

[2] `http://sel.ist.osaka-u.ac.jp/pret/`

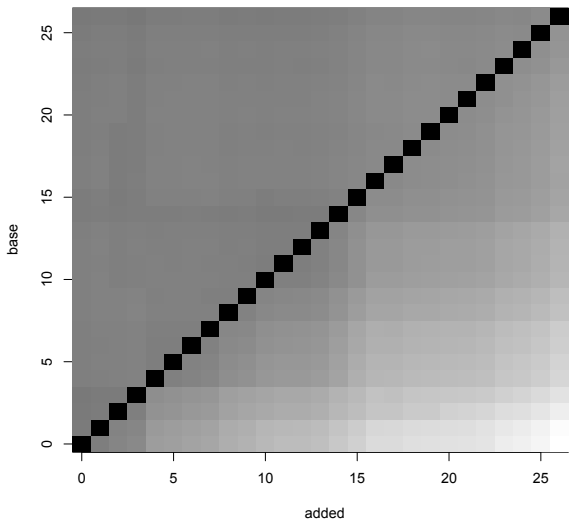

Figure 4: Heatmap of relative increase for gzip -9 (Gamma=0.1)

- Single project
  1. a simple straightforward evolution
  2. an entire history with branching and parallel evolution
  3. a limited history to recent releases, and
  4. a limited history that intermediate variants are missing
- Forked projects
  5. a history with two forked projects
  6. a complex history with more than two projects including branching and merging

To evaluate the accuracy for analyzing a single project, we arranged four data sets by selecting releases from PostgreSQL database project. They have a branch of major releases and branches for maintaining each minor releases.

dataset1 Pgsql-major: 12 major releases of PostgreSQL from 7.0 to 9.2.0. This dataset emulates sequential evolution of software product.

dataset2 Pgsql-8-ALL: all minor releases in PostgreSQL version 8. This dataset contains several branching and parallel evolutions with large number of variants.

dataset3 Pgsql8-latest: take up to 5 last minor releases from each major branches of PostgreSQL 8. This dataset emulates the situation that old releases in branches are lost.

dataset4 Pgsql8-annually: 25 versions of PostgreSQL. This dataset emulates the situation that developers have an incomplete set of variants. The latest variant in our dataset is released in September 2012 so we picked up the variants released around every September from 2005 to 2012.

To evaluate the accuracy for analyzing variants with project fork, we selected *FFmpeg*[3] and *Libav*[4] project which are libraries for processing multimedia data. They were originally a single project, but Libav is forked from FFmpeg with developers.

dataset5 FFmpeg: Several adjacent releases around project fork into FFmpeg and Libav.

We also selected BSD family operating systems: 4.4BSD, FreeBSD[5], NetBSD[6], and OpenBSD[7]. Those operating systems are derived from 4.3BSD but now they are independent projects. They affected each other after branching so we can say that those projects have complex evolution relationships with branching and merging.

dataset6 BSD: releases of 4.4BSD, FreeBSD, NetBSD, and OpenBSD. These products are in complex evolution relationships including branching and merging.

Table 3 shows summary of evaluation result. Column *#true-edges* shows number of edges in the true evolution graph. Column *#output* shows the number of edges in an estimated graph output by a tool. Column *#correct* shows number of correct edges, i.e. the edges included in both the estimation output and the true evolution graph. Column *precision* and *recall* shows ratio of #correct divided by #output and #true-edges respectively. In each dataset, the best values of precision and recall are emphasized by bold face and underline. Column set *#errors* provides breakdown of error reason. Column *reverse* shows the number of error edges whose reverse edges are include in the true evolution graph. Column *skip* shows the number of error edges that are not come from the direct base product, but from upstream of the direct base product. For example, 2-skip error means an edge is come from *true base product of base product of base product*.

Only for dataset1, dataset5 using gzip, and dataset6 using gzip or bzip2, EEGL shows better accuracy than PRET. For dataset4, EEGL ties with PRET only when gzip or xz is used. For dataset2 and dataset3, accuracy of EEGL is worse than PRET.

Figure 5 shows example of estmated graphs for dataset5 obtained by EEGL with gzip and PRET. Thin black arrows are correct edges. Incorrect edges are shown as colored bold edges with reason. Dashed gray arrows are correct edges but not include in the estimation. We can see that direction of derivation is correctly estimated by EEGL. In both graphs, branching after FFmpeg 0.5.4 is misaligned to Libav or FFmpeg 0.5.5.

4.1 Discussion on Evaluation Result

EEGL achieved higher precision on dataset1, and comparable precision on dataset4, therefore EEGL could be good at large change between products. When a product faces

[3] `http://www.ffmpeg.org/`
[4] `http://libav.org/`
[5] `http://www.freebsd.org/`
[6] `http://www.netbsd.org/`
[7] `http://www.openbsd.org/`

Table 3: Evaluation Result

data	#true-edges	method	#output	#correct	precision	recall	#errors			
							reverse	skip 1	skip 2	skip ≥ 3
dataset1	12	EEGL(gzip)	12	12	**1.00**	**1.00**	0	0	0	0
		EEGL(bzip2)	12	12	**1.00**	**1.00**	0	0	0	0
		EEGL(xz)	12	12	**1.00**	**1.00**	0	0	0	0
		PRET	12	11	0.917	0.917	1	0	NA	NA
dataset2	143	EEGL(gzip)	143	105	0.734	0.734	3	22	5	3
		EEGL(bzip2)	143	58	0.406	0.406	11	22	8	33
		EEGL(xz)	143	122	0.853	0.853	4	9	2	0
		PRET	143	130	**0.909**	**0.909**	8	1	NA	NA
dataset3	37	EEGL(gzip)	37	24	0.649	0.649	0	6	1	0
		EEGL(bzip2)	37	21	0.568	0.568	0	4	4	1
		EEGL(xz)	37	29	0.784	0.784	0	1	1	0
		PRET	37	32	**0.865**	**0.865**	0	0	NA	NA
dataset4	24	EEGL(gzip)	24	20	**0.833**	**0.833**	0	0	0	0
		EEGL(bzip2)	24	14	0.583	0.583	0	2	3	1
		EEGL(xz)	24	20	**0.833**	**0.833**	0	0	0	0
		PRET	24	20	**0.833**	**0.833**	0	0	NA	NA
dataset5	15	EEGL(gzip)	15	14	**0.933**	**0.933**	0	0	0	0
		EEGL(bzip2)	15	1	0.0667	0.0667	2	4	1	1
		EEGL(xz)	15	7	0.467	0.467	3	4	0	0
		PRET	15	11	0.733	0.733	2	1	NA	NA
dataset6	17	EEGL(gzip)	15	10	**0.667**	**0.588**	0	0	0	0
		EEGL(bzip2)	15	10	**0.667**	**0.588**	0	0	0	0
		EEGL(xz)	15	8	0.533	0.471	1	1	0	0
		PRET	15	9	0.600	0.529	3	0	NA	NA

large change, source files may be split or merged, or large part of a source file may be rewritten. Such cases delude PRET, since PRET stands on finding similar files with certain similarity threshold. Since EEGL does not employ any threshold and compression tools are able to find similar portion in considerably modified files, EEGL may have a capability to evaluate similarity of products that faces large change correctly.

In case that bzip2 is used in EEGL, accuracy is totally lower than others. Especially, number of skip-errors is outstanding in the results. Considering both this result and anti-monotonicity in the preliminary investigation, bzip2 is possibly not suitable for approximating Kolmogorov complexity of source code.

On the other hand, contrary to the expectation from the preliminary result, not only xz and gzip ties on some data sets, but also precision of xz is far lower than gzip for dataset5. To clarify the difference between gzip and xz, additional experiment with more data sets is required.

One superior point of EEGL to PRET is parameter-free. EEGL only requires product set and compression tools. PRET requires threshold of file similarity as a parameter on execution. This parameter substantially affects estimation result. Therefore, execution of EEGL is not optimized for the data sets.

Through observation of breakdown of the errors, EEGL tends to output more skip errors and less reverse errors in comparison with PRET. The different tendency of errors indicates the chance of improving accuracy by combining the two approaches. In addition, number of skip errors varies according to compression tool or data set. This variation also indicates that combination of compression tools have potential to reduce skip errors.

5. RELATED WORK

Arbuckle [1] proposed to use normalized compression distance (NCD) [5], which is based on Kolmogorov complexity, between versions of a software product for understanding software evolution. NCD is used as measures of quantity of information shared between the versions. He employed gzip(zlib), bzip2(bzlib), ppmd, and lzma as compression algorithms for NCD calculation, then compred the algorithms. Through the comparision, gzip could not catch the difference of versions except for consecutive and nearly identical versions. Bzip2 shows instability of compression ratio. Due to this, gzip and bzip2 were judged unsuitable for understanding software evolution. In contrast, gzip worked fine in our experiment. This difference should result from difference of the objective of the two approaches because finding a consecutive version is enough for EEGL.

Kirk and Jenkins [12] used Kolmogorov complexity for evaluating software obfuscation. They employed bzip2 for approximation of Kolmogorov complexity.

For the purpose of evaluating the value of software product or services, Fujiwara *et al.* [7] proposed a method to analyze the result of interview or questionnaire for stakeholders using Kolmogorov complexity. Result of interview or questionnaire are hierarchically clustered using distance measure based on Kolmogorov complexity. In this study, gzip is used for approximating Kolmogorov complexity.

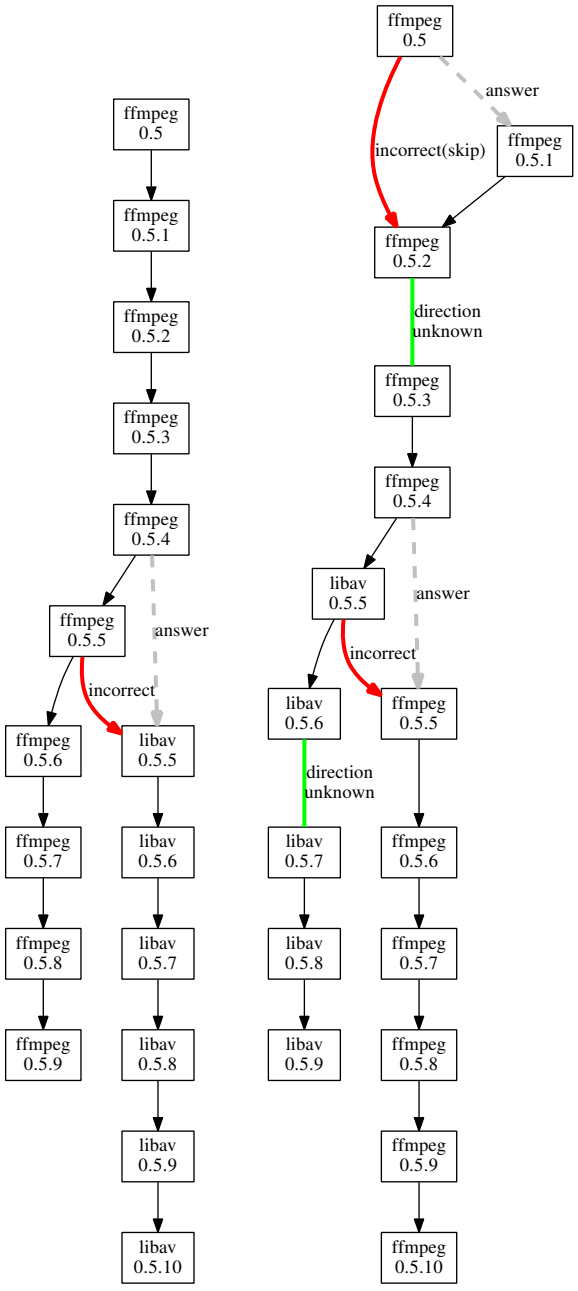

Figure 5: Estimated graphs for dataset5

Edit distance is often used for comparing program versions. For example, diff command[8] finds difference between two files. Diff treats each file as a sequence of text lines, and then calculates the shortest edit script which consists of two kind of basic edit operations, insertion and deletion of a line, between the two sequences. [10, 17, 19, 22] Difference computation between tree structure [4], which allows insertion, deletion, and update of node and moving of sub-tree as basic operation, is also used for comparing source code. [6, 8] The increase of information $I(p_i, p_j) = Z(p_i \cdot p_j) - Z(p_i)$ in this paper can be interpreted as edit distance from product p to q when internal actions of a compression/reconstruction tool are permitted as basic edit operations.

[8] `http://www.gnu.org/software/diffutils/`

6. CONCLUSION AND FUTURE WORK

This paper proposes a method to estimate a product evolution graph using Kolmogorov complexity. Result of evaluation experiment shows the difference in errors with existing method PRET. Lossless compression tools used for approximating Kolmogorov complexity also affect estimation result.

We are planning to improve estimation accuracy by combination with PRET, combination of lossless combination tools, or adopting another lossless compression algorithms.

7. ACKNOWLEDGEMENT

This work was supported by KAKENHI 25730036 and KAKENHI 25220003.

8. REFERENCES

[1] T. Arbuckle. Studying software evolution using artefacts' shared information content. *Sci. Comput. Program.*, 76(12):1078–1097, Dec. 2011.

[2] M. Burrows, D. J. Wheeler, M. Burrows, and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, 1994.

[3] G. J. Chaitin. On the length of programs for computing finite binary sequences. *J. ACM*, 13(4):547–569, Oct. 1966.

[4] S. S. Chawathe, A. Rajaraman, and H. G.-M. an d Jennifer Widom. Change detection in hierarchically structured information. *SIGMOD Rec.*, 25(2):493–504, 1996.

[5] R. Cilibrasi and P. Vitanyi. Clustering by compression. *Information Theory, IEEE Transactions on*, 51(4):1523–1545, April 2005.

[6] B. Fluri, M. Wuersch, M. PInzger, and H. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Softw. Eng.*, 33(11):725–743, Nov. 2007.

[7] Y. Fujiwara, T. Gotoh, and H. Iguchi. Product/service value validation based on kolmogorov complexity (in Japanese). In *Proceedings of Forum on Information Technology 2009*, volume 8, pages 55–62. FIT committee, August 2009.

[8] Y. Hayase, M. Matsushita, and K. Inoue. Revision control system using delta script of syntax tree. In *Proceedings of the 12th International Workshop on Software Configuration Management*, SCM '05, pages 133–149, New York, NY, USA, 2005. ACM.

[9] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40(9):1098–1101, September 1952.

[10] J. W. Hunt and M. D. McIlroy. An algorithm for differential file comparison. Technical Report CSTR 41, Bell Laboratories, Murray Hill, NJ, 1976.

[11] T. Kanda, T. Ishio, and K. Inoue. Extraction of product evolution tree from source code of product variants. In *Proceedings of the 17th International Software Product Line Conference*, SPLC '13, pages 141–150, 2013.

[12] S. R. Kirk and S. Jenkins. Information theory-based software metrics and obfuscation. *Journal of Systems and Software*, 72(2):179 – 186, 2004.

[13] A. N. Kolmogorov. Three approaches to the quantitative definition of information. *International Journal of Computer Mathematics*, 2(1-4):157–168, 1968.

[14] M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, Sept 1980.

[15] M. Lehman, J. Ramil, P. Wernick, D. Perry, and W. Turski. Metrics and laws of software evolution-the nineties view. In *Software Metrics Symposium, 1997. Proceedings., Fourth International*, pages 20–32, Nov 1997.

[16] M. Li, X. Chen, X. Li, B. Ma, and P. Vitanyi. The similarity metric. *Information Theory, IEEE Transactions on*, 50(12):3250–3264, Dec 2004.

[17] D. Maier. The complexity of some problems on subsequences and supersequences. *J. ACM*, 25(2):322–336, Apr. 1978.

[18] G. N. N. Martin. Range encoding: an algorithm for removing redundancy from a digitized message. In *Proceedings of the Video & Data Recording Conference*, Southampton, Jul. 1979.

[19] E. W. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, 1:251–266, 1986.

[20] R. Solomonoff. A formal theory of inductive inference. part I. *Information and Control*, 7(1):1 – 22, 1964.

[21] R. Solomonoff. A formal theory of inductive inference. part II. *Information and Control*, 7(2):224 – 254, 1964.

[22] S. Wu, U. Manber, G. Myers, and W. Miller. An O(NP) sequence comparison algorithm. *Inf. Process. Lett.*, 35(6):317–323, 1990.

[23] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theor.*, 23(3):337–343, Sept. 2006.