Frontiers of Information Technology & Electronic Engineering
www.jzus.zju.edu.cn; engineering.cae.cn; www.springerlink.com
ISSN (print); ISSN (online)
E-mail: jzus@zju.edu.cn

Fast code recommendation via approximate sub-tree matching*

Yichao SHAO[†1,2,3], Zhiqiu HUANG[†‡1,2,3], Weiwei LI[1,2,3], Yaoshen YU[1,2,3]

[1]*School of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 211100, China*

[2]*Key Laboratory of Safety-Critical Software, Ministry of Industry and Information Technology, Nanjing 211100, China*

[3]*Collaborative Innovation Center of Novel Software Technology and Industrialization, Nanjing 210016, China*

[†]E-mail: zqhuang@nuaa.edu.cn

Received Aug. 7, 2021; Revision accepted Mar. 24, 2022; Crosschecked

Abstract: Software developers often write code that has similar functionality to existing code segments. A code recommendation tool that helps developers reuse these code fragments can significantly improve their efficiency. Several methods have been proposed in recent years. Some use sequence matching algorithms to find the related recommendations. Most of these methods are time-consuming and can leverage only low-level textual information from code. Others extract features from code and obtain similarity using numerical feature vectors. However, the similarity of feature vectors is often not equivalent to the original code's similarity. Structural information is lost during the process of transforming abstract syntax trees into vectors. We propose an approximate sub-tree matching-based method to solve this problem. Unlike existing tree-based approaches that match feature vectors, it retains the tree structure of the query code in the matching process to find code fragments that best match the current query. It uses a fast approximation subtree matching algorithm by transforming the subtree matching problem into the match between the tree and the list. In this way, the structural information can be used for code recommendation tasks that have high time requirements. We have constructed several real-world code databases to evaluate the effectiveness of our method, which covers different languages and granularities. The results show that our method outperforms two compared methods in terms of recall value on all the datasets, and can be applied to big datasets.

Key words: Code reuse; Code recommendation; Tree similarity; Structure information
https://doi.org/10.1631/FITEE.2100379 **CLC number:**

1 Introduction

In software development, developers tend to reuse existing code which can achieve the desired functionality or behavior to assist their development procedure. Research shows that, on average, software developers spend about 19% of their development time on web searches, looking mainly for code examples for their tasks (Rahman et al., 2018). Therefore, an automatic code snippet recommendation tool could help developers greatly improve development efficiency.

However, searching instructive code based on a user's programming context is not always easy. Most of the current code recommendation tools are based on textual matching. They represent the context code fragment as tokens or lines of code, calculating the code similarity through sequence matching or a bag-of-words model, and return the most relevant code snippets (Ye, 2002; Holmes and Murphy, 2005; Antunes et al., 2014; Rahman and Roy, 2014). However, these methods fail to capture high-level structural information, which means it would be hard to obtain the best result when there is no highly similar match in the code repository.

Abstract syntax tree (*AST*) carries the structural information of code, but a traditional tree matching process consumes too much time for code recommendation tasks. To solve this problem, some re-

‡ Corresponding author

* Project supported by the National Natural Science Foundation of China (No. 61772270)

ORCID: Yichao SHAO, https://orcid.org/0000-0002-4553-5602

© Zhejiang University Press 2022

searchers propose to extract features from the *AST* of the code and use these, rather than trees, for matching (Jiang et al., 2007; Luan et al., 2018; Uraík et al., 2020). However, because the ideal candidate code length should be larger than the query code, these candidate codes often contain more sub-tree structures and feature patterns. That is to say, although two code snippets may obtain a high similarity score, they may not be related code because the distribution of these features in candidate code can be different from those in query code. Therefore, methods using such techniques may bring more false-positive results.

In this paper, we propose a novel code recommendation method based on *AST* sub-tree matching. Compared with methods that transform all *AST* into multiple features, we retain the tree structure of the query code in the matching process. A candidate code with a complete query code structure will gain a higher similarity score than those that share the same feature sets but have different overall structures, and thus will have a higher chance of being recommended. We use hash values to record every sub-tree in the *AST* and speed up the matching process by comparing the hash value only of sub-trees with the same number of nodes. Our method includes three stages: data processing, coarse-grained searching, and fine-grained re-ranking. First, we construct the code database using *AST*. In this stage, we calculate the hash value and the number of nodes for each sub-tree in *AST*, and store them using a list structure. This helps to reduce the costs of time and space in the matching process. Second, we traverse the *AST* of a given query code snippet to calculate the similarity between the query code and the codebase's candidate code. In this stage, the similarity measurement of two code snippets is the number of nodes in all the most similar sub-trees of their ASTs. Based on the similarity score, a *Top-K* candidate set will be generated. Third, we obtain the *AST* preorder traversal sequence for both the query code and each *Top-K* candidate code. We calculate their similarity using the *SW* algorithm to fully mine the sequence and continuity information in the code and re-rank the *Top-K* list. In this way, we can eventually obtain the recommending result list.

We have carried out an extensive empirical evaluation of our method applied to several large datasets, covering different languages and granularities. We compared our method with two strong baseline studies, and the experimental results showed that our method outperforms two compared methods on all the datasets.

This paper makes the following main contributions:

1. We introduce a new code recommendation algorithm based on sub-tree hashing and the *SW* algorithm. It takes programming context as input and recommends relevant code snippets to assist developers in software development.

2. We implement a code recommendation tool for Java and C. Experimental results show it has good performance in terms of both time consumption and accuracy for different recommending tasks.

The rest of this paper is structured as follows: Section 2 reviews some work related to our study. We present our method in detail in Section 3. Section 4 describes our experiments. The limitations of this study are discussed in Section 5. Finally, we conclude this paper in Section 6.

2 Related work

In this section, we describe some related studies, split into two categories: (1) tree similarity detection; (2) code recommendation

Tree Similarity Detection: Since tree-like data structure has gained more popularity in recent years, many studies have proposed focusing on similarity detection on trees. However, since most existing methods cannot be extended to large-scale datasets, how to achieve fast tree similarity detection is still an open question. Some researchers use tree edit distances to measure the similarity (Zhang and Shasha, 1989; Shasha et al., 1994; Chen and Zhang, 2014). Such edit-distance-based methods are widely used, but they obtain the similarity through the difference between two trees. This is inappropriate for tasks in which one tree contains another, which happens frequently in code recommendation. Other researchers have propose extracting features from trees and transforming the tree similarity detection problem into feature matching on numerical vectors, which is fast and easy (Jiang, et al., 2007; Luan, et al., 2018; Uraík, et al., 2020). However, the similarity of feature sets is not equivalent to the similarity of trees. These

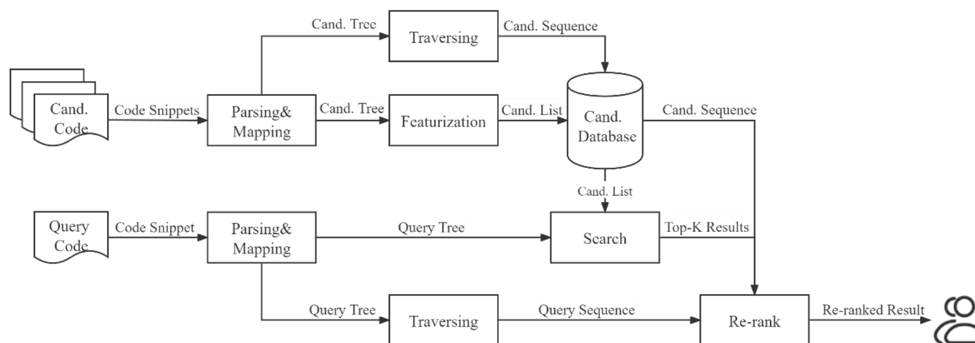

Fig. 1 The framework of code recommendation

methods abandon the original structure of the trees and the continuity information of nodes, and may return false high similarity results.

Code Recommendation: Code snippet recommendation is a hot issue that attracts many researchers. Most researchers focus on improving the precision and speed of recommending to improve the users' experience. Techniques like information retrieval (Sahavechaphan and Claypool ; Jiang et al., 2017) and pattern matching (Jiang, et al., 2007; Uraík, et al., 2020) are widely used in their work. Most existing methods are based on textual information. They treat code snippets as a set of tokens or code lines (Ye, 2002; Holmes and Murphy, 2005; Antunes, et al., 2014; Rahman and Roy, 2014; Ai et al., 2019). Some researchers parse the code snippets into *AST* and use tree-based similarity detection techniques to obtain recommendation results (Luan, et al., 2018; Uraík, et al., 2020). The advantage of such methods is that they exploit the syntax information of the code, but they often consume more time.

3 Method

Fig. 1 illustrates the overall architecture of our framework.To use the codebase efficiently, we first calculate and store the hash value and node number of each sub-tree in their *ASTs*. Then, we traverse the *AST* of the given query code to calculate the similarity between the query code and all the candidate code in the codebase. Multiple candidate code segments with the highest similarity will be selected. Finally, we calculate the *AST* preorder sequence similarity between the query code and the top-K candidate code using the *SW* algorithm. We re-rank the candidate code list based on the combination of two kinds of similarity and return the final recommendation result.

Next, we describe the details of each step using the code fragment in Listing 1 as a running example.

Listing 1 A simple piece of code used as the running example through Section 3.

```
for (int i = 0; i < 10; i++)
    a += 0.5;
```

3.1 Data processing

In this part, we show the procedure for data processing, i.e., how we transform candidate source code files into the form needed for the recommendation.

3.1.1 Tree parsing

Since our method is based on *AST*, the first thing we need to do is parse the code. We parse Java code with Javalang[1], and parse C code using Clang[2]. Our method can work either at file-level or method-level. For individual Java methods, we added a foo class so that the parser can process them, while for file-level code, we excluded package declarations and import statements, since they are often generated by the IDE and not relevant for recommending purposes

3.1.2 Featurization

We first map each node in the *AST* to a hash value. Fig. 2 visualizes the simplified parse tree and the corresponding hash value for each node of the

[1] https://github.com/c2nes/javalang
[2] https://clang.llvm.org/

code snippet in Listing 1. In principle, different *AST* nodes should have different values. However, to improve the detection effect, we map similar nodes to the same hash value to avoid the reduction of similarity caused by minor differences among similar nodes (Yang et al., 2018). For example, in Fig. 2, both *FLOATING_LITERAL* and *INTEGER_LITERAL* belong to numerical types, so they are given the same hash value of 17. We classify nodes according to their functions. The specific mapping rules may change according to different languages or parsers. In our experiment, we used *Javalang* as the parser of Java, and have recorded the classification details in the Appendix. Nodes under the same classification will be mapped to the same hash value. In a specific implementation, it can be supplemented or adjusted according to the required recommending effect.

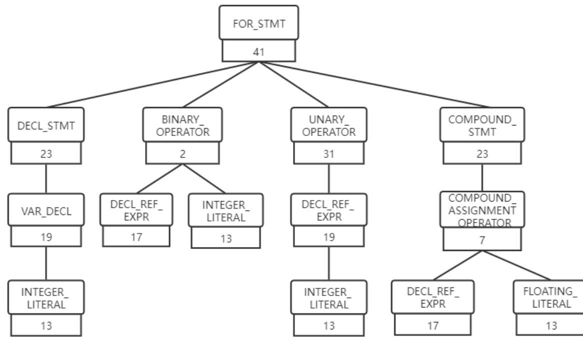

Fig. 2 AST and the hash value of each node for code in Listing 1

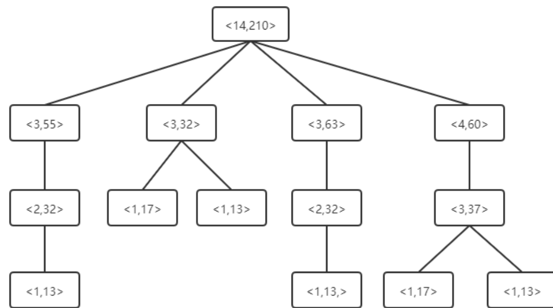

Fig. 3 Hash tree for code in Listing 1, where each node records the sub-tree information rooted at the current node

Then we calculate the hash value of each sub-tree in the *AST*. As Fig. 3 shows, we use a <node number, hash value> tuple to represent the information of the sub-tree rooted at the current node. We record the node number of each sub-tree and simply define the hash value of a tree as the sum of the hash values of each node in this tree. For the snippet i < 10 in Listing 1, its hash value would be *Hash(BINARY_OPERATOR)* + *Hash(DECL_REF)* + *Hash(INTEGER_LITERAL)* = *2* + *17* + *13* = *32*, and obviously its node number is 3.

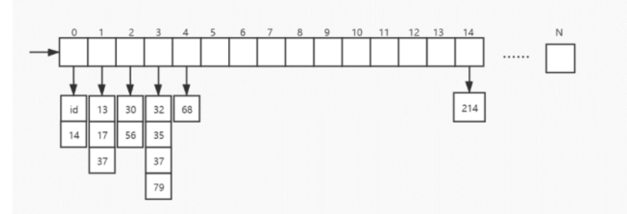

Fig. 4 Intermediate representation of candidate code in Listing 1

To speed up the subsequent matching and save memory space, we discard the tree structure and store sub-tree information in the form of lists. The intermediate representation of the code fragment in List 1 is shown in Fig. 4. The first dimension of the list represents the number of nodes from 1 to N, and the hash values of sub-trees with corresponding node numbers are joined into one list. Each hash value list is kept to speed up the search process. We also record the index of code and the total number of nodes in the tree at the start of the list.

3.2 Searching

In this part, we calculate the similarity scores between the query code and each candidate code and obtain a candidate code set containing K code snippets with the highest similarities.

Like candidate code processing, we map the AST of a given query code to a hash tree, similar to Fig. 3. We first filter out the candidate codes that cannot reach the size threshold according to the number of nodes. We traverse the query hash tree in preorder and search for the hash value of the current node in the candidate hash value list. Since sub-trees with different node numbers represent different structures, only pairs of sub-trees with the same node number need to be compared. For each sub-tree node in the query hash tree, we obtain the hash value list with the same node number and search for the target hash value using a binary search. If found, the sub-tree represented by the current node is regarded as a match,

and all its child nodes will be skipped in the subsequent traversal. We take the total number of nodes under all common sub-trees as the similarity score and record K code fragments with the highest score. Algorithm 1 shows the process of searching, which we will introduce in three parts: preliminary filtering, sub-tree validation, and approximate matching.

Algorithm 1 The Searching Algorithm

Input: Query tree, *Q*; Lists of candidate code, *L*; Size of target candidate set, *K*
Output: Top-*K* result list, *T*;

```
1  V←∅
2  for each i in L
3    if sizeMatched(Q, i)  // Filter based on node number.
4      V←V ∪ { <i, traverseTree(Q, i)> }
5    end if
6  end for
7  T ← Rank candidate set V and get Top-K result
   // Traverse query tree recursively
8  function traverseTree (N: sub-tree node, C: list)
9    totalNum ← node number of query tree
10   sub-treeNum ← sub node number of current tree N
11   matchedNum ← 0
12   if isSearched(N, C[sub-treeNum])
       // Validate the sub-tree.
13     if validated(totalNum, sub-treeNum, N, C)
14       return N
15     end if
16   end if
     // If not found, traverse all the sub-trees.
17   for each j in sub-tree of N
18     matchedNum ← matchedNum + traverseTree(j, C)
19   end for
     // Approximate matching for the current node.
20   if exceedThreshold(matchedNum, sub-treeNum)
21     matchedNum ← matchedNum + 1
22   end if
23   return matchedNum
24 end
```

3.2.1 Preliminary filter based on node number

When applying code recommendations to real tasks, the vast majority of code snippets in code databases are irrelevant to the query code. However, we need to process all possible pairs of methods to find out potential results. This can be extremely costly, especially on very large datasets. For candidate code fragments whose number of nodes is less than the query code, their maximum similarity score is the ratio of the number of common nodes to the total number of query nodes. We use size-based heuristics to aggressively eliminate unlikely candidate code snippets upfront. The intuition is that two methods with considerably different sizes are very unlikely to implement the same, or even similar, functionality. (Saini *et al*., 2018) In addition, useful code recommendations are usually larger than query code so that developers can obtain a reference from the extra part. So for each candidate code, we first judge whether it satisfies formula (1):

$$|candidate_n| > \lambda * |query| \qquad (1)$$

where $|candidate_n|$ represents the node number of the current candidate tree, $|query|$ represents the node number of the query tree, and λ is an adjustment factor.

If formula (1) is satisfied, subsequent matching will be carried out. Otherwise, the candidate code will be omitted. This heuristic can lead to some false negatives, especially for code fragments with the same function but which differ in texture or structure. (Saini *et al*., 2018). However, in our experiments described below in 4.2.3, we observed little or no impact of this on the recall value of the final result under appropriate parameter settings.

3.2.2 Sub-tree validation

Since we use only one number to represent the hash value, sub-trees containing different nodes may be mapped to the same hash value, which we call a hash collision. Generally, with the increase of the node number of sub-trees, hash collision is more likely to occur, and when the proportion of sub-trees in the query tree increases, their influence on the final result is more significant. For these sub-trees, additional verification is performed. We adopt a simple but effective method, that is, to verify whether the subsequent nodes of this sub-tree root can also be found in the candidate sub-trees. If they all can be found, the two sub-trees are considered to be matched. We use the following formula to decide the number of nodes to be verified:

$$T(c,q) = \left[\frac{5c^2}{(c+10) * q} \right] \qquad (2)$$

where $[t]$ denotes the largest integer no more than t, c denotes the node number of the current sub-tree, and q

denotes the node number of the whole query tree.

We found that almost all false-positive recommending results caused by a hash collision can be eliminated at a low extra cost using this verification algorithm. The corresponding experimental result lists are in the Appendix.

3.2.3 Approximate sub-tree matching

We can obtain the total number of sub-tree nodes repeated between the query code and each candidate code through the matching process. However, exact sub-tree matching cannot fully reflect the structural similarity. Here, we use the query tree in Fig. 5 as an example. For illustration, we marked the node id and omitted the node number and hash value in this figure.

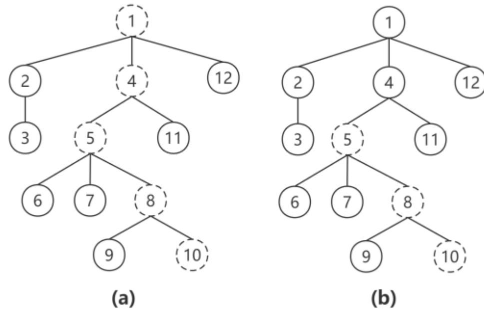

Fig. 5 A query tree example in a match without (a) and with (b) approximate sub-tree matching. The solid circle represents a matched node and the dotted circle represents an unmatched node.

Consider a candidate code fragment whose *AST* is almost the same as the query code, the only difference being that node 10 is either missed or replaced by a node with a different hash value. In this instance, all sub-trees containing node 10 obtain different hash values, which means all sub-tree nodes on the path from node 10 to the root of the query tree cannot be successfully matched (Fig. 5a). Hence, the similarity for this tree is only 7/12 = 58.3%, though there is only a different node between the query code and the candidate code. Such minor differences in *AST* are common in software development (Baxter et al., 1998), and it is reasonable to think that it will be more evident for unfinished code.

So rather than searching for identical sub-trees, we prefer to find similar ones. We set a similarity threshold T to judge whether a sub-tree is matched. For each none-leaf node in the query tree, if no exact match is found in the candidate hash list, additional judgments will be made based on the sub-tree rooted at the current node. That is, if the proportion of matched nodes in the whole sub-tree exceeds T, the current node will be deemed to be matched. For example, as Fig. 5b shows, if we set T as 80%, among all nodes on the path from root node 1 to node 10, nodes 1 and 4 are matched while nodes 5, 8, and 10 are unmatched, and the similarity score should be 9/12 = 75%, which is a better reflection of code similarity.

3.3 Re-ranking

According to our above algorithm, when the number of common leaf nodes is fixed, code fragments with a similar structure to query code can obtain a higher similarity score. In some cases, although some code fragments are quite different from the structure of the query code, they may score higher for similarity than related code because they contain more common sub-trees. Such a situation can easily occur when there is no near-exact matching for the current query in the codebase. In addition, several candidate codes may share the same similarity score, in which case we cannot determine the order of these methods in the recommendation list. To solve these problems and reduce false positive results, we introduce a re-ranking method based on the *SW* algorithm.

The SW algorithm is used for sequence alignment, which means finding a similar region between two sequences. The purpose of the algorithm is not to compare the whole sequence, but to find similar fragments between the two sequences and determine their similarity.

For query code and each candidate code, we traverse their hash tree in a depth-first order to generate node sequences. Then we use the SW algorithm to obtain the similarity scores between them. The details of *SW* are presented as follows (Smith and Michael, 1981):

Assuming that $A = a_1a_2...a_n$, and $B = b_1b_2...b_m$ are node sequences to be aligned. A similarity is given between sequence elements a_i and b_j.

$$s(a_i, b_j) = \begin{cases} match & a_i = b_j \\ mismatch & a_i \neq b_j \end{cases} \quad (3)$$

We set up a matrix H whose size is $(n+1) \times (m+1)$. Its first row and first column are initialized with 0.

Based on the initial value of scoring matrix H, the scores of other elements are calculated using the following formula:

$$H_{i,j}(2 \le i, 2 \le j) = max\begin{cases} H_{i-1,j-1} + s(a_i, b_j) \\ H_{i-1,j} + gap \\ H_{i,j-1} + gap \\ 0 \end{cases} \quad (4)$$

The highest value of H is returned as the similarity score between two sequences.

We combine the *SW* sequence similarity with the tree-list similarity obtained in the filtering phase as the basis for the re-ranking process, rather than using a single similarity score. This is because the *SW* algorithm cannot deal with different code statement orders, while tree-list similarity covers such cases and can supplement the final result. The final similarity score is calculated using formula (5).

$$Score(query, candidate_n) = \delta \cdot TLSim(query, candidate_n) + SWSim(qeury, candidate_n) \quad (5)$$

where *TLSim* denotes the tree-list similarity from the searching phase, and *SWSim* denotes the *SW* sequence similarity from the re-ranking phase. δ is an adjustment factor.

4 Experiments

In this section, we describe our evaluation of the effectiveness of the method proposed in this paper. Our experiments were conducted on a 2.60-GHz CPU (Intel i5) PC running Windows 10 OS with 8 G memory.

4.1 Experimental design

4.1.1 Datasets

We used three real-world datasets to evaluate the performance of our method on code recommendation: a Java code repository IJaDataset (Svajlenko and Roy, 2019) and two C code datasets, the OJSystem (Mou et al., 2014) and OJNUAA, from pedagogical programming open judge (OJ) systems.

The IJaDataset-2.0 is a large Java repository from the SECold Project[3], containing 25,000 open-source Java projects. We extracted method-level code fragments and kept those with executable lines between 20 and 30. Then we randomly sampled 300 code fragments and manually created partial code snippets by deleting 30-50% of code lines from the original code snippets, aiming to check whether our method could produce appropriate recommendations when an exact match exists in the code database. Three developers with three years of Java project development experience were asked to judge the returned results. If two or more developers thought that the result corresponded to the query code, then we took and recorded it.

The OJSystem contains 104 programming problems together with different source codes students submitted for each problem. There are 500 files submitted under each problem. Two different source codes under programming problems can be regarded as similar because they realize the same functionality. We randomly selected 20 code files as queries under each programming problem and removed them from the candidate set to see if our method could recommend other files under the same problem according to these queries.

We constructed OJNUAA similar to the OJSystem. It contains 99 programming problems, each with more than 200 files submitted from a school programming online judgment system. OJNUAA covers a wide range of programming problems from simple to complex, making the average length of code files shorter than those in the OJSystem, and the submitted codes for fundamental problems are mostly similar to each other. We also randomly collected 20 code files as queries for each problem on this dataset.

The statistical information of the above datasets[4] is shown in Table 1.

Table. 1 Dataset size information.

Project Name	File/method numbers	Lines of code
IJadataset	718,525	18,237,218
OJSystem	52,000	1,885,262
OJNUAA	34,797	902,133

[3] http://www.secold.org/projects/seclone
[4] https://github.com/melond/RecommendationSamples

Table 2 The experimental results from different methods over all datasets

		SENSORY	Aroma	Our method
IJaDataset2.0	Recall@1	100%	99%	99.3%
	Recall@10	100%	100%	100%
	MRR	1	0.995	0.996
OJSystem	Recall@1	67.3%	76.2%	81.1%
	Recall@10	78.6%	86.7%	92.1%
	MRR	0.723	0.821	0.847
OJNUAA	Recall@1	81.5%	91.8%	94.5%
	Recall@10	90.3%	95.5%	97.3%
	MRR	0.909	0.946	0.967

4.1.2 Metrics

We chose *Recall@K* and Mean Reciprocal Rank (*MRR*) as measurement metrics to evaluate the performance of our proposed method.

Recall@K is defined as the percentage of query code snippets for which the original method body is found in the *top-K* methods in the re-ranked search result list. The *Recall@K* was calculated as:

$$Recall@K = \frac{|Relevence@K|}{|Queries|} \quad (6)$$

where the numerator */Relevance@K /* denotes the number of query code snippets for which the original method body is found in the *Top-K* methods in the result list. The denominator */Queries/* denotes the number of all queries.

MRR is defined as the average of the reciprocal ranks for all queries, where the reciprocal rank of a query is the inverse of the rank of the first relevant result. A larger *MRR* value means a higher ranking for the first relevant methods. The *MRR* is calculated as:

$$MRR = Q^{-1} \sum_{q=1}^{Q} FRank_q^{-1} \quad (7)$$

where Q denotes the total number of query codes, and $FRank_q$ denotes the rank of the first relevant result for query q.

4.2 Results

We investigated two research questions (RQs) to validate the effectiveness of our method.

RQ1. How effective is our method in recommending code snippets?

In this study, we attempted to propose a new code recommendation tool. We introduced an approximate sub-tree matching algorithm to search for candidate code sets and used the *SW* algorithm to re-rank the list. In this RQ, to verify our method's effectiveness, we compared our experimental results with those of two strong baseline works: SENSORY and Aroma.

SENSORY(Ai, et al., 2019) is a practical code recommendation tool. It uses the granularity of code statements rather than tokens as input, using the BWT algorithm to perform an ordered subsequence search. Since SENSORY is a Java method level tool while OJ databases are constructed by C files, we used clang to parse the *AST* so SENSORY could process the C code.

Aroma (Luan *et al*. 2018) is a state-of-the-art code recommendation tool. It vectorizes the AST features and uses a bag-of-words-like strategy to calculate the similarity between the query code and code in the database. It then re-ranks the result list and uses an intersection phase to merge different code snippets in the result list. Since the intersection phase will change the original code snippets from datasets, a manual judgement may be required to assess the correctness of the final result. It is hard to apply to big-scale automated validation and may introduce manual biases. Therefore, we did not use the final intersection stage in the experiment, but obtained the experimental results according to the previous part.

Table 2 shows the experimental results of these three methods applied to different datasets. The re-

sults show that our method outperformed the compared methods in most situations.

In the experiment on IJaDataset, we treated the results corresponding to the query as a hit, and all three methods achieved good performance. This shows that these methods can make relevant recommendations when there is an exact match in the code dataset in most cases. On OJSystem and OJNUAA, our method performed the best, and SENSORY performed poorly. This is because SENSORY uses a strict code line matching strategy. Its matching process is based on textual information, while the OJ datasets are classified according to functionality. Therefore, it is difficult for SENSORY to recommend the correct result when no exact textually similar candidate code is in the codebase. Aroma chooses to convert the AST into the form of feature vectors and uses feature vectors to calculate the similarity. However, this bag-of-words-like method ignores the distribution of features in the original abstract syntax tree. Many irrelevant codes will appear with high similarity because they contain similar features to the query code, even though they may not be in the same distribution. Such a phenomenon is more obvious for large code fragments in the codebase and may cause the actual more relevant code to be excluded, thus affecting the final result. In contrast, our method considers the code structure information and stores the candidate codes in the form of sub-tree hash values so that the candidate codes with a more complete sub-tree structure will have higher similarity. We believe these are the reasons why our method performed the best on these datasets.

Table 3 Time costs of three methods applied to the IJaDataset

Approach	Average Recommending Time (seconds)
SENSORY	232.3
Aroma	0.4
Our Method	2.1

Recommending speed is also important for code recommendation tools. The average time complexity of our method for data processing and matching are denoted O(n) and $O(m*f)$ respectively, where n denotes the total AST node number of all candidate code, m denotes the node number of the query tree, and f represents the number of candidate code snippets after size-based filtering. Table 3 shows the recommending time cost of three methods applied to the IJaDataset, which contains 718,525 files and 18,237,218 code lines. SENSORY is based on statement sequence matching, so it is time-consuming and not scalable to large datasets. Since Aroma computes similarity based on numerical vectors, it had the best time performance. The time cost of our method was higher than that of Aroma, but still acceptable on such a big dataset.

Our method requires additional storage space to store the hash representation of the AST nodes for code snippets in the code database. Similarly, taking the IJaDataset as an example, its source file occupies about 0.63 GB, while the intermediate representation in our method occupies 2.47 GB, which is about 4 times that of the code source file. The growth of additional space is linearly related to the size of the code database, and it is acceptable for today's data centers.

In summary, our code recommendation method worked well at the method level and file level and outperformed the baseline methods over all datasets. Also, our method had a good time performance and can be applied to big datasets. Based on the experimental results, we believe that our method can help developers quickly find the code they need.

RQ2. How do the parameters affect the experimental results?

To explore the effects of various parameters on the experimental method, we performed a parameter sensitivity experiment. We tested the two parameters that have the greatest impact on the method: λ in formula (1) and δ in formula (3). The parameters *match*, *mismatch,* and *gap*, required when using the *SW* algorithm, do not have a great influence on the experimental results (Yang et al, 2018). Therefore, we set them to the empirical values 2, -2, and -1, respectively (Kamalpriya and Singh, 2017).

To answer RQ2, because recommendations should be made repeatedly under each parameter setting in our experiment, and the experiment using the IJaDataset needs much manual verification, we chose only the OJSystem and OJNUAA for automated validation.

Our method uses formula (1) for the size similarity filter. A larger adjustment factor λ can filter out

more candidate codes and speed up the overall recommendation, but may also lead to more false negatives. We predefined the range of λ from 0 to 1.4 and used Recall@1 as the metric to estimate the recommending effect.

The experimental results are shown in Fig. 6. The recall rate was close when λ was between 0 and 0.8, and a significant decrease occurred when λ was greater than 0.8. We conclude that the size-based filtering had little or no impact on the final recommendation results when λ was lower than 0.8. At the same time, to reduce unnecessary matching as much as possible and speed up the recommendation, we decided to set λ to 0.8.

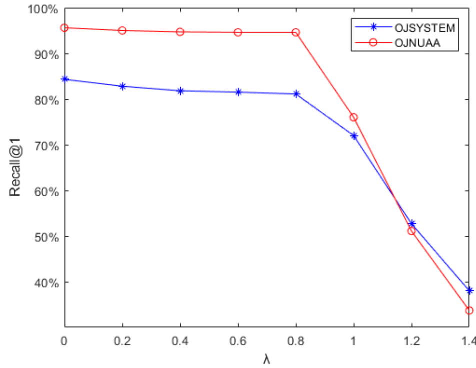

Fig. 6 Results of our method with different λ values over two datasets

Our method uses formula (3) to combine both the tree-list similarity and sequence similarity into one final score. We balance the influence of two kinds of similarity by parameter δ, e.g., the lower the value of δ, the greater the influence of sequence similarity on the final result. We re-ranked the first 50 code segments after the coarse-grained search due to the high time consumption of the *SW* algorithm. We predefined the range of δ from 0.1 to 100, and used Recall@1 as the metric to estimate the recommending effect.

The experimental results over two datasets using Recall@1 are shown in Fig. 7. The recall value using OJNUAA was better than that using OJSystem. This is because OJNUAA covers a broader range and includes many fundamental programming problems. The answers to these basic problems are similar, so the recommendation will be much easier. In addition, the boundaries of some programming problems in the OJSystem are not clear, and there are several problems with similar or almost identical functionality, which will also have a specific impact on the results.

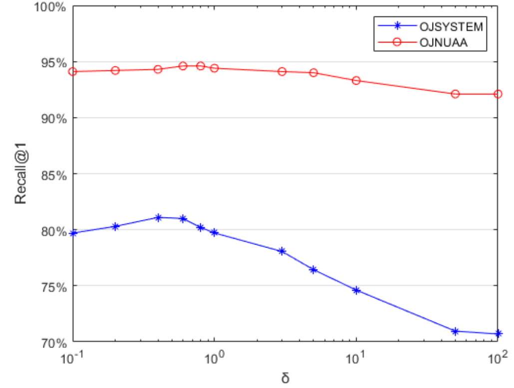

Fig. 7 Results of our method with different δ values over two datasets

Using the OJSystem dataset, our method achieved the best recall results when δ was 0.6, and as δ continued to grow, the recall rate finally dropped to around 0.7 (Fig. 7). Using OJNUAA, the recall reached the maximum value when δ was set to 0.8. The reason for this difference is that since the average lines of code for each file in the OJSystem are larger than those in OJNUAA, sequence similarity plays a more critical role in the final result. The best selection of δ was different for different datasets, but when the value of δ was between 0.6 and 0.8, our method achieved good results over both datasets. Also, we conclude that the final recommendation results can be significantly improved after the re-ranking phase.

In summary, the best selection of δ may change according to the dataset, but our method can achieve good results for both datasets when δ is between 0.6 and 0.8.

5 Limitations

In this section, we will discuss some limitations associated with our work.

The query set: Our method is not being directly implemented by developers in the actual development process because such an experimental setting needs a lot of manual validation and may introduce error caused by individual biases. Instead, we performed

large-scale automated evaluations to test the accuracy of our method. We constructed three real-world code databases and carried out two different types of experiments. On the IJaDataset, we tested whether the methods could recommend matching code according to partial code, while for OJSYSTEM and OJNUAA, we evaluated the effectiveness of recommending systems when there may be not a near-exact match in the codebase. Although these two kinds of experimental settings cannot fully cover real development scenarios, they can reflect the performance of the code recommendation system in typical cases.

Parameter setting: During the experiment, we set several parameters based on experience or existing experimental results, such as the threshold T in approximate sub-tree matching and score and penalty parameters in the SW algorithm. We did not conduct complete experiments to analyze the effect of each parameter on the final result, which may have affected the performance of our method to some extent. But since the experimental results under the current parameter settings were satisfactory, the parameters in the experimental process should be reasonable. We will fully investigate the effects of each parameter and their combinations in our future work.

6 Conclusions and future work

In this paper, we have presented a search-based code recommendation technique that combines an approximate sub-tree matching algorithm and a sequence matching algorithm. More specifically, for each candidate code, we first calculate and store the hash value and node number of each sub-tree in their *AST*. Then, we traverse the *AST* of the given query code to calculate the similarity between the query code and all the candidate code in the codebase. *Top-K* candidate code segments are selected according to a similarity score. We also calculate the *AST* preorder sequence similarity between the query code and the *Top-K* candidate code using the *SW* algorithm. Finally, we re-rank the candidate code list based on the two kinds of similarity and return the final recommendation result. To evaluate the effectiveness of our method, we conducted experiments on several real-world code databases. Experimental results showed that our method can recommend code with high recall and significantly outperform compared methods.

In the future, we will consider introducing a clustering process, grouping functionally similar code fragments together and accelerating the searching speed. We will also consider optimizing the storage structure of the intermediate representation to further reduce the additional storage space required.

Contributors

Yichao SHAO and Zhiqiu HUANG designed the algorithm, implemented the tool, and drafted the manuscript. Weiwei LI and Yaosheng YU helped organize the manuscript. Zhiqiu HUANG and Yaosheng YU revised and finalized the paper.

Compliance with ethics guidelines

Yichao SHAO, Zhiqiu HUANG, Weiwei LI, and Yaosheng YU declare that they have no conflict of interest.

References

Ai, L., Z. Huang, W. Li., *et al.*, 2019. SENSORY: Leveraging Code Statement Sequence Information for Code Snippets Recommendation. 2019 IEEE 43rd Annual Computer Software and Applications Conference, p.27-36. https://doi.org/10.1109/COMPSAC.2019.00014

Antunes, B., Furtado, B., Gomes, P., 2014. Context-Based Search, Recommendation and Browsing in Software Development, Springer New York, P.45-62. https://doi.org/10.1007/978-1-4939-1887-4_4

Baxter, I. D., Yahin, A., Moura, L., *et al.*, 1998. Clone detection using abstract syntax trees. Software Maintenance, Proceedings. International Conference on. IEEE, p.368-377 https://doi.org/10.1109/ICSM.1998.738528

Chen, SH, Zhang, *et al.,* 2014. An improved algorithm for tree edit distance with applications for RNA secondary structure comparison. Springer-Verlag New York, p.778-797. https://doi.org/10.1007/s10878-012-9552-1

Jiang, LX, Misherghi, *et al*., 2007. DECKARD: Scalable and accurate tree-based detection of code clones. *PROC INT CONF SOFTW ENG*, p.96-105. https://doi.org/10.1109/ICSE.2007.30

Jiang, H., Nie L., Sun Z., *et al.*, 2017. ROSF: Leveraging Information Retrieval and Supervised Learning for Recommending Code Snippets, p.34-46. https://doi.org/10.1109/TSC.2016.2592909

Luan, S., Yang, D., Barnaby, C., *et al.*, 2018. Aroma: Code Recommendation via Structural Code Search. 3(OOPSLA), 1-28. https://doi.org/10.1145/3361527

Mou, L., Li, G., Zhang, L., *et al.,* 2014. Convolutional Neural Networks over Tree Structures for Programming Language Processing. Thirtieth AAAI Conference on Artificial Intelligence. https://arxiv.org/abs/1409.5718

Rahman, M. M., Roy, C. K., 2014. On the Use of Context in Recommending Exception Handling Code Examples. IEEE 14th International Working Conference on Source Code Analysis and Manipulation, p.285-294. https://doi.org/10.1109/SCAM.2014.15

Rahman, M. M., Roy, C. K., David, L. O., 2016. RACK: Automatic API Recommendation using Crowdsourced Knowledge. IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, p.349-359 https://doi.org/10.1109/SANER.2016.80

N Sahavechaphan , K Claypool, 2006. XSnippet: mining For sample code. Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, p.413-430 https://doi.org/10.1145/1167473.1167508

SHASHA, WANG, JTL, *et al.,* 1994. Exact and approximate algorithms for unordered tree matching. *IEEE TRANS SYST MAN CYBERN*, p.668-678. https://doi.org/10.1109/21.286387

Svajlenko, J., Roy, C., 2019. The Mutation and Injection Framework: Evaluating Clone Detection Tools with Mutation Analysis. IEEE Transactions on Software Engineering, p1060-1087. https://doi.org/10.1109/TSE.2019.2912962

Yang, Y., Z. Ren, C. Xin , *et al.*, 2018. Structural Function Based Code Clone Detection Using a New Hybrid Technique. IEEE 42nd Annual Computer Software and Applications Conference. p.286-291. https://doi.org/10.1109/COMPSAC.2018.00045

Saini, V., Farmahinifarahani, F., Lu, Y., 2018. Oreo: Detection of clones in the twilight zone. Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, p.354-365 https://doi.org/10.1145/3236024.3236026

Smith, T. F., Waterman, M. S., 1981. Identification of common molecular subsequences. Journal of molecular biology, 147(1), p.195-197. https://doi.org/10.1016/0022-2836(81)90087-5

Kamalpriya, C. M., Singh, P., 2018. Enhancing program dependency graph based clone detection using approximate subgraph matching. IEEE 11th International Workshop on Software Clones, p.1-7. https://doi.org/10.1109/IWSC.2017.7880511

Appendix: Recommendation results before and after sub-tree validation

	Recall@1	False positives caused by hash collision
Without validation	75.8%	24.3%
With validation	81.1%	0.3%

Formula (2) is used to verify the sub-tree pairs that may produce a hash collision. We tested Recall@1 before and after using hash verification on the OJSystem dataset and collected the false-positive results that appeared in first place in the recommendation list. We analyzed the proportion of recommendation errors caused by hash collision. The experimental result is shown in table above.

The table shows that 24.3% of the false-positive results were caused by hash collisions without the verification phase. After implementing hash validation using formula (1), we observed very few cases caused by hash collision, thus improving the recommendation effect. We conclude that the hash verification process can largely eliminate the negative impact of hash collision.

Appendix: Categories of node types for javalang

Function	Number	Example
Class	10	TypeDeclaration ClassDeclaration InterfaceDeclaration
Type	4	Type BasicType ReferenceType
TypeParameter	1	TypeParameter
Pair and Array	3	Annotation

		ElementValuePair ElementArrayValue
MemberDeclaration	4	MethodDeclaration FieldDeclaration ConstructorDeclaration
VariableDeclaration	7	ConstantDeclaration VariableDeclaration LocalVariableDeclaration
ExceptionHandleClause	3	TryResource CatchClause CatchClauseParameter
BasicStatement	7	Statement BlockStatement TryStatement
BranchStatement	2	IfStatement SwitchStatement
LoopStatement	3	WhileStatement DoStatement ForStatement
LoopControlStatement	3	BreakStatement ContinueStatement ReturnStatement
LoopControl	3	SwitchStatementCase ForControl EnhancedForControl