Batch Source-Code Plagiarism Detection Using an Algorithm for the Bounded Longest Common Subsequence Problem

R. A. Castro Campos[1], F. J. Zaragoza Martínez[2]
Departamento de Sistemas, UAM Azcapotzalco, Mexico City, Mexico

E-mail: racc@correo.azc.uam.mx[1], franz@correo.azc.uam.mx[2]

Abstract — **Source-code plagiarism detection is an unfortunate but necessary activity when reviewing assignments of programming courses. While being reasonably easy to fool, string-based comparisons offer a high degree of accuracy with almost no false positives and usually a good string similarity metric is the length of their longest common subsequence. In the case of two strings, the dynamic programming algorithm for this calculation unfortunately takes quadratic time even if the strings are equal. In this paper we present an algorithm that, given a batch of source-code files, efficiently finds all pairs of similar files by preprocessing the files and then using a fast branch-and-bound algorithm to find only those pairs whose longest common subsequence is indicative of plagiarism.**

Keywords — **Plagiarism detection, longest common subsequence, branch and bound, source-code**

I. Introduction

Determing the similarity of source-codes in order to detect potential plagiarism is a difficult problem that may be subjective at times and, when the amount of source-codes to compare is huge, it is not practical to do it manually. In the last decades, many computer-assisted techniques to detect source-code plagiarism have been developed such as syntax tree matching and program dependence graph isomorphism detection [1]. However, sophisticated techniques require a considerable amount of analysis time and are not false-positive free [2] which is important if the students were to be sanctioned. Because of this, textual comparison of source-codes is preferred when both the analysis time and accuracy are of importance even if some cases are missed. In the context of approximate string matching, a very popular string metric is the length of the longest common subsequence, a special case of the edit distance [3, 4].

The longest common subsequence problem consists on finding the longest sequence of characters that are common to the given strings respecting the order of apparition of the characters in the strings. One of the most well-known algorithms for finding the longest common subsequence of two strings $A_0A_1...A_{m-1}$ and $B_0B_1...B_{n-1}$ is a variant of the dynamic programming algorithm for the edit distance, first developed by Wagner and Fischer and with a running time of O(mn) using O(max(m, n)) space [3]. See formula 1.

$$L(i,j) = \begin{cases} 0 & \text{if } i = m \vee j = n \\ L(i+1, j+1) + 1 & \text{if } A_i = B_j \\ max \begin{Bmatrix} L(i, j+1) \\ L(i+1, j) \end{Bmatrix} & \text{if } A_i \neq B_j \end{cases}$$

Formula 1. Length of the longest common subsequence of two strings A and B.

However our goal is to quickly find wheter two strings are similar enough to suspect plagiarism and we are not really interested in calculating the exact length of the longest common subsequence for strings that we can quickly determine that are not very similar. In particular, to check if two strings are strictly identical there exists a trivial O(n) algorithm and it can be seen that a recursive implementation of the Wagner-Fischer algorithm degenerates in such algorithm. See Fig. 1.

			a	*a*	*a*	*a*	*c*	
		0	1	2	3	4	5
a	0	④	3	2	1	0	0
a	1	3	③	2	1	0	0
a	2	2	2	②	1	0	0
a	3	1	1	1	①	0	0
b	4	0	0	0	0	⓪	⓪
	5	0	0	0	0	⓪	0

Fig. 1. The dynamic table for the strings *aaaab* and *aaaac*. The circled entries are the ones calculated by the recursive Wagner-Fischer.

It has been shown [5] that only $p(m+n - 2p + 1)$ entries of the dynamic table are needed to calculate the longest common subsequence of two strings where p is the lenght of such subsequence. That is, the existence of easy and not so easy instances of this problem is fundamented; the easy instances are the ones where the strings are either extremely similar or extremely dissimilar [6].

II. Proposed Algorithm

In this paper we develop a simple algorithm that is faster than dynamic programming for the task of finding the lenght of the longest common subsequence between similar source-codes and that invests as little time as possible for very dissimilar source-codes. For this we have developed a modified Wagner-Fischer algorithm that, instead of using dynamic programming or depth-first search via recursion, uses an iterative breadth-first search algorithm and runs in linear space.

The modification of the original algorithms requires to preprocess the input. This preprocessing, first introduced in [6], defines for each input string a function that given a symbol and an index, calculates in constant time the next closest index that also contains such symbol. Given a string of length m and an alphabet $\sum = \{\sigma_0, \sigma_1, ..., \sigma_{s-1}\}$ the preprocessing can be done in O(sm) time and space. See formula 2.

$$C_A(\sigma, i) = \begin{cases} m & \text{if } i = m \\ i & \text{if } A_i = \sigma \\ C_A(\sigma, i+1) & \text{if } A_i \neq \sigma \end{cases}$$

Formula 2. Definition of the precomputed function for each input string

In our case, the overhead of preprocessing the input source-codes is amortized by the fact that we aim to find all the pairs of similar source-codes of a given set, and we only need to preprocess each source-code once.

With the previous function defined, it is possible to modify the Wagner-Fischer algorithm. See formula 3.

$$L(i,j) = \begin{cases} 0 & \text{if } i = m \vee j = n \\ L(i+1, j+1) + 1 & \text{if } C_A(B_j, i) = i \\ L(i, j+1) & \text{if } C_A(B_j, i) = m \\ max \begin{Bmatrix} L(C_A(B_j, i) + 1, j+1) + 1 \\ L(i, j+1) \end{Bmatrix} & \text{otherwise} \end{cases}$$

Formula 3. Modified Wagner-Fischer algorithm

In contrast with the original algorithm, the previous definition always increments the second index j. This allows us to process the dynamic table column after column and thus facilites an iterative and linear space implementation of breadth-first search, instead of using the depth-first search recursive approach.

To implement this algorithm we need two auxiliary arrays. The first one contains the calculated entries of the current column and the second will contain the entries of the next column. Although both arrays have a max length equal to the lenght of the columns of the dynamic table, not every entry of a given column may have to be considered. Because of this and since the entries will not necessarily be processed in row order, it is needed that both arrays also act as linked lists: adding a new entry for the next column links it with the last inserted entry for the same column.

Additionally, using breadth-first search to calculate the entries of the dynamic table requires to propagate the length of the already found subsequences to the entries about to be calculated. If there exists more than one way to add the same entry to the list, the recorded length will be the maximum of lengths propagated to such entry. See Fig. 2.

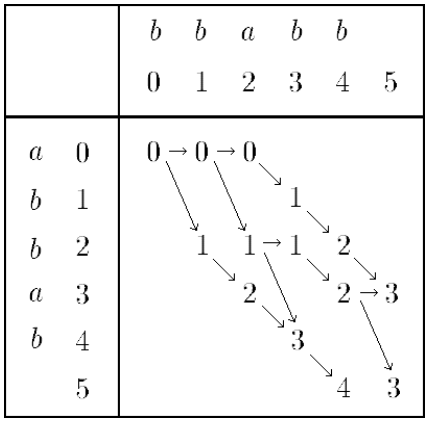

Fig. 2. Running of the breadth-first search algorithm for the strings *abbab* and *bbabb*. Diagonals increase the length by one.

Finally, it is possible to discard some branches of the search tree if we need to find a common subsequence of at least a given length. To apply this branch and bound criteria we just need to examine the propagated length of a given entry and stop the search if the requested lower bound for the longest common subsequence cannot be reached starting from that entry even in the best case. See Fig. 3.

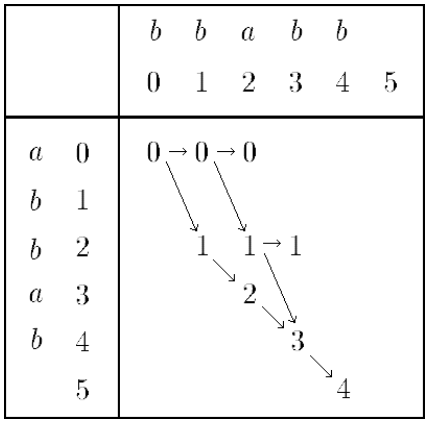

Fig. 3. Running of the breadth-first search algorithm with 4 as the requested lower bound for the length of the longest common subsequence.

III. Results

The tests consisted in finding the number of pairs of a given set of strings whose longest common subsequence length was at least some percentage of the length of the longest string. To design the tests, we analyzed the frequency of apparition of the tokens found in 1154 source-codes written as assignments by students of introductory programming courses taught using the C language [8]. See Table I.

TABLE I
FREQUENCY OF TOKENS IN 1154 CODES WRITTEN IN C

Token	Freq.	Token	Freq.
%=	2	<=	468
/=	3	%	494
?	3	*character literal*	535
Default	3	&&	547
Float	3	+	1138
Sizeof	4	++	1309
Switch	4	*	1332
->	5	For	1355
|	6	-	1401
-=	7	#	1608
Void	8	If	1647
*=	9	==	1872
Continue	12	>	1881
Double	12	&	2413
floating point literal	27	<	2819
Return	28	[	2987
Case	32	]	2987
--	36	.	3578
While	39	*string literal*	3941
:	41	}	5328
!	76	{	5359
||	106	=	5720
Int	136	,	6432
>=	147	*integer literal*	7192
Break	177	)	11858
!=	185	(	11863
/	236	;	15916
+=	281	*identifier*	45845
Else	382		

Using the information from the table above we decided to use a binomial distribution over an alphabet of size k to generate the characters of 300 strings of length n. For a given k and n, we also ran the tests with different values for the lower bound. The tests were run on a 2.4 GHz Intel Core 2 Duo computer with 2GB of RAM and 4MB of L2 cache. See fig. 4, 5, 6 and 7.

The set of source-codes used to analyze the frequency of apparition of the tokens is available from the authors by request. The implementation of the algorithms and the random string generator using a binonmial distribution can be found at http://sites.google.com/site/rccuam/lcs.

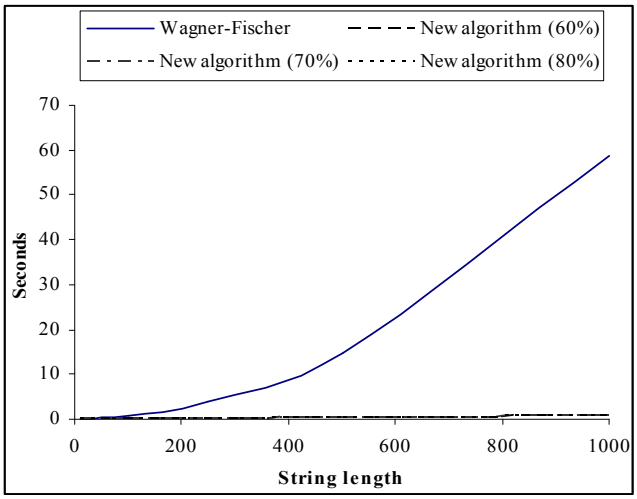

Fig. 4. Running time for $k = 1$

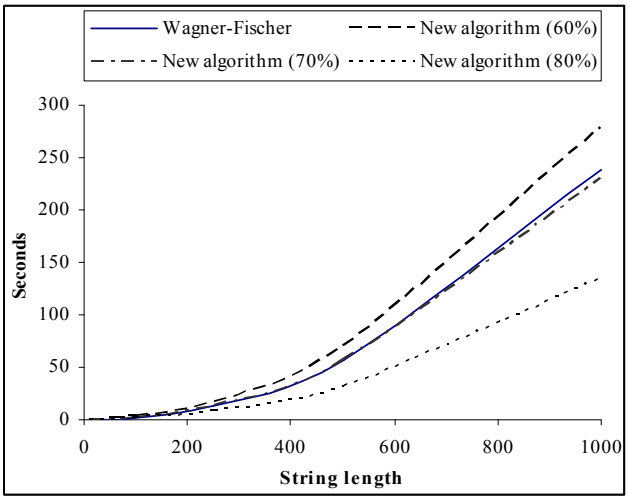

Fig. 5. Running time for $k = 2$

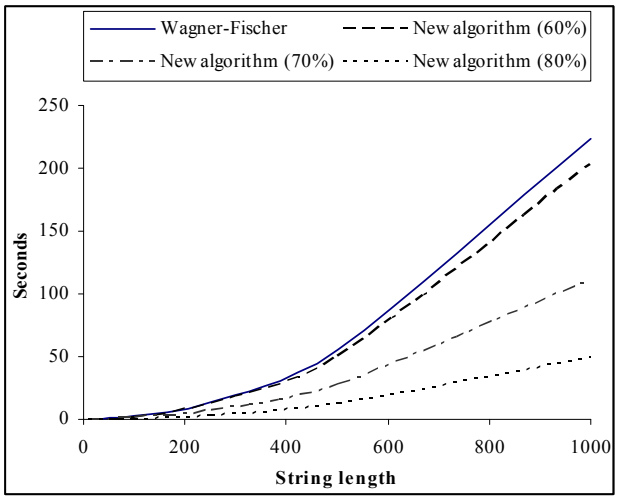

Fig. 6. Running time for $k = 8$

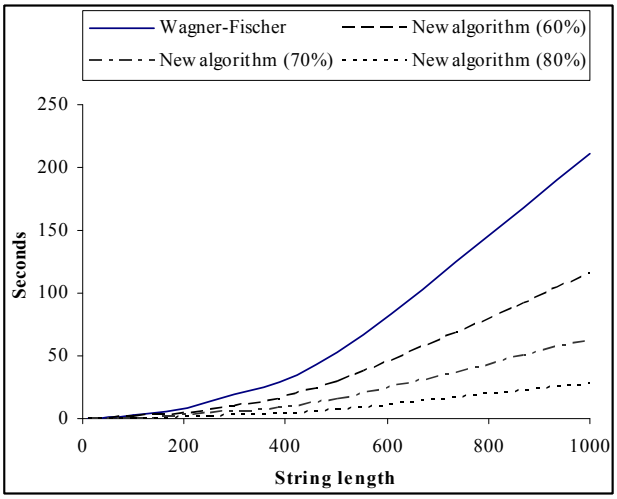

Fig. 7. Running time for $k = 32$

We also ran the algorithm with the tokenized source-codes as input using different lower bounds. These codes were written by students of two different groups of the same introductory course. There were 12 assignments per group and these were not the same for both groups. See Fig 8.

From the tests we can observe that the breadth-first search approach is more effective than dynamic programming for extremely similar strings and that branch and bound considerably speeds up the execution when it is possible to quickly determine that the strings are not similar enough. For relatively similar strings the branch and bound value plays a major role, performing better for values of 70% or more. It appears that out worst case, almost as fast as dynamic programming, occurs when we reach but not exceed the lower bound since branch and bound becomes less effective and we are forced to calculate the exact length of the longest common subsequence.

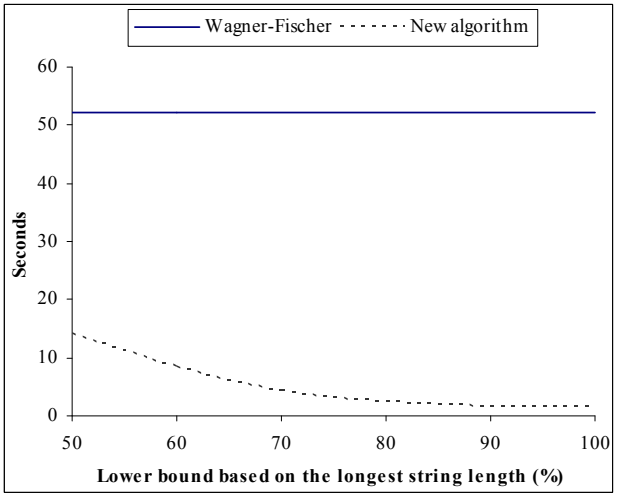

Fig. 8. Running time for finding similar pairs of 1154 source-codes

However, the most visible slow down is caused by an increase in the length of the strings. This is because the number of different longest subsequences greatly increases along with the length of the input strings [9]. This means that the algorithm will still run in O(*mn*) for difficult input strings or small lower bounds.

IV. Conclusion

In this paper we presented a simple and efficient algorithm for string-based plagiarism detection in a set of source-codes that are expected to be either very similar or greatly differing. In particular, it finishes in a few seconds even for relatively large sets of source-codes of student assignments. The proposed algorithm outperforms dynamic programming except when the longest common subsequence length reaches but does not exceed a bound around 70% or less of the length of the longest input string. However, the loss of performance in this case is negligible.

The obtained results also indicate the importance of searching for algorithms that successfully take advantage of the specific properties of a problem, even if not they are not asymptotically faster than previously known algorithms. Additionally, many of these problem-specific algorithms are illustrative and simple enough to be taught in undergraduate programming courses, as well as easy to incorporate into open source plagiarism detection tools.

References

[1] C. K. Roy, J. R. Cordy, and R. Koschke, “Comparison and evaluation of code clone detection techniques and tools: A qualitative approach,” Sci. Comput. Program., vol. 74, no. 7, pp. 470–495, May 2009.

[2] M. Ahmadzadeh and E. Mahmoudabadi, “Pattern of Plagiarism in Novice Students’ Generated Programs: An Experimental Approach,” Journal of Information Technology Education, vol. 10, p. IIP 195–IIP 205, 2011.

[3] R. A. Wagner and M. J. Fischer, “The String-to-String Correction Problem,” J. ACM, vol. 21, no. 1, pp. 168–173, Jan. 1974.

[4] G. Navarro, “A guided tour to approximate string matching,” ACM Comput. Surv., vol. 33, no. 1, pp. 31–88, Mar. 2001.

[5] C. Rick, “New Algorithms for the Longest Common Subsequence Problem,” University of Bonn, 1994.

[6] L. Bergroth, H. Hakonen, and T. Raita, “A Survey of Longest Common Subsequence Algorithms,” in Proceedings of the Seventh International Symposium on String Processing Information Retrieval (SPIRE’00), Washington, DC, USA, 2000, pp. 39–48.

[7] A. Apostolico and C. Guerra, “The longest common subsequence problem revisited,” Algorithmica, vol. 2, no. 1, pp. 315–336, 1987.

[8] B. W. Kernighan, The C Programming Language, 2nd ed. Prentice Hall Professional Technical Reference, 1988.

[9] R. I. Greenberg, “Fast and Simple Computation of All Longest Common Subsequences,” CoRR, vol. cs.DS/0211001, 2002.