The Journal of Systems and Software 137 (2018) 130–142

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Scalable code clone detection and search based on adaptive prefix filtering

Manziba Akanda Nishi*, Kostadin Damevski

Department of Computer Science, Virginia Commonwealth University, United States

A R T I C L E   I N F O

Article history:
Received 27 June 2017
Revised 21 September 2017
Accepted 17 November 2017
Available online 20 November 2017

MSC:
00-01
99-00

Keywords:
Code clone detection
Prefix filtering
Software maintenance

A B S T R A C T

Code clone detection is a well-known software engineering problem that aims to detect all the groups of code blocks or code fragments that are functionally equivalent in a code base. It has numerous and wide ranging important uses in areas such as software metrics, plagiarism detection, aspect mining, copyright infringement investigation, code compaction, virus detection, and detecting bugs. A scalable code clone detection technique, able to process large source code repositories, is crucial in the context of multi-project or Internet-scale code clone detection scenarios. In this paper, we focus on improving the scalability of code clone detection, relative to current state of the art techniques. Our adaptive prefix filtering technique improves the performance of code clone detection for many common execution parameters, when tested on common benchmarks. The experimental results exhibit improvements for commonly used similarity thresholds of between 40% and 80%, in the best case decreasing the execution time up to 11% and increasing the number of filtered candidates up to 63%.

1. Introduction

Developers introduce clones in a code base mainly when reusing existing code blocks without significant alteration, or when certain code blocks are implemented by developers following a common mental macro (Roy and Cordy, 2007; Baxter et al., 1998; Kamiya et al., 2002). Researchers have shown that developers tend to perform software maintenance tasks more effectively when they have the results of code clone detection (Chatterji et al., 2013,2011). Empirical studies have noted that code clones are widespread, and that a significant portion of source code (between 5% and 20%) is copied or modified from already implemented code fragments or blocks (Roy and Cordy, 2007, 2008a; Baker, 1995).

Performing code clone detection across numerous software repositories is a common use case. Specific applications for large scale code clone detection include querying library candidates (Ishihara et al., 2012), categorizing copyright infringement and license violations (Koschke, 2012; German et al., 2009), plagiarism detection (German et al., 2009; Koschke, 2012), finding product lines in reverse engineering (Hemel and Koschke, 2012; German et al., 2009), tracing the origin of a component (Davies et al., 2011), searching for code blocks in large software repositories (Yamashina et al., 2008; Keivanloo et al., 2011a), and spotting analogous applications in Android markets (Chen et al., 2014; Sajnani et al., 2016). However, most existing code clone detection techniques have difficulty scaling up to extremely large collections of source code (Svajlenko et al., 2013; Roy et al., 2014).

Among the tools aimed towards large scale code clone detection, a common limitation is in the complexity of differences among clones they can detect. For instance, scalable token based approaches (Kamiya et al., 2002; Hummel et al., 2010; Ishihara et al., 2012) have difficulty detecting near miss (Type-3) code clones, which can occur more frequently than other types of clones (Roy et al., 2014; Roy and Cordy, 2010; Svajlenko et al., 2014). Parallel and distributed clone detection techniques like D-CCFinder (Livieri et al., 2007) can be more burdensome to manage, requiring specialized hardware or software support, while tree based code clone detection technique, such as Deckard (Jiang et al., 2007), place higher demands on memory.

In this paper, we propose a token-based code clone detection technique aimed at scalability and detecting Type-3 clones, consisting of two main steps: filtering and verification. In the filtering step we aim to significantly reduce the number of code blocks for comparison, removing from consideration blocks that do not have any possibility of being code clones. In the verification step we determine whether candidate pairs that survived the filtering step are really code clones. This two step process greatly reduces the runtime of code clone detection, allowing the technique to scale up to very large corpora. This technique is based on the *adaptive prefix filtering heuristic* (Wang et al., 2012), which is an extended

* Corresponding author.
E-mail addresses: nishima@mymail.vcu.edu (M.A. Nishi), kdamevski@vcu.edu (K. Damevski).

https://doi.org/10.1016/j.jss.2017.11.039
0164-1212/© 2017 Elsevier Inc. All rights reserved.

version of *prefix filtering heuristic* (Sarawagi and Kirpal, 2004; Vernica et al., 2010) previously applied towards code clone detection in SourcererCC (Sajnani et al., 2016). To our knowledge, SourcererCC is the best scaling code clone detection tool able to detect Type-3 clones. In this paper, we demonstrate improvements in execution time relative to SourcererCC, while obtaining the same accuracy, for many common similarity thresholds, when evaluated on a large scale inter-project source code corpus (Ambient Software Evoluton Group, 2017).

A separate novel idea presented in this paper is the effective application of our technique to code clone search, without modification, in addition to code clone detection. Code clone search is a related problem to code clone detection where the user specifies a single code block (i.e. query block) to search for in a corpus of many code blocks. Once indexed, the corpus should be able to serve numerous such queries. Ours is among few techniques that can be applied to both of these problems. The contributions of this paper are the following:

- A novel code clone detection technique that can scale to very large scale source code repositories (or sets of repositories) with the ability to detect Type-1, Type-2, and Type-3 code clones, while maintaining high precision and recall.
- An extension of our proposed technique so that it can be effectively utilized for code clone search without modification.

We have organized the rest of this paper as follows. Section 2 describes the background and related work in both code clone detection and code clone search. Section 3 describes the adaptive prefix filtering heuristic, which we utilized in our code clone detection approach. Section 4 describes the design of a code clone detection system based on our technique, which includes several necessary optimization steps. Section 5 describes the experimental results evaluating our proposed approach, as well as a comparison to the recent code clone detection tool SourcererCC. Section 6 concludes the paper, summarizing it's contributions.

2. Background and related work

Based on the nature of the similarity between code blocks, the software engineering community has identified four types of code clones, by which code clone detection techniques can be organized. Syntactically equivalent code blocks are called Type-1 clones. Type-2 code clones are code blocks that are syntactically comparable but are slightly contrasting in terms of variable names, function names, or identifier names. If two code blocks contain statements that have been inserted, altered, expunged, there is a gap in statements, or statement order differs, then these are called Type-3 clones. Semantically equivalent code blocks are called Type-4 clones (Sajnani et al., 2016).

Scaling code clone detection to work across multiple repositories is a specific area of interest among researchers. Popular and notable examples of large-scale code clone detection include CCFinderX (Kamiya et al., 2002), which is one of the foremost token based code clone detection tools able to scale up to large repositories and detect Type-1 and Type-2 code clones. In Hummel et al. (2010), an inverted index-based approach was first proposed, detecting Type-1 and Type-2 clones. Deckard (Jiang et al., 2007) is another tool that aims to scale to large source code repositories. It uses a tree-based data structure and detects clones by identifying similar subtrees and is able to detect up to Type-3 code clones. NiCad (Cordy and Roy, 2011) is a scalable code clone detection tool that can detect Type-3 clones using a technique based on parsing, normalization and filtering. When compared using similar execution parameters, CCFinderX scales up to 100 million lines of code, Nicad scales up to 10 million lines of code, while Deckard scales up to 1 million lines of code (Sajnani et al., 2016).

Parallel, distributed or online (i.e. incremental) techniques add another dimension in examining scalable code clone detection technique. For instance, iClone (Göde and Koschke, 2009) is the first incremental code clone detection technique that detects code clones in the current version of code repository by leveraging executions on previous versions of the same repository. It uses a suffix tree-based and token-based approach that can detect Type-1 and Type-2 code clones. A scalable distributed code clone detection tool named D-CCFinder was proposed in Livieri et al. (2007), which can scale code clone detection in a distributed environment to detect Type-1 and Type-2 clones. In Svajlenko et al. (2013) a scalable code clone detection technique has been introduced where input files are partitioned into smaller subsets and a shuffling framework is utilized to allow the clone detection tool to execute on each of the subsets separately, enabling it to detect code clones in parallel.

Recently, the SourcererCC (Sajnani et al., 2016) code clone detection tool proposed a token-based prefix filtering code clone detection technique, which greatly reduces the number of candidate code clone pairs, enabling it to detect up to Type-3 clones in Internet-scale source code repositories. SourcererCC is the best scaling tool on a single machine that we are aware of. The approach described in this paper extends SourcererCC with an adaptive approach that allows for even greater gains in performance for large-scale source code datasets.

SourcererCC is based on two filtering heuristics, prefix filtering and token position filtering, which reduce the number of candidate pairs that require costly pairwise comparison of all of their tokens (Sarawagi and Kirpal, 2004; Vernica et al., 2010). These two filtering heuristics attempt to rapidly, with few token comparisons, detect pairs of code blocks that diverge very significantly from each other. To perform this task, a subset (or prefix) is isolated in each of the two code blocks, where if there are no matching tokens in the subsets then we can safely reject them as a candidate pair, without proceeding further, and without attempting to compare all of their tokens. On the other hand, a single matching token in the subset allows the pair to proceed to further scrutiny as a code clone.

Recently, an extremely scalable code clone detection tool VUDDY (Kim et al., 2017) has been presented. VUDDY's purpose is to detect vulnerable code clones for security improvement. While VUDDY has been shown to be significantly faster than SourcererCC it is designed to only detect Type-1 and Type-2 code clones. So far, the precision and recall of VUDDY has only been evaluated for relatively few instances of code with security vulnerabilities.

Wang et al. (2012) recently proposed an additional filtering heuristic to those used in SourcererCC, called *adaptive prefix filtering*. This technique posits that deeper prefix lengths, which attempt more aggressive filtering at a higher performance cost, can achieve good performance on some types of input. An adaptive prefix filtering technique attempts to estimate the right level of filtering for each candidate by optimizing the trade-off between the cost of deeper filtering with the benefit of reducing the number of candidates. This paper applies adaptive prefix filtering to code clone detection, and evaluates it's adequacy.

2.1. Code clone search

Code clone search is a related area of research where a single code block is supplied as a query to be matched in large source code corpus, retrieving a list of code clones. Code clone search requires the source code to be pre-indexed, and for the similarity threshold between the query block and it's clones to be specified at query time, instead of, at indexing time. This twist makes it challenging for typical code clone detection to be used without