Scalable and Incremental Clone Detection for Evolving Software

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

Abstract

Code clone management has been shown to have several benefits for software developers. When source code evolves, clone management requires a mechanism to efficiently and incrementally detect code clones in the new revision. This paper introduces an incremental clone detection tool, called ClemanX. *Our tool represents code fragments as subtrees of Abstract Syntax Trees (ASTs), measures their similarity levels based on their characteristic vectors of structural features, and solves the task of incrementally detecting similar code as an incremental distance-based clustering problem. Our empirical evaluation on large-scale software projects shows the usefulness and good performance of ClemanX.*

1. Introduction

The copy-and-paste programming practice often creates exactly matched or similar portions of code, which are called *code clones*. To deal with code clones, there have been several approaches including *detection*, *removal*, *evolution analysis*, *tracking*, and *management* of clones and their groups [4]. Recent research results have shown the benefits of the management of code clones [2]. However, without the support of an incremental clone detection mechanism, when a program changes, current code clone management approaches have to either 1) re-perform the detection process on the entire project, 2) track only previously detected clone groups, or 3) perform the detection only on the changed source files.

The first case would result in inefficiency, since a small change still requires the entire re-detection process. In two other strategies, the tracking of clones might lead to the incompleteness. For example, a developer copied a portion of code from a file and pasted to another file. The first file was not changed, but the second was. Therefore, if clone detection is not re-performed or is performed only on the changed files, this pair of cloned code, which we call a *cross-revision* clone pair, could not be detected and tracked.

Therefore, it calls for an efficient approach for *incremental clone detection* which is as complete and precise as re-detection. In this paper, we introduce such a tool, called **ClemanX**. ClemanX detects clones in three following steps: 1) Generate the code fragments of interest, 2) Detect clone pairs, i.e. pairs of cloned fragments, and 3) Form those pairs into clone groups. When the code changes, ClemanX updates the fragments, the clone pairs, and clone groups in accordance to the changes. Let us describe those steps in details.

2. Code Representation and Similarity

ClemanX represents each source file as an AST and each of its subtrees is considered as a code fragment if it is sufficiently large. The similarity levels of fragments are measured indirectly by the distances of their characteristic vectors. ClemanX constructs characteristic vectors of the fragments using Exas, a structural feature extraction method [3]. For tree-based representation, Exas is interested in characteristic features extracted from *vertical paths* (vertical features) and *sequences of sibling nodes* (horizontal features) in the AST. For such a path or a sequence, its extracted feature is the sequence of labels of its nodes. Then, the characteristic vector of a fragment is a vector of occurrences of such features extracted from its tree-based representation.

Figure 1 shows an illustrated AST representing the statement `if (a > b) a = b;` and some of its vertical (in dashed rectangles) and horizontal features (in dotted rect-

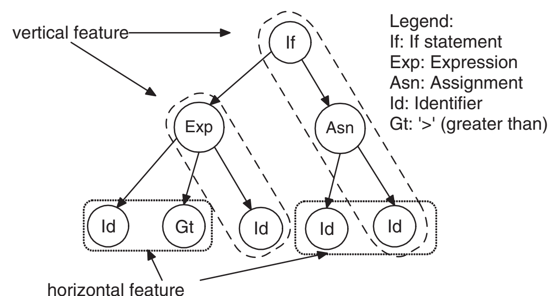

Figure 1. Example of Exas Features

978-1-4244-4828-9/09/$25.00 2009 IEEE Proc. ICSM 2009, Edmonton, Canada

Feature	Index	Count	Feature	Index	Count
If	1	1	Asn-Id	10	2
Exp	2	1	If-Exp-Id	11	2
Asn	3	1	If-Exp-Gt	12	1
Id	4	4	If-Asn-Id	13	2
Gt	5	1	Exp-Asn	14	1
If-Exp	6	1	Id-Gt	15	1
If-Asn	7	1	Gt-Id	16	1
Exp-Id	8	2	Id-Gt-Id	17	1
Exp-Gt	9	1	Id-Id	18	1

Table 1. Features with Indexes and Counts

angles). Table 1 lists all of its features, with corresponding indexes (i.e. the position of each feature in a vector) and occurrence counts. We had showed that the vector distance of two fragments is bounded by their edit distance, i.e. similar fragments (having small edit distance) will have small vector distance [3]. Thus, vector distance could be used as a code similarity measurement. Details are in [3].

3. Incremental Clone Detection

3.1 Important Concepts

A **fragment** is a subtree of an AST tree parsed from a source file whose size (i.e. the number of nodes of the subtree) is larger than a user-defined threshold. Each fragment has a characteristic vector. A pair of two fragments is a **clone pair** if their vector distance does not exceed a user-defined threshold. Such fragments are called cloned fragments or **clones**. All the clones and clone pairs of a program could be represented as a clone graph. **Clone graph** is an undirected graph of which each node represents a cloned fragment and each edge represents a clone pair. Since a fragment might be cloned several times, ClemanX captures the related clones in a group with the concept of **clone group**. A clone group is the set of fragments corresponding to a connected component of the clone graph.

Incremental clone detection is defined as two tasks: initial detection and incremental detection. **Initial detection** is to detect the clones in the first run and could be considered as the classic clone detection problem (i.e. in standalone mode). This task is formally defined as follows: given a set of fragments F, build the clone graph and find the clone groups via its connected components. **Incremental detection** is to update the current clone pairs and groups in accordance with code changes. It is formally defined as follows: given two sets of added (F_+) and deleted fragments (F_-), update the clone graph and its clone groups correspondingly to those changes.

Since each fragment is represented as a vector (i.e. a point) and a clone pair is two close vectors (i.e. having a small distance), each group can be considered as a cluster of close vectors (i.e. similar fragments). This incremental clone detection problem is in fact an **incremental distance-based clustering problem**.

3.2 Locality-Sensitive Hashing

To find the clone pairs, one could perform pairwise comparison on all fragments in F. However, because F usually has a large number of fragments, pairwise comparison is not efficient. Thus, we should compare only fragments that are likely to be clones, i.e having similar vectors. To find those fragments, ClemanX uses locality-sensitive hash functions (LSHs) [1].

An LSH function is a hash function for vectors such that the probability that two vectors having a same hash code is a *strictly decreasing* function of their corresponding *distance*. In other words, vectors having smaller distance will have higher probability to have the same hash code, and vice versa. Then, if we use LSH functions to hash the fragments into buckets based on the hash codes of their vectors, the fragments having similar vectors tend to be hashed into the same buckets, and the other ones are less likely to be so. The more independent hash functions are used, the higher the probability of such hashing occurs.

3.3 Initial Detection Algorithm

The pseudo code for our initial detection algorithm is presented in Figure 2. B and G are two maps to store the buckets and the clone graph. $B[i]$ denotes the bucket corresponding to the hashcode i and $G(u)$ denotes the clone group that contains fragment u. (The clone graph is stored as adjacent lists: $G(u)$ contains the adjacent nodes of u, i.e. the clones of u, and u itself).

Firstly, each fragment u in F is hashed into N buckets indexed by its hash codes produced from N independent hash functions (lines 2-4). N is a user-defined parameter. Then, fragments of each bucket are compared pairwise to detect the clone pairs. When a clone pair is detected, it is added to the clone graph and its corresponding group (by function `Add`). Such addition is as follows: if there is no existing group containing either fragment in the pair, a new group is created for the pair. Otherwise, their two groups (of two fragments) are merged into a common group.

The collection of resulting clone groups after all such additions (lines 6-7) is denoted by $C(G)$. Since a cloned fragment could be enclosed in a larger fragment, a clone group could have all of its members being enclosed by the members of another clone group, thus, should not be redundantly reported. Function `Filter` is used to find such redundant groups in $C(G)$ and eliminate them in the final report.

Figure 3 shows an illustrated example for this algorithm. The left part of Figure 3 shows the fragment set F con-

```
1   function InitialDetection
2     for each fragment u ∈ F
3       for each hash function h
4         B[h(u)] ← B[h(u)] ∪ {u}
5     for each bucket B[i]
6       for all fragments u, v ∈ B[i]
7         if ‖u − v‖ ≤ δ then Add(u, v)
8   return Filter(C(G))
9
10  function Add(u, v)
11    if G(u) = ∅ then G(u) ← {u}
12    if G(v) = ∅ then G(v) ← {v}
13    if G(u) ≠ G(v)
14      G(u) ← G(v) ← G(u) ∪ G(v)
```

Figure 2. Initial Detection Algorithm

```
1   function IncrementalDetection
2     for each fragment u ∈ F−
3       G(u) ← G(u)\{u}
4       for each hash function h
5         B[h(u)] ← B[h(u)]\{u}
6     for each fragment u ∈ F+
7       for each hash function h
8         B[h(u)] ← B[h(u)] ∪ {u}
9     for each changed bucket B[i]
10      for all fragments u, v ∈ B[i]
11        if u ∈ F+ and ‖u − v‖ ≤ δ then Add(u, v)
12  return Filter(C(G))
```

Figure 4. Incremental Detection Algorithm

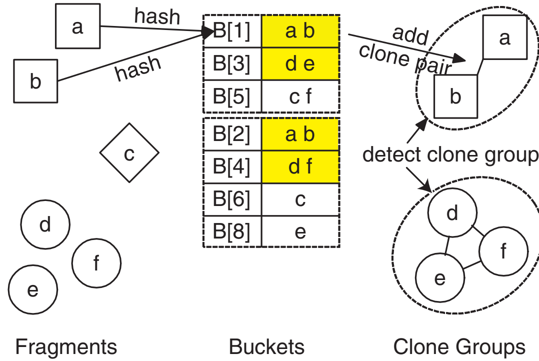

Figure 3. Example of Initial Detection

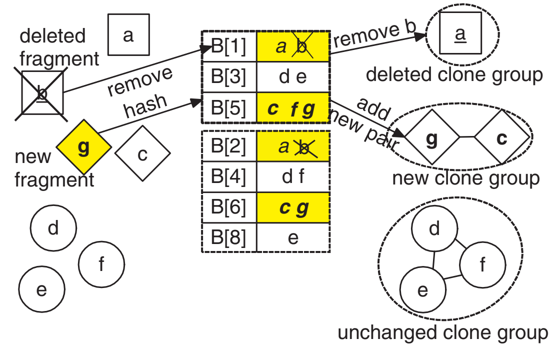

Figure 5. Example of Incremental Detection

taining six fragments from a to f which form two clone groups $\{a, b\}$ and $\{d, e, f\}$. They are hashed by two hash functions $h1$ and $h2$ into the buckets $B[i]$ represented in the middle parts of the figure (Buckets of $h1$ have odd indexes and those of $h2$ have even indexes).

Then, by pairwise comparison of the fragments in each bucket, the clone pairs are detected and added to the clone graph represented in the right part of Figure 3. For example, the clone pair (a, b) is hashed to and detected from buckets $B[1]$ and $B[2]$, (d, e) is from $B[3]$, and (d, f) is from $B[4]$.

When clone pairs are being detected from the buckets and added to the clone graph, the clone groups are also being built. These clone groups form connected components in the clone graph, which will be gradually expanded as more clone pairs are detected. For example, when (a, b) and (d, e) are detected, two groups $\{a, b\}$ and $\{d, e\}$ are formed. Then, when (d, f) is detected, $\{d, e\}$ is expanded into $\{d, e, f\}$. At last, after all possible clone pairs are added, they are filtered and reported.

Note that due to the nature of LSH, non-cloned fragments such as c and f might be hashed to the same bucket ($B[5]$) while cloned fragments such as e and f might not be hashed to the same bucket. Thus, such pair might not be represented as a clone pair in the clone graph. If more hash functions are used, there is higher probability that they are hashed to the same bucket and be detected. In this example, the clone pair (e, f) is still detected in the clone group $\{d, e, f\}$ due to the connectivity of (d, e) and (d, f). This result implies a benefit of considering clone groups as connected components in the clone graph. That is, some cloned fragments might not be detected from the buckets, but are still detected by their clone relation to other fragments.

3.4 Incremental Detection Algorithm

The pseudo code for our incremental detection algorithm is presented in Figure 4. Firstly, each deleted fragment in F_- is removed from its group and buckets (lines 2-5). Secondly, all newly added fragments in F_+ are hashed into buckets as in the initial run (lines 6-8). Then, they are compared to other (existing or newly added) fragments in those buckets to find the new pairs (lines 9-11) and to update the clone graph. After updating, function `Filter` removes singleton groups. Redundant groups are kept for future incremental detection, but are not reported.

Figure 5 continues the illustrated example. Assume that the fragments change: 1) b is deleted, and 2) g, a clone of c, is created. ClemanX performs as follows: b is removed

Revisions	KLOC	Incremental		Standalone		Difference	
GEclipse		Cov	Time	Cov	Time	Pair	Line
1000-1000	259	37%	32	37%	32	0	0
1000-1010	259	37%	3.7	37%	32	0	0
1000-1050	260	37%	1	37%	32	0	0
1000-1100	261	37%	0.9	37%	32	0	0
1000-1500	268	37%	0.3	37%	33	0	0
1000-2000	328	35%	0.5	35%	39	0	0
Columba							
100-100	80	10%	15	10%	15	0	0
100-110	182	15%	3.6	15%	31	0	0
100-150	186	15%	1.1	15%	31	0	0
100-200	193	15%	1.1	15%	32	0	0
jEdit							
3000-3000	170	7%	25	7%	25	0	0
3000-3010	170	7%	2	7%	16	0	0
3000-3050	170	7%	0.6	7%	17	0	0
3000-3100	171	7%	0.7	7%	17	0	0
3000-3500	173	6%	0.3	6%	18	0	0
3000-4000	175	7%	0.3	7%	18	0	0

Table 2. Standalone vs Incremental Detection

from its buckets $B[1]$ and $B[2]$, and from its clone group $\{a, b\}$. Then, g is hashed into the buckets $B[5]$ and $B[6]$.

There are four changed buckets $B[1]$, $B[2]$, $B[5]$, and $B[6]$. However, only $B[5]$ and $B[6]$ have newly created fragments. Pairwise comparison on those buckets adds a new clone pair $\{c, g\}$ to the clone graph, and thus, forms a new clone group. The group used to contain b now becomes a singleton group, thus, is removed by function `Filter`. The new group $\{c, g\}$ and the unchanged clone group $\{d, e, f\}$ are reported. It could be easy to check that, if initial detection algorithm is applied on the fragment set, i.e. the set $\{a, c, d, e, f, g\}$, it will also detect two above groups. In other words, both incremental detection and initial detection will produce the same result.

4. Evaluation

Our experiments were conducted on a computer with the following configuration: Windows XP, AMD Athlon 64x2 Dual Core 5200+ GHz, 3GB RAM, 80GB HDD. The performance is evaluated in three criteria: runtime efficiency, precision, and completeness. However, it is impractical to verify all existing clone groups, especially for large-scale projects. Therefore, completeness of clone detection is often indicated by coverage (**Cov**), i.e. the percentage of non-overlapping cloned lines of code in the entire code base. Precision is estimated by manual examination of 100 random groups. In JDK 5, precision is about 88% for the similarity threshold of 0.9 and is 96% for that of 0.95 with the running time of about 7 minutes.

Table 2 shows ClemanX's performance when running on multiple consecutive revisions of subject systems. The shown running time in the incremental mode is the *average time* for the processing of each revision in the corresponding revision range, while that of standalone is for *only* the processing of the last one. For example, processing time from revision 1,000 to 2,000 of GEclipse is about 500 seconds, thus, on average, each revision is processed in about 0.5 seconds. In contrast, the standalone run for only revision 2,000 takes 39 seconds. The result implies that an incremental run is much faster than a standalone run.

Project	Revisions	New	Changed	Cross-rev. pair
Columba	20-360	557	464	3,842
jEdit	2,000-14,000	13,463	1,538	2,129
GEclipse	1-21,000	4,014	3,129	45,927

Table 3. Tracking Changes of Clone Groups

Beside comparing the coverage values, we also matched the clone pairs and cloned lines between their results. The numbers of different clone pairs **Pair** and cloned lines **Line** are all zeros. It means that ClemanX in the incremental mode has the same result as its standalone mode.

ClemanX can also detect the changes to clone groups and report the newly created and changed groups, i.e. groups having newly added and/or deleted clones. Table 3 shows the total number of such groups of the subject projects. Column **Cross-rev. Pair** shows the number of *cross-revision* clone pairs. The table shows that, for Columba, on average, there are about 11 of such clone pairs in each revision (3842/340), and 3 newly created and changed groups.

5. Conclusions

This paper introduces ClemanX, a novel *incremental* tree-based clone detection tool. Our tool represents source code fragments as AST's subtrees, measures their similarity based on the characteristic vectors of structural features, and solves the task of incrementally detecting similar code as an incremental distance-based clustering problem. The experiments on real-world systems show that it is able to efficiently and incrementally detect useful clone groups.

Acknowledgment. This work was funded in part by a grant from Vietnam Education Foundation 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] E. Duala-Ekoko and M. P. Robillard. Tracking Code Clones in Evolving Software. In *ICSE '07*, pp. 158-167, 2007.

[3] 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*, 2009.

[4] R. Tairas - Bibliography of Code Detection Literature. http://students.cis.uab.edu/tairasr/clones/literature/.