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

A R T I C L E   I N F O

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

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

A B S T R A C T

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

* 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.
doi:10.1016/j.ins.2010.04.022

appear 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

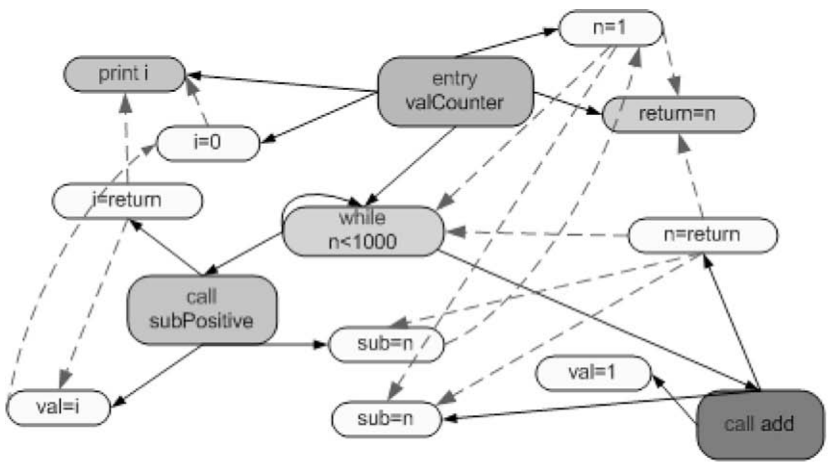
Since we use program dependence graph in the proposed framework, we briefly summarize it in this section.

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)



```
int subPositive(int sub, int val)
{
    ...
}

int valCounter()
{
    int n = 1;
    int i = 0;
    while ( n < 1000 )
    {
        i = subPositive(n, i);
        n = add(n, 1);
    }
    printf("The value = %d\n", i);
    return n;
}

int add(int sub, int val)
{
    ...
}
```

Fig. 1. An example of the program dependence graph (PDG).

As demonstrated in the experiments in Section 4, the proposed approach achieved superior performance in terms of not only robustness but also speed. The subgraph isomorphism used in our approach can be defined as follows [14]:

A bijective function f: $V \to V'$ is a subgraph isomorphism from G to G′ if there exists a subgraph $S \subset G'$ such that f is a graph isomorphism from G to S, where the graph isomorphism is defined as a bijective function f from a graph $G = (V, E, \mu, \delta)$ to a graph $G' = (V', E', \mu', \delta')$ if $\mu(v) = \mu'(f(v))$, $\forall e = (v_1, v_2) \in E$, $\exists\ e' = (f(v_1), f(v_2)) \in E'$, such that $\delta(e) = \delta(e')$, $\forall e' = (v'_1, v'_2) \in E'$, $\exists e = (f^{-1}(v'_1), f^{-1}(v'_2)) \in E$, such that $\delta(e') = \delta(e)$.

γ-isomorphic is a specific subgraph isomorphism. It can be further defined as:

A graph G is γ-isomorphic to G′ if there exists a subgraph $S \subseteq G$ such that S is subgraph isomorphic to G′, and $|S| \geqslant \gamma|G|$, $\gamma \in (0, 1]$.

For the details of program dependence graph and its implementation, we refer readers to [18,14].

3. Pattern mining of cloned codes

In this section, we present our pattern mining approach for cloned codes in software systems. Fig. 2 shows the system structure. Different with the regular data mining techniques such as clustering [6] or association rule [19], we propose a novel framework combining spatial space analysis and graph-based mining in order to effectively and efficiently handle the challenging pattern mining issues in large-scale software. Specifically, it includes a five-step process. Firstly, we adopt program dependence graph to represent and analyze the source code and analyze the source code. Then, both spatial pattern search and graph-based pattern mining are applied to, respectively, find candidate patterns. After that, false positive pruning and pattern composition techniques are exploited to update the pattern database and discover more accurate and meaningful patterns of cloned codes. Finally, the detected software patterns can be used for various applications, such as pattern analysis, related defect discovery, and code optimization.

3.1. PDG transformation

The investigated software system can be very large including many files. For simplicity, we assume that all the files within a software system are concatenated together to construct a single sequence. The purpose of pattern mining of cloned codes in a software system is to find the patterns and their appearance numbers within this sequence.

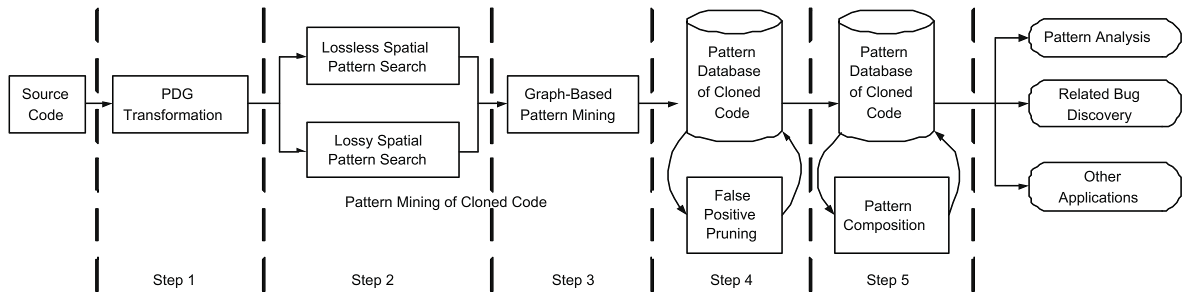

Source Code
PDG Transformation
Lossless Spatial Pattern Search
Lossy Spatial Pattern Search
Pattern Mining of Cloned Code
Graph-Based Pattern Mining
Pattern Database of Cloned Code
False Positive Pruning
Pattern Database of Cloned Code
Pattern Composition
Pattern Analysis
Related Bug Discovery
Other Applications
Step 1
Step 2
Step 3
Step 4
Step 5

Fig. 2. The system structure of pattern mining in software systems.

In order to tolerate various software modifications, we exploit program dependence graph to represent the source codes. The results are usually many graph nets with nodes and edges. The range of the number of nodes in each graph may vary from hundreds to several thousands depending on the inherent logic relationship inside the software files. Due to the high complexity of these graphs, directly searching for identical or similar subgraphs is computationally prohibited. Although many approaches have been proposed to reduce the searching complexity and thus get less optimal approximated results, they usually only work well for the graph having hundreds of nodes at most. Moreover, the speed of direct graph search is usually very slow since the computational complexity may increase exponentially with the size of the source codes. This greatly limits their applications in software system where the software's size is commonly very large with millions of lines of codes. We propose a novel searching scheme to deal with this problem. Specifically, we introduce spatial pattern searching techniques into the graph space since this kind of algorithm has an advantage that the computational complexity is much lower than that of graph matching. In order to apply the proposed spatial pattern searching algorithm, we need to firstly do a PDG transformation. Fig. 3 illustrates the main steps and Fig. 4 shows an example. Specifically, we firstly rearrange the graph nodes according to their associated line indices in the software sequence. Then because the types of nodes and edges are finite and known as stated in [13,14], we can index all nodes and edges by their types. For example, in our experiments, two types of edges, control dependence edge and data dependence edge are used. Seven types of node similar to [14] are also exploited for the analysis. By doing this, each line of nodes and edges can be converted to an index string. After applying a hashing algorithm, each line of code can be encoded to a hash value. Finally, an encoded graphic sequence is constructed to represent the original source code. It will be used for further pattern mining.

The transformation from graph representation to the encoded graphic sequence is a one to one mapping. The transformation from the source code to the graph representation in the first step is a n to one mapping. Although losing some information, this graph transformation based mapping is nonetheless very desirable because it provides the ability to tolerate not only the simple modifications such as format alteration, and identifier renaming, but also more complicated changes such as statement reordering, insertion and control replacement. Compared with the common token-based approaches [7,9,3], PDG captures a software's inherent logic relationship and thus gives a more efficient representation of software's logic structure.

3.2. Spatial pattern search

There are two options in the spatial pattern search step: lossless search and lossy search.

3.2.1. Lossless search

Lossless search is a thorough search for the whole sequence. The advantage of this method is that it can comprehensively find all the cloned pairs of codes in the sequence. The disadvantage is that the computational cost is rather expensive. For a sequence having n lines of codes, $n!$ times of comparisons are needed to find the pairwise matches.

3.2.2. Lossy search

Lossy spatial search methods can achieve much more efficient clone detection than the lossless search by compressing the data and sacrificing a part of information. Different search techniques have been studied in the literatures [8,9,3]. In this paper, we use Winnowing algorithm [9] to show an example.

Winnowing [9] is an efficient local fingerprinting algorithm designed to guarantee that matches of a certain length are detected. Given a set of documents, the purpose is to find the substring matches between them that satisfy two properties:

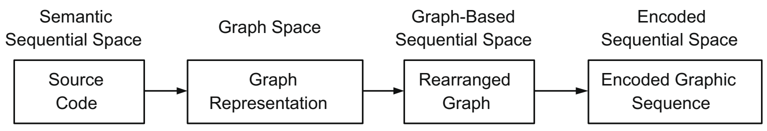


Fig. 3. The PDG transformation in the proposed framework.

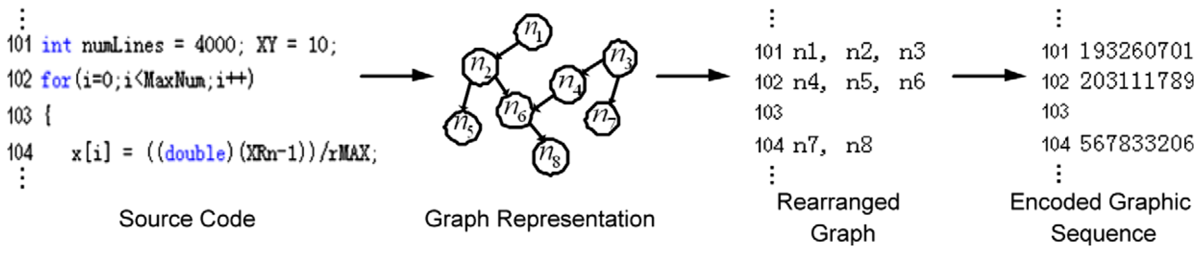


Fig. 4. An example of the proposed PDG transformation.

(1) at least as long as the guarantee threshold t; (2) not shorter than the noise threshold k. Both constants t and $k \leqslant t$ are chosen by the user. We avoid matching pattern below the noise threshold by considering only hashes of k-grams. In our experiments, these constants were empirically selected for different software code. For example, $k = 10$, $t = 20$ could be used initially in a particular test. Then a fine-tuning process could be used to further find the optimal values. For a sequence of hashes $h_1,\ldots,h_n$, if $n > t - k$, then at least one of the h_i must be chosen to guarantee detection of all matches of length at least t. This suggests an approach as follows. Let the window size be $\omega = t - k + 1$. Consider the sequence of hashes $h_1h_2,\ldots,h_n$ that represents a document. Each position $1 \leqslant i \leqslant n - \omega + 1$ in this sequence defines a window of hashes $h_i,\ldots,h_{i+\omega-1}$. To maintain the guarantee, it is necessary to select one hash value from every window to be a fingerprint of the document. Specifically, in each window, Winnowing selects the minimum hash value. For example, in the encoded graphic sequence of Fig. 4, if window length is selected as 4 and lines 101–104 are in the same window, then 193260701 is the minimal hash value in Winnowing algorithm. If there is more than one hash with the minimum value, select the rightmost occurrence. All selected hashes then construct a fingerprint list of the document. For the implementation details of Winnowing algorithm, we refer readers to [9].

In practical applications, lossless search and lossy search can be used by different combinations for various purposes. For example, in a medical imaging system such as angiography or Magnetic Resonance Imaging (MRI), one can apply lossy search for the acquisition and visualization modules while still using lossless search for the postprocessing module when the user pays more attention to the postprocessing codes and expects to find more detailed patterns in this module. Moreover, different searching filters can be adopted for different files even within one module. To reduce the spatial searching complexity, our current implementation limits the search to only pairwise matches. In other words, only line to line instead of block to block matches are considered in the spatial search step. When one line of codes is found repeated more than a certain number of times (the predefined threshold), it will be selected as a seed for the next step of graph-based pattern mining. Fig. 5 shows an example of the main pattern mining steps. As we can see in the spatial pattern search, line 3056 has been found as a cloned code to line 103. This pair is then selected as a seed to pull out two corresponding PDGs for the later graph-based pattern analysis.

3.3. Graph-based pattern mining

The matched lines of cloned codes through either lossless or lossy spatial search are usually very scattered. It is desirable to merge them to get bigger and more meaningful blocks. Moreover, when lossy search techniques such as Winnowing algorithm are used, only the selected lines called fingerprints are compared. All other lines are ignored to trade for speed. Obviously, such kind of methods cannot avoid missing lots of matches and thus increase the false negative rate. These weaknesses inherent in the spatial search make it very necessary to further improve the searching results.

In the existing literature, the merging process is usually achieved by using a simple spatial threshold [3]. For example, if the distance between two detected lines in a sequence is less than the threshold, these two lines and the lines between them are regarded to be one pattern and merged together. It is easy to see that this scheme is problematic. First of all, how to determine the threshold is challenging. It is usually very sensitive to different software. Even for one particular software, different thresholds may be needed for different parts. Secondly, the neighboring fingerprints may not have any relationship between each other at all. Neither are the lines between them. Simple concatenating fingerprint lines by spatial information may generate additional errors. Finally, such a scheme is not robust to software reordering and insertion. Since merging is made based on only spatial relationship, it may generate quite different results on reordered softwares even no other modifications are made. For the information loss problem incurred by fingerprinting or data compression, the existing approaches are usually unable to recover the lost information.

To handle the above difficulties, we propose to further use a second step graph-based pattern mining in the pattern finding of cloned codes. For each "seed" line pair, we retrieve the associated PDGs. As illustrated in Fig. 5, seed pair (103, 3056)

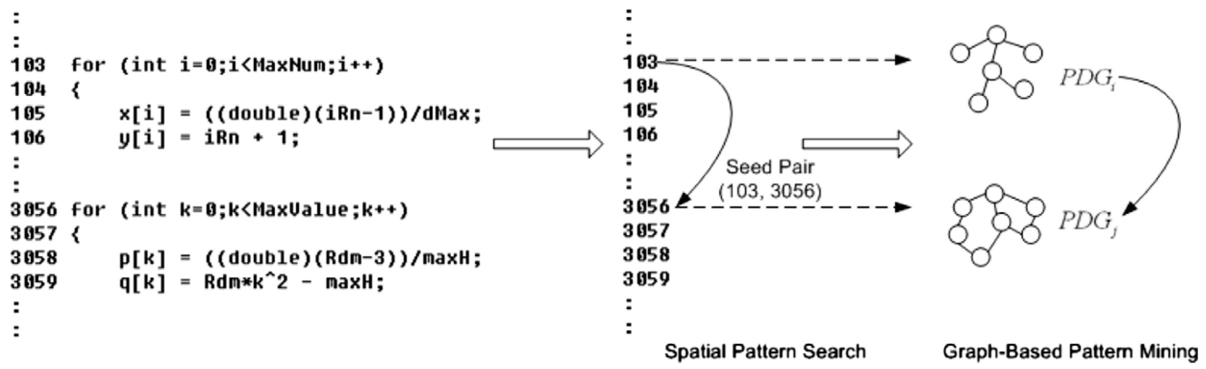

```
:
:
103  for (int i=0;i<MaxNum;i++)
104  {
105       x[i] = ((double)(iRn-1))/dMax;
106       y[i] = iRn + 1;
:
:
3056 for (int k=0;k<MaxValue;k++)
3057 {
3058      p[k] = ((double)(Rdm-3))/maxH;
3059      q[k] = Rdm*k^2 - maxH;
:
:
```

Fig. 5. An example of the proposed pattern mining steps.

retrieves two PDGs, PDG_i and PDG_j, from the source program. Although these two PDGs share at least one or more lines of cloned code, they may also have different codes. For example, in Fig. 5, lines 106 in PDG_i is different with 3059 in PDG_j. Then we use graph matching algorithm on each PDG pair to further detect if there is any matched subgraph inside the PDG pair. In order to reduce computational cost, it only outputs the first match in case there are several. Different graph matching algorithms may be used [20,21]. In our experiments, we applied VF algorithm [20] to detect the isomorphic subgraphs. Although general subgraph isomorphism is NP-complete, this algorithm is computationally efficient for graphs with hundreds and thousands of nodes. To allow more trivial changes, Liu et al. applied γ-isomorphic testing instead of full subgraph isomorphism in [14]. For a better comparison, we also use γ-isomorphic here. Specifically, if h, $h \subset H$ is γ-isomorphic $(0 < \gamma \leqslant 1)$ to h', $h' \subset H'$, the subroutine associated with h' can be regarded as software reuse of that of h. The mature rate γ is selected as 0.9 in experiments.

As we can see, during graph-based pattern mining, each seed "pulls" out a graph net. Therefore, although spatial pattern search loses some information because of data compression, the proposed graph-based pattern mining provides the ability to recover the lost information as long as one node from a graph is found as a seed in the spatial pattern search. Moreover, the problem of merging neighboring detected lines is also solved in an innovative way. Specifically, there is no distance threshold needed anymore since we do not merge the detected lines based on the spatial information in this step. Instead, the detected cloned code patterns are merged through the associated graph.

3.4. False positive pruning

The mined patterns may have false positives. This is mainly because the graph transformation in the first step in Fig. 2 ignores some information to tolerate software modifications. For instance, the code location information such as line indices and file paths is ignored in order to tolerate code clone changes such as reordering and insertion. Moreover, different codes may share identical nodes and edges in their PDG representation if they have similar logic structures. Such graph transformation sacrifices some information with the benefits of tolerating modifications like control replacement and renaming. Since there are several types of nodes and edges in the program dependence graph, in our experiments, we found that using fewer types of nodes and edges will generate more false positives. Meanwhile, it has a benefit that the searching speed can be faster since the complexity of software graphs will be lower. In order to prune false positives, we adopt the following two steps.

3.4.1. Removing tiny patterns

Tiny patterns of cloned codes usually do not have as much meaning as large ones. For example, although some simple short commands repeat hundreds of times in a software, they could not be regarded as intentionally "cloned" codes because they are normally used by the programming language. In other words, they are not the desirable patterns we are looking for. Keeping all such kind of tiny patterns will make the database very big and difficult to mine more meaningful patterns out. Therefore, we regard tiny patterns which are shorter than a threshold, say 5, as false positives and remove them from the database.

3.4.2. Calculating the mapping ratio

Based on an observation that when a code block is reused, an identifier is most likely modified in the same way in all occurrences of the cloned code. Therefore, similar to [11], we also adopt a scheme to evaluate the mapping ratio. Specifically, we generate an identifier mapping list for each pattern candidate. Each mapping associates the original identifier names in one code block to their corresponding new ones in the other code repeats which belong to the same pattern of cloned code. If the overall ratio of all identifier consistency associated to the pattern is more than a predefined threshold, say, 0.8, we regard it as a good pattern. Otherwise, it is a false positive.

The above pruning scheme could not guarantee to remove all false positives. However, we find that they are very effective to greatly decrease the number of false positives.

3.5. Pattern composition

The importance of a cloned code pattern depends on two factors: its size and the repeat frequency. For patterns with same or comparable frequency, the large patterns are usually more meaningful than the smaller ones. Therefore, we further apply a scheme to compose the patterns together and thus improve the pattern database.

3.5.1. Merging overlapped patterns

If one pattern is overlapped with another one, we merge them together to get a new larger pattern. However, different with the false positives, the original patterns are not removed from the database in this situation.

3.5.2. Merging neighboring patterns

If two patterns are in one file and have a spatial distance less than a predefined threshold, say 100 lines, they can be concatenated together to form a new pattern.

3.6. Potential applications

Although this paper focuses on pattern mining of cloned code instead of the pattern applications, we have found several valuable applications that apply pattern mining of cloned code to software development and maintenance.

Pattern analysis can be used to investigate and help improve the efficiency of software systems. The final results of our approach provide two lists of patterns: one ordered by the reused times and the other by the size of patterns. If one pattern appears very frequently, the related code patterns are good targets for code optimization. The reason is that if the efficiency of these frequently used patterns are improved, the overall efficiency of the whole software could be greatly increased. For instance, in one of our experiments, we found such a pattern, which repeatedly opens and closes the same file in different subroutines. Due to the I/O access cost, this pattern obviously could be improved by changing the open and close operation more globally out of each subroutines and simply providing them the file handle. On the other hand, the large patterns usually correspond to some features and may have concrete meanings. In either case, the discovery of cloned code provides opportunities for optimization and improvement. In a word, such kind of pattern analysis can provide a deep understanding of the source code in large software systems having millions of lines of codes.

The mined patterns also have potential to discover related defects. For example, if two pieces of codes are very similar in logic space but have only small part of difference, it is very likely that there are modification defects in these codes. Nevertheless, how to effectively use the mined patterns to discover defects is still an open problem and beyond the topic of this paper. It may need a lot of empirical knowledge. And the performance may be sensitive to different types. In this paper, we only provide our preliminary results and would like to keep this interesting topic for future study. Our experiment with software product lines showed that pattern mining of cloned code could improve the quality and simplify the maintenance of software product lines. Software product lines were pioneered by the Software Engineering Institute and have become a viable software design and development paradigm to improve productivity and quality through large-scale reuse. The maintenance of software product lines, however, can be very challenging as they evolve over a long period of time and when the number of related product lines is large. Traceability of reuse code across product lines is difficult to maintain. Our experiment showed that pattern mining could help recover reused code and dependencies lost due to product line evolution. When defects in reused code are fixed, related reused code in other product lines can be located through pattern mining and the necessary changes can be applied.

4. Experimental results

To demonstrate the effectiveness and efficiency of the proposed approach, we performed experiments on different programs in a health care software system. Statistical data of the test programs used in our experiment is provided in Table 1. All experiments were performed on a 3.2 GHz Pentium IV PC. We implemented our algorithm using Matlab and C/C++ without code optimization.

4.1. Proof of concept

Since it is usually difficult to manually find all actual patterns of cloned code for large software such as `X-CV` or `LVA` in Table 2, in order to show the proof-of-concept, we selected the smallest software `Reader` in our experiments to evaluate the robustness of the proposed approach against the state of the art [3,9,11,14]. Six kinds of modifications were manually applied on 15 patterns of cloned codes in the program `Reader`: (1) changing comments; (2) adding blank space lines; (3) renaming identifiers; (4) reordering statements; (5) inserting statements; and (6) control replacement (for example, substitute *while* with *for*). Table 2 shows the total number of different changes. Then, we tested the performance of five pattern mining approaches on the generated target program. Table 2 also provides the experimental results. Instead of using a successful rate, it lists the number of failures. If a pattern mining method missed a specific kind of modification, we define it as a failure. Thus, Table 2 shows what kind of modifications could cheat each testing pattern mining approaches. In other words, it reflects the ability of each method to handle different types of medications. As we can see, all five algorithms could successfully handle format alterations such as adding comments or blank space lines. Identifier renaming was also solved very well. However, due to only using token-based spatial analysis, JPlag and MOSS could not deal with statement reordering, insertion and control replacement. CPMiner improved the performance for renaming, reordering, and insertion, but could

Table 1
Statistics of the testing data.

Software	Lines of codes	Function
`Reader`	1007	Dicom reader
`Driver`	6553	Driver
`Window`	10,787	Interface
`LVA`	27,312	Image processing
`X-CV`	1,056,795	X-ray imaging

Table 2
Failure case comparisons of pattern mining on program `Reader`.

Modifications	Total changes	JPlag	MOSS	CPMiner	GPlag	Proposed method
Adding comments	25	0	0	0	0	0
Adding space	18	0	0	0	0	0
Renaming	139	13	4	2	1	0
Reordering	43	36	27	11	8	0
Insertion	72	22	31	14	7	0
Control replacement	51	51	49	48	3	0

not handle control replacement. By exploiting a more effective graph-based software representation, GPlag achieved much better results. However, compared with our approach, GPlag still made several mistakes because it excluded the PDGs smaller than a certain size.

4.2. Robustness

In order to demonstrate the effectiveness of the proposed approach, we further applied five different methods on program `Driver`. Since it is much larger than program `Reader`, it is hard to manually find the complete set of actual patterns of cloned code. Instead, we collected all correct patterns found by five different approaches and regarded them as the total number of patterns. Then we calculated the false negative rate (FNR) and the false positive rate (FPR) as follows:

$$\mathrm{FNR} = \frac{\#\text{Total Patterns} - \#\text{Mined Patterns}}{\#\text{Total Patterns}} \times 100\% \tag{1}$$

$$\mathrm{FPR} = \frac{\#\text{Mined False Patterns}}{\#\text{Total Patterns}} \times 100\% \tag{2}$$

Table 3 presents the comparisons of the experimental results. JPlag [3] and Moss [9] have been commonly used to find similar program among a given set. However, as we can see, due to the intrinsic limitations of token-based methods, they were not suitable for detecting patterns of cloned code in a single large program. Both JPlag and MOSS got very high false negative rate and false positive rate. An abstract syntax tree (AST) based approach [12] had better performance. However, because this approach disregarded the information about variables, it ignored data flows and thus was fragile to statement reordering as well as control replacement. CPMiner [11] improved the performance but still had very higher false negative rate and false positive rate since it is a token-based method as well. A detailed look of the source codes revealed that there are many control replacements such as while/for, and switch/if-else in the cloned codes in this program. Fig. 6 illustrates an example of clone code pattern which was detected by our approach but missed by all token-based methods. As we can see, it has a lot of tricky modifications such as reordering, insertion, renaming, and control replacement. This is why token-based methods did not work as well as GPlag and our approach. Compared with GPlag, our approach achieved even better results not only because it did not discriminate the small size PDGs as GPlag but also due to the more effective and efficient integrated framework using both spatial space analysis and graph-based pattern matching. Without neglecting the small size PDGs and adopting a lossless filter, GPlag may have potential to achieve similar mining results as the proposed method for small-scale softwares since both of them exploit graph matching. However, as shown in the speed analysis, due to the intrinsic limitations of pure graph analysis, GPlag suffered from the exponentially increased computational complexity with the size of software code.

For other testing programs, our approach discovered many patterns of cloned codes too. Table 4 shows the number of total cloned codes and the number of associated patterns mined from these cloned codes. Fig. 7 illustrates an example of the mined pattern of cloned code from the testing software.

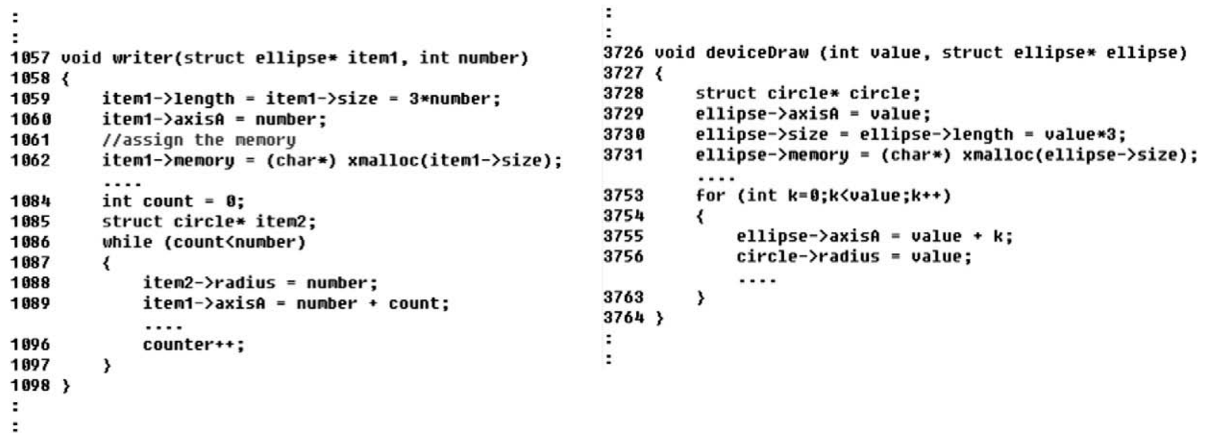
We randomly investigated 50 patterns for each program and found only 0–2 false positives in these testing cases. Moreover, it seemed that the larger the size of the pattern, the less likely it was a false positive. Most found false positives had a relative low rank in the mined pattern list ordered by the patterns' size. The reason may be that there is more information in the large patterns which could be exploited to ensure the correctness.

The proposed approach also provides the ability to find potential clone-related defects. For example, Fig. 7 illustrates the defects we found in the testing program `LVA`. As we can see, most identifiers were renamed in the cloned codes except the

Table 3
Robustness comparisons on program `Driver`.

Software	JPlag	MOSS	AST	CPMiner	GPlag	Proposed method
FNR (%)	51	42	22	19	10	2
FPR (%)	25	23	20	14	7	1

Note: FNR refers to False Negative Rate; FPR is False Positive Rate.



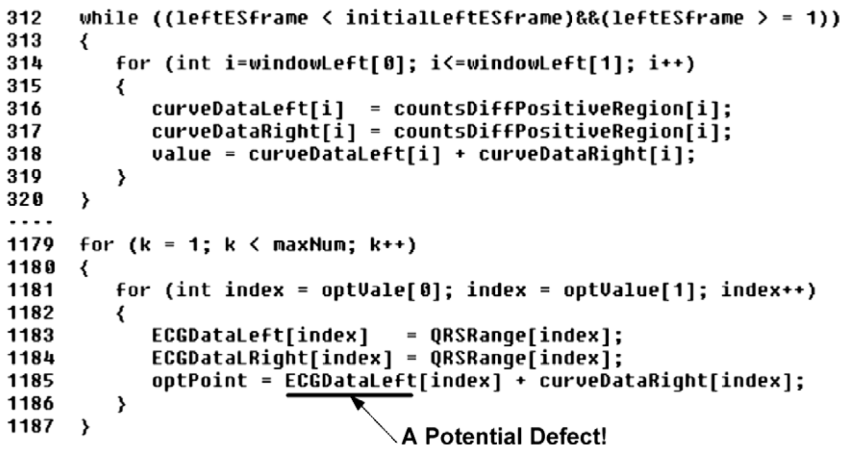
```
:
:
void writer(struct ellipse* item1, int number)
{
    item1->length = item1->size = 3*number;
    item1->axisA = number;
    //assign the memory
    item1->memory = (char*) xmalloc(item1->size);
    ....
    int count = 0;
    struct circle* item2;
    while (count<number)
    {
        item2->radius = number;
        item1->axisA = number + count;
        ....
        counter++;
    }
}
:
:
```

```
:
:
void deviceDraw (int value, struct ellipse* ellipse)
{
    struct circle* circle;
    ellipse->axisA = value;
    ellipse->size = ellipse->length = value*3;
    ellipse->memory = (char*) xmalloc(ellipse->size);
    ....
    for (int k=0;k<value;k++)
    {
        ellipse->axisA = value + k;
        circle->radius = value;
        ....
    }
}
:
:
```

Fig. 6. An example of mined pattern.

Table 4
Patterns mining results using our approach.

Software	Window	LVA	X-CV
#Cloned codes	1839	5126	91,711
#Patterns	43	91	5572



```
312   while ((leftESframe < initialLeftESframe)&&(leftESframe > = 1))
313   {
314      for (int i=windowLeft[0]; i<=windowLeft[1]; i++)
315      {
316         curveDataLeft[i]  = countsDiffPositiveRegion[i];
317         curveDataRight[i] = countsDiffPositiveRegion[i];
318         value = curveDataLeft[i] + curveDataRight[i];
319      }
320   }
....
1179  for (k = 1; k < maxNum; k++)
1180  {
1181     for (int index = optVale[0]; index = optValue[1]; index++)
1182     {
1183        ECGDataLeft[index]   = QRSRange[index];
1184        ECGDataLRight[index] = QRSRange[index];
1185        optPoint = ECGDataLeft[index] + curveDataRight[index];
1186     }
1187  }
```

Fig. 7. An example of mined pattern and its related potential defect.

Table 5
Pattern related defects found using our approach.

Software	Window	LVA	X-CV
#Potential defects	0	5	79
#Confirmed defects	0	3	15

variable `curveDataRight` in line #1185. Therefore, it is very likely that this was a careless mistake by the developer. Table 5 shows such errors the proposed method found for the testing programs. These potential defects can be further checked by the software developers to determine if they are real defects. As we can see from Table 5, some detected potential defects have been confirmed by software developers. The effectiveness of detecting defects may vary for different defect types as well as different testing softwares.

4.3. Speed

We have compared the efficiency of the proposed approach with GPlag. The program `LVA` totally has 357 different subroutines. For each time we took a part of the whole subroutines and applied both approaches. Fig. 8 shows the average pat-

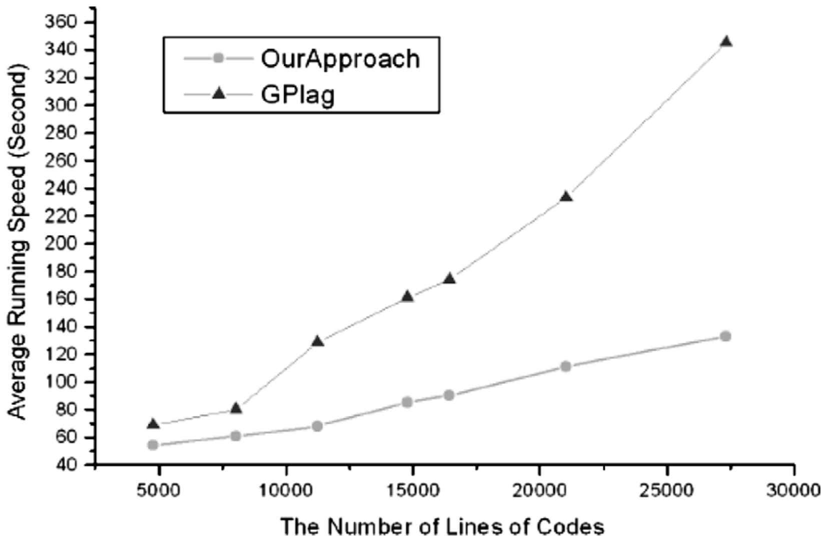


Fig. 8. Speed comparisons.

tern ming time of GPlag and our approach in different cases. Considering GPlag has to set several parameters to get a tradeoff between accuracy and efficiency, we experimentally chose these parameters under a condition that it could achieve comparable results with our approach. As we can see, when the program size was small, there was no big difference between the speed of these two approaches. However, when the program's size increased, the speed of our method outperformed GPlag a lot. This is mainly because the computational complexity of fingerprinting based spatial search is much lower than that of direct graph matching. Although our approach also exploits PDG-based graph matching, it is only used for the seed graphs. Thus the computational cost is much less than that of GPlag.

5. Conclusions

In this paper, we have presented a pattern mining framework for cloned codes in software systems. Compared with the common practice of using a token-based method, the proposed approach exploits a graph-based analysis and solves the problems of complicated modifications such as statement reordering, insertion and control replacement. Compared with the methods using only graph matching, our approach reduces the high computational complexity and thus greatly improves the algorithm's efficiency. Preliminary experimental results have shown the superior performances of the proposed approach compared with existing methods for pattern mining of cloned codes in large software systems.

Several important issues of the proposed pattern mining framework remain unsolved and will be the subjects of future research: (1) In the current implementation of the proposed approach, PDG nodes are reordered only based on the line location without considering file information. A hierarchical structure including such kind of information may expedite searching speed and improve pattern accuracy. (2) As discussed in Section 3.2, the current framework has two options in the spatial pattern search step and can only be manually chosen for different software modules or files. A scheme to automatically switch between the lossless search and the lossy search based on the investigation purpose and module's characteristics will be very interesting. It could greatly improve our approach's efficiency as well as the accuracy. (3) How to evaluate the mined patterns and exploit them with prior knowledge for system optimization is under investigation.

Acknowledgment

This work is supported in part by the Open Projects Program of National Laboratory of Pattern Recognition and the President Fund of GUCAS.

References

[1] B.S. Baker, On finding duplication and near duplication in large software systems, in: Proceedings of the Second Working Conference on Reverse Engineering, Toronto, Canada, 1995, pp. 86–95.
[2] X. Chen, B. Francia, M. Li, B. Mckinnon, A. Seker, Shared information and program plagiarism detection, IEEE Transactions on Information Theory 50 (7) (2004) 1545–1551.
[3] L. Prechelt, G. Malpohl, M. Philippsen, Finding plagiarisms among a set of program with JPlag, Journal of Universal Computer Sciences 8 (11) (2002) 1016–1038.
[4] Y. Huang, H. Xiong, W. Wu, P. Deng, Z. Zhang, Mining maximal hyperclique pattern: a hybrid search strategy, Information Sciences 177 (3) (2007) 703–721.

[5] Y.-C. Hu, G.-H. Tzeng, C.-M. Chen, Deriving two-stage learning sequences from knowledge in fuzzy sequential pattern mining, Information Sciences 159 (1–2) (2004) 69–86.
[6] H.-L. Hu, Y.-L. Chen, Mining typical patterns from databases, Information Sciences 178 (19) (2008) 3683–3696.
[7] M.J. Wise, YAP3: improved detection of similarities in computer program and other texts, in: Proceedings of the 27th SIGCSE Technical Symposium on Computer Science Education, Philadelphia, PA, USA, 1996, pp. 130–134.
[8] D. Gitchell, N. Tran, A utility for detecting similarity in computer programs, in: Proceedings of the 30th ACM Special Interest Group on Computer Science Education Technical Symposium, New Orleans, LA, USA, 1998, pp. 266–270.
[9] S. Schleimer, D. Wilkerson, A. Aiken, Winnowing: local algorithms for document fingerprinting, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, 2003, pp. 76–85.
[10] T. Kamiya, S. Kusumoto, K. Inoue, CCFinder: a multilinguistic token-based code clone detection system for large scale source code, IEEE Transactions on Software Engineering 28 (7) (2002) 654–670.
[11] Z. Li, S. Lu, S. Myagmar, Y. Zhou, CP-Miner: A tool for finding copy-paste and related bugs in operating system code, in: Proceedings of the Sixth Symposium on Operating System Design and Implementation, 2004, pp. 289–302.
[12] I.D. Baxter, A. Yahin, L. Moura, Santapos, M. Anna, L. Bier, Clone detection using abstract syntax trees, in: Proceedings of the International Conference on Software Maintenance, 1998, pp. 368–377.
[13] J. Krinke, Identifying similar code with program dependence graphs, in: Proceedings of the Eighth Working Conference on Reverse Engineering, 2001, pp. 301–309.
[14] C. Liu, C. Chen, J. Han, P.S. Yu, GPLAG: detection of software plagiarism by program dependence graph analysis, in: Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Philadelphia, PA, USA, 2006, pp. 872–881.
[15] M. Garey, D. Johnson, Computers and Intractability: A Guide to the Theory of NP–Completeness, W.H. Freeman Co., 1979.
[16] W. Qu, M. Jiang, Y. Jia, Software reuse detection using an integrated space-logic domain model, in: Proceedings of the IEEE International Conference on Information Reuse and Integration, Las Vegas, NV, USA, 2007, pp. 638–643.
[17] P.E. Livadas, T. Johnson, An optimal algorithm for the construction of the system dependence graph, Information Sciences 125 (1–4) (2000) 99–131.
[18] J. Ferrante, K.J. Ottenstein, J.D. Warren, The program dependence graph and its use in optimization, ACM Transactions on Programming Language and Systems 9 (3) (1987) 319–349.
[19] C.-Y. Wang, S.-S. Tseng, T.-P. Hong, Flexible online association rule mining based on multidimensional pattern relations, Information Sciences 176 (12) (2006) 1752–1780.
[20] L.P. Cordella, P. Foggia, C. Sansone, M. Vento, Performance evaluation of the VF graph matching algorithm, in: Proceedings of the 10th ICIAP, IEEE Computer Society Press, 1999, pp. 1172–1177.
[21] J.R. Ullmann, An algorithm for subgraph isomorphism, Journal of the Association for Computing Machinery 23 (1976) 31–42.