ISSN:0975-9646

# Detection of Clones in Sparse and Dense Data Sets Using Efficient Data Mining Techniques : A Comparative Study

Puli Manjeera[#], Panuganti.Ravi[*]

[#]*Department of CSE, Chaitanya Engineering College, Visakhapatnam.*
**Senior Assistant Professor, Department of CSE, Chaitanya Engineering College, Visakhapatnam.*

**Abstract-- Code clones are similar program structures recurring in variant forms in software system(s). Several techniques have been proposed to discover simple clones i.e., method level clones. But identifying structural clones has been a difficult task because this requires am iterative scan of the database. Structural clones show a bigger picture of simple clones. Hence identification of the structural level clones improves the performance of the system under development by enhancing the properties like reusability, maintainability and re-engineering. So to identify the structural clones a number of approaches have been developed but the efficiency of those algorithms are less. Hence in this paper we would like to propose two different techniques one for association mining and one for clustering to identify the structural clones. The proposed technique would not only scan sparse data but also dense data to identify the clones. We also try to detect exact and near miss clones. The techniques used would be mining frequent patterns using prefix trees and an efficient density based clustering algorithm. At last we make a comparison between the existing method and the one proposed in this paper.**

**Keywords--structural Clones, FP-Tree, Re-use, Maintenance.**

## I. INTRODUCTION

Code clones are similar program structures of considerable size and significant similarity. In large scale systems many projects are developed in parallel and the code related to those projects are stored in a centralized system which consists of duplicated data at different levels. The information obtained from previous sources shows that 30 to 40% [2][3][4] of the code obtained from large scale systems consists of clones. Hence identifying these clones would reduce rework and would be useful for re-engineering and maintenance. Code duplication is easy but it makes software maintenance more complicated.

Simple clone detectors usually detect clones larger than a certain threshold (e.g., clones longer than 5 LOC). Higher thresholds risk false negatives, while lower thresholds detect too many false positives. In comparison, Clone Miner can afford to have a lower threshold for simple clones, than a stand-alone simple clone detector, without returning too many false positives. This is because it can use the grouping as a secondary filter criterion, to filter out small clones that do not contribute to structural clones. These small simple clones may just be noise when considered individually, but when they are combined to form structural clones, they can indicate bigger cloned entities.

Previous clone detection [1] work was only limited to textual matches or near misses only on complete function bodies. Whereas this paper presents some practical methods for detecting exact and near miss clones for arbitrary fragments of program source code. And also the current clone detection approaches are not scalable to very large codes. Hence they cannot be used for real-time detection in large systems, thereby reducing their usefulness for clone management.

In this paper we concentrate on seven different levels of clones out of which some levels can be done manually [5][6] and some levels need our approach. The levels are as follows:

Level-1: Repeating groups of simple clones
a) In the same method
b) In different methods
Level-2: Repeating groups of simple clones
a) In the same file
b) Across different files
Level-3: Method clone sets
Level-4: Repeating groups of method clones
a) In the same file
b) Across different files
Level-5: File clone sets
Level-7: Repeating groups of file clones
a) In the same directory
b) Across different directories
Level-7: Directory clone sets.

This can be done by using density based Clustering and frequent item set mining without candidate generation with the help of FP-tree algorithm [8]. The proposed algorithms and their performance results are given in the coming sections. We apply frequent item set mining to levels 1b, 2b, 3, and 6b. Similarly we apply clustering to levels 3, 5, 7. We try to detect the clones not by their line numbers as was done in the previous paper. Rather we take a different approach so that clones present at different line numbers within different methods or files are detected. We try to make this approach applicable to any of the languages like C, C++, Java, etc.,. At last we make a comparison between the previous method and our method with the help of a case study.

The remaining section is organized as follows. In section 2, we give a procedure to identify the simple clones. In section 3 we organize the data for further analysis. In section 4, we briefly review about the improved FP-tree method and discuss about the algorithm and its usefulness in clone detection. In section 5, we introduce effective density based clustering technique and its algorithm to search for clones at the directory level. Section 5, is dedicated to the results obtained by applying our technique to a particular project and making a comparative study. And at last reference papers that have helped in guiding this paper are listed.

## II. Detection Of Simple Clones Using Lexical Analysis

We Consider a clone detector which is based on parsing or lexical analysis. The information can be obtained directly, otherwise we can deploy program analysis to obtain this information. This tool uses Repeated Tokens Finder (RTF), a token-based simple clone detector, as the default front-end tool [6]. RTF tokenizes the input source code into a token string, from which a suffix array based string-matching algorithm directly computes the SCSets, instead of computing them from the clone pairs. RTF currently supports Java, C++, Perl, and VB.net. RTF also performs some simple parsing to detect method and function-boundaries.

## III. Re-Organizing The Data Obtained

Once the data about the simple clones is obtained they need to be organized in a way such that the data can be used for further analysis like mining and clustering to find out structural clones. We list simple clones for each method or file, depending on the analysis level. We need to check the function boundaries while performing the analysis. With this arrangement of simple clones, we get a different view of the simple clones' data, with simple clones arranged in terms of methods or files. A sample of this format is shown in Fig. 1. The first data row means that the file No. 10 contains three instances of SCSet 9 and one instance each of SCSets 15, 28, 38, and 40. The interpretation is likewise for the other rows. At this stage, we can easily filter out methods or files that do not participate in cloning at all (i.e., contain no simple clones).

| File identifier | Simple clone set instances |
|---|---|
| ……. | ……. |
| 10 | a, b, c, e, f, o |
| 11 | a, c, g |
| 14 | a, c, d, e, g |
| ……. | ……. |

Fig.1. Clones per file

From the above data we detect the groups of simple clones in different files or different methods. With this we have the data ready for further analysis.

## IV. Detecting Repeating Groups Of Simple Clones

To detect recurring groups of simple clones in different files or methods, we apply the same data mining technique that is used for "market basket analysis" [7]. The idea behind this analysis is to find the items that are usually purchased together by different customers from a departmental store. The input database consists of a list of transactions, each one containing items bought by a customer in that transaction. The output consists of groups of items that are most likely to be bought together. The analogy here is that a file or a method corresponds to a transaction and the SCSets, represented in that file or method, correspond to the items of that transaction. Our objective is to find all those groups of SCSets whose instances occur together in different files or methods.

In our data, one file or method may contain multiple instances of the same SCSet. We could normalize the data by removing the duplicates, but by doing so, we would miss out important information - where multiple instances of an SCSet are part of a valid structural clone across files or methods. For example, we have three instances of SCSet 9 present in both files 12 and 14 shown in Fig. 1, so 9-9-9-15 is a valid level 2-B structural clone across these two files. But if the data is normalized by removing duplicates, then we would not get this complete structural clone.

Hence we would perform frequent closed item-set mining where only those subsets are reported which are not subsets of any bigger frequent itemset. The technique proposed by us is an improved FP-Tree algorithm known as Prefix Tree. Compared with Apriori [1] and its variants which need several database scans, the FP-growth method only needs two database scans when mining all frequent itemsets. The first scan counts the number of occurrences of each item. The second scan constructs the initial FP-tree which contains all frequency information of the original dataset. Mining the database then becomes mining the FP-tree.

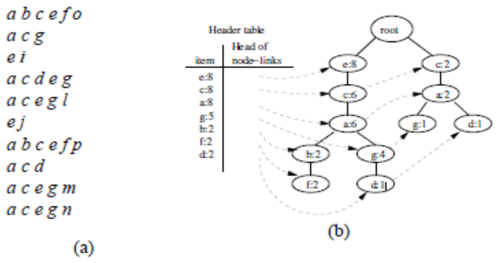

Fig.2. An example FP-Tree

But the FP-Tree for a sparse data set will be big and bushy, due to the fact that they do not have many shared common prefixes. However for dense data sets they are more compact. The pseudo code for our new method is shown below in fig.3

```
Procedure FPgrowth*($T$)
Input:    A conditional FP-tree $T$
Output:  The complete set of FI's
         corresponding to $T$.
Method:
1. if $T$ only contains a single path $P$
2. then for each subpath $Y$ of $P$
3.     output pattern $Y \cup T.base$ with
       count = smallest count of nodes
       in $Y$
4. else for each $i$ in $T.header$
5.     output $Y = T.base \cup \{i\}$ with $i.count$
6.     if $T.array$ is not NULL
7.         construct a new header table
           for $Y$'s FP-tree from $T.array$
8.     else construct a new header table
           from $T$;
9.         construct $Y$'s conditional
           FP-tree $T_Y$ and its array $A_Y$;
10.    if $T_Y \neq \emptyset$
11.        call FPgrowth*($T_Y$);
```

Fig.3. Algorithm FPgrowth*

Once this method is performed the data is once again organized in such a way that it is suitable for further analysis.

## V. Detecting File And Method Clones

Higher level clones cannot be identified using the above approach. Hence we perform an efficient clustering algorithm on the above obtained data and detect directory level and file level clones. Clustering is a process of grouping the data objects into classes so that data objects within a class are highly similar to one another but dissimilar to data objects in other class based on attribute values describing these data objects.

To discover clusters with arbitrary shape, we use density-based clustering method. We have used the DBScan method for this purpose. DBSCAN (Density-Based Spatial Clustering of Applications with Noise) is a density based clustering algorithm. The algorithm grows regions with sufficiently high density into clusters and discovers clusters of arbitrary shape in spatial databases with noise.

It defines a cluster as a maximal set of *density-connected* points. Consider Fig.4 for a given e represented by the radius of the circles, and, say, let *MinPts* = 3. DBSCAN searches for clusters by checking the e-neighborhood of each point in the database. If the e-neighborhood of a point ***p*** contains more than *MinPts*, a new cluster with ***p*** as a core object is created. DBSCAN then iteratively collects directly density-reachable objects from these core objects, which may involve the merge of a few density-reachable clusters. The process terminates when no new point can be added to any cluster.

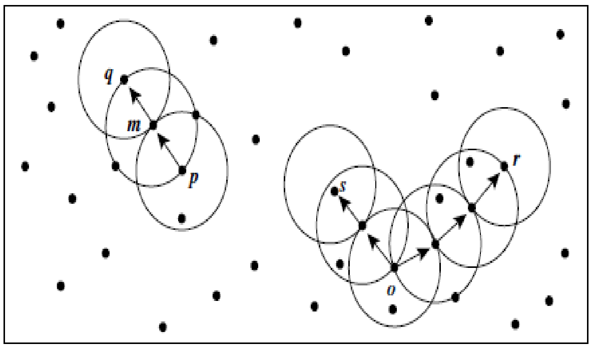

Fig.4. Density reach ability and density connectivity in density-based clustering.

So based on this methodology both the above mentioned methods are applied iteratively one after the other until all the clones are detected.

## VI. Implementation Of The Tool

We have successfully applied the methodology on a Buffer and Buffer class systems as shown below with the details.

TABLE-I

| | **BUFFER** | **BUFFER CLASS** |
|---|---|---|
| No. of files | 5,310 | 2,010 |
| LOC | 2,34,567 | 1,23,787 |
| No. of directories | 209 | 336 |
| No. of methods | --- | 8,581 |

The results obtained in the previous system and that of our system is shown below.

TABLE-II (Previous System)

| **MINCOVER** | **SUPPORT-50%** | **SUPPORT-90%** |
|---|---|---|
| No. of groups | 283 | 186 |
| No. of Mc sets covered | 153 | 107 |
| % of MC sets covered | 51.5% | 47% |
| Methods covered by groups | 1,417 | 1,075 |
| % of Methods covered by groups | 17% | 13% |
| Min. no. of methods in a group | 2 | 2 |
| Max. no. of methods in a group | 192 | 180 |
| Avg. no. of methods in a group | 39 | 46 |

TABLE-III (Proposed System)

| **MINCOVER** | **SUPPORT-50%** | **SUPPORT-90%** |
|---|---|---|
| No. of groups | 301 | 234 |
| No. of Mc sets covered | 180 | 123 |
| % of MC sets covered | 59.8% | 52.56% |
| Methods covered by groups | 1,501 | 1,211 |
| % of Methods covered by groups | 19% | 15% |
| Min. no. of methods in a group | 2 | 2 |
| Max. no. of methods in a group | 201 | 185 |
| Avg. no. of methods in a group | 45 | 51 |

## VII. Conclusion

In this paper we emphasized on higher level cloning**.** The process is started by finding simple clones (that is, similar code fragments). Increasingly higher-level similarities are then found incrementally using data mining techniques of finding frequent closed item sets, and clustering. We believe our technique is both scalable and useful. In this paper we have tried to improve the efficiency of the system by using efficient data mining techniques. Implementing good visualizations for higher-level similarities is also an important part of our work. We have tried to identify the clones based on their addresses and not on their physical locations and we have tries to cater this technique to various languages.

## Acknowledgements

We would wish to thank our Head of the Department and the other faculty members who have helped us for this paper. I would also like to thank the selection committee who has reviewed this paper.

## REFERENCES

[1] Kozaczynski, W., Ning, J. and Engberts, A. Program concept recognition and transformation. IEEE Transactions on Software Engineering, 18(12):1,065-1,075, December 1992.

[2] Baker, B. S. On finding duplication and near-duplication in large software systems. In Proceedings of the 2nd Working Conference on Reverse Engineering (WCRE), pages 86-95, 1995.

[3] Ducasse, S, Rieger, M., and Demeyer, S. A language independent approach for detecting duplicated code. In Proceedings of the International Conference on Software Maintenance (ICSM), pages 109-118, 1999.

[4] Mayrand J., Leblanc C., Merlo E. Experiment on the automatic detection of function clones in a software system using metrics. In Proceedings of the International Conference on Software Maintenance (ICSM), pages 244-254, 1996.

[5] Krine, J. Identifying similar code with program dependence graphs. In Proceedings of the Eight Working Conference on Reverse Engineering (WCRE), pages 301-309. Stuttgart, Germany, October 2001.

[6] Koschke, R., Falke, R., and Frenzel, P. Clone Detection Using Abstract Syntax Suffix Trees. In Proceedings of the 13th Working Conference on Reverse Engineering (WCRE), pages 253-262, 2006.

[7] Kamiya, T., Kusumoto, S, and Inoue, K. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. IEEE Transactions on Software Engineering, 28(7):654 – 670, July 2002.

[8] Rieger, M. Effective Clone Detection without Language Barriers. Ph.D. Thesis, University of Bern, 2005.

[9] Basit, H. A., Puglisi, S., Smyth, W., Turpin, A., and Jarzabek, S. Efficient token based clone detection with flexible tokenization. In Proceedings of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE), pages 513-516, September 2007.

[10] Han, J., and Kamber, M., Data Mining: Concepts and Techniques, Morgan Kaufman Publishers, 2001.