Literature Survey of Clone Detection Techniques

Sonam Gupta  
Research Scholar,  
Suresh Gyan Vihar University  
Jaipur (Rajasthan), India

P. C Gupta, Ph.D  
Department of Computer Science & Engineering  
University of Kota  
Kota (Rajasthan), India

ABSTRACT
Code clones are the codes which have same code in the system and so it is difficult to locate all the same codes in the system when any change is to be done. Researchers have proved that almost 70% of the effort done during maintenance is just because of the occurrence the clones in the system. A number of approaches had been given earlier to detect various types of clones [39]. This paper presents the systematic literature review of all the detection approaches researched so far. Along with it this paper also gives the advantages to implement them and also all the defects due to which they were not able to completely detect the clones. It also gives a novel approach to automatically detect the clones irrespective of the matter that whether the code is in same order or any statement has been inserted, deleted or modified in the code fragment.

Keywords
Clones maintenance, Program dependence graph, tree-based approach, false positives, and hybrid approach

1. INTRODUCTION
Maintenance effort of the system increases with the increase in complexity of the system. There are four types of maintenance namely corrective maintenance, adaptive maintenance, perfective maintenance and preventive maintenance [36]. Corrective maintenance is the reactive modification for a software product performed after delivery to correct discovered problems. Adaptive maintenance is the modification of a software product performed after delivery, to keep a software product usable in a changed or changing environment. Perfective maintenance is the modification of a software product after delivery to improve performance or maintainability. Preventive maintenance is the modification of a software product after delivery to detect and correct latent faults in the software product before they become effective faults. Arthur states that only one-fourth or one-third of all life-cycle costs are attributed to software development and that some 67% of life-cycle costs are expended in the operation-maintenance phase of the life cycle [35,37]. The major challenge during maintenance is the difficulty to trace the product or the process that created the product, changes are not adequately documented, lack of change stability and ripple effect when making changes. There are many approaches that had given by many researchers to detect such types of changes primarily known as clones. In this paper we have presented a systematic literature survey for the detection of clones. The rest of the paper is organized as follows.

Section II consists of the literature survey of past 10 years in the field of clone detection along with it the section also gave a novel approach to overcome the flaws of previous approaches. Section III consists of conclusion and future work and section IV consists of all the references from where the survey has been done.

2. LITERATURE SURVEY
This section includes the literature survey of past 10 years in the clone detection techniques. This will help to find the various merits and demerits of various approaches developed so far so that a new and better approach of clone detection can be developed.

Text-Based Approach: The code fragments are considered as sequence of text and are compared with each other as the basis of various transformations like removing comments, whitespace, newlines etc. There are various researchers who found various techniques to detect clones on the basis of text. Baker [2,3] used line-based string matching algorithm by making the tokens of each sequence/line of the text by help of a tool named as Dup. This technique consists of all the basic properties of text-based technique, along with this it also replaced identifiers, variables and types with a special parameter so that even if the name of variable is different the clone can be identified. But this tool was not able to support exploration and navigation between the duplicated codes and moreover it cannot detect clones if the code is written in different style. This problem was overcome by Koshke et al.by [23] by finding the clones on the basis of tokens rather than lines but this was not able to keep track that whether the identifiers had been renamed consistently or not after transformation. Johnson used [19] used Karp-Rabin fingerprinting algorithm to detect clones on purely text basis. In this technique each character is included in atleast one substring and then the matching of those substrings is done. The disadvantage with this technique is that it had the restriction of keeping 50 lines match resulting in more number of false positives. Cordy et al. [9] used this text-based approach to detect the near miss clones for HTML web pages. In this technique firstly syntactic constructs are identified and then used as smallest comparison but this did not normalize any code. Ducasse et al. [12,13] presented an approach using dynamic pattern matching which is language independent but this was not able to identify meaningful clones because the cohesiveness of the code gets effected. Marcus[28] gave a novel approach which used latent semantic indexing [14] and does not compare the whole code rather than that it identify clones by creating certain domains of comments and identifier matching. But this cannot detect clones if the structure is same but the name of identifiers is different. All the above detected techniques clearly show that although the cost of applying the approach is very less but the code having line break, change in variable name, type, change of parenthesis etc. cannot be identified and tested that whether it is a cloned code or not.

Token-based approach: This technique parse the whole source code into sequence of tokens thereby overcoming the problems faced in text-based approach like change in space, identifier name etc. Kamuya et al.[6]developed a tool named as CCFinder in which each line is divided into tokens and then added together to form a single token, so that even if the
name of identifier etc. is changed then it will not effect the
detection of clone even if the structure of the code is same but
there are some minor avoidable changes in the code. Even
even though Baker [2,4] also used the token scheme to detect the
code but it did not use any transformation technique resulting
in detection of false positives. For more flexible tokenization
RTH [7] used suffix array rather than suffix tree so that
unnecessary tokens can be removed so as to reduce the false
detection but this technique is more complex to implement.
To overcome the problem of CCFinder and Dup i.e. cannot
detect clones if there are minor changes in the code, CPMiner
[26,27] was introduced which can ignore insertion/deletion or
any modifications of code upto 1-2 statements only. Juergens
et. al [40] gave a plug-in in visual studio which can detect the
clones in Java and C# but the approach was not able to handle
the defects at programmer side itself. Almost same approach
was given by Kawaguchi et. al [41] but it was developed for
C++ and C# but it did not overcome the problem as in [40].

Tree-based approach: Rather than creating tokens for each
statement this technique creates subtrees of a fragment of the
code and the code is said to be a cloned code if the
the corresponding sub trees match.. this is done by creating the
AST of the code. CloneDR [8] is a well known tool which
uses this technique. It generates the parse tree and then by the
help of hash functions the subtrees are matched. But this
technique was not able to identify similar clones. This
problem was overcome by CCdiml [31] tool given by Bauhaus
but this was not able to identify renamed identifiers. Yang [34]
also proposed an approach based on the same technique which
finds the syntactic difference between the versions of the
system and creates their parse tree. Nahler et. al [33] gave the
approach which convert the AST into XML and then by using
data mining technique[1] it extract the clones. This approach
was further refined by Evas & Fraser [15] to find near miss
clones by using only AST leaves rather than the tree, but
again it was not able to detect much of the exact clones. Duala
Ekoko et. al [38] hd developed a tool named Clone Tracker in
Java but again the number of false positives are much more in
it and it is not able to detect post programming. Hoan Anh
[43] developed a clone management tool in Java but it
increases the time to find the clones. All the above mentioned
researches clearly show that AST is not able to find the
modified clones along with it the cost of search space also
increases. It does not follow the data flow and also cannot
detect the clones is the statements are reordered. All the
drawbacks of AST can be easily overcome by the use of
PDG-based technique.

PDG-based technique: Program Dependency Graph (PDG)
[20,24,25] overcome the problems faced in AST and also
maintain the data and control flow [80] and therefore it
become easier to clones semantically as well as syntactically.
Komondoor and Horwitz [20,21] gave an approach known as
PDG-DUP which used program slicing method to identify the
code clones without changing its semantics. Gallagher and
Lucas [16] extended the work of Komondoor et. al by
applying program slices on all the variables of a code but
could not come to any conclusion. PDG technique was used
as an iterative approach by Krinke[24] for finding the
maximal similar subgraph but it was not able to give a
formula that can be used on any type of system to find the
code. All the researchers are using the PDG technique came
to the conclusion that although PDG-based techniques can
find non-contiguous clones but it cannot be applied to large

Metric-based technique: This technique does not compare
the code directly instead it calculated different metric (like
number of source lines, number of functions etc) for the code
and then those metrics are compared. Mayrand et. al [29] used
this technique and calculated the metrics from names, layouts,
expression and control flow but it was not able to identify
segment based copy-paste operation. Kontogiannis et. al [22]
used markov model but it can only measure the similarity
between the codes rather than finding the exact clones. They
modified the approach by calculating the metrics on the basis
of begin-end block and the code is said to be cloned code only
if their metrics is approximately same. Di Ducca et. al [11]
also used this approach to find the clones in static HTML
pages by calculating their degree of similarity. This was done
by calculating the Levenshtein distance of the code [67].

Hybrid approaches: These approaches are the combination of
various approaches discussed above. Koschke et. al [23]
used tree based and token based technique to identify the
clones. From the tree-based technique firstly a suffix tree is
created and then it is compared by using token-based
approach. This is used only to find exact and type-II clones.
Almost similar approach was given in Microsoft’s new
phoenix framework [32] but it can only identify clones with
change in name of identifier not of change in its type.
Greenan[17] gave the same approach using sequence
matching algorithm. Jiang et. al. [18] used the AST in
Euclidean space and then Locating Sensitive Hasing (LSH)
[10] is used to cluster the vectors based on similarity.
Dynamic Pattern Matching technique was proposed by
Balazinska [6] in which the characteristics metrics are
computed for the code and then the clusters are identified
gave an approach based on dynamic change tracking and
resolution in Java language but it failed in detecting the data
flows and examining the clones at the programmer’s level.

3. PROPOSED APPROACH
All the advantages and disadvantages of various approaches
discussed in section II clearly show that although many
techniques but still none is able to find the clones correctly.
So we are proposing a hybrid technique which is able to find
more number of true positives. This approach will find all the
clones in the system irrespective of their place and will show
the same to the programmer so that after or during the
development of the code the programmer itself can identify
the chunks which contain the clone and can decide whether to
remove the clone or it is a good smell. In the proposed work
one or two fragments can be compared. It will give the result
in form of chunks so it will be called as chunk-chuck
extraction algorithm. Firstly, the statements of the code will
be examined serially if the statements are found to be
reordered then the semantic-preserving transformation will be
applied to the code so that reordering of the code does not
affect the procedure of identifying the clone. This approach
will be a bit faster removing the sluggish behavior of many
approaches discussed above and also will handle all types of
clones [39].
4. CONCLUSION AND FUTURE WORK

In this paper we have surveyed papers of past 10 years and found that there many approaches given by various researchers to detect the clones which primarily includes text based, token based, tree based, PDG based, metric based and hybrid technique. Although many algorithms had been developed based on these approaches but still none is able to find the clone with accuracy and efficiency. Some algorithms can detect only a particular type of clone and some are so slow that whole system comes to a bottleneck if large system is to be compared. So we have also proposed an algorithm which will find all types of clones that too with accurate clones and more efficiency. Moreover by using the clone-chunk extraction algorithm the developer can decide whether to remove it or not and then can mention the same in the documentation so that the maintenance team do not waste their time in resolving the issues regarding the clones.

Our future work will be to implement the clone-chunk extraction algorithm and check it on various systems.

5. REFERENCES


[25] 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.


