Algorithm to Determine Extended Edit Distance between Program Codes

Kazuki Anzai, Yutaka Watanobe
Graduate Department of Computer Science and Engineering
University of Aizu
Fukushima, 965–8580, Japan
Email: {m5231114, yutaka}@u-aizu.ac.jp

***Abstract*—An algorithm to determine the extended edit distance between program codes is presented. In addition to the conventional Levenshtein distance, the extended edit distance considers some common operations to a program code to find similar programs more accurately. To calculate the distance, the algorithm employs dynamic programming techniques as well as an algorithm for solving the minimum cost flow on a bipartite graph.**

In this paper, details of the algorithm and experimental results are presented. These experiments were conducted with source code submitted to an online judge system, where a number of source codes for each programming problem are located. The results show that the proposed algorithm can find source code that cannot be found by the conventional Levenshtein distance, with a higher probability.

I. Introduction

Information technology plays an important role in all aspects of our life, and a number of activities in computer education have been promoted by different organizations. Coding skills and computational thinking are becoming increasingly important not only for programmers but also for employers in many fields. Thus both interest in, and demand for, programming education and related e-learning systems is rising.

A number of tools and methods have been proposed for novice programmers to acquire coding skills beyond conventional school lessons. E-learning tools that employ online materials (e.g. texts and videos), gamification, visual programming and programmable robots are representative examples. Among such tools, Online Judge Systems (OJSs), which provide many problems at different proficiency levels and judge submitted source code, have become popular as a service to acquire coding skills online [1]. Different related services for induction courses and personal assessment have also been developed in industries. Generally, in an OJS, source code submitted and judged for each problem (task) is accumulated in the corresponding database and such resources can be consulted by other learners as educational materials. Furthermore, the source code and related trial-and-error history of learners can be valuable assets to make the e-learning system more intelligent and then improve learning efficiency. For example, utilizing data in an online judge system, algorithms for logic error detection, learning path recommendations and estimation of problem difficulty have already proposed [2]–[7].

Among many applications of the accumulated source code, in this paper, we focus on an algorithm to find similar source code from a set of target source code with the following motivations:

- Learners can search similar source code from a large number of solutions provided by others to find bugs in their codes
- Learners can refactor their code by consulting similar code for the target problem
- Teachers can identify cheats (learners who have copied the work of others) in submissions for a specific problem
- Clustering source code to fasten related algorithms (e.g. search functions)
- Developing other machine learning algorithms related to, for example, code clones, classification, automated debugging and code completion.

To estimate distance between given source code, the Levenshtein distance, which measures distance between given strings, can be employed. However, in many cases, the Levenshtein distance is not suitable for comparing program codes because of their structural characteristics. So, in this paper, first we define an extended edit distance between given source code and propose the corresponding algorithm. The algorithm consists of three phases. In the first phase a given code is divided into functions. In the second phase each function is divided into blocks. In the third phase, each block is divided into tokens and other preprocessing tasks are performed. The distance between two blocks is calculated by a dynamic programming technique. Finally, the correspondence between blocks and functions of the two source codes is considered by an algorithm to find the minimum cost flow. To evaluate the proposed algorithm, several experiments are conducted using resources in an online judge system.

In terms of these experiments, we prepared three patterns. One of them is the extended edit distance, and the other two were for comparison. The pattern that uses the extended edit distance could yield a smaller cost compared to the others, with high probability. Although the execution time of the program increased, accuracy was improved. Therefore, the extended edit distance can contribute to programming education as well as to the above mentioned applications.

The rest of this paper is organized as follows. Section 2

978-1-7281-4882-3/19/$31.00 ©2019 IEEE
DOI 10.1109/MCSoC.2019.00033

discusses related works before Section 3 defines the extended edit distance. Next, Section 4 provided details of the proposed algorithm. Section 5 demonstrates results of the experiments before, finally, Section 6 concludes the paper.

II. Related Work

First, we should understand the features of given texts to define and calculate their edit distance considering their structures and elements as well as the significance of orders. Several approaches that calculate the similarity of given source codes exist. For example, there are algorithms that use the Longest Common Subsequence (LCS) [8] and calculate the edit distance after fixing the code in intermediate language [9], [10]. There are also approaches that calculate the similarity between two documents by extending edit distance. However, there has been no research conducted that extends the edit distance to measure similarity between source codes [11].

Second, we should consider the purpose of providing edit distance to target users. There are existing approaches that compile given source codes and then measure distance. Generally, such systems are aimed at detecting theft. Their goal is to find programs that behave similarly. However, even if we can find a source code that behaves similarly to a sample, that is not always appropriate. For example, if the purpose is to consult similar source codes, it is preferable that the samples presented are similar in terms of source code structure. There are many approaches that compile given codes into a syntax tree in intermediate language before comparison [12], [13]. However, even if similarity after compilation is high, the structure of the original code is not necessarily similar.

III. Definition of Extended Edit Distance

Generally, the edit distance represents the degree of similarity between sentences. The definition of edit distance is the number of operations required to make two sentences the same by three kinds of edit operation:

- Insert one character
- Delete one character
- Replace one character

However, this research deals with source code rather than sentences. For each source code, the above three editing methods are not sufficient, because even if the normal editing distance is a very large value, the behavior of the program may be very similar. For example, if a variable name used in a program is changed by modification of even one character, and the variable is used many times in the program, the normal edit distance would become a large value. As another example, if the order of declaring functions is different, in normal editing distances will be large.

Taking into account the above mentioned features, we define the extended edit distance by increasing editing operation types for source code. The operations for the extended edit distance are as follows:

- Insert one token
- Delete one token
- Replace one token
- Insert one line
- Delete one line
- Replace one line
- Change variable names
- Change function names
- Change the order of functions
- Change the order of blocks

IV. Proposed Algorithm

A. Overview

Fig. 1 shows an overview of the proposed algorithm to obtain the extended edit distance between two given source codes. The algorithm consists of three layers each of which performs divide and corresponding conquer operations. The first layer divides a source code into functions and measures their similarity to calculate the distance between the source codes. The second layer divides the obtained function into blocks and measures their similarity to calculate the distance between functions. The third layer divides the obtained block into tokens to calculate the distance between two blocks.

B. Calculating extended edit distance between two blocks

In this process, the edit distance is extended to measure the distance between blocks by using dynamic programming. Some pre-processing is performed as follows.

- Divide blocks by token
- Replace variable names and function names

First, we divide a block into tokens. The minimum units (character strings such as element names and operators) that cannot be broken down into more detailed units are called tokens. By doing this, it is possible to change the operation for each character to the operation for each token. The rationale for changing to token-by-token operation is as follows. Suppose for example that there are codes in which *break* and *continue* statements are used in a certain point, and there are

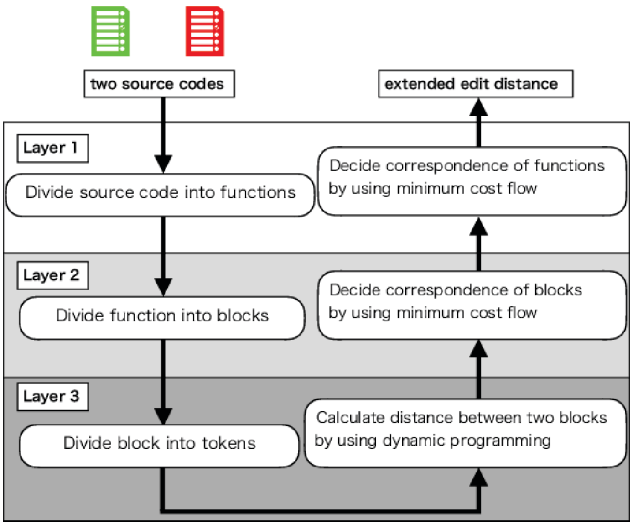

Fig. 1. Overview of the proposed algorithm

other codes in which if and $while$ statements are used in a certain point. The edit distance between $break$ to $continue$ is 9. On the other hand, the edit distance between if and $while$ is 4, which is less than half of nine. In this way, a difference will be occur depending on the combination of reserved words. It is possible to prevent the difference by operating on a token basis rather than manipulating it by one character.

Second, we replace variable names and function names. If the cost of replacing a variable name is close to 0, it is prudent to perform an operation to change all variable names to the same variable name. It can be expressed by reducing the cost of rewriting the token representing the variable name to the token representing another variable name. Therefore, for each token, it is necessary to check whether it is representing variable names or function names.

To elaborate, we explain why we should change all variable names to the same variable name. There are cases when the edit distance is 1 even though the variables have the same meaning. Further, there are cases when the edit distance is 0 even though the variables have different meanings. For example, suppose that there are two variables named ”cnt” and ”count”, and that these meanings are the same. Then, the edit distance is 1. Next, there are two variables both of which are named ”res” but their meanings are different. Then, the edit distance is 0. Therefore, we should consider how to reduce extra distance. The same logic applies to function names.

C. Dividing functions into blocks and perform matching

An if statement, a for statement, and a $while$ statement can be considered as one block, and a group of sentences between them is also defined as a block. As shown in Fig. 2, the operation for making the two codes the same by swapping the order of the blocks can be regarded as a problem of matching blocks that have the same meaning. It should be noted that one block must not be matched with more than one block at this time. Therefore, if the number of blocks in each function is different, we add an empty block to one function that has less blocks than the other function until the number of blocks is the same. This problem can be reduced to the problem of bipartite matching, by constructing a bipartite graph as follows. Make each block correspond to a node. For a pair of blocks contained in the same function, we do not add edges between the corresponding nodes. For every pair of blocks contained in a different function, we add an edge between the corresponding nodes. The edge weight between two nodes is the extended edit distance between the blocks.

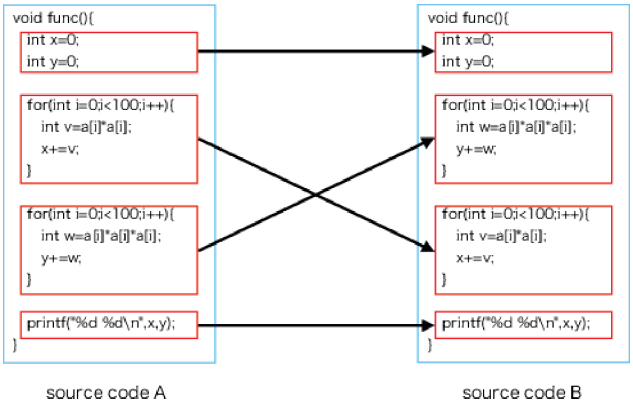

Fig. 2. Example of dividing function into blocks

D. Dividing codes into functions and perform matching

As per the algorithm to calculate the distance between functions by the matching of blocks, the distance between whole codes is calculated by the matching of functions. In this case, a bipartite graph, where each node corresponds to a function, is constructed.

E. Minimum cost flow

The problem of matching on a bipartite graph can be approached using minimum cost flow [8]. The minimum cost flow can be solved by the primal-dual method [14]. First, we make a set of directed edges. The first end-point of the edge is the node corresponding to the source code S, and the edge of another end-point is the node corresponding to the source code T. The capacity of all edges is 1, and the cost is the same as the weight of the edge. Next, we add two nodes, one for the source and another for the sink. For all nodes that correspond to source code S, we add edges from the source to them where the cost is 0, and the capacity is 1. For all nodes that correspond to source code T, add edges from them to the sink where the cost is 0, and the capacity is 1. Fig. 3 shows the graph that is constructed by this procedure. Matching cost is the sum of costs when as much flow as possible from the source to the sink occurs in the directed graph.

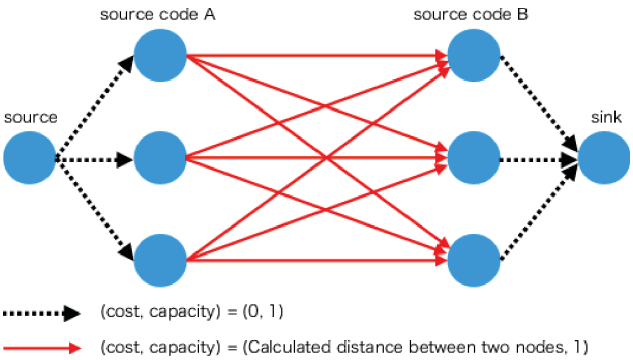

Fig. 3. Bipartite flow network

F. Dynamic programming

The distance between two blocks is calculated by a dynamic programming technique. The recursive formula for dynamic programming is defined as follows.

$$
\begin{aligned}
&dp(i,j) = \\
&\left(\begin{array}{l}
\begin{cases}
0 \quad (i = S.length, j = T.length) \\
dp(i, j+1) + 1 \quad (i = S.length) \\
dp(i+1, j) + 1 \quad (j = T.length) \\
dp(i+1, j+1) \quad (S_i = T_j) \\
min \begin{pmatrix} dp(i+1,j) \\ dp(i,j+1) \\ dp(i+1,j+1) \end{pmatrix} + 1 \quad (other)
\end{cases} \\
\\
min \begin{cases}
dp(i, j + getLineSize(T,j)) + getLineCost(T,j) \\
(i = S.length, j \ \ is \ \ beginning \ \ of \ \ a \ \ line) \\
dp(i + getLineSize(S,i), j) + getLineCost(S,i) \\
(j = T.length, i \ \ is \ \ beginning \ \ of \ \ a \ \ line) \\
min \begin{pmatrix}
dp(i + getLineSize(S,i), j) \\ \quad +getLineCost(S,i)) \\
(i \ \ is \ \ beginning \ \ of \ \ a \ \ line) \\
dp(i, j + getLineSize(T,j)) \\ \quad +getLineCost(T,j) \\
(j \ \ is \ \ beginning \ \ of \ \ a \ \ line) \\
dp(i + getLineSize(S,i), \\ \quad j + getLineSize(T,j)) \\ \quad +getLineCost2(S,i,T,j) \\
(i \ \ is \ \ beginning \ \ of \ \ a \ \ line, \\
j \ \ is \ \ beginning \ \ of \ \ a \ \ line)
\end{pmatrix} (other)
\end{cases}
\end{array}\right)
\end{aligned}
$$

The upper half shows the Levenshtein distance transitions. $dp(i,j)$ is the minimum cost distance between S' and T'. S' is a substring of S ($S' = S_i + S_{i+1} + ... + S_{S.length-1}$). T' is a substring of T ($T' = T_i + T_{i+1} + ... + T_{T.length-1}$). S and T represent two blocks. $dp(0,0)$ is the minimum cost distance between two blocks. $getLineSize(X,Y)$ returns the size of the array X from the index Y to the index of the next end-of-line. $getLineCost(X,Y)$ returns the cost that is $getLineSize(X,Y) \times 0.9$. $getLineCost2(X,Y,Z,W)$ returns the cost that is $max(getLineSize(X,Y), getLineSize(Z,W)) \times 0.8$.

G. Implementation

Algorithms 1, 2, 3 and 4 are pseudo codes to calculate the distance between two codes, calculate the distance between two functions, calculate the distance between two blocks and perform bipartite matching respectively. G is a two-dimensional array and represents the adjacency matrix. $G_{i,j}$ represents the weight of the edge that connects nodes i and j. Note that -1 means that there is no edge between node i and node j. It is assumed that all elements are initialized at -1.

Algorithm 1 Calculate distance between two codes

Require: S, T (there are strings of two source codes)
Ensure: X
  $n \Leftarrow$ the number of functions in S
  $m \Leftarrow$ the number of functions in T
  $x \Leftarrow$ array of string (divide S into functions)
  $y \Leftarrow$ array of string (divide T into functions)
  for i = 1 **to** i = n **do**
    for j = 1 **to** j = m **do**
      $G[i][n+j] \Leftarrow Algorithm2(x_i, y_j)$
    end for
  end for
  $X \Leftarrow Algorithm4(G)$

Algorithm 2 Calculate distance between two functions

Require: S, T (there are strings of two functions)
Ensure: X
  $n \Leftarrow$ the number of blocks in S
  $m \Leftarrow$ the number of blocks in T
  $x \Leftarrow$ array of string (divide S into blocks)
  $y \Leftarrow$ array of string (divide T into blocks)
  for i = 1 **to** i = n **do**
    for j = 1 **to** j = m **do**
      $G[i][n+j] \Leftarrow Algorithm3(x_i, y_j)$
    end for
  end for
  $X \Leftarrow Algorithm4(G)$

Algorithm 3 Calculate distance between two blocks

Require: S, T (there are strings of two blocks, variable names and function names are unified)
Ensure: X
  $n \Leftarrow$ the number of tokens in S
  $m \Leftarrow$ the number of tokens in T
  $x \Leftarrow$ array of string (divide S into tokens)
  $y \Leftarrow$ array of string (divide T into tokens)
  $dp \Leftarrow$ two dimensional array of integer
  for i = 0 **to** $i = n$ **do**
    for j = 0 **to** $j = m$ **do**
      $dp[i][j] \Leftarrow$ INTMAX
    end for
  end for
  $dp[n][m] \Leftarrow 0$
  for $i = n$ **to** i = 0 **do**
    for $j = m$ **to** j = 0 **do**
      if $i = 0$ **then**
        $dp[0][0] \Leftarrow dp[i][j] + j$
        if $m - j$ is beginning of a line **then**
          $dp[i][j - getLineSize(y, m-j)] \Leftarrow min(dp[i][j - getLineSize(y, m-j)], dp[i][j] + getLineCost(y, m-j))$
        end if
        $continue$
      end if
      if $j = 0$ **then**
        $dp[0][0] \Leftarrow dp[i][j] + i$
        if $n - i$ is beginning of a line **then**
          $dp[i - getLineSize(x, n-i)][j] \Leftarrow min(dp[i - getLineSize(x, n-i)][j], dp[i][j] + getLineCost(x, n-i))$
        end if
        $continue$
      end if
      if $x[n-i] = y[m-j]$ **then**
        $dp[i-1][j-1] \Leftarrow min(dp[i-1][j-1], dp[i][j])$
      else
        $dp[i-1][j] \Leftarrow min(dp[i-1][j], dp[i][j] + 1)$
        $dp[i][j-1] \Leftarrow min(dp[i][j-1], dp[i][j] + 1)$
        $dp[i-1][j-1] \Leftarrow min(dp[i-1][j-1], dp[i][j] + 1)$
      end if
      if $n - i$ is beginning of a line **then**
        $dp[i - getLineSize(x, n-i)][j] \Leftarrow min(dp[i - getLineSize(x, n-i)][j], dp[i][j] + getLineCost(x, n-i))$
      end if
      if $m - j$ is beginning of a line **then**
        $dp[i][j - getLineSize(y, m-j)] \Leftarrow min(dp[i][j - getLineSize(y, m-j)], dp[i][j] + getLineCost(y, m-j))$
      end if
      if $n - i$ and $m - j$ is beginning of a line **then**
        $dp[i - getLineSize(x, n-i)][j - getLineSize(y, m-j)] \Leftarrow min(dp[i - getLineSize(x, n-i)][j - getLineSize(y, m-j)], dp[i][j] + getLineCost2(x, n-i, y, m-j))$
      end if
    end for
  end for
  $X \Leftarrow dp[0][0]$

Algorithm 4 Bipartite Matching

Require: G (there are adjacent matrix)
Ensure: X
  $n \Leftarrow$ the number of nodes in G
  $source \Leftarrow n+1$
  $sink \Leftarrow n+2$
  for $i = 1$ **to** $i = \frac{n}{2}$ **do**
    for $j = \frac{n}{2}+1$ **to** $j = n$ **do**
      $FlowNetwork[i][j].capacity \Leftarrow 1$
      $FlowNetwork[i][j].cost \Leftarrow G[i][j]$
    end for
  end for
  for $i = 1$ **to** $i = \frac{n}{2}$ **do**
    $FlowNetwork[source][i].capacity \Leftarrow 1$
    $FlowNetwork[source][i].cost \Leftarrow 0$
    $FlowNetwork[i+\frac{n}{2}][sink].capacity \Leftarrow 1$
    $FlowNetwork[i+\frac{n}{2}][sink].cost \Leftarrow 0$
  end for
  $X \Leftarrow MinCostFlow(FlowNetwork, n+2, source, sink, \frac{n}{2})$

H. Complexity of the algorithm

The Bellman-Ford algorithm [8] is employed to perform minimum cost flow. The time complexity of the Bellman-Ford algorithm is $O(VE)$ where V is the number of nodes and E is the number of edges. E is V^2 because the bipartite graph is a complete graph. Therefore, the time complexity of minimum cost flow is $O(FV^3)$ where F is the number of nodes. In other words, the time complexity of the minimum cost flow is $O(V^4)$. The time complexity to calculate the extended edit distance between two blocks is $O(L^2)$ where L is the number of tokens. The time complexity to divide the functions into blocks and perform matching to calculate the distance between two functions is $O(M^2L^2 + M^4)$ where M is the number of blocks. The time complexity to divide the codes into functions and perform matching to calculate the distance between two codes is $O(N^2M^2L^2 + N^2M^4 + N^4)$ where N is the number of functions.

V. Experiment

To evaluate the proposed algorithm, we have conducted an experiment. The machine model is shown in Table I.

A. Experimental data

Table II provides details of the experimental data. For the experiment, we use source code submitted to each problem that are included in an online judge system, Aizu Online Judge (AOJ) [15], [16]. In each problem, we choose accepted codes and rejected (wrong answer) codes that were submitted within a certain term. The set of selected accepted codes were all submitted before the set of selected rejected codes because we need to avoid code cloning to experiment appropriately. In AOJ, wrong answer is one of the statuses that is returned by the judge system. Also, wrong answer is returned when the submitted source code generates output data different from judge data. Problem 1 is related to basic algorithm which visits and fills cells in a 2 dimensional grids by depth-first search (DFS). Problem 2 to 5 are related to basic programming and structure which represents a number of dice and rolling them.

TABLE I
Experimental environment

OS	macOS Mojave 10.14.4
CPU	2.2 GHz Intel Core i7
Memory	8 GB 1600 MHz DDR3

B. Experimental contents

For each rejected code, we find similar codes in the following three ways from the accepted code.

- Pattern A: Code found when searching for the smallest edit distance.
- Pattern B: Code found when searching for the smallest edit distance extended for editing per token.
- Pattern C: Code found when searching for the smallest extended editing distances proposed in this paper.

C. Results

Table III shows the average of the minimum edit distance per rejected code, and the average of the execution time per rejected code.

Table IV shows the number of times that two patterns yield different source codes and probability that two patterns yield different source code.

Determination of the number of times that two patterns yield different source code occurs as follows.

For each rejected code, if the following two conditional statements are true, the number is incremented by one.

- The smallest edit distance calculated by pattern C is smaller than the smallest edit distance calculated by another pattern.
- For each accepted code, if the edit distance between the rejected and accepted code calculated by pattern C is the smallest edit distance of pattern C, the edit distance between the rejected and accepted code calculated by another pattern is not the smallest edit distance of another pattern.

The probability that two patterns yield different source code is calculated as the number of times that two patterns yield different source code divided by the number of rejected codes.

TABLE III
Average results of the three patterns

Problem number	Pattern	Average of the minimum edit distance per rejected code	Average of the execution time per rejected code (sec.)
1	A	470.773	0.72596
	B	227.833	0.35429
	C	161.907	16.9013
2	A	486.195	1.31486
	B	215.623	0.50214
	C	155.238	19.7280
3	A	734.583	2.20858
	B	325.057	0.68798
	C	254.760	39.7583
4	A	832.552	5.38987
	B	387.468	1.67741
	C	304.010	127.117
5	A	883.291	5.49755
	B	444.567	1.87037
	C	343.151	111.313

TABLE II
DETAILS OF EXPERIMENTAL DATA

Problem number	Problem name	Problem id	Number of accepted codes	Number of rejected codes	Submission date term of accepted codes (mm/dd/yyyy)	Submission date term of rejected codes (mm/dd/yyyy)
1	Red and Black	1130	300	300	07/14/2009 - 12/27/2013	01/31/2014 - 07/04/2018
2	Dice I	ITP1_11_A	400	400	06/25/2014 - 03/11/2016	03/13/2016 - 07/27/2018
3	Dice II	ITP1_11_B	300	300	06/25/2014 - 03/11/2016	04/01/2016 - 07/25/2018
4	Dice III	ITP1_11_C	500	500	06/25/2014 - 03/15/2017	03/18/2017 - 07/27/2018
5	Dice IV	ITP1_11_D	450	450	06/25/2014 - 03/15/2017	03/30/2017 - 07/27/2018

TABLE IV
COMPARING RESULTS OF THE THREE PATTERNS

Problem number	Pattern	Number of times that both patterns yield different source code.	Probability that both patterns yield different source code.
1	A and C	242	0.8067
	B and C	197	0.6567
2	A and C	333	0.8325
	B and C	208	0.5200
3	A and C	248	0.8267
	B and C	196	0.6533
4	A and C	411	0.8220
	B and C	347	0.6940
5	A and C	373	0.8289
	B and C	344	0.7644

D. Discussion

The results suggest that the proposed algorithm can find source code that cannot be found by the conventional Levenshtein distance, with a higher probability. From the definition of the extended edit distance, it is clear that the difference of these minimum costs is achieved by extended operations that were added to the conventional Levenshtein distance. Because of the characteristics of programming language, these operations are effective to calculate the distance. Therefore, it is considered pattern C is better than the other. The average of the minimum edit distance calculated by pattern C is smaller than that calculated by patterns A and B. For all problems, pattern C yielded different source code from pattern A with a probability of over 0.8 and pattern B with a probability of over 0.5. Needless to say, the probabilities that patterns A and C yield different source code are quite high. However, the probabilities that patterns B and C get different source code exhibit little variation, which warrants investigation. First of all, the results of problem number 1 and problem number 3 exhibit almost the same probabilities. Secondly, results from problem number 2 to problem number 5 always increase. Generally, the larger the average edit distance of each pattern, the larger the probability that pattern B and C get different source code. Additionally, it is assumed that the larger the average edit distance of each pattern, the more difficult to implement, the larger the code length and the greater the number of functions/blocks. As a result, it is assumed that the greater the number of functions/blocks, the higher the need to change the order of the functions/blocks. Therefore, the more difficult to implement, the greater the need for the extended edit distance algorithm. Although the execution time of the program increased, accuracy was improved.

VI. CONCLUSION

An algorithm to find extended edit distance has been presented by applying some common operations to a program code in addition to the conventional Levenshtein distance. To calculate the distance defined, dynamic programming techniques are utilized as well as an algorithm for solving the minimum cost flow on a bipartite graph.

Details of the algorithm and results of experiments have been presented. The results show that the proposed algorithm is able to find source code that cannot be found by the conventional Levenshtein distance, with a higher probability. Thus, the proposed approach could be used for developing different applications for searching and clustering similar source code as well as for other data mining applications.

ACKNOWLEDGMENT

The present study was supported by JSPS KAKENHI Grant Number 19K12252.

REFERENCES

[1] S. Wasik, M. Antczak, J. Badura, A. Laskowski, and T. Sternal, "A survey on online judge systems and their applications," *ACM Comput. Surv.*, vol. 51, no. 1, pp. 3:1–3:34, Jan. 2018. [Online]. Available: http://doi.acm.org/10.1145/3143560

[2] Y. Teshima and Y. Watanobe, "Bug detection based on lstm networks and solution codes," in *2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, Oct 2018, pp. 3541–3546.

[3] Y. Yoshizawa and Y. Watanobe, "Logic error detection algorithm for novice programmers based on structure pattern and error degree," in *2018 9th International Conference on Awareness Science and Technology (iCAST)*, Sep. 2018, pp. 297–301.

[4] T. Saito and Y. Watanobe, "Learning path recommender system based on recurrent neural network," in *2018 9th International Conference on Awareness Science and Technology (iCAST)*, Sep. 2018, pp. 324–329.

[5] C. M. Intisar and Y. Watanobe, "Classification of online judge programmers based on rule extraction from self organizing feature map," in *2018 9th International Conference on Awareness Science and Technology (iCAST)*, Sep. 2018, pp. 313–318.

[6] C. M. Intisar and Y. Watanobe, "Cluster analysis to estimate the difficulty of programming problems," in *Proceedings of the 3rd International Conference on Applications in Information Technology*, ser. ICAIT'2018. New York, NY, USA: ACM, 2018, pp. 23–28. [Online]. Available: http://doi.acm.org/10.1145/3274856.3274862

[7] C. M. Intisar, Y. Watanobe, M. Poudel, and S. Bhalla, "Classification of programming problems based on topic modeling," in *Proceedings of the 2019 7th International Conference on Information and Education Technology*, ser. ICIET 2019. New York, NY, USA: ACM, 2019, pp. 275–283. [Online]. Available: http://doi.acm.org/10.1145/3323771.3323795

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. [Online]. Available: http://mitpress.mit.edu/books/introduction-algorithms

[9] N. Kawamitsu, T. Ishio, T. Kanda, R. G. Kula, C. D. Roover, and K. Inoue, "Identifying source code reuse across repositories using lcs-based source code similarity," in *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, Sep. 2014, pp. 305–314.

[10] F. Al-Omari, I. Keivanloo, C. K. Roy, and J. Rilling, "Detecting clones across microsoft .net programming languages," in *2012 19th Working Conference on Reverse Engineering*, Oct 2012, pp. 405–414.

[11] M. M. M. Fuad and P. Marteau, "The extended edit distance metric," in *2008 International Workshop on Content-Based Multimedia Indexing*, June 2008, pp. 242–248.

[12] J. Kim, H. Choi, H. Yun, and B.-R. Moon, "Measuring source code similarity by finding similar subgraph with an incremental genetic algorithm," in *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, ser. GECCO '16. New York, NY, USA: ACM, 2016, pp. 925–932. [Online]. Available: http://doi.acm.org/10.1145/2908812.2908870

[13] C. Ragkhitwetsagul, J. Krinke, and D. Clark, "Similarity of source code in the presence of pervasive modifications," in *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, Oct 2016, pp. 117–126.

[14] H. Z. Aashtiani, T. L. Magnanti *et al.*, "Implementing primal-dual network flow algorithms," 1976.

[15] "Aizu online judge," https://onlinejudge.u-aizu.ac.jp/.

[16] Y. Watanobe, "Development and operation of an online judge system," *IPSJ Magazine*, vol. 56 (10), pp. 998–1005.