2017 IEEE/ACM 39th IEEE International Conference on Software Engineering Companion

Fast and Flexible Large-Scale Clone Detection with CloneWorks

Jeffrey Svajlenko Chanchal K. Roy
Department of Computer Science, University of Saskatchewan, Saskatoon, Canada
jeff.svajlenko@usask.ca chanchal.roy@usask.ca

***Abstract*—Clone detection in very-large inter-project repositories has numerous applications in software research and development. However, existing tools do not provide the flexibility researchers need to explore this emerging domain. We introduce CloneWorks, a fast and flexible clone detector for large-scale clone detection experiments. CloneWorks gives the user full control over the representation of the source code before clone detection, including easy plug-in of custom source transformation, normalization and filtering logic. The user can then perform targeted clone detection for any type or kind of clone of interest. CloneWorks uses our fast and scalable partitioned partial indexes approach, which can handle any input size on an average workstation using input partitioning. CloneWorks can detect Type-3 clones in an input as large as 250 million lines of code in just four hours on an average workstation, with good recall and precision as measured by our BigCloneBench.**

***Keywords*-code clone, clone detection, flexible, scalable, fast**

I. Introduction

Clone detection tools locate code clones, exact or similar code fragments, within or between software systems. Developers create clones when they reuse code using copy-paste and modify, although clones can arise for a variety of reasons [1]. By managing or refactoring their clones, developers can improve and maintain software quality, reduce development costs and risks, prevent and detect bugs and more [1].

One of the most active topics in clone research is the detection of clones within very-large inter-project source repositories containing on the order of thousands of software projects or more. This has many applications, including: studying global open-source developer behavior, mining the seeds of new APIs [2], license violation detection [3], similar mobile app detection [4], inter-scale clone search [5], and so on.

In order to achieve these emerging applications, fast, scalable and flexible clone detection tools are needed. While a small number of scalable tools and techniques are available [2]–[4], [6]–[10], they have limitations. Most of the techniques in the literature do not support the most important and most common Type-3 clones [2], [3], [8], [9]. Some require extraordinary hardware, in particular large amounts of memory, or distribution across a cluster [8], [9], which can be costly and difficult to setup. Others are domain-specific detectors designed for specific use-cases only [3], [4], [7].

The existing tools are also not very flexible, and customizing their clone detection beyond simple clone similarity thresholds and clone size parameters is difficult or not supported. Users need to be able to customize the detection process in order to target specific clones types, or to target new kinds of clones for novel clone detection use-cases and studies. In particular, the large-scale inter-project clone detection domain is rich in new opportunities for study. It is beneficial if the user can customize an existing tool rather than develop a new tool from scratch to achieve their goals, including how the source-code is processed before clone detection.

We introduce CloneWorks, a fast and flexible clone detector for large-scale clone detection experiments. CloneWorks gives the user full control over the representation of the source-code for clone detection, by allowing the user to specify the normalizations, transformations, filtering and other processing performed on the source-code, including custom processing by a plug-in architecture. Fast and scalable clone detection is achieved using a modified Jaccard similarity metric [11] and the sub-block filtering heuristic [6] with clone indexes. Input partitioning is used to scale within memory constraints, regardless of the input size. CloneWorks scales to 250MLOC in just four hours on an average workstation, with recall and precision competitive with the state of the art tools, including for the important Type-3 clones which are challenging and time-consuming to detect. CloneWorks supports the detection of Java, C and C# clones at the block, function and file granularity. CloneWorks is publicly available at http://jeff.svajlenko.com/cloneworks, including a demo video.

II. Definitions

Code Fragment: A continuous region of source code specified by its source file, start and end line numbers.
Code Clone: A pair, (f_1, f_2), of similar code fragments.
Type-1 Clone: Identical code fragments, except for differences in white space, layout and comments [1].
Type-2 Clone: Identical code fragments, except for differences in identifiers and literals, as well as Type-1 differences [1].
Type-3 Clone: Similar code fragments that differ at the statement level; statements are added, modified and/or removed [1].

III. The CloneWorks Approach

The CloneWorks approach is summarized in Fig. 1. It consists of two components: the flexible *input builder* and the fast and scalable *clone detector*. The input builder is used to extract the code fragments from the input source files and transform them into a set of terms representation for clone detection. The user has full control over the processing of the source-code, including normalizations, transformations and

978-1-5386-1589-8/17 $31.00 © 2017 IEEE
DOI 10.1109/ICSE-C.2017.3

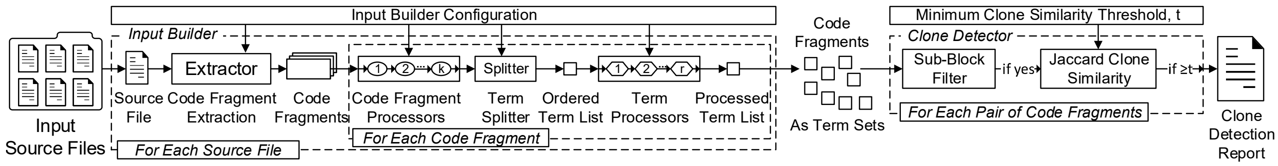

Fig. 1. The CloneWorks Approach

filtering, applied at the code-fragment and term level. A variety of processors are provided, and the user can provide their own by a plug-in architecture.

Clone detection is performed by a modified Jaccard similarity metric, with the denominator modified for clone detection, as shown in Eq. 1. The metric takes a pair of code fragments, f_1 and f_2, as the sets of terms they contain, including duplicates, and computes their minimum term intersection ratio. A pair of code fragments are reported as a clone if their similarity satisfies a given minimum threshold, t. Clone detection is scaled in execution time using the sub-block filtering heuristic [6] with partial clone indexes, which efficiently skips the comparison of many code fragment pairs that cannot satisfy the threshold. Fast clone detection is achieved by keeping the index and code fragments fully in memory in efficient data-structures. Scalability in memory is achieved using input partitioning with our partitioned partial clone indexes.

$$s(f_1, f_2) = \frac{|f_1 \cap f_2|}{max(|f_1|, |f_2|)} = min(\frac{|f_1 \cap f_2|}{|f_1|}, \frac{|f_1 \cap f_2|}{|f_2|}) \quad (1)$$

IV. Flexible Input Builder

The input builder extracts and transforms the code fragments into sets of terms, for any definition of a term. This can be as simple as tokenizing the code fragments into the set of language tokens they contain, or any imaginable representation of the code fragments as term sets. Users customize their code fragment representation by specifying the transformations, normalizations and filtering to be applied at the code-fragment and term levels. Users can add their own custom processing by a plug-in architecture. This gives the user full control over the representation of the code fragments for clone detection. The set of terms representation ultimately determines the kinds of detected clones, allowing the targeting of any clone type, or any novel kind of clone needed for a task or study.

The input builder is shown as executed per source file in Fig. 1. First, the source file is parsed and code fragments at a specified granularity are extracted. Extraction also normalizes the code fragments to remove Type-1 clone differences. The code fragments are then processed by k user-specified code-fragment processors. This includes the application of source transformations and normalizations, and/or the filtering of undesired code-fragments based on source analytics. Next, the code fragments are split into terms. The term splitter can either split the code fragments by language token, or by text line. The user can produce a custom term definition by using code-fragment processors to layout the code such that there is one desired term per line, then split by line. The terms are kept in their order of occurrence, and processed by r user-specified term processors. Like the code-fragment processors, these can apply transformations, normalizations and filtering, but at the term level. For example, they may be used to transform, filter, split, combine, and so on, the terms based on some conditions. Lastly, the code fragments' processed terms are converted into unordered set representations, and written to a file for later use with the clone detector. The input builder is multi-threaded, and processes multiple source files in parallel.

Code-Fragment Processors: A number of code-fragment processors for common source normalizations are provided, including: consistent identifier renaming, identifier normalization, literal normalization, conditional expression normalization, abstraction or filtering of any non-terminal in the language grammar, and so on. They are provided as stand-alone executables implementing a particular input/output and call behavior. They are called with the granularity and language of the code fragments as input parameters, as well as any custom input parameters, and are expected to take the code fragments as input and output them after processing. Users can use their own custom processors, implemented using any language or technology, by providing an executable that adheres to this behavior. The input builder sets up an execution chain of the processors in their specified order, with the code fragments exchanged in a simple standard format. The input builder collects the final processed code fragments for term splitting.

Term Processors: A number of term processors are provided, including: filter operator tokens, filter separator tokens, term stemming, n-gram transformation, term-joiner, string splitter, string normalizer, case normalizer, term hashing, and so on. They are implemented as Java classes, and users can add their own by implementing the term processor interface and adding their class, and its dependencies, to the distribution. They are discovered at runtime, and configured with the parameters specified by the user. They receive an ordered list of terms as input, and are expected to output that list after their defined processing. The term processor can return an empty list to filter the entire code fragment.

V. Fast and Scalable Clone Detection

CloneWork's clone detection process is summarized in Fig. 2. The modified Jaccard similarity metric is executed for each pair of code fragments, and those that exceed the minimum clone similarity threshold are reported as clones. To scale this computation, we use the sub-block filtering heuristic [6]. This heuristic computes a subset containing the $|f| - \lceil t|f| \rceil + 1$ least common terms in a code fragment f for a given similarity threshold t. The least common terms are identified by computing the global-term-frequencies across all of the code fragments. For two code fragments to possibly

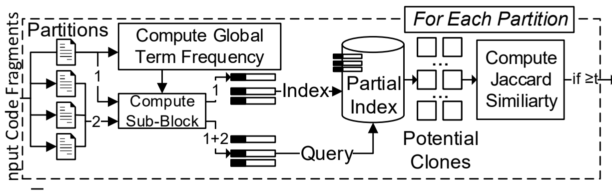

Fig. 2. Fast and Scalable Clone Detection

satisfy the modified Jaccard similarity threshold, their sub-blocks must overlap by at least one term. Code fragments whose sub-blocks do not overlap do not need to be checked if they are a clone. The sub-block heuristic is efficiently implemented using a partial clone index [6], where each code fragment is indexed by only the terms that appear in their sub-blocks. Querying the index with the terms in a code fragment's sub-block returns all potential clones of that code fragment to be checked. This is done for each code fragment.

CloneWorks uses a very fast implementation of this partial index approach. The index is implemented for constant-time queries, and the indexed code fragments are kept in memory. The code fragments are stored as hash-sets of their terms for linear-time evaluation of the modified Jaccard metric. We prioritize execution time at the cost of high memory usage.

To scale within available memory, we use input partitioning. The code fragments are split into a number of partitions, and clone detection is executed independently for each partition. Only the code fragments within the current partitioned are indexed and stored within memory. This *partitioned partial index* is queried for the code fragments in all of the partitions, returning all potential clones that involve a code fragment from the current partition. The code fragments from the other partitions are efficiently streamed into memory to query the index and evaluate their potential clones, but are not retained in memory. This is repeated for each partition to check all of the potential clones in the system. The number of partitions is chosen such that the available memory is not exceeded, allowing scalability to any input size. We call this the *partitioned partial indexes* approach. Multi-threading is exploited in each step of the computation.

VI. Usage

CloneWorks consists of three command-line tools, `cwbuild` for running the input builder, `cwdetect` for clone detection, and `cwformat` for formatting the clone results. The user begins by preparing the source code with `cwbuild`. They specify the directory or list of input source files, the source language(s) and the code fragment granularity. The user also provides a configuration file listing the code-fragment processors, term splitting and term processors to use. The processors are specified by name in the order they are to be applied, as well as any parameters they take. The processor plug-ins are discovered and configured at runtime. The input builder outputs a manifest of the source files and the code fragments as term-sets. The user then uses `cwdetect` to run clone detection over the code fragments. They specify the minimum similarity threshold

```
cfproc=rename-consistent # Rename identifiers
cfproc=abstract literal  # Normalize literals
termsplit=line  # Split into code statements
termproc=Joiner # Concatenate terms into single term
termproc=Hasher md5 # Perform md5 hashing on the terms
```

Fig. 3. Example Input Builder Configuration File for Type-2 Detection

for the modified Jaccard metric, the minimum and maximum sizes of the clones to detect, and the maximum partition size if input partitioning is desired. Since CloneWorks is designed for large-scale clone detection, it produces a very compact clone results file ideal for programatic analysis. `cwformat` is used to produce a clone result file ideal for human inspection. It can produce an XML output with or without source-code embeded, and is user-extensible to new formats. By separating CloneWorks into separate tools, users can add custom processing between each step, or replace any component, as needed for their task.

VII. Example Clone Detection Scenarios

We show how CloneWorks can target the first three clone types. CloneWorks includes these configurations as common usages. Fig. 3 is an example `cwbuild` config file for Type-2.

Type-1: Code fragment extraction automatically normalizes the code for Type-1 clone differences. The input builder can then be configured to split the code fragment into terms by line, with each line being a single code statement. A term-processor can be used to join these code statements together with uniform whitespace delimitation. A code fragment is then represented by the set of a single term, which is its Type-1 normalized source text. Clone detection with a 100% threshold will yield all Type-1 clones. To reduce memory load, the 'Hash' term processor can be used to replace the Type-1 normalized texts with their shorter hash-string.

Type-2: This extends the Type-1 configuration with the addition of two code-fragment processors to perform Type-2 normalizations. One processor to consistently rename the identifiers, and one to normalize the literal values. Type-2 detection is then achieved with a 100% similarity threshold.

Type-3: We provide two configurations for Type-3 detection, one conservative and one aggressive. The conservative configuration splits the code-fragments by language tokens, and uses term processors to filter the separator and operator type tokens. This represents code fragments as the set of identifiers, literals, primitives and keywords they contain. The aggressive configuration uses code-fragment processors to normalize identifier names and literal values, before splitting by line. This represents the code fragments as the set of statement-level code patterns they contain. Clone detection is then achieved with a similarity threshold such as 70%.

The following are some envisioned usages. These are neither included in CloneWorks nor implemented, but show how advanced novel clone detection experiments could be built on top of CloneWorks using the flexible input builder.

Domain-Specific Clones: A researcher may want to study only the clones within a particular programming domain (e.g., test code, database code, etc.). Using one of the type-specific

TABLE I
RECALL AND PRECISION MEASURED BY BIGCLONEBENCH

Tool	T1	T2	VST3	ST3	Precision
CloneWorks (Conservative)	100	99	94	62	93
CloneWorks (Aggressive)	100	100	100	96	83
iClones [16]	100	82	82	24	93
NiCad [17]	100	100	100	95	80
SourcererCC [6]	100	98	93	61	86

configurations as a base, the researcher could add processors to analyze the code fragments and filter those not part of the target domain. Code-fragment processors could implement regex patterns to look for indicative high level code patterns, while term processors could examine API usages for filtering.

Semantic Clones: Suppose a researcher has created a topic modeler for source code, which takes a code fragment as input and returns a list of its semantic topics, in order to detect semantic clones. They can accomplish this with CloneWorks by developing a code fragment processor that integrates with their topic modeler and transforms the code fragments into their semantic topics for splitting into topics as terms. The researcher can take advantage of CloneWork's parsers, input builder and fast clone detector without additional efforts.

VIII. PERFORMANCE EVALUATION

We evaluate CloneWork's performance using our BigCloneBench [12]–[14]. We measure recall and precision for the conservative and aggressive Type-3 configurations discussed in Section VII. The results are summarized in Table I per clone type, including the Very-Strongly ($\geq$90% similarity) and Strongly Type-3 (70-90% similarity) categories from BigCloneBench [13]. For comparison, we also include the top performing tools from our previous evaluation studies [6], [13], [15]. With the aggressive configuration, CloneWorks leads in recall performance alongside NiCad, while maintaining competitive precision. With the conservative configuration, CloneWorks leads in precision with the second best recall.

We evaluate the execution time of CloneWorks by executing it for IJaDataset-2.0 [13], a large inter-project Java repository (250MLOC). We use an average workstation with a 3.6GHz quad-core i7-2600, 12GB of memory, and solid-state drive. For comparison, we also execute SourcererCC [6], the only other tool to scale on a single workstation. We execute both with a minimum clone size of 10 lines, and a minimum similarity of 70%. CloneWorks with the conservative configuration requires just 4.2 hours, and the aggressive configuration requires 10.2 hours, while SourcererCC requires 109.8 hours. CloneWorks is one to two orders of magnitude faster, while matching the recall and precision of the state of the art tools.

IX. RELATED WORK

Liveri et al. [9] distributed CCFinder over a large cluster using input partitioning. We scaled existing tools without modification using partitioning and filtering heuristics at a cost of a reduction in recall [10]. Ishihara et al. [2] scaled Type-1/2 detection using hashing. Hummel et al. [8] were the first to use an index for scalable detection, but required a cluster to hold the index. Others have scaled in domain-specific ways [2]–[5], which cannot be used for general detection. With Sajnani et al. [6], we introduced SourcererCC, the first tool to use the sub-block filtering technique with a partial clone index. CloneWorks is distinct in that it adds flexible source transformation and processing with the input builder, extends to our novel partitioned partial indexes approach, and an efficient in-memory implementation, which reduces execution time by up to two orders of magnitude. Our NiCad [17] also provides flexible source transformations, but does not scale to large inter-project repositories. CloneWorks provides a finer granularity of control over the source-code processing, including custom processing by a plug-in architecture.

X. CONCLUSION

CloneWorks is fast and flexible clone detector for large-scale clone detection experiments. It allows the user to fully customize the representation of the source code for clone detection, to target specific clone types or to perform custom clone detection experiments. It performs clone detection with the modified Jaccard metric and sub-block filtering heuristic implemented efficiently with our partitioned partial index approach. It can scale Type-3 clone detection to an input of 250MLOC in just four hours with good recall and precision.

REFERENCES

[1] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," Queen's University, Tech. Rep. 2007-541, 2007, 115 pp.

[2] T. Ishihara, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto, "Inter-project functional clone detection toward building libraries - an empirical study on 13,000 projects," in *WCRE*, 2012, pp. 387–391.

[3] R. Koschke, "Large-scale inter-system clone detection using suffix trees and hashing," *J. Softw.: Evol. Process*, vol. 26, no. 8, pp. 747–769, 2014.

[4] K. Chen, P. Liu, and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on Android markets," in *ICSE*, 2014, pp. 175–186.

[5] I. Keivanloo, C. Forbes, and J. Rilling, "Similarity search plug-in: Clone detection meets internet-scale code search," in *ICSE-SUITE*, 2012.

[6] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "SourcererCC: Scaling code clone detection to big-code," in *ICSE*, 2016.

[7] I. Keivanloo, J. Rilling, and P. Charland, "Internet-scale real-time code clone search via multi-level indexing," in *WCRE*, 2011, pp. 23–27.

[8] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt, "Index-based code clone detection: incremental, distributed, scalable," in *ICSM*, 2010.

[9] S. Livieri, Y. Higo, M. Matushita, and K. Inoue, "Very-large scale code clone analysis and visualization of open source programs using distributed ccfinder: D-ccfinder," in *ICSE*, 2007, pp. 106–115.

[10] J. Svajlenko, I. Keivanloo, and C. K. Roy, "Big data clone detection using classical detectors: an exploratory study," *J. Softw.: Evol. Process*, vol. 27, no. 6, pp. 430–464, 2015.

[11] I. Keivanloo, C. K. Roy, and J. Rilling, "Sebyte: Scalable clone and similarity search for bytecode," *Science of Computer Programming*, vol. 95, Part 4, pp. 426 – 444, 2014.

[12] J. Svajlenko and C. Roy, "BigCloneEval: A clone detection tool evaluation framework with BigCloneBench," in *ICSME*, 2016.

[13] J. Svajlenko and C. K. Roy, "Evaluating clone detection tools with BigCloneBench," in *ICSME*, 2015, pp. 131–140.

[14] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, "Towards a big data curated benchmark of inter-project code clones," in *ICSME*, 2014, pp. 476–480.

[15] J. Svajlenko and C. K. Roy, "Evaluating modern clone detection tools," in *ICSME*, 2014.

[16] N. Göde and R. Koschke, "Incremental clone detection," in *CSMR*, 2009.

[17] C. K. Roy and J. R. Cordy, "Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *ICPC*, 2008, pp. 172–181.