Applied Deep Code Search for Algorithm Implementation using Pseudocode

by

Sai Charishma Valluri

A Creative Component submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
Wei Le, Major Professor
Qi Li

The student author, whose presentation of the scholarship herein was approved by the program of study committee, is solely responsible for the content of this dissertation/thesis. The Graduate College will ensure this dissertation/thesis is globally accessible and will not permit alterations after a degree is conferred.

Iowa State University

Ames, Iowa

2022

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

LIST OF LISTINGS

Page

ACKNOWLEDGMENTS

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research and the writing of this non-thesis report. First and foremost, I would like to express my sincere gratitude to my major professor, Dr. Wei Le, for her continuous support throughout my research, for her patience, motivation, and immense knowledge. Her guidance helped me in all the times of research and writing of this non-thesis report. Besides my advisor, I would like to thank my committee member, Dr. Qi Li, for her encouragement, insightful comments and suggestions.

I would also like to thank my lab mates, Adithya Kulkarni, for his continuous support throughout the experiment process and for his innovative inputs that helped me resolve any roadblocks, Mohna Chakraborty, for her help in selecting the baseline and initial clarifications, Yonas Sium, for his support throughout the project.

ABSTRACT

Code retrieval tools and techniques play a key role in facilitating the software developers to retrieve code snippets from open-source projects given a natural language query. With natural language description as an input, the code search tool searches for the most relevant code snippets amongst the code. In this creative component, we train the state-of-the-art code search tool, Deep Code Search [1], with the dataset of algorithms and study the algorithm implementation in the result with the pseudo code as an input query. We trained the Deep Code Search for three different programming language settings, Java, C, and a combination of Java and C. Our results show that Deep Code Search can identify the algorithm implementation in different languages for the given pseudo code.

CHAPTER 1. INTRODUCTION

To implement a functionality or a coding problem, software developers can implement the code independently or search for code online. With the existence of repositories like GitHub, GitLab, and BitBucket, hosting millions of open source projects, there are opportunities to satisfy the search needed by the developers for functionality implementations, programming issues, etc. Previous studies have even revealed that more than 60% of developers search for source code every day [2], [3].

Many code search approaches have been proposed, focusing on Information Retrieval and Natural Language techniques. This creative component focuses on a code search tool called DeepCS [1] that uses a novel deep neural network called CODEnn (Code-Description Embedding Neural Network). Instead of matching the text similarity, CODEnn embeds code snippets and natural language descriptions into a high-dimensional vector space, so that code snippets and their corresponding description have similar vectors.

The approach to extending Deep Code Search to establish pseudo-code and source code for algorithm implementation consists of two phases, offline training and search phase.

Offline training phase: In the offline training phase, the CODEnn model takes a training corpus that contains code elements and the corresponding descriptions, i.e., the method name, API sequence, tokens, description tuples. The architecture of the CODEnn model consists of three modules.

Code embedding module: It deals with three aspects of the source code, method name, API sequence, and tokens of the method body. Each aspect is embedded individually and then combined into a single vector representing the entire code.

Description embedding module: It embeds the natural language descriptions of the source code into vectors.

Similarity module: It measures the cosine similarity between the vectors of the source code and its respective description. The higher the similarity, the more related the code is to the description.

Each code snippet in the training corpus is jointly embedded with its natural language description into a high-dimensional vector space.

Search phase: Search phase deals with how the search functionality works in the *DeepCS*. When a user enters a natural language query, DeepCS returns the relevant code snippets through the trained CODEnn model. Before a search starts, DeepCS embeds all the code snippets in the search codebase into vectors using the CoNN module of the offline training phase in an offline manner. During the online search, when a developer enters a natural language query, It embeds the query into a vector using the trained DeNN module of the offline training phase and estimates the cosine similarities between the query vector and all code vectors. Finally, the top K code snippets, whose vectors are most similar to the query vector, are returned as the search results.

CHAPTER 2. APPROACH

2.1 Overview

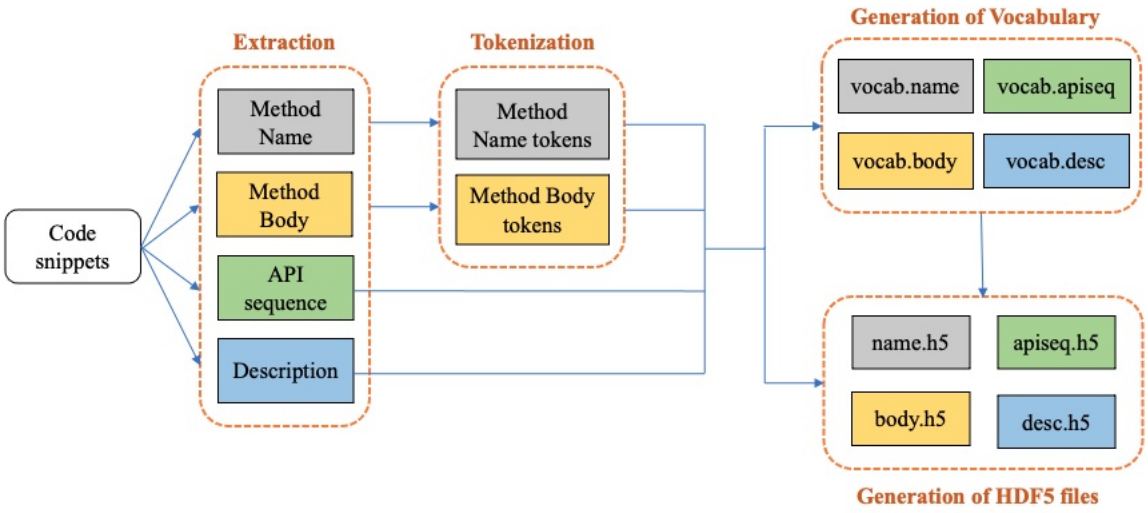

Figure 2.1 Overview

We extract the method name, method body, API sequence, and description from each code snippet of our source code. In detail, the method name and method body are extracted from each code snippet. API sequence is generated by running the code given by the authors of Deep Code Search [1]. The description is extracted by going through every method of the file in the dataset and getting comments above the method. Once extraction is completed, the method name and method body are further split into tokens according to the camel case [4]. Then, we generate the respective vocabulary and HDF5 file for each code element. Once the required files are generated, we replace the files from the Pytorch version of Deep Code Search's GitHub repository and follow the instructions in their README file to train the Deep Code Search model.

2.2 Extraction

Method Name: For the given code snippet, we extract the method name from the method's signature. In detail, we look for the word that exists before the '(' braces in the method's signature and extract it. Because as per Oracle's Java Documentation [5], every method's signature contains the method name before its parameters list.

For example, let's consider the code snippet in Figure 2.2. Its method's signature is '*public static Calendar toCalendar(final Date date)*', and from the signature, as per the Java method Documentation, *toCalendar* is the method name for this code snippet. So, we extract the word *toCalendar* as the method name for this code snippet.

Method Body: To extract the method body from the given code snippet, we look for the first occurrence of '{' braces in the snippet. Once we found the '{' braces, we consider the words/lines from that point to the end of the code snippet. If we consider the code snippet in Figure 2.2, the words starting from '{' in line 1 to the '}' in line 6 are considered as the method body for the code snippet.

In such a way, once the method body of the code snippet is extracted, we remove the stop keywords, Java keywords, symbols, operators, and variable names from the method body. In the Figure 2.2, 'final', 'return' are the Java keywords, '{' , '}' , ';' , '.' , '(' , ')' are the symbols in the method body, '=' is the operator, and '*c*' is the variable in the method body. So, all these keywords and symbols are removed from the method body. The leftover words, Calendar, getInstance, setTime, and date are considered as the method body tokens.

API Sequence: To extract the API Sequence from the code snippet, we used the code provided by one of the Deep Code Search authors. The code takes the method body as an input and checks for the constructor calls. If it finds one, it stores the name of the class into an array and checks for any method calls using the respective class object's name in the rest of the code snippet. If it finds any such method calls, it appends the method name along with its class name into the same list where we initially store the class name. After it checks all the lines of the code

snippet, it outputs the API sequence stored in it, priority-wise, by following the rules described in the Deep Code Search paper [1].

Let's consider the code snippet in Figure 2.2. In the first line of its method body, it contains the constructor call of the class *Calendar* along with its method *getInstance*. So, in this case, the code appends *Calendar.getInstance* into its list and checks for any method calls in the rest of the code using its class object *c*. In the second line of the method body, we observe that the class object *c* calls a method called *setTime*, so the code adds *Calendar.setTime* into its list and checks for the remaining lines of the method body. In the case of the code snippet in Figure 2.2, it doesn't have any other method calls or constructor calls, so the code returns *Calendar.getInstance* and *Calendar.setTime* as the output.

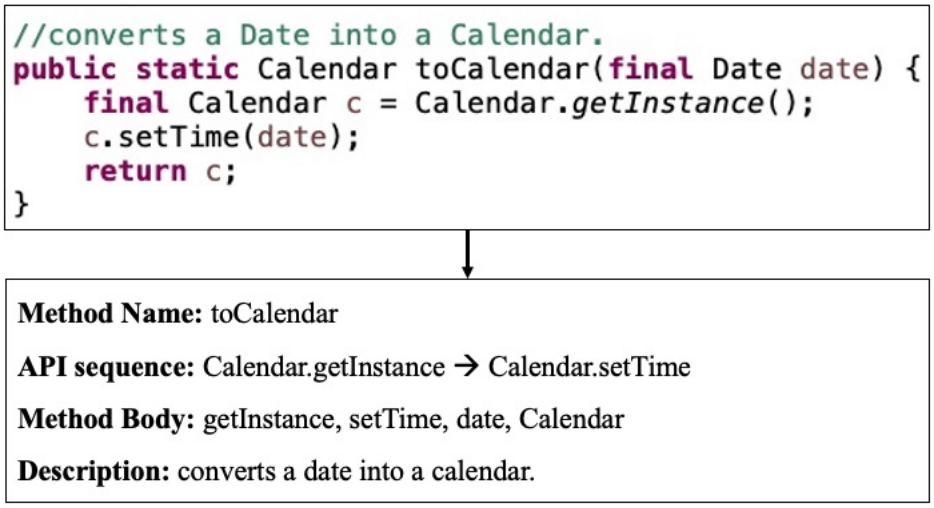

Figure 2.2 Extraction of code elements from a Java Code Snippet

Description: To extract the description from the code snippet, we considered any lines that exists before the method and starts with '\ ∗ ∗' (Java-doc comments) or '\\' (in-line comments). The authors in Deep Code Search [1] considered only the Java-doc comments since all their code snippets contain the description/summary of the methods in Java-doc comments. But, in our case, in some of our code snippets, we observed that the description/summary of the method is

mentioned in in-line comments. So to extract the description, we also considered in-line comments along with Java-doc comments.

If we consider the code snippet in the Figure 2.2, it contains the summary of the method in in-line comments. So, we consider *converts a Date into a Calendar.* as the description of this code snippet.

2.3 Tokenization

The tokenization process is to split the given word based on its Camel case [4]. The Deep Code Search [1] tokenizes only the method name and method body of its code snippets. Since we aim to replicate the same procedure of DCS, we also apply the tokenization to the method name and method body.

For any word, if we encounter an uppercase character inside it, we split that particular word at the uppercase character, resulting in two words/tokens. In this way, we tokenize all the words of the code snippet's method name and method body. Figure 2.3 shows an example of tokenizing method name and method body. In the example, method name is *toCalendar*, since it contains an uppercase character between it, we split the word into two tokens/words, *to* and *calendar*. In the similar way, from the example, method body has the words, *getInstance* and *setTime* with uppercase characters in them, so these two words are split into four words, *get*, *instance*, *set*, and *time*.

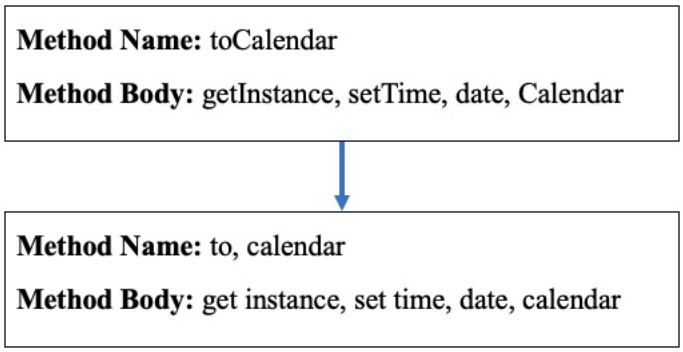

Figure 2.3 Tokenizing Method name and Method body

2.4 Generation of Vocabulary

After the tokenization process, we generate four vocabulary files for the four code element files. For each unique word in the code element file, we assign a value to it with 0 as the initial value and increment by one for each word. Figure 2.4 shows a sample method body file after tokenization. The vocabulary generated for the file will be '{ calendar: 0, get: 1, instance: 2, set: 3, ..., trip: 18, a2: 19 }'. Even though the file has 21 words in it, our vocabulary ends with the number 19 because of the repetition of the words *get*, *test* in lines 3 and 4. By the end of this phase, we will have four vocabulary files in JSON format for the four code elements, namely, method name, method body, API sequence, and description.

```
1  calendar get instance set time date
2  header binary test load
3  amino acid arg standard get size gly
4  test round trip a2
```

Figure 2.4 Example of a Method Body file

2.5 Generation of HDF5 Files

We generate an HDF5 file for each of the four code elements of the code snippet. The HDF5 file structure consists of two datasets, Indices, and phrases. The indices dataset consists of two columns, length and position. The length column in the indices dataset refers to the number of words in a particular line, and the position column refers to the current position of the respective line in the file. For the first line, the position value will be 0, and for the next lines, the position value will be the sum of its preceding line's length and position value. The phrases dataset of the HDF5 file consists of values of all the words from their respective vocabulary file.

An example of indices dataset of an HDF5 file for a sample Method body file in Figure 2.4 is shown in the Figure 2.5. The first line in the Figure 2.4 contains six words, so the length and position for the first line are 6 and 0, respectively. For the second line, its length is 4, and the

position will be the sum of the length and position of the first line, which is 6 in this case. Similarly, the length and position for the third line will be 7 and 10 (sum of 4 and 6 from the second line), and the length and position for the fourth line will be 4 and 17, respectively. The phrases dataset of an HDF5 file for the same example would be [0, 1, 2, 3, ..., 18, 19] because the word '*calendar*' has the value 0 in its vocabulary from the Section 2.4, word '*get*' has value 1, word '*instance*' has value 2 and so on. We calculate the Indices and Phrases dataset for each file and create an HDF5 file.

Indices

Length	Position
6	0
4	6
7	10
4	17

Figure 2.5 Hdf5 file generation for a sample method body

2.6 Deep Code Search model

To train the Deep Code Search model, We use the PyTorch version of the Deep Code Search from their GitHub repository [6]. The existing data files from the repository are replaced with the file of our dataset generated in Section 2.4 and Section 2.5. Once the model is trained, we give the pseudo-code of an algorithm as an input query to find its algorithm implementation in the resultant code snippets.

2.6.1 Dataset preparation

For training the Deep Code Search model, after generating the vocabulary files, we divide each code element file (i.e., method name, method body, API sequence, description files) into two files, train and valid, with 75% and 25% of the original file respectively. After generating training

and validation files, we generate HDF5 files for both of them. In this way, we generate the training and validation HDF5 files for all four code elements.

```
#training data
    'train_name':'train.name.h5',
    'train_api':'train.apiseq.h5',
    'train_tokens':'train.tokens.h5',
    'train_desc':'train.desc.h5',
#test data
    'valid_name':'valid.name.h5',
    'valid_api':'valid.apiseq.h5',
    'valid_tokens':'valid.tokens.h5',
    'valid_desc':'valid.desc.h5',
#vocabulary info
    'vocab_name':'vocab.name.json',
    'vocab_api':'vocab.apiseq.json',
    'vocab_tokens':'vocab.tokens.json',
    'vocab_desc':'vocab.desc.json',
```

Figure 2.6 Data for the Deep Code Search model

Figure 2.6 shows the data files that are generated for training the Deep Code Search model.

2.6.2 Training the model

We used the PyTorch version of the Deep Code Search's GitHub Repository to train the model for our project. In the data folder of the PyTorch, we replace the existing files of the model with the respective files that are generated in the Section 2.6.1, and use the same parameters that are used in the Deep Code Search [1] since we aim to replicate the work of the Deep Code Search for algorithm implementation.

Initially, we started training the model for 10 epochs, and we observed that after epoch 3, the loss value of the model started increasing. The loss value for epoch 2 is 0.38, and the loss value for epoch 3 is 0.36. Until epoch 3, the loss value is continuously decreasing, but for epoch 4, the loss value is 0.39 and started increasing with epochs. So to prevent over-fitting, we stopped training the model at epoch 3 and used the model generated for the search process.

2.6.3 Search

Deep Code Search generates a model for each epoch. Since we got the least loss value for epoch 3, we used the model generated at epoch 3 as our Deep Code Search model for searching the pseudo-codes.

Before the online search starts, *DeepCS* embeds all the code snippets in the search codebase into vectors using the trained model in an offline manner, and stores all the code vectors in a file. When we enter the pseudo-code as a query, *DeepCS* generates a query vector and checks the k nearest code vectors in the search codebase.

Figure 2.7 shows an example of how the *DeepCS's* search works with a sample input. It takes the Input query and number of results in a text entry format and gives the relevant code snippets as an output with its match value. For example, in the Figure 2.7, the input query is *convert an inputstream to a string* and the number of results is *5*. The resultant five code snippets are marked with a number in the figure followed by their matched value.

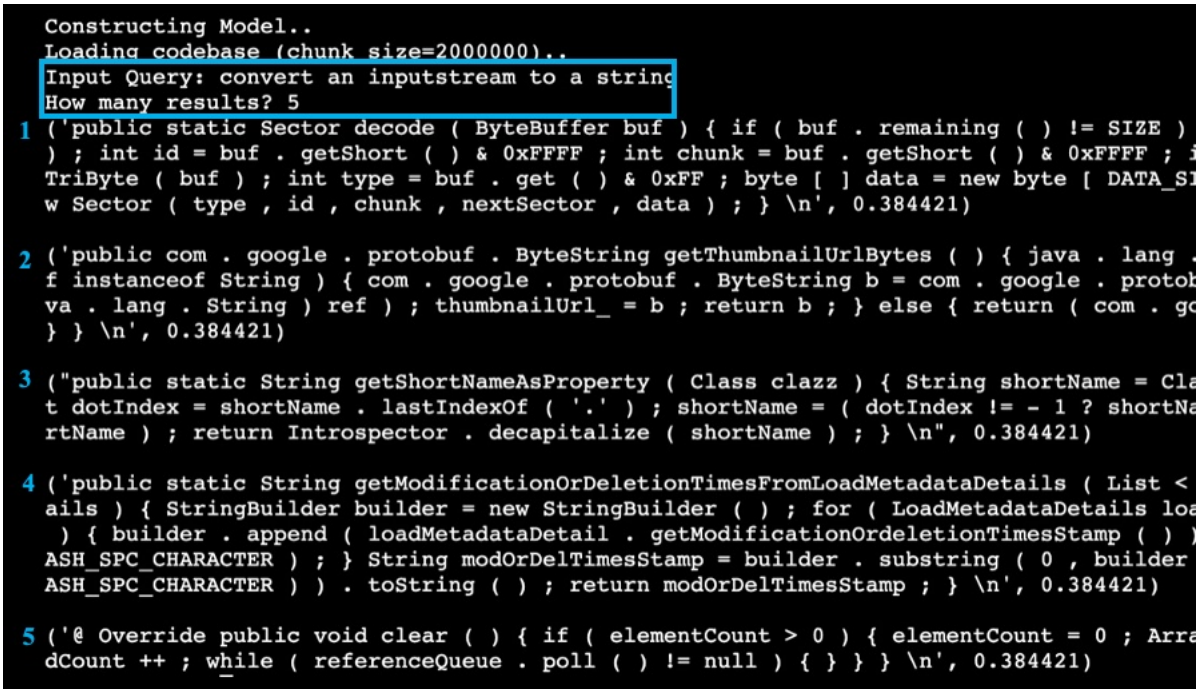

Figure 2.7 Search in Deep Code Search

CHAPTER 3. EXPERIMENTS AND RESULTS

3.1 Experiment setup

We collected the pseudo code from *Introduction to Algorithms* [7] and *Algorithms Design Manual* [8], a total of 103 algorithms. We used the *Stony Brook Algorithm Repository* [9] to construct the source code database. The reason for choosing this repository is because it provides the ground truth and documents the algorithms(s) implemented in the project. However, it did not provide the location of the algorithms in the code. This repository contains 67 C and 27 Java real-world open-source projects, among which 51 C and 22 Java projects are used. These projects implemented 28 algorithms out of 103 algorithms we collected.

We conducted the experiments in three settings. To evaluate whether *DeepCS* can handle different languages, we experimented each setting with three languages: C, Java, and a mixture of C and Java, where we searched C/Java/ mixed languages using C/Java/mixed languages trained CODEnn models, respectively. All the experiments are done in google colaboratory.

The first experiment uses the project known to implement the queried pseudo code of the algorithm as the search codebase. This experiment aims to determine how effectively the *DeepCS* can find the algorithms in one project. Figure 3.1 represents the first setting of the experiment. Our source code contains 51 C and 22 Java projects. For the first experiment, we use each project at once as a search codebase and give the pseudo-code of the algorithm implemented in that project as an input query. So, we repeat this experiment 51 times by changing the search codebase for the C language. Similarly, for Java and mixed language as well.

The second experiment uses all the *Stony Brook Algorithm Repository* projects as a search codebase. Since we have 51 C and 22 Java projects in our search code, we use all the 51 C projects as one search codebase and query the pseudo-codes of the algorithms implemented in all these 51 C projects as the input. We do the same for Java and mixed languages. This experiment

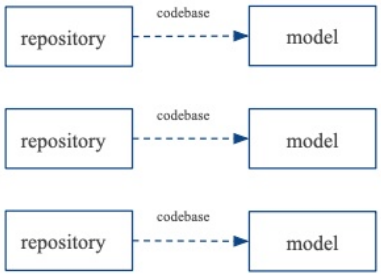

Figure 3.1 Single Repository

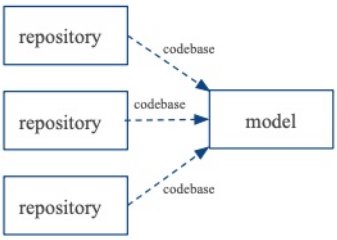

Figure 3.2 Multiple Repositories

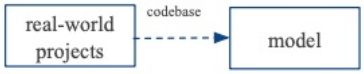

Figure 3.3 Real-world Projects

aims to determine how effectively the *DeepCS* can find an algorithm among multiple projects. Figure 3.2 represents the second setting of the experiment.

Finally, for the third experiment, we collected 10 C and 10 Java popular projects from the GitHub repository with the most stars. The third experiment evaluates whether *DeepCS* can effectively find the algorithm implementations in practice where we don't know the ground truth. Figure 3.3 represents the third setting of the experiment. We use the 10 C projects as one search codebase and query with the pseudo-codes of the algorithms implemented in them. We experiment likewise for Java and mixed language.

3.2 Results

Table 3.1, Table 3.2 and Table 3.3 shows the results of C, Java and mixed languages respectively. Each table lists the results of three experiments under one repository, multiple repositories, and real-world projects. Our experiments were done on 17 C and 28 Java Algorithms and mixed language with 17 algorithms that have both C and Java implementations.

Table 3.1 Results on C Language

Search Codebase	Top 1	Top 10	Top 25
One repository	3	7	11
Multiple repositories	0	3	9
Real-world projects	0	0	1

Table 3.2 Results on Java Language

Search Codebase	Top 1	Top 10	Top 25
One repository	0	6	17
Multiple repositories	0	2	10
Real-world projects	0	6	14

Table 3.1, Table 3.2 and Table 3.3 shows that among the top 25 output list, under one repository, *DeepCS* found the known algorithms for 9 of 17 C projects, 17 of 28 Java projects, and 11 of 17 mixed-language projects. Among the top 10 output list, *DeepCS* found 9 of 17 C projects, 10 of 28 Java projects, and 4 of 17 mixed languages projects under Multiple repositories. The results for Real-world projects, *DeepCS* was able to find 1 of 17 for C projects, 14 of 28 for Java projects, and 1 of 17 for mixed-language projects. The performance of *DeepCS* is best on Java projects, followed by C projects. We also observed that we could get better results for C even though the Deep Code Search authors haven't practically implemented the *DeepCS* for C language or mixed language.

3.3 Examples of Results

The results of the experiments are manually annotated, so we may make mistakes when confirming the results. When annotating, we confirmed algorithm implementation for a

Table 3.3 Results on Mixed languages

Search Codebase	Top 1	Top 10	Top 25
One repository	1	6	9
Multiple repositories	1	2	4
Real-world projects	0	1	1

pseudo-code, if the conditions and loops match with the pseudo-code, and variables and mathematical operators in the code match with the pseudo-code. Since the implementation may not completely follow the pseudo code, we also tried to understand the code to determine if the algorithm is implemented. We now show examples of code search results that demonstrate the advantages of DeepCS.

In the Listing 3.1 and Listing 3.2, we show the LCS algorithm's pseudo-code and its algorithm implementation, ranked 1, when experimented with the first setting in C Language. The listings show that the implementation is the same irrespective of the variables. In Listing 3.1, lines 2-4 initialize the new variables of a particular length and new arrays, which are implemented in lines 1-5 in algorithm implementation 3.2. Similarly, lines 5-18 in Listing 3.1 is implemented by lines 6-24 in its implementation 3.2. We observe that, for this example, instead of matching the text similarity, *DeepCS* was able to output the code snippets relevant to the logical implementation of the query because *DeepCS* retrieves the code snippets related to a natural language query according to their vectors by its embedding technique. Apart from the logical implementation, we also observe that '*nwarrow*' in line 13 of the pseudo-code is the same as '*NEITHER*' in line 19 of the algorithm implementation, '*uparrow*' in line 16 of the pseudo-code is the same as '*UP*' in line 22 of the algorithm implementation, and '*leftarrow*' in line 18 of the pseudo-code is same as '*LEFT*' in line 25 of the algorithm implementation. We observe that, for this example, *DeepCS* focused on recognizing the semantically related words and handled irrelevant/noisy keywords in the query because of its embedding technique for the query and code snippets and the technique of retrieving the relevant code snippets for a query.

```
1 LCS-LENGTH(X,Y)
2   m = X.length
3   n = Y.length
4   let b[1..m,1..n] and c[0..m,0..n] be new tables
5   for i = 1 to m
6     c[i,0] = 0
7   for j = 0 to n
8     c[0,j] = 0
```

```
for i = 1 to m
  for j = 1 to n
    if x_i == y_j
      c[i,j] = c[i-1,j-1] + 1
      b[i,j] = "\nwarrow"
    else if c[i-1,j] \geq c[i,j-1]
                                c[i,j] = c[i-1,j]
                                b[i,j] = "\uparrow"
        else if c[i,j] = c[i,j-1]
                               b[i,j] = "\leftarrow"
return c and b
```

Listing 3.1 Pseudo code for Longest common subsequence algorithm

```
  int n = a.length();
  int m = b.length();
  int S[][] = new int[n+1][m+1];
  int R[][] = new int[n+1][m+1];
  int ii, jj;
  for (ii <= n) for init for update
      S[ii][0] = 0;
      R[ii][0] = UP;
  for (jj <= m) for init for update
      S[0][jj] = 0;
      R[0][jj] = LEFT;
  for (ii <= n) for init for update
      for (jj <= m) for init for update
          if (a.charAt(ii-1) == b.charAt(jj-1)) {
              S[ii][jj] = S[ii-1][jj-1] + 1;
              R[ii][jj] = UP_AND_LEFT; }
          else{
              S[ii][jj] = S[ii-1][jj-1] + 0;
              R[ii][jj] = NEITHER; }
          if (S[ii-1][jj] >= S[ii][jj]) {
```

```
21                  S[ii][jj] = S[ii-1][jj];
22                  R[ii][jj] = UP; }
23              if (S[ii][jj-1] >= S[ii][jj]) {
24                  S[ii][jj] = S[ii][jj-1];
25                  R[ii][jj] = LEFT; }
26          ....
27      ....
28      return new String(lcs)
```

Listing 3.2 Algorithm implementation for LCS in mixed language

In the Listings 3.3 and 3.4, we show an example of *Floyd Warshall* Algorithm and its implementation, ranked 12, when experimented with the first setting in Java Language. The listings show that the pseudo-code and its implementation are not the same, but *DeepCS* was able to understand the meaning of the query. We observed that in Listing 3.4, lines 3-5 initialize the matrix *sum* similar to the line 3 in its pseudo code 3.3, and lines 7-12 in Listing 3.4 implements lines 4-7 in its pseudo code 3.3. For this example, we see that *DeepCS* was able to successfully return the code snippets with the same logical implementation because of embedding code snippets (from the search codebase) and the input query into vectors. From the vectors, it can estimate the distance, identify their semantic relation and return the code snippets nearer to the query vector based on the distance. This process helps in handling irrelevant/noisy keywords.

```
1 FLOYD-WARSHALL(W)
2     n = W.rows
3     D=W
4     for k = 1 to n
5       for i = 1 to n
6         for j =1  to n
7          d_ij = min(d_ij,d_ik+d_kj)
8     return D
```

Listing 3.3 Pseudo code for Floyd Warshall Algorithm

```
...
final int[][] sums = new int[vertices.size()][vertices.size()];
for (int i = 0; i < sums.length; i++) {
    for (int j = 0; j < sums[i].length; j++) {
        sums[i][j] = Integer.MAX_VALUE;}}
...
for (int k = 0; k < vertices.size(); k++) {
    for (int i = 0; i < vertices.size(); i++) {
        for (int j = 0; j < vertices.size(); j++) {
      ...
          final int summed = (ikCost != Integer.MAX_VALUE && kjCost != Integer.
    MAX_VALUE) ? (ikCost + kjCost) : Integer.MAX_VALUE;
          ...
...
return allShortestPaths;
...
```

Listing 3.4 Algorithm Implementation for Floyd Warshall Algorithm in Java

Another advantage of *DeepCS* is its associative search. In the below example, *DeepCS* not only returned the code snippets with matched keywords but also recommended the semantically relevant ones. This is important because it increases the search scope, especially when the search codebase is small. Listings 3.5 and 3.6, shows the 23rd result of the pseudo-code query, *Breadth First Search* Algorithm in C language. We observed that the pseudo-code doesn't have any keywords head, rear, or visited, but *DeepCS* was able to identify the semantic meaning, i.e., checking if the node is visited and adding them into a queue, if not visited, in the lines 20-23 of algorithm implementation. From the Listing 3.6 we also see that lines 14-23 implement lines 12-21 in its pseudo code.

From the pseudo code and algorithm implementation of breadth-first search, we observe that even though they both don't have the matching keywords or same data structure implementation, they have the same functionality/logic with different data structure implementation. For this

example, we observe that *DeepCS* not only returned the snippets with matching keywords but also recommended results with semantically related ones.

```
BFS(G,S){
    for @each vertex u in G.V - s@{
        $u.color = WHITE$
        $u.d = infty$
        $u.pi = NIL $
    }
    $s.color = GRAY$
    $s.d = 0$
    $s.pi = nil$
    @let Q be an emptyset @
    ENQUEUE(Q,s)
    while $Q != emptyset ${
    u = DEQUEUE(Q)
        for $v in G.Adj[u]${
            if $v.color == WHITE$ {
                $v.color = GRAY$
                $v.d = u.d + 1$
                $v.pi = u$
                ENQUEUE(Q,v)
      }
        $u.color = BLACK$
        }
    }
    }
```

Listing 3.5 Pseudo code for Breadth First Search Algorithm

```
public void BFS() {
int head = 0;
int rear = 0;
int[] queue = new int[mVexs.size()];
boolean[] visited = new boolean[mVexs.size()];
```

```
for (int i = 0; i < mVexs.size(); i++)
    visited[i] = false;
    for (int i = 0; i < mVexs.size(); i++) {
        if (!visited[i]) {
            visited[i] = true;
            queue[rear++] = i;
        }

        while (head != rear) {
            int j = queue[head++];
            ENode node = mVexs.get(j).firstEdge;
            while (node != null) {
                int k = node.ivex;
                if (!visited[k]){
                    visited[k] = true;
                    queue[rear++] = k;
                }
                node = node.nextEdge;
            }
        }
    }
}
```

Listing 3.6 Algorithm Implementation for Breadth First Search in C

CHAPTER 4. SUMMARY AND FUTURE WORK

4.1 Conclusion

This creative component presents the extension of *DeepCS* to establish the mappings between pseudo-code and source code, with C, Java, and mixed languages. This approach consists of four phases: extracting the method name, method body, API sequence, and description from the code snippets, tokenizing the method name and method body based on the Camel case, generating vocabulary and hdf5 files. *DeepCS* is trained on three settings, one repository as a search codebase, multiple repositories as a search codebase, and real-world projects as a search codebase. When searching a project that implements a known algorithm, *DeepCS* found 4 (6%),19 (31%), and 37 (60%) correct implementations out of 62 queries in the top 1, 10, and 25 of the output list, respectively.

4.2 Limitations and Future Work

The Tables, 3.1, 3.2, 3.3 showed that Java outperformed the C and mixed languages settings. The reason might be because the Deep Code Search paper limited its scope to the Java code. Despite the advantages such as associative search, *DeepCS* could still return inaccurate results. It sometimes ranks partially relevant results higher than the exact matching results. It might be because *DeepCS* ranks results by just considering their semantic vectors. In future work, more code features could be considered to further adjust the results, and modifying the architecture can be considered to implement the model for multiple languages.

BIBLIOGRAPHY

[1] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. Deep code search. In *Proceedings of the 2018 40th International Conference on Software Engineering (ICSE 2018)*. ACM, 2018.

[2] Raphael Hoffmann, James Fogarty, and Daniel SWeld. Assieme:findingand leveraging implicit references in a web search interface for programmers. In *In Proceedings of the 20th ACM Symposium on User Interface Software and Technology (UIST)*. ACM, 2007.

[3] Kathryn T.Stolee, Sebastian Elbaum, and Daniel Dobos. Solvingthesearch for source code. In *ACM Transactions on Software Engineering and Methodology (TOSEM)*. ACM, 2014.

[4] Wikipedia. Camel case - https://en.wikipedia.org/wiki/camelcase.

[5] Method's signature format - https://docs.oracle.com/javase/tutorial/java/javaoo/methods.html.

[6] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. Deep code search - https://github.com/guxd/deep-code-search, 2018.

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[8] Steven S Skiena. *The algorithm design manual*, volume 2. Springer, 1998.

[9] Stony brook algorithm repository http://algorist.com/algorist.html.