# Algorithm Identification in Programming Assignments

Pranshu Chourasia Ganesh Ramakrishnan Varsha Apte Suraj Kumar
{pranshuiitb,ganesh,varsha,surajkumar}@cse.iitb.ac.in
Indian Institute of Technology - Bombay. Mumbai, India

## ABSTRACT

Current autograders of programming assignments are typically program output based; they fall short in many ways: e.g. they do not carry out subjective evaluations such as code quality, or whether the code has followed any instructor specified constraints; this is still done manually by teaching assistants. In this paper, we tackle a specific aspect of such evaluation: to verify whether a program implements a specific *algorithm* that the instructor specified. An algorithm, *e.g.* bubble sort, can be coded in myriad different ways, but a human can always understand the code and spot, say a bubble sort, *vs.* a selection sort. We develop and compare four approaches to do precisely this: given the source code of a program known to implement a certain functionality, *identify the algorithm* used, among a known set of algorithms. The approaches are based on code similarity, Support Vector Machine (SVM) with tree or graph kernels, and transformer neural architectures based only source code (CodeBERT), and the extension of this that includes code structure (GraphCodeBERT). Furthermore, we use a model for explainability (LIME) to generate insights into why certain programs get certain labels. Results based on our datasets of sorting, searching and shortest path codes, show that GraphCodeBERT, fine-tuned with *scrambled source code*, i.e., where identifiers are replaced consistently with arbitrary words, gives the best performance in algorithm identification, with accuracy of 96-99% depending on the functionality. Additionally, we add *uncalled function source code elimination* to our pre-processing pipeline of test programs, to improve the accuracy of classification of obfuscated source code.

**ACM Reference Format:**
Pranshu Chourasia Ganesh Ramakrishnan Varsha Apte Suraj Kumar. 2022. Algorithm Identification in Programming Assignments. In *30th International Conference on Program Comprehension (ICPC '22), May 16–17, 2022, Virtual Event, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3524610.3527914

## 1 INTRODUCTION

The assessment of programming assignments submitted by students in a course on programming is a time consuming exercise. For the assessment to be fair and effective, a grader has to not only ensure that the program *works* as expected but often has to carry out a *subjective* evaluation which involves carefully reading and *understanding* the source code. The grader may have to figure out *how* a program is doing what it is doing and may have to evaluate it for various subjective qualities such as commenting, choice of variable names, or the elegance of the algorithm used. The grader may further have to ensure that the program meets specific constraints on the code (*e.g.*, use of certain data structures or algorithms) set by the instructor. All this would imply a non-trivial amount of time required for grading just one student's program. With class sizes of programming courses increasing to thousands in universities, and to tens of thousands in MOOCs (massively open online courses), assessment based on source-code comprehension has become a nightmare in terms of human resource requirement and has a compelling need to be automated.

There's a growing body of emerging research that formulates the problem of a computer "understanding" the semantics of a given source code as a machine learning "task". Much of the focus has been primarily to aid industrial software development. Tasks of interest include code summarization, method name generation, code completion and code generation based on natural language [23, 25, 31]. Most of these employ language models from the natural language processing domain, such as n-gram, Recurrent Neural Networks and Convolutional Neural Networks, and specialize them for the programming language tasks. One such recent work is CodeBERT [20], which is a bimodal pre-trained model for programming language (PL) and natural language (NL), and was demonstrated to work well for tasks such as natural language code search, and code document generation.

Similarly, in recent years, work on auto-assessment of programming assignments on aspects *other than* functionality, that require comprehension of the program, has started emerging. The tasks of interest here include, but are not limited to (i) giving useful feedback to students that can help them complete their programming assignment, and (ii) grading on qualitative metrics according to instructor-provided rubrics [22, 30, 37].

In this paper, we consider a specific problem within the realm of assessment of programming assignments, which is of *identifying* the *algorithm* that a student has used in programming a certain functionality. This is of special interest in an educational setting. Let us say, an instructor gives an assignment to students to implement sorting using Quick Sort. Current autograders *only* grade for functionality, which they check by running the submitted program on a variety of testcases. A student who could not get quicksort working, but say, could implement insertion sort, could write and submit insertion sort code, which would then pass all the testcases perfectly, even though it is a completely invalid submission. Another example where algorithm identification is required is if students are asked to implement any of the known sorting algorithms, and a grading rubric awards different marks based on the algorithm implemented. If we have to do this at scale, the autograder itself (and not a human) must be able to *identify* algorithms used in programming assignment submissions. To our knowledge, this paper is the first time this problem has been addressed, in general for recognising algorithms of *any* functionality.

ACM ISBN 978-1-4503-9298-3/22/05…$15.00
https://doi.org/10.1145/3524610.3527914

We describe and contrast four learning approaches to solve this problem. The first uses a plagiarism detection tool (MOSS [34]) which produces similarity scores between pairs of programs. The second uses an SVM model with tree and graph kernels [24, 27] corresponding to the AST (Abstract Syntax Tree) and CFG (Control Flow Graph) respectively of the source code; the third is CodeBERT [20], which provides a contextual embedding of the source code based on the transformer neural architecture and the fourth is GraphCodeBERT [21] which extends CodeBERT to learn structural representations of source code through their DFGs (Data Flow Graph). We compared the four methods by applying them to a dataset from a competitive coding website, which included source code for three types of functionalities which can be implemented using multiple algorithms: sorting (six algorithms), searching (two algorithms) and shortest paths (two algorithms). We find that the CodeBERT model is able to achieve an accuracy of 90-99 % on the task of identifying the specific algorithm used in these programming tasks, and GraphCodeBERT improves this further to 96-99%. Furthermore, both can identify algorithms "camouflaged" to look like some other algorithms. CodeBERT catches such camouflaging based on misleading identifier names, and GraphCodeBERT is able to identify the actual algorithm being used in the code that might be littered with redundant code fragments that do not get invoked.

Specifically we make the following contributions in this paper:

- We propose and implement a machine learning approach based on CodeBERT and GraphCodeBERT to solve the task of algorithm identification in source code.
- We show how adding the steps of *scrambling* the source code identifiers, and *dead function code elimination* in the processing steps can make the identification robust.
- We propose two other methods: similarity score based, and an SVM based approach that captures the program structure, and show that these do not perform as well as the CodeBERT based methods for this task.
- We use an Explainability model (LIME: Local Interpretable Model-agnostic Explanations [32]) to understand how CodeBERT and GraphCodeBERT work in identifying the algorithm, and what input features contribute the most to the classification.

The rest of the paper is as follows: Section 2 presents the background required (MOSS, kernels, CodeBERT,GraphCodeBERT); Section 3 presents our methodology for identification using each of these models. Section 4 presents results of the identification, and their analysis and discussion. Section 5 presents related work, and we conclude the paper in Section 6.

## 2 BACKGROUND

In our work on algorithm identification in source code of programming assignments, we present and explore three different methodologies: plagiarism detection, Support Vector Machine (SVM) classification with tree and graph kernels, and transformer neural network based architectures. Our goal in evaluating various methods was to identify as computationally lightweight a model as possible that would yield good results on the task of automatic algorithm identification. In the next few sections we review the background required to understand how and why these models and methods can be applied to our task.

### 2.1 MOSS

MOSS (Measure Of Software Similarity) [34] is a software tool for determining the similarity of programs. The main application of MOSS has been to detect plagiarism in programming assignment submissions by students. It takes as input a set of programs and calculates their pair-wise similarity and gives a percentage similarity score to each pair. It is capable of spotting similarity between programs despite change of variable names, or movement of methods within the source code. It can also identify similarity between code fragments. Our interest in MOSS was was based on the question: given a set of programs which carry out the same functionality, will MOSS flag a pair of programs as more similar if they use the same *algorithm*, than a pair of programs that use different algorithms? In other words would a high similarity score in MOSS, imply that the programs used the same algorithm? If so, can we use MOSS along with a labeled dataset for algorithm identification?

We answer this question in Section 3.1, where we describe how we use the MOSS similarity scores of a set of programs used as training data, and the similarity score of a test program with all these programs, to label the algorithm that the test program implements.

### 2.2 Kernel Methods

Kernel methods, a popular technique used in machine learning, are a lightweight means of calculating the similarity between structured representation of any two entities. While tokens and related features are useful as an essential program characteristic, it is evident that the program *structure* is also very key determinant in the semantics of a program. A program can be represented by structures such as an AST, DFG and CFG [33]. A variety of Kernel functions exist in literature that can help us compare these structures; for our purpose, we need a *tree kernel* for the AST of the code and a *graph kernel* for the CFG of the code [27, 36].

There are mainly two well-known tree kernels: the subtree kernel, and the subset tree kernel; where a subtree of a tree, is a node of the tree with all its descendents, whereas a subset tree is a less constrained subset of the original tree. An efficient method for computing both of these was proposed by Moschitti [27], along with a readily available implementation, which we use in this work.

There are various types of graph kernels available for comparing graphs: Shortest Path Kernel, Graphlet Kernel, Subgraph Matching Kernel, Weisfeiler-Lehman Kernel, Random Walk Kernel etc. The survey by Kriege et al [24] gives a comprehensive description of these kernels. The following brief outline here is based largely on that survey:

- Shortest Path kernels: compare the number and lengths of the shortest paths between all pairs of vertices in two graphs.
- Random walk kernel: It compares labeled sequences generated by random walks in two graphs.
- Subgraph Matching Kernel: It computes bijections between all subgraphs of at most $k$ vertices of two graphs. Additionally, a custom vertex matching kernel can also be used.

- Graphlet Kernel: Graphlets are subgraphs of a fixed size. This kernel is based on counting number of graphlets of a particular pattern that appear in a graph.
- Weisfeiler-Lehman Kernel: This kernel entails performance of the following steps for a few iterations: represent nodes by a sorted list of its neighbours and use a compression hash and relabel the node. Use of this kernel has yielded high accuracy in matching similar graphs.

The essential premise of our kernel method approach (Section 3.2). is that programs that implement the same algorithm must be similar in their ASTs or CFGs, and further that one or more of the above kernels may capture that similarity.

## 2.3 CodeBERT

CodeBERT is a "bimodal pre-trained transformer" model which can be used for a variety of NL-PL tasks [20]. It is "bimodal" in that it is trained on a data set that includes both source code and natural language (comments, code documentation, etc). It learns a contextual embedding of the source code which can then be used for several downstream tasks.

The CodeBERT model is based on multi-layer bidirectional Transformer Encoder [41]. The encoder is composed of a stack of 12 identical layers. Each layer has two sub-layers. The first is a multi-head self-attention layer, and the second is a simple, position-wise fully connected feed-forward network. There is residual connection around each of the two sub-layers, followed by layer normalization. The total number of model parameters is 125M.

It has been trained on a massive dataset from github repositories (8 million records), because it does self-supervised learning. That is, it does not require a *labeled* or *annotated* dataset, rather it learns using masked language modeling (MLM) and replaced token detection.

CodeBERT can be further fine-tuned for a variety of NL-PL tasks such as searching code using natural language or document generation. We fine-tuned CodeBERT using a labeled dataset of source codes which are known to be implementing a certain algorithm for a given functionality (Section 3.3). Although our algorithm identification task does not look like an NL-PL task, it can also be thought of in that manner, where the task is to give a one-word natural language "descriptor" to the algorithm used in the code.

## 2.4 GraphCodeBERT

CodeBERT uses only a sequence of tokens from source code, and natural language descriptions to learn code embeddings. However this often results in confusion in what the program is actually doing, if, e.g., only a small part of the source code is actually invoked during run time. GraphCodeBERT [21] addresses this problem by including *Data Flow Graphs (DFG)* as an input to the model. GraphCodeBERT is also transformer based, but they also use a *variable sequence* derived from the AST as an input. Further, they use a mask matrix that captures attention between a pair of variables if there is an edge between them in the dataflow graph, and between the variables from the dataflow graph and the original source code token.

GraphCodeBERT is pre-trained on two additional tasks apart from MLM: Edge prediction of the DFG, and "alignment" prediction between the data flow variable and the source code token.

Section 3.4 describes how we use CodeBERT to address the limitations of all other approaches, including CodeBERT, to identify algorithms in source code.

In the next section we present details of our algorithm identification methods based on the above four broad approaches.

## 3 ALGORITHM IDENTIFICATION METHODS

Recall our problem statement, which is as follows:

*Design a system which takes as input, source code which is known to carry out a given functionality (*e.g.*, sorting), and identifies the algorithm used by this source code, among a known set of algorithms for this functionality.*

An algorithm can be coded in different ways. *e.g.* the bubble sort algorithm when coded has a lot of variations due to different coding styles of the programmers, ascending or descending order of sorting, type of data structure used to store data (vectors, arrays, lists, *etc.*), type of data (strings, integers, decimal numbers, *etc.*), type of loops (for *vs.* while), organization into functions, whether arguments are passed by value or reference, and finally, even the use of different programming languages (C, Python, Java, *etc.*). Regardless of these variations, expert human programmers are able to recognize an algorithm based on the "logic" visible in the code, even if the code is not well commented, or variable names are not helpful. We aim for our ML solution to be similarly robust.

A data driven automated approach to solve this problem would necessarily be based on learning the statistical patterns that are common to codes implementing the same algorithm, and then check whether a test program matches those statistical patterns. As described earlier, we develop methods based on source code similarity scorers, tree and graph kernels, and finally neural architectures. We now discuss the implementation details of all four methods (MOSS, Kernel, CodeBERT, GraphCodeBERT) for the task of algorithm identification.

## 3.1 Algorithm Identification using MOSS

We use MOSS to devise a "first principles" statistical method to identify an algorithm in a given source code. The following are the steps of our method to train our model for a set of algorithms $S$ for a given functionality

1. Start with a dataset of size $N$ of *reference source codes* for *each* of the types of algorithms for this functionality. If there are $K$ type of algorithms (*i.e.*, $K$ is the size of the set $S$), the total number of programs will be $N \times K$.
2. For the reference source code sets for each given type of algorithm, use MOSS to calculate a similarity score (in percentage) for *each pair of programs*. Do this for each type of algorithm.
3. Find the minimum and maximum pair-wise similarity of the reference set for each algorithm $a$. Denote these by $\sigma_l^a$ and $\sigma_h^a$ respectively.

Now apply the following steps to classify a test program $P$.

1. Using MOSS, calculate the similarity score of the test program with each of the $N$ reference programs for each algorithm $a$ in the set $S$. $N \times K$ similarity scores will be calculated here.

(2) Let $\sigma_l(P, a)$ be the lowest similarity score that the test program $P$ has with any reference program which implements algorithm $a \in S$. Similarly, let $\sigma_h(P, a)$ be the highest similarity score program that the test program $P$ has with any reference program which implements algorithm $a \in S$.
(3) A program is classified as implementing algorithm $a^*$ if its maximum similarity score with programs of algorithm $a^*$ is greater than the minimum similarity score of the reference set for algorithm $a^*$, and its maximum similarity score with programs of other algorithms is lower than the minimum similarity scores of those reference sets. That is $\sigma_h(P, a) < \sigma_l^a \ \forall a \neq a^*$ and $\sigma_h(P, a^*) > \sigma_l^{a^*}$

## 3.2 Support Vector Machines with Tree/Graph Kernel

Here, we use well known representations of programs that capture structure, such as AST and CFG [33]. These can be used along with the tree and graph kernels reviewed earlier to implement kernel-based methods such as the Support Vector Machine (SVM).

Our implementation pipeline for this approach is as follows:

(1) If using the AST, convert a program into its AST representation using the Tree-Sitter compiler tool [15] which gives a linear representation of the code to tree kernel.
If using CFG, we first use joern tooling system [1] to convert the code into a CFG and export this graph into graphvix dot format. Then we use graph-easy [3] to convert the CFG from dot format into graphml format. This is required because we use GraKel kernel method implementation [36] which has support for graphml format with support for conversion into adjacency matrix representation.
(2) Once we have the AST and CFG in usable formats, we apply the Tree/Graph Kernels [27] as discussed in the previous section, on AST/CFG respectively to find the kernel matrix. We have employed the tree kernel implementation of SVM-LIGHT-TK 1.5 [18] and graph kernel implementation of GraKel [9]
(3) Apply SVM Classifier on Precomputed Kernel Matrix to obtain the prediction for test programs.

## 3.3 CodeBERT for Algorithm Identification

The CodeBERT pre-trained model can be downloaded from the HuggingFace Transformer Library [10]. There are different settings for using CodeBERT in downstream tasks. For the task of algorithm identification, we formulate it as feeding code tokens as input to the model with an additional fully connected classification layer as the last layer of the model. The problem of identifying the correct algorithm is posed as a multi-class classification problem.

| Model HyperParameters | CodeBERT | GraphCodeBERT |
|---|---|---|
| Code+Dataflow seq length | 256 | 512 + 128 |
| Batch Size | 16 | 16 |
| Learning Rate | 5e-5 | 2e-5 |
| Number of Epochs | 8 | 16 |
| Optimizer | Adam | Adam |
| Max Gradient Norm | 1.0 | 1.0 |

**Table 1: Hyperparameters for CodeBERT, GraphCodeBERT**

We use model hyper-parameters as shown in Table 1; they are similar to the ones used in the original CodeBERT and GraphCodeBERT [14] paper for demonstrating their results.

## 3.4 GraphCodeBERT for Algorithm Identification

We use the distribution found on github [14] for implementing our GraphCodeBERT based algorithm identification pipeline. We again have a last fully connected layer for classification. For the DFG variable sequence, the GraphCodeBERT distribution itself comes with a parser module, which uses the Tree-sitter tool to first create the AST from the source code, and then their own code finds the dataflow dependencies and outputs the DFG and the variable sequence.

## 4 RESULTS AND ANALYSIS

In this section we present the results of our algorithm identification method, applied to three source code datasets [19]. We first describe the dataset, then the performance metrics of the task such as precision and recall scores. We then delve into explaining how our method works and what features of the source code can be attributed to identifying an algorithm, using the LIME explainability model [32]. This helped us identify a major drawback of our model, which we fixed. We present the improved results also in this section.

## 4.1 Dataset

The dataset is crawled from the AIZU Online Judge [2]. We collected source code for three categories of functionalities: sorting, searching, and shortest paths. The dataset for sorting functionality consists of six types of sorting algorithms: bubble, insertion, selection, counting, merge, and quick sort; the searching functionality consists of two types of algorithms: linear and binary search; and the shortest path in graphs consists of two types of algorithms: single source and all pair shortest path. While the shortest path algorithms are not for *exactly* the same functionality, we claim that the class of shorted path algorithms still fits under our algorithm identification problem statement, since the single source shortest path algorithm may be re-purposed as an all pairs shortest path algorithm. The dataset sizes are specified in Table 2. For the sorting dataset, 85% of the data was used for training, and 15% of the data was used as test data. For search and shortest paths since the dataset was smaller, the split was 80% – 20%, so as to have an adequately sized test dataset.

## 4.2 Results and Insights

The metrics used for the evaluation of our algorithm identification method on the unseen test data are precision (P) and recall (R) in percentage [8].

The results for all the models we used (except GraphCodeBERT) for sorting, searching and shortest paths algorithms are reported in Table 3. Our results show that the baseline performance of algorithm identification using MOSS is not so good, and neither is SVM with Random Walk graph kernel. Results are better for the other SVM kernels, with the best one being for Weisfeiler Lehman Kernel. The precision and recall with CodeBERT however are both *100%*. This

| Algorithm | Source Code Count |
|---|---|
| Bubble Sort (B) | 5229 |
| Insertion Sort (I) | 6139 |
| Selection Sort (S) | 4817 |
| Counting Sort (C) | 2384 |
| Merge Sort (M) | 2738 |
| Quick Sort (Q) | 1785 |
| Linear Search (Lin) | 3861 |
| Binary Search (Bin) | 4440 |
| Single Source Shortest Path (SSSP) | 353 |
| All Pair Shortest Path (APSP) | 365 |

**Table 2: Sizes of the datasets**

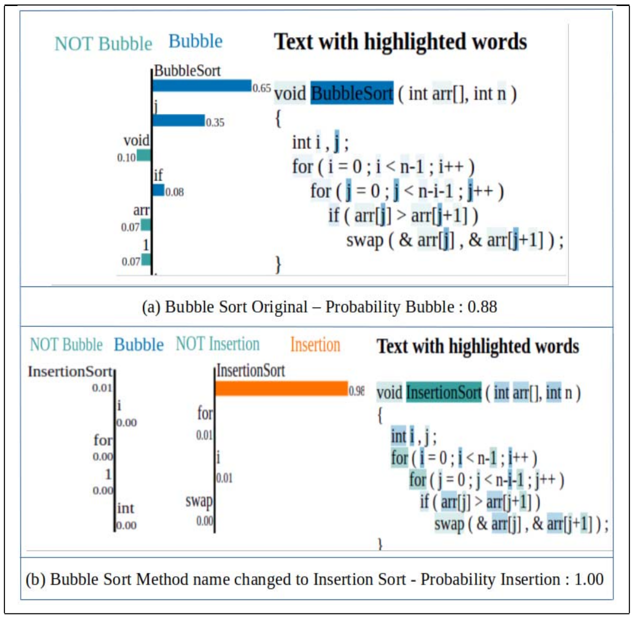
score is very surprising at first glance, and motivated us to dig deeper into understanding *how* and *why* the CodeBERT model predicts the correct class so accurately. What are the signals and patterns it responds to and are important in labeling an algorithm? The next section answers this question.

| | Sorting | | Searching | | Path | |
|---|---|---|---|---|---|---|
| **Model** | **P %** | **R %** | **P %** | **R %** | **P %** | **R %** |
| CodeBERT | 100 | 100 | 100 | 100 | 100 | 100 |
| SVM with TK | 82.00 | 78.00 | 75.00 | 71.00 | 80.00 | 79.00 |
| SVM with WLK | 85.00 | 83.00 | 86.00 | 86.00 | 87.00 | 85.00 |
| SVM with SMK | 83.00 | 77.00 | 84.00 | 81.00 | 85.00 | 85.00 |
| SVM with NHK | 79.00 | 76.00 | 78.00 | 77.00 | 84.00 | 84.00 |
| SVM with GK | 73.00 | 71.00 | 75.00 | 73.00 | 82.00 | 80.00 |
| SVM with RWK | 53.00 | 49.00 | 51.00 | 47.00 | 76.00 | 75.00 |
| MOSS | 67.00 | 65.00 | 57.00 | 51.00 | 65.00 | 64.00 |

**Table 3: Precision & Recall scores for three datasets (original source code). Kernels: TK: Tree, SMK: SubGraph Matching, GK: Graphlet, RWK: Random Walk, NHK: Neighbourhood Hash, WLK: Weisfeiler Lehman**

### 4.2.1 *Model Explanation and Features Insight.*

To attribute the prediction of the neural network to its input features, we use the *LIME* (Local interpretable Model-agnostic Explanations) approach. LIME works using "local surrogate" models, which are sparse linear approximations of the original black box model, created by perturbing the input and checking the model's behaviour with respect to these perturbations. We have used the implementation of LIME provided by the *transformer-interpret* library [17]. The output of this tool includes the prediction probabilities for each label, and numeric attributions of these probabilities to each token in the test source code. This helps us understand why the identification works when it does, and why it fails when it does not work. This can also help highlight existing problems in the identification approach, that did not reveal themselves through accuracy scores, due to some peculiar characteristics of the particular test data that was used to generate those scores.



```
void BubbleSort ( int arr[], int n )
{
   int i , j ;
   for ( i = 0 ; i < n-1 ; i++ )
      for ( j = 0 ; j < n-i-1 ; j++ )
         if ( arr[j] > arr[j+1] )
            swap ( & arr[j] , & arr[j+1] ) ;
}
```

(a) Bubble Sort Original – Probability Bubble : 0.88

```
void InsertionSort ( int arr[], int n )
{
   int i , j ;
   for ( i = 0 ; i < n-1 ; i++ )
      for ( j = 0 ; j < n-i-1 ; j++ )
         if ( arr[j] > arr[j+1] )
            swap ( & arr[j] , & arr[j+1] ) ;
}
```

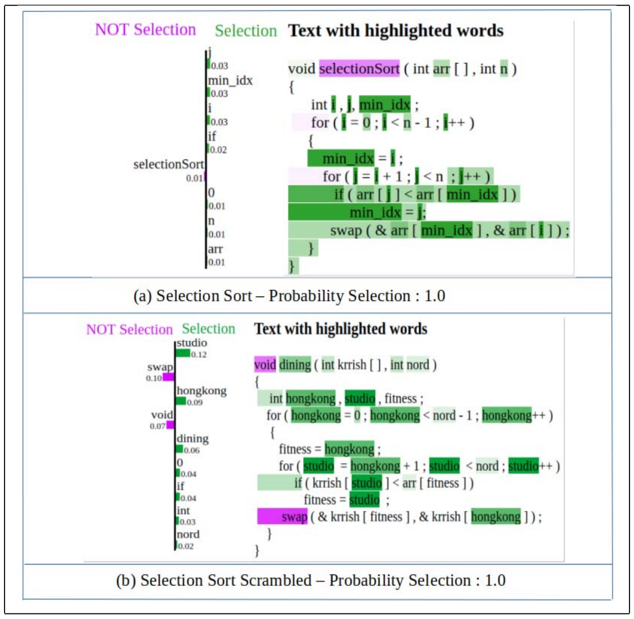
(b) Bubble Sort Method name changed to Insertion Sort - Probability Insertion : 1.00

**Figure 1: LIME explanation of bubble sort classification. (a) Original, (b) Method name changed to InsertionSort. Code taken and modified from [4].**

In Figure 1, we present the output of LIME when applied to a program that implemented bubble sort. From left to right, LIME output depicts the probabilities of the algorithms, the attribution scores to various tokens of the source code, and then the source code with the important tokens highlighted. Figure 1(a) shows that the name of the method "bubblesort" has been attributed a large score. This pattern was visible in all the test programs that we analyzed using LIME. This suggests that the other patterns in the code had very little to do with the classification; it was all based on the favourable choice of a method's name.

This explanation exposes a major vulnerability in the method - someone who intentionally want to submit, say, bubble sort instead of insertion sort, could simply rename her/his method to "insertionsort" and get the program labeled correctly. Figure 1(b) shows this case. Here, we took the same program and simply changed the method name to "InsertionSort", and it was classified as insertion sort by our method. Thus while CodeBERT showed a precision and recall score of 100% this was not at all based on any true "comprehension" of which algorithm the code was implementing, but on a superfluous signal of the method name.

We updated our ML training pipeline to address this problem, as described next.



```
void selectionSort ( int arr [ ] , int n )
{
    int i , j, min_idx ;
   for ( i = 0 ; i < n - 1 ; i++ )
    {
      min_idx = i ;
      for ( j = i + 1 ; j < n  ; j++ )
        if ( arr [ j ] < arr [ min_idx ] )
          min_idx = j;
        swap ( & arr [ min_idx ] , & arr [ i ] ) ;
    }
}
```

```
void dining ( int krrish [ ] , int nord )
{
  int hongkong , studio , fitness ;
  for ( hongkong = 0 ; hongkong < nord - 1 ; hongkong++ )
  {
    fitness = hongkong ;
    for ( studio  = hongkong + 1 ; studio  < nord ; studio++ )
      if ( krrish [ studio ] < arr [ fitness ] )
        fitness = studio  ;
    swap ( & krrish [ fitness ] , & krrish [ hongkong ] ) ;
  }
}
```

**Figure 2: LIME Explanation of source code (a) without scrambling and (b) with scrambling. Code taken and modified from [7].**

## 4.3 Results with Scrambled Source Code

We updated our training pipeline by adding a step of *scrambling* the source code. Now, before feeding the token sequence to CodeBERT in the training phase, we replace each *identifier* by a random word, with consistent replacement across one program. Although scrambling is not required in the test phase, to get insight into how scrambling helped in the classification, Figure 2 shows the LIME explanation for an original test source code and its scrambled version. We can see now that the names of the functions do not carry any weight in the classification. Instead, variables which carry some semantics regarding the algorithm implemented, receive more importance scores. In case of selection sort, it is the variable that holds the index of the minimum element - regardless of the name of this variable.

Table 4 shows the results of CodeBERT for the three datasets where training was done with scrambled source code. We still obtain very high precision and recall for sorting and searching, with accuracy of 97.9% and 99.1% respectively. The accuracy for shortest paths is 90%, which is still good given the small size of the dataset. Cross validation was not performed due to computational time and resource constraints, however for reporting results the data samples were shuffled multiple times inorder to maintain generality.

Figure 3 shows the confusion matrix for sorting, for both CodeBERT and GraphCodeBERT. The most confusion is between selection sort and bubble sort. We conjecture that this is because both algorithms involve swapping.

The confusion matrices for searching and shortest paths also similarly had large diagonal values. A relatively higher confusion was seen in binary search labeled as linear search compared to the other way around (11 out of 890 binary search codes were labeled as linear search, using GraphCodeBert, while all 771 linear search codes were labeled correctly). In case of shortest paths, there was negligible confusion.

| | CodeBERT | | | GraphCodeBERT | | |
|---|---|---|---|---|---|---|
| **Dataset** | **A %** | **P %** | **R %** | **A %** | **P %** | **R %** |
| Sorting | 97.66 | 98.10 | 97.70 | 98.29 | 98.51 | 98.37 |
| Searching | 99.10 | 99.05 | 99.15 | 99.34 | 99.30 | 99.38 |
| Shortest Path | 90.34 | 90.38 | 90.33 | 96.55 | 96.64 | 96.54 |

**Table 4: Accuracy (A), Precision (P), Recall (R) Scores for for CodeBERT and GraphCodeBERT when model is trained with scrambled source code.**

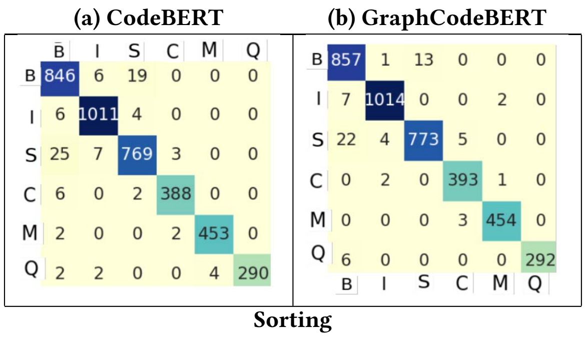
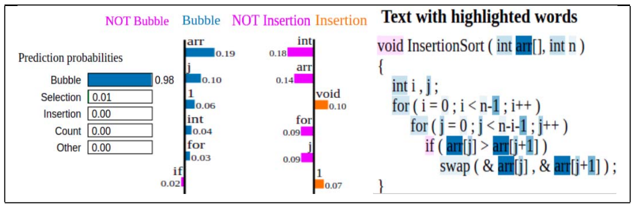


**Figure 3: 6x6 Confusion Matrix of Sorting Test Dataset using (a) CodeBERT, (b) GraphCodeBERT. (In the figure: B/I/S/C/M/Q represent (Bubble Sort/Insertion Sort/Selection Sort/Counting Sort/Merge Sort/QuickSort)**

4.3.1 *Model Insights using LIME - Scrambled Tokens Dataset.* We were again curious about what are the input features which contribute to a prediction. This section describes some insights gained using LIME.

Figure 4 shows the output for the same bubble sort program whose method name was changed to InsertionSort. We can see now that the method name gets no importance at all, and the program is correctly classified as bubblesort.

To understand the working of the model further, we identified a set of programs known to be implementing bubblesort, on which



```
void InsertionSort ( int arr[], int n )
{
  int i , j ;
  for ( i = 0 ; i < n-1 ; i++ )
     for ( j = 0 ; j < n-i-1 ; j++ )
        if ( arr[j] > arr[j+1] )
           swap ( & arr[j] , & arr[j+1] ) ;
}
```

**Figure 4: Bubble Sort, deliberately named as InsertionSort, correctly classified as Bubble Sort, after adding scrambling to the pre-processing steps. Code taken from [4].**

we first ran MOSS, to get their pair wise similarity scores. Subsequently, we selected those which had a low similarity score on MOSS, indicating that there is no "obvious" similarity between them. This allowed us to study features based on which the CodeBERT based model is able to recognize algorithms in source codes that are implementing the same algorithm, but are still "dissimilar". We did the same for insertion and selection sort.

In Figure 5, we present the LIME output for three examples each from this set for bubble sort, insertion sort, and selection sort. The third example for each sort is of a python program. We find that even when the model is trained on C/C++ programs, it does well on classifying python programs. There are various patterns that the model seems to be picking up for identifying an algorithm.

For bubblesort, the figure shows that the array to be sorted is a major contributor for the bubblesort prediction. We conjecture that this is because in bubblesort, the array is referenced more often than in other sorts because of the swapping operation. In the next example, the "flag" variable gets high attribution because in many bubble sort codes, a flag is set if there is any swapping. We also see the importance of the variable "temp" - required for swapping.

In the insertion sort examples we see that the "while" token contributes significantly because the search for the position to insert the next element in is typically done with a while loop. Finally, the selection sort examples show the importance of the identifier which holds the position of the minimum element.

In case of the searching algorithm examples (Figure 6), the signals that mark a linear search are the "if" token, the element to be searched, the variable holding the array length. A binary search seems to be indicated by an "if" *with* an "else" condition, and the "middle" index used to divide the array.

## 4.4 Data-Constrained Modeling

The amount of fine-tuning data for our algorithm identification task is typically expected to be much smaller than is available for other code comprehension tasks such as auto-completion. For such data-constrained tasks, it might be beneficial to freeze some layers of the CodeBERT transfomer encoder. Thus we studied the impact of dataset size, on the performance of our method, while freezing some layers (all, 3, 6, or 9 layers ) of the transformer encoder. The F1 score [16] plots for sorting and searching algorithm datasets are shown in the Figure 7.

The results demonstrate that layer freezing is not helping improve the model for any sized dataset. However, results with as low as only 50% of the data, and upto 6 layers frozen are almost equal to the results using all the data and no layers frozen. These results are encouraging for small-sized dataset regimes.

## 4.5 Improved Results with GraphCodeBERT

Figure 8 shows the example, of CodeBERT explained by LIME, where the source code contains the functions for both selection sort and bubble sort. The main function does not even call selection sort, but the program gets labeled as selection sort. This is a clear example of CodeBERT misclassifying code because it considers only source code tokens. E.g. here, the *min_idx*, i.e. the variable that carries the index of the minimum element gets more weight than the tokens that favour bubble sort.

This kind of "code obfuscation" is again relevant in the scenario of graded programming assignments - students may try certain algorithms coded in the form of functions which they never call, as they could not get them working, but not delete that code from their assignment submissions. Thus, it is important to correctly identify such obfuscated code.

We tested a similar example on GraphCodeBERT and found that GraphCodeBERT labels it correctly (Figure 9). Overall, as seen in Table 4, the performance of GraphCodeBERT is superior to CodeBERT on all the test datasets, including the shortest paths dataset.

Figure 10 shows the source code, the LIME text highlighting and scoring, and the prediction probabilities made by GraphCodeBERT on the source code shown. This source code is again an example scenario where a student may write functions for two sorting algorithms but call only one. Here even though bubblesort is not called at all, GraphCodeBERT gives it a prediction probability of 0.5, along with countsort. This result was more than a little surprising to us, given that GraphCodeBERT is supposed to take into account the graphical structure of the program. However, this result indicates that even GraphCodeBERT is influenced a lot more by the code tokens, and less by the flow graph that it takes as input.

To handle this problem, we added one more step to our code pre-processing pipeline. We now pass the source code through a *dead function code elimination* step, in the pre-processing phase, before the identifier token scrambling step. Figure 10 now shows the same source code, with "un-called" functions eliminated from the source code, which is then correctly classified as "bubblesort" by our algorithm.

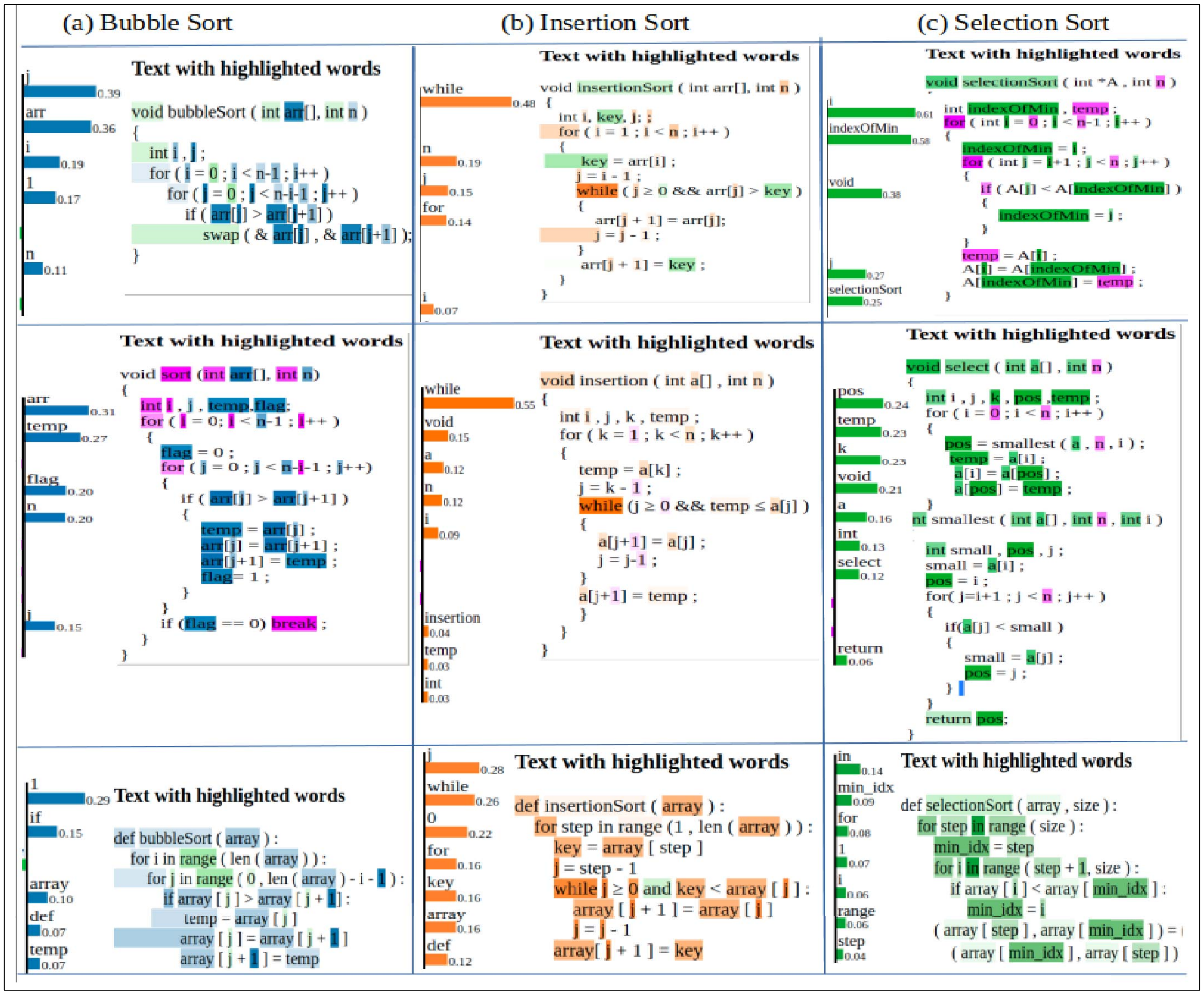
## 4.6 Model Limitations and Threats to Validity

The main limitation of the method is that it will not work for algorithms for which a sizable labeled dataset does not exist - the model is tuned separately for each algorithm type (e.g. here we have three models, one each for sorting, searching and path finding). The other limitation includes various forms of deliberate obfuscation. The following are further threats to validity of our results:

- Large proportion of mistakes in the labeling of the dataset: The labeling of the dataset is simply assumed to be correct, trusting that submissions that are open to the public on the website, would be implementing the correct algorithm.
- Student submissions for some reason look very different from competition submissions.
- Dataset only from one website: the underlying probability distribution of data if sampled from diverse sources may be different; e.g. if the data set contains a disproportionately large number of obfuscated code files.

## 4.7 Epoch vs Accuracy

We studied the accuracy improvement with increasing number of epochs (for the sorting dataset), so that we run the training for desired number of epochs, but not too many more, so as to avoid overfitting. Figure 11 shows that the accuracy of the CodeBERT model saturates early, in about 8 epochs, while GraphCodeBERT needs more than 16 epochs to converge. This is expected as GraphCodeBERT makes an attempt to learn the data flow information



```
void bubbleSort ( int arr[], int n )
{
    int i , j ;
    for ( i = 0 ; i < n-1 ; i++ )
        for ( j = 0 ; j < n-i-1 ; j++ )
            if ( arr[j] > arr[j+1] )
                swap ( & arr[j] , & arr[j+1] );
}
```

```
void insertionSort ( int arr[], int n )
{
  int i, key, j; ;
  for ( i = 1 ; i < n ; i++ )
  {
    key = arr[i] ;
    j = i - 1 ;
    while ( j ≥ 0 && arr[j] > key )
    {
      arr[j + 1] = arr[j];
      j = j - 1 ;
    }
    arr[j + 1] = key ;
  }
}
```

```
void selectionSort ( int *A , int n )
int indexOfMin , temp ;
for ( int i = 0 ; i < n-1 ; i++ )
{
    indexOfMin = i ;
    for ( int j = i+1 ; j < n ; j++ )
    {
        if ( A[j] < A[indexOfMin] )
        {
            indexOfMin = j ;
        }
    }
    temp = A[i] ;
    A[i] = A[indexOfMin] ;
    A[indexOfMin] = temp ;
}
```

```
void sort (int arr[], int n)
{
   int i , j , temp,flag;
   for ( i = 0; i < n-1 ; i++ )
   {
      flag = 0 ;
      for ( j = 0 ; j < n-i-1 ; j++)
      {
         if ( arr[j] > arr[j+1] )
         {
            temp = arr[j] ;
            arr[j] = arr[j+1] ;
            arr[j+1] = temp ;
            flag= 1 ;
         }
      }
      if (flag == 0) break ;
   }
}
```

```
void insertion ( int a[] , int n )
{
   int i , j , k , temp ;
   for ( k = 1 ; k < n ; k++ )
   {
      temp = a[k] ;
      j = k - 1 ;
      while (j ≥ 0 && temp ≤ a[j] )
      {
         a[j+1] = a[j] ;
         j = j-1 ;
      }
      a[j+1] = temp ;
   }
}
```

```
void select ( int a[] , int n )
{
   int i , j , k , pos ,temp ;
   for ( i = 0 ; i < n ; i++ )
   {
      pos = smallest ( a , n , i ) ;
      temp = a[i] ;
      a[i] = a[pos] ;
      a[pos] = temp ;
   }
nt smallest ( int a[] , int n , int i )

   int small , pos , j ;
   small = a[i] ;
   pos = i ;
   for( j=i+1 ; j < n ; j++ )
   {
      if(a[j] < small )
      {
         small = a[j] ;
         pos = j ;
      }
   }
   return pos;
}
```

```
def bubbleSort ( array ) :
    for i in range ( len ( array ) ) :
        for j in range ( 0 , len ( array ) - i - 1 ) :
            if array [ j ] > array [ j + 1] :
                temp = array [ j ]
                array [ j ] = array [ j + 1 ]
                array [ j + 1 ] = temp
```

```
def insertionSort ( array ) :
    for step in range (1 , len ( array ) ) :
        key = array [ step ]
        j = step - 1
        while j ≥ 0 and key < array [ j ] :
            array [ j + 1 ] = array [ j ]
            j = j - 1
        array[ j + 1 ] = key
```

```
def selectionSort ( array , size ) :
    for step in range ( size ) :
        min_idx = step
        for i in range ( step + 1, size ) :
            if array [ i ] < array [ min_idx ] :
                min_idx = i
        ( array [ step ] , array [ min_idx ] ) = (
            ( array [ min_idx ] , array [ step ] )
```

**Figure 5: Feature Insights for each type of sorting algorithm, LIME results are shown for three source code examples: two in C, one in python. Codes taken (a-top) [4], (b-top) [5], (a-bottom) [11], (b-bottom) [12], (c-bottom) [13] and rest from [2]**

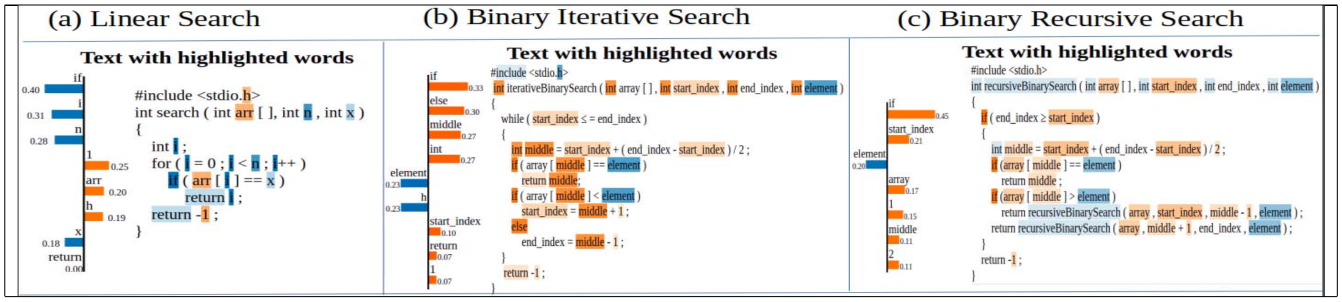
of source code in addition to the code tokens. Also, CodeBERT accuracy starts decreasing after 8 epochs, due to overfitting.

## 5 RELATED WORK

The problem of intelligent autograding of programming assignments that goes beyond output checking, has garnered significant attention in the past decade. The 'intelligent' tasks that such approaches have considered include subjective feedback and partial grading for non-working code. The GradeIt autograding system [29] gives friendlier syntax error messages and also applies some *rewrite rules* to *repair* small problems with the submitted code and then grade it. However, tasks such as feedback on code whose *logic* does not work, or grading submitted programs on subjective aspects of "quality", or some advanced conceptual specifications, require *code comprehension* that need AI/ML techniques.

There has been some work in applying ML techniques to autograding. Shashank *et. al.* [37, 38] use a machine learning approach to grade programs according to a *rubric*. They define a set of features that correspond to the quality metrics given in the rubric, and train a general, 'question-independent' regression model based on these features on a dataset graded by experts.

Piech *et. al.* [30] propose an approach for feedback, in which a neural network is used for creating embeddings such that a program's post condition space is a linear map of its pre-condition space. These linear maps serve as the feature space which is used to propagate instructor feedback to students. Kaleeswaran *et. al.* [22]

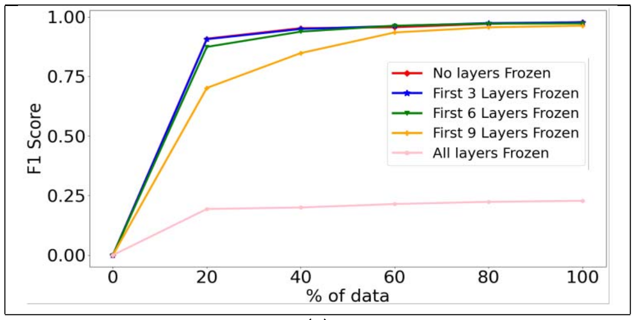


```
#include <stdio.h>
int search ( int arr [ ], int n , int x )
{
    int i ;
    for ( i = 0 ; i < n ; i++ )
        if ( arr [ i ] == x )
            return i ;
    return -1 ;
}
```

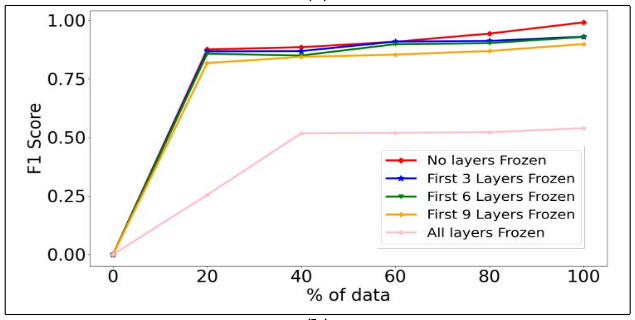
```
#include <stdio.h>
int iterativeBinarySearch ( int array [ ] , int start_index , int end_index , int element )
{
  while ( start_index ≤ = end_index )
  {
    int middle = start_index + ( end_index - start_index ) / 2 ;
    if ( array [ middle ] == element )
      return middle;
    if ( array [ middle ] < element )
      start_index = middle + 1 ;
    else
      end_index = middle - 1 ;
  }
  return -1 ;
}
```

```
#include <stdio.h>
int recursiveBinarySearch ( int array [ ] , int start_index , int end_index , int element )
{
  if ( end_index ≥ start_index )
  {
    int middle = start_index + ( end_index - start_index ) / 2 ;
    if (array [ middle ] == element )
      return middle ;
    if (array [ middle ] > element )
      return recursiveBinarySearch ( array , start_index , middle - 1 , element ) ;
    return recursiveBinarySearch ( array , middle + 1 , end_index , element ) ;
  }
  return -1 ;
}
```

**Figure 6: Feature Insights for Searching Algorithms. Code (a) taken from [6], (b) and (c) from [2]**



**(a)**



**(b)**

**Figure 7: F1 Score vs % of data used with layer freezing for Sorting (a) and Searching (b).**

use a semi-supervised approach based on clustering of student submissions to give feedback to students.

There has also been work outside of the autograding domain in classifying source codes. These use more conventional machine learning models with feature engineering [26, 40], and cluster or classify source codes into different functionalities - which is a different problem than the one considered here.

Shalaby et al [35] present an approach for classifying the source code either pairwise (e.g. greedy vs dynamic programming), or a multi-class labeling e.g. identifying a graph program as doing DFS, vs shortest paths, or using trees. They use features such as number of loops and nested loops and other such constructs. Their best result is only with pairs classification (90% accuracy), while multi-class labeling accuracy for various scenarios remains in the range of 50-80%. Another notable contribution is in [28], where the authors create a CNN model using source code features that do not include variable and function names, thus basing their prediction only on the structure of the source code. They classify source code into advanced algorithm classes such as Combinatorial Optimization Problems vs Number theory problems etc. They get high accuracy (> 90%) in their predictions, and it is likely that their method is resilient to deliberate obfuscation, since they also ignore identifiers. However, it is not clear how their method will extend to “simpler” algorithms carrying out the exact same functionality.

Taherkhani [39] considers precisely the problem of this paper, for the functionality of sorting. But the approach requires determining signature characteristics based on which a decision tree. For this approach to ‘scale’ to general algorithm recognition, such signature characteristics have to be found for all such algorithms.

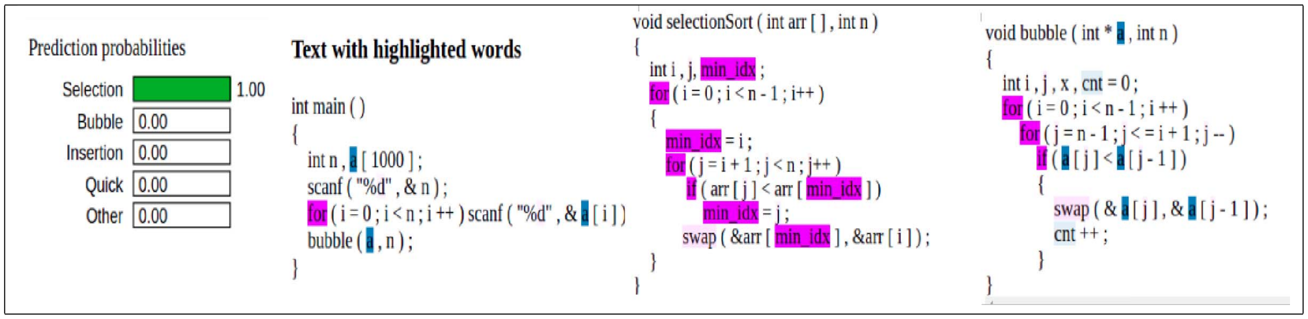
Apart from the above (which is specific to sorting), none of the earlier works consider our specific problem, of a generalized method of recognising algorithms for the same functionality, and few achieve high accuracy. Furthermore, none of the previous works provide any explanation for their results - thus it is not clear how generalizable they are.

## 6 CONCLUSION

In this paper, we proposed and compared four methods for identifying an algorithm, among a known set of algorithms for a given function, in source code known to be implementing that function. We studied the performance of our methods on datasets for three functions: sorting, searching and shortest paths. Accuracy results for the datasets we tested on are excellent. We also used interpretability models to obtain insights of what patterns in the code lead to certain classification outputs on the algorithms.

Our algorithm identification solution addresses an important aspect of grading of intermediate level programming assignments, where sometimes the assignment requires that the student code implements the specified algorithm and not some alternative one. Our solution using GraphCodeBERT is not only robust to misleading identifier names, but also to camouflaging efforts such as burying the actual algorithm used under code fragments (function definitions) that are not called or not relevant to the program flow.

However, we find that the labeling largely fails when the number of lines in code is “large" - i.e. there is code irrelevant to the actual algorithm. Thus here we need to first determine where in the source code to focus the model’s attention, and then make the identification

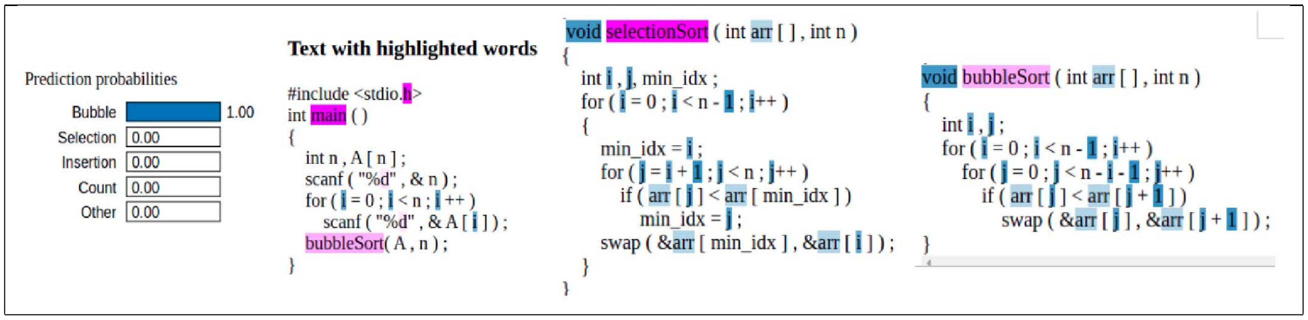


```
int main ( )
{
  int n , a [ 1000 ] ;
  scanf ( "%d" , & n ) ;
  for ( i = 0 ; i < n ; i ++ ) scanf ( "%d" , & a [ i ] )
  bubble ( a , n ) ;
}
```

```
void selectionSort ( int arr [ ] , int n )
{
  int i , j, min_idx ;
  for ( i = 0 ; i < n - 1 ; i++ )
  {
    min_idx = i ;
    for ( j = i + 1 ; j < n ; j++ )
      if ( arr [ j ] < arr [ min_idx ] )
        min_idx = j ;
    swap ( &arr [ min_idx ] , &arr [ i ] ) ;
  }
}
```

```
void bubble ( int * a , int n )
{
  int i , j , x , cnt = 0 ;
  for ( i = 0 ; i < n - 1 ; i ++ )
    for ( j = n - 1 ; j < = i + 1 ; j -- )
      if ( a [ j ] < a [ j - 1 ] )
      {
        swap ( & a [ j ] , & a [ j - 1 ] ) ;
        cnt ++ ;
      }
}
```

**Figure 8: Mislabeled Example by CodeBERT due to obfuscated code. Code taken and modified from [2]**

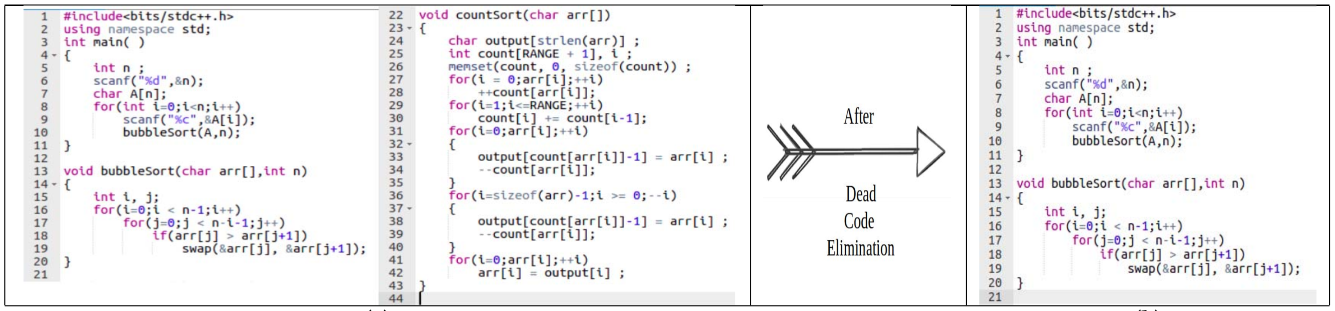


```
#include <stdio.h>
int main ( )
{
  int n , A [ n ] ;
  scanf ( "%d" , & n ) ;
  for ( i = 0 ; i < n ; i ++ )
    scanf ( "%d" , & A [ i ] ) ;
  bubbleSort( A , n ) ;
}
```

```
void selectionSort ( int arr [ ] , int n )
{
  int i , j, min_idx ;
  for ( i = 0 ; i < n - 1 ; i++ )
  {
    min_idx = i ;
    for ( j = i + 1 ; j < n ; j++ )
      if ( arr [ j ] < arr [ min_idx ] )
        min_idx = j ;
    swap ( &arr [ min_idx ] , &arr [ i ] ) ;
  }
}
```

```
void bubbleSort ( int arr [ ] , int n )
{
  int i , j ;
  for ( i = 0 ; i < n - 1 ; i++ )
    for ( j = 0 ; j < n - i - 1 ; j++ )
      if ( arr [ j ] < arr [ j + 1 ] )
        swap ( &arr [ j ] , &arr [ j + 1 ] ) ;
}
```

**Figure 9: Correctly labeled Example by GraphCodeBERT with obfuscation. Code taken and modified from [4] and [2]**



(a)

```
#include<bits/stdc++.h>
using namespace std;
int main( )
{
    int n ;
    scanf("%d",&n);
    char A[n];
    for(int i=0;i<n;i++)
        scanf("%c",&A[i]);
        bubbleSort(A,n);
}

void bubbleSort(char arr[],int n)
{
    int i, j;
    for(i=0;i < n-1;i++)
        for(j=0;j < n-i-1;j++)
            if(arr[j] > arr[j+1])
                swap(&arr[j], &arr[j+1]);
}

void countSort(char arr[])
{
    char output[strlen(arr)] ;
    int count[RANGE + 1], i ;
    memset(count, 0, sizeof(count)) ;
    for(i = 0;arr[i];++i)
        ++count[arr[i]];
    for(i=1;i<=RANGE;++i)
        count[i] += count[i-1];
    for(i=0;arr[i];++i)
    {
        output[count[arr[i]]-1] = arr[i] ;
        --count[arr[i]];
    }
    for(i=sizeof(arr)-1;i >= 0;--i)
    {
        output[count[arr[i]]-1] = arr[i] ;
        --count[arr[i]];
    }
    for(i=0;arr[i];++i)
        arr[i] = output[i] ;
}
```

(b)

```
#include<bits/stdc++.h>
using namespace std;
int main( )
{
    int n ;
    scanf("%d",&n);
    char A[n];
    for(int i=0;i<n;i++)
        scanf("%c",&A[i]);
        bubbleSort(A,n);
}

void bubbleSort(char arr[],int n)
{
    int i, j;
    for(i=0;i < n-1;i++)
        for(j=0;j < n-i-1;j++)
            if(arr[j] > arr[j+1])
                swap(&arr[j], &arr[j+1]);
}
```

**Figure 10: (a) Example of Obfuscated code not correctly identified by GraphCodeBERT and (b) Same Example after elimination of uncalled function code, now correctly identified by GraphCodeBERT. Code taken and modified from [4] and [2]**

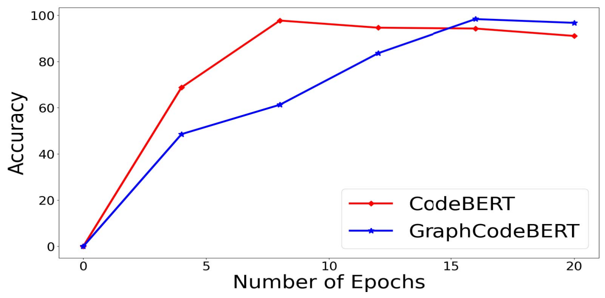


**Figure 11: Epoch vs Accuracy**

work. Thus if obfuscation is present not only due to unreachable function definitions, but just lines of code littered *inside* relevant functions, this is much harder to circumvent. We plan to address this problem in our future work.

Further future work includes widening of the scope of this work in the programming education domain to identifying classes of algorithms used (e.g recursive vs iterative), programming styles, paradigms (functional vs object oriented) and so on.

## REFERENCES

[1] Code Analysis. Joern Tool. https://joern.io.
[2] Dataset. Aizu Online Judge. http://judge.u-aizu.ac.jp/.
[3] Dependencies Graph. Metacpan Making Graph Easy. https://metacpan.org/pod/Graph::Easy.

[4] GeeksForGeeks. Bubble sort. https://www.geeksforgeeks.org/bubble-sort/.
[5] GeeksForGeeks. Insertion sort. https://www.geeksforgeeks.org/insertion-sort/.
[6] GeeksForGeeks. Linear Search. https://www.geeksforgeeks.org/linear-search/.
[7] GeeksForGeeks. Selection sort. https://www.geeksforgeeks.org/selection-sort/.
[8] Google. Precision and Recall. https://developers.google.com/machine-learning/crash-course/classification/precision-and-recall.
[9] Graph Kernel. GraKel repository. https://ysig.github.io/GraKeL/0.1a8/.
[10] Huggingface. Open source CodeBERT model. https://huggingface.co/microsoft/codebert-base.
[11] Programiz. Bubble sort. https://www.programiz.com/dsa/bubble-sort.
[12] Programiz. Insertion sort. https://www.programiz.com/dsa/insertion-sort.
[13] Programiz. Selection sort. https://www.programiz.com/dsa/selection-sort.
[14] Repository. CodeBERT and GraphCodeBERT. https://github.com/microsoft/CodeBERT/tree/master/GraphCodeBERT.
[15] Repository. Tree Sitter. https://tree-sitter.github.io/tree-sitter/.
[16] Towards Data Science. F1 Score. https://towardsdatascience.com/the-f1-score-bec2bbc38aa6.
[17] Transformer Intercept. Model for Explainability. https://github.com/cdpierse/transformers-interpret.
[18] Tree Kernel. Moschitti. http://disi.unitn.it/moschitti/Tree-Kernel.htm.
[19] Pranshu Chourasia and Suraj Kumar. 2022. Code and Dataset Repository for ICPC 2022. https://drive.google.com/drive/folders/1tWKL4c75zKAcKoVqKry8y5jxx-7yyAvC.
[20] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. CodeBERT: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).
[21] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2020. GraphCodeBERT: Pre-training Code Representations with Data Flow. *CoRR* (2020). https://arxiv.org/abs/2009.08366
[22] Shalini Kaleeswaran, Anirudh Santhiar, Aditya Kanade, and Sumit Gulwani. 2016. Semi-supervised verified feedback generation. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering.*
[23] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. 2021. Code prediction by feeding trees to transformers. In *IEEE/ACM 43rd International Conference on Software Engineering (ICSE).*
[24] Nils M Kriege, Fredrik D Johansson, and Christopher Morris. 2020. A survey on graph kernels. *Applied Network Science* 5, 1 (2020).
[25] Jian Li, Yue Wang, Michael R Lyu, and Irwin King. 2017. Code completion with neural attention and pointer networks. *arXiv preprint arXiv:1711.09573* (2017).
[26] Jonathan I. Maletic and Naveen Valluri. 1999. Automatic software clustering via Latent Semantic Analysis. *14th IEEE International Conference on Automated Software Engineering* (1999), 251–254.
[27] Alessandro Moschitti. 2006. Making tree kernels practical for natural language learning. In *11th conference of the European Chapter of the Association for Computational Linguistics.*
[28] Hiroki Ohashi and Yutaka Watanobe. 2019. Convolutional Neural Network for Classification of Source Codes. *2019 IEEE 13th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)* (2019), 194–200.
[29] Sagar Parihar, Ziyaan Dadachanji, Praveen Kumar Singh, Rajdeep Das, Amey Karkare, and Arnab Bhattacharya. 2017. Automatic grading and feedback using program repair for introductory programming courses. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education.*
[30] Chris Piech, Jonathan Huang, Andy Nguyen, Mike Phulsuksombati, Mehran Sahami, and Leonidas Guibas. 2015. Learning program embeddings to propagate feedback on student code. In *International conference on machine Learning.* PMLR.
[31] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *Proc. of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation.*
[32] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. " Why should i trust you?" Explaining the predictions of any classifier. In *Proc. of the 22nd ACM SIGKDD.*
[33] Vitaly Romanov, Vladimir Ivanov, and Giancarlo Succi. 2020. Approaches for Representing Software as Graphs for Machine Learning Applications. In *2020 International Computer Symposium (ICS).*
[34] Saul Schleimer, Daniel S Wilkerson, and Alex Aiken. 2003. Winnowing: local algorithms for document fingerprinting. In *Proc. of ACM SIGMOD.*
[35] Maged Shalaby, Tarek Mehrez, Amr Mougy, Khalid Abdulnasser, and Aysha Al-Safty. 2017. Automatic Algorithm Recognition of Source-Code Using Machine Learning. 170–177. https://doi.org/10.1109/ICMLA.2017.00033
[36] Giannis Siglidis, Giannis Nikolentzos, Stratis Limnios, Christos Giatsidis, Konstantinos Skianis, and Michalis Vazirgiannis. 2020. GraKeL: A Graph Kernel Library in Python. *J. Mach. Learn. Res.* 21 (2020).
[37] Gursimran Singh, Shashank Srikant, and Varun Aggarwal. 2016. Question independent grading using machine learning: The case of computer program grading. In *Proc. of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.*
[38] Shashank Srikant and Varun Aggarwal. 2014. A system to grade computer programming skills using machine learning. In *Proc. of the 20th ACM SIGKDD.*
[39] Ahmad Taherkhani. 2010. Recognizing Sorting Algorithms with the C4.5 Decision Tree Classifier. *2010 IEEE 18th International Conference on Program Comprehension* (2010), 72–75.
[40] Secil Ugurel, Robert Krovetz, C. Lee Giles, David M. Pennock, Eric J. Glover, and Hongyuan Zha. 2002. What's the code? Automatic classification of source code archives. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, D. Hand, D. Keim, and R. Ng (Eds.). 632–638.
[41] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems.* 5998–6008.