

# Code Clone Detection Using String Based Tree Matching Technique

Ali Selamat and Norfaradilla Wahid  
*Universiti Teknologi Malaysia  
Malaysia*

## 1. Introduction

As the world of computers is rapidly evolving, there is a tremendous need of software development for different purposes. As we can see today, the complexity of the software being developed is different from one to another. Sometimes, developers take the easier way of implementation by copying some fragments of the existing programs and use the codes in their work. This kind of work is known as code cloning. Somehow the attitude of cloning can lead to the other issues of software development, for example plagiarism and software copyright infringement (Roy and Cordy, 2007). In most cases, in order to figure out the issues and help better software maintenance, we need to detect the codes that have been cloned (Baker, 1995). In the web applications development, the chances of cloning are bigger since there are too many open source software available on the Internet (Bailey and Burd, 2005). The applications are sometimes just a 'cosmetic' of another existing system. There are quite a number of researches in software code cloning detection, but not so particularly in the area of web based applications.

## 2. Background of the problem

Software maintenance has been widely accepted as the most costly phase of a software lifecycle, with figures as high as 80% of the total development cost being reported (Baker, 1995). As cloning is one of the contributors towards this cost, the software clone detection and resolution has got a considerable attention from the software engineering research community and many clone detection tools and techniques have been developed (Baker, 1995).

However, when it comes to commercialization of the software codes, most of the software house developers tend to claim that their works are 100% done in-house without using other codes copied from various sources. This has caused a difficulty to the intellectual property copyright entities such as SIRIM and patent searching offices in finding the genuine software source codes developed by the in-house companies. There is a need to identify the software source submitted for patent copyright application as a genuine source code without having any copyright infringements. Besides, cloning somehow brings up the issue

of plagiarism. The simplest example can be seen in the academic field where students tend to copy their friends' works and submit the assignments with only slight modifications.

Generally, in software development process, there is a need for components reusability either in designing and coding. Reuse in object-oriented systems is made possible through different mechanisms such as inheritance, shared libraries, object composition, and so on. Still, programmers often need to reuse components which have not been designed for reuse. This may happen during the initial stage of systems development and also when the software systems go through the expansion phase and new requirements have to be satisfied. In these situations, the programmers usually follow the low cost copy-and-paste technique, instead of the costly redesigning-the-system approach, hence causing clones. This type of code cloning is the most basic and widely used approach towards software reuse. Several studies suggest that as much as 20% - 30% of large software systems consist of cloned codes (Krinke, 2001). The problem with code cloning is that errors in the original must be fixed in every copy. Other kinds of maintenance changes, for instance, extensions or adaptations, must be applied multiple times, too. Yet, it is usually not documented from where the codes were copied. In such cases, one needs to detect them. For large systems, detection is feasible only by automatic techniques. Consequently, several techniques have been proposed to detect clones automatically (Bellon et al., 2007).

There are quite a number of works that detect the similarities by representing the code in a tree or graph representation and also by using string-based detection, and semantic-based detection. Almost all of the clone detection techniques have the tendency of detecting syntactic similarities while only some detect the semantic part of the clones. Baxter in his work (Baxter et al., 1998) proposed a technique to extract cloned pairs of statements, declarations, or sequences of them from C source files. The tool parses source code to build an abstract syntax tree (AST) and compares its subtrees by characterization metrics (hash functions). The parser needs a "full-fledged" syntax analysis for C to build AST. Baxter's tool expands C macros (define, include, etc) to compare code portions written with macros. Its computation complexity is  $O(n)$ , where  $n$  is the number of the subtree of the source files. The hash function enables one to do parameterized matching, to detect gapped clones, and to identify clones of code portions in which some statements are reordered. In AST approaches, it is able to transform the source tree into a regular form as we do in the transformation rules. However, the AST based transformation is generally expensive since it requires full syntax analysis and transformation.

Another work (Jiang et al, 2007) presented an efficient algorithm for identifying similar subtrees and applied it to tree representations of source code. Their algorithm is based on a novel characterization of subtrees with numerical vectors in the Euclidean space  $R^n$  and an efficient algorithm to cluster these vectors with respect to the Euclidean distance metric. Subtrees with vectors in one cluster are considered similar. They have implemented the tree similarity algorithm as a clone detection tool called DECKARD and evaluated it on large code bases written in C and Java including the Linux kernel and JDK. The experiments show that DECKARD is both scalable and accurate. It is also language independent, applicable to any language with a formally specified grammar.

Krinke (Krinke, 2001) presented an approach to identify similar code in programs based on finding similar subgraphs in attributed directed graphs. This approach is used on program dependence graphs and therefore considers not only the syntactic structure of programs but

also the data flow within (as an abstraction of the semantics). As a result, it is said that no trade-off between precision and recall - the approach is very good in both.

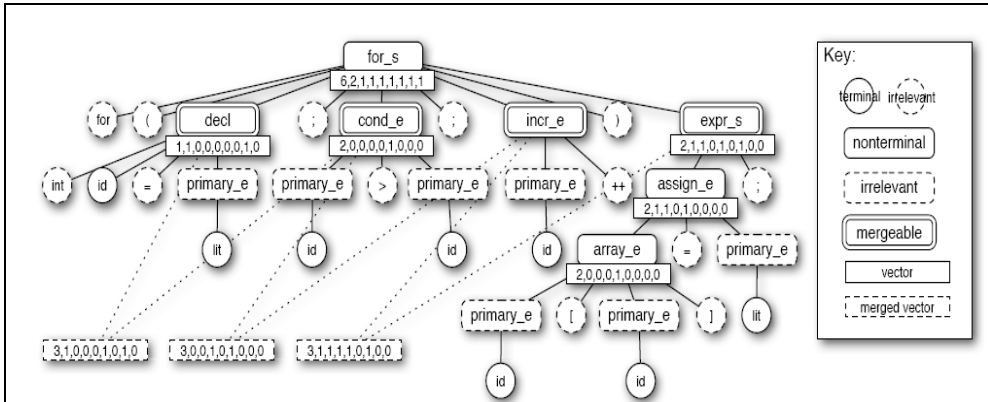


Fig. 1. A sample parse tree with generated characteristic vectors[14]

Kamiya in one of his works (Kamiya et al., 2002) suggested the use of suffix tree. In the paper they have used a suffix tree matching algorithm to compute token-by-token matching, in which the clone location information is represented as a tree with sharing nodes for leading identical subsequences and the clone detection is performed by searching the leading nodes on the tree. Their token-by-token matching is more expensive than line-by-line matching in terms of computational complexity since a single line is usually composed of several tokens. They proposed several optimization techniques specially designed for the token-by-token matching algorithm, which enable the algorithm to be practically useful for large software.

### 3. Problem Statement

As we can see from the previous works, some of the works are scalable, and are able to detect more than one type of clone. But some of them face the trade-off of the computational complexity. This could be due to the fact that most of the techniques apply expensive syntax analysis for transformation. From the literature that has been done, more than half of existing techniques used tree-based detection as it was more scalable. However, most of the techniques perform a single layer detection which means after the transformation into normalized data e.g. tree, graph, and etc, the process of finding the similarities of codes, i.e. code clones, were done directly by processing each node in the data. All possible clones need to be searched directly without some kind of filtering, which can increase the cost of computational process.

As ontology is being widely used nowadays, we cannot deny its importance in the current web technology. The major similarity of ontology and clone detection works is that both can be represented as tree. Besides that, many works have been done to do the mapping of different ontologies between each other, to find the similarities of the concepts among them. This activity is actually almost the same with what needs to be done in detecting clone codes.

Since there are some kinds of similarities in both problems, detecting clones in a source code can be done using the same way as mapping the ontologies. The research question of this thesis is to identify the possibility of using a technique of ontology mapping to detect clones in a web-based application. Obviously there will be no ontologies used in the experiments since we are dealing with source codes and not ontology. But we will use the technique of mapping to detect clones.

In order to achieve the objective, there are a few questions that need to be addressed:

- (a) What are the attributes or criteria that might be possible to be cloned in web documents?
- (b) What are the approaches that had been proposed in the previous research in the ontology mapping area than had been used in clone detection tool?
- (c) What are the issues of the recovered approach and how to solve it?

## 4. Literature Review

### 4.1 Code Cloning

Code duplication or copying a code fragment for reuse by pasting with or without any modifications is known as code smell in software maintenance. This type of reuse approach of existing code is called code cloning and the pasted code fragment (with or without modifications) is called a clone of the original. Several studies show that duplicated code is basically the result of copying existing code fragments and using them by pasting with or without minor modifications. People always believe that the major cause of cloning is by the act of copying and pasting. Some say that it may happen accidentally. In some cases, a new developed system is actually a 'cosmetic' of another existing system. This type of case usually happens in the web based application. They tend to modify the appearance of the application or system by changing the background colour, images, etc.

Refactoring of the duplicated code is another prime issue in software maintenance although several studies claimed that refactoring of certain clones is not desirable and there is a risk of removing them. However, it is also widely agreed that clones should at least be detected. Several studies have shown that, the cost of maintenance is promisingly increasing wherever there are clones in the source code compared with the codes without any clones. Definition of code cloning had been mentioned in different researches and some of them used different terminologies to refer to the code cloning.

According to Koschke (Koschke, 2006), a clone is one that appears to be a copy of an original form. It is synonymous to 'duplicate'. Often in literature, there was a misconception of code clone and redundant code. Even though code clone usually leads to code redundancy, but not every redundant code is harmful, on the other hand cloned codes are usually harmful especially for the maintenance phase of software development life cycle. Baxter in his outstanding work (Baxter et al., 2008), stated that a clone is a program fragment that is identical to another fragment", Krinke (Krinke, 2001) used the term "similar code", Ducasse (Ducasse et al. 1999) used the term "duplicated code", Komondoor and Horwitz (Komondoor and Horwitz, 2001) also used the term "duplicated code" and used "clone" as an instance of duplicated code. Mayrand and Leblanc (Mayrand and Leblanc, 1996) used metrics to find "an exact copy or a mutant of another function in the system".

All these definitions of clones carry some kind of vagueness (e.g., "similar" and "identical") and this imperfect definition of clones makes the clone detection process much harder than the detection approach itself. Generally, it can be said that code clone pair is a fragment of

code that is syntactically or semantically identical or similar. From all arguments above, we could simplify the clones into four types:

- (a) An exact copy without modifications (except for white spaces and comments) i.e. identical.
- (b) A syntactically identical copy; only variable, type or function identifiers have been changed. i.e. nearly identical.
- (c) A copy with further modifications; statements have been changed, added, or removed i.e. similar.
- (d) A code portion that is partly similar to another code fragment. It may involve some deletion, modification and addition from the original code i.e. gapped clone.

According to our understanding from Ueda (Ueda et al., 2002), we may group the second and the third types as a single major type called renamed code clone. Renamed code clone still has similar structures between each other. So it is part of this report that the framework proposed should at least be capable to detect the identical clone and renamed code clone.

Figure 2 shows an example of cloned code. Obviously the code in the example has the same code structure and the pair is considered similar.

```
1 int sum = 0 ;
2
3 void foo (Iterator iter){
4     for(item = first (iter); has_more(iter); item = next(iter) ){
5         sum = sum + value (item);
6     }
7 }

1 int bar ( Iterator iter ){
2 int sum = 0 ;
3     for(item = first (iter); has_more(iter); item = next(iter)) {
4         sum = sum + value (item);
5     }
6 }
```

Fig. 2. Example of a pair of cloned code in traditional program

## 4.2 Code Cloning in web applications

In this era, the computer has been a powerful tool to solve various kinds of problems in our everyday lives. The WWW has been the pit stop for people to find, share and exchange information all around the world. Today's web sites are not only a collections of static web sites that only display information but they also offer a lot more tasks and functions in more critical domains such as e-business, e-finance, e-government, e-learning, and so on that apply dynamic web pages with richer contents that are being retrieved from databases and such. These types of web applications require a lot more work efforts in their development life-cycles and thus require a lot more investments. People need to realize that web applications are not only meant for the Internet but if we can have at least a local area

network (LAN), we can still have web application and people within the network can still access the system.

Normally, web applications development needs shorter time of development processes and fuzzy initial requirement, thus brings to a lot of latent changes over the development life cycle (Jarzabek and Rajapakse, 2005). Since there is a need of shorter development time, there is a possibility of an increase in code cloning activities. Programmers are often forced to copy the code from existing works so that they can shorten the time to develop and make their jobs easier.

We can see there are quite a number of researches that have been carried out in the area of code cloning especially in traditional software (e.g. developed for stand alone application, using C, C++, etc) in the past decade. However, we can say that such researches are still in their infancy states for web-based application. The statement is due to the small number of researches that can be found available. Most of the researches revolve in the code clone detection whereby different strategies of detection are used. Callefato(2004) conducted an experiment of semi-automated approach of function cloned detection. Lucca et al. (2002) introduced the detection approach based on similarity metrics, to detect duplicated pages in Web sites and applications, implemented with HTML language and ASP technology. Meanwhile, Lanubile and Mallardo(2003) introduced a simple semi-automated approach that can be used to identify cloned functions within scripting code of web applications.

Some of the researches adopted the approaches that have been done in traditional software. The most frequently appeared in the researches is the use of CCFinder (Kamiya et al., 2002) as the clone detector which can detect exact clones and parameterized clones. While most of the researches are discussing about the strategies of clone detection, Jarzabek and Rajapakse(2005) conducted a systematic research to find out the extent of cloning in web domain in comparing with traditional software. They found out that cloning in web application has significantly exceeded the rate for traditional software. Jarzabek also introduced metrics that might be useful for similar studies.

### 4.3 Existing Work of Code Cloning Detection

Code cloning detection has been an active research for almost two decades. Within that period many tools and techniques have been invented either for commercial use or for academic purposes. At the same time, a number of issues have been raised along the researches in terms of number of clones detected, types of detected clones, the recall and precision, the scalability, and the coupling towards language i.e. language dependent/independent. Various researches have shown that their tools can detect almost up to 70% of clones within a particular source code.

According to Jiang (Jiang et al, 2007) the researches in this area can be classified into four main bases; string-based, token-based, tree-based and semantic-based. According to this classification, we found out that most of the clones that were detected could involve in two general ways which are syntactically and semantically. The first type of clones is usually found from the similarity of functions including scripting e.g. JavaScript, VBScript, the classes, the attributes, etc. On the other hand the semantic clones is related to the meaning of the content i.e. knowledge represented, sequence of declaration of statement (Baxter et al, 1998), etc.

Figure 3 shows the relationship of classification of detection and the type of clones detected. Most of the reports show that most of them tend to find clones syntactically rather than semantically. Syntactic clone detections cover from string-based to tree-based works. Some tree-based works also show the ability of finding clone semantically. Appendix B of this report presents some of previous works in clone detection area.

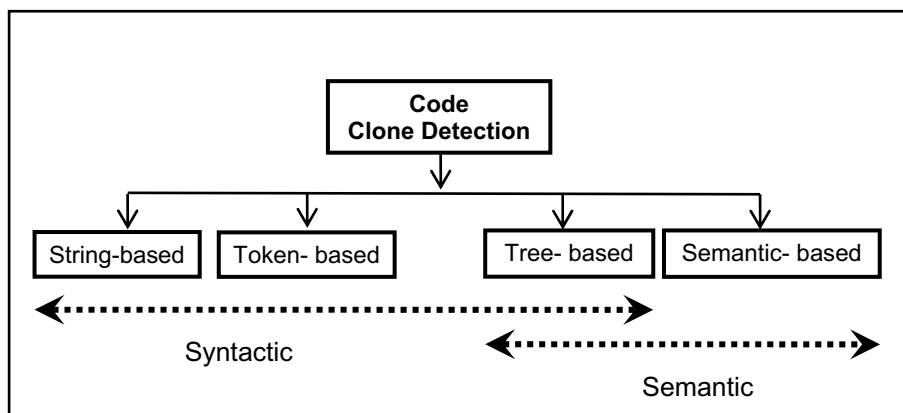


Fig. 3. Variations of clone detection research and the classification of detection

#### 4.4 Semantic Web

It is obvious that the term ‘ontology’ has become a key word within the modern computer science world. It is becoming more important in fields such as knowledge management, information integration, cooperative information systems, information retrieval, and electronic commerce. One application area which has recently seen an explosion of interest is the so-called Semantic Web.

The Semantic Web is an evolving extension of the World Wide Web(WWW) in which the web content can be expressed not only in natural languages, but also in a format that can be read and used by automated tools, thus permitting people and machines to find, share and integrate information more easily. It was derived from W3C director Tim Berners-Lee's vision of the Web as a universal medium for data, information, and knowledge exchange.

In building one layer of the Semantic Web on top of another, there are some principles that should be followed; downward compatibility and upward partial understanding (Antoniou et al., 2003).

At the bottom we find XML, a language that lets one write structured Web documents with a user-defined vocabulary. XML is particularly suitable for sending documents across the Web. RDF is a basic data model, like the entity-relationship model, for writing simple statements about Web objects (resources). The RDF data model does not rely on XML, but RDF has an XML-based syntax. Therefore in Figure 4 it is located on top of the XML layer. RDF Schema provides modelling primitives for organizing Web objects into hierarchies. Key primitives are classes and properties, subclass and subproperty relationships, and domain and range restrictions. RDF Schema (RDF-S) is based on RDF. RDF Schema can be viewed as a primitive language for writing ontologies. But there is a need for more powerful ontology



languages that expand RDF Schema and allow the representations of more complex relationships between Web objects.

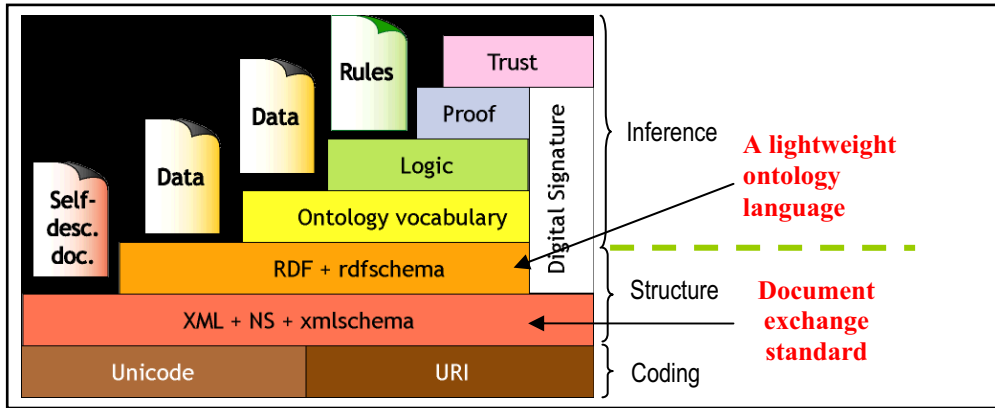


Fig. 4. Architecture of Semantic Web

The logic layer is used to enhance the ontology language further, and to allow writing application-specific declarative knowledge. The proof layer involves the actual deductive process, as well as the representation of proofs in Web languages (from lower levels) and proof validation. Finally trust will emerge through the use of digital signatures, and other kinds of knowledge, based on recommendations by agents we trust, or rating and certification agencies and consumer bodies. The Web will only achieve its full potential when users have trust in its operations (security) and the quality of information provided.

#### 4.5 Clone Detection Evaluation

As we see in the previous researches, there are many clone detection techniques and their corresponding tools, and therefore, a comparison of these techniques/tools is worthy in order to pick the right technique for a particular purpose of interest. There are several parameters with which the tools can be compared. These parameters are also known as clone detection challenges. In the following we list some of the parameters we use for comparing the different tools/techniques:

- (a) *Portability*: The tool should be portable in terms of multiple languages and dialects. Having thousands of programming languages in use with several dialects for many of them, a clone detection tool is expected to be portable and easily configured for different types of languages and dialects tackling the syntactic variations of those languages
- (b) *Precision*: The tool should be sound enough so that it can detect less number of false positives i.e., the tool should find duplicated code with higher precision. Formula shown in (1)
- (c) *Recall*: The tool should be capable of finding most (or even all) of the clones in a system of interest. Often, duplicated fragments are not textually similar. Although editing activities on the copied fragments may disguise the similarity with the original, a cloning relationship may exist between them. A good clone detection tool will be robust enough in identifying



such hidden cloning relationship so that it can detect most or even all the clones of the subject system. Formula shown in (2)

(d) *Scalability*: The tool should be capable of finding clones from large code bases as duplication is the most problematic in large and complex system. The tool should handle large and complex systems with efficient use of memory. In this thesis it can be concluded by analyzing computational time taken for different sizes of testing

(e) *Robustness*: A good tool should be robust in terms of the different editing activities that might be applied on the copied fragment so that it can detect different types of clones with higher precision and recall. In this thesis we apply the robustness by listing the type of clones the respective clone detector finds and their frequencies.

$$\text{Precision, } p = \frac{\text{number of correct found clone}}{\text{number of all found clone}} \quad (1)$$

$$\text{Recall, } r = \frac{\text{number of correct found clone}}{\text{number of possible existing clone}} \quad (2)$$

## 5. Proposed technique of code clone detection

We will start by defining the relation assumed by our model between ontologies and source code, on the one hand, and source code and instances on the other hand. A set of documents can serve as a base to extract ontological information (Stumme., and Maedche, 2001). In this model we represent the source codes using XML parse tree. Hence we assume that the ontological information in this case are all the tagging name in XML trees, i.e. known as concept in this thesis as stated in the following formal definition. The instances are all similar concepts that are actually populated in the source code. An instance will consist of the concept itself and the attributes and value of that concept.

In ontology mapping, given two schemas, A and B, one wants to find mapping  $\mu$  from the concepts in A into the concepts of B in such a way that, if  $a = \mu(b)$ , then b and a have the same meaning. This clone detection basically uses the same concept as in ontology mapping work.

**Definition 1:** if  $a = \mu(b)$ , then b and a have the same meaning, hence derived code clones. Our strategy is to do a one-to-one mapping since using specific shared ontologies might request for a specific domain of knowledge for different applications that need to be compared. The idea is to derive mappings from candidate concept A to the concepts A' with the same names as in the selected ontologies.

**Definition 2:** Ontology,  $O' = \{C', R', CH', rel', OA'\}$  where (a) C is the set of concepts in each of the nodes in the tree, (b)  $CH' \subseteq C' \times C'$  is concept hierarchy or taxonomy, where  $CH'(C'_1, C'_2)$  indicates that  $C'_1$  is a subconcept of  $C'_2$ , (c)  $rel: R' \rightarrow C' \times C'$  is a function that relates the concepts non-taxonomically,  $R'$  is the set of relations where  $R' = \emptyset$ , (d) OA is a set of ontology axioms, where OA' is the properties of concepts, in practical the contents of tags, the attribute and the value.

Figure 5 shows the overall phases of clone detection. The key idea of the proposed technique is by combining detection by the structural information and the instance-based detection as both of the techniques have their own strengths and weaknesses (Todorov, 2008) and has

been discussed in the previous chapter. The output of the processes will be a set of similar fragments of code, i.e. under different types of clones between two different systems. In our framework we assume that the population phase had already taken place and there exists a set of source codes so that:

- (a) It covers all concepts of the source code trees. "Covers" is understood as: Instances of every concept can be found in at least one of the trees in the collection of source code trees. Every tree contains instances of at least one concept,
- (b) A tree node is considered to be assigned to a concept node if and only if it provides instances of that concept with a higher cardinality than a fixed threshold (Fig. 6).

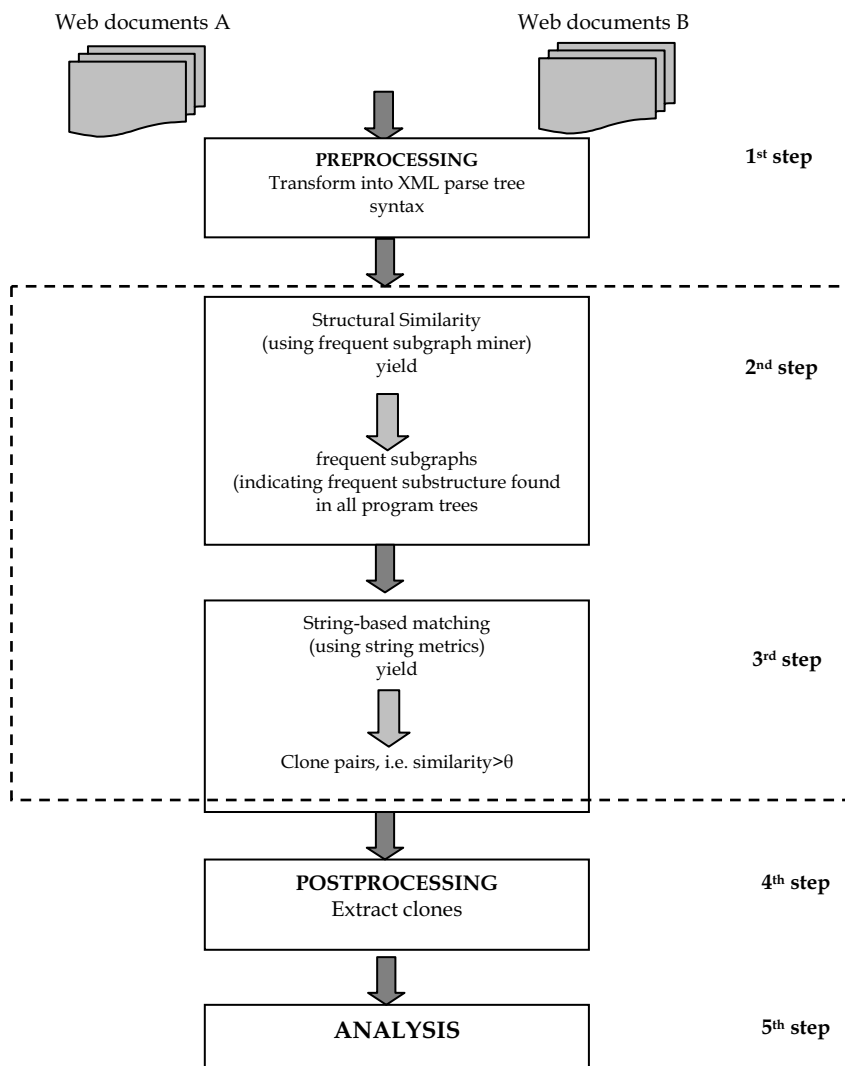


Fig. 5. Diagrammatic view of clone detection technique

In the sequel we will deal with hierarchical trees. We are concerned with studying their similarities on purely structural level, so let us assume that only the concept nodes are labeled but not for the relations. Under these assumptions we provide the following definition of a hierarchical source code tree.

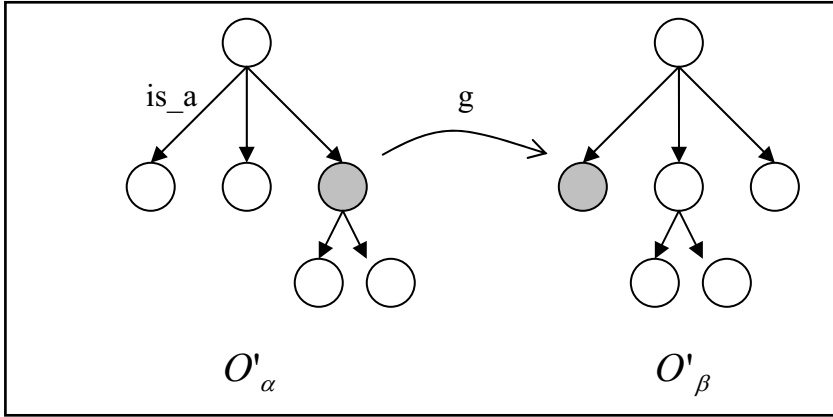


Fig. 6. Title Mapping between concepts of  $O'_\alpha$  and  $O'_\beta$

**Definition 3:** A hierarchical tree is a couple  $O' := (C, is\_a)$ , where  $C$  is a finite set whose elements are called concepts and  $is\_a$  is a partial order on  $C$ . We proceed to formalize the association of tree of different source codes. Let  $\alpha$  be a set of hierarchical source code trees of system 1 and  $\beta$  a set of hierarchical source trees of system 2 satisfying the assumption 1 and 2. Let  $\gamma: \alpha \rightarrow \beta$  be an injection from the set of source code trees of system 1 to the set of source code trees of system 2. For every subtree of  $O'_\alpha \in \alpha$  and subtree of  $O'_\beta \in \beta$  that can be mapped so that there exists  $\gamma(O'_\alpha) = O'_\beta$ , it exists an injection  $g: C_{O'_\alpha} \rightarrow C_{O'_\beta}$  which map concepts in source code tree of system 1 to system 2. The following figure shows an illustration of the mapping between trees.

### 5.1 Structural Tree Similarity

As mentioned in the previous chapter, we are going to do two layers of tree comparing. The first layer is about the structural tree similarity. It eventually provides some kind of filtering to the model since it finds parts of the trees which is similar between each other before we do the real similarity comparison. As has been discussed in the literature review, XML tree is actually a directed rooted tree which can be represented formally using the definition of graph  $G = (V, E)$ . So a source code tree can formally be represented by the following definition:

**Definition 4:** Let  $O'_X$  be a source code tree. A source code tree corresponding to 0 is a directed rooted tree  $G(V(G), E(G))$ , so that (a)  $V(G) = C$ , (b)  $E(G) \subseteq C \times C$  such that  $\exists f$  where  $\langle f(c_i, c_j) \rangle \in E_{O'_X} \Leftrightarrow \langle c_i, c_j \rangle \in is\_a$ .

Todorov (Todorov, 2008), used Bunke's graph distance metric to calculate the distance of source code structure based on maximal common subgraph. We are not going to find the maximal subgraph since this technique is often an NP-complete problem and it has been used several times in the previous works of clone detection. So instead of using maximal common subgraph, we used the frequent subgraph miner available. Before that we start by giving a couple of definitions which are needed before introducing the distance ratio. The distance ratio is used to find out number of programs that could have high similarities of structures between each other. All definitions are given for general graphs and are also applicable for trees.

**Definition 5:** Graph Isomorphism. A bijective function  $\mu: V_1 \rightarrow V_2$  is a graph isomorphism from graph  $G_1(V_1, E_1)$  to a graph  $G_2(V_2, E_2)$  if for any  $v_1^1, v_2^1 \in V_1$   $\langle v_1^1, v_2^1 \rangle \in E_1 \Leftrightarrow \langle \mu(v_1^1), \mu(v_2^1) \rangle \in E_2$ .

**Definition 6:** Subgraph isomorphism. An injective function  $\mu: V_1 \rightarrow V_2$  is a subgraph isomorphism from  $G_1$  to  $G_2$  if it exist a subgraph  $S \subseteq G_2$  so that  $\mu$  is a graph isomorphism from  $G_1$  to  $S$ .

**Definition 7:** Graph distance ratio: We simplify the distance of graph  $G_1$  and  $G_2$  by using the following ratio since we are using frequent subgraph mining. Let  $F_{G_1}$  (3) and  $F_{G_2}$  (4) as sets of frequent subgraphs which owned by  $G_1$  and  $G_2$ .

$$F_{G_1} = \{ t_1, t_2, t_3, \dots, t_m \} \quad (3)$$

$$F_{G_2} = \{ t_1, t_2, t_3, \dots, t_n \} \quad (4)$$

Distance of  $G_1$  and  $G_2$  (5) can be calculated as follows:

$$d(G_1, G_2) = 1 - \frac{\#(t_{G_1} \cap t_{G_2})}{\max(\#t_{G_1}, \#t_{G_2})} \quad (5)$$

## 5.2 Preprocessing

The initial idea is by doing the detection with combination of tree detection and string detection. For this reason, the clone detection will start with the pre-processing where all documents will be standardized into XML documents in order to get the tags and contents of each node. We are going to test the model on HTML, ASP, PHP and JSP systems. Web page documents from system A and system B need to be compared to detect the cloning. Then the XML will be parsed into a tree. Fig. 7 shows the main process of pre-processing.

To minimize the code for each file, all XML codes will be cleaned to eliminate all useless lines of code so that we could maximize the code comparing without trying to compare the formatting information which is only used for the purpose of information appearance to the end user. For each and every XML source code, the tag names will be taken and inserted into a file called 'vocabulary' that will be used for XML node matching. Duplicate entries in the vocabulary will then be deleted from the list to minimize number of entries in the vocabulary.

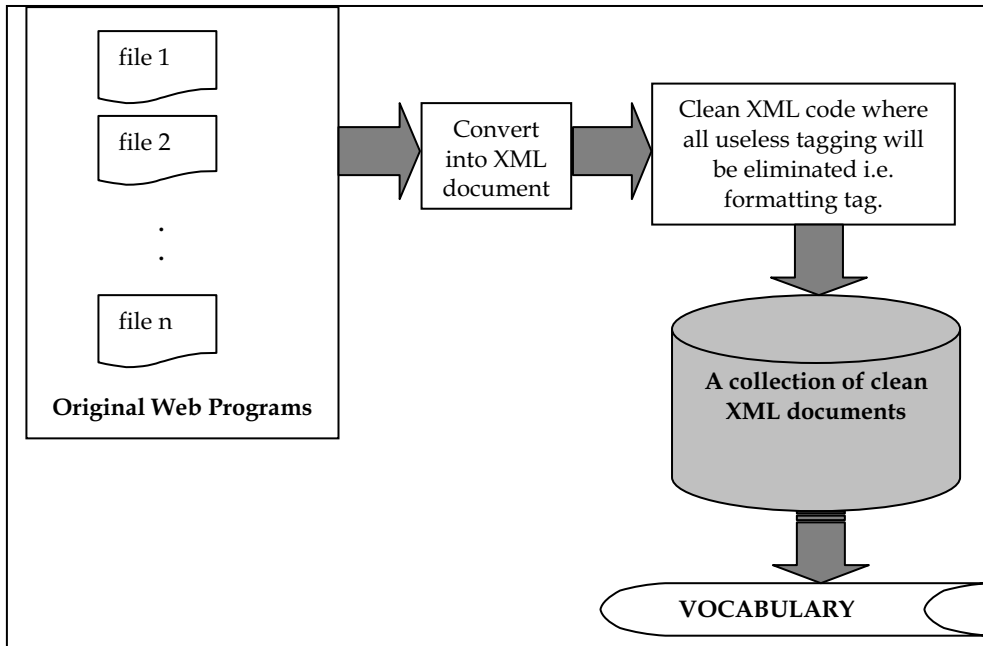


Fig. 7. Preprocessing phase

### 5.3 Frequent subgraph mining

The detection process will then start with the structural comparison of the tree. The comparison of nodes is done between  $O'_\alpha$  and  $O'_\beta$  which represent two different systems.

After generating the frequent subgraphs, we store the shared subtree of different programs or source codes in a cross table. Table 1 shows the example of cross-table used to compare programs across two systems.

Program, $p$	$p_1$	$p_1$	...	...	...	...	$p_n$
$p_1$	$t_1, t_2$	-	-	$t_1, t_3, t_4$	-	-	-
$p_1$	-	$t_1, t_2$	-	-	-	-	-
...	-	-	-	-	-	-	-
...	-	-	$t_1, t_3, t_4$	-	$t_1, t_3, t_4$	-	-
$p_m$	-	-	-	-	-	-	$t_1, t_3, t_5$

Table 1. Example of cross table used to compare programs across two systems

For each subtree in the table that was generated by the frequent subgraph miner, we set the minimum size i.e. number of edges of subtree is 5 and the maximum is 6. The decision is made after an initial experiment where it showed the number of frequent subtree generated was not too large or too small in comparison with other value of parameter. Then the string-

based matching will be done as described above. The example of a frequent subgraph between two trees is shown in Fig. 8.

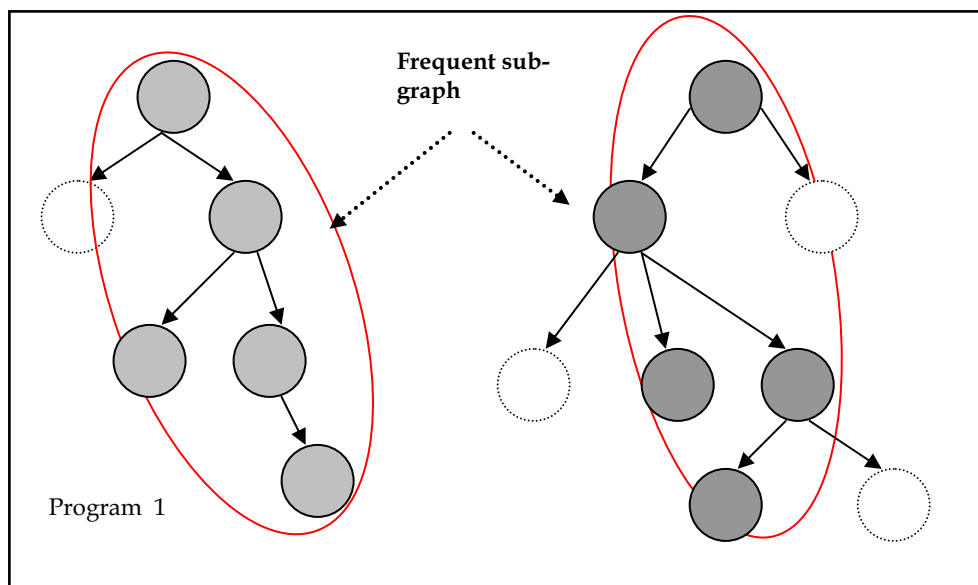


Fig. 8. Illustration of frequent subgraph of two trees

#### 5.4 String based matching

In Fig. 9, an example of instance-based matching is presented. We found that depth of fragment 1 and fragment 2 are equal to three, so the comparison using string metric was done but in the original experiment we used five and six instead of depth equal to three. If the similarity is above the threshold, the string will be taken as a clone pair. Instead of using vocabulary as in our initial experiment, we compare the similar structure of subgraph found and it is recorded in the table above.

As mentioned before, we assumed that all elements of the subtree were treated as an instance i.e. including the node name, attribute and value, etc. We took all the elements as a string for simplicity in order to calculate the similarities of the set of instance in fragment 1 i.e. set A and the set of instance of fragment 2 i.e. set B. The last stage of the code clone detection would be the post-processing. At this stage, all clones will be extracted from the original code for further analysis.

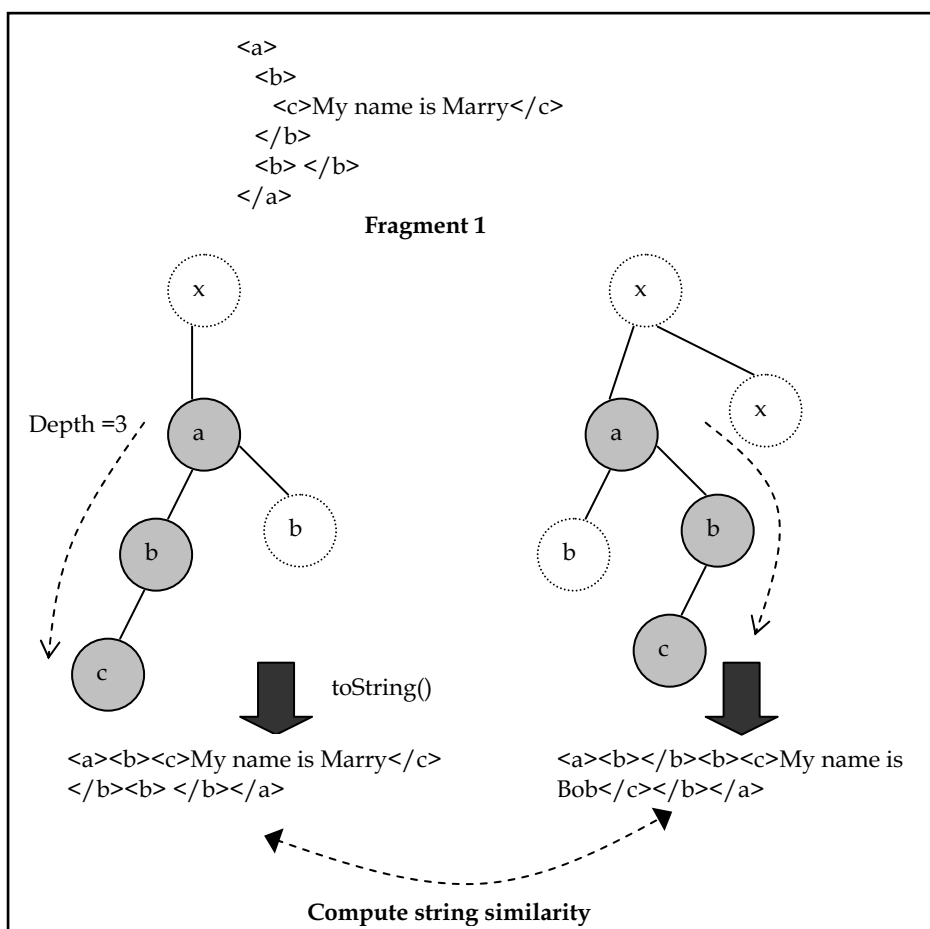


Fig. 9. A pair of source code fragment classified as nearly identical

### 5.5 Clone detection algorithm

The previous subtopics explained the process of the proposed clone detection. The process can be summarized into general algorithm in Fig. 10:

Variable: *threshold, p, minNode.*

begin

**Step 1:** Define parameter *threshold, p, minNode.*

**Step 2:** Convert all files of system A and system B into XML and clean files

**Step 3:** Generate frequent subgraphs and record in cross table, D.

**Step 4:** For all subgraphs in D,  
begin-while



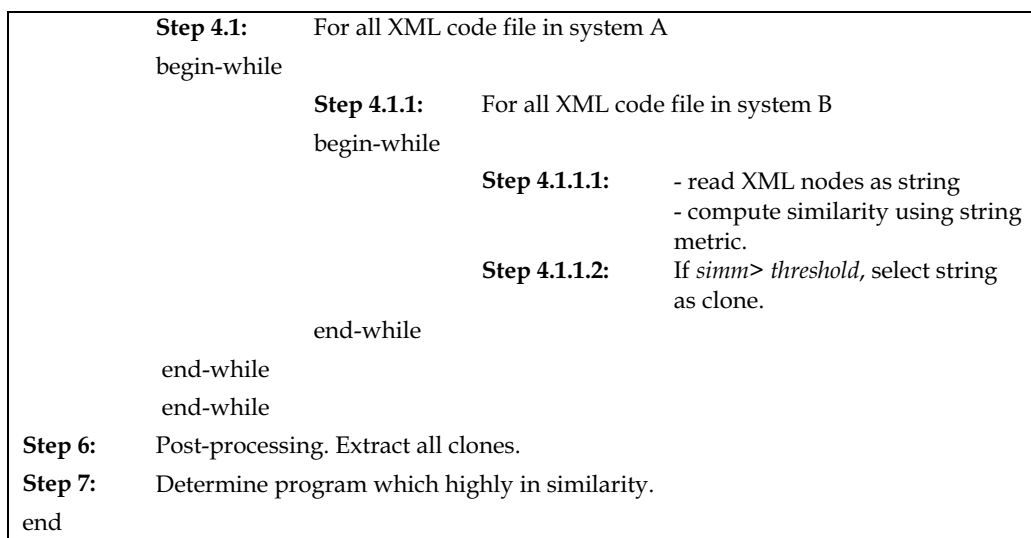


Fig. 10. Clone Detection Algorithm

## 6. Experimental Result and Discussions

This chapter primarily presents the results obtained by searching for clone pairs using a methodology inspired from ontology mapping works. As mentioned in the previous chapter, in the ontology mapping work, the author used maximal common subgraph in the first layer and the calculating of the instances similarities using Jaccard coefficient. So in our methodology, instead of using maximal common subgraph, we are using frequent common subgraph miner as the maximal common subgraph technique is frequently reported as NP-complete problem.

Generally, methodology consists of four main stages, i.e. preprocessing stage, frequent subtree generation using frequent subgraph miner, subtree similarity computation, and extraction of clone pairs and analysis. The frequent subtree mining in this work is taken as the process of getting candidate cloned pairs which have similar subtree structure by only taking into account the node tags and omitting any other elements of the tree such as attributes, labels or values of the tree.

Before we discuss in depth about the result of code clone detection, we will discuss the experiment that has been carried out in this project. The following first two subchapters will discuss the pre-processing stage. In this stage, we do the preparation where the original source code is transformed into inexpensive standardized form and a representation of web source programs in order to induce the data into the frequent subgraph miner.

### 6.1 Data Representation

A few group of system files were used for testing purposes. We divided the data into three different sizes of data; i.e. small data size, medium size, and large size where all the programs were taken from open sources web applications. The original web program was in HTML, ASP and PHP format to test the portability of our system where our system is

considered portable if it managed to process different types of web programming languages.

As mentioned before, all programs need to be transformed into a standard form of program. In our system, we transformed the programs into XML format where the transformations were inexpensive. This was because we needed to make sure that the entire program was in a valid form of tagging so that we could extract all the tagging names of each XML tree.

```
<?php
$conn=mysql_connect("localhost","","");
$db= mysql_select_db("inventory");

?>
```

*(a) Original PHP code*

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"
"http://www.w3.org/TR/REC-html40/loose.dtd">
<HTML>
<HEAD>
<!-- Converted by AscToTab 4.0 - fully registered version -->
<TITLE>Converted from "C:\Documents and Settings\DiLLa
DiLLoT\Desktop\dbase.php"</TITLE>
<META NAME="Generator" CONTENT="AscToHTM from www.jafsoft.com">
</HEAD>
<BODY BGCOLOR="white">

<!-- User-specified TABLE structure -->
<TABLE ID="table_1" BORDER=2 CELSPACING=0 CELLPADDING=2>
<TR VALIGN="TOP">
<TD>&lt;?php<BR>
$conn=mysql_connect("localhost","","");<BR>
$db= mysql_select_db("inventory");<BR>
<BR>
?&gt;<BR>
</TD>
</TR>
</TABLE>
<!-- end of user-specified TABLE structure -->

<!-- Converted by AscToTab 4.0 - fully registered version -->
</BODY>
</HTML>
```

*(b) Generated HTML code*

Fig. 11. Transformation of original PHP code into HTML code

### 6.1.1 Original Source Programs into XML Format

The first step for data normalization is by converting all programs into the HTML format. This is to make sure that all XML documents generated are in a valid form of tagging. As for now, the process is done by using a freeware tool called AscToTab which can transform any form of text into HTML or RTF format. This stage needs to be done manually. After all the

transformations are done, by using our system, the HTML programs will be converted into XML documents for further processes. For any program which is not an HTML program e.g. PHP and ASP, the programs are treated as text files. Transformation using the tool is done by applying formatting tags such as <br>, <p>, etc onto the text code. Fig. 11 shows an example of an original source code transformation into HTML. After transformation into HTML, the code is then converted into XML to ensure their validity. This process can be done automatically using our system where it provides a function to convert HTML into XML. Fig. 12 shows a result of converting HTML into XML.

```
<?xml version="1.0" encoding="iso-8859-1" ?>

<root>
  <doctype>HTML PUBLIC &quot; -//W3C//DTD HTML 4.0
Transitional//EN&quot; &quot;http://www.w3.org/TR/REC-html40/loose.dtd&quot;</doctype>
  <html>
    <head>
      <comment>Converted by AscToTab 4.0 - fully registered version</comment>
      <title>
        <text>Converted from &quot;C:\Documents and Settings\DiLLa
DiLLoT\Desktop\dbase.php&quot;</text>
      </title>
      <meta NAME="Generator" CONTENT="AscToHTM from www.jafsoft.com" />
    </head>
    <body BGCOLOR="white">
      <comment>User-specified TABLE structure</comment>
      <table ID="table_1" BORDER="2" CELLSPACING="0" CELLPADDING="2">
        <tr VALIGN="TOP">
          <td>
            <text>&lt;?php
$conn=mysql_connect(&quot;localhost&quot;,&quot;&quot;,&quot;&quot;);
$db= mysql_select_db(&quot;inventory&quot;); ?&gt;</text>
          </td></tr>
        </table>
        <comment>end of user-specified TABLE structure</comment>
        <comment>Converted by AscToTab 4.0 - fully registered version</comment>
      </body>
    </html>
  </root>
```

Fig. 12. XML form of the previous HTML code

### 6.1.2 Subtree Mining Data Representation

XML representation is not suitable to be fed directly into our frequent subgraph miner. So, as the solution we need to represent the tree structure into a simpler form of data. This is important so as to reduce the complexity of the mining process.

The simplest way is by representing the trees as a node and edge lists. Before generating the data, we extract all node names or tagging in XML code and treat them as a bag of concept or vocabulary, as had been used in Project I. The subgraph mining data is represented as a list of nodes and edges as in Figure 13 below. In the figure, t represents tree, v represents vertex and e represents edge. Label of node in figure below represents the node name or tagging of the XML. But instead of putting the node name in the list, we put the index of vocabulary as we had explained before.

```
t # <name of the graph>
v 0 <label of node 0>
v 1 <label of node 1>
...
e <node a> <node b> <edge label>
e <node x> <node y> <edge label>
...
```

Fig. 13. XML form of the previous HTML code

```
[0] text
[1] title
[2] a href
[3] p
[4] h1
[5] meta http-equiv
[6] head
```

*(a) Vocabulary/ Bag of concepts*

```
t # s1.XML_10.xml
v 0 1
v 1 6
v 2 0
v 3 3
...
e 0 1 1
e 0 2 1
e 2 3 1
...
```

*(b) A tree represented as list of vertices and edges*

Fig. 14. Example of tree as vertices and edges list

Fig. 14 shows the example of vocabulary generated and the tree representation following the format in Fig. 13. In example below, [v 0 1] means that node name for vertex0 is title and [e 0 1 1] means there is an edge between vertex0 and vertex1. The last digit 1 is the default labelling for all edges since we are working with trees instead of graphs, so we need to omit any labelling of all edges. Data in Fig. 14(b) will be fed in the frequent subgraph miner.

<pre> t # 24 v 0 11 v 1 10 v 2 3 v 3 0 v 4 12 e 0 1 1 e 1 2 1 e 1 3 1 e 3 4 1 =&gt; [2.0] [s2.XML_21.xml, s1.XML_10.xml ] t # 26 v 0 11 v 1 10 v 2 3 v 3 0 v 4 9 e 0 1 1 e 1 2 1 e 1 3 1 e 3 4 1 =&gt; [2.0] [s2.XML_21.xml, s1.XML_10.xml ] t # 28 v 0 11 v 1 10 v 2 3 v 3 0 v 4 2 e 0 1 1 e 1 2 1 e 1 3 1 e 3 4 1 =&gt; [2.0] [s2.XML_20.xml, s1.XML_11.xml ] </pre> <p>(a) Example of frequent subtrees generated</p>	<pre> GSpan subgraph miner found 76 frequent fragments &gt;&gt;SIMILAR SUB TREE CROSS-TABLE ; x= for system 1, y= for system 2  BetweenFile[0][0]:77,72,70,68,66,64, 38,36,34,32,30,  BetweenFile[0][1]:176,171,169,155,15 3,151,149,147,145,143,141,139,137,13 2,130,128,116,114,112,110,108,106,10 4,102,100,93,91,89,87,85,77,72,70,68 ,66,64,56,54,52,50,48,46,44,42,40,38 ,36,34,32,30,26,24,  BetweenFile[1][0]:192,184,164,162,16 0,120,118,95,77,72,70,68,66,64,62,60 ,58,38,36,34,32,30,28,  BetweenFile[1][1]:77,72,70,68,66,64, 38,36,34,32,30, </pre> <p>(b) Example of cross- table containing subtree id shared between different files</p>
--	---

Fig 15. Frequent subtrees generated by graph miner

## 6.2 Frequent Subtree Mining

As we mentioned before, we used four well-known frequent subgraph miners to get similar substructure of trees that existed between the files. We induced the data as in the previous example into the miner and the miner will generate all frequent subtrees or substructures

that existed among the files. There are a few configurations that need to be set before doing the mining. The configurations are:


(a) *MinimumFrequencies* sets the minimum frequency (support) a subgraph must have to get reported. In the experiment, we set the value as low as 10% so that the miner will be capable to find all similar substructures even though the appearances in the codes are not so frequent,

(b) *MinimumFragmentSize* sets the minimum size (in edges) a frequent subtree must have in order to be reported,

(c) *MaximumFragmentSize* sets the maximum size (in edges) a frequent subtree can have in order to be reported.

In the experiment, we set the value of (b) and (c) with 5 edges in size. We selected this size after some preliminary experiments where this value is capable to generate the average number of subtrees. So instead of using minimal depth of a subtree, we used the minimum and maximum fragment size. After executing the graph miner, a list of frequent subtree will be generated from the system as well as the original tree that holds that particular subtree. So to summarize, we can generate a cross-table which contains all subtrees IDs that were shared among different files in different systems.

```
String s = <root><doctype>HTML PUBLIC &quot; -//W3C//DTD HTML 4.0
Transitional//EN&quot; &quot;http://www.w3.org/TR/REC-
html40/loose.dtd&quot;</doctype><html><head><title><text>Converted
from &quot;C:\Documents and Settings\DiLLaDiLLoT\Desktop\
dbase.php&quot;</text></head></root>
```



```
<root>
  <doctype>HTML      PUBLIC      &quot;-//W3C//DTD      HTML      4.0
  Transitional//EN&quot; &quot;http://www.w3.org/TR/REC-
  html40/loose.dtd&quot; </doctype>
  <html>
    <head>
      <comment>Converted by AscToTab 4.0 - fully registered
      version</comment>
      <title>
        <text>Converted from &quot;C:\Documents and
        Settings\DiLLa DiLLoT\Desktop\dbase.php&quot;</text>
      </title>
      <meta NAME="Generator" CONTENT="AscToHTM from
      www.jafsoft.com" />
    </head>
    ...
  </root>
```

Fig. 16. Code fragment containing original frequent subtree

### 6.2.1 String Metric Computation

The most challenging part of the system is to extract the original subtree from the original XML documents according to the frequent subtree generated above. Once the original subtree is successfully extracted, it will be taken as a string so that we can compute the similarities of the subtree with another subtree from another file using a string metric. We realized that this technique is suitable in finding cloned pairs. Instead of using frequent subgraph miner, we used vocabulary element to find the subtree rooted with a node which has a similar name with the vocabulary element. As we discussed before, it was quite expensive to do it this way. So, for the similarity computation, we used all the string metric that were stated in section 4 before. In the following example, we will show how the subtree is represented as a string before we can compute the similarity. Consider if the bold italic lines match the frequent sub tree, the string equivalent would be as in Fig. 16.

### 6.3 Experimental Setup

The implementation process was done using Java language as the base language. To support the program, we used a Java library named Chilkat Java which can be found available on the Internet. This library offers a few features like XML parser and tree walker ability. All process of converting web pages into standard XML and generating the vocabulary for mapping purposes were done automatically in the program. All executions were done using an Intel® Core Duo 1.86GHz machine with 1.5 GB of RAM.

The following settings were meant for the Ontology-Winkler Similarity part. We set the most lenient values for all those parameter:

- (a) The similarity threshold,  $\theta$  is set to 0.7
- (b) Define the similarity,  $Sim$  as  $Sim(s_1, s_2) = Comm(s_1, s_2) - Diff(s_1, s_2) + winkler(s_1, s_2)$  where  $Comm(s_1, s_2)$  is the commonality value between two strings and  $Diff(s_1, s_2)$  is the difference between two strings
- (c) Omit the  $Winkler(s_1, s_2)$  calculation from the equation to simplify the programming. Value for  $Winkler$  is set to 0.1
- (d) Value of parameter  $p$  is set to 0.6 as the original author of this technique reported that the result for their experiment works best with this value.

### 6.4 Experimental Results

This subsection is mainly to present the result of our code clone detection using all the metrics that have been discussed in the previous chapter. In this experiment, we will consider any valid candidate clone as a clone even though that code fragment is in fact an accidental clone.

Several experiments were conducted to investigate the performances of our clone detection program. The experiments were executed using the same parameters setting and data setups as in the previous subchapter. We conduct the experiment using two open source management systems as (a) Module 1.1 - 54.9Mb(size), >4000 files, (b) Tutor 1.55 - 49.7Mb(size), >3000 files.

For the test, we randomly selected the files from these two systems for comparison where the test is done on a different number of files. By using 1.5Gb RAM processor, the number of files that can be processed are quite limited and it only allowed a small size of detection



which is less than 100 files. We split the testing in three groups of testings. Table 2 shows information of data being used for the testing process.

Number of testings	Number of files(NOF)	Line of codes(LOC)
Testing #1	10	259 lines
Testing #2	30	575 lines
Testing #3	50	1200 lines

Table 2. Data for program testing

As it is shown , our experiment is basically on a different subgraph miner and different string metric or similarity coefficients. The following figure shows an example of the realtime output of detection using our program which has been written in Java.

```
>>COMPARE FILES:

*Compare file between:
D:/Documents/MASTER/4thSEM/Project II/Code Clone
Detection/procl/XML_10.xml
D:/Documents/MASTER/4th SEM/Project II/Code Clone
Detection/proc2/XML_20.xml
#1:
<head>Thisisatest<title>Thisisatest<text>ThisisatestThisisatestThis
isatest<meta><meta>
<head>Thisisatest<title>Thisisatest<text>ThisisatestThisisatestThis
isatest<meta><meta>
      = 1.0

*Compare file between:
D:/Documents/MASTER/4th SEM/Project II/Code Clone
Detection/procl/XML_10.xml
D:/Documents/MASTER/4th SEM/Project II/Code Clone
Detection/proc2/XML_20.xml
#1:
<head>Thisisatest<title>Thisisatest<text>ThisisatestThisisatestThis
isatest<meta><meta>
<head>Thisisatest<title>Thisisatest<text>ThisisatestThisisatestThis
isatest<meta><meta>
```

Fig. 17. Realtime output from the clone detection system

We show the result of the experimental testing using our default subgraph miner which is GSpan miner. As mentioned before, the test is done on three different sizes of files as shown above. Table 3 shows some of the graphical output using Jaro Winkler and Levenshtein Distance as the string metric.

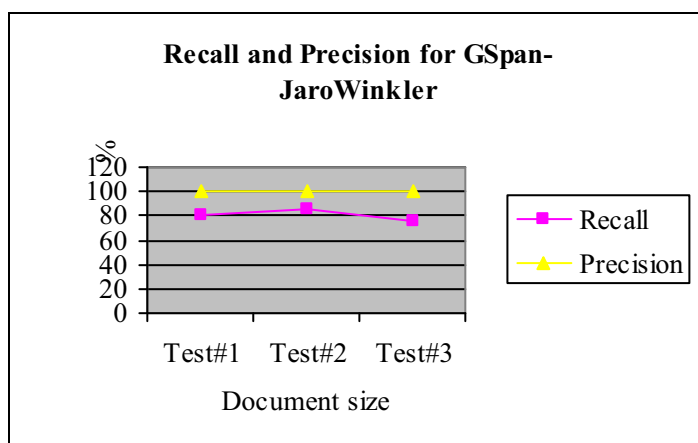


Fig. 18. Recall and precision for GSpan-Jaro Winkler

Fig. 18 to Fig. 23 show the result of using GSpan frequent subgraph miner. As we can see from the diagram, mining the similar structure using GSpan miner generated almost similar graph trends where the value generated is almost similar between these two string metrics.

As shown in the figure, all cloned pairs that were found by our code clone detection system were all positive clones. This situation yielded our precision to be 100% for small size or bigger data. But there was a trade-off for the recall. Our system only managed to find a small number of clones, where most of the clones found were identical clones, but we can say the limitation is on searching for similar clones.

Another big issue shown in the data above is the computational time taken was rapidly increasing as the number of documents increase. This is practically not feasible for detecting cloned pairs. However, we may need more testing done to find out whether the line will keep increasing towards the infinite as the number of document increased.

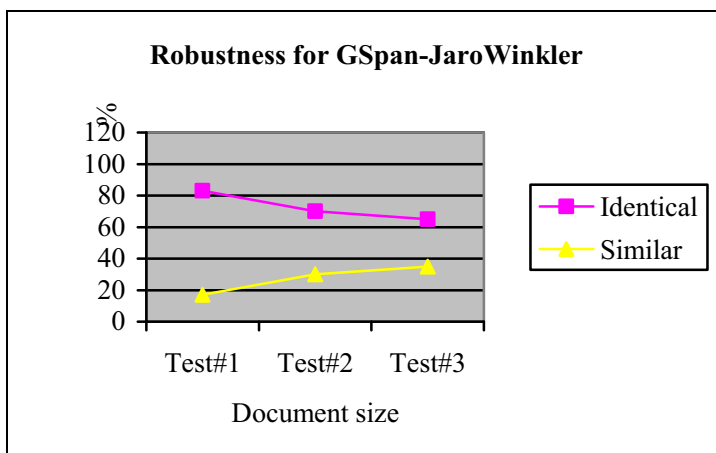


Fig. 19. Robustness of GSpan-Jaro Winkler

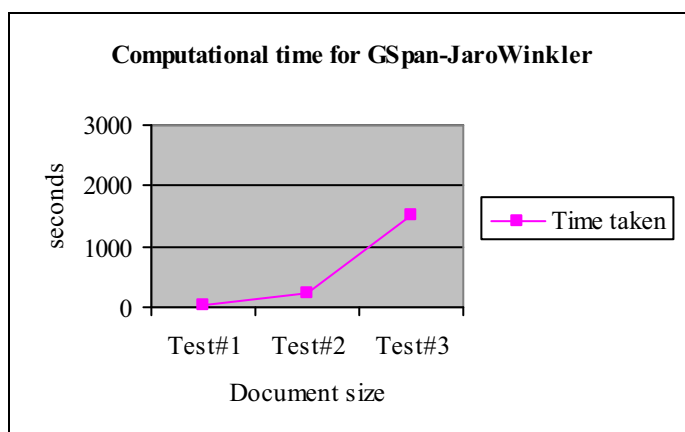


Fig. 20. Computational time for GSpan-JaroWinkler

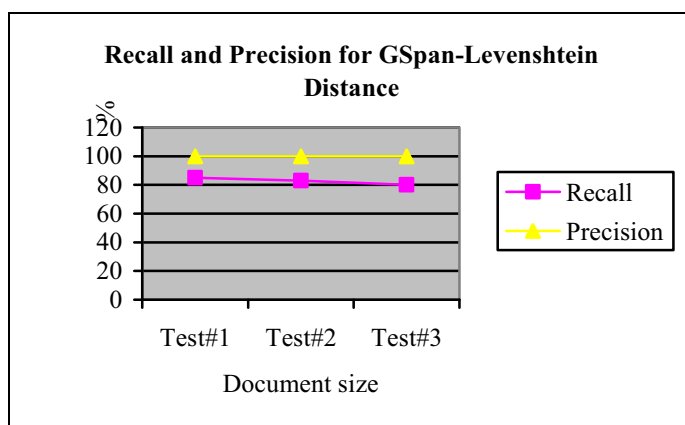


Fig. 21. Recall and Precision for GSpan-Levenshtein Distance

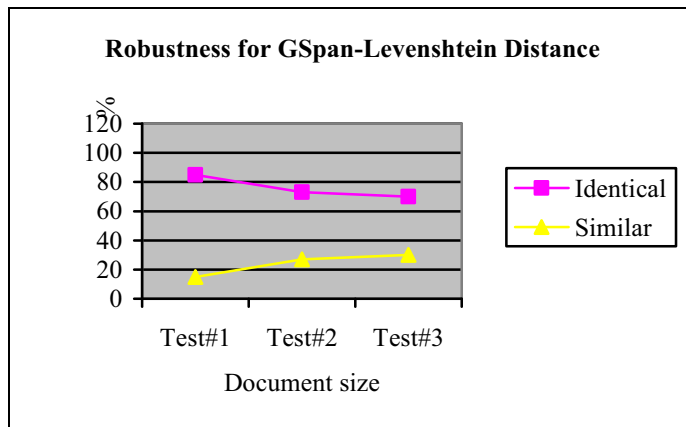


Fig. 22. Robustness for GSpan-Levenshtein Distance

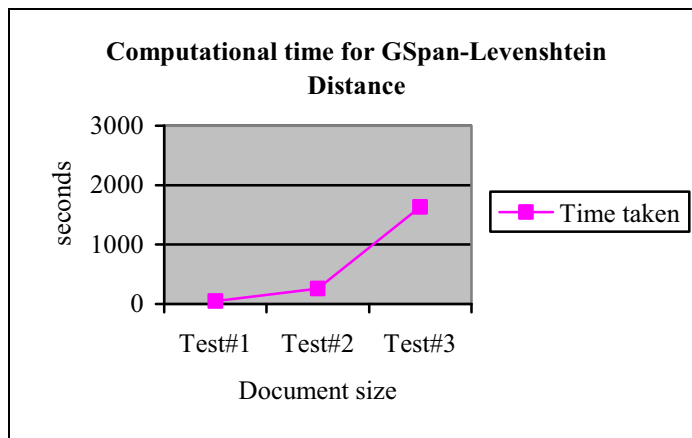


Fig. 23. Computational time for GSpan-Levenshtein Distance

Generally, there is not much difference in the trends of graphs using different frequent subgraph miner. The major difference is on the overall computational time of the detection as different frequent subgraph miner offers different performance in generating frequent subgraph. The result shows that Gaston offers the best computational time, followed by gSpan, FFSM and MoFa.

### 6.5 Limitation of the Code Clone Detection Program

As we can see, the overall result of our code clone detection program did not show the good result that we had expected. In general, we noticed that for each and every subgraph miner and string metric used, the computational time increased rapidly as the number of source codes increased. This is practically not healthy for code clone detection or for any experiments related to this area, e.g. plagiarism.

Another significant concern is the results showed a big trade-off between the recall and precision. From the precision view, the program managed to achieve very good results but not from the recall view, where only a part of all expected clones were found during the detection. For analysis purposes, we identified a few points that may affect the overall results. The points are:

- (a) The computational time taken may be affected by the pre-processing time taken to convert the original code into the XML form.
- (b) It may also be affected by processing taken by sub graph miner. The miner most probably will generate all subtrees from the code subtrees which sometimes reached thousands of subtree even for only a small number of source code being tested before it identifies which subtrees are the frequent ones.
- (c) We need a higher specification of a machine to perform the test as the current machine is only capable to test less than 100 source files per time. We have initially tested more than 100 times, but the computational time had gone to infinite.
- (d) The program is only capable to detect identical clones and near identical clones since our program is using the string-based detection. As we know the strength of string-based detection is it is able to detect more languages i.e. it is language independent, but the weakness is in terms of the robustness where it is only able to detect identical and near identical clones.
- (e) The clones found were always the same size as the particular testing since we already predefined the fragment size of frequent subtree in the frequent subgraph miner. So, there might be similarities between the clones and the differences may only be a node in a subtree. Fig. 24 shows the illustration of the scenario. Assume that the shaded part of the tree is taken as a frequent subtree by the subgraph miner and detected as a clone in a source code. It shows there are nodes in both frequent subtrees that intersect and the subtrees actually can be taken as a single clone but our program was unable to do that.

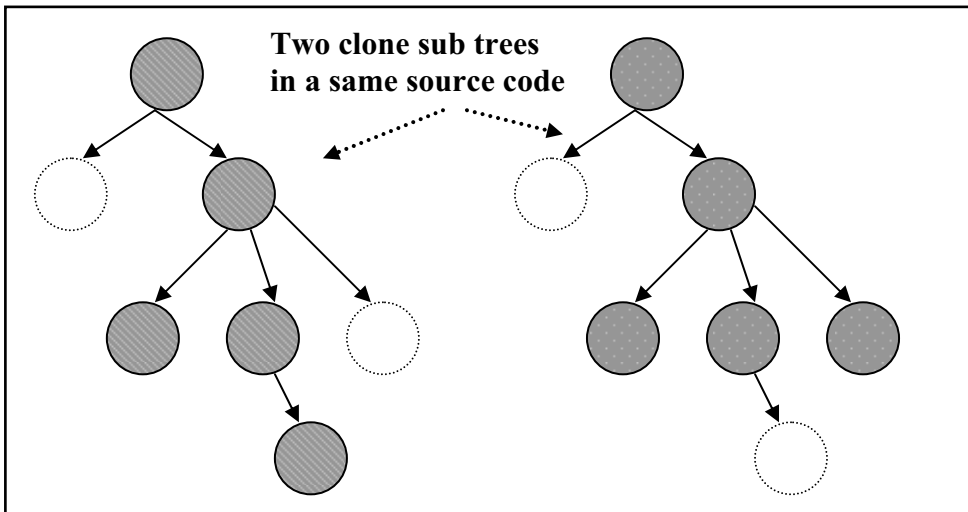


Fig. 24. Two close clones cannot be taken as a single clone

## 7. Conclusion

As the number of web pages extensively increases across the time, the number of possible clones among source codes may also increase. The programmer is always trying to find the easiest way to write the coding and that might result to cloning and would risk the maintenance of the system. As we know, the overall aim of this project is to be familiar with the ability of ontology mapping technique to solve the clone detection between files of different systems. There are already many researches that had done the code clone detection but none of them had used the ontology mapping as part of the detection.

From the findings that we get from the previous chapter, we know that there is a possibility of using a mapping technique to detect clones. Somehow the results shown are not so good and of course the next process should be to refine the proposed methodology in order to get a better result. Below are the strengths of our system:

- (a) Capable in finding structural similarity among XML tree, i.e. structural clone,
- (b) Capable in finding structural similarity among XML tree, i.e. structural clone.

In order for us to get a good result of clone detection, we need to do some refinements to the methodology. Below are a few things that can be considered as the project moves on in aiming for a better recall and precision such as:

- (a) Refine the process of generating vocabulary
- (b) Pre-processing phase where original codes were transformed into standard codes need to be refined to make sure all scripting and dynamic web pages lines of code e.g. PHP and ASP code clones can be detected as well
- (c) In the process of mapping the tags using vocabulary, enhance the searching towards the end of every single page
- (d) Manipulate the subgraph miner so that number subtree generated would be lenient without having any redundancy of subtrees, etc.

## 8. Acknowledgements

This work was supported by the Ministry of Science & Technology and Innovation (MOSTI), Malaysia, and the Research Management Center, Universiti Teknologi Malaysia (UTM), under Vot 79266.

## 9. References

- Al-Ekram, R.; Kapser, C. & Godfrey, M. (2005). Cloning by Accident: An Empirical Study of Source Code Cloning Across Software Systems, *International Symposium on Empirical Software Engineering*
- Antoniou, G. & Van Harmelen, F. (2003). *Web Ontology Language: OWL*, In *Handbook on Ontologies in Information Systems*, 67–92
- Bailey, J. & Burd, E. (2002). Evaluating Clone Detection Tools for Use during Preventative Maintenance, *Proceeding Second IEEE International Workshop Source Code Analysis and Manipulation (SCAM '02)*, IEEE, 36–43
- Bailey, J. & Burd, E. (2002). Evaluating Clone Detection Tools for Use during Preventative Maintenance, *Proceeding Second IEEE International Workshop Source Code Analysis and Manipulation (SCAM '02)*, IEEE, 36–43

- Baker, B. S. (1995). On finding duplication and near- duplication in large software system, *In Proc. 2<sup>nd</sup> Working Conference on Reverse Engineering*, 1995, Toronto, Ont., Canada, IEEE, 86-95
- Baxter, I.; Yahin, A.; Moura, L. & Anna, M. S. (1998). Clone detection using abstract syntax trees. *In Proc. Intl. Conference on Software Maintenance*. Bethesda, MD, USA, IEEE, 368-377
- Baxter, I.D. & Churchett, D. (2002). Using Clone Detection to Manage a Product Line, *In ICSR7 Workshop*
- Bellon, S.; Rainer, K. & Giuliano, A. (2007). Comparison and Evaluation of Clone Detection Tools, *In Transactions on Software Engineering*, 33(9), 577-591
- Benassi, R.; Bergamaschi, S.; Fergnani, A. & Misell, D. (2004). Extending a Lexicon Ontology for Intelligent Information Integration, *European Conference on Artificial Intelligence (ECAI2004)*, Valencia, Spain, 278-282
- Borgelt, B. & Berthold, M. R. (2002). Mining Molecular Fragments: Finding Relevant Substructures of Molecules, *IEEE International Conference on Data Mining (ICDM 2002, Maebashi, Japan)*, 51-58 IEEE Press, Piscataway, NJ, USA
- Brank, J.; Grobelnik, M. & Mladenić, D. (2005). A survey of ontology evaluation techniques, *In SIKDD 2005 at Multiconference IS 2005*
- Breitman, K. K.; Casanova, M. A. & Truszkowsk, W. (2006). *Semantic Web: concepts, technologies and applications*, Springer
- Calasanz, R.T.; Iso, J.N.; Bejar, R.; Medrano, P.M. & Soria, F.Z. (2006). Semantic interoperability based on Dublin Core hierarchical one-to-one mappings, *International Journal of Metadata, Semantics and Ontologies*, 1(3), 183-188
- Calefato, F.; Lanubile, F.; Mallardo, T. (2004). Function Clone Detection in Web Applications: A Semiautomated Approach, *Journal Web Engineering*, 3(1), 3-21
- De Lucia, A.; Scanniello, G. & Tortora, G. (2004). Identifying Clones in Dynamic Web Sites Using Similarity Thresholds, *Proc. Intl. Conf. on Enterprise Information Systems (ICEIS'04)*, 391-396
- Di Lucca, G. A.; Di Penta, M.; Fasilio, A. R. & Granato, P. (2001). Clone analysis in the web era: An approach to identify cloned web pages, *Seventh IEEE Workshop on Empirical Studies of Software Maintenance*, 107-113
- Di Lucca, G. A.; Di Penta, M. & Fasolino, A. R. (2002). An Approach to Identify Duplicated Web Pages, *COMPSAC*, 481-486
- Dou, D. & McDermott, D (2005). Ontology Translation on the Semantic Web, *Journal on Data Semantics (JoDS) II*, 35-57
- Ducassee, S.; Rieger, M. & Demeyer, S. (1999). A Language Independent Approach for Detecting Duplicated Code, *Proceeding International Conference Software Maintenance (ICSM '99)*
- Ehrig, M. (2006). *Ontology Alignment: Bridging the Semantic Gap*, New York, Springer
- Ehrig, M. & Sure, Y. (2000). Ontology Mapping - An Integrated Approach, *Lecture Notes in Computer Science*, No. 3053. 76-91
- Estival, D.; Nowak, C. & Zschorn, A. (2004). Towards Ontology-Based Natural Language Processing, *DF/RDFS and OWL in Language Technology: 4th Workshop on NLP and XML (NLPXML-2004)*, ACL 2004, Barcelona, Spain
- Fox, M. S.; Barbuceanu, M.; Gruninger, M. & Lin, J. (1998). An Organization Ontology for Enterprise Modeling, *In Simulating Organizations: Computational Models of*



- Institutions and Groups*, M. Prietula, K. Carley & L. Gasser (Eds), Menlo Park CA, AAAI/MIT Press, 131-152
- Gašević, D. & Hatala, M. (2006). Ontology mappings to improve learning resource search, *British Journal of Educational Technology*, 37(3), 375-389
- Gruber, T. R. (1993). A translation approach to portable ontologies, *Knowledge Acquisition*, 5(2), 199-220
- Huan, J.; Wang, W. & Prins, J. (2003). Efficient Mining of Frequent Subgraph in the Presence of Isomorphism, in *Proceedings of the 3rd IEEE International Conference on Data Mining (ICDM)*, pp. 549-552
- Ichise, R. (2008). Machine Learning Approach for Ontology Mapping Using Multiple Concept Similarity Measures, *Seventh IEEE/ACIS International Conference on Computer and Information Science (icis 2008)*, IEEE, 340-346
- Visser, P. R. S. & Tamma, V. A. M. (1999). An experience with ontology-based agent clustering, *Proceedings of IJCAI-99 Workshop on Ontologies and Problem-Solving Methods (KRR5)*, Stockholm, Sweden, Morgan Kaufmann, 1-12
- Jiang, L.; Mishergchi, G.; Su, Z.; Glondou, S. (2007). DECKARD: Scalable and Accurate Tree-based Detection of Code Clones, *In Proc. 29th IEEE International Conference on Software Engineering (ICSE 2007)*, IEEE, 96-105
- Jin, T.; Fu, Y.; Ling, X.; Liu, Q. & Cui, Z. (2007). A Method of Ontology Mapping Based on Sub tree Kernel, *IITA*, 70-73
- Kamiya, T.; Kusumoto, S. & Inoue, K. (2002). CCFinder: a multilinguistic token-based code clone detection system for large scale source code, *IEEE Transactions on Software Engineering*, 28(7), 654-670
- Kapser, C. & Godfrey, M. W. (2006). Clones considered harmful, *In Working Conference on Reverse Engineering*
- Komondoor, R. & Horwitz, S. (2001). Using slicing to identify duplication in source code, *In Proceedings of the 8th International Symposium on Static Analysis*, July 16-18, 2001, Paris, France, In SAS. 40-56
- Kontogiannis, K.; De Mori, R.; Merlo, E.; Galler, M. & Bernstein, M. (1996). Pattern matching for clone and concept detection, *Automated Soft. Eng.*, 3(1/2), 77-108
- Krinke, J. (2001). Identifying Similar Code with Program Dependence Graphs, *Proceedings of the Eight Working Conference on Reverse Engineering*, October 2001, Stuttgart, Germany, IEEE, 301-309
- Lanubile, F. & Mallardo, T. (2003). Finding Function Clones in Web Applications, *Proceeding Conference Software Maintenance and Reengineering*, 379- 386
- Li, Z.; Lu, S.; Myagmar, S. & Zhou, Y. (2006). CP-Miner: finding copy-paste and related bugs in large-scale software code, *IEEE Computer Society Transactions on Software Engineering*, 32(3), 176-192
- Maedche, A. & Staab, S. (2002). Measuring Similarity between Ontologies, *Lecture Notes in Computer Science*, 251
- Mayrand, J. & Leblanc, C. (1996). Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics, *In Proc. Conference on Software Maintenance*
- McGuinness, D. L. (1999). Ontologies for Electronic Commerce, *Proceedings of the AAAI '99 Artificial Intelligence for Electronic Commerce Workshop*, Orlando, Florida

- Nijssen, S. & Kok, J. N. (2004). A Quickstart in Frequent Structure Mining can make a Difference, *LIACS Technical Report*
- Noy, N. F. (2004). Semantic Integration: A Survey Of Ontology-Based Approaches, *In ACM SIGMOD Record, Special Section on Semantic Integration*, 33(4), 65-70
- Qian, P. & Zhang, S. (2006a). Ontology Mapping Approach Based on Concept Partial Relation, *In Proceedings of WCICA*
- Qian, P. & Zhang, S. (2006b). Ontology Mapping Meta-model Based on Set and Relation Theory, *IMSCCS* (1), 503-509
- Koschke, R. (2006). Survey of Research on Software Clones, *Dagstuhl Seminar Proceedings*
- Rajapakse, D. C. & Jarzabek, S. (2005). An Investigation of Cloning in Web Applications, *5th Intl Conference on Web Engineering (ICWE'05)*, Washington, DC, IEEE, 252-262
- Roy, C. K. & Cordy, J. R. (2007). A Survey on Software Clone Detection Research, *Technical Report 2007-541, School of Computing, Queen's University, Canada*
- Sabou, M.; D'Aquin, M. & Motta, E. (2006). Using the semantic web as background knowledge for ontology mapping, *ISWC 2006 Workshop on Ontology Mapping*
- Schleimer, S.; Wilkerson, D. S. & Aiken, A. (2003). Winnowing: Local Algorithms for Document Fingerprinting, *Proceeding SIGMOD International Conference Management of Data*, 76-85
- Stevens, R. D.; Goble, C. A. & Bechhofer, S. (2000). Ontology-based knowledge representation for bioinformatics, *Brief Bioinform* 1(4), 398-416
- Stoilos, G.; Stamou, G. & Kollias, S. (2005). A String Metric for Ontology Alignment, in *Proceedings of the ninth IEEE International Symposium on Wearable Computers*, Galway, 624- 237
- Stumme, G. & Maedche, A. (2001). FCA-Merge: BottomUp Merging of Ontologies, *IICAL*, 225-234
- Stutt, A. (1997). Knowledge Engineering Ontologies, Constructivist Epistemology, Computer Rhetoric: A Trivium for the Knowledge Age, *Proceedings of Ed-Media '97*, Calgary, Canada
- The World Wide Web Consortium Official Site at [www.w3c.org](http://www.w3c.org)
- Todorov, K. (2008). Combining Structural and Instance-based Ontology Similarities for Mapping Web Directories, *The Third International Conference on Internet and Web Applications and Services*
- Ueda, Y.; Kamiya, T.; Kusumoto, S. & Inoue, K. (2002). Gemini: Maintenance support environment based on code clone analysis, *In Proceedings of the 8th IEEE Symposium on Software Metrics (METRICS'02)*, Ottawa, Canada, 67-76
- Ueda, Y.; Kamiya, T.; Kusumoto, S. & Inoue, K. (2002). On detection of gapped code clones using gap locations, *In Proceedings 9th Asia-Pacific Software Engineering Conference (APSEC'02)*, Gold Coast, Queensland, Australia, 327-336
- Vallet D.; M. Fernández & P. Castells (2005). An Ontology-Based Information Retrieval Model, *Proc. Second European Semantic Web Conf. (ESWC '05)*
- Visser, P. R. S.; Jones, D. M.; Beer, M. D.; Bench-Capon, T. J. M., Diaz, B. M. & Shave, M. J. R. (1999). Resolving Ontological Heterogeneity in the KRAFT Project, *In Proceedings of Database and Expert Systems Applications 99, Lecture Notes in Computer Science* 1677, Berlin, Springer, 688-697
- Winkler, W. E. (1999). The State of Record Linkage and Current Research Problems, *Statistical Society of Canada, Proceedings of the Section on Survey Methods*, 73-79

- Yan, X. & Han, J. (2002). gSpan: Graph-Based Substructure Pattern Mining, *Proc. 2002 of Int. Conf. on Data Mining (ICDM'02)*, Expanded Version, UIUC Technical Report, UIUCDCS-R-2002-2296
- Zhang, Z.; Xu, D. & Zhang, T. (2008). Ontology Mapping Based on Conditional Information Quantity, *IEEE International Conference on Networking, Sensing and Control, 2008, ICNSC 200*, Sonya, IEEE, 587-591