CodEX: Source Code Plagiarism Detection Based on Abstract Syntax Trees

Mengya Zheng[1,2], Xingyu Pan[1,2] and David Lillis[1,2]

[1] Beijing Dublin International College, University College Dublin, Ireland
[2] School of Computer Science, University College Dublin, Ireland
{mengya.zheng,xingyu.pan}@ucdconnect.ie, david.lillis@ucd.ie

Abstract. CodEX is a source code search engine that allows users to search a repository of source code snippets using source code snippets as the query also. A potential use for such a search engine is to help educators identify cases of plagiarism in students' programming assignments. This paper evaluates CodEX in this context. Abstract Syntax Trees (ASTs) are used to represent source code files on an abstract level. This, combined with node hashing and similarity calculations, allows users to search for source code snippets that match suspected plagiarism cases. A number of commonly-employed techniques to avoid plagiarism detection are identified, and the CodEX system is evaluated for its ability to detect plagiarism cases even when these techniques are employed. Evaluation results are promising, with 95% of test cases being identified successfully.

1 Introduction

CodEX is a general-purpose search engine to allow users to search for source code snippets, developed as part of an undergraduate final year project. The user interface supports three different methods providing queries to the system:

- *keyword-based* search: obtains keywords from the user and returns source files relevant to these keywords from GitHub[3].
- *description-based* search: allows users to input questions or descriptions of code, and returns relevant StackOverflow[4] answers to similar questions.
- *source code snippet* search: obtains a complete source code structure (code snippet) from the user, and returns source files containing similar code snippets from GitHub.

One potential use for the source code snippet search feature is the detection of plagiarism among programming assignments. Given a repository of existing source code, a suspected case of plagiarism can be used as a search snippet, with CodEX providing a list of similar code snippets. This paper focuses on this use

[3] https://github.com
[4] http://stackoverflow.com/

case, in the context of five common tactics used by students to avoid detection and a combination of these.

Due to the syntax differences of variant source code languages, different APIs or supporting external classes are needed to implement the source code snippet searching feature. CodEX applies a general algorithm based on Abstract Syntax Trees (ASTs) to solve this problem. Initially, Python and Java code snippets were chosen as examples, since their AST structures vary to some degree. Successfully applying this approach to multiple languages indicates the potential for extending the system to other programming languages in the future.

In Section 2 of this paper, related work is listed and discussed with comparison to CodEX; in Section 3, the implementation approach of the source code snippet search feature is illustrated in detail; evaluation results are displayed in Section 4 with brief analysis; the last section draws conclusions and outlines the future work of this project.

2 Related Work

Multiple code search engines already exist that support keyword-based searching in multiple programming languages, such as searchcode[5], codas[6], unspsc[7], Debian[8]. etc. CodEX distinguishes itself from these through its code-snippet searching module and plagiarism detection. Regarding code snippet searching, there are already several pioneers who have achieved high performance in this area [6, 14, 2]. A good example is Sourcerer [2], which breaks down each piece of code into different entities, uniquely identifiable elements from the source code, and then abstract these entities into fingerprints. To integrate these entities into an organic whole, relations between these entities' fingerprints are recorded.

Other variations of fingerprinting are frequently proposed by Information Retrieval researchers. Among these, Syntax Tree Fingerprinting [4], is a significant inspiration for the implementation of CodEX. In this work, fingerprints were used to represent nodes of Abstract Syntax Tree [10]. These fingerprints contain a weight, a hash value and other important information of the node for code snippet comparison. Based on this research, CodEX combines several useful and portable techniques including AST, entity division, node weight and hash values to implement the code-snippet searching algorithm, while sacrificing some less important information such like entity relations and other fields in Syntax tree fingerprints as they can be suggested by the hierarchical structure of AST.

Regarding plagiarism among programming assignments, multiple commonly used techniques for avoiding plagiarism detection should be considered when developing source code plagiarism detection tools [5]. To deal with these techniques, several pioneers have achieved high performance in their work. One example is a fingerprinting approach called Winnowing [12], which divides source code into

[5] `http://searchcode.com`
[6] `http://www.codase.com`
[7] `http://www.unspsc.org/search-code`
[8] `https://codesearch.debian.net`

k-grams (where k is chosen by users), and chooses a subset of these k-grams for comparison based on their hash values. This approach sacrifices full comparison between source files to achieve high efficiency. Al-Khanjari et al. [1] developed a Java program plagiarism detection tool that applied Attribute Counting Metrics (ATMs) [13]. This was illustrated to be less effective than the structural metrics employed by Kristina and Wise [13], as structural information is more important for source files. Greenan [7] applied the length of the longest common subsequence (LCS) algorithm [3] to calculate a similarity score between two method-level code snippets in a line-by-line manner, which gives positive points for matching lines and negative points for mismatching lines. This method works well for method-level code comparison because the types of code structures are known to be the same, and additional significant information such like method names, parameters and return types are easy to record as independent fields. However, it is difficult to abstract all the significant information of an entire source file and store them in independent fields. Moreover, the code structures of two compared code snippets are unknown and possibly different, making it impossible to compare them line-by-line.

3 Methods

In this section, the idea and advantages of ASTs are discussed in detail initially. Following this, a number of techniques are identified in Section 3.2 that are frequently used by students to avoid plagiarism detection. The implementation strategy for code snippet searching is then demonstrated in Section 3.3.

3.1 Abstract Syntax Trees (ASTs)

As an abstract form of source code structure, ASTs capture the structural features of code snippets and represent them in a hierarchical tree structure. Compared to Concrete Syntax Trees (CSTs) [15], another type of syntax tree used to give an exact representation of source code, ASTs are more suitable for code snippet searching because they represent source code at an abstract level where unimportant elements such as grammar symbols can be ignored as noise. Therefore, only the structure and general meaning of source files are considered for comparison.

This advantage makes a significant difference for the implementation: AST node types and attributes are classified to declare the component usage and function, which allows noise to be removed easily. The fact that ASTs translate the structure of source code into parent-child or sibling relationships among tree nodes enables the efficient depth-first top-to-bottom traversal strategy [8]. To support code snippet searching for Java and Python, two supporting tools are required. Firstly, the Python AST API[9] is used to process Python source file. Secondly, Javalang[10] is a package implemented in Python that provides AST

[9] `https://docs.python.org/3/library/ast.html`
[10] `https://pypi.org/project/javalang/`

generators for Java source files. As Javalang is not a fully-developed AST API, it was necessary to implement additional methods to fulfill necessary analysis for Java source files.

3.2 Techniques for Avoiding Plagiarism Detection

In developing this module, it was necessary to identify methods that students use to avoid plagiarism detection. Additionally, certain differences between source code submissions would have no effect on the overall functionality of the code, but would cause a simple matching algorithm to fail. Six diverse techniques were initially identified for particular focus, as follows:

- *output statement insertion/removal*: inserting or removing output statements (e.g. "print" in Python) are inserted into the original code.
- *comments insertion/removal*: inserting or removing comments to the original code.
- *identifier name modification*: modifying identifier names (variable names, method names, class names etc.).
- *change code order*: changing the order of some pieces of source code to avoid plagiarism detection (e.g. changing the order of function declarations). However, only order changes between sibling nodes are relevant to this technique, while those between parent-child nodes are not plagiarism techniques as they have a functional effect on the source file.
- *combine multiple code blocks from multiple source files*: copying multiple code blocks from different source files where each piece of copied code is not serious enough to be defined as plagiarism. This technique cannot be easily detected by traditional source code search engines which simply evaluate the similarity between the input code and individual source files.
- *meaningless code insertion*: this is the trickiest plagiarism technique that challenges plenty of researchers in the field [9]. It is difficult to define what kind of code is "meaningless", as this description is fuzzy. To give it a clearer description, "meaningless" code includes those code without any functional effect on the source file, e.g. unused variable declaration statements, code blocks that are never reached.

3.3 Implementation

The implementation of the source code snippet search feature follows the classical information retrieval process: preprocessing, indexing, searching. These are addressed in the following sections.

Source File Pre-processing As is shown in Figure 1, in the pre-processing phase, the source code snippets are firstly parsed into ASTs. Next, some noise (unimportant nodes in green) is removed from these trees. Some plagiarism techniques related to noise are detected during this step. Here, four types of noise

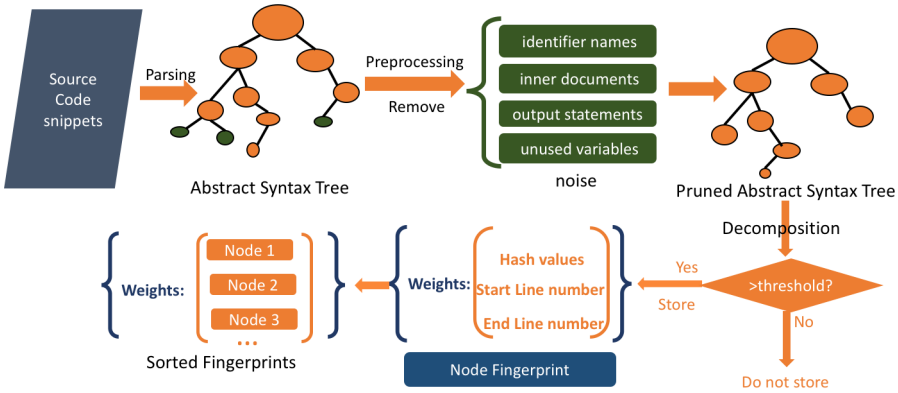

Fig. 1. Pre-processing & Indexing Phases.

are removed: identifier names, inner documents (comments), output statements and unused variables. These unimportant nodes are pruned from Python ASTs and Java ASTs with different strategies.

For the cleaning step in Python source files, a "visitor" is implemented to traverse the AST of each code snippet and invoke diverse removal operations on different types of noise. This visitor is an instance of a class implementing the abstract `NodeTransformer` class provided by Python AST API. In this class, multiple methods are defined and implemented for different types of nodes to make modifications to their attribute values and return the modified nodes. These methods are implemented as filters for source files. For each type of noise, there is a specific filter implemented in the visitor to remove it from the original nodes. When removing the identifier names, only those identifier names defined within the code snippets are removed but those standard class/method names defined by external classes are retained as they have a functional effect on the source files. Next, all the code snippets are serialized as Strings in nested lists to represent the hierarchical structures. However, for Java source file code cleaning, it was necessary to implement additional serialization and noise cleaning strategies as they are not provided or directly supported by Javalang.

Unused variables, the variables that are defined or initialized but never called in the source file are also detected in this phase, as they are regarded as one category of meaningless elements. However, removing them demands another traversal of the entire tree as the list of unused variables can only be obtained after the first traversal. Early experimentation indicated that the time required for this process was excessive and so it was removed from the current version of the system in order to improve the response time for users. Including a more

efficient implementation is left to future work, which assumes more available computational resources.

In summary, noise cleaning is performed during the preprocessing phases for both Python and Java, after which the output insertion/removal, inner documents (comment) insertion/removal and identifier name modifications techniques are all addressed. Also, a partial solution to the "meaningless code insertion" problem is also proposed and demonstrated in this phase. Therefore, what has been left to the latter phases are those plagiarism techniques involving structural modifications.

Indexing In the indexing phase, the cleaned source code is represented by appropriate data structures for further comparison and calculation. As Figure 1 shows, after the preprocessing phase, the source files have been cleaned and turned into pruned ASTs. Nevertheless, there are still some steps to execute before the similarity calculation. To summarise this process: firstly, a fingerprint is generated for each tree nodes to represent its relevant information. In order to avoid redundancy, not all nodes need to be analyzed and stored; instead, only important nodes are stored for further comparison. Therefore, a benchmark determining what constitutes an "important" node nodes is determined, which also reflects the granularity of code snippet searching algorithms. Secondly, following the suggestion in [10], a weight is associated with each fingerprint in order to measure each node's contribution to the similarity calculation. Hence, the weight information of all code components needs to be calculated and recorded during the indexing phase. Thirdly, to show users the matching areas between two source files, line numbers and node weights are calculated during each recursion of the serialization step as this information is not fully provided by the AST APIs or other supporting classes. As displayed in Figure 1, these weights are stored along with the fingerprints and the nodes' hash values. Next, all sibling nodes are sorted in each abstract syntax tree in order to deal with the order-change plagiarism technique. Finally, the source files in the corpus are indexed to different data structures, which has a substantial effect on the time complexity in searching phase.

These three key tasks in the indexing phase are discussed in detail as follows:

1. **Node Hashing Granularity**: As mentioned above, in the indexing phase, the code snippet is decomposed into relatively fine-grained code components of diverse granularity. Here, "relatively fine-grained" means that the code structure is decomposed thoroughly enough to expose all matching elements, while some tiny elements such as constants and parameters are not decomposed as independent nodes to avoid a cumbersome comparison process. This does not mean that these tiny elements will be ignored but they are contained within some bigger nodes. However, now that tiny elements need to be treated in a different way, a specific benchmark deciding what elements are "tiny" should be determined in advance. This benchmark is essential because it also determines the granularity of the indexing phase, which directly affects the accuracy and efficiency of the search algorithm. Inspired by the

code clone detection research of Nilsson et al. [11] and Chilowicz's syntax tree fingerprinting method [4], a specific weight threshold is pre-defined as this benchmark. As demonstrated in Figure 1, all the nodes with weights below this threshold are regarded as "tiny" nodes. After empirical testing, this threshold was set to 10, which had the effect of filtering nodes such as simple variable declarations and short Boolean operations. It is likely that this threshold will be different for other source languages. AST nodes with weights above the threshold are stored in as fingerprints, which consist of their weight, line number information and hash values. Before hashing, the serialized form of an abstract syntax tree is a long string containing all nodes' information in its nested structure. The hashing step turns these long strings into uniquely-compressed Strings of fixed length, thus facilitating the time efficiency of further code comparison as the lengths of compared strings have been shortened to a same fixed value.

2. **Sibling Sorting:** A change in order between sibling nodes may frequently have no functional effect on a code snippet, in contrast to parent-child node pairs. However, this feature can be exploited to avoid plagiarism detection. For example, function definitions are siblings and their ordering does not affect the execution of the program, but changing their order would fool some text-based plagiarism detection approaches. Therefore, sibling nodes need to be re-arranged to tackle relative plagiarism techniques. Hence all sibling node lists are sorted.
3. **Index Structures:** In Chilowicz's fingerprinting method [4], all the subtrees are classified based on their weights and sorted in decreasing order, enabling a top-to-bottom comparison between code snippets in which high-level nodes are compared before their child nodes are checked. This method avoided having to make comparisons between small subtrees in the bottom level, however it did not achieve high time efficiency. The main factor lowering the efficiency is the plain structure of code snippets, where all node fingerprints are classified into separated lists mapping to their corresponding weights (Sorted Fingerprints in Figure 1). This structure ignores the tree hierarchies and leads to repetitive comparison between subtrees of the same weight. In order to facilitate the time efficiency of code snippet comparison process, the hierarchical structure is restored in the structure used to represent queries, which enable an efficient depth-first traversing algorithm for the searching phase.

Searching To search for suspected plagiarism cases, the user supplies a program or code snippet. This is passed through two different search strategies. Firstly, the query code is compared with all code snippets in the index to obtain a similarity score for each. The output of the search is a ranked list of code snippets from the index based on their similarity to a query code snippet.Secondly, a global similarity score is computed, by comparing the query to the corpus as a whole. Those snippets from the corpus that have contributed to this global score (assuming a suspected plagiarism case has been detected) are made available to

the user. Given that plagiarism detection requires a manual judgment (not all similar code is plagiarised), these search outputs are aimed at providing the user with the most likely sources of plagiarised code.

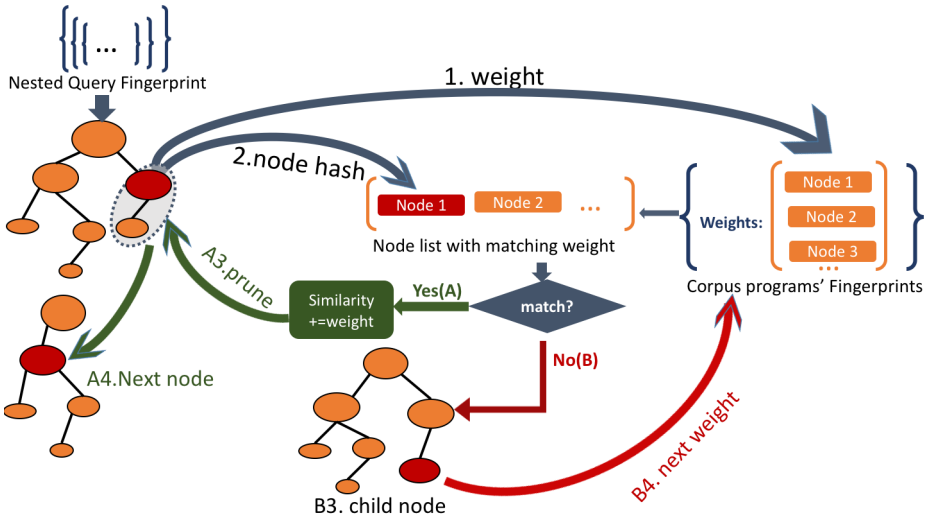

Fig. 2. Depth-First Searching Algorithm.

Local Similarity Matching (weight-based depth first search): Figure 2 shows the mechanism for calculating local similarities. The similarity score is calculated during depth-first traversal through a nested query fingerprint. This query fingerprint is represented as a tree in this diagram, as it is based on an abstract form of its original AST hierarchy.

The first step is to search the weight of current query node within the indexed dictionary of the corpus, in which each different weight is a key mapping to a list of corresponding node fingerprints. The corresponding fingerprint list is extracted out for comparison. Next, the second step is the comparison between the hash values of the query nodes and that of all nodes in the list. In this stage, a matching judgment is made to each node in the fingerprint list, and invokes two branches A and B for further processing:

– **Branch A**: If some node in the fingerprint list has the same hash value as the this query node, then this judgment goes to the Branch A (the green branch in the diagram), where the similarity score is increased by the weight of the matching node. After the similarity score is updated, the subtree derived from this query node is pruned from the query tree in the third step (A3). The tree-pruning step aims to avoid further unnecessary comparison between its subnodes and the corpus, since the matching of current query

node also implies matching of all its subtrees. This is why in the searching phase the nested indexed structure can achieve higher time efficiency than plain indexed structure proposed in [4] (because tree pruning cannot be done to the flat index structure). After step A3, another recursion is triggered for the next node. This new node can have the same weight as the last one, which means that they are in the same list mapped to one weight, whereupon there is no need to search the node weight in the corpus fingerprint list again. By contrast, if the new node has a smaller weight than the previous node, it needs to be searched in the fingerprint list again.

- **Branch B**: If there is no node in the fingerprint list with the same hash value as the query node, Branch B is triggered (the red branch in the diagram). In this branch, whether the sub-nodes of the current query node can find a matching target in further recursions is unknown as no match is found in current recursion. Therefore no subtree-pruning step is required in this branch. Therefore, new recursions are triggered in step B3. In new recursions, the target weight becomes smaller and is searched within the corpus dictionary again, after which the same procedures happen to the new query node.

After the traversal is complete, a similarity score is calculated as the total weight of matching code blocks from each source file in the corpus. A threshold is pre-defined to indicate whether a similarity score is high enough to indicate a suspected plagiarism case. The similarity scores are normalised as the proportion of the matching weight over the total weight of query code snippet to avoid snippet length having an impact. The suspected plagiarism threshold must consequently lie between 0 and 1. Empirical testing resulted in a threshold of 0.6 being identified. Although this threshold achieved good accuracy, it still needs to be tuned with more test cases in the future stages for higher accuracy.

Global Similarity & Matching Block Threshold: Global similarity is the similarity score between the query code snippet and the whole corpus, which is different from the similarity between each single source file in the corpus (local similarity) as there is only one global similarity score for each query. The aim of global similarity is to detect plagiarism queries with multiple copied code blocks from the different sources, which cannot be shown by local similarity when each copied code block is not big enough to be defined as plagiarism. However, this global similarity makes false positives more likely, as a query may have a high global similarity based on matching many small code snippets throughout the corpus. To avoid this issue, only "big" matching code blocks are used in the global similarity calculation. Therefore, another threshold needs to be set for defining "big" matching blocks (called the "blockThreshold"). This threshold is based on the weight of a code block, and empirical testing indicates that a weight of 50 is suitable. This had the effect of filtering out scattered small pieces of code blocks while retaining complete code structures such as big loops and methods. In addition, at most one matching block is taken from a source file to calculate the global similarity. This is to avoid defining normal queries with multiple scat-

tered matching blocks as plagiarism. Therefore, among all matching blocks from each source file, the matching code block with the highest weight is selected to contribute to the global similarity calculation.

After all matching blocks are gathered, the sum of the weight of these code blocks is calculated. Next, the similarity score is normalised as the proportion of this sum over the total weight of query code. Finally, this global similarity score is compared with the pre-defined plagiarism threshold. If the threshold is exceeded, this query is suspected to have copied blocks from multiple source files, and these component source files are marked in the result list.

4 Results

To evaluate the accuracy of the code-snippet searching function and the reliability of its plagiarism detection functionality, both original code snippets and plagiarism code snippets were input to this searching module as test cases. For each of the techniques to avoid plagiarism detection identified in Section 3.2, 10 pairs of original & "plagiarised" code snippets were designed using corresponding techniques. Of these, 5 pairs of test cases are in Python and the other 5 are in Java. Additionally, 10 cases were also created whereby a mixture of all the individual techniques were employed.

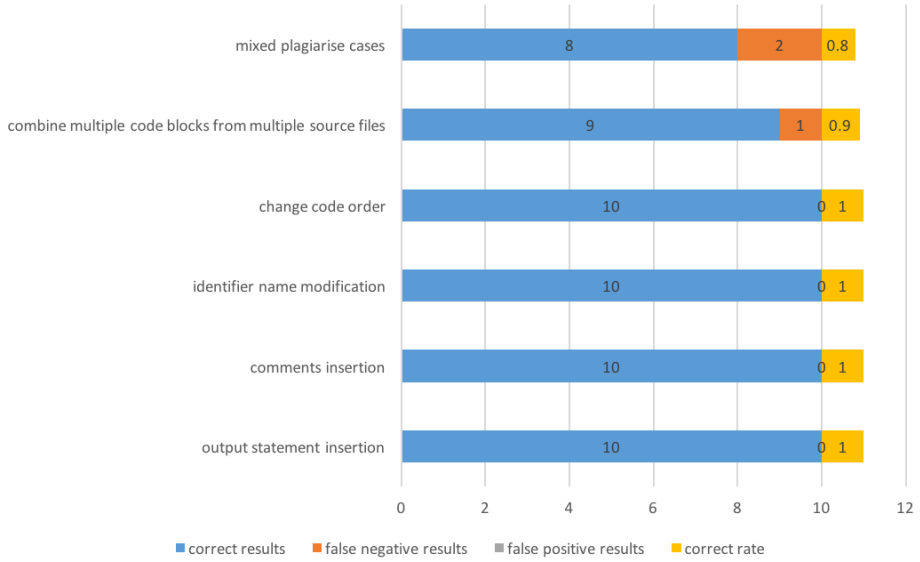

Fig. 3. Source Code Plagiarism Detection Searching Evaluation Results.

Firstly, in the context of code searching in general, no false-positive issue was observed in the evaluated results. Despite the abstractions used in search-

ing, only source files containing similar code snippets are returned, in which the matching lines between the input query and source files are highlighted for comparison. For most test cases, the returned results ranked in the top 10 all contain multiple matching lines. For some other tested code snippets, there are only 7 or 8 returned results containing matching lines, and these test cases are all self-designed structures that are not commonly used. In general, the code-usage searching function achieves precision of 0.84 and is proved to be helpful to find the relative usage of query code snippets.

Regarding the plagiarism detection function, as Figure 3 shows, all similarity scores assigned to plagiarism test cases purely applying identifier name modifications, output insertions and sibling-code-order changes are 1. which suggests that these techniques are unlikely to succeed in avoiding plagiarism detection in the CodEX system. For the "combining multiple code blocks" technique, each case consists of four code snippets, and contains at least one original block and three copied source code blocks. These copied source code blocks vary in both size and distribution patterns. According to the evaluation results, 90% of these plagiarism cases can be detected by CodEX and are assigned high global similarities (in excess of the similarity threshold), and the only one plagiarism case that was not detected as the plagiarism case is found to contain small scattered copied code blocks.

Apart from test cases about one single plagiarism technique, the 10 mixed test cases applying all the plagiarism techniques are also designed to test the overall plagiarism detection accuracy of CodEX. 80% of these mixed-plagiarism source files are judged as plagiarism cases by the system and 20% of them are not detected as plagiarism cases as their copied code blocks are scattered and relatively small.

In summary, CodEX achieves accuracy of 84% for the source code snippet search, and achieves high performance on detecting all the five plagiarism techniques introduced before as well as the mixed plagiarism cases based on them, with a 95% success rate overall.

5 Conclusion

As a source code search engine, CodEX supports code snippet search and source code plagiarism detection, distinguishing itself from most traditional code search engines. As is demonstrated by the evaluation results, five commonly-used plagiarism techniques can be detected accurately. Additionally, a partial solution to the "meaningless code insertion" issue is proposed. Therefore, CodEX brings convenience to both programmers and education staff, and its high accuracy proves the feasibility of several innovated code searching algorithms. For the future work, all the thresholds should be tuned further with larger quantities of test cases to achieve higher performance. Additionally, removing unused variables and code was not implemented in this version due to speed constraints; an efficient mechanism for this can be added in future. Finally, CodEX should be extended for more programming languages other than Java and Python.

References

1. Al-Khanjari, Z.A., Fiaidhi, J.A., Al-Hinai, R.A., Kutti, N.S.: Plagdetect: A java programming plagiarism detection tool. ACM Inroads 1(4), 66–71 (Dec 2010), http://doi.acm.org.ucd.idm.oclc.org/10.1145/1869746.1869766
2. Bajracharya, S., Ngo, T., Linstead, E., Dou, Y., Rigor, P., Baldi, P., Lopes, C.: Sourcerer: a search engine for open source code supporting structure-based search. vol. 2006, pp. 681–682. ACM (2006)
3. Bukh, B., Ma, J.: Longest common subsequences in sets of words. SIAM Journal on Discrete Mathematics 28(4), 2042–2049 (2014)
4. Chilowicz, M., Duris, ., Roussel, G.: Viewing functions as token sequences to highlight similarities in source code. Science of Computer Programming 78(10), 1871–1891 (2013)
5. Clough, P.: Plagiarism in natural and programming languages: an overview of current tools and technologies (2000)
6. Gitchell, D., Tran, N.: Sim: A utility for detecting similarity in computer programs. SIGCSE Bull. 31(1), 266–270 (Mar 1999), http://doi.acm.org.ucd.idm.oclc.org/10.1145/384266.299783
7. Greenan, K.: Method-level code clone detection on transformed abstract syntax trees using sequence matching algorithms. Student Report, University of California-Santa Cruz, Winter (2005)
8. Korf, R.E.: Depth-first iterative-deepening: An optimal admissible tree search. Artificial intelligence 27(1), 97–109 (1985)
9. Li, Y., Wang, L., Li, X., Cai, Y.: Detecting source code changes to maintain the consistence of behavioral model. pp. 1–6. ACM (2012)
10. Neamtiu, I., Foster, J., Hicks, M.: Understanding source code evolution using abstract syntax tree matching. ACM SIGSOFT Software Engineering Notes 30(4), 1–5 (2005)
11. Nilsson, E.: Abstract syntax tree analysis for plagiarism detection (2012)
12. Schleimer, S., Wilkerson, D., Aiken, A.: Winnowing: local algorithms for document fingerprinting. pp. 76–85. ACM (2003)
13. Verco, K.L., Wise, M.J.: Software for detecting suspected plagiarism: Comparing structure and attribute-counting systems. ACSE 96 (1996)
14. Whale, G.: Plague: plagiarism detection using program structure. School of Electrical Engineering and Computer Science, University of New South Wales (1988)
15. Wile, D.S.: Abstract syntax from concrete syntax. In: Proceedings of the 19th international conference on Software engineering. pp. 472–480. ACM (1997)