Current Trends in the Search for Similarities in Source Codes with an Application in the Field of Plagiarism and Clone Detection

Patrik Hrkút, Michal Ďuračík, Štefan Toth, Matej Meško
University of Žilina
Žilina, Slovakia
{Patrik.Hrkut, Michal.Duracik, Stefan.Toth, Matej.Mesko}@fri.uniza.sk

Abstract — **There are many methods and approaches for determining the similarity between two source codes. Many of them were inspired by developments in the field of NLP (Natural Language Processing) since the source text can be considered a special type of text. These methods have been implemented in many software tools and surprisingly, many of them are still in use (MOSS, JPlag). Artificial intelligence brought new procedures in the area of NLP, and they were also applied to the area of source code analysis. The article provides an overview of the methods for similarity detection in the source code, which we have not yet found in the literature to such an extent. Although it is certainly not exhaustive, it provides an overview of approaches from the oldest to those that are only beginning to gain attention at the present time.**

I. Introduction

Detection of the similarity of source codes or code snippets can be used for various purposes in the field of informatics. One of them, which we encounter most often, is the detection of plagiarism in source codes, which finds its application mainly in universities and their IT faculties. However, it can also be used in other applications, for example, searching for similar errors in existing code, code refactoring based on duplicate parts of the code detecting and simplifying the entire application code consequently.

We have been working on the detection of similar parts of code in our research for several years. This area, like other areas of informatics, has undergone very rapid development in recent years, which was mainly affected by artificial intelligence. Nevertheless, methods and tools developed earlier (we mean a horizon of 15-20 years in the past) are still used because they achieve results that are still applicable in solving many problems. For this reason, we decided to include these methods in our review as well. Several reviews of plagiarism detection tools based on searches for similarities in the source code are available in the literature [1], [2], [3], [4]. On the contrary, when studying the available literature, we did not find an overview of methods based on artificial intelligence, especially on neural networks, and this was the main impulse for the creation of this paper.

II. Similarity detection

A. Methods for detecting similarities

Source code is a text written in a programming language that differs from natural language in its grammar and syntax. On the other hand, since it is a text in a broader sense, it is possible to look at this type of text as a regular text. This thought led to the idea of using the ideas and methods used in the field of natural language processing (NLP) to analyze the source code. These procedures have proven to be only partially effective when processing source code, mainly to search for similarities [5], but many methods have found their inspiration precisely from similar methods that were developed for natural language. In particular, the first attempts to process the source code were based on lexical analysis methods. However, if we look at the source code as text, we do not capture its structure because we process it linearly (e.g., in the form of tokens). If we want to make better use of the code structure, we should consider using graphs. Several graph representations can be used to detect similarities in source codes, such as the Control flow graph (CFG), Program Dependence Graph (PDG), or Abstract syntax tree (AST). The development of artificial intelligence has also been shown in detecting similarities in source code. Many methods trained their models on graph representations (AST, CGF, etc.). Finally, deep learning methods (Deep Code Search) have found their application in this area as well. Whatever representation (tokens, graphs) is used to describe the source code, the result usually needs to be transformed into a one-dimensional sequence of data, which we can consider as a vector. If we need to determine the similarity of two vectors, a very often used method is to determine how close these vectors are to each other in space. For this, known metrics are used, which we will introduce below.

B. Metrics for determining similarity

If two source codes (snippets) are similar, their vectors representation created from their characteristics should also be similar. Similarity, as already mentioned, is defined as the spatial proximity of given vectors. Among the most common metrics used to determine the similarity of vectors, we can include the following:

- *Euclidean distance*: a measure of the distance between two points in space, which is calculated as the square root of the sum of the squares of the differences of their coordinates.
- *Cosine similarity*: a measure of the similarity of two vectors in space, determined as the cosine of the angle between them.
- *Manhattan distance*: a measure of the distance between two points in a plane according to the sum of the absolute values of the differences of their coordinates.
- *Jaccard coefficient*: a measure of the overlap of two sets, which is calculated as the ratio of the number of common elements to the total number of elements.

Several other metrics can be used, depending on the specific problem and the type of data being processed. However, in most of the methods described, either cosine similarity or calculation of the distance between two vectors using the Euclidean distance is used.

III. Methods based on lexical analysis

A. N-grams method

One of the oldest approaches is the use of similarity detection based on determining text similarity. The idea is based on comparing the substrings found in the compared source codes. It is ideal to compare all substrings against all other substrings, but this is computationally very demanding. Therefore, such a search is optimized using methods known as document fingerprinting [6]. With this method, a set of hash codes (*a fingerprint*) is created for each document, which dramatically reduces the total number of comparisons, making the whole process faster and less computationally demanding. The compared code is first preprocessed, when insignificant characters such as white characters are omitted, then n-grams are created from the text (An *n-gram* is a collection of n successive items in a source code that may include keywords, numbers, symbols, etc.). For a selected subset of n-grams, hash codes are calculated and used as a fingerprint of the document. Documents with a high number of matching fingerprints are marked as similarity candidates. The popular MOSS (Measure Of Software Similarity) system is based on this principle. The key to determining the similarity of source codes (or their sections) is the choice of fingerprints that will be selected for a given document. MOSS uses the *winnowing algorithm* [7] to select fingerprints. At the top level, it first applies a sliding window of a certain size to the list of hash codes. At each step, the algorithm records the rightmost minimum value in the window (if it has not been recorded already). When the window reaches the end of the sequence of hashes, the recorded set of hashes is taken as the fingerprint of the document.

Such a procedure significantly reduces the number of fingerprints required for similarity detection. According to the results produced, MOSS is therefore very effective, e.g. when looking for plagiarism. If the original intention was to compare only two projects, there were also implementations extending the search to the larger set of projects [7].

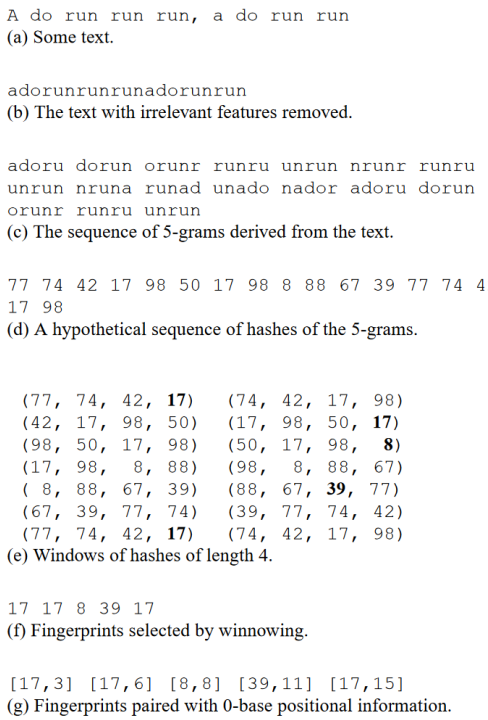

Fig. 1. Creating fingerprints in MOSS [8]

B. Token-based method

Another method based on syntactic analysis, is used by the JPlag system [9]. JPlag first converts the source code into tokens. Tokens are usually the smallest independent lexical units in source code that can be analyzed and processed by a parser. Tokens in the source code include e.g.

- Keywords
- Names of variables, functions, methods, classes
- Operators
- Literals
- Other characters which define the structure of the program.

Depending on the specific programming language, the definition of tokens may vary, but in general, tokens are considered the basic building blocks of source code. These tokens are connected into a long string, from which substrings are then selected for comparison. JPlag uses GST (*Greedy String Tiling*) algorithm to find *the Longest Common Subsequence* of a string. It is a heuristic algorithm based on one-to-one matching and is able to deal with the transposition of substrings [10].

To optimize the whole process, JPlag implements the *Running Karp-Rabin* algorithm [11], which uses sliding-window hashing to find exact pattern matches and gradually adjusts the hash values according to new data added to the text. The algorithms combination makes the JPlag system the perfect tool for searching for plagiarism in source codes, especially when comparing pairs of files (projects).

IV. METHODS USING GRAPH REPRESENTATIONS

A graph representation of source code creates a graph structure that represents the structure of the code, including its dependencies. In the graphical representation, a graph is created, where nodes represent code elements (e.g., classes, functions, variables) and edges represent relationships between them (e.g., function call, variable assignment). This approach allows more accurate identification of code similarity that might not appear when comparing a sequence of tokens. On the other hand, creating a graph representation of the code is more difficult and requires a more complex algorithm compared to token generation.

A. Control flow graph

A *control flow graph* (CFG) represents the flow of a program using a directed graph, where each node of the graph represents a basic block of the program (a basic block is a sequence of instructions that are executed without branching or jumping) and each edge corresponds to the flow of control between the basic blocks. A CFG is created by analyzing the code and tracing the control flow in the program. Similarity detection using (CFG) consists of analyzing the structure and flow of a program, instead of analyzing its syntactic structure.

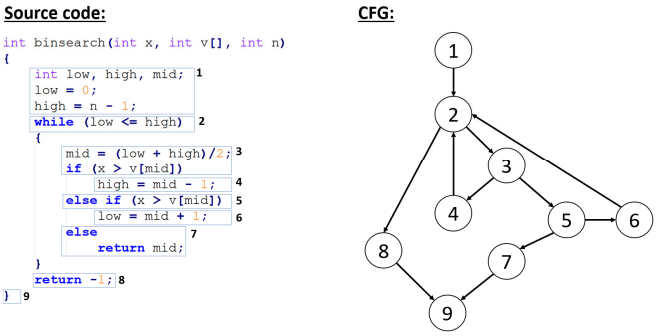

Fig. 2. An example of CFG [12]

When detecting similarities using CFG, graphs are first created for each source code file to be compared. CFGs are then compared using a graph-matching algorithm. The comparison usually results in lists of functions or methods that occur in multiple files and that have a similar structure or control flow. One tool for finding similarities using CFG is *SourcererCC* [13]. This tool uses the technique of cross-boundary function extraction, which makes it possible to identify parts of code that are connected to other parts of code through function interfaces. SourcererCC also uses other techniques to increase the accuracy of the similarity search, such as detecting plagiarism with different syntactic structures, searching for modified and moved copies, and others.

Other authors use CFG similarity detection to detect malware [14], or other security issues [15], but CFG is also used to detect plagiarism [16] [17] and [18].

B. Program Dependence Graph

Program Dependence Graph (PDG) [19] is a directed graph that represents dependencies between entities in source code, such as variables, function calls, assignments, and control structures. Unlike CDG, it contains not only application dependencies but also data dependencies [20].

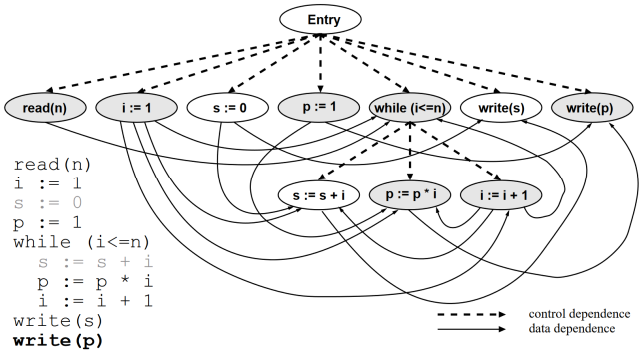

Fig. 3. An example of PDG [21]

Similarity detection using PDG is based on a comparison of the PDGs of two or more source code files [22], looking for similarities in the dependencies between the mentioned entities in the source codes. In a PDG, if the dependencies between entities are similar, the subgraphs are similar too. Similarity search algorithms in PDGs usually try to find isomorphic subgraphs, i.e. subgraphs that have the same structure and the same dependencies between entities [23]. When comparing PDGs, the syntactic structure of the program is usually ignored, as it focuses on the dependencies between the above entities. This means that similarity search algorithms using PDG can also detect similarities that have different syntactic structures but the same dependencies between entities.

PDGs are used in some plagiarism detection tools, such as *CloneDR*. The CloneDR algorithm works with both types of graphs (PDG and PDG), which are used for comparing programs and searching for plagiarism. The use of these types of graphs with various improvements appeared in the contributions of several authors and have applications in many fields of informatics (plagiarism [23], code refactoring [24], or software engineering [25]).

C. Abstract syntax tree

Abstract syntax tree (AST) [26] is a tree representation of the syntactic structure of the code, which allows to analyze and compare the source code at a higher level of abstraction. The advantage of AST is that it allows identifying code similarities even in cases where tokens in different code files are written differently, but the syntactic structure is the same. The AST can be obtained using a parser included with most compilers (C# – Roslyn, Java – JavaParser, PHP – PHPStan, C-clang, etc.).

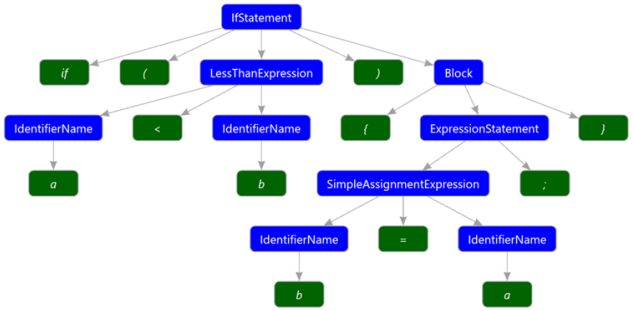

Fig. 4. An example of AST generated by Roslyn

When searching for similarities using AST, first an AST is created from the source code. Then, it is normalized by removing irrelevant nodes. The similarity measure can be calculated based on various factors, such as the size of the subtree, the number of matching nodes, the number of edits required to convert one tree (or subtree) to another, and so on. Since the comparison of graphic representations, or their parts are computationally demanding, trees are usually converted to vector representation (describing the characteristics of AST nodes). The similarity search then compares the vectors by some metric (described in section II.B).

This approach has become very popular and a large number of methods and tools have been created based on AST similarity detection, e.g. CCS (*Code Comparison System*) [27], a scalable architecture based on AST fingerprinting [28], a general-purpose search engine (CodEX) [29] or a plagiarism detection system based on fuzzy Petri nets and AST [30].

We also have dealt with this method of detecting similarity in our research. Our similarity search method was based on AST, which we used to represent source codes [31]. Unlike previous methods, we focused on finding similarities in large source code bases, where several problems had to be solved [32]. When optimizing the search of a large number of files and vectors representing individual parts of AST trees, we used the incremental clustering method [33], [34]. The result was the creation of a system for searching large-scale source codes, which is described in detail in [35].

V. Similarity Detection Using Neural Networks

The application of neural networks has also been found in the search for similarities in the source code since these networks make it possible to more accurately identify similarities between source codes, even if these codes are different in syntax and structure. Neural networks can also identify more complex patterns in code that previous algorithms would miss, and they can also be trained on various types of code. In addition, they can be trained on an amount of data that would be difficult for previous algorithms. In addition, they can learn from data, which means that there is no need to manually create rules to find similarities in the code. Many methods are taken from NLP and modified for source code processing needs. We will detail some of them in the next sections.

A. Siamese Neural Network

Siamese Neural Network (SNN) [36] is a type of neural network that compares two input patterns and determines how similar they are to each other. An SNN consists of two identical branches that have the same architecture and share weights. Each branch takes one of the input patterns and converts it to a vector. This idea can also be applied to comparing source codes. The authors in [37] use *Term Frequency-Inverse Document Frequency* (TF-IDF) used in NLP to determine the weights of a series of word vectors. Then the SNN model is built to learn the semantic vector representation of code snippets. Cosine similarity is used to determine the similarity measure. A similar detection approach with slightly different vector generation for the SNN input is mentioned in [38].

B. CodeBERT and CodeGraphBERT

The *CodeBERT* neural network [39] is based on the transformer architecture [40], which was originally used in the language model BERT (*Bidirectional Encoder Representations from Transformers*), in NLP domain. However, CodeBERT, unlike BERT, was specifically designed for working with code and uses several techniques to improve performance on tasks that are specific to programming. The CodeBERT training process uses the masked language modeling (MLM) technique that is also used in BERT, but with some modifications to better deal with the specifics of the source codes. The technique randomly select some tokens from the code and replace them with the *[MASK]* token, then train the network to predict these masked tokens. CodeBERT also uses the cross-lingual pre-training technique, which means that the network is trained on several programming languages.

The model trained with CodeBERT is represented in the form of vectors. Similar to the previous cases, the similarity measure between the vectors can be calculated to obtain the comparison results using CodeBERT. Another approach to evaluating similarity is offered by the *CodeBERTScore* calculation [41]. CodeBERT is also used in other applications, e.g. in the automatic correction of bugs in the source code [42], or for detecting code vulnerabilities [43].

CodeGraphBERT [44] is a neural network model for source code representation, again based on the transformer architecture [40]. Unlike CodeBERT, CodeGraphBERT uses the CDG concept – code representation using a graph, where vertices represent tokens and edges represent relationships between tokens. Various code sources such as GitHub and other open-source repositories were used to train CodeGraphBERT. The training data was chosen to include different programming languages, libraries, and applications. When using CodeGraphBERT, code is represented using language tokens and syntax trees. The model can then generate a vector representation of the code suitable for comparison..

C. Code2Vec

Code2Vec [45] is a machine-learning model used to represent source code. It works based on contextual vector representations, which are obtained using the word2vec method (NLP) [46]. Unlike the word2vec model, which works with tokens in the text, code2vec works with the AST tree of the source code. When training a model, an AST is first built for each feature in the training data. Then, the AST for each function is represented by a low-dimensional vector that captures the structure and meaning of the code. After creating an AST for each function in the training data, these ASTs are assembled into one large AST, the root of which is a fictitious function. This large AST is then trained using the following code prediction method, i.e., the code following the current code is predicted. Compiling this large AST from individual function ASTs ensures that the model captures the global structure of the code, not just the structure of individual functions [45].

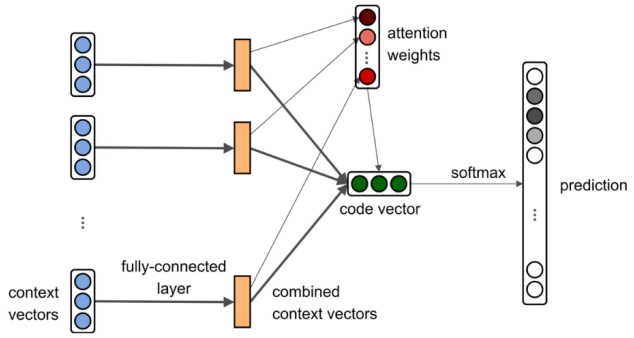

Fig. 5. Neural network architecture used in code2vec [45]

Similarity detection in this method is again based on comparing vector representations of the source code. If the vector representations of the code are close to each other in space, it means that the codes are similar. Applications of this approach can be found e.g., in [47], where code2vec is used to classify sections of the source code, solving code reusability [48], or searching for semantically similar code libraries [49].

D. Abstract syntax tree neural network (AST NN)

As we already mentioned, an abstract syntax tree (AST) is a tree representation of program code that captures the structure of the language and the relationships between different parts of the code. AST NN [50] is neural network designed to work with AST. The source code processing process begins by converting the source code to an AST. Then this representation of the code is used as input to a neural network that is trained to recognize certain patterns in the code. These patterns can be, for example, identifying certain types of variables or detecting the presence of certain patterns in the code.

For each node in the AST tree, a vector is created representing its properties, such as the type of the node, its children, attributes, and so on. These vectors are subsequently processed using a neural network that can learn the relationships between individual nodes and their context in the AST tree. Once the network is trained, it can be used to compare vector representations of different AST trees. Based on the distance between the vectors, it is possible to determine the degree of similarity between the codes. To determine the similarities, the known metrics for determining the similarities of vectors, which we mentioned in the text above, are used again.

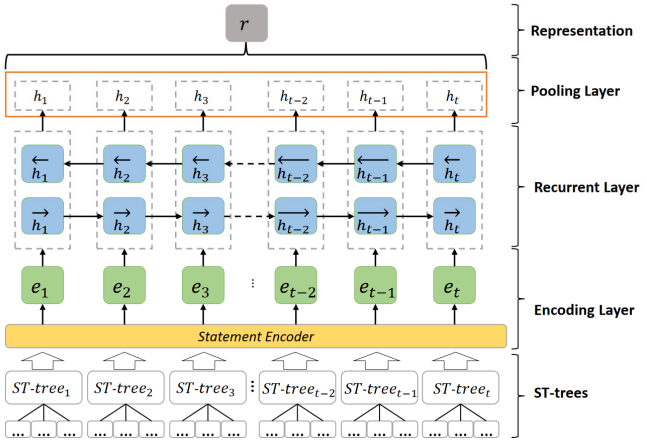

Fig. 6. The architecture of AST-based Neural Network [50]

AST NNs are often used to solve tasks such as code classification, code generation, code error detection, or finding similarities in source code. In the literature, we can find its use to identify and classify software vulnerabilities [51] or to locate potential bugs in the application [52].

E. Deep Code Search

Deep Code Search [53] is a technique used to search for relevant code in large source code repositories. Neural networks used in Deep Code Search are usually trained using a pre-training technique on a large amount of code and then fine-tuned for specific tasks such as finding code similar to the given code. The result of the trained model is a set of vectors that describe the given code. At the search input, a vector is created from the compared part of the source code and this vector is compared with the vectors trained by the entire neural network (cosine similarity is used).

As a proof-of-concept application, the authors implement a code search tool named *DeepCS* using the proposed model. They evaluated DeepCS on a large scale codebase collected from GitHub [53]. The collected dataset consisted of methods programmed in java programing language. They compare their approach with *Lucene* and *CodeHow* tools. They experiments showed, that their approach was effective and outperformed the related approaches.

VI. Discussion

One of the most common applications of finding similarities in source codes is plagiarism detection. We have been dealing with this issue for several years and have published several articles [31]-[35], so in the discussion we will focus mainly on this area of application of the described methods.

Currently, as mentioned in the article, there exist different approaches to searching for plagiarism or clones in the source code. Some of these methods have also found application in the real world.

SourcererCC and *CloneDR* are among the basic tools for searching for source code clones that developers routinely utilize. In addition to these tools, we commonly find proprietary implementations of these algorithms directly in integrated development environments (IDE). These algorithms primarily serve developers to remove duplication in the source code. Algorithms found in IDEs are usually implemented on top of syntax trees and are used for code refactoring. Practice shows that even algorithms that do not use methods based on artificial intelligence are sufficient for detecting duplication in the source code.

In the case of searching for plagiarism in the source code, the situation is similar. Currently, freely available tools that can be used to search for plagiarism in the source code include the *MOSS* and *JPlag*. Both systems use on relatively well-known algorithms that are based on lexical analysis methods. Our experience shows that JPlag and MOSS system is sufficient to detect many plagiarism types. Both systems can detect various source code modifications. When looking for plagiarism in the source code, we need to answer a question what we can consider plagiarism. We often come across situations where a whole group of students has the same assignment and thus each of them must create a basically similar application. It is obvious that with a properly specified assignment, two exactly the same source codes should not be created under normal circumstances. On the contrary, when we look at the problem from a higher perspective, every single solution should lead to the same goal. In these cases, the features of some algorithms, thanks to which they can identify plagiarism even at a higher logical level, would be counterproductive. In practice, we believe that modifying the source code in such a way that it is not detected as plagiarism using the JPlag and MOSS systems requires a certain amount of knowledge and time from the student. This effort is often greater than the effort needed to create his own solution. If the student submitted plagiarism due to a lack of knowledge, in most cases he does not have enough knowledge how to alter the code itself.

In addition to freely available solutions, we can now encounter a boom in commercial systems. Currently, the most famous systems include copyleaks.com, codequiry.com and codeio.com. The advantage of these systems over JPlag and MOSS is that they contain their own database of source codes against which they can compare embedded solutions. This database allows searching for plagiarisms even outside the group of works for which we are looking for matches. We also mention in [34], that this database must be able to efficiently provide stored data. Effective indexes [54] in database systems must be used to make the data available effectively. In general, it is not possible to easily find out what algorithm these systems use because they consider it a trade secret. All of them state using AI to some extent.

VII. Conclusion

In our article, we discussed the methods for detecting similarities in the source code. The basic algorithms developed for detecting similarities in source code based on similarity detection in text documents are still used today. They show good results even in the current era. Most of the currently used tools for detecting plagiarism in source code are built on these algorithms (MOSS, JPlag). In recent years, the development of algorithms in this area has been strongly influenced by advances in the field of neural networks. Thanks to this, several new algorithms have been created, which are based on well-known algorithms for generating AST trees and vectors, and they apply innovative methods based on artificial intelligence to these structures. These innovative approaches improve the detection capabilities, which in the future will enable the creation of new systems that will use these algorithms. The given list of different approaches is certainly not exhaustive, but it gives an overview of the direction the field is heading. In the future, we can expect even more sophisticated models using neural networks, especially deep learning.

References

[1] A. M. E. T. Ali, H. M. D. Abdulla, and V. Snášel, “Overview and Comparison of Plagiarism Detection Tools,” in Databases, Texts, Specifications, Objects, 2011.

[2] Z. Duric and D. Gaević, “A Source Code Similarity System for Plagiarism Detection,” Comput. J., vol. 56, pp. 70–86, 2013.

[3] A. Bugarín-Diz, M. Carreira, M. Lama, and X. Pardo, “Plagiarism detection using software tools: A study in a Computer Science degree,” Apr. 2005.

[4] M. Agrawal and D. K. Sharma, “A state of art on source code plagiarism detection,” in 2016 2nd International Conference on Next Generation Computing Technologies (NGCT), 2016, pp. 236–241. doi: 10.1109/NGCT.2016.7877421.

[5] M. Ďuračík, E. Krsak, and P. Hrkút, “Using concepts of text based plagiarism detection in source code plagiarism analysis,” Apr. 2017.

[6] Y. Kim and S. Ross, “An Approach to Document Fingerprinting,” Apr. 2015. doi: 10.1007/978-3-319-27974-9.

[7] D. Sheahen and D. Joyner, “TAPS: A MOSS Extension for Detecting Software Plagiarism at Scale,” Apr. 2016, pp. 285–288. doi: 10.1145/2876034.2893435.

[8] S. Schleimer, D. S. Wilkerson, and A. Aiken, “Winnowing,” in Proceedings of the 2003 ACM SIGMOD international conference on Management of data, New York, NY, USA: ACM, Jun. 2003, pp. 76–85. doi: 10.1145/872757.872770.

[9] L. Prechelt and M. Phlippsen, “JPlag: Finding plagiarisms among a set of programs,” 2000.

[10] M. Wise, “String Similarity via Greedy String Tiling and Running Karp−Rabin Matching,” Unpublished Basser Department of Computer Science Report, Apr. 1993.

[11] R. M. Karp and M. O. Rabin, “Efficient randomized pattern-matching algorithms,” IBM J Res Dev, vol. 31, no. 2, pp. 249–260, Mar. 1987, doi: 10.1147/rd.312.0249.

[12] R. Al-Ekram and K. Kontogiannis, “Source code modularization using lattice of concept slices,” Apr. 2004, pp. 195–203. doi: 10.1109/CSMR.2004.1281420.

[13] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V Lopes, “SourcererCC: Scaling Code Clone Detection to Big-Code,” in 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), 2016, pp. 1157–1168. doi: 10.1145/2884781.2884877.

[14] P. P. F. Chan and C. Collberg, “A Method to Evaluate CFG Comparison Algorithms,” in 2014 14th International Conference

on Quality Software, 2014, pp. 95–104. doi: 10.1109/QSIC.2014.28.
[15] S. Cesare and Y. Xiang, “Malware Variant Detection Using Similarity Search over Sets of ControlFlow Graphs,” Apr. 2011, doi: 10.1109/TrustCom.2011.26.
[16] D.-K. Chae, J. Ha, S.-W. Kim, B. Kang, and E. G. Im, “Software plagiarism detection: A graph-based approach,” in International Conference on Information and Knowledge Management, Proceedings, Apr. 2013, pp. 1577–1580. doi: 0.1145/2505515.2507848.
[17] Y. Li, J. Jang, and X. Ou, “Topology-Aware Hashing for Effective Control Flow Graph Similarity Analysis,” CoRR, vol. abs/2004.06563, 2020, [Online]. Available: https://arxiv.org/abs/2004.06563
[18] H.-Y. Tsai, Y. Huang, and D. Wagner, “A Graph Approach to Quantitative Analysis of Control-Flow Obfuscating Transformations,” IEEE Transactions on Information Forensics and Security, vol. 4, pp. 257–267, Apr. 2009, doi: 10.1109/TIFS.2008.2011077.
[19] G. Snelting, T. Robschink, and J. Krinke, “Efficient path conditions in dependence graphs for software safety analysis,” ACM Transactions on Software Engineering and Methodology, vol. 15, no. 4, pp. 410–457, Oct. 2006, doi: 10.1145/1178625.1178628.
[20] J. Ferrante, K. J. Ottenstein, and J. D. Warren, “The program dependence graph and its use in optimization,” ACM Transactions on Programming Languages and Systems, vol. 9, no. 3, pp. 319–349, Jul. 1987, doi: 10.1145/24039.24041.
[21] K. Androutsopoulos, D. Clark, M. Harman, J. Krinke, and L. Tratt, “Survey of Slicing Finite State Machine Models,” Apr. 2023.
[22] J. Krinke, “Identifying Similar Code with Program Dependence Graphs,” Apr. 2001, pp. 301–309. doi: 10.1109/WCRE.2001.957835.
[23] Z. Zhang, H.-H. Yan, and X.-W. Zhang, “Code Similarity Detection by Program Dependence Graph,” in Proceedings of the 2016 International Conference on Computer Engineering and Information Systems, Paris, France: Atlantis Press, 2016. doi: 10.2991/ceis-16.2016.50.
[24] T. Henderson and A. Podgurski, “Sampling code clones from program dependence graphs with GRAPLE,” Apr. 2016, pp. 47–53. doi: 10.1145/2989238.2989241.
[25] S. Horwitz and T. Reps, “The use of program dependence graphs in software engineering,” in Proceedings of the 14th international conference on Software engineering - ICSE ’92, New York, New York, USA: ACM Press, 1992, pp. 392–411. doi: 10.1145/143062.143156.
[26] G. Fischer, J. Lusiardi, and J. von Gudenberg, “Abstract Syntax Trees - and their Role in Model Driven Software Development,” in International Conference on Software Engineering Advances (ICSEA 2007), 2007, p. 38. doi: 10.1109/ICSEA.2007.12.
[27] B. Cui, J. Li, T. Guo, J. Wang, and D. Ma, “Code Comparison System based on Abstract Syntax Tree,” in 2010 3rd IEEE International Conference on Broadband Network and Multimedia Technology (IC-BNMT), 2010, pp. 668–673. doi: 10.1109/ICBNMT.2010.5705174.
[28] M. Chilowicz, E. Duris, and G. Roussel, “Syntax tree fingerprinting for source code similarity detection,” in 2009 IEEE 17th International Conference on Program Comprehension, 2009, pp. 243–247. doi: 10.1109/ICPC.2009.5090050.
[29] M. Zheng, X. Pan, and D. Lillis, “CodEX: Source Code Plagiarism Detection Based on Abstract Syntax Trees,” Apr. 2018.
[30] V. Shen, “Novel Code Plagiarism Detection Based on Abstract Syntax Tree and Fuzzy Petri Nets,” International Journal of Engineering Education, vol. 1, pp. 46–56, Apr. 2019, doi: 10.14710/ijee.1.1.46-56.
[31] M. Ďuračík, E. Kršák, and P. Hrkút, Source code representations for plagiarism detection, vol. 870. 2018. doi: 10.1007/978-3-319-95522-3_6.
[32] M. Ďuračík, E. Kršak, and P. Hrkút, “Issues with the Detection of Plagiarism in Programming Courses on a Larger Scale,” in ICETA 2018 - 16th IEEE International Conference on Emerging eLearning Technologies and Applications, Proceedings, 2018. doi: 10.1109/ICETA.2018.8572260.
[33] P. Hrkút, M. Ďuračík, M. Mikušová, M. Callejas-Cuervo, and J. Zukowska, Increasing K-Means Clustering Algorithm Effectivity for Using in Source Code Plagiarism Detection, vol. 1154 CCIS. 2020. doi: 10.1007/978-3-030-46785-2_10.
[34] M. Ďuračík, E. Kršák, and P. Hrkút, “Searching source code fragments using incremental clustering,” Concurr Comput, vol. 32, no. 13, 2020, doi: 10.1002/cpe.5416.
[35] M. Duracik, P. Hrkut, E. Krsak, and S. Toth, “Abstract syntax tree based source code antiplagiarism system for large projects set,” IEEE Access, vol. 8, 2020, doi: 10.1109/ACCESS.2020.3026422.
[36] J. BROMLEY et al., “SIGNATURE VERIFICATION USING A ‘SIAMESE’ TIME DELAY NEURAL NETWORK,” Intern J Pattern Recognit Artif Intell, vol. 07, no. 04, pp. 669–688, Aug. 1993, doi: 10.1142/S0218001493000339.
[37] C. Xie, X. Wang, C. Qian, and M. Wang, “A Source Code Similarity Based on Siamese Neural Network,” Applied Sciences, vol. 10, no. 21, p. 7519, Oct. 2020, doi: 10.3390/app10217519.
[38] Y. Wu and W. Wang, “Code Similarity Detection Based on Siamese Network,” in 2021 IEEE International Conference on Information Communication and Software Engineering (ICICSE), IEEE, Mar. 2021, pp. 47–51. doi: 10.1109/ICICSE52190.2021.9404110.
[39] Z. Feng et al., “CodeBERT: A Pre-Trained Model for Programming and Natural Languages,” CoRR, vol. abs/2002.08155, 2020, [Online]. Available: https://arxiv.org/abs/2002.08155
[40] A. Vaswani et al., “Attention Is All You Need,” Jun. 2017.
[41] S. Zhou, U. Alon, S. Agarwal, and G. Neubig, “CodeBERTScore: Evaluating Code Generation with Pretrained Models of Code,” Feb. 2023.
[42] E. Mashhadi and H. Hemmati, “Applying CodeBERT for Automated Program Repair of Java Simple Bugs,” in 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), 2021, pp. 505–509. doi: 10.1109/MSR52588.2021.00063.
[43] X. Yuan, G. Lin, Y. Tai, and J. Zhang, “Deep Neural Embedding for Software Vulnerability Discovery: Comparison and Optimization,” Security and Communication Networks, vol. 2022, pp. 1–12, Jan. 2022, doi: 10.1155/2022/5203217.
[44] D. Guo et al., “GraphCodeBERT: Pre-training Code Representations with Data Flow,” Sep. 2020.
[45] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, “code2vec: Learning Distributed Representations of Code,” CoRR, vol. abs/1803.09473, 2018, [Online]. Available: http://arxiv.org/abs/1803.09473
[46] T. Mikolov, K. Chen, G. Corrado, and J. Dean, “Efficient Estimation of Word Representations in Vector Space,” Jan. 2013.
[47] B. Arora, S. VC, G. R. Dheemanth, M. Thakral, and N. S. Kumar, “Code Semantic Detection,” in 2021 Asian Conference on Innovation in Technology (ASIANCON), 2021, pp. 1–6. doi: 10.1109/ASIANCON51346.2021.9544660.
[48] B. RamyaSree, B. Ramakrishna, M. I. Harshitha, A. Kavya, P. Reshvanth, and N. V Krishna Rao, “Code Component Retrieval Using ode2Vec,” in 2021 Fifth International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC), IEEE, Nov. 2021, pp. 1044–1048. doi: 10.1109/I-SMAC52330.2021.9640648.
[49] M. Talari, K. C. N, and C. R. K. Reddy, “CODE2VEC Based Cognitive Agent System to Retrieve Relevant Code Component from Repository,” E3S Web of Conferences, vol. 184, p. 01064, Aug. 2020, doi: 10.1051/e3sconf/202018401064.
[50] 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 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), 2019, pp. 783–794. doi: 10.1109/ICSE.2019.00086.
[51] G. Partenza, T. Amburgey, L. Deng, J. Dehlinger, and S. Chakraborty, “Automatic Identification of Vulnerable Code: Investigations with an AST-Based Neural Network,” in 2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC), 2021, pp. 1475–1482. doi: 10.1109/COMPSAC51774.2021.00219.
[52] H. Liang, L. Sun, M. Wang, and Y. Yang, “Deep Learning With Customized Abstract Syntax Tree for Bug Localization,” IEEE Access, vol. PP, p. 1, Apr. 2019, doi: 10.1109/ACCESS.2019.2936948.

[53] X. Gu, H. Zhang, and S. Kim, “Deep code search,” in Proceedings - International Conference on Software Engineering, IEEE Computer Society, May 2018, pp. 933–944. doi: 10.1145/3180155.3180167.
[54] M. Kvet, J. Papán, “The Complexity of the Data Retrieval Process Using the Proposed Index Extension”, IEEE Access, vol. 10, 2022.