Contents lists available at ScienceDirect

The Journal of Systems & Software

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

CCStokener: Fast yet accurate code clone detection with semantic token[☆]

Wenjie Wang [a,b], Zihan Deng [a,b], Yinxing Xue [a,*], Yun Xu [a,b,**]

[a] School of Computer Science and Technology, University of Science and Technology of China, Hefei 230027, China
[b] Key Laboratory on High Performance Computing of Anhui Province, Hefei 230027, China

A R T I C L E I N F O

Article history:
Received 7 June 2022
Received in revised form 2 December 2022
Accepted 15 January 2023
Available online 18 January 2023

Keywords:
Code clone detection
Semantic token
Near-miss clones
Scalable detection

A B S T R A C T

Code clone detection refers to the discovery of identical or similar code fragments in the code repository. AST-based, PDG-based, and DL-based tools can achieve good results on detecting near-miss clones (i.e., clones with small differences or gaps) by using syntax and semantic information, but they are difficult to apply to large code repositories due to high time complexity. Traditional token-based tools can rapidly detect clones by the low-cost index (i.e., low frequency or k-lines tokens) on sequential source code, but most of them have the poor capability on detecting near-miss clones because of the lack of semantic information.

In this study, we propose a fast yet accurate code clone detection tool with the semantic token, called CCStokener. The idea behind the semantic token is to enhance the detection capability of token-based tool via complementing the traditional token with semantic information such as the structural information around the token and its dependency with other tokens in form of n-gram. Specifically, we extract the type of relevant nodes in the AST path of every token and transform these types into a fixed-dimensional vector, then model its semantic information by applying n-gram on its related tokens. Meanwhile, our tool adopts and improves the location–filtration–verification process also used in CCAligner and LVMapper, during which process we build the low-cost *k-tokens* index to quickly locate the candidate code blocks and speed up detection efficiency. Our experiments show that CCStokener achieves excellent accuracy on detecting more near-miss clone pairs, which exhibits the best recall on Moderately Type-3 clones and detects more true positive clones on four java open-source projects. Moreover, CCStokener attains the best generalization and transferability compared with two DL-based tools (i.e., ASTNN, TBCCD).

1. Introduction

Software developers usually reuse some code fragments by pasting them with or without minor modifications. Code clone detection can be useful for code maintainability (Choi et al., 2011; Mockus, 2007; Thummalapenta et al., 2010), plagiarism detection (Liu et al., 2006; Cosma and Joy, 2012), bug detection (Li and Ernst, 2012; Sobrinho et al., 2021; Rahman et al., 2010), software evolution study (Bakota et al., 2007; Saha et al., 2010) and malicious software detection (Sheneamer and Kalita, 2016). Existing researches report that code clone is common in large systems, Kamiya et al. (2002) has reported 29% cloned code in JDK, Baker (1995) has detected that 22.3% of Linux code contains clones. From the perspective of reality and practicality, token-based tools are popular in large-scale clone detection. For example, SourcererCC (Sajnani et al., 2016) is able to scale up to a large inter-project repository (250MLOC).

Numerous tools have been developed for clone detection, and they differ in processing speed and detection ability. AST (Abstract Syntax Tree)-based (Yang, 1991; Jiang et al., 2007; Baxter et al., 1998), PDG (Program Dependency Graph)-based (Zou et al., 2020; Wang et al., 2017; Liu et al., 2006) tools can have good effect on detecting near-miss clones (i.e., clones with small differences or gaps (Roy and Cordy, 2007)) with syntax and semantic information, but they have complex time-consuming process (e.g., subgraph isomorphism), which makes them hard to apply to large repositories. In addition, most DL (Deep Learning)-based (Zhang et al., 2019; Yu et al., 2019; Wu et al., 2020b; Li et al., 2017) tools usually require a large-size well-labeled training dataset for attaining a reasonably good model. Besides, the feature extraction and clone pair comparison process also consumes a lot of time in large repositories. Hence, a fast yet accurate

[☆] Editor: Dr. Shane McIntosh.
[*] Corresponding author.
[**] Corresponding author at: School of Computer Science and Technology, University of Science and Technology of China, Hefei 230027, China.
E-mail addresses: wjiewang@mail.ustc.edu.cn (W. Wang), dzh16@mail.ustc.edu.cn (Z. Deng), yxxue@ustc.edu.cn (Y. Xue), xuyun@ustc.edu.cn (Y. Xu).

clone detection tool is desired, especially in a way of enhancing the traditional token-based clone detection with semantic information.

Traditional token-based tools (Li et al., 2006; Sajnani et al., 2016; Wang et al., 2018; Murakami et al., 2013) have superior scalability to detect quickly. They use a series of strategies to quickly query candidate code blocks. SourcererCC (Sajnani et al., 2016) exploits an optimized inverted index on low-frequency tokens and designs filtering heuristics to quickly detect clones. CCAligner (Wang et al., 2018) and LVMapper (Wu et al., 2020a) build k-lines index by sliding on code lines, and use the index to locate candidate blocks, which will significantly decrease the number of clone-comparisons.

However, most traditional token-based tools have the poor capability of detecting near-miss clones due to lack of semantic information. Token-based tools can be divided into two categories based on detecting granularity (i.e., *line granularity* and *word granularity*). The tools of *line granularity* (e.g., CCAligner (Wang et al., 2018)) usually normalize source code line by line, treat a line as the basic unit to detect clone, and calculate the similarity of a code pair by the coverage of common subsequences of their code lines. It is common in near-miss clones that code lines can be shuffled without affecting program functionality and several modifications may occur on lines. Nevertheless, the near-miss clones may be rejected by the *line granularity* tools after these changes. The tools of *word granularity* (e.g., SourcererCC (Sajnani et al., 2016), NIL (Nakagawa et al., 2021)) often transform a code block into a word sequence and treat a word as the basic unit to detect clone. It may not detect some near-miss clones because too many renamed identifier names lower the similarity of clone pairs. In general, pure token-based tools without syntax or semantic information can be easily cheated by near-miss clones.

In this paper, to well support detecting near-miss clones for token-based tools, we propose a fast yet accurate code clone detector with the semantic token, named CCStokener. CCStokener is a *word granularity* token-based detector, which adds semantic information to tokens, and enhances the ability to detect near-miss clones with these semantic tokens. CCStokener significantly improves the capacity of traditional token-based tools (e.g., SourcererCC, CCAligner, NIL etc.) from the following three aspects.

First, we add semantic information for tokens to enrich the features of these tokens. We consider semantic information of tokens from multiple aspects, including the structural information and the relationship of tokens (i.e., *data dependency*, *operational relation*). AST expresses the structural characteristics of a program, and its leaf node is the original token of source node. In order to describe the structural information of a token, we extract the type of relevant nodes of its path and construct them into a fixed-dimensional vector (see in Table 1). Data dependency describes the data flow of variables. Besides, variables in the same expression show a special function together, we call that *operational relation*. To describe the semantic information, we adopt *data dependency* and *operational relation* to make isolated tokens related to each other, and enhance the feature of every token by applying n-gram with its related tokens. Hence, a token can be represented as a fixed-dimensional vector, which captures the local characteristic of its program.

Second, we propose a multi-round algorithm to compute the similarity of candidate pairs, which is different from that (e.g., LCS) used in traditional token-based tools. After the first aspect, a code block can be abstracted as a collection of fixed-dimensional vectors, each of which represents an original token. Given two code blocks, we measure their similarity by computing the similarity of two vector collections. During the calculation process, we gradually reduce the threshold in each round, match

Table 1
Relevant inner nodes in AST.

ID	Node type	LANG	ID	Node type	LANG
T1	If Condition	J&C[a]	T14	Switch Body	J&C
T2	If/Else Body	J&C	T15	Switch Condition	J&C
T3	Method Definition	J&C	T16	Variable Declaration	J&C
T4	Loop Condition	J&C	T17	Assert Condition	J
T5	Loop Body	J&C	T18	Assert Body	J
T6	Array Selector	J&C	T19	Throw Body	J
T7	Logical Expn.	J&C	T20	Try Body	J
T8	Numeric Expn.	J&C	T21	Catch Body	J
T9	Condition Expn.	J&C	T22	Finally Body	J
T10	Assign Expn.	J&C	T23	Lambda Expn.	J
T11	Method Invocation	J&C	T24	Constructor Invoc	J
T12	Return Statement	J&C	T25	Class/Array Creator	J
T13	Case Body	J&C			

[a]J refers to Java language, C refers to C language.

the vectors whose cosine similarity is higher than the threshold, and finally calculate the average of the matched cosine similarities as the similarity of two vector collections.

Third, in order to avoid pair-wise comparison for code blocks in the large repository, we adopt and improve the location–filtration–verification in CCAligner (Wang et al., 2018) and LVMapper (Wu et al., 2020a). In the location step, different with CCAligner, we build a global low-cost *k-tokens* index on *Action tokens* (i.e., called tokens, data type tokens, etc.), which is inspired by Saini et al. (2018). In the filtration step, we design a series of strategies to filter dissimilar candidates, thereby preventing unnecessary time overhead in the subsequent step. In the verification step, we use the aforementioned multi-round algorithm to calculate the similarity of candidate pairs.

To evaluate, we conduct extensive experiments by comparing CCStokener with several state-of-the-art clone detection tools. We verify the effectiveness of our tool on BigCloneBench (Sajnani et al., 2016) and eight open-source projects. Meanwhile, we compare the generalization of CCStokener with DL-based tools (including ASTNN and TBCCD) on four java projects. Comparing with the state-of-the-art tools, such as CCAligner (Wang et al., 2018), Deckard (Jiang et al., 2007), NiCad (Roy and Cordy, 2008), NIL (Nakagawa et al., 2021), LVMapper (Wu et al., 2020a), SourcererCC (Sajnani et al., 2016), we achieve the best recall on detecting Type-3 and Type-4 clones, and nearly 100% recall on Type-1 and Type-2 clones. In particular, CCStokener achieves a recall of 53% on Moderately Type-3 on the BigCloneBench, and has a significant improvement on detecting java projects. In addition, CCStokener has better generalization and transferability than ASTNN and TBCCD. We put our tool and experimental results on the website.[1]

Our main contributions are listed as follows:

(1) We propose a novel semantic token representation, which can describe the structural information and data dependency, and we apply n-gram on related tokens to capture local characteristics of the program.

(2) We borrow the idea of the location–filtration–verification process in CCAligner and LVMapper, and adapt to our *word granularity* token-based detector to achieve the effect of fast detecting and scale to a large repository.

(3) We empirically compare CCStokener with the state-of-the-art token-based and DL-based tools, such as CCAligner, NIL, ASTNN, TBCCD, etc. Results show that CCStokener is significantly superior to other token-based tools in detection accuracy and has better generalization compared with DL-based tools.

[1] https://github.com/CCStokener/CCStokener

2. Background

2.1. Terminology

Code fragment is usually a continuous segment of source code, it can contain a function, *begin-end* blocks or a sequence of statements (Sheneamer and Kalita, 2016). *Code block* is a function of a program (Roy and Cordy, 2007). *Clone pair* is a pair of code fragments (Sheneamer and Kalita, 2016), which is syntactically or semantically similar with each other. According to the degree of similarity, code clone generally falls into four types (Roy and Cordy, 2007; Sheneamer and Kalita, 2016; Saini et al., 2018; Rattan et al., 2013):

- **Type-1 (textual similarity)**: syntactically identical code fragments, except for differences in white space, comments and layout.
- **Type-2 (lexical similarity)**: in addition to Type-1 clone differences, syntactically identical code fragments, except for differences in identifier names and literal values. Type-2 clone reflects the differences in lexical tokens.
- **Type-3 (syntactic similarity)**: also known as near-miss clone or gap clones. In addition to Type-1 and Type-2 clone differences, syntactically similar code fragments differ at the statement level.
- **Type-4 (semantic similarity)**: code fragments may be dissimilar in syntax structures, but perform the same task.

Sajnani et al. (2016) further divided Type-3 and Type-4 into four fine-grained types: Very Strongly Type-3 (VST3) for similarity range in [0.9, 1.0), Strongly Type-3 (ST3) for similarity range in [0.7, 0.9), Moderately Type-3 (MT3) for similarity range in [0.5, 0.7), and Weakly Type-3&Type-4 (WT3/T4) for similarity range in [0.0, 0.5).

2.2. Challenges in detect near-miss clones

Near-miss clones have many characteristics that make them difficult to be detected. Software developers usually copy a code fragment, and make some modifications on lines or add some new lines to implement new functionality. In addition, code obfuscation can happen in the form of line shuffle if the data dependency of the program does not change. For example, the position of *if-structure* and *else-structure* can be swapped with each other because the swapping operation will not change functionality of the program. Furthermore, near-miss clones commonly appear in current large-scale software systems, which requires a relatively high efficiency of the clone detection tools.

Traditional token-based (Wang et al., 2018; Sajnani et al., 2016; Roy and Cordy, 2008; Wu et al., 2020a) clone detection tools can use the advantage of sequences to rapidly detect clones, and they are easily scaling up to large repositories, *but most tools cannot properly detect near-miss clones because of lack of semantic information*. Token-based tools can be divided into two categories based on detecting granularity, including *word granularity* (e.g., SourcererCC (Sajnani et al., 2016)) and *line granularity* (e.g., CCAligner (Wang et al., 2018), NiCad (Roy and Cordy, 2008)). *Word granularity* tools treat a token as the basic unit to detect clones. SourcererCC (Sajnani et al., 2016) transforms source code into token sequence. Then it verifies clones by the ratio of the same tokens. *Line granularity* tools consider a normalized line as the basic unit to detect clones. CCAligner (Wang et al., 2018) designs a novel e-mismatch index by sliding code windows on lines to detect large-gap clones. NiCad (Roy and Cordy, 2008) computes length of the longest common sequence on normalized source code to get clones, which makes it unable to detect obfuscated clones well. Most traditional token-based tools detect clones on lexical source code, but they do not take into account structural or semantic information, thereby resulting in poor detection effect on the near-miss clones.

AST-based (Jiang et al., 2007; Baxter et al., 1998; Gao et al., 2019), PDG-based (Wang et al., 2017; Zou et al., 2020; Liu et al., 2006; Higo et al., 2011) and DL-based (White et al., 2016; Zhang et al., 2019; Yu et al., 2019; Wu et al., 2020b) tools utilize the structural and semantic information of source code, and achieve good effect on detecting near-miss clone, *but they have the problem of excessive detection time and high difficulty to scale to large repositories*. AST and PDG of programs contain high-dimensional features, which rarely are influenced by several modifications on code lines. Deckard (Jiang et al., 2007) transforms AST into vectors, and calculates the distance of vectors to approximate the editing distance of trees. CCGraph (Zou et al., 2020) approximates the similarity of PDG by WL (Weisfeiler–Lehman) graph kernel to decrease the complexity of subgraph isomorphism, but it still spends 2 h on detecting 10M LOC (lines of code) dataset. DL-based tools have surprising precision and recall on detecting near-miss clone, ASTNN (Zhang et al., 2019) achieves the effect of 99.8% (precision) and 88.4% (recall) on BigCloneBench (Sajnani et al., 2016) labeled pairs. But their generalization ability may get hurt when the testing dataset is different from the training dataset (Yu et al., 2019). In addition, they need a long time of training that makes them hard to apply to large code repositories.

Witnessing the limitations of the aforementioned various clone detection tools, it is desired to have an approach, which suits detection on unlabeled datasets and scales up to large-size repositories. We intend to improve the detection ability on near-miss clones of token-based tools. Meanwhile, it needs to maintain the scalability of the token-based tools while ensuring accuracy.

3. Motivation

We first present a motivation example that cannot be detected by most token-based tools and explain the reason, then we describe our observation to detect such clone pairs.

3.1. A motivating example

In this section, we show an example that traditional token-based tools cannot detect. Fig. 1(a) and 1(b) are two solutions that use MD5 algorithm to compress a string in BigCloneBench (Svajlenko and Roy, 2016). They are labeled as a Moderately Type-3 clone, and the tagged token similarity is 0.70, and line similarity is 0.52. Fig. 1(a) compresses parameter string in the default way. Fig. 1(b) compresses string with the custom seed and has more operations for capturing exception.

Line granularity tools (e.g., NiCad, CCAligner, LVMapper) cannot detect this clone pair, due to their limitation of requiring the normalized clone lines to be identical to each other, but some clone lines contain fine-grained modification (e.g. insert or delete some tokens). As line 4 of Fig. 1(a) and line 5 of Fig. 1(b) show, they both pass parameters to the MD5 object, and are clones of each other because of similar functionality. But, the two lines cannot be considered as a clone by *line granularity* tools. *Line granularity* tools are sensitive to modification on lines making them hard to detect near-miss clones.

Word granularity tools (e.g., SourcererCC (Sajnani et al., 2016), NIL (Nakagawa et al., 2021)) also fail to detect this clone. The reason is that they measure similarity by calculating the coverage of the original tokens, but the similarity will decrease sharply when there are too many renamed identifiers, which results in the clone pairs cannot being detected. As Figs. 1(a) and 1(b) show, the MD5 object is named *md5Algorithm* in Fig. 1(a), but is named *digest* in Fig. 1(b). So, it is necessary to find a way to abstract the original tokens with more semantic information, and perform similarity analysis on these semantic tokens.

```
1   private String getMD5(String data) {
2       try {
3           MessageDigest md5Algorithm =
                MessageDigest.getInstance("MD5");
4           md5Algorithm.update(data.getBytes(), 0,
                data.length());
5           byte[] digest = md5Algorithm.digest();
6           StringBuffer hexString = new StringBuffer();
7           String hexDigit = null;
8           for (int i = 0; i < digest.length; i++) {
9               hexDigit = Integer.toHexString(0xFF &
                    digest[i]);
10              if (hexDigit.length() < 2) {
11                  hexDigit = "0" + hexDigit;
12              }
13              hexString.append(hexDigit);
14          }
15          return hexString.toString();
16      } catch (NoSuchAlgorithmException ne) {
17          return data;
18      }
19  }
```

(a) A code block in 3/selected/617455.java

```
1   private static String encryptMD5(String password,
        Long digestSeed) {
2       try {
3           MessageDigest digest =
                MessageDigest.getInstance("MD5");
4           digest.reset();
5           digest.update(password.getBytes("UTF-8"));
6           digest.update(digestSeed.toString().getBytes("UTF-8"));
7           byte[] messageDigest = digest.digest();
8           StringBuffer hexString = new StringBuffer();
9           for (int i = 0; i < messageDigest.length; i++) {
10              hexString.append(Integer.toHexString(0xff &
                    messageDigest[i]));
11          }
12          return hexString.toString();
13      } catch (NoSuchAlgorithmException e) {
14          throw new RuntimeException(e);
15      } catch (UnsupportedEncodingException e) {
16          throw new RuntimeException(e);
17      } catch (NullPointerException e) {
18          return new StringBuffer().toString();
19      }
20  }
```

(b) A code block in 3/selected/1489138.java

Fig. 1. A Moderately Type-3 clone pair in BigCloneBench (Svajlenko and Roy, 2016).

3.2. Observations to detect near-miss clones

According to the characteristics of near-miss clones described in Section 2.2 and its requirements for the detection tools, we have summarized the following three observations.

1. AST path is a way to extract structural information of tokens. AST is much like a blueprint for a program, i.e., the inner nodes of AST build the skeleton, and the leaf nodes fill the facade. The tokens that perform the same function are likely to share a highly similar AST path, *so it is optional that we may measure the similarity of tokens by the similarity of their AST paths.* Variable *digest* in line 5 of Fig. 1(a) and variable *messageDigest* in line 8 of Fig. 1(b) have same AST path, the AST path of variable *digest* is shown in Fig. 3. In addition, equivalence relation exists in the inner nodes, for example, *for-loop*, *while-loop* and *do-while-loop* structures can convert into each other.

However, it is not enough to abstract tokens with structural information. The tokens that share similar path may not perform the same function. The main reason is that the path information is relatively simple, it is necessary to add other information to enrich the feature of tokens.

2. The token relationship can make the isolated token related to each other, which will enrich token information. The path describes the structural information of the tokens themselves, which is not enough to make them distinguish with each other. The token relationships (e.g., data dependency) make the isolated tokens related to each other in the form of a relationship graph. *There are many similar substructures in near-miss clones, so we can let a token capture the local structure in the relationship graph to enhance their semantic information.* Specifically, we can apply n-gram to model each token on the graph, and let the token capture the information of the adjacent substructure of the graph.

3. Building a low-cost index can achieve the purpose of quickly detecting clones. Most token-based tools (e.g., SourcererCC, CCAligner, LVMapper) build a low-cost index to quickly get the candidate code blocks for a given block, which will avoid pairwise comparison for large repositories. CCAligner and LVMapper utilize the location–filtration–verification process to speed up detection. We can borrow the idea and build a low-cost index on tokens. Inspired by Oreo (Saini et al., 2018): *"If two methods perform the same function, they likely call the same library methods and refer the same object attributes"*. In other words, if two functions have a clone relationship, they may have the same token because they have the same API call. We can analyze such tokens that are mostly unchanged during cloning and build an inverted index on these tokens.

4. Methodology

4.1. System overview

Fig. 2 shows the overview of CCStokener, which consists of two stages, i.e., semantic token generation and scalable clone detection. Steps 1–3 are the semantic token generation stage, and steps 4–7 are the scalable clone detection stage. We extract code blocks from source code in step 1 and generate an AST for each code block in step 2. In step 3, we convert every relevant token into a fixed-dimensional vector, which carries the structure and semantic information of the token, and we extract the *Action tokens* of each block. Specifically, we differentiate the role types of tokens according to grammatical rules to get *Action tokens* (Section 4.2.1), and represent structural information of token by the type of AST path (Section 4.2.2), and finally analyze semantic information from token relationship graph (Section 4.2.3).

During scalable clone detection stage, we use the location–filter–verification process, and apply different location strategies to code blocks that contain enough or few *Action tokens*. In step 4, we build a global *k-tokens* inverted index for all code blocks that have enough *Action tokens* (Section 4.3.1). In step 5, for the code blocks with enough *Action tokens*, we look up the *k-tokens* index to locate (Section 4.3.2). For code blocks with few *Action tokens*, we select code blocks with a similar total number of tokens as candidates. In step 6, we filter fake candidates based on the number of total tokens and the overlap of *Action tokens* (Section 4.3.3). In step 7, we use our multi-round algorithm to verify the similarity between this target code block and its candidate code blocks with semantic tokens, thereby obtaining clone pairs (Section 4.3.4).

4.2. Semantic token generation

Semantic token generation stage is the pre-process of clone detection. In this stage, we will obtain the *Action tokens* and assign semantic information to variable tokens, callee tokens, and operator tokens for every code block. The *Action tokens* will be used to

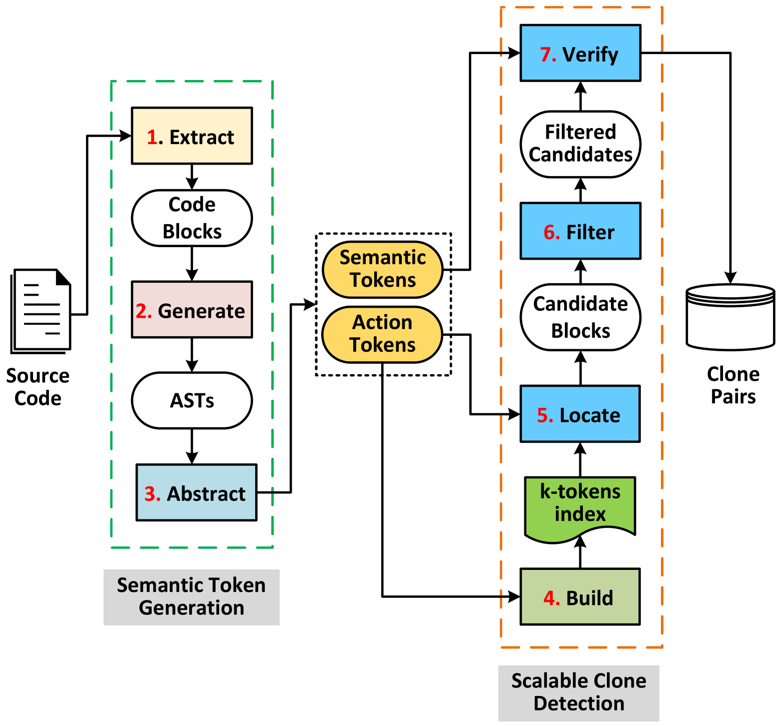

Fig. 2. The overview of CCStokener.

build a global index to quickly query candidate clone blocks (Section 4.3.1). The semantic tokens (i.e., variable tokens, callee tokens, operator tokens) will be used to determine the authenticity of candidate clone pairs in the verification step (Section 4.3.4).

4.2.1. Analysis of token role and Action tokens

Token is the lexical unit that constitutes a method. There are many roles for tokens, including keyword, identifier, constant, operator, and punctuation. Identifier role can be further divided into variable, callee, data type, and so on. The methods that perform similar functionality are likely to call the same methods, have the same class objects, and use some global classes. For example, Fig. 1(a) and 1(b) have same tokens: *MessageDigest*, *update()*, *getBytes()*, *StringBuffer* etc. We call callee tokens, data type tokens, and global class tokens as *Action tokens*. In addition, we only assign semantic information to variable tokens, callee tokens, and operator tokens, because they are related to the functionality of the program. Other roles, such as constant, keyword, etc., are more likely used to decorate the program, and we ignore them.

```
VariableDeclaration := VariableModifier∗ Type DeclaratorList;
VariableModifier := 'final' | Annotation
Type := BasicType | ReferType
ReferType := Identifier
BasicType := 'float' | 'int' | 'double' | 'short' | 'long' | 'char' | 'byte'
        | 'boolean'
DeclaratorList := VariableDeclarator (',' VariableDeclarator)∗
VariableDeclarator := VariableDeclaratorId ('=' VariableInitializer)?
VariableDeclaratorId := Identifier dims?
```

Listing 1: "VariableDeclaration" grammar rule in *Javalang*

We identify the token role by the grammatical rules that are used to generate the AST. Listing 1 is an example of "VariableDeclaration" grammar rule in *Javalang* (Anon, 2012). Every line in the rule consists of two parts, the part before ":=" is rule name, another part after ":=" is component unit of that rule. From the rule, we can know that variable declaration contains three parts: variable modifier, variable type, and variable declarator list, corresponding to *VariableModifier*, *Type*, *DeclaratorList* in Listing 1, respectively. For example, in statement *"StringBuffer hexString = new StringBuffer()"* (line 6 of Fig. 1(a)). *StringBuffer* belongs to *ReferType* grammar, so its role is data type. *hexString* belongs to *Identifier* in *VariableDeclaratorId* grammar, so its role is variable. *"new StringBuffer()"* belongs to *VariableInitializer* grammar, and it needs further parsing.

4.2.2. Analysis of syntactic structure

Variables with the same function may have different names in near-miss clones, so we need to judge the similarity of tokens from other angles rather than their literal, which leads to the demand to abstract the token. AST describes the syntactic structure of a program. The inner nodes in AST present the structural characteristic of a program, and the leaf nodes are the original tokens in a program, as Fig. 3 shows. The tokens that perform the same functionality are likely to share a highly similar AST path. We use the AST path to abstract the structural information of tokens in the leaf node.

Specifically, we perform a depth-first traversal of the AST tree and record the types of relevant nodes in the path. Then, we convert them to a fixed-dimensional vector. Each dimension in the vector represents a kind of node type. The value of each dimension of the vector represents the number of times the node appears in the path. For example, variable *m5Algorithm* can be represented as 25-dim vector: [0,0,1,0,0, ...,1,0,1,...,1, ...,0], the value of the 3rd, 11th, 13th, and 20th dimensions is 1, and the others are 0. Hence, a token can be represented as a fixed-dimensional vector, called *structural vector*.

We analyze Java language (with *Javalang* (Anon, 2012)) and C language (with *ANTLR* (Anon, 2014)), and get the types of relevant inner nodes, as Table 1 shows. We can see that some inner nodes exist in Java but do not exist in C. It is worth noting that we have made equivalent conversions to *for-structure*, *while-structure* and *do-while-structure*, and transform them into types *Loop Condition* (T4) and *Loop Body* (T5). As Fig. 3 shows, we transform the *for-body* of line 9 in Fig. 1(a) into *Loop Body* (T5) type.

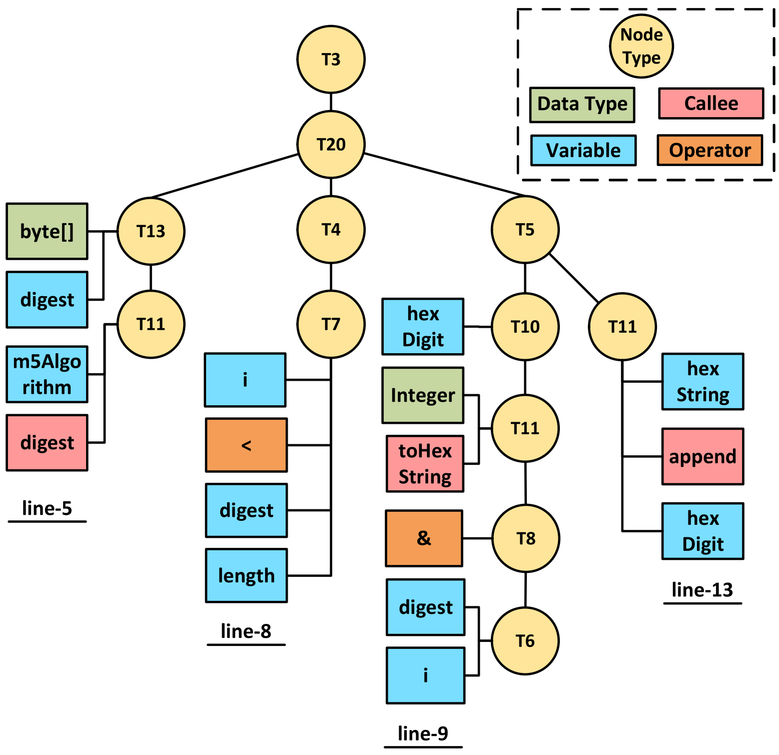

Fig. 3. Part of the AST tree structure of function getMD5() in BigCloneBench source file 3/selected/617455.java; it describes the structure of lines 5, 8, 9, and 13.

Moreover, we divide expression types into three categories, i.e., *Logical Expression* (e.g., $>$, $<$, $==$, $!=$, etc.), *Numeric Expression* (e.g., $+$, $*$, $\gg$, &, etc.) and *Condition Expression* (e.g., &&, ||, etc.). The expression *"i<digest.length"* in line 8 of Fig. 1(a) is represented as *LogicExpression* (T7) type in Fig. 3.

4.2.3. Analysis of token relationship

The path information in AST is not enough to abstract the original token, because tokens with different function may have a similar path. The token relationship organizes the isolated tokens into a graph, and we let each token capture the substructure of the graph to enrich its semantic information. We divide the token relationship more finely into three patterns: *data dependency* for *variable–variable*, *operational relation* for *variable–variable*, *operational relation* for *callee–variable*. Here, we elaborate the three patterns as follows.

Pattern 1: *data dependency* for *variable–variable*. This pattern depicts the data dependency between variables. We mainly analyze the flow dependency, which is the assignment behavior in the expression during the forward execution flow. For example, line 9 in Fig. 1(a), variable *hexDigit* depends on *digest* and *i*, as Fig. 4(a) shows. The data dependencies make isolated variables form an association graph.

We enrich the semantic information of a variable by applying n-gram on its related variables. Specifically, a variable n-gram is a collection that contains all variables reachable in n steps prior to it. We sum the *structural vector* of all variables in the n-gram to get the *semantic vector*. Hence, a variable is represented as a *variable semantic vector*, which can capture the local data flow behavior of its program. For example, variable *hexDigit* in Fig. 4(a) shows its data dependencies till line 11 in Fig. 1(a). The 3-gram *semantic vector* of variable *hexDigit* is the sum of *structural vector* of variables *hexDigit, digest, i* and *m5Algorithm*.

Pattern 2: *operational relation* for *variable–variable*. This pattern describes the relationship between the variables in expressions. There is no data dependency between the variables in the expression, but they collectively reflect a function, and this relationship is reflected on the operator. Hence, the semantic information of the operator can be obtained from the operand in the expression. For example, variable *i* and *digest* in *i<digest.length* (line 8 in Fig. 1(a)) show comparison function together, and the *semantic vector* of operator $<$ is the sum of *semantic vector* of variable *i* and *digest*.

Pattern 3: *operational relation* for *callee–variable*. This pattern describes the relationship between callee token and variable parameters. The intuition is that if two callee tokens perform the same function, they are more likely to have a similar argument list. For every callee token, we add up the *semantic vector* of all variables in argument list and callee's *structural vector* to get a *callee semantic vector*. For example, if we set 3-gram for variable relationship, callee token *append* of Fig. 4(a) captures the information of variable *hexString, hexDigit, digest, i, m5Algorithm*. Callee token *append* of Fig. 4(b) captures information of variable *hexString, messageDigest, i, digest*. These two tokens can be considered similar because of similar substructure.

4.3. Scalable clone detection

In the previous stage, we extract some *Action tokens* and three vector collections (i.e., *variable vector collection*, *operator vector collection* and *callee vector collection*) from a code block.

At the current stage, we borrow the idea of the location–filtration–verification process used in CCAligner and LVMapper and adapt the process to our *word granularity* tool. Algorithm 1 describes the overall process. Moreover, in verification step (Section 4.3.4), we design a multi-round algorithm to calculate the similarity of two vector collections, which will reduce the time complexity from $O(N!)$ to $O(k * N^2)$.

4.3.1. Build inverted index

To prevent pair-wise comparisons on code blocks, it is necessary to find a way to quickly select candidate clone blocks for

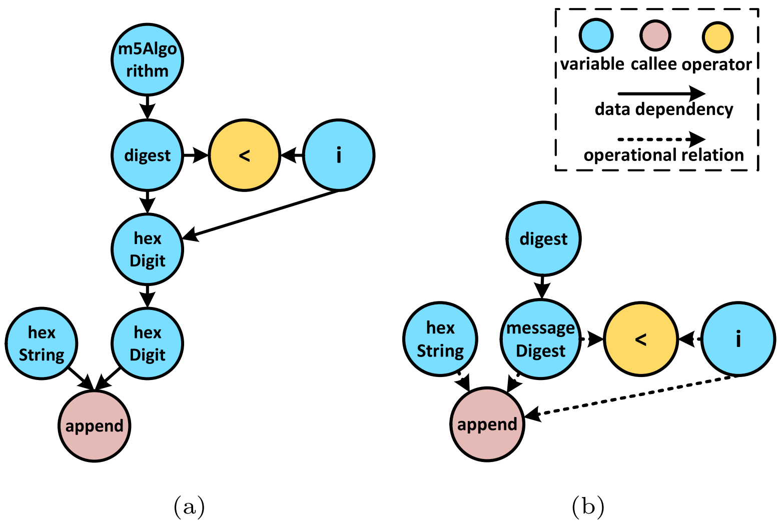

Fig. 4. Token relationship graph. (a) Partial token relation of getMD5(), 3/selected/617455.java; (b) Partial token relation of encryptMD5(), 3/selected/1489138.java.

a given code block. CCAligner (Wang et al., 2018) and LVMapper (Wu et al., 2020a) build *k-lines* index on code blocks, which is not suitable for our *word granularity* tool. Instead, we build the low-cost inverted index on *Action tokens*.

Here, we describe in detail how to build a global *k-tokens* index, corresponding to line 2 of Algorithm 1. First, we sort the *Action tokens* of each block in lexicographic order. Second, for every block, we slide on the sorted *Action tokens* with a window size of k to get a sequence of *k-tokens*. Third, we consider the pair of hash value of *k-tokens* and the block itself as an item, and put it into the global inverted index, whose key is the *k-tokens* hash value, value is a block array. For example, the *Action tokens* of block1 are *A,B,C,D,E*, then its 3-tokens are *A,B,C*, *B,C,D*, *C,D,E*. One item of the inverted index is *hash(A,B,C): [block1]*.

4.3.2. Locate candidate blocks

In location step, we choose the code blocks without missing the real clone for the target block. Generally, if two functions are clones, they will have certain duplicate *Action tokens*, so we can query candidate code blocks through *Action tokens*. However, some user-defined functions in C language do not have API calls, which makes the location strategy by *Action tokens* invalid. Therefore, we divide the code blocks into two parts according to the number of *Action tokens*, and design location strategies for these two parts respectively.

For the blocks with enough *Action tokens*, we select the candidate clone blocks for the target code block with *k-tokens* index. Lines 5–10 of Algorithm 1 are the location step. First, we generate *k-tokens* on the sorted *Action tokens*, m-k+1 *k-tokens* will be generated from a token sequence with length m (lines 5–6). Next, we use the hash value of *k-tokens* as a key to look up the global *k-tokens* index (line 7). After m-k+1 queries, we will get a series of array that contains candidate blocks (line 8). Finally, we merge the result arrays and use it in the filtration step (line 9).

For the blocks with few or no *Action tokens*, we use their total number of tokens as the principle of locating. First, we sort the code blocks in ascending order according to the total number of tokens. Next, for the target code block, we choose the code blocks whose order is followed and the total number of tokens differs with a limited range as candidate clones.

4.3.3. Filter candidate blocks

There are many false candidates in the candidate code blocks after location step, so we need to remove false candidates as much as possible in the filtration step to reduce the time overhead of the subsequent step. Our filtering criteria includes two aspects: the overlap of same *Action tokens* and the difference of total token number. The more similar the clone pair, the more same *Action tokens* they will have. We ignore the candidate blocks that share fewer *Action tokens* with the target block. Moreover, if the number of total tokens in the two code blocks is differing too much, they are probably not a clone.

Lines 11–21 in Algorithm 1 are our filtration step. For every candidate block (line 13), we first count the number of *Action tokens* that are same as that in target block (line 14), then compute the overlap of same *Action tokens* (line 15). Next, we calculate the ratio of total token number of two blocks (lines 16–17). Finally, we keep the code blocks whose overlap of same *Action tokens* greater than β and the token ratio greater than θ as candidates (lines 18–20).

4.3.4. Verify clone pairs

In the verification step, we measure the similarity of candidate pairs to determine whether they are clones. Unlike SourcererCC (Sajnani et al., 2016), which calculates the similarity of original tokens, we abstract tokens with semantic information and convert them into three vector collections (Section 4.2.3), including *variable vector collection*, *operator vector collection* and *callee vector collection*. Calculating tokens with rich semantic information can make the judgment of near-miss clones more accurate.

Now we need to calculate the similarity of candidate pairs, and we describe the process in lines 22–29 of Algorithm 1. Given a candidate pair, we calculate the similarity for the three collections respectively (lines 23–25), use the average of the three similarities as the similarity of pair (line 26), and treat it as a clone pair if its similarity is greater than η (line 26–27).

However, we face the problem of time complexity of $O(N!)$ when calculating the best similarity of two vector collections. For example, supposing the size of vector collection for block *A* and block *B* is *N*, we need to generate *N!* full permutations for the n items of A, and get the best permutation corresponding to the vectors of *B*. The time complexity of *O(N!)* is intolerable to us, so we design a multi-round matching approximation algorithm to reduce the time complexity to $O(k * N^2)$.

We implement the multi-round matching algorithm as Algorithm 2. Given two vector collections, in the calculation process, we set a threshold value and gradually reduce the threshold in multiple rounds (line 4). In each round, we ignore the matched

```
Algorithm 1: Clone Detection
  Input  : $CM$ is a list of code block with more tokens $\{c_1, c_2, ..., c_n\}$. $AT$ is Action tokens of blocks $c_1$-$c_n$. $VT$, $ET$, $CT$ are semantic token collections of $c_1$-$c_n$, which are variable tokens, expression tokens, callee tokens respectively. $\beta$, $\theta$, $\eta$, $\phi$ are thresholds.
  Output: All clone pairs $CP$.
1  $CP \leftarrow \emptyset$;
   // build a global k-tokens index on Action tokens
2  $kIndex$ = buildKTokenIndex($AT$) ;
3  foreach block $c_j$ in $CM$ do
4      $CC \leftarrow \emptyset$ ;                          // candidate block set
       /* location step:                                           */
5      for $t \leftarrow 1$ to $|AT_j| - k + 1$ do
           // $AT_j[t]$ is t-th token in $AT_j$
6          $kTokens \leftarrow$ concat($AT_j[t], AT_j[t+1], ..., AT_j[t+k-1]$) ;
7          $key \leftarrow$ hash($kTokens$) ;
           // get is a function that returns a list of values searched from the kIndex with a given key
8          $CC_t \leftarrow$ get($kIndex, key$) ;
9          $CC \leftarrow CC \cup CC_t$ ;
10     end
       /* filtration step:                                         */
11     $FC \leftarrow \emptyset$ ;  // candidate blocks after filtering
12     $tt_j \leftarrow$ countTokenNum($c_j$) ;       // count the total token number of $c_j$
13     foreach candidate block $c_k$ in $CC$ do
14         $sat \leftarrow$ countSameAction($AT_j, AT_k$) ;      // count the number of same Action tokens of $c_k$
15         $ato \leftarrow sat/\min(|AT_j|, |AT_k|)$
           $tt_k \leftarrow$ countTokenNum($c_k$) ;
16         $tr \leftarrow \min(tt_j, tt_k)/\max(tt_j, tt_k)$ ;    // token ratio
17         if $ato > \beta$ and $tr > \theta$ then
18             $FC \leftarrow FC \cup c_k$ ;
19         end
20     end
       /* verification step                                        */
21     foreach candidate block $c_k$ in $FC$ do
22         $simVT \leftarrow$ verifySim($VT_j, VT_k, \phi$) ;
23         $simET \leftarrow$ verifySim($ET_j, ET_k, \phi$) ;
24         $simCT \leftarrow$ verifySim($CT_j, CT_k, \phi$) ;
25         if $(simVT + simET + simCT)/3 > \eta$ then
26             $CP \leftarrow CP \cup (c_j, fc_k)$ ;
27         end
28     end
29 end
30 return $CP$ ;
```

vectors (line 6, 8), match two vectors whose cosine similarity is higher than the threshold (line 9–14), and finally, we calculate the average of the matching similarities as the similarity of two vector collections (line 18). There are two loops of traversal in each round, making the time complexity $O(N^2)$. Assuming there are k rounds in total, the final time complexity is $O(k * N^2)$. The more rounds, the closer the approximate result is to the real similarity, but at the same time the amount of calculation will increase. In our experience, we set the decrease step ϕ as -0.1.

```
Algorithm 2: verifySim
  Input  : $P$ is a vector collection, $Q$ is another vector collection, $\phi$ is the decrease step of threshold (it is a negative number).
  Output: Similarity of the two vector collection.
1  $totalSim \leftarrow 0$ ;
   // array $MP$,$MQ$ are used to mark whether the vectors at the corresponding position have been matched
2  initialize array $MP$ to size $|P|$ with default value 0. ;
3  initialize array $MQ$ to size $|Q|$ with default value 0. ;
4  for $t \leftarrow 1.0$ to $0.0$ step $\phi$ do
5      for $i \leftarrow 0$ to $|P|$ do
           // skip i-th vector in $P$ if it has been matched
6          if $MP[i] == 1$ then continue ;
7          for $j \leftarrow 0$ to $|Q|$ do
               // skip j-th vector in $Q$ if it has been matched
8              if $MQ[j] == 1$ then continue ;
               // calculate cosine similarity of two vectors
9              $sim \leftarrow$ cosine($P[i], Q[j]$) ;
               // match i-th and j-th vector of $P$, $Q$, respectively
10             if $sim >= t$ then
11                 $totalSim \leftarrow totalSim + sim$ ;
12                 $MP[i] \leftarrow 1$ ;
13                 $MQ[j] \leftarrow 1$ ;
14             end
15         end
16     end
17 end
18 return $totalSim/\max(|P|, |Q|)$ ;
```

5. Evaluation

We evaluate the effectiveness of our tool by answering the following four questions.

RQ1. How to set the filtration thresholds β and θ, the verification threshold η to achieve better detection ability.

RQ2. How *effective* is CCStokener, when compared with traditional clone detection tools?

RQ3. How *effective* is CCStokener, when compared with DL-based tools?

RQ4. How *scalable* is CCStokener, when compared with the traditional token-based tools?

RQ5. How are the acceleration effect of the location–filtration process of CCStokener and the effectiveness of the verification algorithm?

5.1. Experimental setup

5.1.1. Experimental datasets

Our experiments are performed on BigCloneBench (Sajnani et al., 2016) and eight open-source projects, including four java projects (i.e., Ant-1.10.1, JDK-1.2.2, Maven-3.5.0, OpenNLP-1.8.1) and four C projects (i.e., Cook-2.34, Linux-1.0, PostgreSQL-6.0, Redis-4.0.0).

BigCloneBench (BCB), released by Sajnani et al. (2016), is built by mining clones from IJaDataset-2.0(Svajlenko and Roy, 2016), which consists of 43 subfolders (or functionalities), about 3 million source files and 250 million lines of code. BigCloneBench contains 8 million labeled true clone pairs and 279 thousand labeled false clone pairs, and it further divides the labeled true clone pairs into T1, T2, VST3, ST3, MT3, and WT3/T4 clones. However, with the huge amount of code segments, it is impossible to label all positive clone pairs. CCStokener and other tools report lots of true clone pairs excluded from the labeled data, which causes a problem in computing precision and recall. If we consider the overlap of reported and BigCloneBench labeled true pairs as true positive to calculate precision, it will falsely reduce precision. So we randomly select 400 clone pairs from the results and manually verify the authenticity. We will describe our verification strategy in Section 5.1.2 **RQ2**.

The eight open-source projects mentioned above are famous and widely used across operating system, programming language, database, and system management. In addition, CCAligner (Wang et al., 2018), LVMapper (Wu et al., 2020a) and NIL (Nakagawa et al., 2021) perform clone detection on these projects to compare the detection ability against other tools. In order to reflect the difference of detection ability in real projects, we also use them to compare CCStokener with other traditional tools and DL-based tools.

In respect of scalability, we extract 1M, 10M, 20M, 30M, 250M LOC datasets from IJaDataset-2.0, and compare the detection time of CCStokener with the state-of-the-art token-based tools on these five datasets. We use cloc[2] to count the lines of each dataset.

5.1.2. Experimental settings

We use the *ANTLR* (Anon, 2014) and *Javalang* (Anon, 2012) tools to obtain ASTs for C and Java codes respectively, then analyze their grammar files and add semantic information to the tokens according to the method described in Section 4.2. We conduct our experiments on an Ubuntu 18.04 desktop with Intel(R) Xeon(R) Gold 5120 CPU @ 2.20 GHz, NVIDIA GeForce GTX 1080 Ti and 503 GB Memory, and we limit the memory usage to 20 GB for each tool.

To answer the proposed questions, we design experiments for them separately.

(1) RQ1: There are three parameters in CCStokener that need to be set, i.e., the filtration thresholds β and θ, the similarity threshold η. The filtration thresholds β and θ influence the number of pairs that are computed similarity in the verification step. The larger the filtration thresholds, the smaller number of pairs in the subsequent step, but the more likely the positive clone pairs are to be filtered. Moreover, the similarity threshold will affect final precision and recall. The larger the similarity threshold, the more false negative clone pairs, resulting in the lower recall. The setting of these thresholds requires a trade-off.

During the filtration step, there are two thresholds. θ controls the ratio of the total number of tokens of a candidate pair, and β controls the ratio of the number of same *Action tokens* of a candidate pair. We use the labeled true and false clone pairs in BigCloneBench to determine the value of these thresholds. Specifically, we set a list of θ and β, then observe how many pairs of true and false clones can be reserved under a certain θ or β, and finally choose a value that is balanced. It is worth noting that we use the labeled true pairs except WT3/T4 clones, because the similarity of WT3/T4 clones is lower than 0.5 and they will mislead the configuration of filtration parameters.

In terms of similarity threshold, we also use the labeled true and false clone pairs in BigCloneBench. Specifically, firstly we set a list of η. Second, use our validation algorithm to compute the similarity of a true or false clone pair. Third, under a certain η, count the proportion of the reported true clones in total true clones and the proportion of the reported false clones to total false clones. Finally, we choose a proper η that can balance the two factors.

Table 2
Configurations of clone detection tools.

CCAligner	Min length 6 lines, window size q=6, edit distance e=1, min 60% similarity.
Deckard	Min length 50 tokens, 85% similarity, 2 token stride.
LVMapper	Min length 6 lines, window size k=3, filtering threshold 0.1, verification threshold is variable.
NiCad	Min length 6 lines, blind identifier normalization, identifier abstraction, min 70% similarity.
NIL	Min length 6 lines, 5-grams, filtration threshold 0.1, verification threshold 0.7.
SourcererCC	Min length 6 lines, min similarity 70%.

(2) RQ2: In order to evaluate the detection ability of CCStokener and other clone detection tools, we design experiments on the widely used benchmark (i.e., BigCloneBench) and realistic projects (i.e., eight open-source projects), respectively. We use CCAligner, Deckard, LVMapper, NiCad, NIL, and SourcererCC as the comparison tools.

BigCloneBench is a popular Java benchmark that reports recall automatically. However, when calculating precision, since there are plenty of true clone pairs not labeled by BigCloneBench, it is necessary to manually verify the authenticity of clone pairs. At the same time, due to the large quantity of the reported clone pairs (reaches tens of thousands), we need to sample some clone pairs and approximate the precision of the sample as the overall precision. Specifically, we randomly select 400 clone pairs from the result of each tool. Three researchers validate all of these 400 clonal pairs. For the unclear pairs, we discuss together to decide the authenticity. When judging whether a clone pair is true or false, we inspect from three aspects: (a) textural similarity, (b) structural similarity, (c) functionality similarity. For clone pairs that overlap more than 70% in textural or structural perspective, we consider them as true clone pairs. For the clone pairs overlap more than 40% and less than 70% in textural or structural perspective, and they do the same work, such as copy file, we consider them as true pairs. Otherwise, we consider them as false clone pairs. In actual, most of the indistinguishable clone pairs are belonging to MT3 and WT3/T4 clone pairs that have a lower similarity. The proportion of the indistinguishable pairs is about 10%.

We set up two experiments on BigCloneBench to compare the detection ability of CCStokener and other tools. In the first experiment, we use the recommended configurations of the comparison tools in their literature, see Table 2. For CCStokener, we use the configurations from RQ1. Then we perform clone detection to compare the precision and recall of CCStokener with other tools. In the second experiment, since the similarity thresholds of these tools are different, to identify the impact of the threshold on the tool's results, we set a list of thresholds from 0.5 to 0.7, and compare the precision and recall of these tools under each threshold.

In order to evaluate the detection ability of CCStokener and other clone detection tools on realistic projects, we perform clone detection on eight open-source projects. Considering these eight projects are not labeled, we cannot calculate their recalls. Therefore, we compare the total number of clone pairs detected by different tools and their precision. As we do to calculate precision on BigCloneBench, we randomly select 400 clone pairs from the results, manually verify the authenticity by three researchers, and finally calculate the precision. For the tools that report less

[2] https://github.com/AlDanial/cloc

than 400 pairs, we inspect all the reported pairs. Besides, the number of reported clone pairs is not sufficient to demonstrate the detection ability of the tool, but the number of reported true clone pairs. For a clear and intuitive comparison of different tools, we approximate the product of the number of reported clone pairs and the computed precision as the number of reported true positive clone pairs. We can then compare the detection ability of CCStokener and other tools using the number of approximate true positive clone pairs.

(3) RQ3: Our tool, CCStokener, does not require training for clone detection, while DL-based tools need well-labeled datasets to perform well. In the realistic environment, it is usually impossible to obtain the well-labeled results of all code repositories to perform training for clone detection, that is, the DL-based methods need to train the model on some code repositories and perform detection on others. In order to evaluate the generalization and transferability of CCStokener and DL-based tools, We choose 4 real projects (i.e., JDK-1.2.2, Ant-1.10.1, Maven-3.5.0, OpenNLP-1.8.1) and 2 DL-based tools (i.e., ASTNN, TBCCD), then train the model with BigCloneBench labeled pairs and test on these 4 projects.

We extract the tagged true clone pairs (about 8 millions) and tagged false clone pairs (about 279 thousand) from BigCloneBench, use these pairs to train clone detection model for ASTNN and TBCCD. To a certain extent, these clone pairs can train models that meet the real situation. Then, we test the generality of the trained model of ASTNN and TBCCD on 4 open-source projects.

Considering that the four projects are not labeled, the DL-based tools (ASTNN and TBCCD) need labeled pairs to validate their precision and recall. So we use the reported clone pairs from traditional tools. Specifically, we use NiCad, CCAligner, LVMapper and NIL to perform clone detection on the four java projects, randomly sample 200 clone pairs from the results of each tool, manually verify the authenticity of sampled pairs to get the true pairs, finally get about 800 tagged true clone pairs on each project. To get the tagged false pairs, we pair up some dissimilar code blocks, and get about 1600 false pairs on each project.

(4) RQ4: Compared with AST- and PDG-based tools, token-based tools usually have better scalability. In order to compare the scalability of CCStokener with other token-based tools (i.e., NiCad, SourcererCC, CCAligner, LVMapper, NIL), we perform clone detection with these tools on the extracted 1M, 10M, 20M, 30M, 250M LOC datasets respectively, and get detection time of each tool to compare their scalability.

(5) RQ5: To explore the acceleration effect of location and filtration steps, we extract 12,000 blocks from the BigCloneBench dataset, and input different sizes of code blocks from 0 to 12000 for CCStokener, then check the number of clone candidates after the location and filtration steps. We verify the speed-up effect by comparing the number of candidate clones in the three cases: Pairwise (no Location or Filtration), Localization (no Filtration), and Localization + Filtering. Moreover, our multi-round verification algorithm may lead to a lower calculation of the similarity of two vector collections compared with the optimal algorithm, we conduct experiments on the BigCloneBench dataset to examine the degree of influence.

5.2. RQ1: Parameters setting

There are two filtration parameters β and θ, and one verification parameter (i.e., similarity threshold) η to be set. For filtration parameters, we set a list of β and θ, i.e., 0.3, 0.4, 0.5, 0.6, 0.7, respectively. Then we count the proportion of true and false labeled clone pairs to be filtered under a certain β and θ.

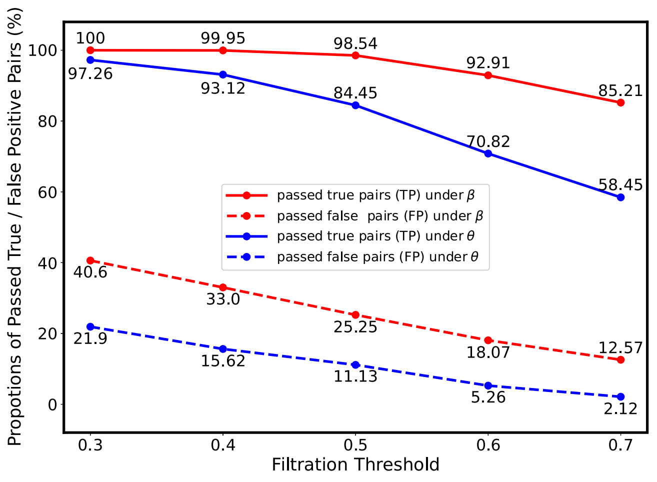

Fig. 5. The proportions of passed True/False positive clone pairs under parameters β or θ.

For the verification parameter, we set a list of η, i.e., 0.5, 0.55, 0.6, 0.65, 0.7. Then count the proportion of true and false labeled clone pairs to be reported under a certain η.

Fig. 5 shows the proportions of true positive and false positive clone pairs that would pass filtration step under parameters β and θ. The red lines are variation under different parameter β, and the blue ones are influenced by θ. Moreover, the solid lines describe the passed true clone pairs with specific parameter β or θ, the slash lines describe the passed false clone pairs. The larger the filtration parameters, the more true clone pairs would be filtered. The small the filtration parameters, the more false clone pairs would pass the filtration step and more time will be wasted on verifying the negative pairs. We should choose a suitable filtration parameter. For parameter β (the red lines), we can see that with its increase, the passed true and false clone pairs both decrease. When its value is set to 0.3, 0.4, 0.5, the proportion of passed true pairs is larger than or equal to 98.54%, but it obviously declines at 0.6, only 92.91%, declining about 5.63%. In addition, the passed false clone pairs are acceptable, about 25.25%, so we set the parameter β as 0.5. For parameter θ, the proportion of passed true and false pairs declines step by step as well. For the purpose of letting more true pairs pass through filtration step, we set θ as 0.4, where 93.11% true pairs and 15.62% false pairs pass. From Fig. 5 we can also find that parameter θ (number of *Action tokens*) has a stronger influence than parameter β (number of total tokens).

The similarity threshold η affects the precision and recall. Fig. 6 depicts the proportions of true and false labeled BigCloneBench clone pairs to be reported, we show the influence on T1, T2, VST3, ST3, MT3 and false clones separately. The red lines describe the proportions of passed true clone pairs, including T1, T2, VST3, ST3 and MT3. Due to the similarity of T1, T2, and VST3 are larger than 0.9, the passed proportions are always 100%. The similarity threshold will influence more on MT3 clones, we can see a clear downward trend. The threshold seems to have fewer influences on false clones (the blue line), if we set η to 0.5, the passed proportion is only 21.90%. After our analysis, we find that this was due to the fact that the generated false clones are vastly different in source code. Furthermore, we can see that there is a sharp decline of the passed true pairs at threshold 0.7. That is because the similarity of MT3 clone pairs is between 0.5 and 0.7. These MT pairs are sensitive to similarity thresholds in this range. In order to report as many true pairs as possible and fewer false pairs, we choose η as 0.65 where there is a declined trend.

Table 3
Recall(%) and precision(%) results of traditional clone detectors (without training) on BigCloneBench.

Tools	Recall(%)						Precision(%)
	T1 (35,787)	T2 (4,573)	VST3 (4,156)	ST3 (14,997)	MT3 (79,756)	WT3/T4 (7,729,291)	
CCSTOKENER	**100**	**99**	**98**	92	**53**	**2.3 (180,330)**	90.2
CCALIGNER	**100**	**99**	97	70	10	0.2 (12,540)	78.8
DECKARD	60	58	62	31	12	1.0 (77,293)	34.8
LVMAPPER	**100**	**99**	**98**	82	19	0.3 (23,923)	88.5
NICAD	**100**	**99**	**98**	**93**	0.8	0.0 (12)	94.5
NIL	**100**	97	94	67	11	0.2 (13,918)	94.1
SOURCERERCC	**100**	97	93	60	5	0.0 (2,005)	**98.8**

T1, T2, VST3, ST3, MT3 and WT3/T4 are recalls reported by BigCloneBench.
The numbers in parentheses of WT3/T4 are the numbers of true clone pairs reported by BigCloneBench.
The highlighted part proves that CCSTOKENER has better recall on MT3 and WT3/T4 clones.
The number under T1, T2, VST3, ST3, MT3 and WT3/T4 is the total number in BigCloneBench, respectively.

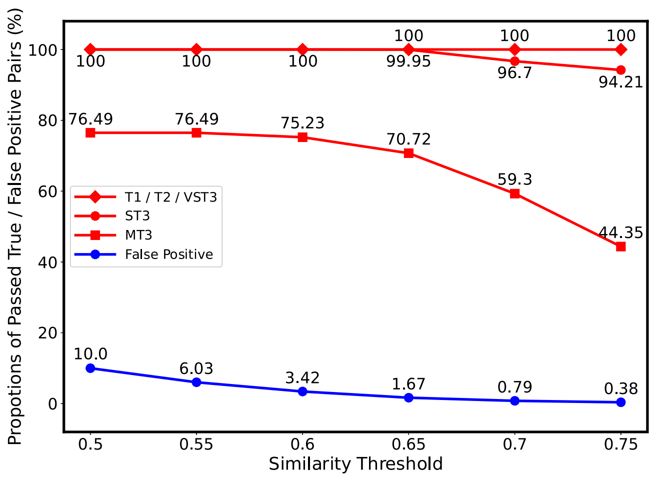

Fig. 6. The proportions of passed True/False positive clone pairs under parameters η.

Answer to RQ1: We set the filtration threshold β to 0.5, and θ to 0.4 to achieve a better filtration effect. We set the similarity threshold η to 0.65 to balance the number of reported true and false clone pairs.

5.3. RQ2: Effectiveness against other tools

In this research question, we want to explore the clone detection capability of CCSTOKENER compared to the state-of-the-art token-based tools such as CCALIGNER, DECKARD, LVMAPPER, NICAD, NIL, SOURCERERCC, and DL-based tools, such as ASTNN and TBCCD. We verify the effectiveness of the detection on two kinds of datasets, i.e., BigCloneBench and eight open-source projects.

5.3.1. Experiments on BigCloneBench

In the first experiment, we use the recommended configurations from these tools' literature. We list the clone detection results of the tools on the BigCloneBench dataset in Table 3. The recalls of T1, T2, VST3, ST3, MT3, WT3/T4 are reported by BigCloneBench automatically. The precision is validated by three researchers manually. In the second experiment, we set a range of similarity threshold, i.e., 0.5, 0.6, 0.7. We select CCSTOKENER and the tools that perform well and have better precision in the first experiment (i.e., NICAD, NIL, SOURCERERCC), and use these tools to conduct clone detection. Finally, we compare the precision and recall of these tools at different similarity thresholds.

The detection results of different tools with recommended configurations are list in Table 3. CCSTOKENER has good results in terms of precision and T1, T2, VST3, and ST3 recalls. The precision of CCSTOKENER is 90.2%. The recall on Type-1 and Type-2 of CCSTOKENER approaches 100%. The reason for not reaching 100% is that *Javalang* made an error while parsing some files. Meanwhile, CCSTOKENER has similar detection abilities on VST3 and ST3 with other tools, the best recall (98%) on VST3 and the second recall (92%) on ST3.

Notably, as highlighted in Table 3, CCSTOKENER has the best recall on detecting MT3 (53%, 41,937 pairs) and WT3/T4 (2.3%, 180,330 pairs) clones, which proves the better detection ability of CCSTOKENER on near-miss clones. Other tools, i.e., NICAD, SOURCERERCC, CCALIGNER, LVMAPPER, DECKARD, have the poor ability on detecting MT3 and WT3/T4 clones. The reason is that most MT3 and WT3/T4 clones have major changes in the code, including the renaming of variable names and the modification of code lines, which will make other token tools invalid. NICAD, CCALIGNER, and LVMAPPER are *line granularity* methods, they will cause mismatches due to minor changes in the lines, even if the code lines are still similar after modifications. SOURCERERCC and NIL require the proportion of the same tokens in a clone pair to reach a certain threshold, but the repetition of the token of many near-miss clones is usually low. CCSTOKENER is not sensitive to fine modifications on code lines, and uses the structural and semantic information of the program, so it achieves reasonable effect on clones with several textural differences.

In general, traditional clone detection tools perform well on detecting high textural-similarity clones (T1, T2, VST3, ST3), but CCSTOKENER improves a lot on the MT3 clones, while having better performance on T1-ST3. In addition, the total number of ST3 is about twice that of T1, seventeen times that of T2, nineteen times that of VST3, and five times that of ST3. The improvement on MT3 proves the practical value of CCSTOKENER.

Table 4 shows the clone detection results of our second experiment. It shows the recall and precision of different similarity thresholds. With the decrease of similarity threshold, the recall of all tools increases, but those precision decreases. Under all similarity thresholds, the recalls (T1, T2, VST3, ST3) of CCSTOKENER rank first or second among all tools, the recall of MT3 is best and obviously higher than that of other tools. Moreover, the precision of CCSTOKENER rank first or second. Additionally, the precision of other tools decreases rapidly at threshold 0.5, about 20% drop of NICAD, about 27% drop of NIL, about 23% drop of SOURCERERCC, but CCSTOKENER drops about 9% and achieves 82%. That is because traditional tools detect clones from the perspective of textural similarity, with the low similarity threshold these tools cannot well distinguish the true or false clones. But CCSTOKENER detects from the perspective of semantic similarity, although there is little semantic information matched under low similarity, the matched information has reasonable interpretation for the clone. From Table 4, we can see that with the same threshold, the detection ability of CCSTOKENER is better than other tools.

Table 4
Recall(%) and precision(%) with different similarity thresholds on BigCloneBench.

Similarity thresholds	Tools	Recall(%)					Precision(%)
		T1	T2	VST3	ST3	MT3	
0.7	CCStokener	**100**	**99**	**98**	92	**47**	95
	NiCad	**100**	**99**	**98**	**94**	1	95
	NIL	**100**	97	94	67	11	94
	SourcererCC	**100**	97	93	60	5	**99**
0.6	CCStokener	**100**	**99**	**98**	93	**61**	91
	NiCad	**100**	**99**	**98**	**95**	6	**93**
	NIL	**100**	97	94	72	17	83
	SourcererCC	**100**	97	94	67	15	91
0.5	CCStokener	**100**	**99**	**98**	93	**66**	**82**
	NiCad	**100**	**99**	**98**	**95**	31	73
	NIL	**100**	97	94	73	22	56
	SourcererCC	**100**	97	97	69	26	59

5.3.2. Experiment on open-source projects

We perform clone detection on eight open-source projects with CCStokener and five traditional tools, including CCAligner, LVMapper, NiCad, NIL, SourcererCC. The number of detected clone pairs and the number of approximate true clone pairs of these tools are presented in Fig. 7, we remove clone pairs with less than 6 lines in these tools. As we described in Section 5.1.2 **RQ2**, we approximate the number of true clone pairs with the product of the number of total reported clone pairs and the precision. The first four subplots are the results of four java projects, the last four subplots are the results of C projects. SourcererCC cannot perform clone detection on C projects, we do not plot its results.

As Fig. 7 shows, CCStokener detects more true clone pairs on all four java projects and two C projects (Cook-2.34, Redis-4.0.0). In particular, CCStokener has a significant improvement in detecting java projects, the true positive clone pairs detected on Ant-1.10.1, JDK-1.2.2, and OpenNLP-1.8.1 are twice as high as the traditional methods. The reason is that there are more syntactic structures in java language, we can get more detailed semantic information. For the four C projects, CCStokener detects more or nearly the same number of clone pairs compared to the best in traditional tools. CCStokener and LVMapper have similar performance on Linux-2.34 and PostgreSQL-6.0, but CCStokener detects more clones on Cook-2.34 and Redis-4.0.0. Due to the relatively simple structure of C language, CCStokener has a slight improvement in the detection of C projects.

Answer to RQ2: For BigCloneBench, CCStokener makes a significant improvement on detecting MT3 clones, and achieves a similar detection ability to those tools that have the best recall on T1, T2, VST3 and ST3. Moreover, CCStokener can detect more true clone pairs on four java open-source projects. Furthermore, the experiments on java open-source projects and BigCloneBench with same similarity threshold demonstrate the validity of the semantics used by CCStokener.

5.4. RQ3: Effectiveness against DL-based tools

In this section, we want to compare the effectiveness of CCStokener with DL-based tools from the perspective of generalization and transferability. Most DL-based tools have performed good detection ability when they train and test on the same dataset. For example, ASTNN has 99.8% precision and 88.4% recall on BigCloneBench, TBCCD has 97% precision and 96% recall on BigCloneBench. But, it is impossible to get the well-labeled code pairs on all code repositories in reality, which results that DL-based tools need to train with some labeled datasets and test on other datasets. By contrast, CCStokener does not require training to perform clone detection and presents similar detection capability on different datasets.

We use ASTNN and TBCCD as comparison tools, train them with tagged pairs from BigCloneBench, test with the sampled pairs from the detection results of NiCad, CCAligner, LVMapper and NIL on four open-source projects, totaling about 800 true pairs and 1600 false pairs of each project. In addition, we calculate the similarity of these tagged true and false clone pairs with CCStokener, thereby verifying whether they can be correctly distinguished by CCStokener or not.

As Table 5 shows, CCStokener has the best precision and F1-score on the four projects, TBCCD has the best recall on the four projects. As a whole, CCStokener has the best generalization and transferability, ASTNN renders the worst. The reason why the recall of CCStokener can exceed 90% is that most true pairs detected by NiCad, CCAligner, LVMapper and NIL are T1, T2, VST3 or ST3, these true pairs are relatively similar. On these relatively similar clone pairs, the failure of ASTNN and TBCCD to achieve good results further illustrates the practicality of these two methods. Although TBCCD has the best recall on four projects, it recognizes many false clone pairs as true, resulting in the lower precision. The inability to detect effectively on similar clone pairs proves that the DL-based tools require well-labeled datasets, however, it is impossible to label all datasets in real-world environments.

Answer to RQ3: CCStokener has better generalization than ASTNN and TBCCD, it has best precision (over 95%) and F1-score (over 90%) on the four projects. ASTNN and TBCCD have poor detection ability when train with tagged pairs of BigCloneBench and test with pairs of four projects.

5.5. RQ4: Scalability against token-based tools

Most token-based tools usually have good scalability. The PDG-based methods require more time to detect due to the problem of subgraph isomorphism. DL-based methods usually require training, which makes them unsuitable for the large code repository such as 100MLOC. We run CCStokener and other token-based tools (i.e., NiCad, SourcererCC, CCAligner, LVMapper, NIL) on the generated 1M, 10M, 20M, 30M, and 250MLOC datasets to explore the difference of scalability between CCStokener and other token-based tools.

The detection time across the different scale of datasets are listed in Table 6. As shown in Table 6, CCStokener detects fastest on 1M LOC, and ranks second on 10M, 30M and 250M LOC. It takes a lot of time to calculate the similarity of semantic tokens (i.e., vectors) in verification step, which is the main reason for the slow speed. NIL needs the least time to perform clone detection on 10M-250M LOC datasets. The reason is that it compares the similarity of code blocks only in terms of the literal token, and it uses optimized LCS algorithm for verification. But, NIL has lower recall on ST3, MT3 and WT3/T4 clones. LVMapper and SourcererCC need more time to perform clone detection on large-scale datasets (250 MLOC). CCAligner and NiCad are not able to scale to the datasets larger than or equal to 20M LOC (marked as '-' in Table 6), so they have poor scalability.

In general, CCStokener has better scalability on the generated datasets. There are two reasons for achieving better results. First, CCStokener filters out numerous false candidate blocks in advance through the location and filtration steps, preventing them

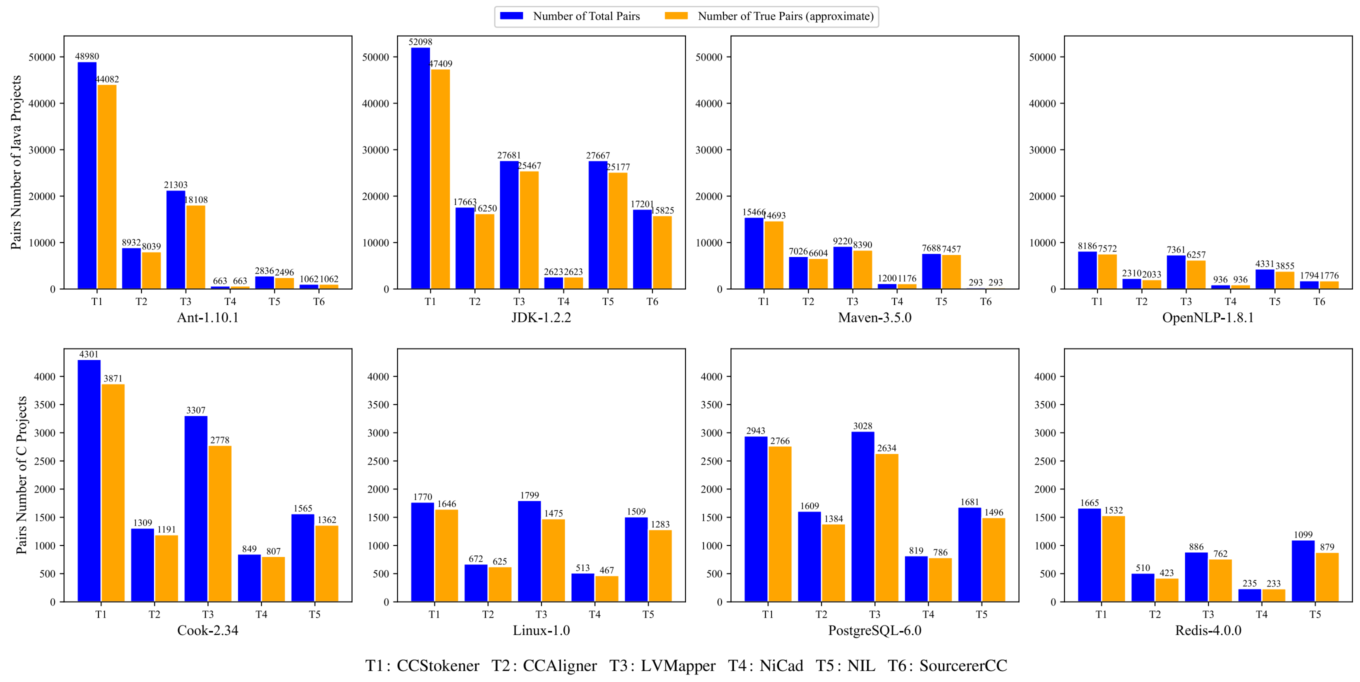

Fig. 7. The number of total clone pairs and approximate true pairs of CCStokener and traditional clone detectors on eight open-source projects.

Table 5
Precision(P), recall(R), F1-score(F1) on open-source projects.

Projects	CCStokener			ASTNN			TBCCD		
	P(%)	R(%)	F1(%)	P(%)	R(%)	F1(%)	P(%)	R(%)	F1(%)
Ant-1.10.1	96.52	92.90	**94.68**	65.65	65.65	65.65	71.96	96.56	82.47
JDK-1.2.2	98.62	91.93	**95.16**	59.97	60.61	60.29	70.01	96.58	81.17
Maven-3.5.0	97.34	96.11	**96.72**	57.80	44.29	50.12	86.37	97.80	91.72
OpenNLP-1.8.1	96.13	86.78	**91.22**	43.01	86.64	57.48	73.58	95.26	83.03

from entering the subsequent verification step, corresponding to 5–21 lines of Algorithm 1. Second, we optimize the verification process to reduce its time complexity, corresponding to Algorithm 2. We will verify the effects of these two aspects in RQ3, respectively.

Answer to RQ4: In comparison with the other token-based tools on the generated datasets, CCStokener ranks the second on 10M, 30M and 250M LOC when performing clone detection. NIL needs least time on 10M-250M LOC. LVMapper and SourcererCC can scale to 250M LOC, but take around twice the time of CCStokener. CCAligner and NiCad fail to run on datasets greater than or equal to 20M LOC.

5.6. RQ5. Acceleration of location–filtration process and effectiveness of verification

5.6.1. Acceleration of location–filtration process

Compared with pair-wise matching code blocks to judge clones, our location and filtration steps can quickly obtain possible candidates, and pre-filter some fake clones to speed up the entire clone detection process. In order to verify the acceleration effect of this process, we input different numbers of code blocks for CCStokener, observe the number of candidate code blocks after the location and filtration steps, and compare these two steps with the pair-wise matching category, the result is in Fig. 8.

In Fig. 8, the x-axis represents the number of blocks and the y-axis represents the logarithm of the number of candidate blocks. As shown, the location and filtration steps reduce the number of clone candidates drastically. For example, if there are 8000 code blocks, pair-wise matching can produce 31,996,000 candidate pairs. But in CCStokener, the candidate number first reduces to 218,384 after location step, then reduce to 46,752 after filtration step, nearly 0.15% of the number of pair-wise matching. Thus, the location–filtration process in CCStokener has a significant acceleration effect.

5.6.2. Effectiveness of verification algorithm

The multi-round algorithm (i.e., Algorithm 2) in verification step can reduce the time complexity from $O(N!)$ to $O(k * N^2)$. In order to verify the effectiveness of the algorithm, we take the BigCloneBench dataset as input and calculate the precision, recall, and detection time of the multi-round algorithm and the optimal algorithm respectively.

As shown in Table 7, MT3 and WT3/T4 recalls of the optimal algorithm (63% and 8.6%, respectively) are greater than that of the multi-round algorithm (55% and 3.4%, respectively), but the precision of the optimal algorithm (85.7%) is less than the multi-round algorithm (90.2%). The reason is that the optimal algorithm makes the similarity calculation more accurate, thereby reporting more clone pairs, causing an increase in recall. But the added clone pair contains some fake clones, so the precision of the optimal algorithm will also decrease. Moreover, it is worth noting that the detection time of the multi-round algorithm is far better than the optimal algorithm, the multi-round algorithm needs 6m14s, 96% less time than the optimal algorithm, which needs 2h46m15s. Therefore, compared with the optimal algorithm, the

Table 6
Detection time for different input size.

LOC	CCSTOKENER	CCALIGNER	LVMAPPER	NICAD	NIL	SOURCERERCC
1M	4s	1 m 8 s	11 s	4 m 10 s	10 s	5 m 40 s
10M	5 m 28 s	48 m 36 s	7 m 44 s	10 h 36 m	1 m 23 s	27 m 52 s
20M	28 m 49 s	–	19 m 24 s	–	20 m 11 s	1 h 2 m
30M	1 h 9 m	–	1 h 35 m	–	44 m 18 s	1 h 32 m
250M	26 h 48 m	–	52 h 25 m	–	7 h 40 m	56 h 41 m

Table 7
The Recall(R), precision(P), and detect time of different verification algorithm.

Algorithm	Recall(%)						Precision(%)	Time
	T1	T2	VST3	ST3	MT3	WT3/T4		
multi-round	99	98	97	92	53	2.3	90.2	6 m 14 s
optimal	99	98	97	93	63	8.6	85.7	2 h 46 m

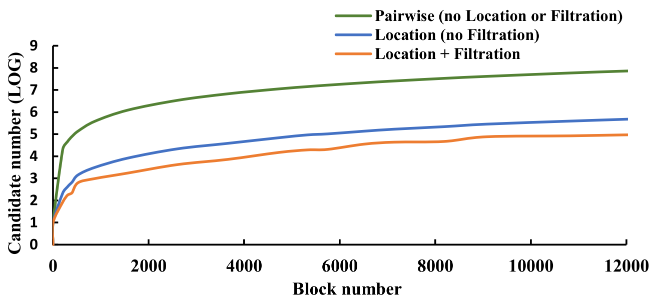

Fig. 8. Growth of the number of clone candidates with the increase of block number.

multi-round algorithm has similar effects on precision and recall, and the detection time is significantly superior to that of the optimal algorithm.

Answer to RQ5: The number of candidates for a target block is significantly reduced to 0.15% of pair-wise matching, which proves the acceleration effect of the location and filtration steps. Moreover, the multi-round algorithm of verification step achieves similar recall and precision with the optimal algorithm, but the multi-round algorithm saves 96% of time.

5.7. Threats to validity

Threats to validity are mainly reflected in two aspects. First, there is no standard criterion for determining whether a clone pair is true, which leads to the difference in precision on the BigCloneBench dataset compared to other literature. We mitigate this threat by evaluating unclear clone pairs by three researchers who have at least six years of programming experience. The proportion of indistinguishable clone pairs is about 10%. Second, different configurations of clone detection tools may affect their precision, recall and scalability. To reduce the limitation, we use the configurations in the previous literature (Sajnani et al., 2016) or contact their authors to maximize their performance. In addition, We set a list of similarity thresholds, and explore the performance of each tool under the same similarity threshold in Section 5.3.1.

6. Related work

Numerous tools have been developed for clone detection. Currently, source code can be mainly represented as string sequences, Abstract Syntax Tree (AST), Program Dependency Graph (PDG). Most researchers design their tools based on the three representations.

Among the token-based clone detection tools (Kamiya et al., 2002; Roy and Cordy, 2008; Wang et al., 2018; Sajnani et al., 2016; Wu et al., 2020a), according to the detection granularity, they can be divided into *line granularity* (e.g., CCFINDER, NICAD, CCALIGNER, LVMAPPER) and *word granularity* (e.g., SOURCERERCC). CCFINDER (Kamiya et al., 2002) uses suffix tree to find identical subsequences to detect clones. NICAD (Roy and Cordy, 2008) parses code fragments with user-specified rules, and computes the longest common length of on the normalized lines. CCALIGNER (Wang et al., 2018) builds e-mismatch index on code blocks, uses it to detect large-gap clones. LVMAPPER (Wu et al., 2020a) expands the large-gap clone to the large-variance clone and improves the sequencing alignment in bioinformatics to achieve better scalability. SOURCERERCC (Sajnani et al., 2016) extracts the valid tokens from the code fragments and performs clone detection with token comparison. However, both *word granularity* and *line granularity* tools do not use the semantic information of programs, they are easily cheated by near-miss clones. For example, variables renaming can invalidate *word granularity* tools, modification and adjustment of the code lines can reduce the detection effect of the *line-granularity* tools. Our tool abstracts semantic tokens from structural information of AST path and semantic information of token relationship, and judges the similarity of tokens from a semantic point of view to detect clones.

Among the PDG-based and AST-based tools, CLONEDR (Baxter et al., 1998) and DECKARD (Jiang et al., 2007) transform the code into abstract syntax tree (AST), and detect similar subtrees to find clones. Sargsyan et al. (2016) generates PDG by LLVM and modified the PDG by removing isolated nodes to improve time performance. CCSHARP (Wang et al., 2017) use a series of categories to decrease the overall computing quantity and computes the similarity of two PDGs with algorithm VF2. CCGRAPH (Zou et al., 2020) detects clones by using an approximate graph matching algorithm to reduce calculation. However, subgraph isomorphism is an NP-hard problem (Johnson, 1987), the time-consuming process of PDG-based tools making them hard to apply to large repositories. Moreover, Duala-Ekoko and Robillard (2010) propose a concept of abstract clone region descriptors (CDRs), which describe clone regions using a combination of their syntactic, structural, and lexical information. They use some fine-grained syntax structural ('for', 'if', 'try' etc.) to describe a block and trace the evolution of clone pairs effectively and accurately. Our tool, CCSTOKENER, utilizes the semantic, structural information to abstract a token, and achieve well improvement on detecting near-miss clone pairs.

Among the DL (machine learning)-based tools, DEEPSIM (Zhao and Huang, 2018) encodes code control flow and data flow into a matrix to capture the semantic information, and measures code functional similarity by deep learning model. Oreo (Saini et al., 2018) estimates semantic similarity and metric similarity to detect clones between syntax and semantic zone. CDHL (Wei and Li, 2017) detects clones by learning representations and Hamming distance of code fragments. ASTNN (Zhang et al., 2019) splits the large AST into a sequence of small statement trees, and trains an RNN model to detect clones. TBCCD (Yu et al., 2019) captures the structural information from AST and lexical information from code tokens, and uses the position-weighted combination method to embed tokens. DL-based methods achieve good precision and recall on well-labeled datasets, but considering the complexity of the real environment, they may be not suitable for the scenarios without the well-labeled training dataset.

7. Conclusion

In this paper, we propose a fast yet accurate clone detection tool, called CCStokener, which aims at improving the detection ability of near-miss clones in large-scale code repositories. We utilize the structural information from the AST path and semantic information from token relationships to generate semantic tokens. We adopt and improve the location–filtration–verification process to achieve the goal of fast clone detection. Experiments show that CCStokener has a better recall on MT3 of BigCloneBench than other tools and detects more clone pairs on open-source projects. Meanwhile, results prove that CCStokener can scale up to large code repositories without compromising detection accuracy.

CRediT authorship contribution statement

Wenjie Wang: Conceptualization, Methodology, Software, Investigation, Data curation, Writing – original draft, Validation. **Zihan Deng:** Validation. **Yinxing Xue:** Writing – review & editing. **Yun Xu:** Resources, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The data that has been used is confidential.

Acknowledgments

We would thank the anonymous reviewers for their insightful comments to improve the quality of the paper. This work is supported by the State Administration of Foreign Experts Affairs under Grant No. BP0719016, by the National Science Foundation of China under Grant No. 61672480, and by the National Science Foundation of China under Grant No. 61972373.

References

Anon, 2012. javalang. URL https://github.com/c2nes/javalang.

Anon, 2014. ANTLR. URL https://github.com/antlr/antlr4.

Baker, B.S., 1995. On finding duplication and near-duplication in large software systems. In: Proceedings of 2nd Working Conference on Reverse Engineering. IEEE, pp. 86–95.

Bakota, T., Ferenc, R., Gyimothy, T., 2007. Clone smells in software evolution. In: 2007 IEEE International Conference on Software Maintenance. pp. 24–33. http://dx.doi.org/10.1109/ICSM.2007.4362615.

Baxter, I.D., Yahin, A., de Moura, L.M., Sant'Anna, M., Bier, L., 1998. Clone detection using abstract syntax trees. In: 1998 International Conference on Software Maintenance, ICSM 1998, Bethesda, Maryland, USA, November 16-19, 1998. IEEE Computer Society, pp. 368–377. http://dx.doi.org/10.1109/ICSM.1998.738528.

Choi, E., Yoshida, N., Ishio, T., Inoue, K., Sano, T., 2011. Extracting code clones for refactoring using combinations of clone metrics. In: Proceedings of the 5th International Workshop on Software Clones. pp. 7–13.

Cosma, G., Joy, M., 2012. An approach to source-code plagiarism detection and investigation using latent semantic analysis. IEEE Trans. Comput. 61 (3), 379–394. http://dx.doi.org/10.1109/TC.2011.223.

Duala-Ekoko, E., Robillard, M.P., 2010. Clone region descriptors: Representing and tracking duplication in source code. ACM Trans. Softw. Eng. Methodol. 20 (1), http://dx.doi.org/10.1145/1767751.1767754.

Gao, Y., Wang, Z., Liu, S., Yang, L., Sang, W., Cai, Y., 2019. TECCD: A tree embedding approach for code clone detection. In: 2019 IEEE International Conference on Software Maintenance and Evolution. ICSME, pp. 145–156. http://dx.doi.org/10.1109/ICSME.2019.00025.

Higo, Y., Yasushi, U., Nishino, M., Kusumoto, S., 2011. Incremental code clone detection: A PDG-based approach. In: 2011 18th Working Conference on Reverse Engineering. IEEE, pp. 3–12.

Jiang, L., Misherghi, G., Su, Z., Glondu, S., 2007. Deckard: Scalable and accurate tree-based detection of code clones. In: 29th International Conference on Software Engineering. ICSE'07, IEEE, pp. 96–105.

Johnson, D.S., 1987. The NP-completeness column: An ongoing guide. J. Algorithms 8 (2), 285–303. http://dx.doi.org/10.1016/0196-6774(87)90043-5.

Kamiya, T., Kusumoto, S., Inoue, K., 2002. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. IEEE Trans. Softw. Eng. 28 (7), 654–670.

Li, J., Ernst, M.D., 2012. CBCD: Cloned buggy code detector. In: 2012 34th International Conference on Software Engineering. ICSE, IEEE, pp. 310–320.

Li, L., Feng, H., Zhuang, W., Meng, N., Ryder, B., 2017. CCLearner: A deep learning-based clone detection approach. In: 2017 IEEE International Conference on Software Maintenance and Evolution. ICSME, pp. 249–260. http://dx.doi.org/10.1109/ICSME.2017.46.

Li, Z., Lu, S., Myagmar, S., Zhou, Y., 2006. CP-Miner: Finding copy-paste and related bugs in large-scale software code. IEEE Trans. Softw. Eng. 32 (3), 176–192.

Liu, C., Chen, C., Han, J., Yu, P.S., 2006. GPLAG: detection of software plagiarism by program dependence graph analysis. In: Eliassi-Rad, T., Ungar, L.H., Craven, M., Gunopulos, D. (Eds.), Proceedings of the Twelfth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Philadelphia, PA, USA, August 20-23, 2006. ACM, pp. 872–881. http://dx.doi.org/10.1145/1150402.1150522.

Mockus, A., 2007. Large-scale code reuse in open source software. In: First International Workshop on Emerging Trends in FLOSS Research and Development. FLOSS'07: ICSE Workshops 2007, IEEE, p. 7.

Murakami, H., Hotta, K., Higo, Y., Igaki, H., Kusumoto, S., 2013. Gapped code clone detection with lightweight source code analysis. In: 2013 21st International Conference on Program Comprehension. ICPC, IEEE, pp. 93–102.

Nakagawa, T., Higo, Y., Kusumoto, S., 2021. NIL: large-scale detection of large-variance clones. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 830–841.

Rahman, F., Bird, C., Devanbu, P., 2010. Clones: What is that smell? In: 2010 7th IEEE Working Conference on Mining Software Repositories. MSR 2010, pp. 72–81. http://dx.doi.org/10.1109/MSR.2010.5463343.

Rattan, D., Bhatia, R., Singh, M., 2013. Software clone detection: A systematic review. Inf. Softw. Technol. 55 (7), 1165–1199. http://dx.doi.org/10.1016/j.infsof.2013.01.008, URL https://www.sciencedirect.com/science/article/pii/S0950584913000323.

Roy, C.K., Cordy, J.R., 2007. A survey on software clone detection research. In: Queen's School of Computing TR. Vol. 541. No. 115. Citeseer, pp. 64–68.

Roy, C.K., Cordy, J.R., 2008. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In: 2008 16th IEEE International Conference on Program Comprehension. IEEE, pp. 172–181.

Saha, R.K., Asaduzzaman, M., Zibran, M.F., Roy, C.K., Schneider, K.A., 2010. Evaluating code clone genealogies at release level: An empirical study. In: 2010 10th IEEE Working Conference on Source Code Analysis and Manipulation. pp. 87–96. http://dx.doi.org/10.1109/SCAM.2010.32.

Saini, V., Farmahinifarahani, F., Lu, Y., Baldi, P., Lopes, C.V., 2018. Oreo: Detection of clones in the twilight zone. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 354–365.

Sajnani, H., Saini, V., Svajlenko, J., Roy, C.K., Lopes, C.V., 2016. Sourcerercc: Scaling code clone detection to big-code. In: Proceedings of the 38th International Conference on Software Engineering. pp. 1157–1168.

Sargsyan, S., Kurmangaleev, S.F., Belevantsev, A.A., Avetisyan, A., 2016. Scalable and accurate detection of code clones. Program. Comput. Softw. 42 (1), 27–33. http://dx.doi.org/10.1134/S0361768816010072.

Sheneamer, A., Kalita, J., 2016. A survey of software clone detection techniques. Int. J. Comput. Appl. 137 (10), 1–21.

Sobrinho, E.V.d.P., De Lucia, A., Maia, M.d.A., 2021. A systematic literature review on bad smells–5 W's: Which, when, what, who, where. IEEE Trans. Softw. Eng. 47 (1), 17–66. http://dx.doi.org/10.1109/TSE.2018.2880977.

Svajlenko, J., Roy, C.K., 2016. Bigcloneeval: A clone detection tool evaluation framework with bigclonebench. In: 2016 IEEE International Conference on Software Maintenance and Evolution. ICSME, IEEE, pp. 596–600.

Thummalapenta, S., Cerulo, L., Aversano, L., Di Penta, M., 2010. An empirical study on the maintenance of source code clones. Empir. Softw. Eng. 15 (1), 1–34.

Wang, P., Svajlenko, J., Wu, Y., Xu, Y., Roy, C.K., 2018. CCAligner: a token based large-gap clone detector. In: Proceedings of the 40th International Conference on Software Engineering. pp. 1066–1077.

Wang, M., Wang, P., Xu, Y., 2017. CCSharp: An efficient three-phase code clone detector using modified PDGs. In: Lv, J., Zhang, H.J., Hinchey, M., Liu, X. (Eds.), 24th Asia-Pacific Software Engineering Conference, APSEC 2017, Nanjing, China, December 4-8, 2017. IEEE Computer Society, pp. 100–109. http://dx.doi.org/10.1109/APSEC.2017.16.

Wei, H., Li, M., 2017. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In: Sierra, C. (Ed.), Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017. ijcai.org, pp. 3034–3040. http://dx.doi.org/10.24963/ijcai.2017/423.

White, M., Tufano, M., Vendome, C., Poshyvanyk, D., 2016. Deep learning code fragments for code clone detection. In: 2016 31st IEEE/ACM International Conference on Automated Software Engineering. ASE, pp. 87–98.

Wu, M., Wang, P., Yin, K., Cheng, H., Xu, Y., Roy, C.K., 2020a. LVMapper: A large-variance clone detector using sequencing alignment approach. IEEE Access 8, 27986–27997.

Wu, Y., Zou, D., Dou, S., Yang, S., Yang, W., Cheng, F., Liang, H., Jin, H., 2020b. SCDetector: software functional clone detection based on semantic tokens analysis. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering. pp. 821–833.

Yang, W., 1991. Identifying syntactic differences between two programs. Softw. Pract. Exp. 21 (7), 739–755. http://dx.doi.org/10.1002/spe.4380210706.

Yu, H., Lam, W., Chen, L., Li, G., Xie, T., Wang, Q., 2019. Neural detection of semantic code clones via tree-based convolution. In: 2019 IEEE/ACM 27th International Conference on Program Comprehension. ICPC, IEEE, pp. 70–80.

Zhang, J., Wang, X., Zhang, H., Sun, H., Wang, K., Liu, X., 2019. A novel neural source code representation based on abstract syntax tree. In: 2019 IEEE/ACM 41st International Conference on Software Engineering. ICSE, IEEE, pp. 783–794.

Zhao, G., Huang, J., 2018. Deepsim: deep learning code functional similarity. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 141–151.

Zou, Y., Ban, B., Xue, Y., Xu, Y., 2020. CCGraph: a PDG-based code clone detector with approximate graph matching. In: 2020 35th IEEE/ACM International Conference on Automated Software Engineering. ASE, IEEE, pp. 931–942.

Wenjie Wang received the B.S. degree in computer science and technology from Southwest Jiaotong University (SWJTU), Chengdu, China, in 2019, and M.S. degree in computer science and technology from University of Science and Technology of China (USTC), Hefei, China, in 2022. His main research interest is code clone detection.

Zihan Deng received the B.S. degree in computer science and technology from University of Science and Technology of China (USTC), Hefei, China, in 2018, where he is currently pursuing the B.S. degree in computer science and technology. His main research interest is code clone detection.

Yinxing Xue received the B.E. and M.E. degrees in software engineering from Wuhan University, Wuhan, China, in 2005 and 2007, respectively, and the Ph.D. degree in computer science from the National University of Singapore, Singapore, in 2013. He is currently a pre-tenure Research Professor with the University of Science and Technology of China, Hefei, China, where he has been a Research Professor with the School of Computer Science and Technology since 2018. His research interests include software program analysis, software engineering, and cyber security issues (e.g., malware detection, intrusion detection, and vulnerability detection).

Yun Xu received the Ph.D. degree in computer science from the University of Science and Technology of China (USTC), in 2002. He is currently a Professor with the Department of Computer Science, USTC. He has published over 80 refereed articles in areas of high-performance computing, software engineering, bioinformatics, and so on. His research has been supported by the National Natural Science Foundation of China and the Project of Ministry of Education of China. He has advised over 50 Ph.D. and M.S. students.