INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

Qualitative Analysis and Comparison of Plagiarism-Detection Systems in Student Programs

A. B. Kleiman *T. Kowaltowski*

Technical Report - IC-09-08 - Relatório Técnico

March - 2009 - Março

The contents of this report are the sole responsibility of the authors.
O conteúdo do presente relatório é de única responsabilidade dos autores.

Qualitative Analysis and Comparison of Plagiarism-Detection Systems in Student Programs

Alan Bustos Kleiman and Tomasz Kowaltowski[*]

Instituto de Computação
Universidade Estadual de Campinas (UNICAMP)
Caixa Postal 6176
13084-971 Campinas, SP, Brazil

Alan.Kleiman@students.ic.unicamp.br
Tomasz.Kowaltowski@ic.unicamp.br

Abstract. *Plagiarism in student coursework has become increasingly common and significant effort has been undertaken to face this problem. In this work we focus on the plagiarism in computer programs. We implemented some of the algorithms we discuss so that we could perform a direct and qualitative comparison, with emphasis on the program pre-processing phase. Our main conclusion is that pre-processing may be more important than the comparison algorithm itself and we point to new directions for future work.*

Resumo. *Plágio em tarefas de alunos é um problema que vem aumentando ao longo do tempo e instituições de ensino têm trabalho considerável para enfrentá-lo. Neste trabalho, examinamos o problema do ponto de vista de plágio em tarefas de programação. Implementamos alguns dos algoritmos descritos com a finalidade de efetuar uma comparação direta e qualitativa, com ênfase na fase de pré-processamento de programas. Nossa conclusão principal é que o pré-processamento pode ser até mais importante do que o algoritmo de comparação em si, e apontamos novas direções para trabalhos futuros.*

1. Introduction

Plagiarism in college courses is an increasingly common problem as described for instance in [Maurer et al. 2006]. In our work, and particularly in our tests, we focused on automating detection of plagiarism in computer programs, with emphasis on introductory courses such as basic programming, data structures and so on. This seems to limit somewhat the scope of our work, as these programs are typically not very large and are usually quite straightforward in design. Nonetheless, it is quite clear that our conclusions are applicable to larger programs. We plan to continue this work, optimizing the detection mechanism and attempting to eliminate the weak spots we found. A more complete description of this project can be found in [Kleiman 2007].

[*] The project described in this article was carried out as part of the first author's Master program under the supervision of the second author. The text of the dissertation may be downloaded from `http://www.ic.unicamp.br/˜tomasz/misc/kleiman.pdf`

We studied some of the plagiarism detection systems accessible online and described in the literature. Among them, three systems turned out to be most interesting and were studied in more detail:

- JPLAG developed mainly by Guido Malpohl at the University of Karlsruhe [JPlag 2007]
- MOSS (*Measure of Software Similarity*) maintained by Alex Aiken at the University of Stanford (formerly at the University of California in Berkeley) [MOSS 2007]
- SID (*Shared Information Distance*) developed by the in Bioinformatics Groups of the Universities of California in Santa Barbara and of Waterloo [SID 2007]

In order to carry out the comparison and to evaluate our improvements, we implemented the same algorithms used by JPLAG, MOSS and SID as described in the following sections. It should be noticed that the complete description of the details of the three systems is not publicly available. However they can be used through Internet accessible servers so that it is possible to get some conclusions about their implementation by submitting test data.

2. Preliminaries

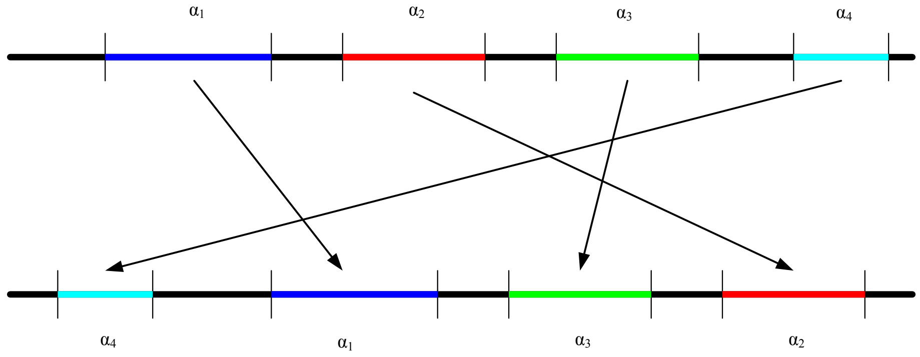
Most existing plagiarism detection systems are based on pairwise comparison of submitted programs and involve basically three steps: (i) source program pre-processing resulting in program string representations; (ii) pairwise comparison between program strings and calculation of similarity measures; and (iii) post-processing which maps comparison results to the original programs.

Pre-processing. Pre-processing tries to reduce the "noise" in a given set of programs. Among possible "attacks" by a plagiarist, some of the most common and obvious techniques involve reformatting whitespace in a program (in languages where whitespace is not significant), changing variable names and modifying comments. Thus most systems will ignore variable names and whitespace, and eliminate comments so that such changes do not influence the results. Besides this "noise" elimination the pre-processing phase transforms the source program into a sequence of *tokens* so that the string manipulation becomes simpler and more efficient. Usually this task can be realized by a lexical analyzer but in some cases, including ours, a syntax analyzer may be required. It will be shown further on that more elaborate pre-processing techniques can improve significantly the quality of plagiarism detection.

String comparison and similarity measures. Both JPLAG and MOSS try to determine a set of common substrings for the string representations of programs. There are several algorithms described in the literature which can be used to determine a set of common substrings for two given strings. The intuitive idea can be seen in Figure 1where α_i's are the common substrings.

String comparison in JPLAG is based on the *Greedy String Tiling (GST)* algorithm described in [Wise 1993]. The algorithm determines a set of substrings such that the sum of their lengths is maximal. Its running time is given by $O(n^2)$ where n is the length of the longer string. In practice the execution time is improved by using the *Running Karp Rabin (RKR)* [Karp and Rabin 1987] algorithm to compute efficiently hashes of substrings; as a result the tiling algorithm is also referred to as *Running Karp Rabin Greedy String Tiling (RKRGST)*. It determines an optimal set of common substrings when the minimal size of

Figure 1. Common substrings of two strings



a substring is allowed to be 1. In practice a larger value, typically 5, is adopted so that optimality is not guaranteed, but the practical results are more satisfactory.

In MOSS string comparison is based on the *winnowing* algorithm described in [Schleimer et al. 2003]. This method also takes advantage of the RKR algorithm to compute its hashes but the number of hashes retained for each string is much smaller than the computed one. The set of hashes is divided into a series of overlapping *windows* of a certain size (typically 5 to 10) and, for each window, the hash with the lowest value is selected as its representative (thus the term *winnowing*). This reduces the number of hash values significantly and allows for more efficient similar substring detection. As a matter of fact, the hash sets can be precomputed for each program separately and kept in a data base instead of being recalculated for each pairwise comparison. Because of this fact the method is particularly interesting when large numbers of programs should be checked.

The SID system described in [Li et al. 2000, Chen et al. 2003] operates in a different way and does not compute directly common substrings. A similarity measure of two programs is computed by determining the amount of information they share. The definition of shared information is based on Kolmogorov complexity (see for instance [Li and Vitányi 1997]) which is not computable. Instead a variant of the Lempel-Ziv [Ziv and Lempel 1977] compression algorithm with a variable-length window and edit distance is used. The authors prove that their shared information algorithm will produce a similarity measure that is at least half of that of any other algorithm and thus theoretically optimal. In spite of its theoretical elegance the practical results are somewhat disappointing in our context; we shall comment this fact further on.

Post-processing. In most systems the post-processing phase maps the common substrings, i.e. token subsequences, to the original source programs and exhibits the results in a visual form convinient for human inspection. It should be stressed that the final conclusion about occurrence or not of plagiarism must be reached through an examination by a human reader. Because of that post-processing is very important in practice. JPLAG and MOSS exhibit results using the same system which is very intuitive. For each pair of suspicious programs, an HTML page is built in which the two source texts appear side by side, with their parts which correspond to common substrings determined in the previous phase highlighted in different colors. In the case of JPLAG this mapping is very precise. In the case of MOSS, due to some loss of information during the winnowing process, the mapping is less precise and some lines of the source programs may not be identified with the right colors. Since the shared information algorithm used by the SID system does not

Figure 2. Example of a plagiarized program segment

Original program	Plagiarized program

if(a<b)
bar(s);
else {
if (t)
s=a;
else
s=foo(b);
}
	{**if**(a>=b) {
	if((!t))
	{ s=foo(b); }
	else
	{ s=(a); }
	} **else** {
	bar(s); }
	}

compute directly the set of common substrings, it must use a different approach, but the details are not available.

3. Improved pre-processing

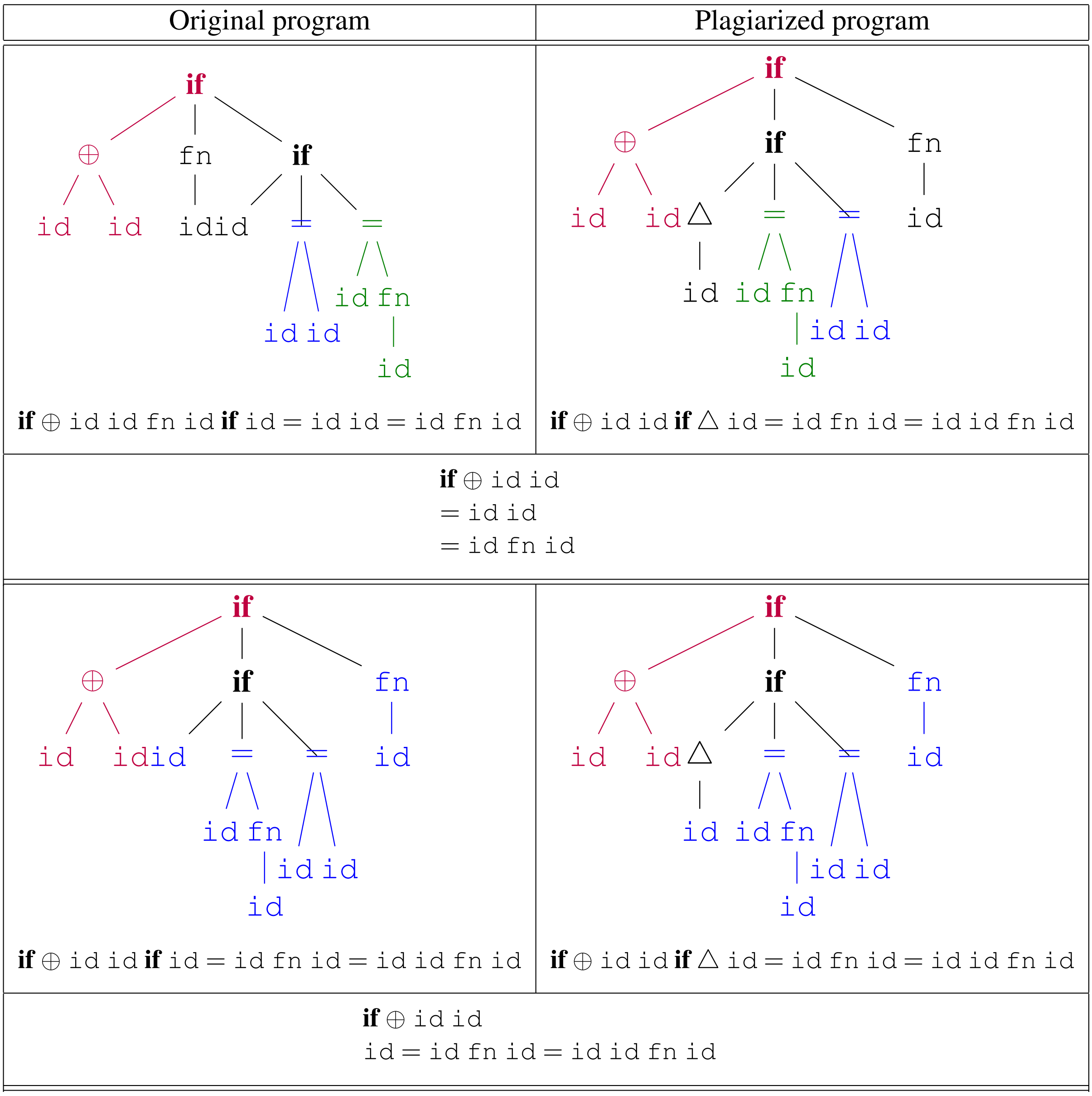
We mentioned already that our work focused on improving the pre-processing stage using various string comparison techniques, but mainly RKRGST as described in [Wise 1993]. Our primary method of improvement involved a normalization of program trees built during syntax analysis. Through this normalization in many cases we were able to achieve more accurate results than the existing systems. This is because another common strategy for plagiarists is to change the order of some statements and declarations, without changing the meaning of the program. They also can substitute one type of structure for another equivalent one (such as the various loop statements in several programming language). Yet another approach is to insert tokens that do not alter the program in any way (such as extraneous parentheses and curly braces in C).

The normalization is achieved by building the program tree for a given program, replacing some constructs by their equivalents and then recursively applying lexicographical reordering to the resulting structure. During the parsing process we remove elements that serve no semantic purpose such as extra parentheses and braces. The lexicographical reordering may change the meaning of the program but this is not important from our point of view. It should be noticed that the comparison method used for ordering the subtrees of the program tree is not very relevant as long as it is applied in a consistent way. In our case the representative strings (token sequences) are given by the preorder traversal of the normalized trees.

The following two figures illustrate possible results of this improved pre-processing. Figure 2 shows an original program segment in C and its plagiarized copy in which there was an obvious attempt to change the program by switching the order of the conditional statement clauses and by adding unnecessary braces and parentheses.

Figure 3 shows the results without and with reordering of the trees (we assume that in both cases unnecessary parentheses and braces were removed). We exhibit both the trees and the resulting token sequences. In this figure $\oplus$ denotes any binary operator, $\triangle$ any unary operator, `id` any variable identifier and `fn` any function name. In each case we also show a set of common substrings as determined by RKRGST when the minimum size of a substring is set to 3. The tree elements corresponding to these common substrings are highlighted with one color each; those in black do not belong to any common substring.

Figure 3. Program trees and common substrings



	Original program	Plagiarized program
Unordered	**if** ⊕ id id fn id **if** id = id id = id fn id	**if** ⊕ id id **if** △ id = id fn id = id id fn id
	if ⊕ id id = id id = id fn id	
Ordered	**if** ⊕ id id **if** id = id fn id = id id fn id	**if** ⊕ id id **if** △ id = id fn id = id id fn id
	if ⊕ id id id = id fn id = id id fn id	

A natural measure of similarity is given by the expression $2 * s/(p_1 + p_2)$ in which s is the sum of the lengths of the common substrings and p_1 and p_2 are the lengths of the two strings. In this example, when no ordering is applied, the total length of the common substrings is 11 whereas with ordering this length is 14. Lengths of program strings are 15 and 16, and the resulting similarity measures for the unordered and ordered trees result in 0.71 and 0.90; the difference is thus quite significant.

Another plagiarism technique which can fool a detection tool is replacing expressions by their equivalents which are not simply unnecessary parentheses. Simple cases like '(a+b)+c' and 'a+(b+c)' could be handled transforming conveniently the tree for expressions. However the details can become quite complicated if many different ways to express equivalent expressions exist; consider for instance '(a+b)/2.0' and '0.5∗(a+b)'. An alternative to this approach, apparently also adopted by JPLAG, is to ignore simple expressions in most contexts and replace them by a unique token.

Figure 4. Program trees and common substrings with simplified expressions

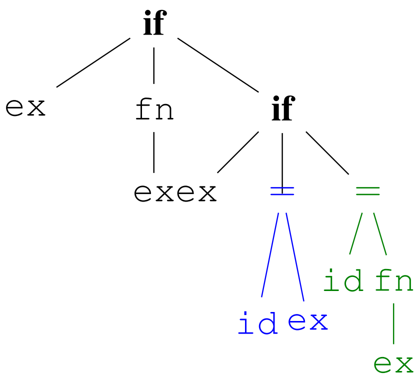
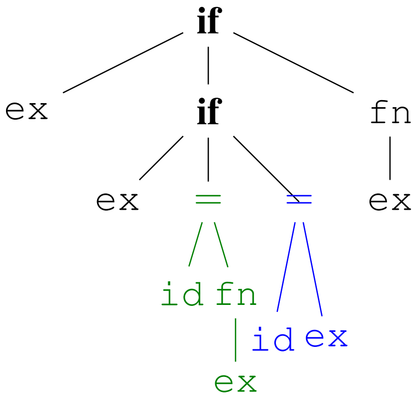
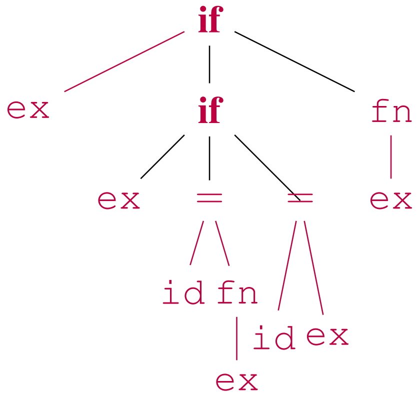
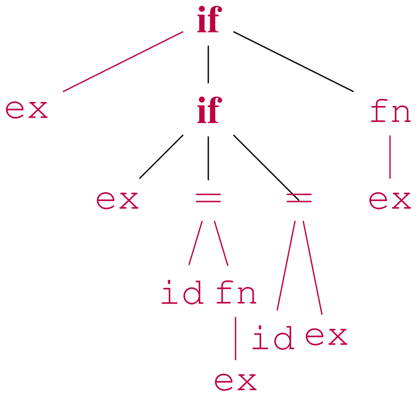
	Original program	Plagiarized program
Unordered	 if ex fn if exex = = id ex id fn ex **if** ex fn ex **if** ex = id ex = id fn ex	 if ex if fn ex = = ex id fn id ex ex **if** ex **if** ex = id fn ex = id ex fn ex
	= id ex = id fn id	
Ordered	 if ex if fn ex = = ex id fn id ex ex **if** ex **if** ex = id fn ex = id ex fn ex	 if ex if fn ex = = ex id fn id ex ex **if** ex **if** ex = id fn ex = id ex fn ex
	if ex **if** ex = id fn ex = id ex fn ex	

Figure 4 shows the results of applying this idea to our previous example (we do keep the expressions that are function calls; simple expressions are represented by ex). Performing the same calculations we get similarity measures of 0.54 and 1.00 for the unordered and ordered trees (the two bottom trees and their strings are identical).

It should be noticed that even without reordering the RKRGST algorithm should match the program segments which changed positions. However the fact that we use minimal size greater than 1 for substrings hampers this possibility. On the other hand, using size 1 would point to trivial one token common sequences "polluting" the results.

4. Implementation and experimental results

We implemented an experimental system, named *PyPlag*, built of two parts. The *parser/lexer*, which produces a string of character tokens for a given C input program, was written as a combination of the *Bison* and *flex* tools and a C compiler. The front-

Figure 5. Comparative results

	GST	GST-E	JPlag	MOSS	SID
01	1.00	1.00	—	—	—
02	1.00	1.00	1.00	0.98	0.75
03	0.98	1.00	1.00	0.23	0.21
04	0.94	0.95	0.50	0.38	0.11
05	0.93	0.95	0.25	—	0.00
06	0.90	0.91	0.97	0.98	0.66
07	0.72	0.61	1.00	0.23	0.15
08	1.00	1.00	0.35	0.98	0.60
09	0.79	0.69	0.25	0.27	0.05
10	0.87	0.84	0.81	0.69	0.26
11	0.82	0.84	0.88	0.29	0.12
12	0.99	0.97	0.81	0.51	0.41
13	0.88	0.86	0.65	—	0.19

end was written in Python and runs some of the comparison methods we studied, and calculates similarities for each pair of programs. Even though the system was aimed at programs in C, it is enough to replace the *parser/lexer* part in order to adapt it to a different programming language. Since our main interest was to study improved pre-processing we did not include some important usual features such as instructor base code and visual display of results, necessary in a production system.

In order to test our ideas we produced a set of 12 programs plagiarized by ourselves from an original base one. Each program in this set was produced with some changes of the base program. These programs are numbered for reference: (01) the base program; (02) copy of the base program; (03) rearrangement of expressions; (04) replacement of some statements by equivalent ones or changing the order of conditional clauses; (05) both changes in 03 and 04; (06) repeating a segment of the program in another place; (07) replacing expressions by other ones, not equivalent; (08) unnecessary braces around statements; (09) to (13) several changes introduced by students who participated in a "plagiarism competition", keeping the programs equivalent. (Notice that 06 and 07 produce syntactically valid but not equivalent programs).

Figure 5 exhibits the results of our implementation when using the RKRGST algorithm with minimal substring size of 5, both with and without expression simplification (columns *GST* and *GST-E*, respectively) compared to those resulting from submissions to the JPlag, MOSS and SID systems for the same set of programs. The first row (base program compared with itself) is exhibited because of the way our system was implemented to check its own results. Empty entries in other rows correspond to MOSS similarity measures below its limit, i.e. are considered very different.

5. Conclusions

Improving pre-processing seems fruitful, and our system achieves results that often appear better than those of other systems. More importantly, our focus is algorithm agnostic: with improved substrings algorithms we can expect to achieve even better results. Furthermore, we believe it is possible to optimize the pre-processing stage further, removing extraneous information and achieving better normalization. One obvious aspect to be considered are expressions which as we mentioned already are more difficult to treat in a systematic way.

Obviously there is a price to be paid for our improvements which is additional processing time and space used to perform more sofisticated pre-processing. However depending on the context in which the processing is carried out it may well be worthwhile.

An interesting aspect of this research was testing the ideas of the SID system. Due to its theoretical foundations it seemed that it should outperform other methods but in practice it did not happen. One problem is that the programs used in our tests were rather short (about 100 lines base program including some comments) and the approximate method used to compute shared information tends to be less reliable in this case. As a result, even for identical programs the similarity indices are less than 1. However the main reason for poor results seems to be that some kinds of transformations used by plagiarists do change the information contents of the corresponding string. For instance, changing the order of two declarations in general does not change the meaning of the program; on the other hand the order of declarations is part of the information contents of each string and thus affects the results. Our reordering could partly improve the results in this case.

References

Chen, X., Francia, B., Li, M., McKinnon, B., and Seker, A. (2003). Shared Information and Program Plagiarism Detection. *IEEE Transactions on Information Theory*, 50(7).

JPlag (2007). `https://www.ipd.uni-karlsruhe.de/jplag/`. Visited on 04/09/2007.

Karp, R. M. and Rabin, M. O. (1987). Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2).

Kleiman, A. B. (2007). Análise e comparação qualitativa de sistemas de detecção de plágio em tarefas de programação. Master's thesis, Instituto de Computação da Universidade Estadual de Campinas. In Portuguese: `http://www.ic.unicamp.br/~tomasz/misc/kleiman.pdf`.

Li, M., Badger, J. H., Chen, X., Kwong, S., Kearney, P., and Zhang, H. (2000). An information-based sequence distance and its application to whole mitochondrial genome phylogeny. *Bioinformatics*, 17(2).

Li, M. and Vitányi, P. (1997). *An Introduction to Kolmogorov Complexity and Its Applications*. Springer Verlag.

Maurer, H., Kappa, F., and Zaka, B. (2006). Plagiarism - A Survey. *Journal of Universal Computer Science*, 12(8).

MOSS (2007). `http://theory.stanford.edu/~aiken/moss/`. Visited on 04/09/2007.

Schleimer, S., Wilkerson, D. S., and Aiken, A. (2003). Winnowing: Local Algorithms for Document Fingerprinting. In *SIGMOD 2003*.

SID (2007). `http://genome.math.uwaterloo.ca/SID/`. Visited on 04/09/2007.

Wise, M. J. (1993). String Similarity via Greedy String Tiling and Running Karp-Rabin Matching. Unpublished.

Ziv, J. and Lempel, A. (1977). A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23(3):337–343.