Code Clone Detection using Sequential Pattern Mining

Ali El-Matarawy
Faculty of Computers and Information, Cairo University

Mohammad El-Ramly
Faculty of Computers and Information, Cairo University

Reem Bahgat
Faculty of Computers and Information, Cairo University

ABSTRACT
This paper presents a new technique for clone detection using sequential pattern mining titled EgyCD. Over the last decade many techniques and tools for software clone detection have been proposed such as textual approaches, lexical approaches, syntactic approaches, semantic approaches …, etc. In this paper, we explore the potential of data mining techniques in clone detection. In particular, we developed a clone detection technique based on sequential pattern mining (SPM). The source code is treated as a sequence of transactions processed by the SPM algorithm to find frequent itemsets. We run three experiments to discover code clones of Type I, Type II and Type III and for plagiarism detection. We compared the results with other established code clone detectors. Our technique discovers all code clones in the source code and hence it is slower than the compared code clone detectors since they discover few code clones compared with EgyCD.

Keywords
Sequential Pattern Mining, Clone Detection, Data Mining

1. INTRODUCTION
It is very common in computer programming to copy part of the program from one place and paste it in another place and then adapt it to fit in the new place. This happens for a variety of reasons [3]. As a result, software systems often contain sections of code that are very similar, called code clones [1]. Previous research shows that a significant fraction (between 7% and 23%) of the code in a typical software system has been cloned [2, 3]. Sometimes code clones are created for legitimate reasons, but other times they are not and they deteriorate the quality of the code. One of the main drawbacks of code clones is that the developer should modify multiple copies of the same pieces of code if a change is needed in a piece of code that has been cloned. Often this does not happen with good quality because the programmer forgets where (s)he duplicated the code and leaves some clones unchanged. Fortunately, several (semi-automated) techniques for detecting code clones have been proposed to help the programmer find code clones and locate the locations of duplicate code [1].

A recent study that was done on industrial systems shows that inconsistent changes/updates to cloned code are frequent and lead to severe unexpected behavior [4]. Several other studies also show that software systems with code clones can be more difficult to maintain [5, 6] and can introduce subtle errors [7, 8]. Thus code clones are considered one of the bad "smells" of a software system [9] and it is widely believed that cloned code can make software maintenance and evolution significantly more difficult. Thus the detection, monitoring and removal of code clones are important topics in software maintenance and evolution [1, 9].

Over the last decade many techniques and tools for software clone detection have been proposed [1]. This includes textual approaches, lexical approaches, syntactic approaches, semantic approaches, among others. Most of them are oriented to a specific computer language and they range from high precision to low precision, and from high recall to low recall [27]. Little work was done to explore the potential of using data mining techniques in code clone detection. In this work, we developed a new method for code clone detection that uses sequential pattern mining [21] for detecting code clones. Our method treats source code lines as transactions and its words as items. Then, we search for the most frequent itemsets. We ran three experiments to evaluate our method and compared it with state-of-the-art clone detectors that use other techniques. Our approach was able to recover all code clones of Type I and Type II. It also recovered clones of Type III with high precision and high recall features. A key feature of our technique is that it is language independent. Our detector was written as a highly optimized database application using Adaptive server SQL anywhere as a database engine and PowerBuilder as a front end tool, which they are very suitable for data mining techniques.

The rest of this paper is organized as follows. After presenting some basic definitions and terminologies regarding clones in section 2, we introduce some related work on clone detection in section 3. In section 4 we introduce an overview for data mining and its techniques, particularly the ones relevant to code clone detection. In sections 5, 6 and 7, we introduce our new approach for detecting clones. Three case studies are reported in section 8. Section 9 analyzes the results and discusses advantages and limitations of our approach. Finally the paper is concluded in section 10 with statement of future work.

2. BASIC DEFINITIONS
Mainly we followed the same basic definitions mentioned in [1, 3].

Definition 1: Code Fragment. A code fragment is a continuous part of the source code, may consist of one or more lines. It can be of any granularity, e.g., function definition, begin-end block, or sequence of statements.

Definition 2: Code Clone. A Clone occurs when a code fragment is an identical to another code fragment according to some basic criteria. These criteria may be syntactical, semantical, or both of them. Clones can be typically identical, or a having some differences such as in renaming identifiers.

Definition 3: Clone Types. There are two main kinds of similarity between code fragments. Fragments can be similar based on the similarity of their program text, or they can be similar based on their functionality (independent of their text). The first kind of clone is often the result of copying a code fragment and pasting it into another location. In the following we provide the types of clones based on both the textual

(Types 1 to 3) [10] and functional (Type IV) [23, 24] similarities.

Type I: Identical code fragments except for variations in whitespace, layout and comments.

Type II: Syntactically identical fragments except for variations in identifiers, literals, types, whitespaces, layout and comments.

Type III: Copied fragments with further modifications such as changed, added or removed statements, in addition to variations in identifiers, literals, types, whitespaces, layout and comments.

Type IV: Two or more code fragments that perform the same computation but are implemented by different syntactic variants.

Definition 4: Plagiarism. Plagiarism is a form of cheating. In other words, plagiarism is an act of fraud. It involves stealing someone else's work and lying about it afterward [25]. We mean here by cheating is trying to steal the others by making a copy from others' document and paste it in your document.

Clone Relation Terminologies

Clone detection tools report clones in the form of Clone Pairs (CP) or Clone Classes (CC) or both. These two terms speak about the similarity relation between two or more cloned fragments. The similarity relation between the cloned fragments is an equivalence relation (i.e., a reflexive, transitive, and symmetric relation) [11]. A clone-relation holds between two code portions if (and only if) they are the same sequences. Sequences are sometimes original character strings, strings without whitespace, sequences of token type, transformed token sequences and so on. In the following we define clone pair and clone class in terms of the clone relation [12]:

Clone Pair: A pair of code portions/fragments is called a clone pair if there exists a clone-relation between them, i.e., a clone pair is a pair of code portions/fragments which are identical or similar to each other.

Clone Class: A clone class is the maximal set of code portions/fragments in which any two of the code portions/fragments hold a clone-relation, i.e., form a clone pair.

3. RELATED WORK

According to [1] clone detection techniques can be divided into string-based, token-based, parse tree-based, metrics-based, PDG-based or hybrids-based techniques.

In *string-based* techniques, the target source program is considered as a sequence of lines/strings. Two code fragments are compared with each other to find sequences of same text/strings. Once two or more code fragments are found to be similar in their maximum possible extent, the target source program is considered as sequence of lines/strings. Two code fragments are compared with each other to find sequences of same text/strings. Once two or more code fragments are found to be similar in their maximum possible extent (e.g., w.r.t maximum no. of lines), they are returned as clone pair or clone class by the detection technique.

In the *token-based* detection approach, the entire source system is lexed/parsed/transformed to a sequence of tokens. This sequence is then scanned for finding duplicated subsequences of tokens and finally, the original code portions representing the duplicated subsequences are returned as clones. Compared to text-based approaches, a token-based approach is usually more robust against code changes such as formatting and spacing.

In the *tree-based* approach, a program is parsed to a parse tree or an abstract syntax tree (AST) with a parser of the language of interest. Similar sub-trees are then searched in the tree with some tree matching techniques and the corresponding source code of the similar sub-trees is returned as clone pair or clone class. The parse tree or AST contains the complete information about the source code. Although the variable names and literal values of the source are discarded in the tree representation, more sophisticated methods for the detection of clones can still be applied. *PDG-based* approaches [13, 14, 15] go one step further in obtaining a source code representation of high abstraction than other approaches by considering the semantic information of the source code. PDG [16] contains the control flow and data flow information of a program and hence carries semantic information. Once a set of PDGs are obtained from a subject program, isomorphic sub-graph matching algorithm is applied for finding similar sub-graphs which are returned as clones.

Metrics-based approaches gather different metrics for code fragments and compare these metrics' vectors instead of comparing code directly. There are several clone detection techniques that use various software metrics for detecting similar code. First, a set of software metrics called fingerprinting functions are calculated for one or more syntactic units such as a class, a function, or a method or even statement and then the metrics' values are compared to find clones over these syntactic units. In most cases, the source code is parsed to its AST/PDG representation for calculating such metrics.

Hybrid-based techniques use a combination of syntactic and semantic characteristics. Leitao [17] provides a hybrid approach that combines syntactic techniques based on AST metrics and semantic techniques (using call graphs) in combination with specialized comparison functions [1].

A comparison of the techniques known from literature has shown that so far there exists no single method that is superior to all other methods in all situations [18, 19]. All techniques have certain advantages and disadvantages. Techniques that detect many clones (high recall) also return many code fragments which are not clones (lower precision). In turn, techniques with a high precision will usually have a lower recall [27**]**.

A related tool, Clone Miner, uses a data mining algorithm in detecting code clones [26]. It has the following features:

- It is developed in C++.
- It use "market basket analysis" to detect code clones
- It detects Type I and Type II only.
- It converts source code to XML using third party tools and then processes the XML files. Hence, if some source files fail to be converted to XML, it cannot process them. It compares the code structurally, i.e., it does the detection on AST.
- It divides the entry type of data into same file, different files or directories, so it has some specific code for each type. (similarly in [27]).

On the other hand EgyCD has the following features:

- It uses Apriori [21] sequential pattern mining algorithm.
- It detects three types (Type I, Type II, Type III) as well as plagiarism.
- It uses a database tool which is more suitable for data mining algorithm.
- It works directly on the source code not on XML representation of the code.
- It detects all code clones regardless the source code is in one file, different files or directories.
- It is not oriented towards a specific language

4. SEQUENTIAL PATTERN MINING

Data mining [20, 21] is the process of extracting interesting (non-trivial, implicit, previously unknown and potentially useful) information or patterns from large information repositories such as: relational database, data warehouses, XML repository, etc. Also data mining is known as one of the core processes of Knowledge Discovery in Database (KDD).

Sequential pattern mining [21] is trying to find the relationships between occurrences of sequential events, to find if there exists any specific order of the occurrences.

In sequential pttern mining [22] frequent itemsets are used to illustrate relationships within large amounts of data. The classical example is the analysis of the buying-behavior of customers. The database consists of a set of transactions, and each transaction is a set of items from a universal itemset *I*.

The goal is to find itemsets *I* that are subsets of many transactions *T* in the database *D*, ($I \subseteq T$). An itemset is called frequent, if it occurs in a percentage that exceeds a certain given support count σ [27]:

$$\sigma(I) = \frac{|\{T \in D\}|\{I \subseteq T\}|}{|D|} \geq \sigma$$

In EgyCD, we are not interested in the percentage of itemsets. Instead, we are interested in their count:

$$\sigma(I) = |\{T \in D\}|\{I \subseteq T\}| \geq \sigma \text{ where } \sigma > 1$$

Most SPM algorithms are based on Apriori algorithm [21], AprioriAll. Sequential pattern mining was first introduced in [23] by Agrawal, and three Apriori based algorithms were proposed. Given the transaction database with three attributes customer-id, transaction-time and purchased-items, the mining process was decomposed into five phases:

Sort Phase: the original transaction database is sorted with customer-id as the major key and transaction time as the minor key, the result is a set of customer sequences.

L-itemsets Phase: the sorted database is scanned to obtain large 1-itemsets according to the predefined support threshold.

Transformation Phase: the customer sequences are replaced by those large itemsets they contain, and all the large itemsets are mapped into a series of integers to make the mining more efficient. At the end of this phase the original database is transformed into a set of customer sequences represented by those large itemsets.

Sequence Phase: all frequent sequential patterns are generated from the transformed sequential database.

Maximal Phase: those sequential patterns that are contained in other super sequential patterns are pruned in this phase, since we are only interested in maximum sequential patterns.

Since most of the phases are straightforward, researchers focused on the sequence phase in [17].

5. GENERAL DESCRIPTION OF EgyCD

Following Apriori-based approaches, our approach builds up larger itemsets (clones in this case) from combining smaller ones and then efficiently searches the source code to verify their presence. It does not compare source code segments as done in many other code clone detectors.

EgyCD tool consists of four steps:

1. The user selects the source files either it is in the directory or in different directories to apply the tool on.
2. The tool transforms the source code to transactions of itemsets.
3. EgyCD algorithm is applied to discover frequent itemsets in the source code that exceed a given frequency threshold.
4. The algorithm prunes all code clones that appear completely in other code clones to avoid duplicate results and report only original clones not included in others.

Now we briefly describe how EgyCD algorithm works. Assume that T is the set of all source code statements, where each statement is considered a transaction. First, the algorithm starts by getting the first itemset F which is the set of all repeated statements in the source code. Then it initializes a counter i to 1. It also initializes a set CC to be equal to F, where CC is a set that will always contain all code clones discovered so far. Set CC_i is a subset of CC that always contains all code clones of length i while i increases from an iteration to the next. Another set S_i will always contain all possible code clones of length i. The second step is to do Cartesian product CC_i x F and store the results in S. The third step is checking each item in the Cartesian product of length i + 1 to see if it exists in the set of all transactions T (i.e., the set of all source code lines in sequence) or not. If an item in the Cartesian product set exists as subsequence of transactions in T, then we add it to the code clones set, CC. Since the result of the Cartesian product can be massive, it is possible to generate the results on the fly in the memory without storing them and process them directly in the third step by checking their presence in the transactions. The fourth step prunes all code clones in CC of length i that exist in code clones of length i + 1. The fifth step is incrementing i by 1. The sixth step is trying to reduce the set F by pruning all items that didn't appear as a last item in any of code clones of length i. Finally the algorithm iterates over steps two to six until all items of the Cartesian product don't exist in any transactions. Below is the pseudo code of the algorithm.

```
T = set of all source code statements
F = set of repeated statements in the
code
CC = F
stillMore = true
i = 1
While (stillMore)
{
        stillMore = false
        S_{i+1} = CC_i x F
        If i > 1 then
                S_{i+1} = Check_Apriori(S_{i+1})
        End if
        For all e ∈ S_{i+1}
        {
```

```
            if e ∈ T then
                  add e to CC
                  stillMore = true
            end if
      }
      prune CC by removing all e ∈ CC
      where |e| = i and e ⊂ f and f ∈ CC
      where |f| = i+1

      i = i + 1
      prune all non used elements in F
}
```

Figure 1. Pseudo-code of EgyCD Algorithm

```
Check_Apriori(S_{i+1})
{
      For all e ∈ S_{i+1}
      {
            X = all elements in e except
            first element

            if e ¢ S_i then
                  prune e from S_i
            end if
      }
      Return S_{i+1}
}
```

Figure 2. Pseudo-code of `Check_Apriori(`S_{i+1}`)`

Below is an example explaining how the algorithm works to detect code clones of Type I.

Type I Example
Suppose we have the following code:

```
c = a+ b;
d = 2 * n;
.........
.........
c = a+ b;
d = 2 * n;
.........
.........
```

The final result should be CC= {(c = a + b; , d = 2 * n;))}

First iteration
F = {`c = a + b;` , `d = 2 * n;`}
CC = FFound = true
i = 1
`stillMore` = true
iteration 1:
{
`stillMore` = false
S_{i+1}={c= a+ b; , d = 2 * n;} x {c = a + b; , d = 2 * n;}
S_{i+1} = {(c = a + b; , c = a + b;)
, (c = a + b; , d = 2 *n;)
, (d = 2 *n; , c = a + b;)
, (d = 2 *n;; , d = 2 *n;)}
CC = { c = a + b; , d = 2 * n;
, (c = a + b; , d = 2 *n;)}
CC = { (c = a + b; , d = 2 *n;) }
`stillMore` = true
i = 2
F = { d = 2 * n; } // after pruning
}
Iteration2:
{
`stillMore` = false
S_{i+1} = { (c = a + b; , d = 2 *n;) } x { d = 2 * n; }
S_{i+1} = { (c = a + b; , c = a + b; , d = 2 * n) }
S_{i+1} = Φ // After `Check_Apriori(S)`
`stillMore` = false
CC = { (c = a + b; , d = 2 *n;) }
i = 3
F = Φ
}
No more loops since `stillMore` = false and
CC = { (c = a + b; , d = 2 *n;) }

6. OPTIMIZATION TRICKS ADDED TO APRIORI

We have done some modifications to Apriori for increasing the speed of EgyCD such as:-

a. Pruning F at the end of each iteration to decrease the cardinality of the first itemset and consequently the cardinality of the resultant set of the Cartesian product.
b. The Apriori property states that any subset of a frequent set is frequent [21]. For stores system sorting items in transactions is meaningless but in code clones sorting statements is a major concept, so we check apriori property only for one subset which is the union of a code clone but after removing the first statement of that code clone and the new added statement.
c. By using the SQL features in where conditions we get all items of Si+1 that exist in sequence in the source code then we check if it is a code clone or not.
d. EgyCD is applied inside the database and not in the application.

7. IMPLEMENTATION DETAILS

The algorithm was implemented in a database application using Adaptive Server SQL Anywhere version 11.0 with add on In-Memory version 11.0 and PowerBuilder 11.5. This has multiple advantages. First, it perfectly matches the application of Apriori-based algorithms which are developed for mining databases. Second, the expressive power of SQL supports processing of transactions very easily and smoothly. Finally, PowerBuilder has powerful visualization capabilities that allow us to visualize code clones in very simple ways and can also be upgraded with new views if needed. For every language to be supported, language specific tables are filled with the style of comments, reserved words and symbols, begin and end markers of compound statements, statements separator, etc.

The proposed algorithm can be applied to Type I, Type II and Type III but not Type IV. It can also be used to detect plagiarism in written text not only in source code. We discuss the specifics of each code clone type below.

We have 2 modes for EgyCD, prune and no prune, if the user wants to see all code clones and its subsets code clones in the source code, (s)he will choose no prune mode and if the user

wants to see only all code clones and the program should delete all the code clones subsets, hence the user should select prune mode.

7.1. Detecting Type I

Applying EgyCD on Type I is very straightforward. The only preprocessing step needed is removing white spaces, tabs, comments, etc. After that we convert the source code into a database structure then we apply the algorithm on it to get all code clones.

7.2. Detecting Type II

In this type, we first apply the preprocessing step applied to Type I. Then we do a transformation by replacing each non-reserved word in the source code by the letter "X". We also replace any data type by the letter "T". This is shown in Figure 3. Then, we convert the source code into a database structure and we apply the algorithm.

`int a, b, c ;`	`int x, y, z ;`
`cin >> a >> b;`	`cin >> x >> y;`
`c = a + b;`	`z = x + y;`
`cout ≪ c;`	`cout ≪ z;`

3.a Source code before transformation

T X, X,X ;	*T X, X,X ;*
cin>>X>>X;	*cin>>X>>X;*
X=X+X;	*X=X+X;*
cout≪X;	*cout≪X;*

3.b Source code after transformation

Figure 3

7.3. Detecting Type III

In order to detect Type III, we do the same process as in detecting Type II. But in converting the source code into database structure, we keep track of the boundaries of code segments (specifically functions, methods and blocks) in a specific database table. These information are segment number, segment start line and end line, file name and parent segment number in case of nested segments.

After applying EgyCD to detect code clones of Type II, we do some calculations to detect code clones of Type III in the segments identified in the source code. As we mentioned in 3.b, each line in the transactions belongs to a code segment; simply the calculations calculates first code segment clones from the code clones that are generated by the proposed algorithm. Now we have N code segments (CS1,CS2,….CSn) that will be similar to each other by a clone percentage. For each code segment, we will calculate its code segment clone percentage, (CSCPi), to be equal to the number of repeated items in the CSi over the total number of items in the CSi. We then remove all code segment clones that will have CSCP less than the percentage that will be defined by the user and those removed code segments will be subtracted from N. If N became 1 then regardless the CSCP of its related code clones segments it will be removed, i.e the value one means no code clones exist for that N code segments. After that we will prune all CS clones that are subset of other CS clones if their parent code segments are similar.

To clarify further the detection of Type III, we give the following example:-

Suppose we have two code fragments of Type III and after applying transformation as in detecting Type II they become as in figure 4. Suppose also that the user sets 55% for the threshold percentage.

`1. TX,X,X ;`
`2. cin >> X >> X;`
`3. X=X+X;`
`4. ---------`
`5. ---------`
`6. ---------`
`7. Cout≪ X;`
	`1. TX,X,X ;`
	`2. cin >> X>> X;`
	`3. ---------`
	`4. X=X+X;`
	`5. Cout≪ X;`

Figure 4. Source code after transformation

For the left code fragment we get its CSCP = 4/7 = 57% and for the right code fragment we get its CSCP = 4/5 =80%. EgyyCD will detect these two code fragments as code clones of Type III and it will display them with their corresponding CSCP.

7.4. Displaying the Plagiarized Text and its Quality

To easily visualize the detected code clones, EgyCD lets the user defines the quality of the code clones in the application setting screen. Four fields are given for controlling the display of code clones, two fields for defining the excellent degree of similarity for code clones, the length of the code clone field and the counting of code clones field. The same two fields are used for defining the good degree of similarity for code clones. If the resultant text clone length is greater than or equal to the length field value for excellent quality and its repetition is greater than or equal to the counter of the code clone field value for excellent quality then the background of this code clone will be in red. However, if the resultant code clone length is greater than or equal to the length field value for good quality and its repetition is greater than or equal to the count code clone field value for good quality then the background of this code clone will be in orange otherwise the text clone background color is green.

By using this way, the user can easily notice and differentiate the most important text clones.

7.5. Calculating Code Clone File Ratio

To submit some information that may be useful to EgyCD users, we calculate a ratio called code clone file ratio (CCFR) for each file selected by the user for detecting code clones inside it. It is equal to the full size in lines of all code clones inside the file over the total size of the file in lines.

CCFR = Size of code clones in the file in lines / size of the file in lines

The user can see this ratio if (s)he displayed again her/his selected files. The user will find that this ratio is calculated and displayed in the row of each file. If the ratio is greater than a specific percentage set by the user in the EgyCD setting then the background color will be red for this row. Otherwise the background will be in white.

8. CASE STUDIES

We have three cases studies. The first case compares the number of code clones and its corresponding time among EgyCD, NICAD [28] and simCAD [29] for Type I. The second case compares the number of code clones and its corresponding time among the same three tools for Type II. The third case study was done on a large Java system to detect code clones of Type I. No need to do a case study for Type III

since Type II is a sub-case from Type III and it will give the whole information we need about the algorithm.

The first and the second case studies are applied for the same set of files. This is the example set of 25 C language files bundled with NICAD's clone detector. We divided them into 5 groups; the first group contains 5 files and each consequent group contains the files of the pervious group and has 5 additional files. So, the last group contains 25 files. The total size of these files is 332 KB and they collectively contain about 8545 lines of code.

The third case study is for a very large scale to show that EgyCD can detect code clones for large scale systems. We randomly selected 2151 files from the JDK. Their size is 21.8 MB.

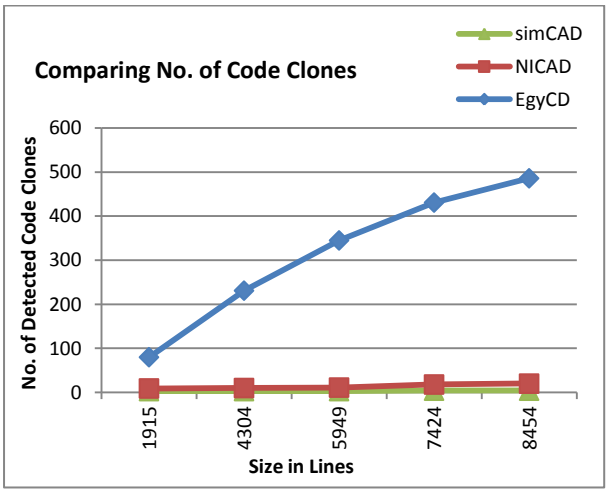
We choose NICAD and simCad tools for three reasons. First, they are relatively mature and acceptable in the scientific community. Second, they are available for use and their authors kindly supported us when using them and running the comparisons. Third, some of them, particularly NICad, were already used to examine some of the target systems and the results were available by the authors which ensure the validity of our results using the same tool on the same system.

8.1. The first case study

The purpose of the first case study experiment is to find out how EgyCD performs relative to other tools in terms of time, total number of code clones and the number of code clones discovered per second. EgyCD was compared against NICAD and simCAD for finding Type I clones.

Seq.	Size in Lines	EgyCD		NICAD		simCAD	
		No. of Clones	T. in Sec.	No. of Clones	T. in sec.	No. of Clones	T. in sec.
1	1915	80	3.00	9	0.08	3	1.2
2	4304	231	5.00	10	0.1	3	1.5
3	5949	345	7.00	11	0.5	3	1.8
4	7424	431	8.00	18	0.6	4	2.1
5	8454	486	12.00	20	0.7	4	2.4

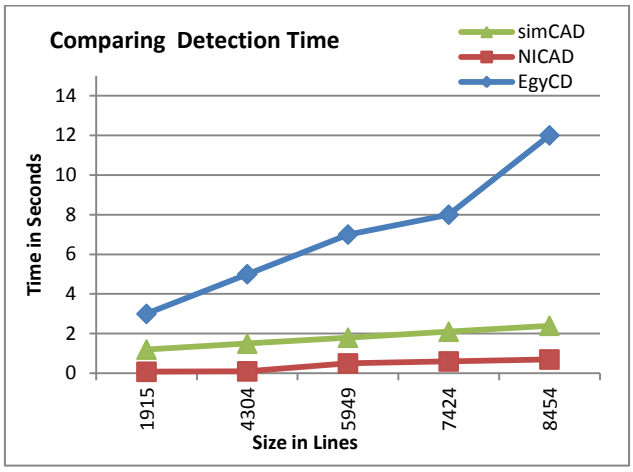
Table 1. Results of Running EgyCD, NICAD and SimCad on C files to detect Type I



Graph (1) Comparison of No. of Type I Code Clones Detected

Graph (1) and Table (1) compare the number of code clones detected by each tool, since EgyCD uses an Apriori-based algorithm, it comprehensively detects all code clones in the source code. Hence, EgyCD has a high precision and high recall; it detects all code clones regardless of whether they are meaningful or not.

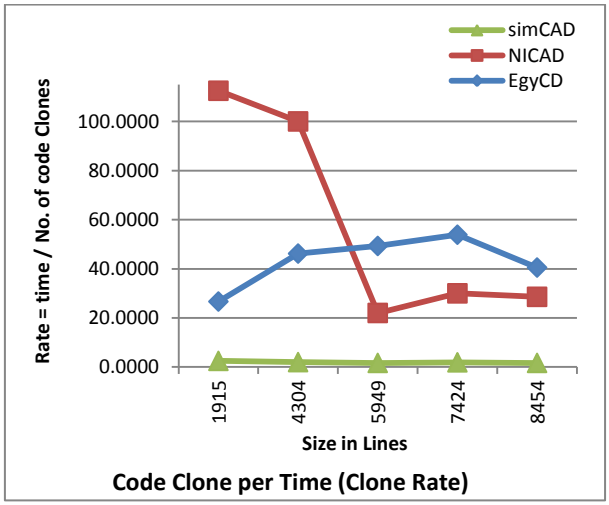
As an Aprior-based algorithm, EgyCD builds code clones without comparing among the source code functions or blocks such as NICAD and simCAD.



Graph (2) Comparison of clone detection time by each tool for Type I

Graph (2) and Table (1) compare the detection time for the same group of files for each tool. EgyCD is slower in comparison with NICAD and simCad, and this is because EgyCD discovers much more code clones than NICAD.

It is also due to the nature of the EgyCD algorithm of building code clones, especially in getting the second itemset of code clones since its cardinality is so high and equals to the square of the first itemset cardinality; the first itemset cardinality equals to all items (lines) in the source code that appear more than once in the source code.



Graph (3). Comparison of the time rates among the three tools

To get the rate comparison among the three tools, we divided the number of code clones detected in the source code by the code clones detection time. We found that EgyCD and simCAD almost have the same values but NICAD is different especially in the first 2 points only as the graph illustrates.

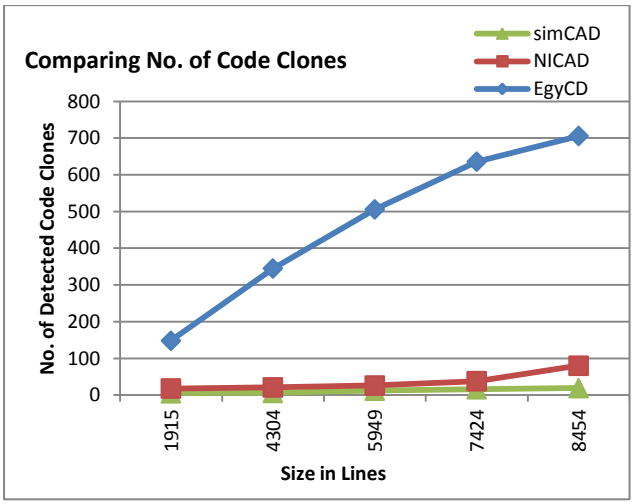
8.2. The second case study

In this case study, we replicated case study 1 for the same purposes but search for code clones of Type II.

The second case study compares the number of code clones and its corresponding time among EgyCD, NICAD and simCAD for Type II**.**

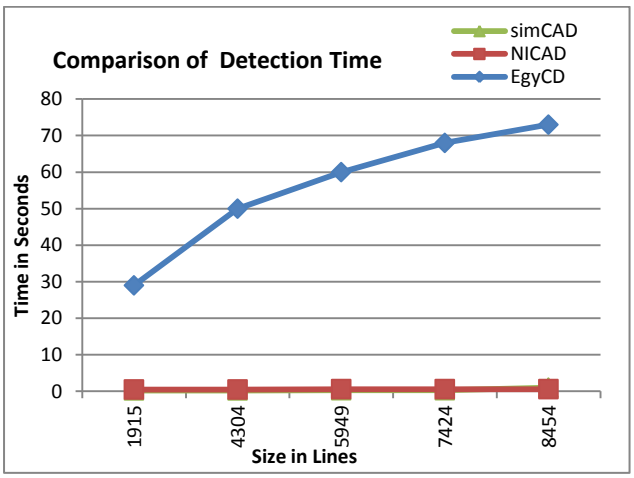
Seq.	Size in Lines	EgyCD		NICAD		simCAD	
		No. of Clones	T. in Sec.	No. of Clones	T. in sec	No. of Clones	T. in Sec.
1	1915	148	29.00	18	0.5	5	0
2	4304	345	50.00	21	0.5	6	0
3	5949	506	60.00	26	0.6	12	0
4	7424	636	68.00	38	0.6	16	0
5	8454	706	73.00	80	0.6	19	1

Table (2) Results of Running EgyCD, NICAD and SimCad on C files to detect Type II



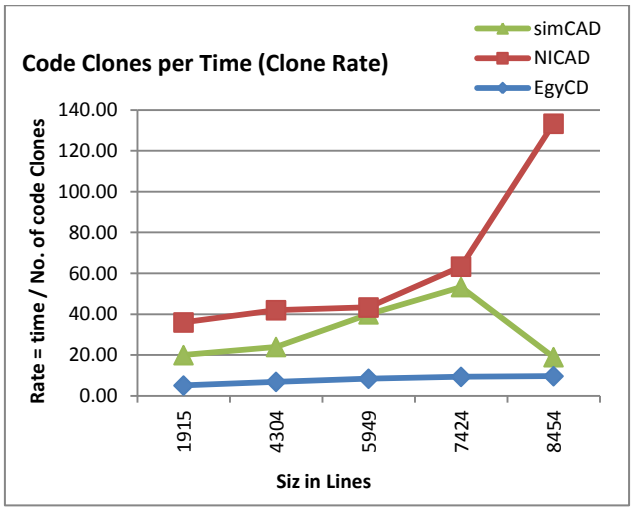
Graph (4). Comparison of No. of code clones detected by each tool for Type II

Graph (4) and Table (2) compare the number of code clones detected by each tool.



Graph (5). Comparison of detection time by each tool for Type II

Graph (5) and Table (2) compare the detection time of code clones by each tool.



Graph (6). Comparison of time rates among the three tools

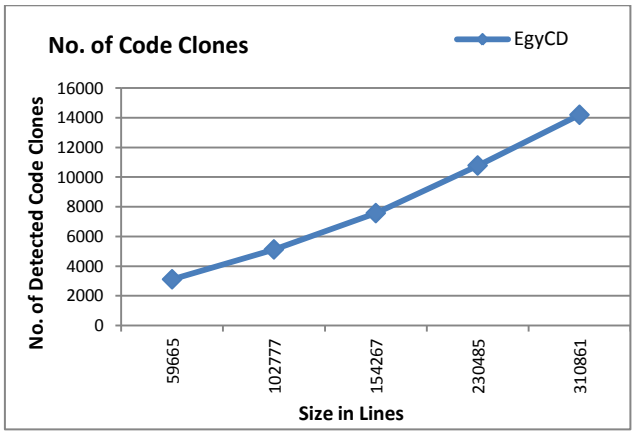
We found that the three tools almost have the same values.

8.3. The third case study

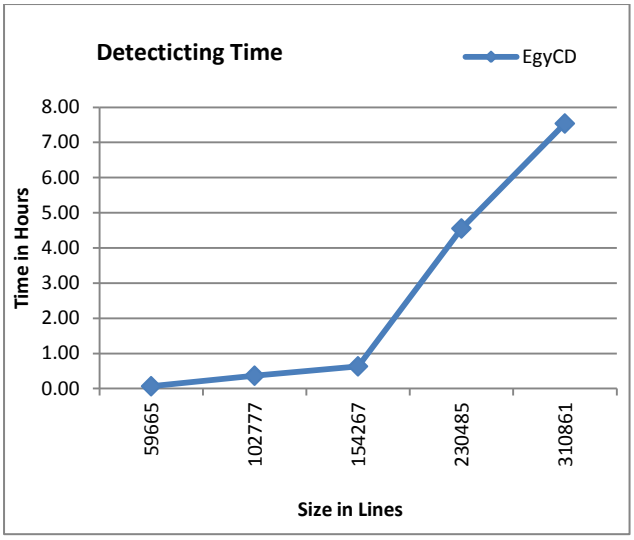
The third case study is done on a large Java system to examine the efficiency of EgyCD in detecting clones in large systems. We selected random files from Java JDK, with total size of almost 21 MB and 310861 LOCs.

EgyCD					
Seq.	No. of Files	No. Of Lines	Clone Size (Lines)	No. of Code Clones	Time in Hours
1	629	59665	10502	3105	0.07
2	868	102777	17895	5118	0.37
3	1315	154267	27392	7565	0.63
4	1782	230485	41328	10772	4.55
5	2151	310861	53420	14189	7.54

Table (3). Results of Running EgyCD on a large system



Graph (7). No. of Code Clones detected by EgyCD in a large system



Graph (8). Code Clones Detection Time by EgyCD in a large system

9. ANALYSIS OF THE RESULTS AND ADVANTAGES AND LIMITATIONS OF EgyCD

In this section we analyze the results of our experiments and discuss the pros and cons of EgyCD. Our experiments showed the following:

- By being Aprior-based, EgyCD has 100% recall and 100% precision. It is very accurate and can detect all code clones and do pruning if required for all code clones that are subset of other code clones. This accuracy comes from using Apriori algorithm and not using an empirical method.
- EgyCD is not oriented to a specific language so it can be applied on an source code.
- It can be applied to Type I, Type II and Type III.
- It can be applied on very large scale systems

On the other hand,

- It is not slow in execution but it is slower than other famous algorithms. This disadvantage comes from two factors, the first one is the algorithm high precision and high recall in code clone detection, the second is that the main core of the algorithm depends on database processing and therefore we switch to hard disk processing many times and this for sure will slow the algorithm execution time. To improve its speed, the user can use EgyCD after specifying her/his interest in a specific clone count, i.e. the user is interested in code clones that are repeated more than a specific number, also this will increase EgyCD speed, and as we mentioned in section 6, we increased the speed of EgyCD by applying it inside the database itself not in the application.

10. CONCLUSIONS AND FUTURE WORK

In this paper, we presented a new clone detection algorithm that utilizes sequential pattern mining to discover code clones. We implemented the algorithm in a database-based language-independent clone detector tool. It detects all code clones in the source code with 100% recall due to the nature of the Apriori-based algorithm. Precision was shown by experimental studies to be very good. The proposed method is not limited to a specific programming language and it can detect code clones of Type I, Type II and Type III. We presented a comparison with other tools that showed the advantages and limitations of the tool.

Future work will include the utilization of multi-threaded database programming and distributed systems to speed up EgyCD. It will also include the deployment of further data mining and non Apriori-based SPM algorithms to further investigate the value of this family of algorithms in clone detection EgyCD.

11. ACKNOWLEDGMENTS

Thanks to. Chanchal K. Roy, for his support and technical comments as well as his encouragement for this work, also thanks for Auni Ku and Ira for their support.

12. REFERENCES

[1] C. K. Roy, J. R. Cordy, R. Koschke, Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. Comparison and Evaluation of Code Clone Detection Techniques, Science of Computer Programming, 74, 470-495, (2009).

[2] B. Baker, On Finding Duplication and Near-Duplication in Large Software Systems, in: Proceedings of the 2nd Working Conference on Reverse Engineering, WCRE 1995, pp. 86-95 (1995).

[3] C. K. Roy and J. R. Cordy, An Empirical Study of Function Clones in Open Source Software Systems, in: Proceedings of the 15th Working Conference on Reverse Engineering, WCRE 2008, pp. 81-90 (2008).

[4] E. Juergens, F. Deissenboeck, B. Hummel and S. Wagner. Do Code Clones Matter? In Proceedings of the 31st International Conference on Software Engineering (ICSE'09), pp. 485–495, Vancouver, Canada, May 2009.

[5] J. H. Johnson. Identifying Redundancy in Source Code Using Fingerprints. In Proceeding of the 1993 Conference of the Centre for Advanced Studies Conference (CASCON' 93), pp. 171–183, Toronto, Canada, October 1993.

[6] B. Baker. On Finding Duplication and Near-Duplication in Large Software Systems. In Proceedings of the Second Working Conference on Reverse Engineering(WCRE'95), pp. 86–95, Toronto, Ontario, Canada, July 1995.

[7] A. Chou, J. Yang, B. Chelf, S. Hallem and D. R. Engler. An Empirical Study of Operating System Errors. In Proceedings of the 18th ACM symposium on Operating systems principles (SOSP'01), pp. 73–88, Banff, Alberta, Canada, October 2001.

[8] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. IEEE Transactions on Software Engineering, 32(3):176–192, 2006.

[9] M. Fowler. Refactoring: Improving the Design of Existing Code. Addison-Wesley, 2000.

[10] S. Bellon, R. Koschke, G. Antoniol, J. Krinke and E. Merlo, Comparison and Evaluation of Clone Detection Tools, Transactions on Software Engineering, 33(9):577-591 (2007).

[11] Toshihiro Kamiya, Shinji Kusumoto, Katsuro Inoue. CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. Transactions on Software Engineering, Vol. 28(7): 654-670, July 2002.

[12] Chanchal Kumar Roy and James R. Cordy, A Survey on Software Clone Detection, September 26, 2007, Technical Report No. 2007-541, School of Computing, Queen's University at Kingston, Ontario, Canada

[13] Raghavan Komondoor and Susan Horwitz. Using Slicing to Identify Duplication in Source Code. In Proceedings of the 8th International Symposium on Static Analysis (SAS'01), Vol. LNCS 2126, pp. 40-56, Paris, France, July 2001.

[14] Jens Krinke. Identifying Similar Code with Program Dependence Graphs. In Proceed- ings of the 8th Working Conference on Reverse Engineering (WCRE'01), pp. 301-309, Stuttgart, Germany, October 2001.

[15] Chao Liu, Chen Chen, Jiawei Han and Philip S. Yu. GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis. In the Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'06), pp. 872-881, Philadelphia, USA, August 2006.

[16] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst., 9(3):319349,1987.

[17] A. Leitlao, Detection of Redundant Code Using R2D2, Software Quality Journal, 12(4):361-382 (2004).

[18] Stefan Bellon, Daniel Simon: Vergleich von Klon-erkennungstechniken, 5th Workshop on Software Reengineering, 2003.

[19] Filip van Rysselberghe, Serge Demeyer: Evaluating Clone Detection Techniques, Proceedings of the International Workshop on Evolution of Large Scale Industrial Applications ELISA 2003.

[20] M.-S. Chen, J. Han, and P. S. Yu. Data mining: an overview from a database perspective. IEEE Trans. On Knowledge And Data Engineering 8, 866-883 (1996).

[21] Q. Zhao, S.S. Bhowmick, Sequential pattern mining: a survey, Technical Report Center for Advanced Information Systems, School of Computer Engineering, Nanyang Technological University, Singapore, (2003).

[22] C. Liu, C. Chen, J. Han and P. Yu, GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis, in: Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2006, pp. 872-881 (2006).

[23] M. Gabel, L. Jiang and Z. Su, Scalable Detection of Semantic Clones, in: Proceedings of the 30th International Conference on Software Engineering, ICSE 2008, pp. 321-330 (2008).

[24] R. Komondoor and S. Horwitz, Using Slicing to Identify Duplication in Source Code, in: Proceedings of the 8th International Symposium on Static Analysis, SAS 2001, pp. 40-56 (2001).

[25] What is Plagiarism? Available online: http://www.plagiarism.org/plagiarism-101/what-is-plagiarism.

[26] Hamid Abdul Basit, Member, IEEE, and Stan Jarzabek, A Data Mining Approach for Detecting Higher-level Clones in Software.

[27] Vera Wahler, Dietmar Seipel, J¨urgen Wolff v. Gudenberg, and Gregor Fischer, Clone Detection in Source Code by Frequent Itemset Techniques, University of W¨urzburg, Institute for Computer Science Am Hubland, D . 97074 W¨urzburg, Germany.

[28] C. K. Roy and J. R. Cordy. "NiCad: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization", Proc. ICPC, 2008, pp. 172-181.

[29] M. S. Uddin, C. K. Roy K. A. Schneider, A. Hindle, "On the Effectiveness of Simhash for Detecting Near-Miss Clones in Large Scale Software Systems", 10.1109/WCRE.2011.12 P: 13 - 22