A Two-stage Approach for Structurally-similar Cross-language Code-pair Detection

Feng Dai
The University of Tokyo
Tokyo, Japan
daifeng@csg.ci.i.u-tokyo.ac.jp

Shigeru Chiba
The University of Tokyo
Tokyo, Japan
chiba@csg.ci.i.u-tokyo.ac.jp

Abstract

A fast and reliable tool to detect structurally-similar cross-language code pairs is crucial when maintaining large projects across different programming languages. In this paper, we present a new tool[1] for detecting such code pairs by adopting a two-stage approach. Our approach first uses models trained on two-level generic ASTs to filter candidates, and uses tree-based editing-distance algorithm for accurate comparison. We create a new cross-language dataset, including Java-Python, Java-C, Python-C structurally-similar method body pairs, and evaluate our approach on this dataset. We manage to obtain a much faster speed and similar detection accuracy in detecting structurally-similar cross-language code pairs compared with the state-of-the-art technique.

CCS Concepts

• **Software and its engineering → Software maintenance tools**.

Keywords

cross-language code pairs, code representation, software engineering, similar code pairs

ACM Reference Format:
Feng Dai and Shigeru Chiba. 2025. A Two-stage Approach for Structurally-similar Cross-language Code-pair Detection. In *The 40th ACM/SIGAPP Symposium on Applied Computing (SAC'25), March 31-April 4, 2025, Catania, Italy*. ACM, New York, NY, USA, Article 4, 9 pages. https://doi.org/10.1145/3672608.3707807

1 Introduction

Similar code pairs are a non-negligible issue in software maintenance [10]. Researchers have studied detection of similar code pairs with the same programming language for decades. This task is also referred as code clone detection. Based on the degree of similarity, [23] classifies code pairs with the same language into four types. Among them, type-3 pairs are these with statement-level insertions, deletions and substitutions, and type-4 pairs are these with the same functionality while ignoring implementation details. Many approaches [12, 13, 15, 24, 25] have been developed for different types of similar code pairs.

Recently, detecting similar code pairs across different programming languages is attracting practitioners' interests. This is because large software projects are getting polyglot. For example, a web-service project may include a server side and multiple client sides. Programmers often use multiple languages for different platforms, such as Java for servers, JavaScript for browsers, Kotlin for Android and Swift for iOS. We are particularly interested in cross-language code pairs with similar control structures because they are important in practice. For example, in web-service development, a structurally-similar code pair on different client sides can be a candidate function on the server side. It is challenging to detect structurally-similar cross-language code pairs. This is because even if two code fragments share the same control structure, they are not identical at the token level. Existing works for traditional cross-language code clone detection have some limitations. The works in [5, 18, 19, 27, 28] are not suitable for structurally-similar cross-language code pair detection because they ignore the importance of control structures and only focus on the same functionality. The works in [1, 14] realize the importance of similar structures. However, they utilize .NET Code Document Object Model (CodeDOM) and thus are only useful in .NET language family.

As far as we know, the state-of-the-art technique for detecting a structually-similar cross-language code pair is proposed by [16]. This work develops a common representation across different programming languages called *generic* abstract syntax trees (ASTs), and adopts tree-based editing-distance algorithm to calculate similarity scores between different generic ASTs. It takes control structures into consideration by calculating the number of edits to make two generic ASTs identical. Although a transformer from an original AST to a generic AST must be developed for every source language in advance, developing such a transformer is not expensive since the transformation is simple node-by-node mapping. The accuracy of this technique is ideal, but a drawback of the technique is its detection speed. Since computing a tree-based editing distance is highly time-consuming, the technique is not applicable for a large code base.

In this paper, we develop a two-stage approach for detecting cross-language code pairs with similar control structures. Our approach improves the speed of the state-of-the-art technique, while keeps very close accuracy. In the first stage, our approach exploits neural-network models to pre-determine a few most-likely candidates. In the second stage, it calculates tree-based editing-distances deterministically to select the most similar code fragment in control structures. To promote the accuracy of the first stage, we develop a new code representation technique called two-level generic AST

[1]The source code is posted in https://github.com/Eduardo95/sccd.

SAC '25, March 31-April 4, 2025, Catania, Italy

ACM ISBN 979-8-4007-0629-5/25/03
https://doi.org/10.1145/3672608.3707807

representation for neural-network models. We discover that constructing *two* different kinds of generic ASTs and giving them to a neural-network model can significantly increase the possibility that the true pair is within the pre-determined candidates, and thus improve final accuracy.

Our contributions are two-fold:

- We improve the speed of detecting pairs of structurally-similar method bodies written in different programming languages by developing a two-stage approach. A unique feature of our approach is to use two different kinds of generic ASTs as inputs for neural network models. Our approach for detecting structurally-similar cross-language code pairs in Java, Python and C pairwise achieves comparable accuracy with the state-of-the-art technique, but a much faster speed.
- We create a code-pair dataset for evaluating a tool for detecting cross-language code pairs with similar control structures. The dataset includes similar Java-Python, Java-C and Python-C code pairs. To the best of our knowledge, we are the first to create a dataset specially targeting at structurally-similar code pairs.

The rest of the paper is organized as follows. Section 2 discusses the motivation of our work. Section 3 presents the overall structure and the details of our approach. In Section 4, we present the results of our experiments to compare our approach with other approaches. In Section 5, we provide a discussion about our experiments and the threats to validity of this paper. Finally, we present related works in Section 6 and conclude this paper in Section 7.

2 Structurally-similar cross-language code pairs

When software development is polyglot, cross-language code pairs with similar control structures are important causes of source code refactoring and software maintenance. A typical scenario is micro-service development where multiple languages are used for different modules. An important principle in micro-service development is single-responsibility principle. If a service breaks the single-responsibility principle, code pairs with similar control structures may appear. For example, a service which is responsible for managing the user posts in a bulletin board system (BBS) may implement some authorization logic, and another service responsible for managing user comments may also implement a similar authorization logic. This will result in code duplication. Although two services implement different functionality, the authorzation logic should be extracted and merged as an independent service.

Another scenario is modern web-based service development. If code fragments from different client sides implement the same logic, structurally-similar code duplication may also happen, and they should be reconsidered and grouped into a function on the server side. For example, in a web application, the grouping-posts feature was not part of the initial application, and has been added later on. The backend API only provides an HTTP endpoint with a GET method to return a list of ungrouped user posts, and the frontend clients need to group the posts by themselves, leading to duplicated code fragments. For maintenance efficiency and system stability, this pair of code fragments should be grouped into a common API in the backend server. In Fig. 1, two code snippets are written in different languages, and we can see a clear correspondence between their control structures: both the code fragments contain an assignment statement, a for statement and a return statement. Within the for loop body, the logic of processing the dict/HashMap is also similar. Although discussion between teams can help to mitigate code duplication to a certain extend, it remains a very tedious issue to track.

Our goal is similar to traditional cross-language clone detection, as both try to find a *similar* code pair. However, we are more focused on similar control structures while traditional cross-language clone detection mainly considers the same functionality and ignores detailed implementations. In real-world development, it is important to detect similar structures as they are good candidates for code refactoring. Finding a pair of code fragments with only the same functionality is also an important research topic, but it is out of the scope of this paper. If two functions have the same functionality but with different algorithms, such difference might be intended by the programmers, and might not be refactored or merged. Therefore, we are more interested in similar control structures.

Tools [1, 5, 14, 18, 19, 27, 28] for traditional cross-language code clone detection are not satisfying for various reasons. They are either not specialized for structurally-similar cross-language code pairs, or not applicable for large structurally-similar cross-language code pair detection. [1, 14] detect cross-language code clones in the .NET language family by translating a program in a .NET language into the .NET Code Document Object Model (CodeDOM). They can represent the logical structure of source code, but are limited in .NET language family. [28] analyzes an enriched Concrete Syntax Tree [21], which is an intermediate state used by [4], to detect cross-language code clones. This method gives some consideration to control structures, but still has some limitations. For example, this method requires two code fragments to have the same code length. [5] analyzes commit logs recorded in version control systems to detect clones, which does not consider control structure similarity. [18, 19, 27] train neural network models for traditional cross-language code clone detection by using data collected from competitive programming contest websites. The problem of these methods is that they require large labeled datasets for training. The dataset they use consists of different solutions by different programmers to the same problem. These code fragments have the same functionality, but not necessarily the similar control structures. It is difficult to collect a large dataset with structurally-similar cross-language code pairs for supervised model training. Moreover, the features these models choose for training the models are more focused on code semantics instead of structures.

To the best of our knowledge, [16] provides a tool for successful structurally-similar cross-language code pair detection. It notices the importance of similar control structures between code pairs, and successfully develops a technique to find such similarity. The idea of this work is to use tree-based editing-distance algorithm with generic ASTs across different programming languages. This technique first transforms a given AST in a source language into *a generic AST*, which is common tree-representation among multiple languages to absorb syntactical differences between them. Then, *tree-based editing distances* are computed to discover similar trees. The code fragments with similar trees are a potential pair with similar control structures. By mapping features from different programming languages into generic nodes, this technique creates

```python
def group_posts:
  res = {}
  for post in posts:
    bucket = res.setdefault(post.owner, [])
    bucket.append(post)
  return res
```

(a) Python code

```java
public Map<String, List<Post>> groupPosts(List<Post> posts){
  Map<String, List<Post>> grouped = new HashMap<>();
  for (Post post: posts){
    if (!grouped.containsKey(post.getOwner())){
      grouped.put(post.getOwner(), new ArrayList<Post>());
    }
    grouped.get(post.getOwner()).add(post);
  }
  return grouped;
}
```

(b) Java code

Figure 1: A structurally-similar code pair in web-services

language-agnostic common intermediate representations, and can encompass syntactic features from different languages while abstracting away syntactical differences. For example, when detecting code pairs between Java and Python, the *if* statements in Java and Python are mapped to the same kind of tree node representing an *if* statement. A list comprehension in Python is mapped to a *loop* node in the generic AST. If a feature is only supported in some of the languages, an exclusive node is still created in the generic AST, such as a *switch* statement in Java. The capability of this technique relies on how the transformation is done and how the generic ASTs are designed. This work provides a simple design and a straightforward node-to-node mapping and shows that their design works well. Although designing such a generic AST for Java and Prolog would be difficult, it is feasible for an Algol-like language family, in which a program consists of nested blocks and procedures.

This technique achieves good accuracy in finding cross-language code pairs with similar control structures. However, there are some drawbacks. The first one is the speed of detection. Constructing a generic AST is not expensive but computing editing distances between generic ASTs is heavily compute-intensive. If the number of generic ASTs is n, $n \times n$ distances must be computed. Due to this problem of being time-intensive, it is unrealistic to use this technique for large-scale detection. The second drawback is that their design of generic ASTs is ad-hoc, which makes it a bit difficult to migrate to a third language.

3 System design

To address the speed issue of tree-based editing-distance algorithm in Section 2, we develop a two-stage approach to detect structurally-similar cross-language code pairs. The system overview of our tool is shown in Fig. 2. In the first stage, we use neural network models to pre-determine a few most-likely candidates, instead of calculating editing-distances of all candidates. In the second stage, we only calculate editing-distances with the pre-determined candidates. Our uniqueness is to develop a new code representation technique called *two-level generic ASTs* as inputs for models to improve the accuracy of the first stage. The technique uses two kinds of generic ASTs with different abstraction levels, one is coarse-grained and another is fine-grained, to model source code. After getting vector representations of source code, we calculate *aggregated cosine similarity scores* and compare them to pre-determine candidates. In this paper, we use Java, Python and C as examples for structurally-similar cross-language code pair detection.

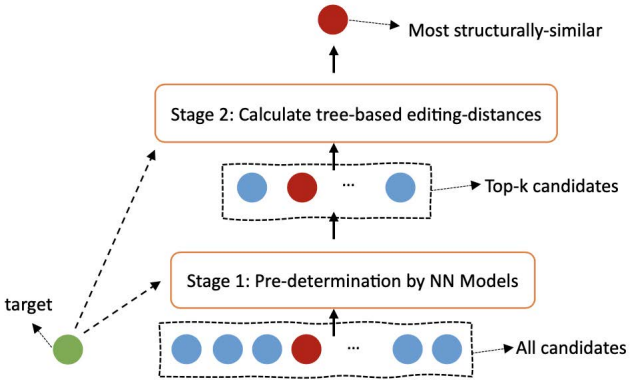

Figure 2: System overview

3.1 Two-level generic ASTs

In the first stage, we use two-level generic ASTs as inputs for models to focus on different aspects of syntactic features. A generic AST is a mapping from language-specific AST nodes to generic AST nodes with deletion, addition and modification. The coarse-grained generic ASTs focus on high-level structural features, and the fine-grained generic ASTs focus on low-level semantic features. The mapping rules for coarse-grained generic ASTs and fine-grained generic ASTs are different. In this paper, we use *JavaParser*[2], Python's *ast*[3] library, and *pycparser*[4] to generate original Java-, Python- and C-specific ASTs. We will introduce how to generate generic ASTs in this section.

A fine-grained generic AST is a rough union of syntactic features from two languages, and it preserves both common syntactic features and language-specific syntactic features. A language-specific syntactic feature means a syntactic feature that exists in one programming language, but not necessarily exists in other languages. For example, type declaration is enforced in Java, but Python does not have static types. Therefore, type declaration is a language-specific feature for Java. A common syntactic feature means a syntactic feature that exists in both languages. For example, a *for* statement exists in Java, Python and C. A common syntactic feature for two programming languages could also be a language-specific

[2]https://javaparser.org/
[3]https://docs.python.org/3.7/library/ast.html
[4]https://pypi.org/project/pycparser/

```
1 There are some AST node types in C:ID, Constant, BinaryOpBitXor, UnaryOpInvert,
      Goto, ...; Also, there are some AST node types in Python: Name, Call, Num
      , FunctionDef, body, Str, BinOpSub, ...
2 Can you generate a mapping of the common syntactic features between two
      languages? If some node types are specific in one particular language,
      just ignore them.
3
```

Figure 3: An example of prompt in generating mappings between two languages

feature for a third language. For example, both Java and Python are object-oriented languages and support class definitions. However, such feature does not exist in C.

The way of generating a fine-grained generic AST is to map common syntactic features into common generic AST nodes, and keep all other language-specific syntactic features as what they are. To do so, we look into the documentation of original ASTs and extract all syntactic features. Then, we adopt large-language models (LLMs) [5] to generate a mapping between syntactic features from two languages. We simply adopt the mapping by the LLM. If one syntactic feature does not have a corresponding mapping in another language, we regard it as a language-specific feature. The prompt we use is shown in Fig. 3:

A coarse-grained generic AST is a rough intersection of syntactic features from two languages. It ignores language-specific syntactic features, and keeps essential features that refer to common key control structures. The way of generating a coarse-grained generic AST is as follows. We first use a large amount of source code and generate original ASTs for all the source code. Then, we count the number of occurrences of each syntactic feature, and sort them based on their number of occurrences. Finally, based on the mapping in fine-grained ASTs and a hyper-parameter *node_num*, we only keep the most-frequently occurring common nodes, and ignore all other nodes. If one node is not included in coarse-grained ASTs, its children are directly concatenated to its parent, and this node is deleted. One certain rule about coarse-grained ASTs is that all kinds of binary operations and unary operations are ignored and they are mapped into common *BinaryOp* and *UnaryOp* nodes. This is to increase the variety of key control structures in the most-frequently occurring nodes. After a coarse-grained generic AST is generated, unnecessary information is ignored, and only key control structures are kept.

3.2 Model training and usage

To train the models in the first stage, a code fragment is first transformed into two generic ASTs designed in Section 3.1, and these ASTs are given as inputs to neural models after being vectorized by the path-based encoding [3]. We use an LSTM-based (Long short-term memory) encoder-decoder model [6] to generate a vector representation for the given AST, since LSTM processes sequences of data with long-range dependencies. This model is trained to predict a method name for the given AST representing its method body. In other words, it is trained so that its decoder's output will be a method name. Thus, the model training involves no true structurally-similar code pair labels.

[5]We use gpt-4-turbo API from OpenAI in this paper.

After the training, when we detect code pairs, the decoder is not used; only the encoder is used. The vector representation from the encoder's output is used as the representation of the given AST to calculate similarity between coarse-grained or fine-grained ASTs, and finally to calculate similarity between code fragments. We below give details about how the models are trained and used. The model overview is shown in Fig. 4.

We use path-based encoding [3] to transform ASTs into model inputs. A path means a shortest sequence of AST nodes from a leaf node to another leaf node with all intermediate nodes through parent-child connections. Since every node can only have one parent, every pair of leaf nodes must be able to trace back to one common ancestor, making a path between two leaf nodes unique. If an AST has n leaf nodes, it has $n \times (n-1) \div 2$ paths. For practical training, We randomly sample m paths as inputs.

After paths are extracted from an AST, we use look-up tables [17] to vectorize the paths, which transfer tokens in paths into numerical vectors for models. In our system, there are three different look-up tables. Two of them are for generic AST nodes. As the mapping rules for coarse-grained generic ASTs and fine-grained generic ASTs are different, they have independent look-up tables, and each of them has all nodes in corresponding generic ASTs. Besides them, there is a method name look-up table. A method name look-up table is consisted of method names either by camel-case or by snake-case. We split them into subtokens and use subtokens for the look-up table. Specifically, <*UNK*> is used for unknown subtokens that are not in the look-up table.

After paths and method names are vectorized, we put them into the encoder-decoder model for training. Both the encoder and the decoder are based on LSTM [11], and the architectures are referred from [3]. Given a generic AST path, the encoder gives nodes embedded by the look-up table to a bi-directional LSTM (bi-LSTM) sequentially, and use the final state from the bi-LSTM as the vector representation of this path. We incorporate attention mechanism to aggregate vector representations of paths from the same generic AST as the vector representation of this generic AST. This representation will also be used for further calculation of cosine similarity scores. The decoder gets the vector representation from the encoder for a generic AST, and it predicts a method name. During training, the predicted method name is compared with the true method name, and if it is not correct, back-propagation is used to modify the weights of models. When the model converges, the vector representation of a given generic AST can successfully be used to predict a method name.

When using the trained models to detect structurally-similar code pairs, we use the vector representations from the encoders to calculate cosine similarity scores. We get two vector representations of a code fragment, one from the model for coarse-grained generic ASTs and one from the model for fine-grained generic ASTs. To compare two code fragments, four generic ASTs are generated, and then four vector representations are generated from the ASTs by the models. By using *cosine similarity*, their similarity scores are calculated for the pair of coarse-grained ASTs and the pair of fine-grained ASTs. The final score is the sum of the two scores. In detail, given a target code fragment t and a candidate code fragment c, we calculate $score_{t-c}^{fine}$ and $score_{t-c}^{coarse}$, and add the fine-grained

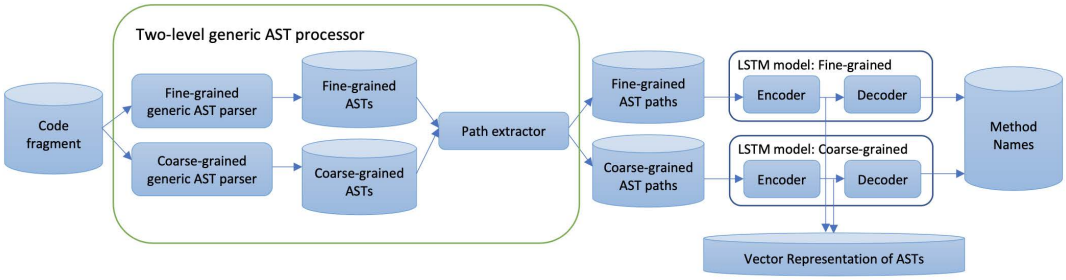

Figure 4: Model structure overview

similarity score and coarse-grained similarity score together to get an aggregated similarity score $score_{t-c}^{aggr}$. The higher the score, the higher possibility the candidate is similar to the target. Since cosine similarity is always between 0 and 1, the aggregated similarity score $score_{t-c}^{aggr}$ is always no larger than 2.

4 Experiments

We want to evaluate both accuracy and speed of our tool against other approaches in detecting structurally-similar cross-language code pairs. The first competitor is tree-based editing-distance algorithm from [16], and it is the state-of-the-art approach. It calculates the edits needed to transform an ordered labeled tree to another. Specifically, this work uses Zhang-Sasha algorithm to calculate tree-based editing distance between two generic ASTs. Since [16] provides their source code to generate Java and Python generic ASTs, we reuse their design in this algorithm. However, [16] does not provide code to generate generic ASTs for C language. We use our coarse-grained ASTs for C in this algorithm.

The second competitor is token-based Jaccard similarity algorithm from [16]. It calculates the frequency of common tokens between two code fragments by Jaccard similarity coefficient. To avoid rare tokens and limit vocabulary size, it splits identifiers to address nomenclature, and removes unnecessary tokens from source code, such as English stopwords and tokens shorter than a certain length.

The third and fourth competitors are pre-trained large neural network models. One is CodeBert [8]. The other one is Unixcoder [9], which is fine-tuned on code search dataset based on CodeBert. They both use a linear token sequence of source code as input. The sequence is generated by tokenizing source code using a lexical analyzer, and splitting camel-case or snake-case identifiers into subtokens. The models will output a vector representation of the sequence. We calculate cosine similarity of two vectors as the similarity of two code fragments.

We also do ablation study for our approach. The fifth to seventh competitors utilize neural network models only and do not combine editing-distance algorithm. We train the models on only one kind of generic ASTs (fine-grained and coarse-grained) and two-level generic ASTs respectively, and evaluate the performance of these models. The rest two competitors combine neural network models with editing-distance algorithm, but the models are trained on single kind of generic ASTs.

To train models on generic ASTs, we use Java-small [2], py150k [22] and CodeNet [20] datasets. They are large code collections based on open-source projects. The training requires no similarity labels. We only need the source files and process them into corresponding generic AST formats.

4.1 Evaluation metrics

Our approach outputs a similar code fragment among a set of candidates, which is similar to an information retrieval system. Therefore, we use average rank (AR), quantile rank, SuccessRate@k and mean reciprocal rank (MRR) [29] to measure the accuracy of approaches. Consider a given target and multiple candidates, we try to find the most structurally-similar code fragment to the target among all the candidates. In our experiment setting, there is one and only one ground truth among the candidates in a single detection. By sorting all candidates according to their similarity scores with the target, the ground truth is ranked among the candidates. If the ground truth is ranked higher, the approach is more powerful in structurally-similar code pair detection.

Consider a detection d in a set of detections D. $R(d)$ is the rank of ground truth. δ_k is the indicator function which returns 1 if the input is less than or equal to k and 0 otherwise. SuccessRate@k or SR@k is the percentage of detections for which the ground truth exists among the top-k candidates. Mathematically, AR, MRR and SR@k are defined as $AR = \frac{\sum_{d\in D} R(d)}{|D|}$, $MRR = \frac{\sum_{d\in D} \frac{1}{R(d)}}{|D|}$, $SR@k = \frac{\sum_{d\in D} \delta_k(R(d))}{|D|}$.

Quantile rank means the rank at a certain point after sorting the ranks of all detections ascendingly. For example, 25% quantile rank means the worst rank of the most well-performing 25% detections. Quantile rank represents the accuracy of the system on a certain proportion of well-performing cases.

We use the average inferring time of all detections to measure the speed of approaches. The inferring time of one detection is the time to calculate the similarity scores between the target and all candidates. To note, for machine learning models, we do not consider training time. Once a model finishes training, it can be used multiple times, and the inferring time is independent of training process.

For hardware configurations, we use an 11th Gen Intel(R) Core(TM) i5-11400 CPU with a total of 6 cores and 12 threads, a 32GB memory, and an Nvidia RTX A6000 GPU. For software configurations, we use

Table 1: Lines of code of datasets

Metrics	Java	Python	C
Average	17	12	18
25% quantile	10	6	10
50% quantile	14	10	15
75% quantile	21	15	23

PyTorch[6] to implement all machine learning models, and Numpy[7] for calculation.

4.2 Evaluation dataset

This paper focuses on detecting structurally-similar cross-language code pairs. To this extend, we find that there is not an appropriate dataset for evaluation. One of the most famous dataset available for cross-language *code clone detection* is AtCoder dataset organized by [19] and used by [16]. It comprises solutions of competitive programming problems from a competitive programming contest platform AtCoder, and the solutions are contributed by different programmers independently. Although solutions from the same problem implement the same functionality, they might have different algorithms, structures and number of code lines. Therefore, it is not suitable for evaluation of structurally-similar cross-language code pair detection. We did rough estimation and over 70 percent cases in current dataset are not suitable.

To conduct proper evaluation, one of the authors organized a new dataset based on the original AtCoder dataset with the help of a graduate student with industrial experiences. Specifically, we look into all programming contest problems in current dataset, and select solutions written by the same author. We find that the same author tends to use the same algorithm but different programming languages for the same problem. We manually check the solutions for each problem to ensure that they have similar control structures. Our criteria include number of code lines, number of key structures such as loops and conditions, and how they are structured. As a contribution, we manage to organize a dataset with 310 pairs of code fragments written in Java and Python, 100 pairs of code fragments written in Java and C and 100 pairs of code fragments written in Python and C. All these pairs have the same functionality with similar control structures. We show the number of lines of code (LoC) in Table 1.

4.3 Results

The results are presented in Table 2, Table 3 and Table 4. In Java-Python experiments, our approach can successfully pick up the most structurally-similar code fragment to the target in 63% of the cases. It achieves the best accuracy in terms of AR, SR@1, SR@3 and MRR, and outperforms all other approaches including editing-distance algorithm. Meanwhile, our approach is 20 times faster for a single detection compared with editing-distance algorithm. In Java-C experiments and Python-C experiments, our approach also leads a large advance over all other approaches except editing-distance algorithm, but is very close to the tree-based editing-distance algorithm. However, compared with tree-based editing-distance algorithm, our tool is 10 times faster for a single detection, and still keeps a very close accuracy.

The results show that our tool improves the accuracy a lot compared with other approaches in detecting structurally-similar cross-language code pairs. It achieves close accuracy compared with tree-based editing-distance algorithm, but with a much faster speed. Although our model is trained on a single Nvidia RTX A6000 GPU card for around 15 hours, the model can be used to inferring cases at a large scale. Tree-based editing-distance algorithm can not be used at the same scale because the time consumption is unacceptable. Therefore, our tool is of practical use in detecting structurally-similar cross-language code pairs.

Moreover, from the ablation study, we find that using two kinds of generic ASTs improves the accuracy compared to using only one kind of generic ASTs for all language combinations. It indicates that using two kinds of generic ASTs improves the representation quality for neural network models, and helps better find code pairs with similar structures. We will provide a discussion about this in later section. We also conduct an experiment to analyze the influence of the hyper-parameter k and see how the changes of it can affect the results. The results are conducted on Java-Python experiments and are presented in Table 5. With the increase of k, the accuracy improves, but the time also increases. The marginal change of accuracy decreases with respect to k, and the time spent is linear to k. We notice that when k increases to a certain extend, the changes of SR and MRR is very small or even zero. Overall, there is a trade-off between accuracy and time when using our tool. We should choose an appropriate hyper-parameter k to balance the speed and accuracy.

5 Discussion

We empirically show that combining a neural-network model with heuristic algorithms works well for detecting structurally-similar cross-language code pairs. Here, we provide a discussion about our generic ASTs and our two-stage approach. The results show that two kinds of generic ASTs work better than single kind of generic ASTs. In fact, this phenomenon can be seen as data augmentation. On the one hand, with only fine-grained generic ASTs, we do focus more on low-level semantic features in certain languages, but there are also some syntactic features that occur very rarely in the whole corpus. Such features may add extra noise when trying to find code pairs with similar structures. On the other hand, with only coarse-grained generic ASTs, we focus on a high-level structure of source code, but some language-specific code details are lost. For example, Python supports list comprehension that offers a shorter syntax to allow programmers to manipulate a list within one line, but such syntactic feature is ignored in coarse-grained generic ASTs. These details also help train a better model in finding code pairs with similar structures. Finding a best design of generic ASTs is subjective and difficult. We show that using two kinds of generic ASTs in an objective way also works.

We also show that a two-stage approach improves the accuracy compared with only using one stage. It is interesting that our two-stage approach works better than the pure editing-distance

[6]PyTorch version: 1.10.0
[7]Numpy version: 1.17.2

Table 2: Comparison between our approach and other approaches for Java-Python pairs

Methods	AR	25% Quan.	50% Quan.	SR@1	SR@3	MRR	Time
Token-based Jaccard similarity algorithm	67	1	15	0.34	0.41	0.391	0.36s
Tree-based editing distance algorithm	**11**	**1**	**1**	**0.51**	**0.64**	**0.605**	131s
CodeBert	95	27	73	0.03	0.06	0.068	0.018s
Unixcoder	63	5	31	0.16	0.21	0.223	0.022s
Single-level (Coarse-grained generic ASTs)	32	3	14	0.14	0.25	0.238	0.021s
Single-level (Fine-grained generic ASTs)	16	2	8	0.21	0.34	0.326	0.026s
Two-level generic ASTs	12	2	6	0.24	0.41	0.366	0.047s
Coarse-grained + Editing distance (Top-15)	30	1	3	0.48	0.5	0.511	6.36s
Fine-grained + Editing distance (Top-15)	14	1	2	0.49	0.61	0.596	6.37s
Our tool (Top-15)	**11**	**1**	**1**	**0.63**	**0.67**	**0.665**	**6.39s**

Table 3: Comparison between our approach and other approaches for Java-C pairs

Methods	AR	25% Quan.	50% Quan.	SR@1	SR@3	MRR	Time
Token-based Jaccard similarity algorithm	28	1	10	0.32	0.42	0.393	0.14
Tree-based editing distance algorithm	**3**	**1**	**1**	**0.75**	**0.86**	**0.82**	282s
CodeBert	17	3	8	0.19	0.35	0.369	0.023s
Unixcoder	18	2	7	0.24	0.44	0.439	0.028s
Only coarse-grained generic ASTs	8	1	3	0.35	0.52	0.48	0.027s
Only fine-grained generic ASTs	10	1	5	0.31	0.43	0.41	0.029s
Two-level generic ASTs	7	1	3	0.44	0.55	0.53	0.056s
Coarse-grained + Editing distance (Top-10)	7	1	1	0.6	0.75	0.71	28.2s
Fine-grained + Editing distance (Top-10)	10	1	2	0.49	0.59	0.56	28.23s
Our tool (Top-10)	**3**	**1**	**1**	**0.73**	**0.89**	**0.81**	**28.3s**

Table 4: Comparison between our approach and other approaches for Python-C pairs

Methods	AR	25% Quan.	50% Quan.	SR@1	SR@3	MRR	Time
Token-based Jaccard similarity algorithm	24	1	16	0.25	0.33	0.32	0.14
Tree-based editing distance algorithm	**3**	**1**	**1**	**0.75**	**0.88**	**0.81**	354s
CodeBert	12	5	6	0.21	0.24	0.329	0.022s
Unixcoder	13	4	6	0.16	0.21	0.223	0.027s
Single-level (Coarse-grained generic ASTs)	14	2	7	0.19	0.39	0.33	0.025s
Single-level (Fine-grained generic ASTs)	12	2	7	0.22	0.36	0.34	0.028s
Two-level generic ASTs	10	2	5	0.24	0.43	0.39	0.052s
Coarse-grained + Editing distance (Top-10)	12	1	1	0.57	0.61	0.6	35.42s
Fine-grained + Editing distance (Top-10)	10	1	1	0.6	0.63	0.63	35.43s
Our tool (Top-10)	**3**	**1**	**1**	**0.76**	**0.88**	**0.82**	**35.49s**

algorithm. This is because the editing distances of some candidates are smaller than the ground truth. After filtering in the first stage, these candidates are blocked outside top-k candidates. Therefore, the editing-distance algorithm in the second stage can successfully pick up the ground truth. In fact, if we enlarge the hyper-parameter k, the accuracy of our two-stage approach will not increase ultimately, but approach the accuracy of the pure editing-distance algorithm.

One threat of validity is our dataset. It contains code pairs with both similar control structures and the same functionality. Although

Table 5: Comparison between different top-k

Methods	AR	25% Q.	50% Q.	SR@1	SR@3	MRR	Time
Top-5	15	1	6	0.47	0.49	0.51	2.16s
Top-10	14	1	1	0.57	0.61	0.61	4.27s
Top-12	13	1	1	0.6	0.64	0.63	5.12s
Top-15	11	1	1	0.63	0.67	0.67	6.39s
Top-20	11	1	1	0.7	0.78	0.75	8.5s
Top-30	10	1	1	0.75	0.81	0.80	12.72s
Top-40	9	1	1	0.75	0.82	0.80	63.43s
Top-50	9	1	1	0.75	0.83	0.80	21.18s
Top-60	9	1	1	0.76	0.84	0.81	31.74s

Table 6: Comparison between two-level generic ASTs with other approaches on AtCoder Dataset

Methods	AR	25% Q.	50% Q.	75% Q.	Time
Jaccard sim. algorithm	86	6	45	158	0.35s
Tree-editing-distance	97	23	66	153	121s
Two-level generic ASTs	89	15	54	149	0.05s
Coarse-grained ASTs	98	27	72	155	0.016s
Fine-grained ASTs	104	26	72	174	0.33s
CodeBert	125	47	109	204	0.019s
Unixcoder	108	25	86	185	0.022s

we are targeting at structurally-similar code pairs, it is extremely hard or impossible to create a dataset with only similar structures without considering functionality. Therefore, we use this dataset for experiments. However, we conduct another experiment to prove that the same functionality has limited impact in the detection. The results are shown in Table 6. This experiment is conducted on Java-Python code pairs without similar control structures, but only the same functionality. This dataset is also based on the original AtCoder dataset and is manually created. We can see that all approaches work badly on these code pairs, indicating that structural dissimilarity could interfere the ability of these approaches a lot. It also indicates that the same functionality has very limited influence on these approaches in finding structurally-similar code pairs.

Further, the dataset is relatively small. Although we select the similar code pairs based on their authors, we still need human experts to read code solutions, and make final decisions. These experts have experiences of working in industries and writing code in multiple programming languages and are trustworthy. But it is still very time-consuming to create a dataset that contains structurally-similar cross-language code pairs in our work. To better validate the effectiveness of our approach against other approaches, a larger dataset is needed for massive evaluation.

6 Related works

Similar code pair detection is a popular topic with a long history. Before, people were more interested in single-language code clone detection. For example, [13] and [25] develop token-based matching algorithms to find clones, while [12] develops an AST-based matching algorithm to detect clones. These approaches rely on predefined rules to generate matching patterns, and use static-analysis algorithms to compare patterns. After machine learning shows its dominating power in other fields, researchers want to apply its power to code clone detection. [15] develops a solely token-based clone detection approach using deep learning. [30] builds a tree-LSTM model based on hashed features of information about code structure and grammar extracted from ASTs. [31] splits ASTs into trees consisting of statement nodes as roots and corresponding AST nodes of statements, and builds a Bi-GRU model upon the encoded vectors of these statement trees. All these approaches build neural network models based on code-related data structures, and rely on true code clone labels.

Compared with single-language code clone detection, cross-language code clone detection has a shorter history, but faster growth in attention since development in multiple programming languages becomes trendy. [19] trains an LSTM-based neural network using ASTs and uses tree-based skip-gram algorithm to initialize node vectors. It also provides a large cross-language code clone dataset based on competitive programming contest submissions. [27] adopts a contrastive learning objective to fine-tune the pretrained model CodeBert to detect clones. [18] trains a model based on API documentation to detect clones. But these approaches still require true labels of code clones. Specially, [26] questions the generalizability of CodeBert, and points out that CodeBert performs less than expected on unseen data in code clone detection task. There are also some works avoiding using machine learning models. [5] analyzes revision logs and version control histories to find cross-language clones. [28] analyzes an enriched Concrete Syntax Tree, but requires two code fragments to have the same code length. [14] detects cross-language code clones in the .NET language family by translating a program in a .NET language into the .NET Code Document Object Model (CodeDOM). However, all these works either focus on functionality and neglect structural similarity, or they are only limited in .NET family of programming languages. One of the most recent works is [16], which utilizes static analysis of code text and ASTs and dynamic analysis of input/output results to find clones. This work provides an approach to detect structurally-similar cross-language code pairs. However, the drawback of this approach is that it is very slow. As far as we know, there is not a fast and accurate tool targeting at detecting structurally-similar cross-language code pairs.

This paper is based on a poster paper [7] written by the same authors. There are three major differences between this paper and the poster paper. First, we redesign and formalize the algorithms for generating two kinds of generic ASTs for Java, Python and C. The new algorithms avoid ad-hoc designs of generic ASTs, and provide an accuracy improvement in the evaluation. Second, we expand our evaluation to Java-C code pairs and Python-C code pairs, which is not included in the poster paper. Third, we conduct an evaluation on Java-Python code pairs with only the same functionality to prove that the same functionality has limited influence on our tool.

7 Conclusion

In this paper, we improve the speed of detecting structurally-similar cross-language code pairs between Java, Python and C pairwise,

and keep a very close accuracy compared with the state-of-the-art approach. We achieve this by developing a two-stage approach, which first determines a few most-likely candidates using neural network models, and then calculates editing distances deterministically to select the most similar code fragment. To promote the accuracy of neural network models, we use two-level generic ASTs as inputs for the models. We conduct our experiments on Java-Python, Java-C and Python-C code pairs, and successfully show that our approach works on these datasets.

Acknowledgments

This work is partly supported by JSPS KAKENHI JP20H00578 and JP24H00688.

References

[1] Farouq Al-omari, Iman Keivanloo, Chanchal Roy, and Juergen Rilling. 2012. Detecting Clones Across Microsoft .NET Programming Languages. *Proceedings - Working Conference on Reverse Engineering, WCRE*, 405–414. https://doi.org/10.1109/WCRE.2012.50

[2] Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A Convolutional Attention Network for Extreme Summarization of Source Code. In *International Conference on Machine Learning*. PMLR, 2091–2100.

[3] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. Code2vec: Learning Distributed Representations of Code. *Proc. ACM Program. Lang.* 3, POPL, Article 40 (jan 2019), 29 pages. https://doi.org/10.1145/3290353

[4] Zoran Budimac, Gordana Rakić, and Miloš Savić. 2012. SSQSA Architecture. In *Proceedings of the Fifth Balkan Conference in Informatics* (Novi Sad, Serbia) *(BCI '12)*. Association for Computing Machinery, New York, NY, USA, 287–290. https://doi.org/10.1145/2371316.2371380

[5] Xiao Cheng, Zhiming Peng, Lingxiao Jiang, Hao Zhong, Haibo Yu, and Jianjun Zhao. 2017. CLCMiner: Detecting Cross-Language Clones without Intermediates. *IEICE Transactions on Information and Systems* E100.D (02 2017), 273–284. https://doi.org/10.1587/transinf.2016EDP7334

[6] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, Doha, Qatar, 1724–1734. https://doi.org/10.3115/v1/D14-1179

[7] Feng Dai and Shigeru Chiba. 2024. A practical tool for detecting cross-language code pairs with similar control structures. In *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing* (Avila, Spain) *(SAC '24)*. Association for Computing Machinery, New York, NY, USA, 1301–1303. https://doi.org/10.1145/3605098.3636134

[8] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, Online, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[9] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. 7212–7225. https://doi.org/10.18653/v1/2022.acl-long.499

[10] Aakanshi Gupta and Bharti Suri. 2018. *A Survey on Code Clone, Its Behavior and Applications*. 27–39. https://doi.org/10.1007/978-981-10-4600-1_3

[11] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Computation* 9, 8 (1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

[12] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In *29th International Conference on Software Engineering (ICSE'07)*. 96–105. https://doi.org/10.1109/ICSE.2007.30

[13] T. Kamiya, S. Kusumoto, and K. Inoue. 2002. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering* 28, 7 (2002), 654–670. https://doi.org/10.1109/TSE.2002.1019480

[14] N.A. Kraft, B.W. Bonds, and R.K. Smith. 2008. Cross-Language Clone Detection. In *Proceedings of the 20th International Conference on Software Engineering and Knowledge Engineering (SEKE'08)* (San Francisco, CA, USA). 54–59.

[15] Liuqing Li, He Feng, Wenjie Zhuang, Na Meng, and Barbara Ryder. 2017. CCLearner: A Deep Learning-Based Clone Detection Approach. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 249–260. https://doi.org/10.1109/ICSME.2017.46

[16] George Mathew and Kathryn T. Stolee. 2021. Cross-Language Code Search Using Static and Dynamic Analyses. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 205–217. https://doi.org/10.1145/3468264.3468538

[17] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. arXiv:1301.3781 [cs.CL] https://arxiv.org/abs/1301.3781

[18] Kawser Wazed Nafi, Tonny Shekha Kar, Banani Roy, Chanchal K. Roy, and Kevin A. Schneider. 2019. CLCDSA: Cross Language Code Clone Detection using Syntactical Features and API Documentation. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering*. 1026–1037. https://doi.org/10.1109/ASE.2019.00099

[19] Daniel Perez and Shigeru Chiba. 2019. Cross-language Clone Detection by Learning over Abstract Syntax Trees. In *Proceedings of the 16th International Conference on Mining Software Repositories* (Montreal, Quebec, Canada) *(MSR '19)*. IEEE Press, Piscataway, NJ, USA, 518–528. https://doi.org/10.1109/MSR.2019.00078

[20] Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. 2021. CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks. arXiv:2105.12655 [cs.SE] https://arxiv.org/abs/2105.12655

[21] Gordana Rakić and Zoran Budimac. 2011. Introducing enriched concrete syntax trees. *Proc. of the 14th International Multiconference on Information Society (IS), Collaboration, Software And Services In Information Society (CSS)* A (01 2011), 211–214.

[22] Veselin Raychev, Pavol Bielik, and Martin Vechev. 2016. Probabilistic Model for Code with Decision Trees. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) *(OOPSLA 2016)*. Association for Computing Machinery, New York, NY, USA, 731–747. https://doi.org/10.1145/2983990.2984041

[23] Chanchal Kumar Roy and James R Cordy. 2007. A survey on software clone detection research. *Queen's School of Computing TR* 541, 115 (2007), 64–68.

[24] Chanchal K. Roy and James R. Cordy. 2008. NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization. In *2008 16th IEEE International Conference on Program Comprehension*. 172–181. https://doi.org/10.1109/ICPC.2008.41

[25] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. 2016. SourcererCC: Scaling Code Clone Detection to Big-Code. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 1157–1168. https://doi.org/10.1145/2884781.2884877

[26] Tim Sonnekalb, Bernd Gruner, Clemens-Alexander Brust, and Patrick Mäder. 2023. *Generalizability of Code Clone Detection on CodeBERT*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3551349.3561165

[27] Chenning Tao, Qi Zhan, Xing Hu, and Xin Xia. 2022. C4: Contrastive Cross-Language Code Clone Detection. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension* (Virtual Event) *(ICPC '22)*. Association for Computing Machinery, New York, NY, USA, 413–424. https://doi.org/10.1145/3524610.3527911

[28] Tijana Vislavski, Gordana Rakić, Nicolás Cardozo, and Zoran Budimac. 2018. LICCA: A tool for cross-language clone detection. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering*. 512–516. https://doi.org/10.1109/SANER.2018.8330250

[29] Ellen M. Voorhees and Dawn M. Tice. 2000. The TREC-8 Question Answering Track. In *Proceedings of the Second International Conference on Language Resources and Evaluation (LREC'00)*, M. Gavrilidou, G. Carayannis, S. Markantonatou, S. Piperidis, and G. Stainhauer (Eds.). European Language Resources Association (ELRA), Athens, Greece. http://www.lrec-conf.org/proceedings/lrec2000/pdf/26.pdf

[30] Hui-Hui Wei and Ming Li. 2017. Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence* (Melbourne, Australia) *(IJCAI'17)*. AAAI Press, 3034–3040.

[31] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A Novel Neural Source Code Representation Based on Abstract Syntax Tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 783–794. https://doi.org/10.1109/ICSE.2019.00086