Technical Report

A needle in the stack: efficient clone detection for huge collections of source code

Simone Livieri
Department of Computer Science
Graduate School of Information Science and Technology
Osaka University
Osaka, Japan
simone@ist.osaka-u.ac.jp

Daniel M. German
Department of Computer Science
University of Victoria
Victoria, Canada
dmg@uvic.ca

Katsuro Inoue
Department of Computer Science
Graduate School of Information Science and Technology
Osaka University
Osaka, Japan
inoue@ist.osaka-u.ac.jp

January 17, 2010

A needle in the stack: efficient clone detection for huge collections of source code

Simone Livieri
Department of Computer Science
Graduate School of
Information Science and Technology
Osaka University
Osaka, Japan
simone@ist.osaka-u.ac.jp

Daniel M. German
Department of Computer Science
University of Victoria
Victoria, Canada
dmg@uvic.ca

Katsuro Inoue
Department of Computer Science
Graduate School of
Information Science and Technology
Osaka University
Osaka, Japan
inoue@ist.osaka-u.ac.jp

***Abstract*—One of the important uses of source code clone detection analysis is plagiarism detection, where a file is compared against a known corpus of source code to try to find potential matches. As the availability of Free and Open Source Software (FOSS) continues to increase it has become important to know if specific source code has been created from copies of FOSS software. Version 5.0.2 of Debian GNU/Linux contains approximately 323 millions SLOCs, distributed in approximately 1.45 million files. Current clone detection tools are incapable of dealing with a corpus of this size, and might either take literally months to complete a detection run, or might simply crash due to lack of resources. In this paper we propose a time and space efficient token-based method to detect clones of a source code file against a known corpus of source code. With an empirical study, we demonstrate that our method is capable of finding clones of a file in a corpus of 100,000 files source code files in a few seconds.**

I. The challenges of Massive Clone Detection

One of the most frequently mentioned uses of clone detection is determining whether a given file has been cloned (partially or as a whole) from another one (for example, by copying code from one of the many open source projects into another one [3]). Open source keeps growing (both in number of projects, and each project in the size of its source code). Version 5.0.2 of Debian GNU/Linux contains approximately 323 millions SLOCs, distributed in approximately 1.45 million files. Debian consists of *only* 12,300 different software packages. According to the data provided by FlossMole [4], July 2008, SourceForge hosted 69,492 different projects.

Unfortunately finding clones in a massive collection of files is often a time consuming task. Livieri et al. described a distributed method for clone detection that scanned for clones in the source code of the FreeBSD ports (0.75 million files, approximately 11 Gigabytes of source code): it required 51 hrs, using 80 different computers [7].

In an experiment we performed, `CCFinder` [5] detected code clones in a collection of about 450 thousands C/C++ source code files in 29 days.

If this clone detection is to be run only once, 29 days can be perceived as an acceptable performance. But, what if an organization is interested in running this detection frequently (for example, as a service to find the clones of a given source file within a given reference corpus of source code)?

As Livieri described, this could be done in a distributed manner using ad-hoc programs (as they did), or using distributed frameworks, such as Apache's Hadoop (in a manner similar to the one described in [10]).

In recent years there has been a significant effort to write efficient clone detection algorithms. Examples of these approaches are [8], [9]. Their analyses and optimizations assume that the data can fit in memory, and do not take into account that, at some point, a massive collection of source code will eventually exhaust the computer's memory. Kim and Notkin [6] reflected on the need of tools to perform efficient origin analysis, and suggested that clone detection can be useful, but it was constrained by its comparison of n-to-n files, and its speed.

II. Massive Clone Detection

A. Definitions

Let Σ be a finite *alphabet* and $\sigma \in \Sigma^m$. A string $w \in \Sigma^n$ is an n-gram of σ if it is a substring of σ of length n; that is, if

$$w = \sigma_i, \sigma_{i+1}, \ldots, \sigma_{i+n-1}$$

for some i, $0 \leq i \leq m-n$; let w_i be the n-gram of σ starting at position i.

Let $c = \{c_0, \ldots, c_{h-1}\}$ denote a *cover* of σ, that is an ordered set of n-grams whose concatenation or partial concatenation is σ. A cover is a *minimal cover* of σ if removing one member destroys the covering property.

We define a partial concatenation of two strings, denoted with $\diamond$ as follows. Given two strings x and y, let $x = x'\alpha$ and $y = \beta y'$; define

$$x \diamond y = x'\alpha y'$$

if $\alpha = \beta$, undefined otherwise. For example, $abcde$ is a partial concatenation of abc and cde.

Let $\omega(\sigma) = \{w_0, \ldots, w_{m-n}\}$ be the ordered set of all the n-grams of σ; $\omega^U(\sigma) = \{w_0^U, w_1^U, \ldots, w_{k-1}^U\}$ be the set of the unique n-grams of σ.

Algorithm 1 DETECT(σ, c, δ)

Require: c cover of s
Require: δ distance vector of c
  $\mathcal{K} \leftarrow p^\sigma(c_0)$
  $i \leftarrow 1$
  while $i < |c|$ **and** $\mathcal{K} \neq \emptyset$ **do**
    $\mathcal{K} \leftarrow$ MERGE$(\mathcal{K}, p^\sigma(c_i), \delta_i)$
    $i \leftarrow i + 1$
  end while
  return $\mathcal{K}$

Algorithm 2 MERGE$(\mathcal{K}, p, d)$

  if $\mathcal{K} = \emptyset$ **or** $p = \emptyset$ **then**
    return $\emptyset$
  end if
  $\mathcal{K}' \leftarrow \{\}$
  $j \leftarrow 0$
  for $i = 0$ **to** $|\mathcal{K}| - 1$ **do**
    $t \leftarrow 0$
    while $j < |p|$ **and** $t \leq d$ **do**
      $t \leftarrow distance(p_j, \mathcal{K}_i)$
      if $t = d$ **then**
        append $\mathcal{K}_i$ to $\mathcal{K}'$
        $j \leftarrow j + 1$
      else if $t < d$ **then**
        $j \leftarrow j + 1$
      end if
    end while
  end for
  return $\mathcal{K}'$

Let $p^\sigma(w) = \{i | w_i = w\}$ be the set of positions within σ at which the n-gram w appears in it.

For a cover $c = \{c_0, \ldots, c_{h-1}\}$, let $\delta(c) = \{\delta_0, \ldots, \delta_{h-1}\}$ be its *distance vector*. Each element of δ is the distances between the element with the same index in c and c_0. The distance between two elements is the difference between the position of the corresponding n-gram in σ.

B. An Algorithm for Massive Code Clone Detection

Finding duplicated code between some source file and a large repository of source code is a special case of pattern-matching, where the input can be considered as a set of patterns that need to be exact-matched in the repository.

We first illustrate our algorithm from an abstract and simplified point of view using, as an example, the simple case of finding the positions in which a specific string (s) occurs inside a longer string (σ).

Let σ be the string `sethesetheses` over the alphabet $\Sigma = \{\mathtt{e}, \mathtt{h}, \mathtt{s}, \mathtt{t}\}$; let the size of the n-grams be 4. Then

$$\omega(\sigma) = \{\mathtt{seth}, \mathtt{ethe}, \mathtt{thes}, \mathtt{hese}, \mathtt{eset}, \mathtt{seth}, \mathtt{ethe}, \mathtt{thes}, \mathtt{hese}, \mathtt{eses}\}$$
$$\omega^U(\sigma) = \{\mathtt{seth}, \mathtt{ethe}, \mathtt{thes}, \mathtt{hese}, \mathtt{eset}, \mathtt{eses}\}$$

and

$$p^\sigma(\mathtt{seth}) = \{0, 5\} \quad p^\sigma(\mathtt{ethe}) = \{1, 6\} \quad p^\sigma(\mathtt{thes}) = \{2, 7\}$$
$$p^\sigma(\mathtt{hese}) = \{3, 8\} \quad p^\sigma(\mathtt{eset}) = \{4\} \quad p^\sigma(\mathtt{eses}) = \{9\}$$

Finding the position in which the string $s = \mathtt{heset}$ occurs inside σ is straightforward. We proceed as follows (see Algorithm 1):

1) determine a minimal cover of s and its corresponding distance vector:
$$c = \{\mathtt{hese}, \mathtt{eset}\} \quad \delta(c) = \{0, 1\}$$
2) initialize the clone set with the positions of the first element of c in σ:
$$\mathcal{K} = p^\sigma(c_0) = p^\sigma(\mathtt{hese}) = \{3, 8\}$$
3) merge $\mathcal{K}$ and $p = p^\sigma(c_1) = p^\sigma(\mathtt{eset}) = \{4\}$ (see Algorithm 2). Two elements are merged if their distance is equal to $\delta_1 = 1$. In our example, we proceed to merge $\mathcal{K}_0 = 3$ and $p_0 = 4$ because $\delta_1 = 1 = 4 - 3$. The result $\{3\}$ becomes the new value of K.
This step is repeated until all the elements in c have been processed. The final value of $\mathcal{K}$ contains the locations within σ where s occurs, in this case position 3; if $\mathcal{K}$ is empty, the string does not occur in σ.

C. Complexity analysis of the algorithm

The time complexity of searching for clones of a string s depends primarily of the following factors:

1) the number of tokens in s — let it be n — and the size of its cover; for the sake of simplicity we will assume that the size of the cover is n;
2) the size of the n-grams: $|w|$; the size of the n-gram is the minimal size of a clone that the algorithm can detect;
3) the number of different n-grams in the corpus $\mathcal{F}$: $|\omega^U(\mathcal{F})|$;
4) the time required to retrieve $p^\mathcal{F}(w)$ for a given n-gram w in the database — we will refer to this as $\tau(w)$; and
5) the expected length of $p^\mathcal{F}(w)$ for any string w: $|p^\mathcal{F}(w)|$

If there is a complete clone of s in the database, the core of the algorithm is computed n times, but in practice, the size of K will be reduced after each iteration. For each n-gram w in c, $p^\mathcal{F}(w)$ is retrieved, and the distance between each element of K and $p^\mathcal{F}(w)$ computed in time proportional to $|p^\mathcal{F}(w)|$. The time to compute K — and find the clones of s — is proportional to $n \times \max_{w \in \omega^U(\mathcal{F})}(|p^\mathcal{F}(w)|)$.

If not enough memory is available, $\tau(w)$ will depend primarily on the following factors:

1) the speed of secondary storage;
2) the indexing algorithm; a typical database that uses B+ trees (such as a relational DBMS) can find a record in $O(log(|\omega^U(\mathcal{F})|/k))$ time, where $|\omega^U(\mathcal{F})|$ is the number of keys in the database (the number of different n-grams), and k is the number of keys that can fit in a given index page (the longer the n-gram, the fewer the keys per page); and

TABLE I
NUMBER OF UNIQUE n-GRAMS ACCORDING TO THEIR SIZE ON A SAMPLE OF 10,000 FILES

Size	3	4	8	16	32
Unique n-grams	8,305	31,250	698,245	5,618,612	9,732,138

3) the length of the record to retrieve from secondary storage into main memory $p^{\mathcal{F}}(w)$.

If we consider the size of the n-gram w as a constant, $\tau(w)$ will have a complexity $O(log(|\omega^U(\mathcal{F})| + |p^{\mathcal{F}}(w)|)$, hence the time-complexity of using a database to find the clones in a file of n tokens is

$$O(n|p^{\mathcal{F}}(w)|log(|\omega^U(\mathcal{F})| + |p^{\mathcal{F}}(w)|)) \quad (1)$$

In practice, unless n is too large, the time to retrieve from secondary storage the records of the n n-grams might be significantly larger than the time required to perform the rest of the operations. In order to bound the worst case scenario (that n is too large) we only perform clone detections at the function/method level: we break the input string into functions, and perform the detection on each substring. Thus, the time required to perform this clone detection, using secondary storage, is proportional to $n \times (log(|\omega^U(\mathcal{F})|) + |p^{\mathcal{F}}(w)|)$, that is, the length of the input string to search multiplied by the average time it takes to find and retrieve — if it exists — an n-gram record from the database.

TABLE II
DISTRIBUTION OF THE FREQUENCY OF N-GRAMS FOR DIFFERENT SIZES OF N-GRAMS

Size	**1st Qu.**	**Median**	**Mean**	**3rd Qu.**	**Max**	**% $\leq$ 5**
4	1	4.0	165.8	16.0	254,500	50%
8	1	2.0	20.7	6.0	44,460	69%
16	1	1.0	2.6	2.0	6,897	92%
32	1	1	1.4	1.0	2,713	99%

To understand how much $\omega^U(\mathcal{F})$ varies with the size of the n-grams, we computed the n-grams of a sample of source code files and determined that 16 was a good balance between the number of different n-grams $|\omega^U(\mathcal{F})|$ and the average size of each n-gram record $p^{\mathcal{F}}$.

III. IMPLEMENTATION: YOCCA

YOCCA and its companion tools implement our approach for detecting code clones between a file and a large collection of source code. They have been written in the Java programming language and they provide all the functionalities needed with the exception of the parsing of source files into syntactical token sequences. Parsing is achieved through the use of `CCFinder` in *pre-processing* mode[1]. The current implementation uses PostgreSQL 8.4 as its database back-end.

The core functionalities provided by our tools are:

[1]Running `CCFinder` specifying the option "`-p`".

TABLE III
TIME IT TOOK TO CREATE THE DATABASE OF EACH CORPUS, AND ITS RESULTING SIZE

# files	1k	10k	100k
Creation Time	5 min	1 hr 8 min	11 hr 9 min
Total Size	60 MBytes	648 MBytes	3.7 GBytes

1) *N-grams extraction.* The pre-processed files are read and divided into blocks corresponding to functions' bodies. Using a sliding window, groups of 16 tokens (the n-grams) are extracted from each block and saved in a file together with their positions. Tokens are stored as 8-bits values; tokens of the same type have the same value. Positions are stored as triplets $\langle fileIndex, blockIndex, tokenIndex \rangle$.
2) *Database population.* The file containing the n-grams and their position is sorted by n-gram in order to group n-gram's positions. Each pair (n-gram, n-gram's positions) is stored in a database table; each value is first serialized in a byte array. An index is created on the table using the n-gram as key.
3) *Clone detection.* Given an input file, the tool extracts its n-grams and then proceeds to detect all the occurrences of each token sub-sequence with a user-specified minimum length (see Section II-B).

An advantage of our method is that the corpus's size can be easily incremented or decreased: only the data relative to the files added (or removed) need to be inserted (or deleted) in the database.

Another advantage is the parallel nature of our method. Most of the tasks performed by our tools are independent from each other and could be simultaneously executed.

IV. PRELIMINARY RESULTS

We have performed a preliminary evaluation of `Yocca`. The intention of this evaluation is to determine how efficient in terms of speed and space `Yocca` is. First, we created 4 different corpus of 100, 1,000, 10,000 and 100,000 files each (C/C++). These files were randomly selected from the source code of Debian 5.0.2. These experiments were performed on a dual 2GHz quad-core CPU workstation equipped with 4 gigabytes of memory and running Ubuntu Linux 9.10. The database resided on a dedicated SATA hard drive.

The time required to create the database is reported in table IV. As it can be seen, the time required to create the database grows linearly with the size of the database, and it is slightly below linear.

We then searched for clones in 100 different files within each corpus (we run `Yocca`, `CCFinder`, and `Simian` [1] 100 times against each corpus; in each run we search for clones within a different file[2]). Table IV shows the resulting running

[2]`CCFinder` is capable of searching clones across two different sets of files. If one set is the corpus, and the other the file to process, its functionality is equivalent to `Yocca`; `Simian`, however, does not, and in every run all clones in the union of the corpus and the input file were reported.

TABLE IV
RUNNING TIMES (IN SECONDS) OF YOCCA ON DIFFERENT CORPORA

Corpus Size	1st Qu.	Median	Mean	3rd Qu.	Max
100	0.270	0.375	0.519	0.495	2.880
1,000	0.260	0.370	0.557	0.530	3.490
10,000	0.260	0.540	0.822	0.865	5.560
100,000	0.308	3.700	9.420	9.535	100.510

TABLE V
RUNNING TIMES (IN SECONDS) OF DIFFERENT CODE CLONE DETECTION TOOLS ON THE 10K FILES CORPUS. EACH WAS RUN 100 TIMES.

Tool	1st Qu.	Median	Mean	3rd Qu.	Max
Yocca	0.260	0.540	0.822	0.865	5.560
CCFinder	144.2	144.7	146.0	145.1	272.6
Simian	62.87	63.41	63.57	64.02	67.63

times for Yocca for each of the corpora. As it can be seen, the median run time of Yocca for the corpus of 100,000 files is less than 4 seconds. For the smaller corpora, the times are very similar. We speculate this is due to costs not directly related to the detection (such as opening the database connection, preparing data structures, and the effect of data caching inside Yocca).

Table V shows the running times of each tool (run 100 times) against the 10k files corpus. The median and maximum processing times of Yocca were 0.5 and 5.56 seconds, respectively, while CCFinder took median 145 seconds and maximum of 273 seconds; as expected, all Simian runs show approximately the same time (median 63 seconds). Yocca performed significantly faster.

V. CONCLUSIONS

In this paper, we have described some of the challenges that the current clone detection tools face when performing massive code clone detection on a corpus of several hundreds of thousands of files. These problems range from running out of memory, to taking weeks of run time to complete.

To address this problem we have presented a method for code clone detection based on n-grams that is time efficient when the corpus data must be stored in disk (i.e. the minimum amount of information to performed the clone detection is read from disk).

Yocca is a code clone detection tool for massive collections of source code based on the proposed method and implemented using a relational database where the n-grams of the corpus are stored.

A preliminary study demonstrated that Yocca is capable of running notably faster than other code clone detection tools (the median detection time of clones of a file in a corpus of 10,000 files was 0.5 seconds, compared to 146 seconds it took CCFinder to do an equivalent detection).

We have shown how *persistent memoization* — where intermediate results, whose computation is the most resource intensive, are persisted on disk and retrieved when needed — greatly helps reducing the running time of expensive analyses.

Intellectual property clearance is one emerging application of code clone detection that can greatly benefit from our approach. With the growth of the complexity of software, there are cases in which a software system cannot be completely developed by a single entity and parts of it need to be obtained from offshore software companies. The increasing availability of Free and Open Source Software (FOSS) made it important to know if specific source code has been created from copies of FOSS software [2]. The existence of code clones between a software system and Open Source Software can be used to determine a possible violation or incompatibility of license's terms.

VI. FUTURE WORK

In our future work we intent to extend or evaluation of Yocca. In particular, we want to determine the performance of Yocca with collections of more than one million files. We also want to formally evaluate its accuracy: that it detects at least as many clones as other token-based clone detection tools, and it does not detect more false positives than them.

ACKNOWLEDGMENT

This work is being conducted as a part of the Stage Project, the Development of Next Generation IT Infrastructure, supported by Ministry of Education, Culture, Sports, Science and Technology. Also, this research was supported by Japan Society for the Promotion of Science, Grant-in-Aid for Scientific Research (A) (No.21240002).

REFERENCES

[1] Simian - Similarity Analyser. http://www.redhillconsulting.com.au/products/simian/, 2010. Accessed January 13, 2010.

[2] M. J. Foley. Microsoft admits its GPL violation; will reissue Windows 7 tool under open-source license. http://blogs.zdnet.com/microsoft/?p=4547, November 2010. Accessed January 17, 2010.

[3] D. M. German, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol. Code siblings: Technical and legal implications. In *Proc. of the 2009 Working Conference on Mining Software Repositories, MSR*, pages 81–90, 2009.

[4] J. Howison, M. Conklin, and K. Crowston. FLOSSmole: A collaborative repository for FLOSS research data and analyses. *International Journal of Information Technology and Web Engineering*, 1(3):17–26, 2006.

[5] T. Kamiya, S. Kusumoto, and K. Inoue. 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.

[6] M. Kim and D. Notkin. Program element matching for multi-version program analyses. In *Proceedings of the 2006 International Workshop on Mining Software Repositories, MSR*, pages 58–64, 2006.

[7] S. Livieri, Y. Higo, M. Matsushita, and K. Inoue. Very-large scale code clone analysis and visualization of open source programs using distributed ccfinder: D-ccfinder. In *"Proceedings of the International Conference on Software Engineering"*, pages 106–115, 2007.

[8] E. Merlo and T. Lavoie. Computing structural types of clone syntactic blocks. In *"Proceedings of the Working Conference on Reverse Engineering"*, pages 274–278, 2009.

[9] N. H. Pham, H. A. Nguyen, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen. Complete and accurate clone detection in graph-based models. In *"Proceedings of the International Conference on Software Engineering"*, pages 276–286, 2009.

[10] W. Shang, Z. M. Jiang, B. Adams, and A. E. Hassan. MapReduce as a general framework to support research in Mining Software Repositories (MSR). In *Proceedings of the International Working Conference on Mining Software Repositories, MSR*, pages 21–30, 2009.