

ClemanX: Incremental Clone Detection Tool for Evolving Software

Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, Tien N. Nguyen
Iowa State University
{tung,hoan,nampham,jafar,tien}@iastate.edu

Abstract

Recent research results have shown more benefits of the management of code clones, rather than detecting and removing them. However, existing clone management approaches are still unsatisfactory, and either incomplete or inefficient, due to the lack of incremental clone detection tool supports. In this paper, we introduce such an incremental clone detection tool, called ClemanX. Our empirical evaluation on real-world software projects shows that ClemanX is highly efficient, complete, precise, and is capable of working incrementally when the code changes.

1 Introduction and ClemanX

Code clones are exactly matched or similar portions of code that are often created by the copy-and-paste programming practice. To deal with code clones, there have been several approaches including automatic *detection, removal, evolution analysis, tracking* and *management* of clones and their groups [2, 4]. Recent research results have shown the benefits of the management of code clones, rather than detecting and removing them [4, 6]. Even in some situations, developers intentionally maintain the code clones and evolve them into distinct code over time [3, 6].

However, without the support of an incremental clone detection mechanism, when a program changes, current clone management approaches have to either 1) re-perform the detection process on the entire project, or 2) track only previously detected clone groups, or 3) perform the detection only on the changed source files. The first case would result in inefficiency¹. In the two other cases, the tracking might be incomplete because some new groups (such as the groups with members belonging to both changed and unchanged files) will not be detected and tracked².

In this paper, we introduce ClemanX, a novel **incremental** clone detection tool. It is implemented based on our Cle-

man framework [7] with a simplified clone detection and updating mechanism. To detect clones, ClemanX generates the fragments, i.e. the portions of code that are potential clones, along with their characteristic vectors from source files. Each source file is represented as an AST and each of its corresponding subtrees is considered as a fragment if its size (i.e. the number of nodes) is larger than a pre-define threshold. The characteristic vectors are constructed based on our structural feature extraction approach, Exas [8], which is accurate and efficient in capturing features of structure-based representations.

For tree-based structures, Exas is interested in two sources of characteristic features: *paths* of various lengths (vertical pattern) and *ordered sets of nodes* at the same level (horizontal pattern) in a tree. The feature of a path or an ordered set is the sequence of labels of their nodes. The characteristic vector of a fragment is a vector of occurrence counts of such features extracted from its corresponding subtree in the AST. To build fragments from the AST, ClemanX traverses the tree in post-order. For each node, the characteristic vector of the corresponding subtree is the sum of the vectors of all of its children nodes and the occurrences of new features created with its root. In [8], we have proved that the distance of such two characteristic vectors is bounded by the editing distance of two corresponding (sub)trees. Therefore, two fragments having a small vector distance are likely to have a small editing distance, i.e. could be considered as a clone pair.

Those pairs are detected by hashing all the fragments into small subsets, called *buckets*, using locality-sensitive hashing [1] and then by pair-wise comparing of all fragments in each bucket to find the pairs whose vector distance is smaller than a pre-defined threshold. Since a locality-sensitive hashing function has a high probability of hashing two close vectors to a bucket (and a low probability for two distant vectors), each bucket tends to contain almost cloned fragments and to have a small size.

To reduce the chance that a clone pair could not be detected because the corresponding two fragments are hashed to different buckets, ClemanX uses multiple independent hash functions. It hashes each fragment into multiple buck-

¹SimScan, the clone detection tool used in [4], was reported to take approximately 20 minutes working on a 30KLOC project.

²Our experiment on JDK6 shows that this kind of new clones is 6%.

ets. Therefore, if such two cloned fragments are missed by a hash function (i.e. are not hashed to the same bucket), they still have other chances to be hashed to the same buckets by the other hash functions.

The detected clone pairs then form a graph, called a *clone graph*, in which nodes represent fragments and edges represent clone relations. Clone groups are considered as the connected components in the graph and could be detected by a graph-traversal algorithm. Since the fragments contained by cloned fragments might also be clones, ClemanX does not report the groups whose all members are contained by the members of another group.

To perform the incremental clone detection, ClemanX stores the list of source files, the set of fragments, buckets, clone pairs, the clone graph, and all clone groups detected from the last run. Then, when source code changes, ClemanX scans the code base to identify new and deleted files. For modified files, ClemanX considers their previous versions as the deleted files and new versions as the newly added files. Fragmentation is applied to new files to construct the new fragments and their vectors. Fragments of deleted files and their related clone pairs are removed from the buckets and the clone graph. New fragments are hashed into buckets and are compared to other fragments in those buckets to detect new clone pairs. Those clone pairs are then added to the clone graph. Because the clone groups from the last run are stored, ClemanX updates them by removing deleted pairs and inserting newly detected pairs, instead of applying the graph-traversal process as in the first run.

2 Evaluation

To evaluate ClemanX, we conducted experiments with various real-world software systems. Our goal is to evaluate the performance of ClemanX, measured by efficiency, precision and completeness, for both standalone and incremental detection modes. All experiments were run on a computer with Windows XP on Intel Core2Duo 2.2 GHz processor and 3GB of RAM. ClemanX is configured to use Exas features with the maximum size of 4 nodes and the fragments with the minimum size of 50 nodes.

Efficiency is indicated by running time. Each experiment was run only once to avoid the effects from disk caching. Experiments show that ClemanX runs highly fast. For example, it processes JDK 5, a 3,200KLOC project, in less than 4 minutes while the state-of-the-art Deckard, reported in [5], takes 14 minutes on JDK 4.2, a 2,400KLOC project.

In general, precision and completeness are calculated as the percentage of the correctly detected clone groups in the total detected groups and the total existing groups, respectively. A group is considered to be correctly detected if all of its members are clones by human criteria. However, it is impractical to verify all existing clone groups, especially

for large-scale projects. Therefore, completeness of clone detection tools are often indicated by cloned lines of code, i.e. the total number of non-overlapping lines of detected clones, and precision is estimated by the manual examination of 100 random groups [5]. The experiment results show that the number of cloned lines detected by ClemanX is approximately equal to those detected by two popular tools SimScan and CCFinder. In JDK 5 project, precision of ClemanX is about 88% for the similarity threshold of 0.9 and is 96% for that of 0.95.

To evaluate ClemanX's ability of incremental clone detection, we run ClemanX on multiple consecutive revisions of the experiment projects. Comparing the detection results of the standalone and incremental runs for the later revisions shows that the detected clones are mostly the same, i.e. incremental runs maintain good precision and completeness as standalone runs. However, the incremental runs are always faster, especially for the revisions with small changes. This is reasonable since in the incremental runs, ClemanX does not need to do fragmentation, feature extraction, hashing, pairing, and grouping for the unchanged fragments.

3 Conclusion

This paper introduces ClemanX, a novel *incremental* code clone detection tool. The experiments on real-world systems show that it is able to efficiently and incrementally detect clone groups with a high degree of precision and completeness.

Acknowledgment. This project was funded in part by a grant from the Vietnam Education Foundation (VEF) for the first author.

References

- [1] A. Andoni and P. Indyk. E2LSH 0.1 User manual. <http://web.mit.edu/andoni/www/LSH/manual.pdf>.
- [2] R. Tairas. Bibliography of code detection literature. <http://www.cis.uab.edu/tairasr/clones/literature/>.
- [3] L. Aversano, L. Cerulo, and M. D. Penta. How clones are maintained: An empirical study. In *CSMR '07*.
- [4] E. Duala-Ekoko and M. P. Robillard. Tracking code clones in evolving software. In *ICSE '07*.
- [5] L. Jiang, G. Mishnerghi, Z. Su, S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *ICSE '07*.
- [6] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. *SIGSOFT Software Engineering Notes*, 30(5):187-196, 2005.
- [7] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Cleman: Comprehensive clone group evolution management. In *ASE '08*.
- [8] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Accurate and Efficient Structural Characteristic Feature Extraction for Clone Detection. In *FASE '09*, pages 440-455. Springer-Verlag, 2009.