Information Sciences 259 (2014) 544–554

Contents lists available at ScienceDirect

Information Sciences

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

Pattern mining of cloned codes in software systems

Wei Qu [a,*], Yuanyuan Jia [b], Michael Jiang [c]

[a] Graduate University of Chinese Academy of Sciences, 80 East Zhongguancun Road, Haidian, Beijing 100190, PR China
[b] Bioengineering Department, University of Illinois, Chicago, IL 60607, USA
[c] Motorola Labs, Motorola Inc., Schaumburg, IL 60196, USA

ARTICLE INFO

Article history:
Received 7 March 2008
Received in revised form 26 February 2010
Accepted 19 April 2010
Available online 30 April 2010

Keywords:
Pattern mining
Software clone detection
Software reuse detection
Software engineering

ABSTRACT

Pattern mining of cloned codes in software systems is a challenging task due to various modifications and the large size of software codes. Most existing approaches adopt a token-based software representation and use sequential analysis for pattern mining of cloned codes. Due to the intrinsic limitations of such spatial space analysis, these methods have difficulties handling statement reordering, insertion and control replacement. Recently, graph-based models such as program dependent graph have been exploited to solve these issues. Although they can improve the performance in terms of accuracy, they introduce additional problems. Their computational complexity is very high and dramatically increases with the software size, thus limiting their applications in practice. In this paper, we propose a novel pattern mining framework for cloned codes in software systems. It efficiently exploits software's spatial space information as well as graph space information and thus can mine accurate patterns of cloned codes for software systems. Preliminary experimental results have demonstrated the superior performance of the proposed approach compared with other methods.

1. Introduction

Software clone detection has received a significant amount of attention in recent years due to its numerous applications such as source code plagiarism detection, open source localization, intellectual property infringement uncovering, software debugging, etc. [1–3]. Code clone and reengineering is a common practice in development of software systems. It is widely used by developers to reduce programming efforts and shorten developing time. However, such kind of software reuse may introduce additional problems including quality instability, intellectual property infringement, redundancy increase, and has possibilities to make the software less efficient [1–3]. How to effectively and efficiently discover these cloned codes, analyze their patterns and thus optimize the software structure becomes a very important issue.

Pattern mining has received much attention recently due to its wide applications [4–6]. Pattern mining of exactly cloned software codes is much easier since it can be solved reasonably well by using regular text search techniques. However, pattern mining of cloned codes for software with modifications is a more difficult task [1,2]. In addition to all of the challenging problems inherent to text searching, pattern mining of cloned codes must deal with alterations from simple modifications such as reformatting, comment changes, identifier renaming to more complicated changes such as statement reordering, insertion, and control logic changes, some of which may be very hard for human beings to identify because of tricky variations and large software size. Our experience with large software systems showed that cloned software modules often ap-

* Corresponding author.
E-mail addresses: weiqu@gucas.ac.cn, quweiusa@gmail.com (W. Qu), yjia2@uic.edu, jiayuanusa@gmail.com (Y. Jia), machiel.jiang@motorla.com (M. Jiang).

0020-0255/$ - see front matter © 2010 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.ins.2010.04.022

pear in different locations (in different files, directories, and product lines). Without tool support, manual identification is impractical.

Most early efforts for pattern mining of cloned codes relied on the use of tokenization and spatial space analysis, where only code location information such as sequential line indices or file paths is exploited. Tokens are the basic units in a programming language, such as keywords, operators, parentheses, etc. Such methods usually consist of two stages: firstly, a parser or a lexical analyzer filters a program into a sequence of tokens; then a sequential analysis method is used to compare token sequences and detect similar counterparts. A string-based approach was proposed by Baker in [1]. Wise [7] proposed a YAP algorithm, which relies on the "*sdiff*" function in UNIX to compare lists of tokens for the longest common sequence of tokens. Gitchell and Tran presented a SIM plagiarism detection system comparing token sequences using a dynamic programming string alignment technique in [8]. JPlag [3] and MOSS [9] are two widely used token-based tools for programming plagiarism detection, especially in academic area. Recently, Kamiya et al. [10] proposed CCFinder, a clone detection technique with transformation rules and a token-based comparison. Li et al. [11] developed a tool for finding copy-paste and related defects in operating system code. It is based on frequent subsequence mining and tokenization techniques. Chen et al. designed a token-based system called SID in [2]. It uses a metric based on Kolmogorov complexity to measure the shared information between two programs.

Although token-based sequential analysis methods can handle format changes and identifier renaming since blanks and comments are ignored by the parser and variables of the same type are filtered into the same tokens, they have intrinsic limitations for pattern mining of cloned codes due to using only spatial space analysis. For example, reordered or inserted statements can break a token sequence which may otherwise be regarded as a duplicate to another sequence. These limitations have recently inspired researchers to exploit other space information. Baxter et al. [12] proposed a tool that transforms source code into abstract syntax trees (AST) and detects code reuse by finding identical subtrees. However, it may introduce many false positives because two code segments with same syntax subtrees may not be necessarily reused code. Komondoor et al. proposed to use program dependence graph (PDG) and program slicing to find isomorphic subgraphs and code duplication. Another PDG-based approach was proposed by Krinke in [13]. This notion is carried further in the work of Liu et al. [14], which improves the plagiarism search efficiency by a PDG-based GPlag algorithm. This work provides a very promising direction to resolve the problems of pattern mining of cloned software codes in logic-domain analysis. However, since general subgraph isomorphism is NP-complete [15], these logic-domain approaches suffer from the exponentially increased computational complexity with the size of software code, and thus limit their use in practice. Although a lossy filter may partially remedy this issue, it does not fundamentally solve the computational complexity of the graph-based algorithms and may introduce a lot more false negatives and false positives.

In this paper, we extend the approach presented in [16] and propose a new framework for pattern mining of cloned codes using a joint space-logic-domain analysis. In particular, it discards the software tokenization, which sacrifices too much information to tolerate identifier renaming. Instead, it exploits graph-based analysis for software representation. Efficient spatial fingerprinting is exploited to generate "seed matches". Graph matching is also used to recover lost information and enhance mining accuracy. Compared with our previous work in [16], the new approach is particularly designed for pattern mining within a single large-scale software program instead of software reuse detection between two programs. Moreover, this paper extends our previous work by exploiting a combination of two filters, one lossy filter and one lossless filter, in spatial pattern search. Finally, both false positive pruning and pattern composition are further adopted to improve the pattern mining performance. The rest of the paper is organized as follows: In Section 2, we give a brief description of program dependence graph. In Section 3, we present the proposed pattern mining framework for cloned software codes. Experimental results on software systems are shown in Section 4. Finally, we provide a summary in Section 5.

2. Program dependence graph

Since we use program dependence graph in the proposed framework, we briefly summarize it in this section.

The program dependence graph (PDG) [17] represents a program as a directed and labeled graph in which the nodes are statements and predicate expression such as variable declarations, assignments, etc., and the edges incident to a node represent both data values and control conditions [18]. Following the notation in [18,14], we define a PDG as follows:

The program dependence graph G for a subroutine is a 4-tuple element $G = (V, E, \mu, \delta)$, *where V is the set of program nodes,* $E \subseteq V \times V$ *is the set of edges,* μ *is the set of nodes' types, and* δ *is the edges' types.*

Fig. 1 illustrates an example of PDG, where the right column is the associated code. As we can see, there are three subroutines. The nodes in the PDG represent individual statements and predicates of a subroutine. The edges represent the data and control dependence among statements and predicates. Specifically, the solid lines are control flow, and the dash lines are data flow.

Similar to [14,13], we also adopt subgraph isomorphism for the analysis of PDGs in our pattern mining framework. However, the proposed method distinguishes itself from these two references in the following aspects: (1) As presented in Section 3, our approach exploits both spatial pattern mining and graph-based pattern mining while [13] and [14] all use PDG-based pattern search only. (2) As discussed in Section 1, these two logic-domain approaches [13,14] suffer from the exponentially increased computation complexity due to the NP-complete problem. Nevertheless, the computation complexity of the proposed method is much lower because of a combination of PDG transformation and spatial pattern mining. (3)