CloneWorks: A Fast and Flexible Large-Scale Near-Miss Clone Detection Tool

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 within large inter-project source-code repositories has numerous rich applications. CloneWorks is a fast and flexible clone detector for large-scale near-miss clone detection experiments. CloneWorks gives the user full control over the processing of the source code before clone detection, enabling the user to target any clone type or perform custom clone detection experiments. Scalable clone detection is achieved, even on commodity hardware, using our partitioned partial indexes approach. CloneWorks scales to 250MLOC in just four on an average workstation with good recall and precision.**

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

I. Introduction

Clone detection tools locate exact or similar source code within or between software systems. An instance of similar code is called a clone, and multiple types of clones have been identified. Of interest is the detection of clones within very-large inter-project source repositories, containing hundreds of millions of lines of code (MLOC) or more. This is motivated by the exciting applications, such as: studying global open-source developer behavior [1], mining the seeds of new APIs [2], license violation detection [3], large-scale clone and code search [4], similar application detection [5], code completion [6], API recommendation and usage support [7], and so on. While a small number of scalable techniques have been published [2], [3], [5], [8]–[12], most do not support the important Type-3 clones [2], [3], [10], [11], some require distribution over a compute cluster [10], [11], while others are designed for domain-specific uses [3], [5], [9]. None of the existing tools provide researchers the flexibility they need to explore new applications of large-scale 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 processing of their source code before clone detection. Specifically, users can specify the normalizations, transformations, filtering and any other processing performed on the source-code, including custom processing by a simple plug-in architecture. By customizing the representation of their source code, users can target any clone type, or perform novel clone detection experiments. Very fast clone detection is achieved using efficient metrics and heuristics, parallelism, and prioritizing speed over efficient memory usage. An efficient input partitioning scheme is used to keep peak memory usage within even commodity memory constraints. CloneWorks executes for an input as large as 250MLOC in just four hours on an average workstation (quad-core, 10GB memory). CloneWorks supports the detection of Java, C and C# clones at the block, function and file granularity. It is available for download at www.jeff.svajlenko.com/cloneworks.

The CloneWorks approach is shown in Figure 1. It is split into two distinct tools: the flexible *input builder*, and the fast and scalable *clone detector*. Clone detection is achieved using a modified Jaccard similarity metric, which represents code fragments as the set of code terms they contain and measures similarity as their minimum overlap ratio. The input builder is used to extract the code fragments from the input source files and convert them to this set representation of clone detection. The user specifies the source-code processing with the input builder to customize this representation for their use-case. The clone detector then evaluates each pair of code fragments with the modified Jaccard metric. This is scaled using the sub-block filtering heuristic [8] with our *partitioned partial indexes* approach. We describe these details in the following sections. Further details are found in our tool demonstration paper [13].

II. Modified Jaccard Clone Similarity

CloneWorks detects clones using the 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 set of code terms (e.g. tokens) they contain, including duplicates. Similarity is measured as the minimum ratio of their intersection. A pair of code fragments is reported as a clone if their similarity satisfies a given minimum threshold (e.g. 70%). The metric can be evaluated in linear time when the code fragment set representations are stored in hash tables. Since the metric is language independent, the set representation of a code fragment can be anything imaginable. It can be the set of language tokens within the code fragment, or code statements, or normalized code statements, or API call patterns, etc. The user uses the input builder to customize the set representation to target specific kinds of clones.

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

III. Flexible Input Builder

The input builder is shown in Figure 1. It is used to extract the code fragments from the input source files, and convert them into a set of code terms format for clone detection with the modified Jaccard metric. The code fragments are prepared over a number of steps. First, a code fragment is processed by k user-specified code-fragment processors. These apply source transformations and normalizations to the

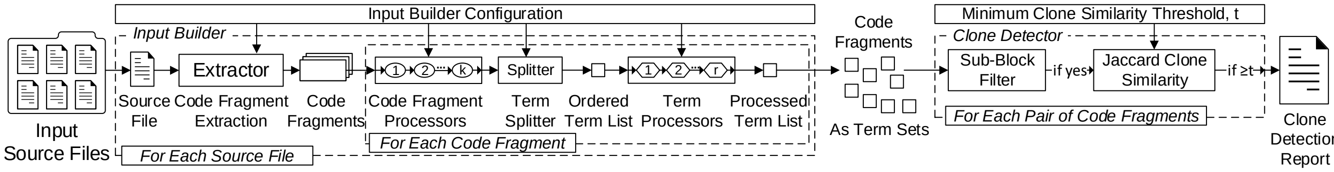

Fig. 1. The CloneWorks Approach

code fragments, and/or filter undesired code fragments from consideration. The code fragment is then split into terms by the term splitter, either by token or by text line. Using the code fragment processors to layout the code in a particular way then splitting by line can produce a custom term definition. The fragment's list of terms is then processed by r user-specified term processors. These are analogous to the code fragment processors, except they apply transformations, normalizations and filtering at the term level. Finally, the code fragment is output as an unordered set of its terms, ready for clone detection. The input builder processes multiple files in parallel, up to the number of available processing cores.

CloneWorks includes a number of code fragment processors including identifier renaming, literal normalization, conditional expression normalization, and non-terminal abstraction or filtering. It includes term processors for token filtering, n-gram transformations, string-splitting, hashing, and so on. Users can implement their own custom code fragment and term processors by a simple plug-in architecture. The user writes a configuration file that specifies the code fragment and term processors to use and their order. The input builder assembles this pipeline for processing the source-code. The user only needs to implement their custom processing logic, and can take advantage of the input builder's structure and multi-threaded processing. Further details and example usages are available in our tool demonstration paper [13].

IV. Fast and Scalable Clone Detector

Clone detection is shown at a high level in Figure 1. Every pair of code fragments is examined by the modified Jaccard similarity metric to determine if it is a clone, given a minimum clone similarity threshold. This is very wasteful, as most pairs of code fragments are not clones. We skip those pairs of code fragments which cannot be clones using the sub-block filtering heuristic [8]. Given code fragments f_1 and f_2 as term sets. If we order their terms by increasing global-term-frequency, then they can only be clones if the prefix of f_1 of size $|f_1|-t|f_1|+1$ overlaps the prefix of f_2 of size $|f_2|-t|f_2|+1$ by at least one term [8], [13]. Potential clones can be efficiently identified by indexing the code fragments by their prefix (sub-block) terms (an *partial clone index*) [8]. Querying the index using the prefix terms of a code fragment returns all of the potential clones of that code fragment that need to be investigated fully by the modified Jaccard metric. Querying the index with every code fragment returns all potential clones in the input. The sub-block filtering heuristic has been found to significantly reduce the number of code fragment comparisons [8].

This clone detection technique is extremely fast when the partial clone index is stored as a hash table for constant-time lookup, the code fragment set representations are stored

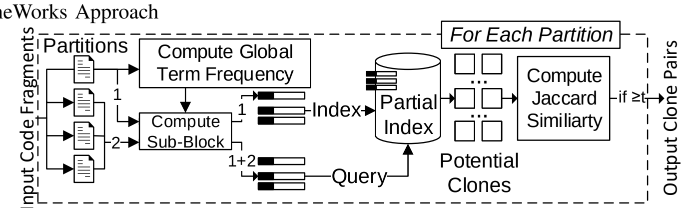

Fig. 2. Fast and Scalable Clone Detection

as hash sets for linear-time computation of the modified Jaccard metric, and the code fragments are kept in-memory for immediate random access when queried. However, this is very memory intensive, and quickly exceeds average memory constraints for larger inputs. To overcome this, we use our *partitioned partial indexes approach*. The code fragments are split into a number of partitions, such that each partition can fit within memory constraints. For each partition, a partial index is constructed for just the code fragments in that partition, and code fragments from each of the partitions are used to query the index to identify the potential clones. This is repeated for each partition to identify all of the potential clones. Only the code fragments of the current partition need to be kept in memory as they are being queried from the index. The other code fragments can be efficiently streamed from disk as they are used to query the index. This partitioned partial index approach is shown in detail in Figure 2.

V. Evaluation

We evaluated two configurations of CloneWorks using our BigCloneBench [14]–[16] clone benchmark. Our conservative (C) configuration represents the code fragments as their sets of tokens, with operator and separator tokens filtered. Our aggressive (A) configuration represents the code fragments as their set of code statements, after identifier and literal normalization. Recall (per clone type) and precision are shown below. These configuration match the best of the competing tools, as per our previous benchmarking studies [8], [15], [17]. We evaluated the execution time and scalability of CloneWorks by executing it for a IJaDataset [18], an inter-project repository with 250MLOC. The conservative configuration completed after just four hours, while the aggressive configuration required just ten hours, 1-2 orders of magnitude faster than the best of the competing tools [8]. Detection was performed on a workstation with a quad core processor and 10GB of RAM.

Config.	T1	T2	VST3	ST3	Precision
CloneWorks-A	100	99	92	65	95
CloneWorks-C	100	99	98	92	86

VI. Conclusion

In this paper we over-viewed the major concepts of CloneWorks, our fast and flexible clone detector for large-scale near-miss clone detection experiments. Further details can be found in our tool demonstration paper [13].

REFERENCES

[1] J. Ossher, H. Sajnani, and C. Lopes, "File cloning in open source java projects: The good, the bad, and the ugly," in *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance*, ser. ICSM '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 283–292.

[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 *2012 19th Working Conference on Reverse Engineering*, Oct 2012, pp. 387–391.

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

[4] I. Keivanloo, C. Forbes, and J. Rilling, "Similarity search plug-in: Clone detection meets internet-scale code search," in *2012 ICSE Workshop on Search-Driven Development - Users, Infrastructure, Tools and Evaluation (SUITE)*, June 2012, pp. 21–22.

[5] K. Chen, P. Liu, and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on android markets," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 175–186.

[6] T. Ishihara, Y. Higo, and S. Kusumoto, "How often is necessary code missing? a controlled experiment," *Software Reuse for Dynamic Systems in the Cloud and Beyond*, vol. 8919, pp. 156–163, 2014.

[7] I. Keivanloo, J. Rilling, and Y. Zou, "Spotting working code examples," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 664–675.

[8] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: Scaling code clone detection to big-code," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 1157–1168.

[9] I. Keivanloo, J. Rilling, and P. Charland, "Internet-scale real-time code clone search via multi-level indexing," in *2011 18th Working Conference on Reverse Engineering*, Oct 2011, pp. 23–27.

[10] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt, "Index-based code clone detection: incremental, distributed, scalable," in *2010 IEEE International Conference on Software Maintenance (ICSM)*, Sept 2010, pp. 1–9.

[11] 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 *29th International Conference on Software Engineering (ICSE'07)*, May 2007, pp. 106–115.

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

[13] J. Svajlenko and C. K. Roy, "Fast and flexible large-scale clone detection with cloneworks," in *Proceedings of the 39th International Conference on Software Engineering (Tool Demonstrations Track)*, 2017, 4 pp.

[14] J. Svajlenko and C. Roy, "Bigcloneeval: A clone detection tool evaluation framework with bigclonebench," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016, to appear.

[15] J. Svajlenko and C. K. Roy, "Evaluating clone detection tools with bigclonebench," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sept 2015, pp. 131–140.

[16] 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 *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, ser. ICSME '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 476–480.

[17] J. Svajlenko and C. K. Roy, "Evaluating modern clone detection tools," in *Proceedings of the 30th International Conference on Software Maintenance and Evolution*, 2014, pp. 321–330.

[18] Ambient Software Evoluton Group, "IJaDataset 2.0," http://secold.org/projects/seclone, January 2013.