Poster: Fast, Scalable and User-Guided Clone Detection

Jeffrey Svajlenko
Department of Computer Science
University of Saskatchewan
Saskatoon, Canada
jeff.svajlenko@usask.ca

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

ABSTRACT

Despite the great number of clone detection approaches proposed in the literature, few have the scalability and speed to analyze large inter-project source datasets, where clone detection has many potential applications. Furthermore, because of the many uses of clone detection, an approach is needed that can adapt to the needs of the user to detect any kind of clone. We propose a clone detection approach designed for user-guided clone detection by exploiting the power of source transformation in a plugin based source processing pipeline. Clones are detected using a simple Jaccard-based clone similarity metric, and users customize the representation of their source code as sets of terms to target particular types or kinds of clones. Fast and scalable clone detection is achieved with indexing, sub-block filtering and input partitioning.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**;

KEYWORDS

Clone Detection, User Guided, Fast, Scalable, Large-Scale

ACM Reference Format:
Jeffrey Svajlenko and Chanchal K. Roy. 2018. Poster: Fast, Scalable and User-Guided Clone Detection. In *Proceedings of International Conference on Software Engineering, Gothenburg, Sweden, 2018 (ICSE'18),* 2 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Clone detection locates instances of similar code, called clones, within or between software systems. One of the most active topics in clone research is the detection of clones within large inter-project source code datasets containing on the order of thousands of software projects or more. This has applications in software research [11], license violation detection [8], building new software libraries [4], API recommendation [6], code completion [3], code search [5, 9], and so on. However, few detection approaches are able to scale to large inter-project source code datasets [13], and none of the existing approaches are user-guided in their configuration, which is needed by researchers and practitioners to explore the emerging applications of inter-project clone detection. To overcome these challenges, we propose a fast, scalable and user-guided clone detection approach.

2 CORE APPROACH

Our approach represents code fragments (blocks, functions, files, and so on) as the unordered set of code terms (e.g. tokens, lines) they contain. Clones are detected using a Jaccard-based [7, 12, 13, 16] clone similarity metric (Eq. 1). Code fragments f_1 and f_2 are reported as a clone if their minimum set overlap exceeds a given threshold (e.g., 70%). We use this metric as it is fast, language-independent, scalable, simple to understand and customizable.

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

By customizing the representation of the code fragments as sets of terms, this simple approach can target any kind clone. Classical clones can be detected by representing the code fragments as the sets of source tokens, lines or statements they contain. Normalization and filtering of the terms can be used to target clones with particular types of differences. Transformations can be used to customize the term definition to detect novel kinds of clones. For example, code fragments could be represented by their set of topics using topic modeling to detect semantically similar clones, while extracting normalized API calls as terms could be used to detect cloned API usage patterns.

3 USER-GUIDED INPUT BUILDER

Our user-guided approach enables the user to easily customize how their code fragments are represented as sets of terms for clone detection. To achieve this we designed a source-code parsing and processing pipeline for extracting the code fragments as sets of terms for chosen term definition. The pipeline uses a plugin architecture so that users can modify any aspect including injecting custom source transformation, normalization and filtering. We call this pipeline the input builder.

The pipeline is executed as in Figure 1. Each source file is parsed and the code fragments of a specified granularity are extracted and pretty-printed. Then, for each code fragment in that file, the following processing occurs. First, a number of user-specified code-fragment processors are applied to the code fragment, which can apply normalizations and transformations to the source syntax. Then, the terms are extracted by term splitting, which outputs the code fragment as a list of the terms it contains, including duplicates, in the order of their occurrence. The term list is then processed by a user-specified sequence of term processors, which take a term list as input and output the same list with some specified modifications, such as term filtering, splitting, combining, transformation, and so

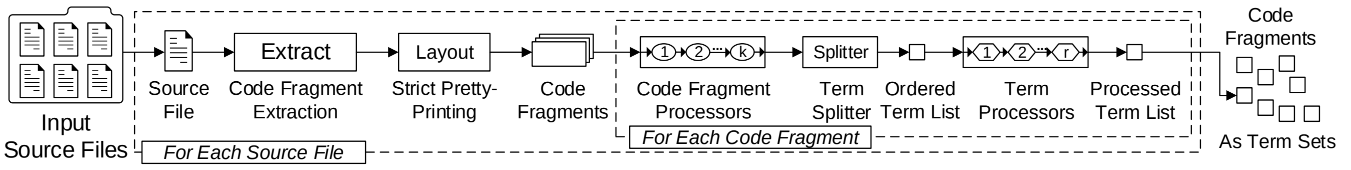

Figure 1: The Input Builder

on. The term list is then reformatted to a set of terms, including duplicates, for clone detection.

Extraction: Language specific extractor plugins parse the source files and output code fragments at the desired granularity (e.g. block, function, file). Extractor plugins can leverage existing parser technology (e.g., TXL [1]).

Pretty-Printing: A strict pretty-printing plugin is used to layout the code fragments for eventually splitting (by newline) into code terms.

Code-Fragment Processing: Code fragment processors are plugins used to apply source-code normalization, abstractions, filterings and transformations. Normalizations such as identifier name and literal value normalizations are common in clone detection. Abstractions can normalize any syntactic structure, and can be used to detect clones with particular kinds of edits. Filtering can remove syntax elements that cause mismatch of otherwise similar source code during clone detection. Transformations can be used to customize the code term definition for clone detection.

Term Processors: These can be used to refine the code terms before clone detection. For example, an n-gram term processor can be used to preserve some syntax order information during clone detection with the Jaccard metric.

4 FAST AND SCALABLE CLONE DETECTION

Jaccard-based clone similarity metrics are commonly used in clone detection for their linear complexity [7, 12, 13]. To scale the Jaccard approach we use Sajnani et al.'s sub-block filtering heuristic [12, 13], which avoids comparing code fragments that cannot satisfy the chosen minimum similarity threshold, and partial indexing approach [13]. Clone indexing [2, 13] is used to quickly identify the potential clones for comparison, as determined by the filter [13]. We have found this approach is exceptionally fast when the code fragments and index are kept in-memory in computationally efficient but memory intensive datastructures [17]. To scale within memory constraints we use determinsitic input partitioning over the clone index and input code fragments. Further details are available in our CloneWorks tool paper [17].

5 RELATED WORK

Livieri et al. [10] used deterministic input partitioning to scale clone detection using a compute cluster. With Keivanloo, we used non-determinsitic partitioning with heuristics to scale existing non-scalable clone detectors to large inter-project datasets [14]. Hummel et al. [2] proposed the use of indexes for scalable clone detection. Koschke et al. [8] scale clone detection with suffix trees for the purpose of license violation detection, but detects only Type-1 and Type-2 clones. Ishihara et al. [4] used hashing with clone index to find Type-1 and Type-2 cloned methods to build new libraries.

A number of works have used a Jaccard-based clone similarity metric [7, 13, 14, 17]. Sajnani et al. showed how this could be scaled using their sub-block filtering heuristic [12] and implemented it with a partial clone index in SourcererCC [13]. We reuse this approach here to scale our user-guided clone detection. We improved the execution time of this approach using computational efficient but memory intensive data-structures, and improved scalability even within limited memory using input partitioning over the code fragments and clone index [15]. We released this implementation as CloneWorks, and demonstrated its performance [17]. Unique to this work is the user-guided approach which can target any kind of clone detection. We initially explored this concept with CloneWorks, and provide a number of fragment and term processors [15, 17]

6 ACKNOLWEDGEMENTS

This research is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

REFERENCES

[1] James R. Cordy. 2006. The TXL source transformation language. *Science of Computer Programming* 61, 3 (2006), 190 – 210.
[2] Benjamin Hummel, Elmar Juergens, Lars Heinemann, and Michael Conradt. 2010. Index-based code clone detection: incremental, distributed, scalable. In *ICSM*. 1–9.
[3] Tomoya Ishihara, Yoshiki Higo, and Shinji Kusumoto. 2014. How Often Is Necessary Code Missing? – A Controlled Experiment. *Soft. Reuse for Dynamic Systems in the Cloud and Beyond* 8919 (2014), 156–163.
[4] Tomoya Ishihara, Keisuke Hotta, Yoshiki Higo, Hiroshi Igaki, and Shinji Kusumoto. 2012. Inter-Project Functional Clone Detection Toward Building Libraries - An Empirical Study on 13,000 Projects. In *WCRE*. 387–391.
[5] Iman Keivanloo, Christopher Forbes, and Juergen Rilling. 2012. Similarity search plug-in: Clone detection meets internet-scale code search. In *SUITE*. 21–22.
[6] Iman Keivanloo, Juergen Rilling, and Ying Zou. 2014. Spotting Working Code Examples. In *ICSE*. 664–675.
[7] Iman Keivanloo, Chanchal K. Roy, and Juergen Rilling. 2014. SeByte: Scalable clone and similarity search for bytecode. *Sci. of Comp. Program.* 95(4) (2014).
[8] Rainer Koschke. 2014. Large-scale inter-system clone detection using suffix trees and hashing. *Journal of Software: Evolution and Process* 26, 8 (2014), 747–769.
[9] Mu-Woong Lee, Jong-Won Roh, Seung-won Hwang, and Sunghun Kim. 2010. Instant Code Clone Search. In *FSE*. 167–176.
[10] S. Livieri, Y. Higo, M. Matushita, and K. Inoue. 2007. Very-Large Scale Code Clone Analysis and Visualization of Open Source Programs Using Distributed CCFinder: D-CCFinder. In *ICSE*. 106–115.
[11] Joel Ossher, Hitesh Sajnani, and Cristina Lopes. 2011. File Cloning in Open Source Java Projects: The Good, the Bad, and the Ugly. In *ICSM*. 283–292.
[12] Hitesh Sajnani and Cristina Lopes. 2013. A parallel and efficient approach to large scale clone detection. In *IWSC*. 46–52.
[13] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. 2016. SourcererCC: Scaling Code Clone Detection to Big-code. In *ICSE*.
[14] Jeffrey Svajlenko, Iman Keivanloo, and Chanchal K. Roy. 2015. Big data clone detection using classical detectors: an exploratory study. *Journal of Software: Evolution and Process* 27, 6 (2015), 430–464.
[15] J. Svajlenko and C. K. Roy. 2017. CloneWorks: A Fast and Flexible Large-Scale Near-Miss Clone Detection Tool. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. 177–179. https://doi.org/10.1109/ICSE-C.2017.78
[16] Jeffrey Svajlenko and Chanchal K. Roy. 2017. CloneWorks: A Fast and Flexible Near-Miss Clone Detection Tool. www.jeff.svajlenko.com/cloneworks. (2017).
[17] Jeffrey Svajlenko and Chanchal K. Roy. 2017. Fast and Flexible Large-Scale Clone Detection with CloneWorks. In *ICSE-C*. 27–30.