

DOCUMENT FINGERPRINTING
USING GRAPH GRAMMAR
INDUCTION

By

PRACH APIRATIKUL

Master of Science

Oklahoma State University

2004

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
In partial fulfillment of
The requirement for
The Degree of
MASTER OF SCIENCE
July, 2004

DOCUMENT FINGERPRINTING
USING GRAPH GRAMMAR
INDUCTION

Thesis Approved:

Dr. István Jónyer
Thesis Adviser

Dr. Johnson Thomas

Dr. Debao Chen

Dr. Al Carlozzi
Dean of Graduate College

ACKNOWLEDGMENT

I would like to express my sincere gratitude to my advisor Dr. István Jónyer for all guidance, advice and enthusiasm whenever I was in need, and for identifying this topic. I'm thankful for Professor Johnson Thomas and Debao Chen for examining my thesis. I am also thankful for Paveen Apiratikul, a Ph.D. candidate, for all generous assistance during this time. Finally, I would like to take this opportunity to express my profound gratitude to my beloved parents for financial support, my relatives and my fiancée for all inspiration, encouragement and patience during my studies at OSU.

TABLE OF CONTENTS

Chapter	Page
1 INTRODUCTION	1
2 RELATED WORK.....	3
2.1 DOCUMENT FINGERPRINTING AND MOSS	3
2.2 SUBDUE.....	5
3 DOCUMENT FINGERPRINTING USING GRAPH GRAMMAR INDUCTION (DFGGI)	9
3.1 CONVERTING MODULE	10
3.2 MATCHING MODULE.....	13
4 EVALUATION	16
4.1 EXPERIMENTAL SETUP	16
4.2 EXPERIMENTAL RESULTS.....	20
4.3 DISCUSSION.....	29
5 CONCLUSIONS.....	33
REFERENCES.....	35

LIST OF TABLES

Table	Page
TABLE 3.1 EXAMPLE OF TEXT RELATIONSHIP.....	11
TABLE 4.1 RELATIONSHIP IN DIFFERENT KINDS OF TEXT PATTERN.....	17
TABLE 4.2 THE EXPERIMENTAL RESULTS	20
TABLE 4.3 COMPARISON EACH STUDENT’S ASSIGNMENT 1 WITH MOSS	23
TABLE 4.4 COMPARISON EACH STUDENT’S ASSIGNMENT 1 WITH DFGGI.....	24
TABLE 4.5 COMPARISON EACH STUDENT’S ASSIGNMENT 2 WITH MOSS	25
TABLE 4.6 COMPARISON EACH STUDENT’S ASSIGNMENT 2 WITH DFGGI.....	26
TABLE 4.7 COMPARISON EACH STUDENT’S ASSIGNMENT 3 WITH MOSS	27
TABLE 4.8 COMPARISON EACH STUDENT’S ASSIGNMENT 3 WITH DFGGI.....	28
TABLE 4.9 COUNTING NUMBER OF FUNCTION	31
TABLE 4.10 COMPARISON SIZE BETWEEN ORIGINAL AND REGISTERED DOCUMENT	32

LIST OF FIGURES

Figure	Page
FIGURE 2.1 GENERAL ARCHITECTURAL FOR DOCUMENT FINGERPRINTING	3
FIGURE 2.2 FINGERPRINTING SOME SAMPLE TEXT	5
FIGURE 2.3 SUBDUE'S DISCOVERY ALGORITHM [6]	6
FIGURE 2.4 INPUT GRAPH	7
FIGURE 2.5 FIRST PRODUCTION GENERATED BY SUBDUEGL	8
FIGURE 2.6 SECOND PRODUCTION BY SUBDUEGL	8
FIGURE 2.7 LAST PRODUCTION	8
FIGURE 3.1 ARCHITECTURAL OF DFGGI	9
FIGURE 3.2 EXAMPLE OF A CONCEPTUAL GRAPH	10
FIGURE 3.3 CONVERTING MODULE'S ALGORITHM	12
FIGURE 3.4 GRAPH ISOMORPHISM	13
FIGURE 3.5 EXAMPLE OF GRAMMAR G1 AND G2	14
FIGURE 3.6 PSEUDO CODE FOR COMPUTING THE TRANSFORMATION COST	15

1 Introduction

Nowadays, digital documents are very easily counterfeited especially in software source code. People tend to represent someone else's work or online documents as if they were their own. For example, in schools and universities, students seem to occasionally plagiarize their assignments from their friends or the Web. It is becoming a serious issue to prove whether someone's document is his own work or not.

Few decades ago, document fingerprinting techniques were proposed to detect and reveal the similarity between documents. These techniques such as MOSS [6], and SCAM[9] work well in finding the complete or partial duplication of documents by comparing a desired document with registered documents. However, most document fingerprinting techniques only focus on detecting the similar text pattern between documents, but are not interested in detecting the relationships among text elements. In practice, to show that documents are similar, document fingerprinting has to compare both text pattern and text relationship. **Text patterns** are words, phrases or an alphabet in the document. **Text relationship** is the association between two text patterns in the document. These can be represented by graphs, where nodes act as text patterns and edges act as text relationships [5]. For example, the phrase "*Joe loves Mary*" can be represented in a graph as $[Joe] \rightarrow (subj) \rightarrow [love] \rightarrow (obj) \rightarrow [Mary]$, where phrases in "[]" represent text patterns, and phrases in "()" represent text relationships).

Document fingerprinting using both text pattern and text relationship methods can evaluate the similarity between two documents more effectively. For example, phrase 1: "*Jim is a dog*" and phrase 2: "*Jimi's dog*" are compared to each other. If document fingerprinting focuses only text pattern, such as MOSS, phrase 1 will be converted to a

set of {"*Jimisdog*"} because only significant words are chosen. More detail is described in section 2.1), and phrase 2 will be converted to a set of {"*Jimisdog*"}). Thus, the comparison of both sets, both phrases are totally the same. However, the meaning of both phrases is absolutely different. On the other hand, if document fingerprinting involves both text pattern and text relationship detection, phrase 1 can be represented with a graph like $[Jim] \rightarrow_{(subj)} [is] \rightarrow_{(obj)} [dog]$, and phrase 2 can be represented with a graph like $[Jimi's\ dog] \rightarrow_{(null)}$. Clearly, both graphs are different.

In this paper, we will transfer documents into graph format that will be represented as nodes and edges. Next, we will find their graph grammar using Subdue [2] (section 2.4). Finally, we will measure the similarity between their graph grammars by using the concept of graph isomorphism [7], and graph transformation.

The next section will explain document fingerprinting, and, in particular, MOSS [section 2.1]. Subdue is described in section 2.2. Our method and document fingerprinting algorithm using graph grammar induction is discussed in section 3. Finally, section 4 gives a conclusion and future work.

2 Related Work

To understand how document fingerprinting works, section 2.1 explains it in terms of MOSS [6], one leading document fingerprinting algorithm. In section 2.2 we will introduce Subdue, a program using for graph grammar extraction.

2.1 Document Fingerprinting and MOSS

Document fingerprinting is a technique for detecting document-based textual similarity. As the document is becoming more available electronically, it is easily copied. Therefore, this method is widely used to detection of copyright violations and plagiarism not only for registered documents--documents contained in a database--but also for documents available over internet. More information about document fingerprinting is available in [3], [6] and [9]. The general architecture for document fingerprinting is shown in Figure 2.1

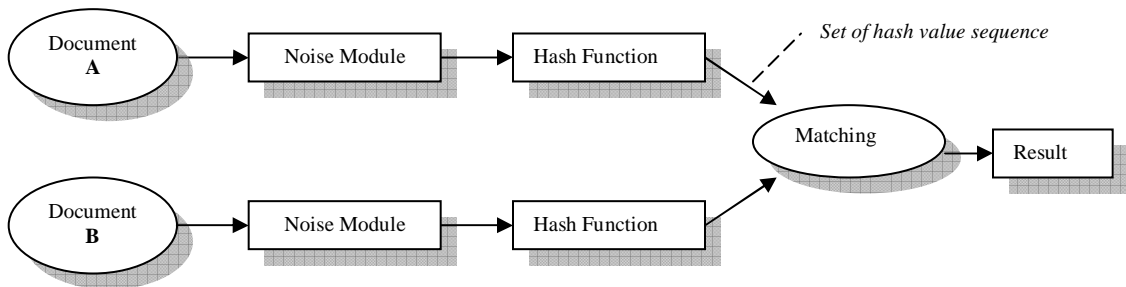


Figure 2.1 General architectural for document fingerprinting

One well known document fingerprinting algorithm is MOSS, which stands for Measure Of Software Similarity. It was developed at Berkeley in 1994. MOSS works primarily on software source code, as the name suggest. It can work with different programming languages, including C, C++, Java, Pascal, etc.

MOSS works as follows. First, it extracts significant words or phrases from the desired documents. This means that all uninteresting or noise data are ignored by applying whitespace insensitivity, and noise suppression [3]. **Whitespace insensitivity** leaves strings unaffected. Normally whitespace insensitivity ignores whitespace characters, capitalization and punctuation. **Noise suppression** is short or common words that we are not interested in, such as “the”, “a”, etc. However, 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 by the length of k-gram as in Figure 2.2(b), 2.2(c). The length of k-gram is the number of alphabets in each sub-string and is individually defined by each user. Then the sequence of sub-string of length k-gram is created. The index number for representing each sub-string is added by using hashing function shown in Figure 2.2(d). Finally, these sequences of index numbers of the two documents are compared to find similarity between the two documents. However, in large documents, we will get a long sequence index. To solve this problem, MOSS chooses only a conditional sequence list of indices, which is defined by user. For example, in Figure 2.2(e), we consider only the sequence of hash values which can be divided by 4 ($0 \pmod{4}$).

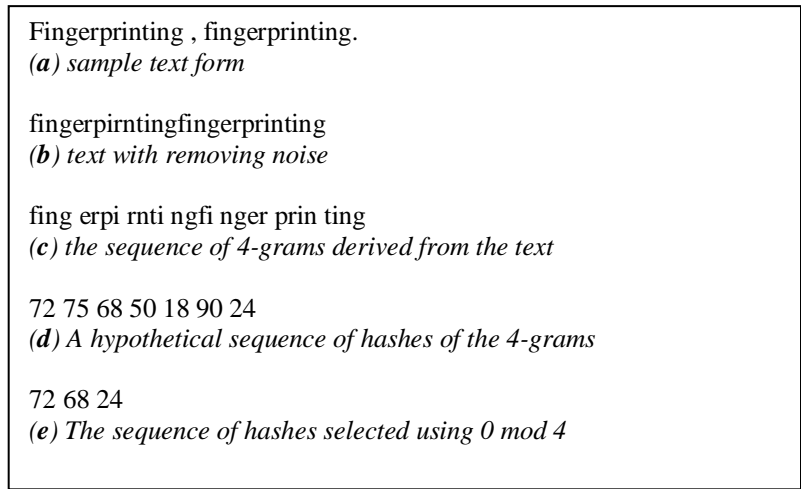


Figure 2.2 Fingerprinting some sample text

2.2 Subdue

Subdue [2] is an algorithm for discovering knowledge in structural data. It can discover interesting and repetitive subgraphs in a labeled graph representation using the minimum description length (MDL) principle, which seeks to minimize the description length of the entire data set. The substructure discovery algorithm used by Subdue is a beam search. Subdue's main discovery algorithm is shown in figure 2.3. Subdue first extends finds all possible single-vertex substructures in another input graph. Each substructure is then evaluated using the Minimum Description Length heuristic, and inserted in the ChildList, in order by the heuristic value [6]. Then the main search is called, which extends each substructure in all possible ways. is called, and return the best substructure in ChildList which contains the minimum MDL value and also its length is less than beam width. By this step, a substructure is extracted and original graph is compressed by replacing all instances of discovery substructures as a single vertex, which represents the substructure's instance. Subdue will repeat this process and will until it

terminates when no more substructures are discovered found. The search returns the best substructure, which has the minimum MDL value. After this step, the best substructure is extracted and replaced by a single vertex in the original input graph, which represents the substructure's instance. This way the graph is compressed. More details about Subdue can be found in [1], [2], [4], [5], and [6].

```

Subdue (Graph, Beam Width, MaxBest, MaxSubSize, Limit)
  ParentList = {}
  ChildList = {}
  BestList = {}
  ProcessedSubs = 0

  Create a substructure from each unique vertex label and its single-
  vertex instances; Insert the resulting substructure in ParentList

  while ProcessedSubs <= Limit and ParentList is not empty do
    while ParentList is not empty do
      Parent = RemoveHead( ParentList)
      Extend each instance of Parent in all possible ways
      Group the extended instance into Child substructures
      for each Child do
        if SizeOf (child) <= MaxSubSize then
          Evaluate the Child
          Insert Child in ChildList in order by value
          if Length (ChildList) > BeamWidth then
            Destroy the substructure at the end of ChildList
        ProcessedSubs = ProcessedSubs +1
        Insert Parent in BestList in order by value
        if Length( BestList ) > MaxBest then
          Destroy the substructure at the end of BestList
        Switch ParentList and ChildList
  return BestList

```

Figure 2.3 Subdue's discovery algorithm [6]

In our work, we use *SubdueGL* [1, 4, 5], an algorithm based on Subdue to generate recursive graph grammars. SubdueGL has more operators that allow SubdueGL more potential performance into extracting graph grammars. One of its most interesting operators is the *Recursify Substructure* search operator which can detect and create recursive productions. This operator adds the connecting edge to the substructure and collects all possible chains of instances during the discovery operator.

RecursifySubstructure starts by checking each instance of the substructure to see if it is connected to any of its other instances by an edge. If so, a recursive production is possible. If a recursive production is found to be the best at the end of an iteration, each such chain of subgraphs is abstracted away and replaced by a single vertex [1, 4, and 5]. See figure 2.5 for an example of a recursive production.

Now, we will give us see an example of SubdueGL’s operation. Consider the input graph shown in figure 2.4, which depicts an artificially domain. In this graph, circles represent vertices, and lines represent edges. Also, we assume that all edges have the same label, which are not shown for a cleaner appearance.

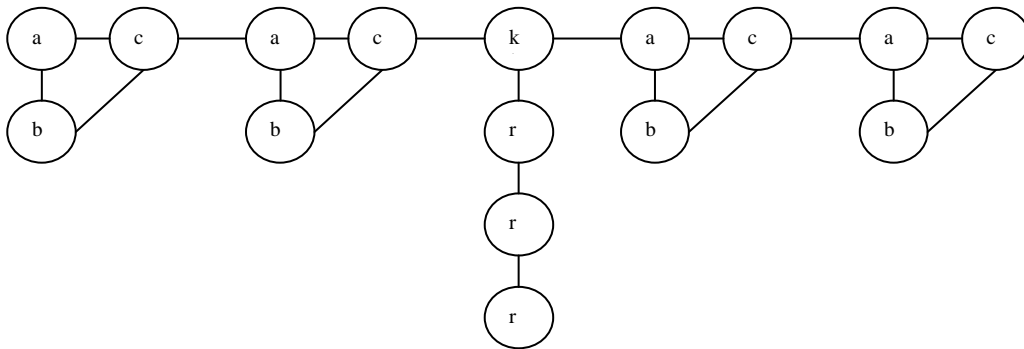


Figure 2.4 Input graph

SubdueGL starts out by collecting all the unique vertices in the graph and expanding them in all possible directions [4]. In the above graph, we see a subgraph having vertices $\{a, b, c\}$ which appear four times, and vertex ‘r’ appears three times. Clearly, subgraph $\{a, b, c\}$ is the most common substructure in this input graph. Thus, SubdueGL gives this subgraph the highest frequency in the top rank. Then SubdueGL try to generate the first rule (substructure) from this subgraph. Executing the *RecursifySubstructure* operator results in the recursive grammar rule show in Figure 2.5. The production covers two lists of the substructure.

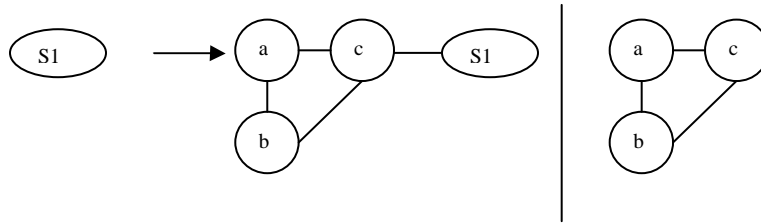


Figure 2.5 First production generated by SubdueGL

After SubdueGL generated the first production, vertex 'r' now is the highest frequency. Then SubdueGL uses same method to discover the substructure of vertex 'r' showing on Figure 2.6. However, in the next iteration, SubdueGL cannot find any recurring substructures that can be abstracted out. The rule in figure 2.7, substructure S3, therefore becomes the final production of SubdueGL.

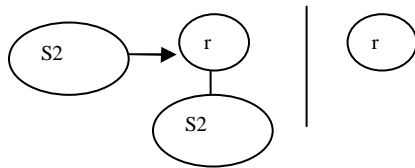


Figure 2.6 Second production by SubdueGL

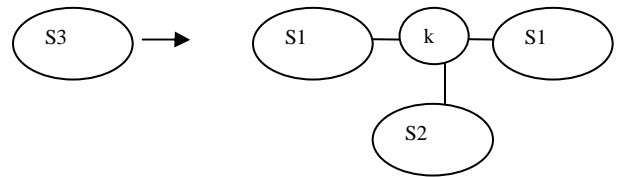


Figure 2.7 Last production

3 Document Fingerprinting using Graph Grammar Induction (DFGGI)

In the introduction, we addressed the problem caused by using only text patterns to detect similarity between documents. Clearly, it would be better to detect similarity not only by text pattern, but also by text relationship. Our solution is to represent documents using graph grammars. Here, both documents are compared by both text pattern and text relationship.

Our algorithm is as follows. First, DFGGI translates documents into graph format. Text patterns are represented by vertices while text relationships are represented by edges in the graph. Then, SubdueGL is used to extract the graph grammar. Finally, the graph grammars of both documents are compared and the result is the percentage of similarity between graph grammar, and hence the documents. DFGGI has two modules which are converting modules and matching module. The converting module translates documents into graph grammar format. Then, the matching module finds out the percentage of similarity between graph grammars. Figure 3.1 shows the architecture of DFGGI.

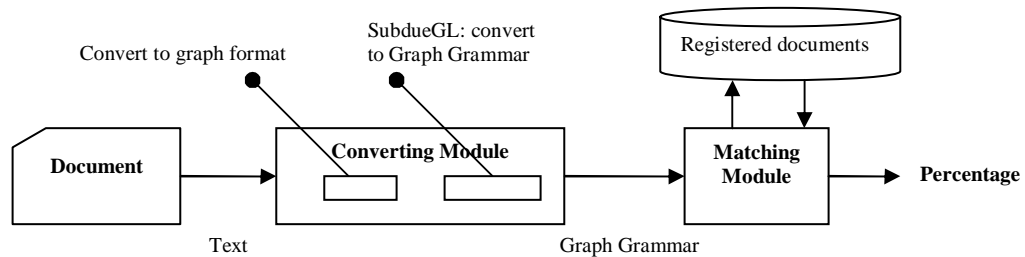


Figure 3.1 Architectural of DFGGI

3.1 Converting Module

The converting module is the first step in DFGGI. The goal of this module is to translate documents into graph grammar format. For now, we only focus on the C programming language. The converting module can be divided to two processes. In the first process documents are translated into conceptual graphs. A conceptual graph is a network of concept nodes and relation nodes [8]. The concept nodes represent entities, attributes, or events (text patterns). The relation nodes identify the kind of relationship between two concept nodes (text relationship). For example, a line `printf("hello");` is represented by a conceptual graph shown in figure 3.2. In the second process, graphs are converted to graph grammar using SubdueGL.

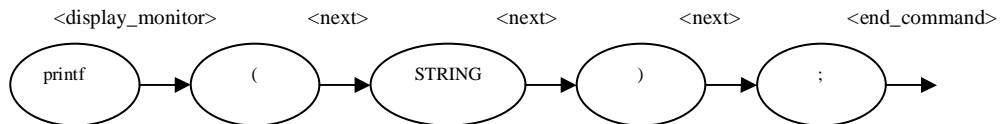


Figure3.2 Example of a conceptual graph

In the first process, the document is converted into graph format. After getting a C language document from the user, unnecessary data is removed from the document before it is translated into graph format. Insignificant data in the C language such as comments in single lines (`//`), and block comments (`/*...*/`) are eliminated. By the assumption that any text under the double quote symbol (“...”) has the same structural meaning in the language, text in double quotes is replaced by the word “STRING”. For example, the phrase `printf("hello");` will be changed to `printf(STRING);`.

Text pattern	Text relationship
printf	<display_monitor>
'('	<next>
STRING	<next>
)'	<next>
','	<end_command>

Table 3.1 Example of text relationship

After eliminating insignificant data, the next step is to find out text relationship between two text patterns. In the converting module, each document is divided to text patterns. For example, this phrase “printf(STRING);” is separated into five text patterns, which are printf, (, STRING,), and ;. In this process, text relationship between two text patterns is also added. The example of relationship between text patterns is shown in table 3.1. The result of this process is represented in terms of a graph consisting of linearly connected vertices. Here, text patterns are written in brackets, and text relationships are written in parentheses. The result is: [printf] → (*display_monitor*) → [(] → (next) → [STRING] → (*next*) → [)] → (*next*) → [;] → (*end_command*) shown in figure 3.2.

Figure 3.3 shows the converting module’s algorithm. The first step of the algorithm is to initialize VFL the Variable and Function List (VFL). This list will contain the default variables, and functions defined in the C programming language. However, when new variables or new user functions are allocated with in documents, these new variables and functions are also stored in the VFL list. The algorithm has two *while* - loops which are the core of the algorithm. Each line and each token will be extracted one by one in order with in those loops. After a token is extracted from a line, a relationship will be defined by its own token. Then the graph will be generated by representing a vertex as a token, and an edge as its relationship. In some conceptual graphs, a graph may

be defined in different ways. Tokens and their relationships may be changed in their label to reflect structural relationships. This modification makes the graph more structurally meaningful. This process will be presented in section 4.1.

```

VFL ← give the default variable and function defined in C-language into VFL list
while (end of file) do
  line ← CatchLine; // take a new line from Input file
  while(end of line) do
    Eliminate insignificant data
    token ← CatchToken(line); // take a new token from a line
    relation ← Relation(token); // define its relation to the token

    /* it is optional definition in different conceptual graphs
    if (token = new variable) then
      VFL ← AddVariable(token); // add a new variable to the VFL list
    if (token = new user function) then
      VFL ← AddFunction(token); // add a new function to the VFL list
    if(token ∈ variable set or function set) then
      token ← ChangeLabel(token, VFL); // change token value to our defined structure
      relation ← ChangeLabel(relation, VFL); // change relation value to our define structure
    */
    Token and its relations will be stored as the vertex and edge of the conceptual graph

```

Figure 3.3 Converting module's algorithm

Finally, SubdueGL is called to find the graph grammar. As discussed in section 2.2, a graph grammar is the abstract of graph driven by the MDL heuristic. It contains substructures that compress the input graph the best. Therefore, comparison between two documents in their abstract of graphs is more effective than in their original one, because an abstract of graph contains only the most significant details of a graph. That is a reason why graphs are modified to graph grammar before making a comparison. Other reasons are that the size of the graph format is large when it is translated from the large document. To store and compare between whole graphs formats are not a good idea always feasible. However, the size of a graph grammar normally decreases after it is

converted from the original graph. Thus, the idea of reducing a graph's size before comparing to each other is an appropriate one.

3.2 Matching Module

The matching module is the last stage of DFGGI. The inputs of this stage are two graph grammars. These graph grammars are compared with one another. Alternatively, one document can be compared to other registered graph grammars in a database. The result is the percentage of similarity between the two grammars.

In order to match graphs, the matching module applies both concepts of graph isomorphism and graph transformation. *Graph isomorphism* is a mathematical perspective on the process whereby two or more domain structures are structurally reconciled [7]. It means that if both graphs are isomorphic, they have the same conceptual graph. For example, in figure 3.4, graph 1 and graph 2 are isomorphic to each other. Thus, it can be concluded that these graphs are the same –one hundred percent similar. However, in most cases, isomorphism between two graphs is not obvious. The concept of graph transformation might be applied in order to check the isomorphism of these graphs.

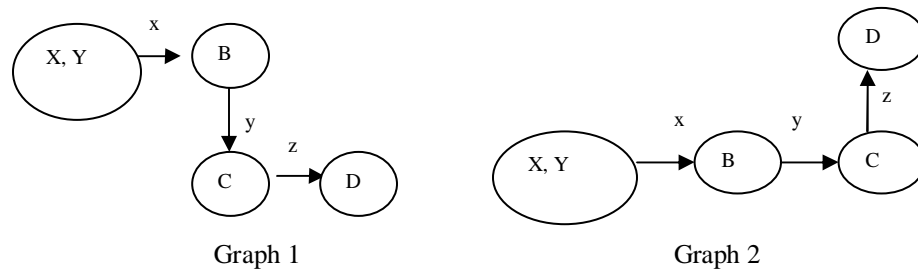


Figure 3.4 Graph isomorphism

Graph transformation is the concept for modifying any vertex and edge in a graph. However, cost for translating should be considered as well. In this work, we define

cost of changing (adding or deleting) a vertex or an edge as one unit. In addition, the cost of changing labels (the contents in a vertex) is counted by the number of modified labels divided by a number of whole labels in a vertex. For example, if we add a vertex and delete an edge in a subgraph, transformation cost is equal to two units—one is counted for adding a vertex and another for deleting an edge. Meanwhile, if we change two labels within a vertex consisting of four labels, the transformation cost of this change is counted as 0.5 unit (modified labels divided by total labels equal $2/4 = 0.5$). To have a clear picture, this process is shown in figure 3.5.

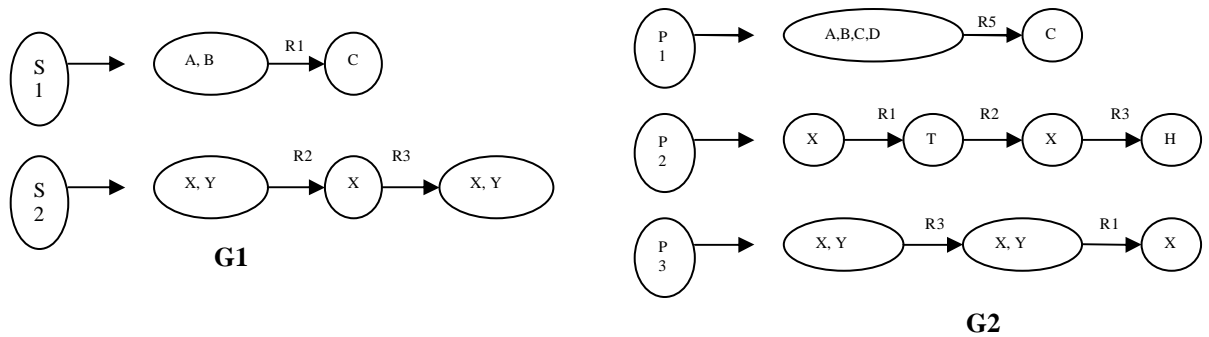


Figure 3.5 Example of Grammar G1 and G2

As shown in figure 3.5, grammar G1 has two rules (productions) which are S1 and S2, and grammar G2 has three rules which are P1, P2, and P3. In matching these graph grammars, rule S1 in grammar G1 is first compared with all rules in grammar G2 which are P1, P2 and P3. The pair of rules which requires the least transformation cost is selected. By the concept of transformation, the cost in the first comparison between subgraph S1 and rules in grammar G2 are 1.5, 6, and 4 units respectively. Certainly, P1 is chosen due to its least transformation cost. Then, rule S2 is compared with all rules in grammar G2 excluding P1 which is already chosen to be a pair of rule S1. The cost of the next comparison is 4, and 2 units respectively. Therefore, rule P3 is selected to be the

closest isomorphic pair of rule S2. Finally, the total transformation cost is computed by adding all translation costs of pair (S1, P1) and of pair of (S2, P3) which equal to $1.5 + 2 = 3.6$ units. Thus, grammar G1 and G2 take 3.6 units to convert into isomorphic graphs.

```

sub1 ← number of sub_graph in G1; // number of sub graph in Grammar 1
sub2 ← number of sub_graph in G2; // number of sub graph in Grammar 2
Total_cost ← 0; // initial total cost for transformation
for(i=0;i<sub1;i++)
{
  MimMatch ← 9999.99;
  for(j=0;j<sub2;j++)
  {
    if(G2[j] != NULL) // if subgraph j in Grammar 2 still exist
    { MatchCost ← Transformation (G1[i],G2[j]);
      if(MinMatch > MatchCost)
      { MinMatch ← Matchcost;
        minsub ← j; }
    } // end if
  } // end for
  save(i, minsub); // Save a pair that has lowest transformation cost
  Delete(G2[minsub]); // Delete subgraph "minsub" in Grammar 2, because it already used
  Total_cost ← Total_cost + sub_cost;
} // end for
return Total_cost // All cost for doing transformation

```

Figure 3.6 Pseudo Code for computing the transformation cost

To mention the percentage of similarity between two graph grammars, S_c is a quantity representing how many concept graphs in grammar G1 and G2 has in common.

Based on Dice coefficient [8], S_c is expressed as

$$S_c = \frac{2n(G_c)}{n(G1) + n(G2)}$$

Where $n(G)$ is the number of vertices and edge in grammar G, and $n(G_c)$ is the degree of connection of the same concept graph in the original graphs G1 and G2. $n(G_c)$ is calculated by subtracting the transformation cost from the $n(G)$. Therefore $n(G_c)$ of grammar G1 equals to $n(G1)$ subtracted by translation cost; and $n(G_c)$ of grammar G2 equal to $n(G2)$ subtracted by translation cost. However, to estimate the similarity between two grammars, we will use the less value of $n(G_c)$ between two grammars. For example above, the similarity between grammar G1 and G2 is $[2 * (4.5) / (8) + (15)] = 0.3913$ or 39.13% similarity.

4 Evaluation

In this section, the performance of the DFGGI algorithm is evaluated using both artificial and real-world domains. In the artificial domain, we test different kinds of conceptual graphs. Sensitivity, frequency of plagiarisms, and file sizes after fingerprinting are analyzed. Our empirical documents were modified in different ways to test DFGGI's performance such as renaming variables and rearranging the structure of the document. Our real-world domains involve testing DFGGI is tested in those programming exercises to detect similarity of student's assignments. In both the artificial and real-world domains the results from DFGGI are compared to the result from MOSS.

4.1 Experimental Setup

As mentioned in the last section, DFGGI first translates documents into conceptual graphs. Then SubdueGL is called to extract its graph grammar. Finally, graph grammars are compared, and the results of similarity between graph grammars can represent the similarity between documents. For DFGGI, the conceptual graphs should be defined carefully to get the right result in various domains. In this section, we define four different kinds of conceptual graph as the benchmark in our study to handle C-language documents.

```
...  
i = 10;  
i = ABC(j); //ABC() is the user function  
printf(" Number %d",i);  
...
```

Figure 4.1 Part of a C language document

1. “**Simple conceptual graph**” is the conceptual graph, in which any relationship between text patterns is not considered. The conceptual graph in this case has the same structure to that of the original document. Thus, all vertices in this graph are extracted exactly from all tokens in the document. All edges or text relationships have the same

value. Thus, all of them are represented as the “NEXT” value. For example, the part of the document in figure 4.1 is translated to the simple conceptual graph shown in figure 4.2.

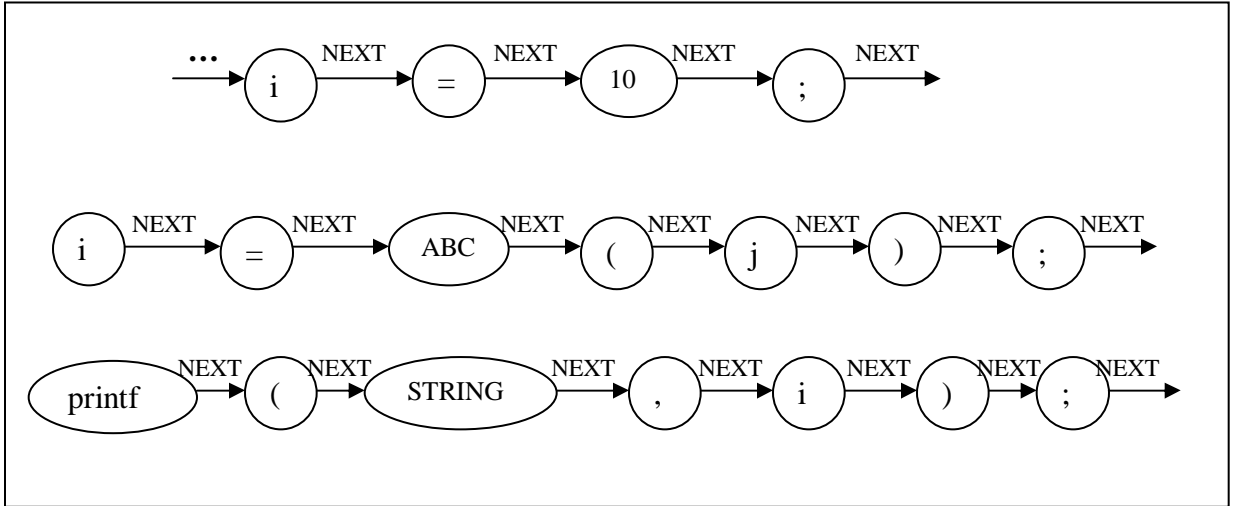


Figure 4.2 Example of simple conceptual graph

2. **“Relative conceptual graph”** is the conceptual graph in which text relationships are also considered. This conceptual graph is quite similar to the simple conceptual graph with the exception that all vertices have their own text relationships instead of using the same value as in the simple conceptual graph. Thus, each edge between vertices is different, and is defined by the specific structure of its vertex. After testing our algorithm using C language documents, we found reasonable vertex structures and their relationship as shown in Table 4.1.

Text pattern	Relationship
Frequency function in C language	“FUNC”
Any function which is created by user	“UFUN”
Any declared Variable	“TYPE”
Any variable in the C language document	“VARI”
Any numeric number	“NUM”
Anything else	“NEXT”

Table 4.1 Relationship in different kinds of text pattern

Relative conceptual graphs are generated in the same way as simple conceptual graphs. However, specific edges or text relationship are added followed by its vertex. For example, the document in figure 4.1 is translated to the relative concept graph shown in figure 4.3.

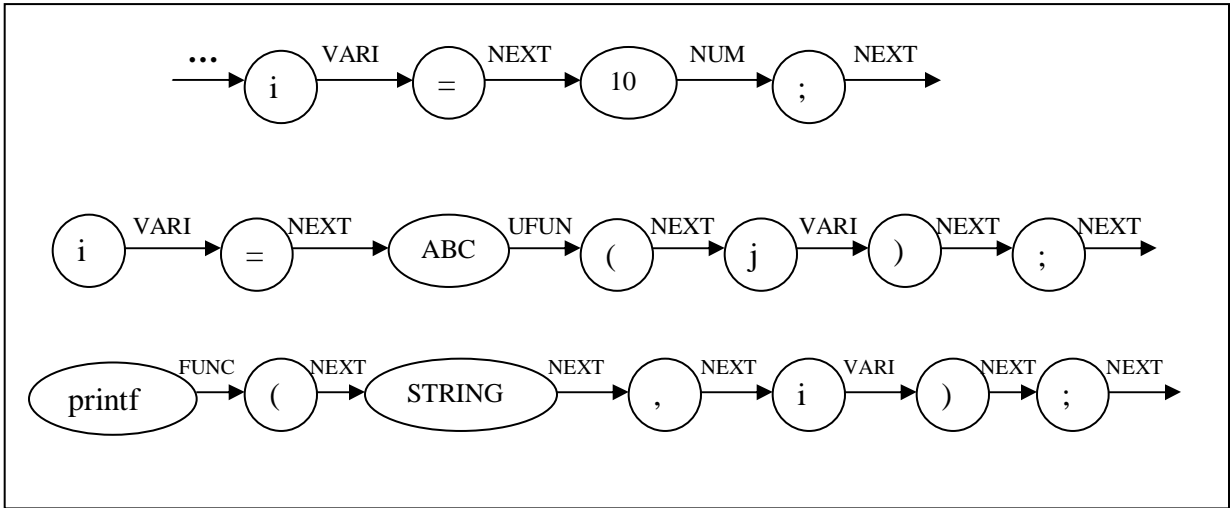


Figure 4.3 Example of relative conceptual graph

3. **Vertex conceptual graph**, is the conceptual graph, in which text relationship is not considered. However, label values in each vertex have better defined structure than the previous conceptual graphs. For example, when a graph is generated, the algorithm gets the token “i”, a variable in the C language document. Vertex conceptual graph does not consider “i” as the label value on its vertex. It changes “i” into “VARI” value to make its vertices have better defined structure behavior, as shown in Table 4.1. Thus, only basic vertex structure remains in this graph. However, all edges or text relationships have the same value as the edges in a simple conceptual graph. The example document in figure 4.1 is translated to the vertex concept graph is shown in figure 4.4.

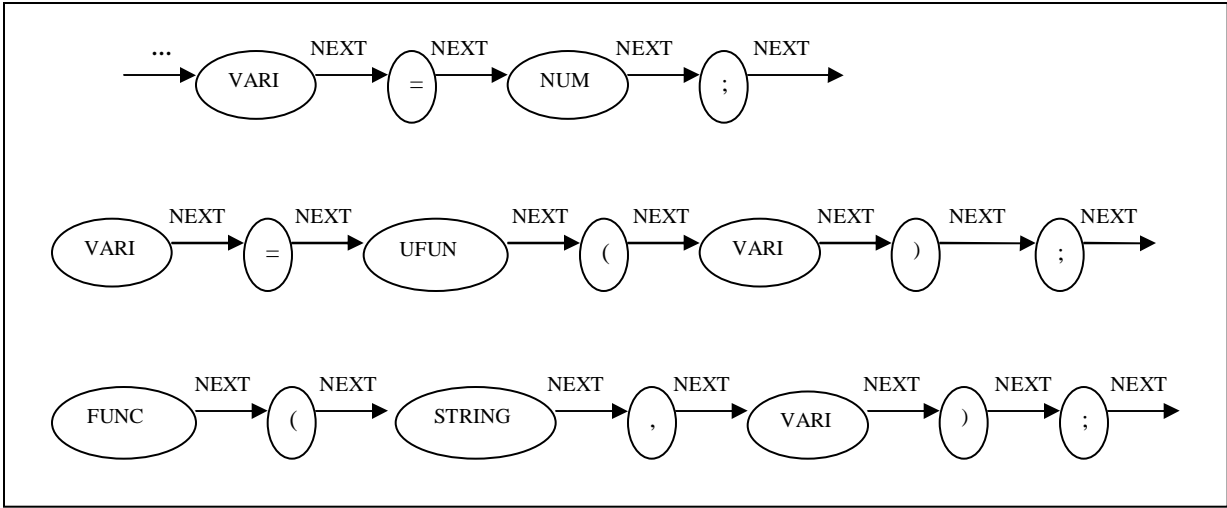


Figure 4.4 Example of vertex conceptual graph

4. “**Complex conceptual graph**”, is the conceptual graph which is the combination of relative and vertex conceptual graph. All vertices in complex conceptual graphs are generated in the same manner as vertices in vertex conceptual graphs. However, edges in complex conceptual graph are generated by the same structure as edges in relative conceptual graphs. The document in figure 4.1 is translated into the complex concept graph, shown in figure 4.5.

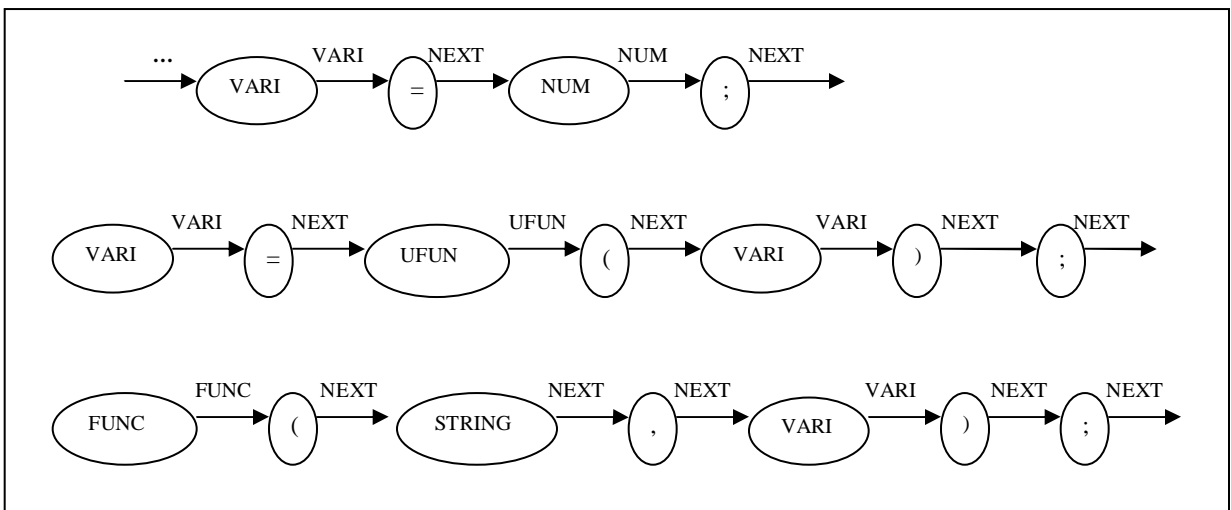


Figure 4.5 Example of complex conceptual graph

4.2 Experimental results

Several experiments using C language documents are conducted in order to test the performance of the DFGGI algorithm. The goal of our algorithm is to detect the similarity between documents. So, we create all empirical documents which are modified from the same original document. By this way, all artificial documents have the same structure as the original. In some documents, only variables are renamed. In others the document structure is rearranged. In this section, we have run our empirical documents through DFGGI in different conceptual graphs as mentioned in section 4.1. The results of our algorithm are compared to the result from MOSS [Aiken, 2004] shown in table 4.2.

Description	SIMPLE	VERTEX	RELATIVE	COMPLEX	MOSS
Itself	100%	100%	100%	100%	99%
Renamed 3 Variables	99.31%	99.84%	98.90%	99.52%	99%
Renamed 6 Variables	98.51%	99.20%	97.94%	99.01%	99%
Renamed 9 Variables	97.99%	99.12%	97.42%	99.01%	99%
Renamed 12 Variables	97.58%	99.12%	97.26%	98.96%	99%
Renamed 15 Variables	96.66%	99.12%	96.35%	98.96%	99%
Rearranged 1 times	47.72%	66.49%	74.77%	65.16%	99%
Rearranged 2 times	56.28%	64.73%	76.88%	65.08%	99%
Rearranged 3 times	55.85%	67.35%	76.06%	63.84%	99%
Rearranged 4 times	56.16%	66.41%	75.44%	59.33%	97%
Rearranged 5 times	54.83%	61.47%	76.47%	60.39%	97%

Table 4.2 the experimental results

As mentioned before, no written description of MOSS is available. Therefore, we cannot explain the results from MOSS in detail. Results from table 4.2 shows that both DFGGI and MOSS have the same results, in case of comparing original document which documents, in which only variables are modified. The average percentage in this case is around 97-99%. However, the results of DFGGI are dissimilar with those of MOSS when the original document is compared to rearranged documents. DFGGI has varied results from different kinds of conceptual graphs in rearranged documents. Using relative conceptual graphs, the highest percentage is around 76%; however, the lowest percentage


appearing in simple conceptual graph is around 54%. In this case, MOSS has the average percentage of similarity around 98%.


Next, DFGGI is tested on real-world domains. The tested documents come from students who turned in these documents as their solution for programming assignments. Three exercises are selected for testing the DFGGI performance. The first exercise is a sorting algorithm that selects the best sorting algorithm for a specific input data set. The second one is the implementation of the RSA cryptosystem. In both exercises above, the average size of student's documents is small, containing around 300 lines in their source codes excluding free space. The last exercise, a pseudo code interpreter, is more complicated. The average student's programming solution in this exercise is approximately 600 lines excluding free space. In each exercise, only 19 programming assignments written in C-language are selected for testing .

Based on the results in the artificial test shown in table 4.2, the relative conceptual graph was found to be the best conceptual graph to handle the C-language domain. Therefore, in this real-world test, DFGGI will generate graphs in the form of relative conceptual graphs. Before discussing the results, we make the assumption that students may plagiarize someone else's work if the percentages of similarity in both files are more than 10% on MOSS algorithm. Because MOSS only considers the similarity by texture comparison, we believe that 10% texture similarity in both files is enough to make a conclusion. However, DFGGI works in different way. Text pattern and text relationship are also considered in DFGGI. In this case, the percentage of similarity in both files has to be more than 50% in order to conclude that both documents have either structure

similarity or text similarity. The DFGGI and MOSS results are shown in table 4.3 through table 4.8.

FileX/ FileY	1		2		3		4		5		6		7		8		9		10		11		12		13		14		15		16		17		18	
	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B
19	4	3	1	1	1	2	32	30	2	2	0	0	9	6	35	32	35	32	0	0	2	2	0	0	2	2	0	0	0	0	6	5	0	0	1	2
18	12	1	11	8	9	9	0	0	8	6	0	0	0	0	8	7	7	8	6	4	0	0	0	0	0	0	0	0	6	8	9	5	0	0		
17	1	1	1	1	1	2	1	1	3	6	0	0	9	4	1	1	1	1	1	1	4	5	0	0	1	2	9	4	0	0	1	2				
16	10	11	11	13	7	13	5	7	11	17	0	0	14	13	0	0	0	0	3	4	3	3	2	8	1	1	8	8	10	11						
15	11	12	11	12	8	13	0	0	9	13	0	0	18	15	5	5	3	4	4	4	0	0	0	0	0	0	11	9								
14	9	12	5	6	5	9	2	2	7	11	0	0	0	0	0	0	3	4	3	4	0	0	0	0	2	2										
13	0	0	0	0	5	9	2	2	1	2	0	0	0	0	0	0	2	2	0	0	3	3	2	8												
12	0	0	0	0	9	6	0	0	0	0	0	0	2	2	2	2	2	2	0	0	0	0														
11	0	0	3	3	3	5	3	4	5	8	0	0	2	2	3	4	3	4	1	1																
10	4	4	8	9	4	7	1	1	4	6	0	0	5	4	1	1	1	1																		
9	4	4	5	5	4	6	31	32	4	5	0	0	13	10	58	59																				
8	4	4	6	6	2	3	35	36	4	5	0	0	0	0																						
7	14	17	12	12	7	15	3	4	10	16	0	0																								
6	0	0	0	0	0	0	0	0	0	0																										
5	6	4	9	8	17	20	5	4																												
4	0	0	4	4	2	3																														
3	15	9	0	0	0																															
2	8	7																																		

 → MOSS detects some similarity between these two documents (at least 10% on both comparison files).

 → Both MOSS and DFGGI agree that these pair documents have some similarity.

A → percentage similarity between file Y to file X
B → percentage similarity between file X to file Y

Table 4.3 Comparison each student's assignment 1 with MOSS

FileX/ FileY	1		2		3		4		5		6		7		8		9		10		11		12		13		14		15		16		17		18			
	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B		
19	49	0	45	61	55	15	59	58	54	31	51	23	42	77	54	41	52	31	49	50	48	49	55	17	54	0	46	49	37	0	49	20	46	61	51	48		
18	57	0	44	63	57	20	50	52	59	40	59	34	44	81	57	47	54	36	51	57	49	54	61	25	66	0	55	61	47	0	55	29	52	70				
17	64	0	51	52	63	9	54	39	60	23	64	22	48	68	57	29	58	23	54	42	55	42	60	7	66	0	57	46	47	0	61	17						
16	49	8	44	88	59	48	46	75	56	63	53	54	42	100	49	65	52	61	49	81	51	82	55	46	59	0	44	77	47	0								
15	12	100	7	100	13	100	7	100	10	100	9	100	6	100	7	100	9	100	8	100	8	100	10	100	24	100	6	100										
14	57	0	45	57	53	10	52	48	59	32	57	25	44	75	56	40	54	30	49	48	55	53	59	18	62	0												
13	28	100	9	100	15	100	15	100	18	100	16	100	12	100	16	100	17	100	12	100	14	100	18	100														
12	56	24	41	94	54	52	48	86	55	70	51	62	39	100	52	77	55	73	44	85	49	88																
11	58	0	51	65	62	21	54	53	66	42	58	29	48	81	60	46	61	39	56	57																		
10	57	0	47	61	60	18	53	50	58	33	59	28	47	79	59	44	55	32																				
9	55	6	45	81	58	38	52	72	60	58	55	47	46	100	53	61																						
8	50	0	45	74	55	28	53	66	59	50	56	41	46	93																								
7	64	0	57	38	70	0	62	27	73	17	63	2																										
6	51	9	36	79	48	36	43	70	48	53																												
5	57	10	47	86	63	45	55	77																														
4	58	0	47	64	59	19																																
3	53	23	42	97																																		
2	60	0																																				





 → DFGGI detects some similarity between these two documents (at least 50% on both comparison files).
 → Both MOSS and DFGGI agree that these pair documents have some similarity.
A → percentage similarity between file Y to file X
B → percentage similarity between file X to file Y

Table 4.4 Comparison each student's assignment 1 with DFGGI

FileX/ FileY	1		2		3		4		5		6		7		8		9		10		11		12		13		14		15		16		17		18			
	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B		
19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	0	0	0	0	0	2	2	5	4	0	0	0	0	0	0	3	2	0	0	0	0	2	2	0	0	0	0	0	0		
17	0	0	0	0	0	0	0	0	4	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0						
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	21	38	3	3	0	0	0	0	0	0								
15	3	2	0	0	0	0	0	0	2	1	0	0	6	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0										
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0										
13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0												
12	1	2	0	0	0	0	3	5	0	0	0	0	0	0	0	0	2	2	1	2	0	0	3	5														
11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0																
10	0	0	0	0	0	0	36	40	6	5	0	0	0	0	0	0	0	0	39	43																		
9	2	2	0	0	0	0	78	77	7	5	0	0	0	0	0	0																						
8	0	0	0	0	0	0	0	0	0	0	0	0	2	3																								
7	5	5	0	0	0	0	0	0	0	0	0	0																										
6	0	0	0	0	4	2	0	0	0	0																												
5	0	0	0	0	0	0	5	7																														
4	2	2	0	0	0	0																																
3	0	0	0	0																																		
2	0	0																																				


 → MOSS detects some similarity between these two documents (at least 10% on both comparison files).


 → Both MOSS and DFGGI agree that these pair documents have some similarity.

A → percentage similarity between file Y to file X
B → percentage similarity between file X to file Y

Table 4.5 Comparison each student's assignment 2 with MOSS

FileX/ FileY	1		2		3		4		5		6		7		8		9		10		11		12		13		14		15		16		17		18	
	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B
19	0	100	0	100	0	100	0	100	0	100	0	100	0	100	0	100	0	100	0	100	0	100	0	100	0	100	0	100	0	100	0	100	0	100	0	100
18	55	34	64	32	45	0	57	42	61	32	61	11	59	45	56	0	62	27	56	55	60	37	63	33	59	38	52	43	66	25	65	34	71	0		
17	0	100	0	100	0	100	0	100	0	100	0	100	0	100	0	100	0	100	0	100	0	100	0	100	0	100	0	100	0	100	0	100	0	100		
16	50	60	57	56	44	0	49	66	53	60	53	33	53	70	59	12	53	50	46	76	55	63	57	58	45	67	45	67	61	53						
15	50	69	58	66	40	0	51	77	56	67	56	45	53	79	55	14	57	63	47	86	50	67	55	65	51	70	46	77								
14	58	45	65	41	34	0	54	48	60	40	70	28	61	55	57	0	62	36	52	60	64	49	61	40	57	45										
13	52	52	54	43	42	0	51	57	55	46	55	25	51	58	54	0	54	40	58	68	54	52	54	45												
12	52	61	51	49	42	0	51	66	54	55	55	34	50	66	53	4	53	48	47	75	53	59														
11	54	56	56	48	39	0	52	61	54	49	62	35	55	64	58	3	55	44	59	71																
10	57	37	59	28	46	0	62	50	62	34	60	11	58	44	61	0	66	32																		
9	46	60	54	57	40	0	57	77	53	59	52	36	49	70	55	10																				
8	34	91	37	85	38	0	36	99	36	85	35	64	35	99																						
7	51	45	60	43	47	0	51	51	54	39	57	21																								
6	46	76	47	66	34	0	49	79	50	71																										
5	50	59	51	48	45	0	50	65																												
4	50	44	57	39	45	0																														
3	17	100	18	100	18	100																														
2	50	63																																		


 → DFGGI detects some similarity between these two documents (at least 50% on both comparison files).


 → Both MOSS and DFGGI agree that these pair documents have some similarity.

A → percentage similarity between file Y to file X
B → percentage similarity between file X to file Y

Table 4.6 Comparison each student's assignment 2 with DFGGI

File1/ File2	1		2		3		4		5		6		7		8		9		10		11		12		13		14		15		16		17		18			
	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B		
19	1	1	0	0	0	0	0	0	0	0	0	0	5	17	0	0	3	2	0	0	0	0	1	11	0	0	0	0	3	7	0	0	0	0	0	0	5	5
18	8	7	0	0	0	0	0	0	0	0	0	0	18	57	0	0	10	6	0	0	0	0	5	5	0	0	0	0	5	12	0	0	0	0	0	0		
17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	11	11	0	0	0	0	0	0	0	0	0	0	0	0		
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0				
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0						
14	12	4	0	0	0	0	0	0	0	0	0	0	12	17	0	0	12	3	0	0	0	0	11	5	0	0	0	0										
13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0												
12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0														
11	10	8	0	0	0	0	0	0	0	0	0	0	5	16	0	0	5	3	0	0	0	0																
10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0																		
9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0																				
8	3	4	0	0	0	0	0	0	0	0	0	0	6	32	0	0																						
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0																								
6	26	7	0	0	0	0	0	0	0	0																												
5	0	0	0	0	0	0	0	0																														
4	0	0	0	0	0	0																																
3	0	0	25	18																																		
2	0	0																																				

 → MOSS detects some similarity between these two documents (at least 10% on both comparison files).

 → Both MOSS and DFGGI agree that these pair documents have some similarity.

A → percentage similarity between file Y to file X

B → percentage similarity between file X to file Y

Table 4.7 Comparison each student's assignment 3 with MOSS

FileX/ FileY	1		2		3		4		5		6		7		8		9		10		11		12		13		14		15		16		17		18				
	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B			
19	53	69	47	85	45	80	55	47	47	70	64	26	55	24	60	42	49	76	61	22	59	58	52	58	60	0	53	63	54	55	67	13	56	51	59	22			
18	44	90	35	100	37	100	48	72	42	95	54	50	48	49	50	64	42	99	52	44	48	79	46	83	59	29	39	81	43	76	57	34	52	80					
17	46	70	42	84	41	79	54	50	44	71	53	19	47	21	44	40	42	73	53	17	55	58	45	54	56	0	44	59	43	48	55	6							
16	37	100	31	100	33	100	42	89	34	100	56	73	41	66	43	80	35	100	46	62	38	94	43	100	50	42	33	97	38	93									
15	47	63	43	80	44	77	50	42	45	67	60	21	54	22	51	32	46	72	55	15	50	50	52	56	57	0	49	58											
14	50	56	44	72	46	70	55	37	49	61	61	14	51	13	54	26	47	63	56	8	53	44	52	47	56	0													
13	30	100	26	100	25	100	31	85	28	100	28	100	39	66	33	65	34	78	29	100	43	66	36	96															
12	51	61	44	77	46	75	56	43	48	66	62	19	55	19	54	31	49	71	59	14	52	48																	
11	57	71	49	86	48	81	55	46	49	70	59	20	59	28	58	38	53	78	56	15																			
10	36	90	29	100	32	100	40	72	34	95	49	50	43	51	42	64	34	98																					
9	65	54	55	67	55	62	62	28	60	56	69	7	63	8	68	23																							
8	50	83	41	96	42	94	51	61	44	85	52	34	51	39																									
7	35	80	34	100	33	96	38	60	32	84	41	34																											
6	41	93	34	100	37	100	46	76	38	97																													
5	69	58	50	66	54	65	58	28																															
4	48	71	44	89	43	84																																	
3	57	38	59	63																																			
2	60	37																																					



 → DFGGI detects some similarity between these two documents (at least 50% on both comparison files).
 → Both MOSS and DFGGI agree that these pair documents have some similarity.
A → percentage similarity between file Y to file X
B → percentage similarity between file X to file Y

Table 4.8 Comparison each student's assignment 3 with DFGGI

4.3 Discussion

In the artificial test, it is clear that both DFGGI and MOSS did well in detecting similarities between original and modified documents. The average similarity in both DFGGI and MOSS were around 97-99%, which is a clear indication of plagiarism. However, in documents with rearranged structures, results from DFGGI have lower percentage in detecting similarity compared to those using MOSS. That is because when the document structures are rearranged, it has a big impact on the way graphs are generated. To solve this problem, we have to create conceptual graphs in different ways, and select the best conceptual graph which returns the best result specific for each domain. Clearly, relative conceptual graphs have the best solution in rearranged documents. The average percentage was 76%. However, the average percentage of simple, vertex, and complex conceptual graphs are 54%, 65% and 62.7%, respectively. Therefore, we chose the relative conceptual graph as the graph format for DFGGI for handling C language documents. We also conclude that well defined relationships between vertices make DFGGI perform better.

In real-world experiments DFGGI had a good performance. As mentioned in section 3, DFGGI uses both text pattern and text relationship between documents for comparison. Thus, DFGGI's results present the combination of structure similarity and texture similarity between documents. Document structure is the main idea to solve this problem. DFGGI detects the similarity by comparing text relationship between documents which is represented as an edge in the conceptual graph. By using text pattern comparison, DFGGI also detects the texture similarity between documents.

Due to comparing programming solutions of students for the same problem, their structure could be somewhat similar, while not being due to plagiarism. However, they should not have texture similarity among them. For this reason, DFGGI might come up with false positives, since it cannot tell whether the DFGGI's result is due to structure similarity, texture similarity or both.

In the first exercise shown in table 4.3 and 4.4, DFGGI and MOSS algorithm detected many similar document pairs. Some pairs have the high percentage of similarity in both MOSS and DFGGI algorithm. In this case, we believe that those pairs are duplicated from one another. However, there are some pairs that only are defined by DFGGI or MOSS algorithm as similar pairs. In this case, we have to look at documents closely to determine whether those pairs are copied. For example, in a pair of document number 10 and 11 in exercise 1, DFGGI results in 56% in comparison (shown in table 4.4) which is high percentage of similarity. However, this pair has only 1 percent of similarity on MOSS shown in table 4.3. Focusing on this document pair, we discover that both documents are not like each other on their source code. However, their structures are quite similar to each another. As shown in the table 4.9, the number of functions present in document 10 is approximately equal to the one in document 11. Conclusively, both of documents have the same structure. The same consideration is also used for checking comparison results which both MOSS and DFGGI agree in detection. Focusing on a pair of document number 4 and 9 in exercise 2, and a pair of document number 6 and 18 in exercise 3 (shown on table 4.5 though 4.8), both pair are reported plagiarism by both algorithms. The percentages of similarity in document 4 and 9 in exercise 2 are 77, and 57 reported by MOSS, and DFGGI respectively. And 18 and 50 percentages of similarity

are detected on a pair of document 6 and 18 in exercise 3. After we look at their source code, clearly both pairs are like each other. Thus, MOSS show to good results in these detection. However, DFGGI algorithm also performs well. In table 4.9, it also presents that both pairs carry, to an extent, the same structure.

Document number & Description	Exercise 1		Exercise 2		Exercise 3	
	#10	#11	#4	#9	#6	#18
if	25	28	13	14	40	46
For	28	31	10	9	2	4
While	4	2	6	6	2	3
Declared external functions	10	8	10	10	1	2

Table 4.9 Counting number of function

In assignment 2 and 3 shown in table 4.5, 4.6, 4.7 and 4.8, most of similar pairs tested by MOSS algorithm are also similar pairs when checked by the DFGGI algorithm. MOSS returns only a few document pairs as similar, although all tested documents should have document structure similarity because they try to solve in the same problem. Due to MOSS algorithm, it will return the percentage of similarity when texture similarity was detected between documents. Unlike MOSS, DFGGI can find similarity in those documents. Not only can DFGGI detect plagiarism, but also the similarity of document structure. Comparing to MOSS, DFGGI algorithm has better performance in detecting document structure.

Finally, we would like to mention the registered documents. Registered documents are documents that are in our database, and are prepared to make the comparison with our desired documents. After the desired document was converted to

graph grammar format, we can keep its graph grammar as the registered document. The size of original documents can be reduced this way. In DFGGI, the size of registered document is reduced from original document around 50% as shown in Table 4.10, and it seems to be more decreasing in size when the original documents got bigger. This is the great advantage of DFGGI especially for the documents which are frequently compared.

Original Document	Registered Documents
3k	3k
8 k	5 k
11 k	5 k
14k	6k
20k	8k
26k	8k

Table 4.10 Comparison size between original and registered document

5 Conclusions

Digital documents nowadays are easily plagiarized. Thus, it is important to find ways to detect similarities between documents. Document fingerprinting is a method to do so. Current algorithms only focus on finding text pattern similarity between documents, but not text relationship. To show two of any documents are similar to one another, document fingerprinting must compare both text pattern and text relationship of both documents. Thus, in this work, we present a new concept using graph grammar induction for document fingerprinting, especially in the C language. Initially, the document is converted to conceptual graph. We also define four different conceptual graphs to handle C-language document. Clearly, the relative conceptual graph is the best performance's behavior in this domain. Relative conceptual graph is more consider in text relationship than other conceptual graphs. Thus, we conclude that if relationships between textures are more considered, DFGGI will do better results.

After documents are translated into conceptual graph, SubdueGL is called to extract its graph grammar. Graph grammar can enables the specification of elaborate graphs using simple production rules. Thus, the results of similarity comparison between graph grammars can represent the similarity between documents. Finally, the percentage of similarity between documents is return after our application is operated. As mention in experience, not only can DFGGI detect plagiarize, but also can it detect the document structure. We assume that all programming solution in the same exercise should share document structure in solving problem. However, in the last experience, MOSS almost can not detect any similarity between student's assignments, although they are in the

same exercise. Unlike MOSS, DFGGI is better detecting document similarity in that exercise.

In this paper, we also study in graph characteristics, graph grammar and the graph isomorphism. This research field provides many attractive topics in both theory and application, and is expected to be one of the key fields in document fingerprinting research.

In the future, apart from investigating more conceptual graph relevance detection algorithms to comply with all conditions necessary and to establish a powerful algorithm, several other issues are on our agenda. Also, using DFGGI in different domains, such as human language understanding, may be successful.

REFERENCES

- [1] Jonyer, I., L.B. Holder, and D.J. Cook, “MDL-Based Context-Free Graph Grammar Induction,” Proceedings of the Sixteenth Annual Florida AI Research Society, 2003.
- [2] University of Texas at Arlington, “The Subdue: Knowledge Discovery System.” via <<http://ailab.uta.edu/subdue/>> (3 April 2004).
- [3] Schleimer, Saul, Daniel Shawcross Wilkerson, and Alexander Aiken, “Winnowing: Local Algorithms for Document Fingerprinting,” SIGMOD 2002, 76-85.
- [4] Jonyer, I., L. B. Holder and D. J. Cook, “MDL-Based Context-Free Graph Grammar Induction and Applications,” International Journal of Artificial Intelligence Tools, March 2004.
- [5] Jonyer, I., L. B. Holder, and D. J. Cook, “Concept Formation Using Graph Grammars,” Proceedings of the KDD Workshop on Multi-Relational Data Mining, 2002.
- [6] Aiken, Alex. “A System for Detecting Software Plagiarism.” <http://theory.stanford.edu/~aiken/publications/papers/sigmod03.pdf> (5 April 2004).
- [7] Sabin, Todd. "Comparing binaries with graph isomorphisms. "2004. via <<http://razor.bindview.com/publish/papers/comparing-binaries.html>> (10 April 2004).
- [8] Montes-y-Gómez, Manuel, Aurelio López-López, and Alexander Gelbukh. “Information Retrieval with Conceptual Graph Matching,” Lecture Notes in Computer Science N 1873, Springer-Verlag, 2000.
- [9] Narayanan Shivakumar, Hector Garcia-Molina. “SCAM: A Copy Detection Mechanism for Digital Documents.” in d-lib magazine. Stanford University, Department of Computer Science.
- [10] Richard M. Karp, and Michael O. Rabin, “Pattern-matching algorithms,” IBM Journal of Research and Development, 31(2):249-260, 1987.

VITA

Prach Apiratikul

Candidate for the Degree of

Master of Science

Thesis DOCUMENT FINGERPRINTING USING GRAPH GRAMMAR INDUCTION

Major Field: Computer

Biographical

Personal Data: Born in Bangkok, Thailand, On April 11, 1980, the son of Pew and
Sugunya Apiratikul

Education: Graduated from Debsirin High School, Bangkok, Thailand in May
1996; received Bachelor of Science degree in Computer Science degree in
Thammasat University, Bangkok, Thailand in May 2000. Completed the
requirements for the Master of Science degree with a major in Computer
at Oklahoma State University in July, 2004.