Source Code Fingerprinting Using Graph Grammar Induction

Istvan Jonyer, Prach Apiratikul, Johnson Thomas

Oklahoma State University, Department of Computer Science
700 N Greenwood Ave, Tulsa, OK 74106

Abstract

In this work we introduce a novel method for source code fingerprinting based on frequent pattern discovery using a graph grammar induction system, and use it for detecting cases of plagiarism. This approach is radically different from others in that we are not looking for similarities between documents, but similarities between fingerprints, which are made up of recurring patterns within the same source code. The advantage to our approach is that fingerprints consist of any part of the text, and has no connection to functionality of the code. Rather, it concentrates on the habits of the coder, which, in most cases, will be very hard to identify by a plagiarizer, and almost impossible to remove.

Introduction

The purpose of a document fingerprinting system is to identifying common segments of text between documents, typically for the purposes of finding citations, references or cases of plagiarism. A special type of document is software source code, which is easily copied or plagiarized, especially that most source code is kept confidential. In this work we develop a new technique for source code fingerprinting using graph grammar induction, and use it to detect plagiarism in academic programming assignments.

Detecting plagiarism, especially in software source code, poses challenges, since the plagiarizing party will most often attempt to hide the traces of the activity. This is most often done by changing details that do not alter the meaning. In an essay, this would mean replacing words with synonyms, and rephrasing sentences. In source code, this would mean changing comments, changing variable names, and rewriting constructs with equivalent ones, like replacing a *for*-loop with a *while*-loop.

Various automated techniques have been proposed for finding similarities in both source code and text documents. The most successful ones dealing with source code examine program structure, realizing that changing variable and function names is irrelevant to the program's functionality, but is effective in fooling a human examiner.

We propose a different technique, in which frequent patterns within the same source code are found, composing a grammar fingerprint. Grammars generated from different source files are then compared to see if the fingerprints in both source files are a close match, thus indicating a possible plagiarism case.

The next section examines previous work on the subject. Then, we introduce our algorithm and outline the experimental methodology. Finally we present our results, and end with conclusions.

Copyright © 2005, American Association for Artificial Intelligence. (www.aaai.org). All rights reserved.

Related Work

Document copy detection techniques have been proposed previously. Some examples include MOSS (Schleimer et al, 2003), SCAM (Shivakumar and Garcia-Molina, 1995) and COPS (Brin, 1995), among which only MOSS (Measure of Software Similarity) addresses the problem of source code copying.

MOSS was developed at Berkeley in 1994 and with different programming languages, including C, C++, Java, Pascal, etc. It works as follows. First, all significant words or phrases are extracted from the source code. All uninteresting or noise data are ignored using the follwing techniques (Schleimer et al, 2003). *Whitespace insensitivity* leaves strings unaffected, but it removes whitespace characters, capitalization and punctuation. *Noise suppression* affects short or common words. Whitespace insensitivity and noise suppression can be defined differently depending on various domains or programming languages.

After the documents are clean of noise, MOSS combines all text in the document together and divides them to small sub-strings. Then sequences of sub-strings of length k are created, all of which are then associated with a sequence number, computed using a hash-function. Finally, these sequence numbers of the two documents are compared. In large documents, we will get a long sequence of indices, which can be shortened by removing a certain set of them, for instance those which can be divided by 4.

Source Code Fingerprinting Using GGI

As mentioned previously, the idea behind our algorithm is that recurring patterns can be found in source code, and any text-based document as well, that have tell-tale signs of the author's identity via his or her habits. One key insight is that capitalization, punctuation and spacing of textual units, among other things, are as important in identifying an author as the text itself. Some systems remove these features and consider them noise, which may even be a requirement for the functioning of the system.

Our approach is to let a data mining system decide what is important in the source code and what is not. That is, we assume that recurring, repeating patterns are important, and comprise the fingerprint, while patterns that occur only once can be disregarded. This is slightly counterintuitive, since code fragments that occur only once may just as well be copied as others. However, we'd like to point out that

our first goal is to identify habits of the code's author, hence developing a fingerprint. We hypothesize that comparing fingerprints will lead to an effective plagiarism detection system.

We are also hoping not to have to define what makes a good fingerprint, but let the data mining algorithm define it for us. If we use a good, general-purpose data miner, we should not need to direct the pattern discovery in any way.

The system will consist of a data mining algorithm capable of generating grammars, which we will call the fingerprint. A secondary algorithm will have the job of computing the similarity of two fingerprints and returning a similarity percentage. The details of the system follow, starting with the data mining system.

SubdueGL

The general-purpose data mining system we are using for generating the fingerprints is called SubdueGL (Jonyer et al 2002). The name stands for Subdue Grammar Learner, and is based on the Subdue system by Cook and Holder (2000) for discovering common substructures in graphs. SubdueGL takes data sets in graph format as its input. The graph representation includes the standard features of graphs: labeled vertices and labeled directed or undirected edges. When converting data to a graph representation, objects and values are mapped to vertices, and relationships and attributes are mapped to edges. We discuss specific graph-representation choices in the next section.

The SubdueGL algorithm follows a bottom-up approach to graph grammar learning by performing an iterative search on the input graph such that each iteration results in a grammar production. When a production is found, the right side of the production is abstracted away from the input graph by replacing each occurrence of it by the non-terminal on the left side. SubdueGL iterates until the entire input graph is abstracted into a single non-terminal, or a user-defined stopping condition is reached.

In each iteration SubdueGL performs a beam search for the best substructure to be used in the next production rule. The search starts by finding each uniquely labeled vertex and all their instances in the input graph. The subgraph definition and all instances are referred to as a substructure. The *ExtendSubstructure* search operator is applied to each of these single-vertex substructures to produce substructures with two vertices and one edge. This operator extends the instances of a substructure by one edge in all possible directions to form new instances. Subsets of similar instances are collected to form new, recursive substructures, and another operator allows for alternative productions. Detailed discussions can be found in (Jonyer 2004).

As all data mining systems, SubdueGL has many options that affect its results. Some of the most important ones are options to disallow alternative productions, recursive productions, productions that contain logical relationships, as well as settings for the number of rules a grammar should contain, both explicitly, and in terms of the size of the input. We discuss these options in a subsequent section where we fine tune these settings using a controlled domain. A sample grammar containing recursion (S_2) and alternative productions (S_3) is shown in Figure 1.

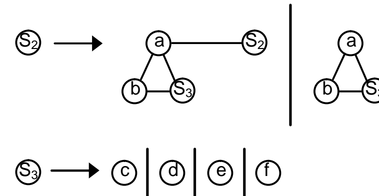


Figure 1 Sample SubdueGL graph grammar productions

Graph Representation of Computer Programs

There are many different ways a computer program can be represented in graph format. Since the specific graph representation has an influence on the performance of the fingerprint generator, we experimented with a number of different approaches, which are shown in Figure 2.

The two main approaches we looked at were: converting the source code directly to graph; and converting the abstract syntax-tree (AST) representation of the program to graph, which was obtained from the compiler.

The AST representation was thought to have great potential, since it represents the structure of the program. To get the AST graph, we compiled the program using a compiler switch that outputs the program's AST, which we then converted to SubdueGL's graph format. Both nodes and edges in the original AST are labeled, so there is a straightforward conversion to graph format.

When converting the source code directly to a graph representation, we also had a few design choices to make. The simplest way to convert source code, or any text for that matter, to graph is to break up the text into tokens and associate them with vertices. Because of the linear nature of text, these tokens are connected via edges. These edges may represent relationships between tokens. Without analyzing the meaning of tokens, however, the only sensible way to label edges is to label them uniformly, perhaps with the label 'next', as shown in Figure 2.

Since we can analyze tokens in source code quite easily and know the function of each token, we decided to experiment with an alternative representation as well, where edges are labeled according to the function of the previous token. If a token is a function name, it is followed by an edge labeled "FUNC", a variable by "VAR", etc.

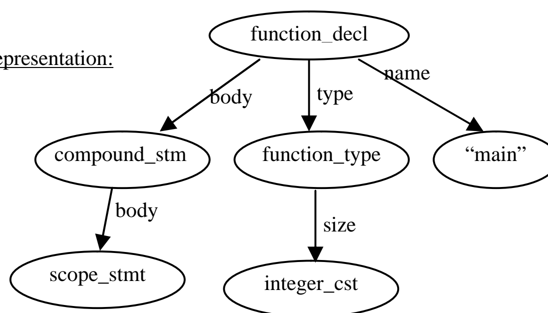
Alternatively, we tried a representation in which the user-defined names (variables, functions, and numbers) are all replaced by a generic token, corresponding to the function of the token, as shown in Figure 2 under "vertex". For the sake of completeness, we combined these previous two representations for a fourth type of graph, shown under "combination" in Figure 2.

In the past we've had great success with representing linear domains, such as DNA and proteins, using the 'backbone'-style graph, also shown in Figure 2. Here, we have a generic backbone, from which the specific tokens hang off. It normally allows for skipping some tokens while including others.

Sample code:

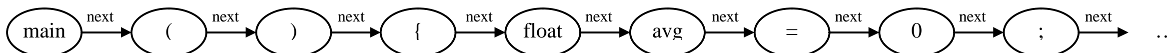
```
main()
{
   float avg = 0;
   ...
```

Partial AST representation:

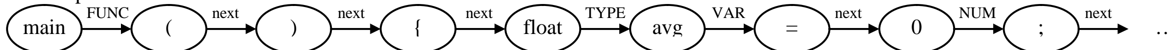


Linear representations:

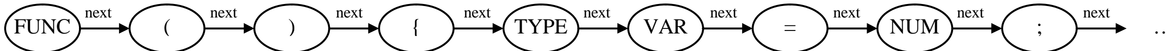
Simple:



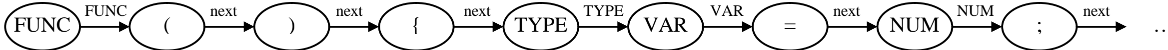
Relationship:



Vertex:



Combination:



Backbone:

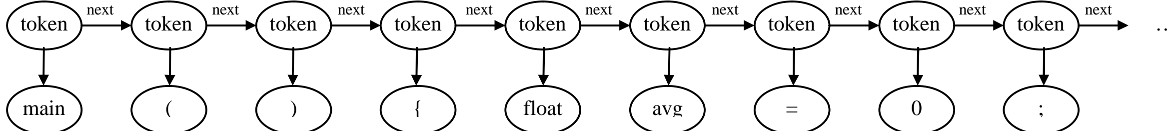


Figure 2 Graph representations of computer programs

Noise Suppression

We also have to consider noise suppression when analyzing source code. In the case of the AST-representations, there is not much noise to speak of, as the compiler already removes all irrelevant information, such as comments, white-space characters, string substitutions, etc. The closest to noise in this case are the actual names of variables, and other user-defined identifiers, which can easily be changed. To address this, we have a version of AST where user defined names are excluded from the tree.

When converting source code directly to a graph we do have to deal with noise suppression more directly ourselves. Generalizing user-defined identifier names was one aspect. Others have to do with completely disregarding text that is irrelevant in terms of the functionality of the code. This would be text that can be easily changed by a plagiarizer, such as strings, and comments. Text that is part of the functional source code is harder to alter in a way that it does not affect the functionality of the code, except perhaps replacing variable and function names, which we addressed previously. Replacing control constructs and assignments is more involved, and most plagiarizers do not go that extra mile. Further, changing the order in which statements appear in a computer program completely alters its meaning, so that is not a possible avenue for cheaters.

Since comments are easy to modify, insert and delete, we do not include any information about them in our graph representation. We do include strings, but not verbatim. Since strings are easily changed, but not easily removed, we decided to include the information that a string is present, but not the actual string itself. Therefore, wherever a string appears, we insert the token “STRING” in our graph representation.

Experimental Results

In this section we describe our experimental results, starting with a controlled domain to see which graph representation and SubdueGL settings are the most productive in defining a fingerprint. We have also delayed the discussion of fingerprint comparison to this section.

Controlled Domain

We selected a C++ program with 720 lines of code, which is an instructor-implementation of a programming assignment. Then, we posed as plagiarizers, and changed

the program to try to fool a grader. We used the two most popular techniques: changing identifier names, and changing the order in which functions appear in the program. Specifically, we changed 3, 6, 9, 12 and 15 identifiers, as well as rearranged the order of functions 1 to 5 times. We converted each of these derivative programs to graph format, used SubdueGL to generate a fingerprint and, finally, compared the fingerprint to the fingerprint of the original program to obtain a percentage similarity between the two.

We performed tests for all graph representations, and with different settings on SubdueGL, comparing the results to MOSS. First, however, let us discuss how the similarity percentage is computed.

To obtain the percentage similarity between two fingerprints, we compare each rule in the first grammar to each rule in the second grammar. Since the right side of each production is a graph, the difference between two rules is measured by the number of graph transformations needed to morph one graph to the other. This is accomplished by a graph isomorphism computation. Even though theoretically this is an NP-complete operation, in practice comparing two full grammars with hundreds of rules typically takes less than a second. This is because the typical grammar rule is small, containing only a handful of vertices.

For each rule in the first grammar we find the closest matching rule in the second grammar. That rule is not considered when matching the rest of the first grammar. All transformation costs are added together, and converted to a percentage, based on the size of the two grammars.

For example, given two grammars G_1 of size 300 (150 vertices + 150 edges total) and G_2 of size 200, it would take 300 transformations to morph one graph to the other. Therefore, 100% corresponds to 300 transformations. If we need 30 transformations to morph one to the other, then we say that they are (300 – 30) / 300 = 90% similar.

Table 1 Controlled domain results: With alternative productions and recursion

Description	Simple	Vertex	Relation.	Combo	AST	MOSS
Itself	100.0%	100.0%	100.0%	100.0%	100.0%	99%
Renamed 3 variables	99.3%	99.6%	98.9%	99.5%	100.0%	99%
Renamed 6 variables	97.5%	99.0%	97.9%	99.0%	100.0%	99%
Renamed 9 variables	98.0%	99.0%	97.4%	99.0%	100.0%	99%
Renamed 12 variables	97.6%	99.0%	97.3%	99.0%	100.0%	99%
Renamed 15 variables	96.7%	99.0%	96.3%	99.0%	100.0%	99%
Rearranged 1 time	44.4%	45.4%	59.4%	56.0%	16.7%	99%
Rearranged 2 times	37.2%	60.9%	60.0%	46.8%	19.2%	99%
Rearranged 3 times	40.6%	48.2%	59.4%	43.4%	17.1%	99%
Rearranged 4 times	47.2%	33.2%	61.2%	39.5%	17.0%	97%
Rearranged 5 times	43.1%	33.2%	63.2%	40.6%	17.3%	97%

Table 1 shows the results obtained such that SubdueGL generated grammars containing alternative and recursive productions. As the table shows, the results are not impressive. While changing variable names does not seem to fool the system, rearranging the order of functions does. Interesting to note that, when examining the grammar rules produced by SubdueGL, no recursive productions appear in any of the grammars. This is expected, since recursive rules in a sequential domain only show up if the pattern described by the production repeats in sequential order. Alternative productions, on the other hand are plentiful.

As the table shows, the AST representation was not sensitive at all to the changing of variable names, which was expected. However, it was extremely sensitive to the rearrangement of the order of functions, which was unexpected. In fact, the AST should have completely removed the effects of that rearrangement, since in graph format a set of functions form a set of connected graphs, which are not linked to one another via edges. What we saw, however, was that the complier had different information based on which function was defined already, and therefore it produced slightly different syntax trees for subsequently defined functions. Since information in the AST is highly structured (each piece of information is contained in additional nodes or subtrees), a small change in available information to the complier can result in large changes in the AST. Unfortunately, this phenomenon defeated the AST-approach as an effective representation.

Experiments using the backbone structure failed to produce meaningful fingerprints. That is, only the backbone structure itself appeared in rules, and not the code tokens. Results from these runs are hence not shown.

Results in Table 2 were generated using settings for SubdueGL that disallow alternative and recursive productions. As we can see, these results are much better. We can see that only the AST representation proved to be too sensitive to the rearrangement of functions, while the others show good results for both variable renaming and function rearrangements. Since all percentages seem reasonable, we have to make a choice on which representation to use based on the subtle differences between them.

The *simple* representation actually performed better than we expected, since renaming and rearranging did not change the similarity very much. In fact, one might argue that it includes more information than the *combo* format and MOSS, since the actual similarity between the files is less than reported by *combo* and MOSS. Namely, when changing a variable, the similarity to the original document decreases, a fact not reflected in our *combination* representation, and in MOSS. The similarity numbers were most proportional to the actual similarity between the file using the *simple* representation.

On the other hand, one might argue that, for the purposes of plagiarism detection, we are not interested in the actual similarity of the programs, but the rather, in the likelihood that one code was copied from the other. In our view this is a matter of personal preference. For instance, when MOSS reports 97%, we expect to see nearly identical codes, which may not be the case at all. A

human examiner often has to look very closely to see the similarities. On the other hand, identical programs have also been reported to be only 97% similar in MOSS. In a third argument one might say that the actual number is irrelevant (e.g., 90% vs 97%), since such high numbers almost certainly indicate a case of plagiarism.

Unfortunately, fingerprints produced by SubdueGL contain about 80 production rules for this problem, which is prohibitively large for inclusion in this paper.

Table 2 Controlled domain results: No alternative or recursive productions

Description	Simple	Vertex	Relation.	Combo	AST	MOSS
Itself	100.0%	100.0%	100.0%	100.0%	99.9%	99%
Renamed 3 variables	98.4%	99.3%	98.4%	99.4%	99.9%	99%
Renamed 6 variables	97.6%	99.3%	97.7%	99.0%	99.9%	99%
Renamed 9 variables	97.4%	99.3%	97.3%	99.0%	99.9%	99%
Renamed 12 variables	97.1%	99.3%	97.0%	99.0%	99.9%	99%
Renamed 15 variables	95.6%	99.3%	95.4%	99.0%	99.9%	99%
Rearranged 1 time	99.1%	97.7%	99.6%	98.7%	41.76%	99%
Rearranged 2 times	99.1%	99.1%	99.5%	98.3%	41.27%	99%
Rearranged 3 times	98.9%	95.1%	98.7%	98.3%	58.57%	99%
Rearranged 4 times	94.1%	90.8%	93.6%	90.4%	57.80%	97%
Rearranged 5 times	94.7%	90.8%	93.6%	90.4%	68.97%	97%

Real-World Experiments

Based on our experience with our controlled domain, we have decided to use both the simple and combo representations in a real-world scenario, in which we compared students' programming assignments from three different assignments from two different classes. In each case 19 programming assignments were analyzed, which were written in the C or C++ programming languages.

Table 3 summarizes the results of comparing each program pair-wise. The upper left triangle shows the first assignment while the lower right shows the second assignment. The third assignment we analyzed did not produce any suspicion of plagiarism, and to save space, is not shown.

We performed the comparison using both the *simple* and *combo* representations, but there were no significant differences between the two approaches, and results shown are those using the *simple* graph format. Most comparisons yielded a similarity between 25 to 45%. The average similarity in assignment 1 was 43%, with a standard deviation of 8.6%. Assignment 2 had an average of 38% and standard deviation of 9.5%. Assignment 3 was also in line with these numbers (35% and 10.6%). These similarities may be a bit higher than one would expect, especially for the purposes of plagiarism detection. However, we must realize that the fingerprints produced may contain a number of similar or even identical components (grammar rules) between fingerprints, which may not point to a case of plagiarism. For instance, almost all code is certain to contain a few *for*-loops with *i* as the iterator: "`for(i=0;i<`". Or even smaller rules, like "[i]" and ") {". Some of these patterns identified by SubdueGL are very common, and are the result of the structure of the programming language, not the habits of the programmer.

Naturally, we are looking for outliers in our matrix of comparisons. Taking into account our statistics described earlier, we conclude that we may only suspect plagiarism if a fingerprint match is greater than the average plus two standard deviations. In our controlled domain we saw that similarities stayed above 90% for very similar programs. We expect this to be reduced if a plagiarizer works hard at fooling the examiner, but it will be very hard to reduce the fingerprint match below the average plus two standard deviations.

For the first assignment the threshold is 60%, for the second it is 57%. As the table shows, there is only one suspicious case at 65% similarity in assignment 1 (programs 8 & 9), and three cases in assignment 2 (programs 4 & 9, 4 & 10 and 9 & 10), indicating three collaborators. These results are corroborated by MOSS, which finds the same suspicious cases: Assignment 1: 8 & 9 are 59% similar; Assignment 2: 4 & 9 are 78% similar, 4 & 10 are 40% similar, and 9 & 10 are 39% similar. All other comparisons in MOSS were much lower, most of them being below 10%. It is interesting to note that when our system finds a high percent match, so does MOSS, and on a lower percent match so does MOSS.

Although each system uses a completely different approach, both came back with the same results. When examining the files manually, it is apparent that 8 & 9 in assignment 1 is a case of plagiarism, and so is 9 & 4 in assignment 2.

The author of program 10 in assignment 2 also committed plagiarism, but took extensive efforts to change the name of each and every identifier in the file, hence the lower match to 4 & 9. Nevertheless, our fingerprint-based system still identified this case of plagiarism, since many other features of the source code remained the same, comprising most of the fingerprint. In all cases discussed here the order of functions was effectively changed.

It should also be noted that the source codes analyzed here were not consulted before the development of the system, and we were unaware of any cases of cheating. They served the purpose of validation, which is also evidenced by the fact that one of the three data sets did not contain cases of plagiarism as reported both by our system and MOSS.

Discussion and Conclusions

In this work we introduce a novel method for source code fingerprinting based on frequent pattern discovery using a graph grammar induction system. This approach is radically different from others in that we are not looking for similarities between documents, but similarities between fingerprints, which are made up of recurring

Table 3 Similarities in two sets of programming assignments

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18			
19	19	33	40	45	47	34	36	43	43	25	38	34	29	21	40	40	35	43		25	**2**
18	31	37	41	42	41	40	41	45	46	39	41	42	42	34	37	38	43		40	39	**3**
17	32	44	35	39	40	27	36	40	41	42	48	31	38	37	36	38		44	47	33	**4**
16	35	49	53	45	54	36	50	50	52	41	49	36	42	40	48		44	42	38	52	**5**
15	33	45	44	44	51	33	43	43	48	43	44	34	42	38		33	45	38	37	31	**6**
14	48	46	40	41	41	25	45	44	46	48	43	30	50		24	23	40	34	44	32	**7**
13	51	47	45	47	44	27	50	51	51	55	50	33		33	40	30	32	42	40	33	**8**
12	18	28	13	25	29	37	25	28	30	31	34		18	42	36	27	**69**	39	46	27	**9**
11	24	47	41	45	44	33	43	47	50	34		**60**	18	49	29	32	**60**	29	45	31	**10**
10	49	51	43	44	44	26	46	46	50		46	45	22	41	35	42	41	35	44	44	**11**
9	34	49	44	52	52	28	48	**65**		36	46	41	13	46	19	54	42	33	36	54	**12**
8	35	46	41	57	52	30	48		48	34	41	35	15	46	23	53	41	20	35	50	**13**
7	37	50	48	51	59	33		42	42	34	45	40	13	49	21	44	43	16	39	46	**14**
6	27	36	23	36	33		37	35	34	37	50	46	18	49	28	35	48	42	45	42	**15**
5	27	41	47	44		32	40	34	32	16	35	30	26	33	36	33	31	41	37	35	**16**
4	30	45	46		41	40	40	39	39	39	41	41	42	41	44	40	41	29	42	40	**17**
3	34	40		34	27	47	45	47	50	38	45	39	15	44	20	50	42	33	37	49	**18**
2	28		39	26	41	40	40	39	39	39	41	41	42	41	44	40	41	29	42	40	**19**
			18	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	

patterns within the same source code. The advantage to our approach is that fingerprintsconsist of any part of the text and has no connection to functionality of the code. Rather it concentrates on the habits of the coder, which, in most cases, will be very hard to identify by a plagiarizer, and almost impossible to remove.

One interesting aspect of the system is that it allows for a "consensus" among a number of source code documents. That is, the average fingerprint similarity of a group of documents will increase if there was a common starting point, or standard element to the assignment, and decrease otherwise. For example, if the assignment involves the implementation of an algorithm given in a textbook, the fingerprints will be more similar (as in assignment 1: 42%) and less similar if an algorithm is to be developed individually (assignment 3: 35%). This in turn allows for the identification of plagiarizers that obtain the code from a different source (e.g., the Internet), whose fingerprint might be much less similar to others in the group.

The choice of using average plus two standard deviations for outlier detection can be revisited in the future, although it is a fairly standard way of determining outliers. Simply picking the extreme outliers and examining them manually until no plagiarism is found is a natural alternative.

We have proven the system both on a controlled domain, and in real-world experiments. A great advantage of this system is that it operates on the text source code, and does not require the program to be compilable. Also, it is language independent, with the exception that strings and comments must be identified by our graph converter.

A natural next step in this line of research is to extend it to non-source code documents, to general document fingerprinting.

References

Brin, S., J. Davis, and H. Garcia-Molina. 1995. "Copy detection mechanisms for digital documents." In *Proceedings of the ACM SIGMOD Annual Conference*, San Francisco, CA.

Cook, D.J. and L.B. Holder. 1994. Substructure Discovery Using Minimum Description Length and Background Knowledge. Journal of Artificial Intelligence Research, Volume 1, 231-255.

Jonyer, I., L. B. Holder and D. J. Cook. 2004. "MDL-Based Context-Free Graph Grammar Induction and Applications," *International Journal of Artificial Intelligence Tools*.

Jonyer, I., L. B. Holder, and D. J. Cook. 2002. "Concept Formation Using Graph Grammars," *Proceedings of the KDD Workshop on Multi-Relational Data Mining*.

Montes-y-Gómez, M., A. López-López, and A. Gelbukh. 2000. "Information Retrieval with Conceptual Graph Matching," *Lecture Notes in Computer Science* N 1873, Springer-Verlag,

Rissanen, J. 1989. Stochastic Complexity in Statistical Inquiry. World Scientific Company.

Shivakumar, N., H. Garcia-Molina. 1995. "SCAM: A Copy Detection Mechanism for Digital Documents." *Proceedings of the Second Annual Conference on the Theory and Practice of Digital Libraries*.

Schleimer, S., D. Wilkerson and A. Aiken. 2003. "Winnowing: Local Algorithms for Document Fingerprinting." *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 76-85.