Code2Img: Tree-based Image Transformation for Scalable Code Clone Detection

Yutao Hu, Yilin Fang, Yifan Sun, Yaru Jia, Yueming Wu, Deqing Zou, and Hai Jin, *Fellow, IEEE*

Abstract—Code clone detection is an active research domain of software engineering. There are two core demands for clone detection: scalable detection and complicated clone detection. For scalable detection, existing approaches treat the source code as a text or token sequence and then calculate their similarity. However, the text-based and token-based approaches are difficult to detect complicated clone types due to the lack of consideration of code structure. The methods based on intermediate representations of code can effectively achieve complex clone types detection but are limited by the complexity of representations to be scalable. In this paper, we propose *Code2Img*, a tree-based code clone detector, which satisfies scalability while detecting complicated clones effectively. Given the source code, we first perform clone filtering by the inverted index to locate the suspected clones. For each suspected clone, we create the adjacency image based on the adjacency matrix of the normalized abstract syntax tree (AST). Then we design an image encoder to highlight the structural details further and refine pixels of the image. Specifically, we employ the Markov model to encode the adjacency image into a state probability image and remove its useless pixels. By this, the original complex tree can be transformed into a one-dimensional vector while preserving the structural feature of the AST. Finally, we detect clones by calculating the Jaccard Similarity of these vectors. We conduct comparative evaluations on effectiveness and scalability with eight other state-of-the-art clone detectors (*SourcererCC*, *NIL*, *LVMapper*, *Nicad*, *Siamese*, *CCAligner*, *Deckard*, and *Yang2018*). The experimental results show that *Code2Img* achieves the best performance among all the comparative tools in terms of both detection effectiveness and scalability. It indicates that *Code2Img* can be applicable to scalable complicated clone detection.

Index Terms—Code Clone, Clone Detection, Scalability.

1 Introduction

With the popularity of the open-source community, more developers are reusing code by copying, pasting, and modifying. While cloning code is more convenient than implementing functionality from scratch, code cloning also raises some issues. For example, the cloned code may not be secure, *i.e.,* vulnerable code. In addition, code cloning can inadvertently lead to code copyright disputes, where code from open-source projects is reused in violation of license terms. In other words, although code cloning brings much convenience to software development, it also reduces software security and increases maintenance costs [1]. Due to these problems, clone detection has become an active area of software engineering and is gradually occupying a prominent position in the field.

Many research works have been proposed to detect code clones, categorized into two dominant groups according to the detection requirements: scalable clone detection and complicated clone detection. In terms of scalable clone detection, they are mainly text-based and token-based methods [2], [3], [4], [5], [6], [7]. These approaches treat the source code as plain text or token sequences and directly calculate their similarities, thus requiring a very short execution time. However, they cannot cope with complicated clone types detection due to the lack of consideration of code structure. To detect complicated clones effectively, clone detectors based on the code intermediate representation have been proposed, such as graph-based [8], [9] and tree-based methods [10], [11]. However, they are not applicable to scalable clone detection. For graph-based approaches, the long generation time and complex graph structure of *program dependency graphs* (PDG) and *control flow graphs* (CFG) lead to a high overall runtime overhead. To save the graph generation time, some researchers have proposed detection approaches based on the *abstract syntax tree* (AST). However, large-scale clone detection requires small memory consumption and a short detection time. The main limitation of the AST-based method is the tree's complex structure, so it has to take up a large amount of memory and a long time for the tree's matching. Hence, it is also difficult to meet the needs of scalable clone detection.

In this paper, we aim to achieve effectiveness and scalability simultaneously in detecting complicated clones from large-scale source code. Our key idea comes from image classification, which can handle millions of images while maintaining a low matching time and memory consumption. Specifically, our paper focuses on addressing one main challenge.

- *Y. Hu, Y. Fang, Y. Sun, Y. Jia, and D. Zou are with National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan, 430074, China. E-mail: yutaohu, yilinfang, sunyifan, yrjia, deqingzou@hust.edu.cn*
- *Y. Wu (corresponding author) is with the School of Computer Science and Engineering, Nanyang Technological University, Singapore, 639798. E-mail: wuyueming21@gmail.com*
- *H. Jin is with National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China. E-mail: hjin@hust.edu.cn*

- *How to efficiently convert the complex tree of a function into an image while preserving the tree structure information?*

To address this challenge, we first perform clone filtering using the inverted index to retain suspected clones. For each suspected clone, we construct its image representation (*i.e.*, adjacency image) by generating an adjacency matrix for its normalized AST. In detail, the adjacency image is composed of the node types and the number of edges of AST, containing the structural feature of the source code. To further improve the detection effectiveness for complicated clones, we resort to Markov models, which have been widely used in other fields as a model that can effectively simplify the model and bring out the structural details of the model [12], [13]. We employ the Markov chain to encode the adjacency image into a state probability image to curve the structural details of the AST. To reduce the complexity of the image to save memory, we remove the useless pixels from the state probability image and generate a one-dimensional vector. Finally, we adopt the Jaccard Similarity to calculate the similarity between the vectors to detect clones.

We implement a novel system, *Code2Img*, and evaluate its detection accuracy and scalability on a widely used benchmark datasets, *BigCloneBench* (BCB) [14]. We compare *Code2Img* with eight state-of-the-art clone detectors, namely *SourcererCC* [15], *NIL* [16], *LVMapper* [17], *Nicad* [7], *Siamese* [18], *CCAligner* [2], *Deckard* [19], and *Yang2018* [20]. As for detection performance, our experimental results show that *Code2Img* outperforms the above detectors, especially for complicated Type-3 clones. As for scalability, *Code2Img* takes only 7 hours and 39 minutes to complete the 250-MLOC codebase detection, which takes the shortest time among the comparative tools. In summary, the results prove that *Code2Img* achieves the best detection effectiveness and efficiency among all the comparative detectors at the same time. It indicates that *Code2Img* can be applicable to scalable complicated clone detection.

Contributions. In summary, our paper makes the following contributions:

- We propose a novel approach to transform an AST into a simple image representation while retaining the structural details of the source code. The transformation helps to avoid high-cost tree comparison and high memory occupation.
- We implement *Code2Img* [1], a tree-based detector that simultaneously balances the effectiveness of complicated clone detection and supports 250-MLOC large-scale code detection.
- We conduct extensive evaluations in terms of effectiveness and scalability. The experimental results show that *Code2Img* is superior to the other eight state-of-the-art clone detectors (*i.e.*, *SourcererCC* [15], *NIL* [16], *LVMapper* [17], *Nicad* [7], *Siamese* [18], *CCAligner* [2], *Deckard* [19], and *Yang2018* [20]).

Paper Outline. Section 2 introduces the related definitions and a motivating example. Section 3 describes the system architecture of *Code2Img*. Section 4 reports the experimental results. Section 5 discusses the threat to validity and future work. Section 6 presents the related work. Section 7 concludes the present paper.

1. https://github.com/tao7777/code2img

2 Definition and Motivation

This section presents the formal definition of code clone types and our insights from a motivating example.

2.1 Definition

In this part, we introduce the formal definitions of clone types that we adopt throughout the paper, following the prior works [21], [22].

```
1  public static Image[][] reversalXandY(final Image[][] array){
2      int col = array[0].length;
3      int row = array.length;
4      Image[][] result = new Image[col][row];
5      for (int y = 0; y < col; y++) {
6          for (int x = 0; x < row; x++) {
7              result[x][y] = array[y][x];
8           }
9      }
10      return result;                                          Source
11 }                                                           Function
```

```
1  public static Image[][] reversalXandY(final Image[][] array){
2      int col = array[0].length; //the column of image pixel
3      int row = array.length;    //the row of image pixel
4      Image[][] result = new Image[col][row];
5      for (int y = 0; y < col; y++) {
6          for (int x = 0; x < row; x++) {
7              result[x][y] = array[y][x];}
8      }                                                       Type-1
9      return result; }
```

```
1  public static LImage[][] reversalXandY(final LImage[][] array) {
2      int col = array[0].length;
3      int row = array.length;
4      LImage[][] result = new LImage[col][row];
5      for (int y = 0; y < col; y++) {
6          for (int x = 0; x < row; x++) {
7              result[x][y] = array[y][x];
8          }
9       }
10     return result;                                          Type-2
11  }
```

```
1  public boolean[][] getAdjacency() {
2      int n = vertices.size();
3      GraphVertex verts[] = vertices.toArray(new GraphVertex[0]);
4      boolean adj[][] = new boolean[n][n];
5      for (int i = 0; i < n; i++) {
6          adj[i][i] = false;
7          for (int j = i + 1; j < n; j++) {
8              adj[i][j] = verts[i].getNeighbors().contains(verts[j])
                   || verts[j].getNeighbors().contains(verts[i]);
9              adj[j][i] = adj[i][j];
10          }
11      }
12      return adj;                                             Type-3
13  }
```

```
1  public EstimatedPolynomial evaluate() {
2    for (int i = 0; i < systemConstants.length; i++) {
3      for (int j = i + 1; j < systemConstants.length; j++)
           systemMatrix[i][j] = systemMatrix[j][i];}
4      try {
5        LUPDecomposition lupSystem = new LUPDecomposition(systemMatrix);
6        double[][] components = lupSystem.inverseMatrixComponents();
7        LUPDecomposition.symmetrizeComponents(components);
8        return new EstimatedPolynomial(lupSystem.solve(systemConstants),
                     SymmetricMatrix.fromComponents(components));
9      } catch (DhbIllegalDimension e) {
10     } catch (DhbNonSymmetricComponents ex) {
11      };                                                     Type-4
12  return null; }
```

Fig. 1: Examples of different clone types

Type-1 Clones (T1, Textual Similarity) are identical code fragments, excluding spaces, blank lines, and comments.

Type-2 Clone (T2, Lexical Similarity) are identical code fragments except for some renamed unique identifiers (*i.e.*, function name, class name, variables).

Type-3 Clone (T3, Syntactic Similarity) are syntactically similar differ only at statement-level. Specifically, these fragments have statements added, modified, or deleted from each other. Moreover, due to the confused boundary

between the definitions of Type-3 and Type-4 clones, the *BigCloneBench* (BCB) [23] dataset further classifies T3 clones into subcategories. It is based on the code similarity score of line-level and token-level after T1 and T2 normalization. Specifically, they are *Very Strongly Type-3* (VST3) ($\geq$ 90% similarity), *Strongly Type-3* (ST3) (70%-90% similarity), and *Moderately Type-3* (MT3) (50%-70% similarity).

Type-4 Clone (T4, Semantic Similarity) indicates syntactically dissimilar fragments that implement the same functionality. T1, T2, and T3 clones demonstrate textual similarity, whereas T4 clones represent functional similarity.

To better elaborate on the different types of clones, we adopt an example from the BCB dataset [23], as shown in Figure 1. They are the T1 to T4 clones of a source function, which all have similar functionality with transposing a two-dimensional matrix. The T1 clone is identical to the source function except for the addition of comments at lines 2-3. The T2 clone differs from the source function only in a variable name, *i.e.,* `LImage` instead of `Image`. Additionally, it can be seen that the T3 clone is textually dissimilar to the source function, as reflected in the function names, variable names, and variable types are completely different. However, they have some syntactic similarities, *i.e.,* transpose the matrix by nested loops. Obviously, detecting T3 clones is harder than the Types 1-2 clones. T4 clone also exchanges the rows and columns of a matrix, adding some exception-catching statements, so the lexical and syntactic differs greatly. Detecting T4 clones requires a deep understanding of the semantics of the code, so it is not practical to implement T4 clone detection for large-scale code. In this paper, we focus on achieving scalable T3 clone detection.

2.2 Motivation

In this part, we use a motivating example to illustrate how we proposed the tree-based image transformation to detect code clones. The example is the T3 clone in Figure 1, which is regarded as a complicated T3 clone (*i.e.,* MT3 in the BCB dataset [23]).

We calculate the Jaccard similarity of the token, AST, and AST's adjacency matrix of the pair of codes, respectively. As for token similarity, we first tokenize the source code to obtain the token sequence of the source code. Then the token sequences of the two functions are put into Set_1 and Set_2, respectively. For AST similarity, we extract the AST of these two functions and then put the nodes of the two ASTs into Set_1 and Set_2. Then we compute the Jaccard similarity of token and AST as in the formula of set_Jacc_sim 1. From the formula, we can calculate that the token's similarity is 0.38 and the AST's similarity is 0.43, as shown in Table 1.

$$set\_Jacc\_sim = \frac{|Set_1 \cap Set_2|}{|Set_1 \cup Set_2|} \tag{1}$$

Even though ASTs can effectively characterize the code structure and even semantics, the structure of ASTs is too complex to be applicable to scalable clone detection. Therefore, we further simplify the ASTs while preserving their structural information. In particular, we extract the number of edges between all node types of the targeted ASTs to construct their adjacency matrices (M_1 and M_2). Then calculate the Jaccard similarity of the adjacency matrices, as in the equation 2 below. The adjacency matrix similarity of example code pair is calculated as 0.63, as shown in Table 1.

$$mtx\_Jacc\_sim = \frac{M_1^T M_2}{\|M_1\|^2 + \|M_2\|^2 - M_1^T M_2} \tag{2}$$

We pick a similarity threshold as in previous work [16], [15]. They set the threshold to 0.7, meaning that the methods based on token, AST, and AST's adjacency matrix all result in false negatives, *i.e.,* the code pair detected non-clones. In detail, the low similarity of the token is since that these share very few tokens. The similarity of AST is low since the number of nodes in AST varies greatly, and the number of identical nodes is even rarer, indicating that it is difficult to detect MT3 clones by directly matching AST. However, after further abstraction of the AST to obtain its adjacency matrix, the similarity of the adjacency matrix is significantly improved. It can be inferred that the simplification of the AST is effective in enhancing its structural information, but still hard to detect T3 clones.

TABLE 1: The similarity of different code representations

Representation	Token	AST	AST's Adjacency Matrix	AST's Probability Matrix
Example's Similarity	0.38	0.43	0.63	0.83
Average Similarity	0.27	0.46	0.59	0.79

Based on the above observations, we further refine the information of the AST's adjacency matrix to highlight the structural features of the AST better to reinforce T3 clone detection. Considering that research work in other fields has demonstrated that Markov chains can be used to simplify the model's complexity and emphasize the model's structural details [12], [13], we optimize the adjacency matrix of AST with the help of the Markov chain model. Specifically, we treat the adjacency matrix of the AST as a state transition matrix. The probability transition matrix of the state transition matrix is then derived using the Markov model. The detailed transformation is as the formula 3, where $prob\_trans\_mtx[i][j]$ presents the state transition probability from node type i to node type j in an AST and $M_1[i][j]$ indicates the number of occurrences of the edges between node type i to node type j in the AST. k and n denote a node type and the number of node types in the AST, respectively.

$$prob\_trans\_mtx[i][j] = \frac{M_1[i][j]}{\sum_{k=0}^{n} M_1[i][k]} \tag{3}$$

Finally, we use the formula 2 to calculate the Jaccard similarity of the probability transition matrices, whose final result is 0.83. It can be seen that it is higher than the similarity of the token, tree, and tree adjacency matrix and exceeds 0.7, so it can be correctly detected as a clone.

In addition, we randomly select 1000 MT3 clones from the BCB dataset and calculate their average Jaccard similarity, as shown in Table 1. It can be seen that only the probability matrix exceeds 0.7 among the total average similarity. In other words, only the probability transition matrix can support complex clone detection. Therefore, we can infer that Markov chains help capture the tree's structural details and thus improve clone detection's effectiveness, especially

for the complicated clone types. Inspired by these insights, we propose a novel approach to transform complex ASTs into probability matrix-based images to obtain a briefer code representation for effective and scalable code clone detection.

3 Approach

In this section, we introduce *Code2Img*, a scalable and efficient code clone detector with a tree-based image transformation.

3.1 System Overview

The framework of *Code2Img* is shown in Figure 2, which consists of four main phases: *Data Preprocessing*, *Clone Filtering*, *Structure Encoding*, and *Clone Detecting*.

- *Data Preprocessing*: This phase aims to normalize the type-specific tokens of the AST and generate the inverted index by N-line hash for each code block of the normalized code.
- *Clone Filtering*: This phase is targeted to search for clone candidates in the inverted index and calculate the N-lines similarity to obtain the suspected clones.
- *Structure Encoding*: For the suspected clones, this phase first transforms the AST into an adjacency image by the adjacency matrix of AST's node types. Then encodes the adjacency image with the Markov model to obtain the state probability image and remove its useless pixels to get the final vector.
- *Clone Detecting*: This phase aims to detect clones within the suspected clones by calculating the Jaccard similarity between their final vectors.

3.2 Data Preprocessing

Code2Img focuses on building a scalable clone detector. For the purpose of a trade-off between scalability and semantic preservation, we use AST as an intermediate representation of the code. Compared to the graph representations of code, generating ASTs does not require any compilation and is time-saving. Therefore, it satisfies the needs of clone detection for large-scale. In addition, it also has the advantages of various development languages and code fragments of different sizes. We perform static analysis to parse the source code and extract its AST. In detail, since we conduct all the experiments on the dataset written in Java, we adopt *Javaparser* [24] to parse the Java files and generate their ASTs.

To ensure the resilience of *Code2Img* for T2 clones, we normalize the type-specific tokens of the AST. Specifically, we analyze all the T2 clone codes in the BCB dataset and find six types of nodes whose tokens change between the clone pairs. They are all node types related to identifier, including `SimpleName`, `Name`, `StringLiteralExpr`, `BooleanLiteralExpr`, `IntegerLiteralExpr`, and `DoubleLiteralExpr`. Figure 3 shows the normalization result of $SourceFunction$ in Figure 1. According to the definition of T2 clones, they differ only in the name of the identifier. Therefore, T2 clones are identical in both the tree structure and the code information after normalizing the tokens of these six types of nodes. So we can infer that the normalization helps *Code2Img* to tolerate T2 clone changes, thus detecting simple clones more efficiently.

After the code normalization, we calculate the *N-line Hash* for the source code. An *N-line* represents a code block consisting of consecutive N lines code. For example, we generate three *3-lines* for a part of $SourceFunction$ of Figure 1, as shown in Figure 3. For the implementation, we stitched the code of these N lines into a long string. Then, we compute a hash value for the text of this string, which is an *N-line Hash* of the code fragment. It can be inferred that a code fragment containing M code lines will generate $M - N + 1$ *N-line Hashes*. So we can obtain nine *3-line Hashes* for the function in Figure 3. We can know that if two pieces of code share an *N-line Hash*, there are N identical consecutive code lines between them. Therefore, the more the same *N-line Hashes* exist in a code pair, the greater their similarity. With the help of *N-line Hash*, *Code2Img* can quickly locate the code pairs that may have similar relationships.

To locate pairs with identical code blocks, we resort to the inverted index. Inverted indexing is an information retrieval technique that allows fast retrieval of documents containing a word as a query and is often used in clone detection techniques [16], [15]. We adopt a dictionary as the data structure to store the inverted index. Specifically, the keys are the *N-line Hash*, and the values are the names of all source codes containing this *N-line Hash*. In other words, all source code names containing this *N-line Hash* can be quickly obtained in the inverted index by searching for the key as this Hash value. Therefore, we create an inverted index for locating the code sharing the consecutive code lines to improve detection efficiency.

3.3 Clone Filtering

With the help of the inverted index, *Code2Img* performs clone location and clone filtration in this phase. The location part is designed to locate the candidate clones that share consecutive lines of code with the target source code. In other words, it helps to filter non-clones where the text has almost no similarity quickly. The filtration part aims to filter out obvious non-clones and those almost identical codes to retain suspected clones that require clone detection.

The location phase targets locating the clone candidates, which are kept for the filtration part. For the target source code, we first extract its AST and perform normalization on it, following section 3.2 described. Then we calculate its *N-line Hashes* and use them as the keys to search for the inverted index. As a result, the codes that share the *N-line Hashes* with the source code can be located by inverted indexing, while those without the same *N-line Hash* are considered completely dissimilar and be quickly filtered out. These retained codes are called candidate clones.

The clone filtration part is proposed to calculate the similarity of *N-lines* between the target code and clone candidates to filter further. The code pairs share a large portion of *N-lines*, meaning they have many identical lines of code, so the two codes are likely clones. Conversely, if the code pairs have few shared *N-lines*, it means that their texts vary significantly, *i.e.,* they are non-cloned pairs. Based on the above analysis, we calculate the similarity of *N-lines* between the two codes, $code\_block\_sim$, by the

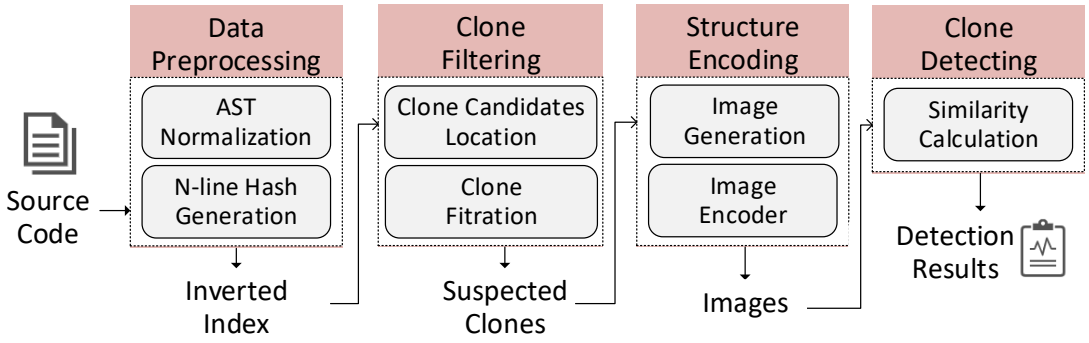


Fig. 2: System overview of *Code2Img*

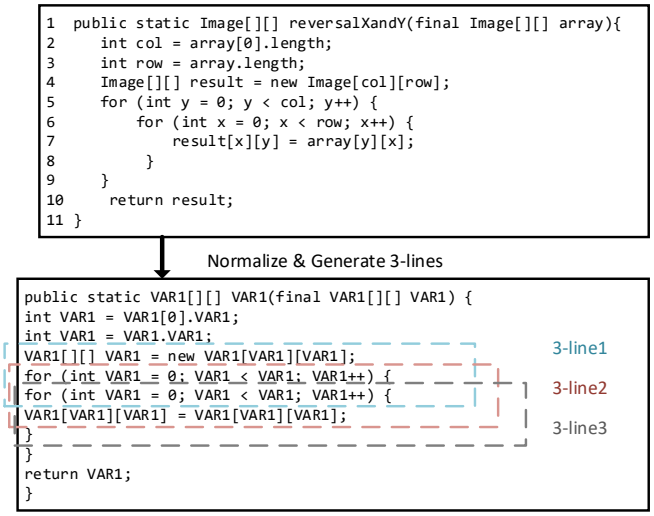


Fig. 3: Example of normalization and *3-lines* generation

following formulas. C_1 and C_2 denote two pieces of source code. $nlines(C_1)$ and $nlines(C_2)$ are the list of N-line Hash generated by C_1 and C_2, whose lengths are $|nlines(C_1)|$ and $|nlines(C_2)|$, respectively. Besides, $common\_nlines$ denotes the number of the their shared *N-lines*. In other words, $nlines(C)$ is the ordered sequence of N-line Hashes.

$$common\_nlines(C_1, C_2) = |nlines(C_1) \cap nlines(C_2)| \quad (4)$$

$$code\_block\_sim = \frac{common\_nlines(C_1, C_2)}{max(|nlines(C_1)|, |nlines(C_2)|)} \quad (5)$$

We set the non-clone threshold $\theta 1$ and the clone threshold $\theta 2$. If the $code\_block\_sim$ is lower than $\theta 1$, it is judged as non-clone. Conversely, if the $code\_block\_sim$ is higher than $\theta 2$, it is regarded clone. Finally, only code pairs with $code\_block\_sim$ greater than $\theta 1$ and less than $\theta 2$ are the suspected clones for image transforming and clone detecting.

The clone location part effectively helps *Code2Img* to locate candidate clones. In addition, the filtration part for normalized code can quickly detect most T1 and T2 clones while filtering out obvious non-clones, retaining the suspected clones. Therefore, only a small portion of code pairs require clone detection, significantly reducing the overall time overhead of *Code2Img*.

3.4 Structure Encoding

AST is a complex code representation, and it is difficult to cope with large-scale detection by directly matching AST for clone detection. However, images have the inherent advantages of fast matching and small space occupation, which match the core requirements of clone detection. Therefore, we convert the AST into images for clone detection for suspected clone pairs.

In the image generation part, we extract the adjacency matrix to create its image representation for the normalized AST. The AST can intuitively reflect the structure of the source code. Besides, the structural information stored in the AST is essential for complex clone types detection. Inspired by this, we characterize the structure of the AST by the adjacency matrix. Specifically, AST is a tree data structure that consists of nodes and edges. The left part of Figure 4 shows an AST's subtree of the normalized $SourceFunction$ in Figure 1, which represents the tree for lines 1-2 of the function. To obtain the detailed number of node types, we parse over 100-MLOC codes using *Javaparser* [24] and find that there are 70 node types in the ASTs, including 58 kinds of type nodes and 12 types of token nodes. The token nodes are the leaf nodes of AST (the double-layer hollow rectangles in Figure 4), and others are the type nodes (the single-layer solid rectangles). For example, the `SimpleName` (with blue double-layer hollow rectangles) denotes a kind of token nodes, whose token is `VAR1`. Besides, the `ArrayAccessExpr` (with orange single-layer solid rectangles) is a type node. These node types and edges mainly curve the structure of AST. Therefore, we construct the adjacency matrix based on the number of edges between different types of nodes and transform the adjacency matrix into an image. In other words, each pixel of the image (*i.e.*, $img_1[i][j]$) indicates the number of occurrences of the edges between node type i to node type j in an AST. The right part of Figure 4 shows an example to illustrate the image transformation. For example, it can be seen that the edge from node `NameExpr` (the pink solid single-layer rectangles) to node `SimpleName` (the blue hollow double-layer rectangles) has appeared twice, so their corresponding pixel is set to 2 in the adjacency image. In addition, since there can be no edge relationship between token nodes, the adjacency image is 58×70 pixels.

Based on the observation in Section 2.2, the adjacency matrix characterizes the general structure of the AST but

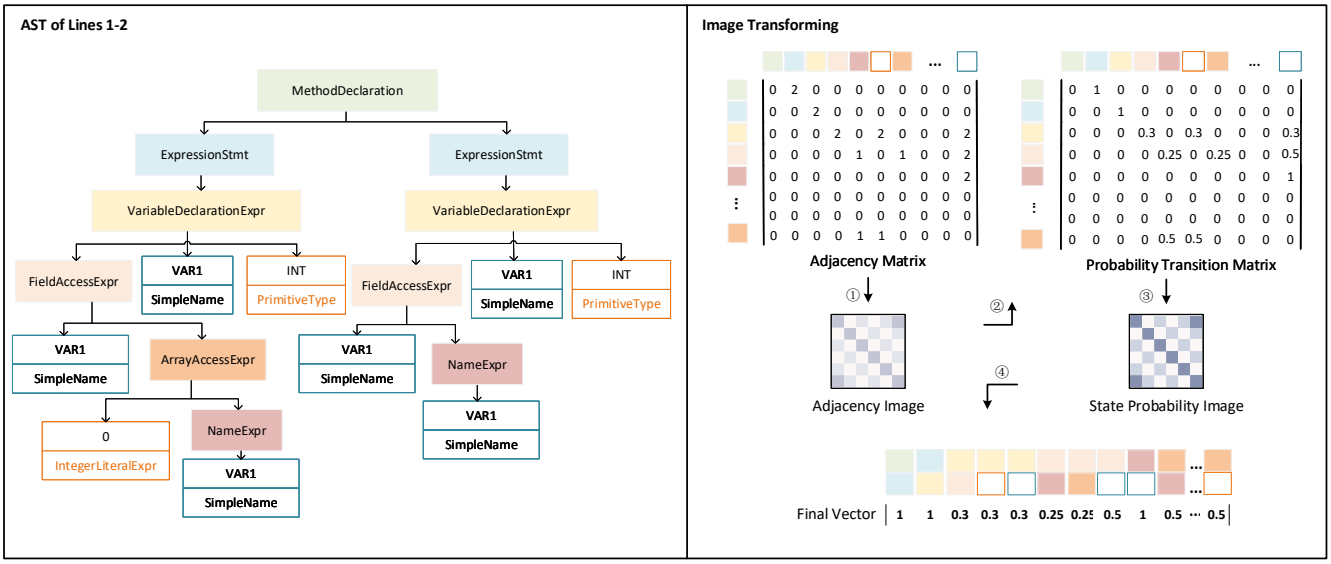


Fig. 4: A detailed example of structure encoding in *Code2Img*

does not reflect the structural details. So we design the image encoder part to further refine the image details and optimize the image structure. Specifically, we resort the Markov model to highlight the critical information of the adjacency matrix to improve the detection effectiveness. Markov chains assume that the probability of state transfer at a certain moment depends only on its previous state, effectively helping to simplify the complexity of the model and emphasize the structural details. Therefore, Markov chains are widely used in many time series models, such as *recurrent neural networks* (RNN) [25]. By considering the adjacency matrix as a transition matrix, it can be converted into an image based on the state probability transition. In detail, the img_1 denotes the adjacency matrix, and the img_2 denotes the probability transition matrix. The transformation equation is as follows.

$$img_2[i][j] = \frac{img_1[i][j]}{\sum_{k=0}^{70} img_1[i][k]} \tag{6}$$

In this way, the new image (namely, state probability image) is also 58×70 pixels, of which the pixel $img_2[i][j]$ presents the state transition probability from node type i to node type j in the AST.

In addition, we find that the AST is a non-loop structure, which means that not all node types of the AST will have edge relations between them. Therefore, the state probability image is sparse, with many pixels being zero (*i.e.*, the useless pixels). Unfortunately, the sparse images waste storage space and have a high time overhead for computing image similarity. To further optimize the image, we propose to remove the image's useless pixels in the image encoder part. Specifically, we analyze over 100-MLOC codes and generate the state probability images of their ASTs. The statistical results show that only 672 state transition probability has appeared in these state probability images. However, the state probability image can store the 4,060 types of transition probability (58×70 pixels), indicating that nearly 5/6 of the space of the image is wasted. Therefore, we set each of these 672 state transitions probability as an element to construct a one-dimensional vector. Finally, a AST can be represented by a 1×672 vector while preserving the structural information of the AST. As shown in final vector in Figure 4, the two colorful squares above each element represent the two states, respectively. The element indicates the probability of transferring from the top rectangle (the previous state) to the bottom rectangle (the next state).

In brief, in the image generation part, *Code2Img* first generates an adjacency image for the AST, representing the adjacency information of the AST. In the image encoder part, the adjacency image is then transformed into a state probability image to highlight the structural information. Finally, a one-dimensional vector is generated by removing useless pixels, thus improving detection efficiency.

3.5 Clone Detecting

In this part, *Code2Img* verifies whether the target source code and each suspected clone are a true clone. We calculate the Jaccard Similarity of the vectors generated by previous phases as formula(1). In addition, we set a verification threshold δ for Jaccard Similarity to distinguish between clones and non-clones. Specifically, if the similarity score $Jacc\_sim$ is greater than the threshold, the code pair is regarded a clone; otherwise, it is a non-clone.

4 Experiments

In this section, all experiments are centered on answering the following Research Questions (RQs):

- RQ1: What is the overall performance of *Code2Img* on code clone detection?
- RQ2: What is the detection performance of *Code2Img* compared to other state-of-the-art methods?
- RQ3: What is the time performance of *Code2Img* compared to other state-of-the-art clone detectors?

4.1 Experimental Settings

4.1.1 Dataset Description

We conduct experiments on a widely used dataset, BigCloneBench (BCB) [14]. It is manually labeled as "clone" and "non-clone" for more than 8,000,000 function pairs, where the "clone" labels include clone types from Type-1 to Type-4, as described in Section 2.1. It should be noted that T4 being semantic clones, identifying them is costly, while *Code2Img* focuses on large-scale clone detection. Therefore, we pay attention to other types rather than T4 clones.

4.1.2 Implementation

For the key parameters, *Code2Img* requires N for *N-lines*, filtration thresholds θ_1, θ_2, and verification threshold δ. To ensure that the parameter values are convincing, we refer to machine learning for parameter selection. Specifically, for parameter selection (RQ1), we randomly divide the dataset into five subsets, we pick one subset as the test set and the others as the training set. We use the training set to determine the critical parameters of *Code2Img* and the test set to test and report the final results. The value of N greatly impacts *Code2Img* in terms of time overhead and accuracy, so its selection needs to be demonstrated by experiment as described in Section 4.2. The threshold θ_1 is a threshold to exclude non-clones, which we set to 0.1, as done by other clone detectors [17], [16]. As for the threshold θ_2, it is a similarity parameter of *N-lines* to determine clones directly. Therefore, we determine the value of θ_2 by calculating the $code\_block\_sim$ for cloned and non-cloned pairs at different values of N, as described in Section 4.2. The validation threshold δ is a parameter of Jaccard similarity, whose value affects the accuracy of *Code2Img*, so we conduct detailed experiments in Section 4.2 to determine it.

For other experiment settings, we run all experiments on a standard server with 128GB RAM and 16 cores of CPU. It is important to note that we run all the experiments with a 32GB memory limit to ensure fairness, especially for large-scale clone detection. Besides, *Code2Img* mainly uses Javaparser [24] to complete data preprocessing.

4.1.3 Metrics

The metrics used to measure the effectiveness of *Code2Img* are the same as others [10], [26], [27], [28], [29].

- *True Positive* (TP): the number of samples correctly detected as clones.
- *True Negative* (TN): the number of samples correctly detected as non-clones.
- *False Positive* (FP): the number of samples incorrectly detected as clones.
- *False Negative* (FN): the number of samples incorrectly detected as non-clones.
- Accuracy=(TP+TN)/(TP+TN+FP+FN)
- Recall=TP/(TP+FN)
- Precision=TP/(TP+FP)
- F1=2∗Precision∗Recall/(Precision+Recall)

4.1.4 Comparative Tools

We compare *Code2Img* with eight existing state-of-the-art scalable code clone detectors that support scalable and complicated T3 (*i.e.*, MT3 clones):

- **SourcererCC** [15] calculates the overlapping similarity of the tokens of two methods to detect clones.
- **LVMapper** [17] detects large-variance clones by finding two similar sequences with more differences.
- **CCAligner** [2] uses a code window that considers e edit distance for matching to detect large-gap clones.
- **Siamese** [18] transforms Java code into four code representations to detect different types of clone.
- **NiCad** [7] detects clones by text-line comparison with the longest common subsequence algorithm.
- **NIL** [16] uses N-grams, an inverted index, and the LCS to detect large-variance clones.
- **Deckard** [19] clusters the characteristic vectors of each AST subtree using predefined rules of two methods to detect clones.
- **Yang2018** [20] generates an AST with function-level units and uses the Smith–Waterman algorithm to calculate the similarity score.

4.2 RQ1 : Overall Effectiveness

In this part, we focus on the detection effectiveness of *Code2Img* with different values of key parameters and select the most appropriate combination of them. Specifically, the key parameters include the clone threshold θ_2, validation threshold δ, and N for *N-lines*, as described in Section 4.1.2.

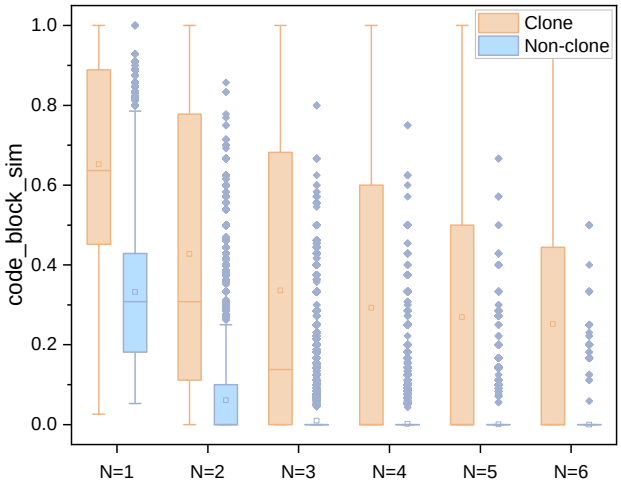


Fig. 5: The $code\_block\_sim$ with different N for *N-lines*

First, we experiment to determine the clone threshold θ_2 in the phase of clone filtering. It determines whether a code pair is directly detected as a clone in the filtering phase. We compute $code\_block\_sim$ for all code pairs in the training set, as described in Section 4.1.2. The distribution of $code\_block\_sim$ for clones and non-clones is shown in Figure 5. It can be seen that as N grows, the $code\_block\_sim$ of both cloned and non-cloned pairs becomes lower because the more lines of code a block contains, the less likely it is to be similar to the other. This means that taking too high a value for the clone threshold θ_2 will result in ineffective filtering of clone pairs and, conversely, incorrectly filtering out non-clone pairs. Therefore, to ensure the availability of *Code2Img* for most values of N, we set the clone threshold to 0.7. As shown in Figure 5, when θ_2 is 0.7, *Code2Img* can filter

out the apparent clone pairs while only incorrectly filtering out a small number of non-clones with N greater than 3.

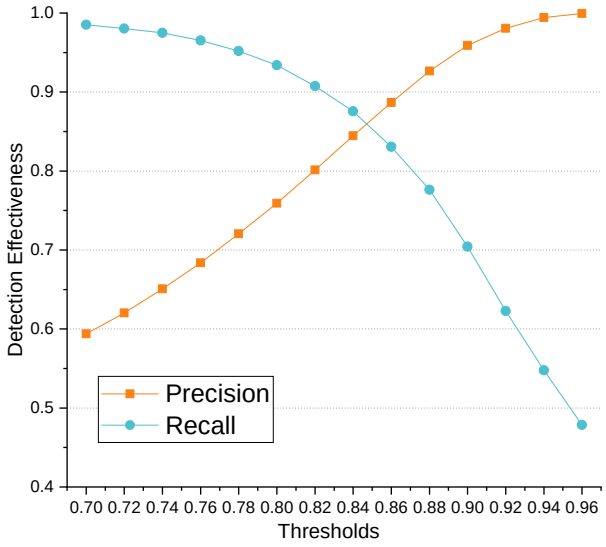


Fig. 6: The detection effectiveness of *Code2Img* with different validation thresholds δ

Second, we conduct an experiment to select the ideal value of the validation threshold δ. It is a critical parameter that determines whether a code pair is a clone or not in the clone detection phase, thus directly affecting the accuracy of *Code2Img*. The value of δ as a threshold for *Jaccard Similarity* ranges from 0 to 1. Besides, the most of the previous works have selected 0.7 as the threshold of similarity. To refine the experiments, we take values between 0.7 and 0.96 for the *Jaccard Similarity* with a step size of 0.02. In addition, since the precision and recall reflect the effectiveness of clone detection, we evaluate the effect of different thresholds of δ on them. The experimental results are shown in Figure 6, which is derived from training set. It can be seen that precision increases slowly after a threshold value greater than 0.9, while recall always decreases smoothly. Also, considering that a low precision causes too many false positives, we choose the ideal value of validation threshold δ as 0.9.

TABLE 2: Recall and execution time for each N value

N	1	2	3	4	5	6
T1	100	100	100	100	100	100
T2	100	100	100	100	100	100
VST3	100	100	100	100	100	99
ST3	100	99	96	89	79	66
MT3	79	70	60	34	23	15
T4	27	15	4	1	-	-
Execution time	12m31s	6m28s	2m12s	1m49s	1m47s	1m46s

For N of *N-lines*, its value directly affects the effectiveness of clone filtering. Specifically, the larger the value of N, the more lines of code are contained within a code block. Then, the fewer the number of code pairs sharing the *N-line* hash, the lower the value of *code_block_sim* will be. Therefore, not only less clone candidates are located in the clone candidates location phase but more pairs are filtered in the clone filtration phase. In other words, the larger the value of N, the fewer pairs of code need clone detection, resulting in shorter time overhead but worse detection results. So we need to choose a reasonable value of N to balance execution time and detection effectiveness. In addition, we exclude code fragments with less than six lines of code in the BCB dataset because these are so short and meaningless to detect. This also means that if we construct experiments with N greater than six, more code fragments will be removed, affecting our sample size. For this reason, we conduct experiments for N from 1 to 6 to evaluate the detection effectiveness of *Code2Img* for the BCB dataset, including the recall of each clone type and the overall execution time. Note that all the experiments are with a validation threshold δ of 0.9.

The results of the experiments for the value of N are shown in Table 2 from evaluating the training set. We can see that when $N = 1$ and $N = 2$, although the detection performance is highly advantageous, the execution time is significantly larger than the others. In addition, when the value of N is greater than 4, the time overhead decreases slightly while the recall decreases significantly, so we do not consider N as 5 or 6. The cases of $N = 3$ and $N = 4$ each have advantages, and both are acceptable in terms of detection effectiveness and time overhead. Therefore, we adopt both $N = 3$ and $N = 4$ parameter settings for *Code2Img*, making it suitable for different clone detection tasks. Specifically, we recommend the $N = 3$ setting for clone detection tasks with a small amount of code, for which the detection effectiveness is important. Conversely, for large-scale code clone detection, we recommend *Code2Img* with $N = 4$.

To conclude, *Code2Img* achieves satisfactory detection with a clone threshold θ_2 of 0.7 and validation threshold δ of 0.9. For the N value, *Code2Img* with N of 3 and 4 has both ideal time overhead and effectiveness, while each has its own strengths. Therefore, we recommend *Code2Img* with $N = 3$ for better detection performance for small-scale detection and N to 4 for more scalable clone detection for large-scale codes.

4.3 RQ2 : Effectiveness Comparison with Others

In this part, we conduct comparative experiments with other state-of-the-art scalable clone detectors on the BCB dataset. We focus on T1 to MT3 clone detection, as described in Section 4.1.1. However, to ensure the integrity of the experiment, we also perform experiments on T4 clones. We first evaluate *Code2Img* and comparative tools for recall of above clone types. The results of *Code2Img* are derived from the test set, as described in Section 2.1. Then we calculate the precision by randomly selecting 400 pairs of detected clones and manually verifying them, as previous research works [15], [17]. The results are shown in Table 3. It should be noted that columns *Code2Img(3)* and *Code2Img(4)* denote *Code2Img* with N values of 3 and 4 for the *N-lines*, respectively.

We can see that *Code2Img* achieves ideal results on precision and recall compared to other tools. For recall, both N values of *Code2Img* outperform the other tools for each

TABLE 3: Recall and precision results for BigCloneBench

Clone Type	Code2Img(3)	Code2Img(4)	CCAligner	SourcererCC	Siames	NIL	Nicad	LVMapper	Deckard	Yang2018
T1	100	100	100	94	100	99	98	99	60	100
T2	100	100	100	78	96	97	84	99	52	100
VST3	100	100	99	54	85	88	97	98	62	99
ST3	96	89	65	12	59	66	52	81	31	73
MT3	55	33	14	1	14	19	2	19	12	25
T4	4	1	-	-	1	-	-	-	1	-
Precision	98	100	61	100	98	86	99	59	35	95

clone type. Specifically, *Code2Img* achieves 100% recall for T1, T2, and VST3. This is because the text-similarity of these clone types is inherently higher, and it benefits from our normalization of the T2 variables, making *Code2Img* resilient to these clone types. For the other T3 types with large textual variance, *Code2Img* achieves the best performance. It is since *SourcererCC*, *NIL*, *LVMapper*, *Nicad*, *Siamese*, and *CCAligner* are all token-based methods, while *Code2Img* is an AST-based detector. In fact, tree representation can preserve the source code's structural information, so it can better capture the similarity of the structure of the code pair rather than staying at the textual similarity. It is the reason why *Code2Img* and *Yang2018* achieve good recall. However, *Deckard* detects clone by clustering the characteristic vectors of each AST's subtree with predefined rules. We find that more than half of the clones have diverse parser tree structures, especially T3 clones, resulting in *Deckard*'s low recall. Both *Code2Img* and *Yang2018* represent the node types of AST instead of the original nodes, but *Code2Img* also uses the Markov model to carve the structural details of the tree further and thus obtain a higher recall. For *T4* clone detection, *Code2Img* and all comparative tools perform poorly, as *Code2Img* and the comparative methods (as described in Section 4.1.4) are all clone detection tools oriented toward large-scale clone detection. Therefore, *Code2Img* obtains the highest recall among all the comparative tools.

The precision of *Code2Img* also performs well, especially much better than that of *NIL*, *LVMapper*, *CCAligner*, and *Deckard*. Because *NIL*, *LVMapper*, and *CCAligner* detect large-variance and large-gap clones, they have to consider more characteristics of these clones. As a result, some code blocks with continuous assignment statements (*e.g.,* constructors) and continuous if statements are incorrectly detected as clones, which leads to their lower precision to detect general clones. Additionally, *Deckard* clusters many subtrees of non-cloned ASTs together, causing massive false positives. For the other tools, their precision is close to or equal to 100%, like that of *Code2Img*. They sacrifice recall for more accurate clone detection, while *Code2Img* achieves good detection precision while maintaining high recall.

We then analyze the difference in detection effectiveness between *Code2Img(3)* and *Code2Img(4)*. As shown in Table 3, *Code2Img(4)* has higher precision but lower recall than *Code2Img(3)*. The reason is that *Code2Img(4)* treats every four consecutive lines as a code block while *Code2Img(3)* contains three lines of code in a block, resulting in a lower $code\_block\_sim$ for *Code2Img(4)*. In other words, with the same threshold value θ_1 of 0.1, *Code2Img(4)* filters out more code pairs than *Code2Img(3)*, which covers a part of T3 clones with low similarity. Therefore, *Code2Img(4)* has a lower recall than *Code2Img(3)*. However, since *Code2Img(4)* filters out those indistinguishable code pairs, it makes it easier for suspected clones (*i.e.,* code pairs that require clone detection) to be detected correctly and therefore have higher detection precision.

In summary, *Code2Img* with N of 3 and 4 obtains the best detection results on each clone type of the BCB dataset compared to *CCAligner*, *SourcererCC*, *Siamese*, *NIL*, *Nicad*, *LVMapper*, *Deckard*, and *Yang2018*.

4.4 RQ3: Scalability Evaluation

In this part, we focus on the runtime of *Code2Img* and other eight comparative clone detectors. All the scalability evaluations are run with a limited quad-core CPU and 12GB of memory, as done in previous studies [15], [2]. As for the dataset, we evaluate the time performance on the whole BCB dataset (*i.e.,* IJadataset [30]). Specifically, we use the CLOC [31] tool to divide it into seven sub-datasets of 1K-, 10k-, 100k-, 1M-, 10M-, 100M-, and 250M-LOC. Then we record the time overhead of each of the above datasets, respectively.

Table 4 shows the time performance of each detector for different input sizes. *Code2Img* almost achieves the best detection efficiency among the comparative tools. It can be seen that the other tree-based methods, *Deckard* and *Yang2018*, can only detect 1-MLOC, indicating that the complexity of the AST limits the scalability of traditional tree-based approaches. Not all token-based methods can meet the needs of large-scale clone detection. *CCAligner* and *Nicad* throw out-of-memory exceptions when detecting 100-MLOC and 250-MLOC codebase, which means that they are limited scalability. *Siamese* is able to complete 100-MLOC but has taken more than six days, indicating poor scalability, so we give up evaluating its 250-MLOC detection time. *LVMapper*'s detection time for 100-MLOC is over 17 hours, while *SourcererCC* takes just over 2 hours, but *SourcererCC*'s execution time at 250-MLOC is the longest. Therefore, these two tools are also not ideal for large-scale detection tasks. The detection time of *NIL* is relatively close to that of *Code2Img(4)*. *NIL* detects clones based on overlapping token similarity with an inherent advantage over *Code2Img* as an AST-based method. However, with the help of *Code2Img*'s

TABLE 4: Scalability results

LOC	Code2Img(3)	Code2Img(4)	CCAligner	SourcererCC	Siames	NIL	Nicad	LVMapper	Deckard	Yang2018
1K	0.548s	0.584s	1s	3s	4s	1s	1s	1s	1s	5s
10K	1s	1s	2s	5s	14s	1s	3s	1s	4s	16s
100K	3s	3s	6s	7s	45s	3s	36s	4s	32s	2m7s
1M	26s	25s	11m52s	37s	45m1s	11s	6m13s	34s	27m12s	1h45m3s
10M	9m10s	4m27s	29m48s	12m21s	14h11m	1m3s	2h10m	22m10s	Error	Error
100M	12h59m4s	1h37m52s	Killed	2h38m	6d16h25m	1h38m29s	Error	17h23m29s	-	-
250M	2d20h17s	7h39m32s	Killed	5d6h55m	-	7h40m7s	Error	3d13h47m	-	-

novel processing of ASTs, the complexity of ASTs is effectively reduced, saving the similarity computation time while preserving the code structure information. As a result, the detection performance of *Code2Img* is significantly better than that of *NIL* with a similar time overhead.

The time overhead of *Code2Img(4)* is significantly lower than that of *Code2Img(3)*. Since *Code2Img(4)* considers more lines of code as a block, it locates fewer clone candidates in the clone candidate location phase and also identifies fewer suspected clones in the clone filtration phase. Consequently, there are fewer code pairs in the clone detection phase. Naturally, *Code2Img(4)* loses part of the detection accuracy with less time overhead. Therefore, we recommend *Code2Img* with N of 4 (*i.e.*, *Code2Img(4)*) for large-scale clone detection tasks, which achieves the fastest detection of 100- and 250-MLOC among all the comparative tools.

To conclude, for small-scale code clone detection (within 10-MLOC), *Code2Img(3)* has an acceptable time overhead and achieves best detection results among the eight state-of-the-art comparative tools. *Code2Img(4)* is suitable for large-scale clone detection (more than 10-MLOC) with the lowest time overhead among the comparative detectors while maintaining good detection performance.

5 Discussion

5.1 Threat to Validity

First, we totally select 70 node types of the AST to obtain a certain number of pixels of an image. However, selecting these types may cause some inaccuracies since the total number of token types parsed by Javaparser is unclear. To mitigate the situation, we perform statistical analysis on the whole BCB dataset (more than 100-MLOC) to record the node types that have appeared and find that there are 58 type nodes and 12 token nodes. Second, the calculation of runtime overheads of *Code2Img* and its comparative tools may also cause some inaccuracies due to the different machine statuses, such as CPU usage. We mitigate the threat by conducting evaluations ten times and reporting the average runtime overhead in our paper. Third, the value settings of key parameters affect the detection performance of clone detector [32]. We conduct experiment with different values for key parameters (clone threshold θ_2, validation threshold δ, N of *N-lines*) to achieve an optimal result. The threat is minimized by employing the empirical value from the experimental results as the default parameter value. The value of the non-clone threshold θ_1 (*i.e.*, 0.1) refers to previous work [17], [16]. As shown in Figure 5, the value of θ_1 may cause some clones to be incorrectly considered as non-clones and therefore affect the recall in the detection results. In particular, the recall decreases rapidly for N greater than 4, as shown in Table 2. Fortunately, the testing results in Table 3 show that *Code2Img* still performs better than the comparative tools, even though there are some *False Negatives*. Finally, another threat lies in precision measurement. Since the BCB dataset contains more than 8,000,000 code pairs, there is no guarantee that code pairs marked as "non-clone" are indeed non-clones rather than being ignored during manual annotation. We randomly select 400 detected clones for three-person cross-validation to mitigate this threat, as in the previous work [15], [17].

5.2 Discussion

5.2.1 *Why* Code2Img *Perform Better*

In terms of effectiveness, *Code2Img* performs better than other scalable detectors. It is since the image representation generated by *Code2Img* is based on the AST, which preserves the structural feature of the source code. In addition, the Markov further helps *Code2Img* to highlight the structural details. This is the reason why *Code2Img* is more accurate than other AST-based methods for detection. However, other scalable detectors sacrifice consideration of code structure in order to guarantee a short time overhead. Therefore, these text- and token-based methods are only able to capture textual similarities, *i.e.,* T1 and T2 clones, while it is difficult to detect T3 clones since they are syntactically similar.

In terms of scalability, *Code2Img* completes the 250-MLOC detection in the shortest running time with a memory limit of 32GB, compared to other scalable detectors. The requirement for large-scale detection is code representation with small memory usage and an efficient similarity comparison algorithm, which coincide with the characteristics of images. Inspired by the image classification, we propose to transform the ASTs into images. The content of the images is encoded to cover the structural details of the AST and eventually converted into a one-dimensional vector. For one-dimensional vectors, the memory occupation is small, and the similarity computation time is very short due to its simplicity. Therefore, compared to other AST-based methods, *Code2Img* can achieve large-scale clone detection, while detecting complicated syntactic clones.

5.2.2 Selection of Comparative Tools

We choose six token-based clone detectors and two AST-based ones as our comparative tools. For token-based tools, we select them according to *NIL*'s comparison tools (*i.e.*, *LVMapper*, *CCAligner*, *SourcererCC*, and *NiCad*). Since they are suitable for scalable clone detection, the efficiency of *Code2Img* can be effectively demonstrated by comparing them. To construct a more comprehensive experiment, we also chose *Siamese*, another recent token-based clone detector, as the comparative tool. Note that although *LVMapper*, *CCAligner*, and *NIL* are clone detectors for large-gap and large-variance, they still report detection results for general code clones in the experiments of their papers. Therefore, they can also be regarded as state-of-the-art code clone detectors. In addition, since *Code2Img* is a tree-based method, we choose two other tree-based methods for comparison. Unfortunately, most tree-based methods are close-source, so we have to choose *Deckard*, an popular open-source tool, as a comparison tool. Then, we replicate a state-of-the-art AST-based clone detection approach, *Yang2018*. Naturally, there exist many learning-based clone detection methods. As we all know, machine learning or deep learning techniques require training sets, the larger the training set, the better the training model. However, the labeled datasets of BigCloneBench are unrepresentative and not large enough, resulting in poor generalization of these models. So we did not choose them as our comparative tools.

5.3 Future Work

Code2Img is an AST-based clone detector capable of detecting scalable complicated T3 clones. In fact, AST can characterize the semantic information of the code to some extent, but currently, *Code2Img* is not applicable to semantic clone detection. The image transformation of *Code2Img* constructs and optimizes the adjacency matrix of AST node types so that the image contains the structural information of AST. However, the images still lack the consideration of semantic information of the AST. We will integrate the semantic and syntactic information of AST into the images for future work to enable *Code2Img* to support semantic clone detection.

In addition, *Code2Img* analyzes a large amount of code and find that there are currently 70 node types in Java language parsed by Javaparser. Then *Code2Img* create the adjacency images based on these node types and egdes between them. However, the Java language evolves rapidly, with a new version being updated about every six months. New node types may appear in the version change. Therefore, in future work, we consider monitoring Java language releases and adding or removing node types automatically to make *Code2Img* more extensible.

6 Related Work

6.1 Scalable clone detection

Some research [33], [34], [35], [36] has illustrated the importance of large-scale clone detection. We categorize tools that can detect databases larger than 100-MLOC of code as scalable code clone detectors, including both research works and commercial tools.

The text-based and token-based detectors can satisfy the needs of large-scale detection. The reason is that the core of these methods is to calculate the similarity of text or token, which is simple and efficient to compute. *SourcererCC* [15] and *CloneWorks* [37] are both code detection cloning methods based on hybrid token and Index technologies, capable of detecting over 100-MLOC codes. *LVMapper* [17] is a token-based method that detects large-variance code clones with the idea of sequencing the alignment of large-variance code clones in bioinformatics. *Siamese* [18] uses four intermediate code expressions to compute the similarity, so while it can detect 100-MLOC codebase, the time overhead is significant. As an improved work of *LVMapper*, *NIL* [16] is a token-based detection method for large-gap and large-variance clones. *CCFinder* [5] and its successor *CCFinderX* [6] convert the target source code into a marker sequence and then use a suffix tree algorithm to detect clones. However, the token-based and text-based approaches lack the consideration of information on code semantics and thus are less effective in detecting T3 clones. In contrast, *Code2Img* achieves good results as an AST-based method detecting T3 clones.

The employment of GPU can effectively improve the clone detection speed, thus meeting the time requirements of large-scale detection. Solutions for GPGPU programming include Nvidia's CUDA [38] and AMD's CTM [39]. *Oreo* [40] is a clone detector that combines information retrieval, machine learning, and metric-based methods to make detection fast by leveraging Siamese Deep Neural Networks (DNN) [41] with GPU. *SAGA* [42] proposes an efficient postfix array-based detector with the help of GPU computing for detection acceleration. In fact, using GPU is a very effective method to accelerate computation, and we believe that future work using GPU to accelerate *Code2Img* can further improve its detection efficiency.

Some commercial software or open-source tools also state that they support clone detection of large codebases. Commercial software includes *BlackDuck* [43], *Scantist* [44] and *FOSSID* [45], which perform clone detection on their own internal codebases, but the amount of code in their codebases has yet to be discovered. Unlike *Code2Img*, which supports code clone detection of 250M-LOC databases at the fine granularity, some open-source software [46], [47], [48], [49] detects large-scale code clones at the file granularity.

In brief, the token-based, text-based, and GPU-using methods and tools can detect clones on a large scale. However, they are inferior to *Code2Img* for T3 clones detection, and *Code2Img* can also demonstrate a reasonable runtime overhead on the 250-MLOC codebase.

6.2 Complicated Type-3 clone detection

In addition to partial token-based approaches, code representation-based approaches can detect complicated T3 clones effectively.

For token-based method, *Nicad* [7] introduces a two-stage approach of identifying and normalizing potential clones using flexible pretty-printing and calculating similarity by the longest common subsequence algorithm of text lines. *CCAligner* [2] is also a token-based method applicable to detect T3 clones with large gaps. Nevertheless, above

tools do not scale to large codes and can only detect partial simple T3 clones, such as VST3 clones.

For code representation-based approaches, the tree-based approach is able to achieve a decent balance between large-scale and T3 clones detection. *TreeCen* [50] is a tree-based method that can detect semantic clones, and it conducts large-scale clone experiments for the Google Code Jam dataset [51], but the exact scale of the dataset is not mentioned. The methods based on code graph representation can capture precise code semantics [8], [52], [9], [11]. However, they perform poorly in large-scale detection tasks, as obtaining accurate graphs often requires compilation, and computing similarities for complex graphs is difficult.

In summary, compared to the token-based approach, *Code2Img* is able to achieve a better detection effectiveness for T3 clones while satisfying scalable detection. The code representation-based methods hardly support clone detection of 100-MLOC. However, as an AST-based method, *Code2Img* generates code representation images that can preserve the code semantics while occupying small memory space and reducing the complexity of similarity computation.

7 Conclusion

In this paper, we propose *Code2Img* with a tree-based image transformation for scalable complicated clone types detection. First, we set up a clone filtration by inverted index to filter out obvious clones and non-clones to reduce clone matches. Then we extract and normalize ASTs for the source code and transform them into simple image representations based on the adjacency matrix of the node types of ASTs. Then we encode the images with the Markov model to enhance the structural details within the images and drop out the useless pixels to refine the images. Finally, we obtain one-dimension vectors and detect clones by calculating their similarity. The experimental results show that *Code2Img* outperforms other eight state-of-the-art clone detectors (*i.e.*, *SourcererCC*, *NIL*, *LVMapper*, *Nicad*, *Siamese*, *CCAligner*, *Deckard*, and *Yang2018*). In terms of scalability, *Code2Img* completes the 250-MLOC clone detection within seven hours and 40 minutes, which is more efficient than other comparative methods. For detection performance, *Code2Img* also achieves the best detection results among the comparative tools, especially for complicated Type-3 clones.

Acknowledgments

We would thank the anonymous reviewers for their insightful comments to improve the quality of the paper. This work is supported by the the National Science Foundation of China under grant No. 62172168 and Hubei Province Key R&D Technology Special Innovation Project under Grant No. 2021BAA032.

References

[1] C. Wu, T. Wen, and Y. Zhang, "A revised cvss-based system to improve the dispersion of vulnerability risk scores," *Science China Information Sciences*, vol. 62, no. 3, pp. 1–3, 2019.

[2] P. Wang, J. Svajlenko, Y. Wu, Y. Xu, and C. K. Roy, "Ccaligner: a token based large-gap clone detector," in *Proceedings of the 40th International Conference on Software Engineering (ICSE'18)*, 2018, pp. 1066–1077.

[3] L. Li, H. Feng, W. Zhuang, N. Meng, and B. Ryder, "Cclearner: A deep learning-based clone detection approach," in *Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME'17)*, 2017, pp. 249–260.

[4] N. Göde and R. Koschke, "Incremental clone detection," in *Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR'09)*, 2009, pp. 219–228.

[5] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[6] T. Kamiya, "Ccfinderx: An interactive code clone analysis environment," in *Code Clone Analysis*, 2021, pp. 31–44.

[7] C. K. Roy and J. R. Cordy, "Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *Proceedings of the 16th International Conference on Program Comprehension (ICPC'08)*, 2008, pp. 172–181.

[8] Y. Wu, D. Zou, S. Dou, S. Yang, W. Yang, F. Cheng, H. Liang, and H. Jin, "Scdetector: software functional clone detection based on semantic tokens analysis," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE'20)*, 2020, pp. 821–833.

[9] G. Zhao and J. Huang, "Deepsim: Deep learning code functional similarity," in *Proceedings of the 26th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE'18)*, 2018, pp. 141–151.

[10] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *Proceedings of the 41st International Conference on Software Engineering (ICSE'19)*, 2019, pp. 783–794.

[11] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin, "Detecting code clones with graph neural network and flow-augmented abstract syntax tree," in *Proceedings of the 27th International Conference on Software Analysis, Evolution and Reengineering (SANER'20)*, 2020, pp. 261–271.

[12] Y. Bengio, R. De Mori, G. Flammia, and R. Kompe, "Global optimization of a neural network-hidden markov model hybrid," *IEEE transactions on Neural Networks*, vol. 3, no. 2, pp. 252–259, 1992.

[13] H. Wigström, "A model of a neural network with recurrent inhibition," *Kybernetik*, vol. 16, no. 2, pp. 103–112, 1974.

[14] "Bigclonebench," https://github.com/clonebench/BigCloneBench, 2022.

[15] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: Scaling code clone detection to big-code," in *Proceedings of the 38th International Conference on Software Engineering (ICSE'16)*, 2016, pp. 1157–1168.

[16] T. Nakagawa, Y. Higo, and S. Kusumoto, "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*, 2021, pp. 830–841.

[17] M. Wu, P. Wang, K. Yin, H. Cheng, Y. Xu, and C. K. Roy, "Lvmapper: A large-variance clone detector using sequencing alignment approach," *IEEE access*, pp. 27 986–27 997, 2020.

[18] C. Ragkhitwetsagul and J. Krinke, "Siamese: scalable and incremental code clone search via multiple code representations," *Empirical Software Engineering*, pp. 2236–2284, 2019.

[19] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*, 2007, pp. 96–105.

[20] Y. Yang, Z. Ren, X. Chen, and H. Jiang, "Structural function based code clone detection using a new hybrid technique," in *2018 IEEE 42nd annual computer software and applications conference (COMPSAC)*, vol. 1, 2018, pp. 286–291.

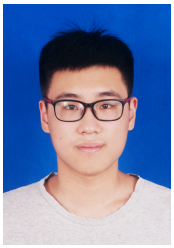
[21] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," *Queen's School of Computing TR*, vol. 541, no. 115, pp. 64–68, 2007.

[22] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Transactions on Software Engineering*, vol. 33, no. 9, pp. 577–591, 2007.

[23] J. Svajlenko and C. K. Roy, "Evaluating clone detection tools with bigclonebench," in *Proceedings of the 2015 IEEE International*

Conference on Software Maintenance and Evolution (ICSME'15), 2015, pp. 131–140.
[24] "Javaparser," https://github.com/javaparser/javaparser, 2022.
[25] W. Yin, K. Kann, M. Yu, and H. Schütze, "Comparative study of cnn and rnn for natural language processing," *arXiv preprint arXiv:1702.01923*, 2017.
[26] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proceedings of the 27th International Conference on Automated Software Engineering (ASE'16)*, 2016, pp. 87–98.
[27] H. Wei and M. Li, "Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code." in *Proceedings of the 2017 International Joint Conferences on Artificial Intelligence (IJCAI'17)*, 2017, pp. 3034–3040.
[28] X. Wang, Q. Wu, H. Zhang, C. Lyu, X. Jiang, Z. Zheng, L. Lyu, and S. Hu, "Heloc: Hierarchical contrastive learning of source code representation," *arXiv preprint arXiv:2203.14285*, 2022.
[29] L. Mou, G. Li, Z. Jin, L. Zhang, and T. Wang, "Tbcnn: A tree-based convolutional neural network for programming language processing," *arXiv preprint arXiv:1409.5718*, 2014.
[30] "Ambient software evolution group: Ijadataset 2.0," http://secold.org/projects/seclone, 2022.
[31] "Cloc: Count lines of code," http://cloc.sourceforge.net, 2022.
[32] T. Wang, M. Harman, Y. Jia, and J. Krinke, "Searching for better configurations: a rigorous approach to clone evaluation," in *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (FSE'13)*, 2013, pp. 455–465.
[33] R. Al-Ekram, C. Kapser, R. Holt, and M. Godfrey, "Cloning by accident: An empirical study of source code cloning across software systems," in *Proceedings of the 2005 International Symposium on Empirical Software Engineering (ISESE'05)*, 2005, pp. 10–30.
[34] M. Gharehyazie, B. Ray, and V. Filkov, "Some from here, some from there: Cross-project code reuse in github," in *Proceedings of the 14th International Conference on Mining Software Repositories (MSR'17)*, 2017, pp. 291–301.
[35] T. Ishihara, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto, "Inter-project functional clone detection toward building libraries-an empirical study on 13,000 projects," in *Proceedings of the 19th Working Conference on Reverse Engineering (WCRE'12)*, 2012, pp. 387–391.
[36] M. Mondal, C. K. Roy, and K. A. Schneider, "Does cloned code increase maintenance effort?" in *Proceedings of the 11th International Workshop on Software Clones (IWSC'17)*, 2017, pp. 1–7.
[37] J. Svajlenko and C. K. Roy, "Cloneworks: A fast and flexible large-scale near-miss clone detection tool," in *Proceedings of the 39th International Conference on Software Engineering (ICSE'17)*, 2017, pp. 177–179.
[38] "Nvidia," https://www.nvidia.com, 2022.
[39] M. Harris, S. Sengupta, and J. D. Owens, "Parallel prefix sum (scan) with cuda," *GPU Gems*, vol. 3, no. 39, pp. 851–876, 2007.
[40] V. Saini, F. Farmahinifarahani, Y. Lu, P. Baldi, and C. V. Lopes, "Oreo: Detection of clones in the twilight zone," in *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE'18)*, 2018, pp. 354–365.
[41] P. Baldi and Y. Chauvin, "Neural networks for fingerprint recognition," *neural computation*, pp. 402–418, 1993.
[42] G. Li, Y. Wu, C. K. Roy, J. Sun, X. Peng, N. Zhan, B. Hu, and J. Ma, "Saga: Efficient and large-scale detection of near-miss clones with gpu acceleration," in *Proceedings of the 27th International Conference on Software Analysis, Evolution and Reengineering (SANER'20)*, 2020, pp. 272–283.
[43] "Blackducks," https://www.blackducksoftware.com, 2022.
[44] "Scantist," https://scantist.com, 2022.
[45] "Fossid," https://fossid.com, 2022.
[46] C. V. Lopes, P. Maj, P. Martins, V. Saini, D. Yang, J. Zitny, H. Sajnani, and J. Vitek, "Déjàvu: A map of code duplicates on github," 2017, pp. 1–28.
[47] M. A. Nishi and K. Damevski, "Scalable code clone detection and search based on adaptive prefix filtering," *Journal of Systems and Software*, vol. 137, pp. 130–142, 2018.
[48] J. Ossher, H. Sajnani, and C. Lopes, "File cloning in open source java projects: The good, the bad, and the ugly," in *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM'11)*. IEEE, 2011, pp. 283–292.
[49] J. Svajlenko, I. Keivanloo, and C. K. Roy, "Scaling classical clone detection tools for ultra-large datasets: An exploratory study," in *Proceedings of the 7th International Workshop on Software Clones (IWSC'13)*. IEEE, 2013, pp. 16–22.
[50] H. Yutao, Z. Deqing, P. Junru, W. Yueming, S. Junjie, and J. Hai, "Treecen: Building tree graph for scalable semantic code clone detection," in *Proceedings of the IEEE/ACM 37th International Conference on Automated Software Engineering (ASE'22)*, 2022.
[51] "Google code jam," https://code.google.com/codejam/contests.html, 2022.
[52] W. Hua, Y. Sui, Y. Wan, G. Liu, and G. Xu, "Fcca: Hybrid code representation for functional clone detection using attention networks," *IEEE Transactions on Reliability*, vol. 70, no. 1, pp. 304–318, 2020.

Yutao Hu received the B.E. degree in Information Security from the China University of Geosciences (Wuhan), Wuhan, China, in 2019, and is currently pursuing the Ph.D. degree in Cyberspace Security at Huazhong University of Science and Technology, Wuhan, China. Her primary research interests lie in vulnerability detection and code clone detection.



Yilin Fang received the B.E degree in information security from Huazhong University of Science and Technology, Wuhan, China, in 2020, and is currently pursuing the M.E degree in cyber security at Huazhong University of Science and Technology, Wuhan, China. His research interests include vulnerability detection and code clone detection.

Yifan Sun is currently a master degree student at Huazhong University of Science and Technology, Wuhan, China. His primary research interests include vulnerability analysis and blockchain security.

Yaru Jia received the B.E. degree in Northwestern Normal University, Lanzhou, China, in 2020, and currently pursuing the master's degree at Huazhong University of Science and Technology, Wuhan, China. Her primary research interests lie in vulnerability detection.

Yueming Wu received the B.E. degree in Computer Science and Technology at Southwest Jiaotong University, Chengdu, China, in 2016 and the Ph.D. degree in School of Cyber Science and Engineering at Huazhong University of Science and Technology, Wuhan, China, in 2021. He is currently a research fellow in the School of Computer Science and Engineering at Nanyang Technological University. His primary research interests lie in malware analysis and vulnerability analysis.

Deqing Zou received the Ph.D. degree at Huazhong University of Science and Technology (HUST), in 2004. He is currently a professor of School of Cyber Science and Engineering, Huazhong University of Science and Technology (HUST), Wuhan, China. His main research interests include system security, trusted computing, virtualization and cloud security. He has always served as a reviewer for several prestigious journals, such as IEEE TDSC, IEEE TOC, IEEE TPDS, and IEEE TCC. He is on the editorial boards of four international journals, and has served as PC chair/PC member of more than 40 international conferences.

Hai Jin received the Ph.D. degree in computer engineering from Huazhong University of Science and Technology (HUST), Wuhan, China, in 1994. He is a Cheung Kung Scholars Chair Professor of computer science and engineering at HUST in China. He was awarded Excellent Youth Award from the National Science Foundation of China in 2001. He is the chief scientist of ChinaGrid, the largest grid computing project in China, and the chief scientists of National 973 Basic Research Program Project of Virtualization Technology of Computing System, and Cloud Security. He is a fellow of the IEEE, a fellow of the CCF, and a member of the ACM. He has co-authored 22 books and published over 700 research papers. His research interests include computer architecture, virtualization technology, cluster computing and cloud computing, peer-to-peer computing, network storage, and network security.