Graph Embedding based Code Search in Software Project

Yanzhen Zou
Key Laboratory of High Confidence Software Technologies, Ministry of Education, Beijing, China
School of Electronics Engineering and Computer Science, Peking University, Beijing, China
zouyz@pku.edu.cn

Chunyang Ling
Key Laboratory of High Confidence Software Technologies, Ministry of Education, Beijing, China
School of Electronics Engineering and Computer Science, Peking University, Beijing, China
lingcy@pku.edu.cn

Zeqi Lin
Key Laboratory of High Confidence Software Technologies, Ministry of Education, Beijing, China
School of Electronics Engineering and Computer Science, Peking University, Beijing, China
linzeqi@pku.edu.cn

Bing Xie
Key Laboratory of High Confidence Software Technologies, Ministry of Education, Beijing, China
School of Electronics Engineering and Computer Science, Peking University, Beijing, China
xiebing@pku.edu.cn

ABSTRACT

Source code search is one of the most important methods to study and reuse software project. Currently, natural language based code search mainly faces the following two challenges: 1) More accurate search results are required when software projects evolve to be more heterogeneous and complex. 2) The semantic relationships between code elements (classes, methods, etc.) need to be illustrated so that developers could better understand their usage scenarios. To deal with these issues, we propose a novel approach to searching a software project's source code based on graph embedding. First, we build a software project's code graph automatically from its source code and represent each code element in the code graph with graph embedding. Second, we search code graph with natural language questions, return corresponding subgraph that composed of relevant code elements and their associated relationships, as the best answer of the search question. In experiments, we select two famous open source projects, *Apache Lucene* and *POI*, as examples to perform source code search tasks. The experimental results show that our approach improves F1-score by 10% than existing shortest path based code graph search approach, while reduces average response time about 60 times.

CCS CONCEPTS

•Software and its engineering → Software creation and management → Software development techniques → Reusability; Information systems → Information retrieval → Specialized information retrieval;

Internetware'18, September, 2018, Beijing, China

https://doi.org/10.1145/3275219.3275221

KEYWORDS

Software reuse; Code search; Code graph; Graph embedding

ACM Reference format:

Yanzhen Zou, Chunyang Ling, Zeqi Lin, Bing Xie. 2018. Graph Embedding based Code Search in Software Project. In *The Tenth Asia-Pacific Symposium on Internetware (Internetware'18), September 16, 2018, Beijing, China, 10 pages.* https://doi.org/10.1145/3275219.3275221

1 INTRODUCTION

Software source code is an important research issue in software engineering. The most common practices for developers during software reuse process are to search source code, find relevant code elements (classes, methods, etc.) and invoke target APIs (Application Programming Interfaces). As general search engines (e.g. Google) perform very inefficiently in retrieving software code, natural language based source code search has been wildly studied in recent years [1][2][3]. Many existing code search tools, like Krugle [4], Ohloh [5] and Sourcerer [6], just process source code as text. The main idea behind them is to utilize text similarity between question/query and source code to find target code elements. Typical similarity calculation models include Boolean Model, Vector Space Model, BM25 Model, etc. [7] However, the accuracy of existing code search tools is usually unsatisfactory as software projects evolve to be more and more heterogeneous and complex. The studies of F.Lv et al. showed that only 25.7%-38.4% of the top 10 results returned by Ohloh is useful [8].

Some researchers pointed out that a key problem of these code search tools is that they could not "understand" natural language question. To deal with this problem, many approaches have been proposed, such as optimizing retrieval process with online documents (e.g. MSDN) [8], expanding semantic similar words [9], query reformulation [10], and so on. Although these methods can improve the accuracy of searching results to some extent, they need accumulate and utilize lots of auxiliary information. Under

the circumstance of lack of auxiliary information on which they depend, they cannot achieve the expected results.

On the other hand, these code search tools only return the most relevant code elements as search result, without illustrating the semantic relationships among them. It is very inconvenient for developers to understand the code element's implementation logic and usage scenario, and reduces the efficiency of software reuse. For this purpose, some researchers pointed out that we should build directed graph of source code (called code graph) and carry code search on the code graph. At the same time, some graph based shallow semantic analysis techniques, such as PageRank [11] and shortest path [12], have been utilized to improve the search accuracy. Searching on the large graph needs heavy computation while interactive code search requires very short delay, which makes the efficiency of the graph search algorithm vital. The latest research on code graph search proposed shortest path based approach, but it usually causes quite long delay. W.K. Chan et al. also pointed out this problem in their paper and used some approximation algorithm to optimize it [12].

In this paper, we propose a novel approach to searching a software project's code graph based on graph embedding. Graph embedding is a kind of representation learning technique, which could effectively represent deep structural information of software code graph. At the same time, graph-embedding process can be finished offline, which reduces the time complexity of online computation to the constant level. As a result, the efficiency of code graph search algorithm can be highly improved.

Compared with existing work, the main contributions of this paper include:

1) We propose an approach to representing a software project's source code with graph embedding. It automatically constructs a software project's code graph and represents each code elements through graph embedding.

2) We propose an approach to searching a software project's code graph based on graph embedding. It can return the corresponding connected subgraph for a natural language question, while the accuracy and efficiency of code search are improved.

3) We implement a graph embedding based code search tool and conduct experiments on two well-known software projects. Compared with the approach of W.K. Chan et al., our approach improves the recall and F1-score of search results, while reduces response time at the same time.

The rest of the paper is organized as follows. Section 2 elaborates our approach. Section 3 presents experiments and results. Section 4 discusses our related work. Finally, Section 5 concludes this paper.

2 OUR APPROACH

The main idea behind our approach is that source code contains abundant structure information, which is very helpful for question understanding and code search. We use an example to illustrate our motivation further. Suppose a developer is now reusing the open source project *Apache Lucene*, he would like to use the *WildcardQuery* class to process the query with wildcards. At this moment, the requirement changes and he need to process the query with regex expressions. He may prefer to use the *RegexQuery* class and refactor the code. However, is it the best way? As shown in Figure 1, we can find a subgraph of *Lucene*'s source code, where both of the two classes *WildcardQuery* and *RegexQuery* are inherited from the superclass *MultiTermQuery*, and they all have *getTerms* method. Therefore, a better strategy for the developer is to use the abstract superclass as the variable type and assigns it different subclass as value.

Motivated by this example, we aim to propose a new approach to searching a software project's code graph effectively. We use graph-embedding technique to represent and utilize structural information in source code, so as to improve the accuracy and efficiency of source code search. As is shown in Figure 2, our approach is composed of the following three components:

(1) Building project code graph and embedding. This component is to parse software source code and build code graph automatically. Then it uses graph-embedding technique to mine the graph and represent nodes/concepts.

(2) Parsing question according to code graph. This component is to preprocess natural language question, match key words of question to candidate nodes in code graph and measure their weight.

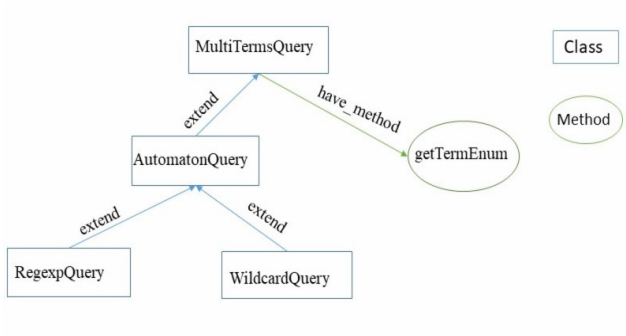

Figure 1: Part of Lucene code graph

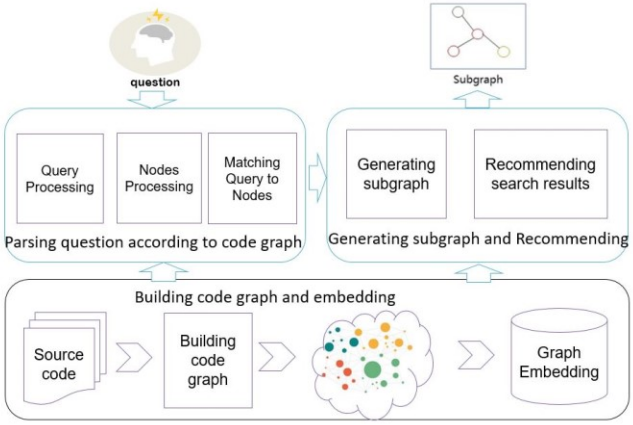

Figure 2: Framework of our approach

(3) Generating subgraph and recommending search results. This component is to construct and return a connected subgraph as search results. Different recommending methods are used here to generate the best answer.

2.1 Building Project Code Graph and Embedding

In a software project, there are mainly two kinds of code elements: one is *class* or *interface*, the other is *method*. The relationships among code elements are generally consistent in different object-oriented programming languages. This paper takes Java as experimental object and considers 6 kinds of relationships among code elements, including the Inherit *(Inh)* relationship between class/interface and class/interface, the Implement (*Imp*) relationship between class and interface, and so on. Table 1 lists all the relationships we mined and used in our work, defined as Rel. We extract all the code elements and their related relationships from source code and build a software project's code graph automatically.

Table 1: Relationships in our code graph

Relation Name	Node type
Inherit(Inh)	Class/Interface A- Class/Interface B
Implement(Imp)	Class A – Interface B
Member(Mem)	Method A- Class/Interface B
Parameter(Par)	Method A- Class/Interface B
Return(Ret)	Method A- Class/Interface B
Call(Call)	Method A-Method B

Project Code Graph. A software project's code graph $G = (V_G, E_G)$ is an unweighted directed graph. Vertex set V_G includes all nodes composed of classes, interfaces and methods. For each edge $e(u,v) \in E_G$, $\exists uRv \in Rel$, where uRv means node u and node v have a relationship R.

There are many source code parsers for Java, such as CheckStyle [13], JAVAssist [14], Yasca [15], JDT [16]. We use Eclipse JDT, a lightweight code-parsing tool, to build the DOM structure based abstract syntax tree. Then we use graph database Neo4j to store our project code graph.

Based on a software project's code graph, we apply graph-embedding technique to represent each code element (node) so as to facilitate question parsing and subgraph generation later. Since graph embedding can be processed offline, the online distance computation is very convenient and fast. We use the LINE (Large-scale Information Network Embedding) approach proposed by J. Tang [17] to conduct the embedding process of the code graph. This approach is highly scalable for large information networks with considerably low time complexity. It can reserve both the local and global structural information via the first order and second order approximation function. The source code of C++ can be found on Github[1] and we re-implement it by Java.

[1] https://github.com/tangjianpku/LINE

Distance between nodes. After graph embedding, each node $v \in V_G$ is mapped to a real value vector $r_v \in R^d$, where d is the dimension of the vector space. The distance between two nodes is defined as the Euclid distance, i.e. $dist(u,v) = \| r_u - r_v \|_2$.

2.2 Parsing Question according to Code Graph

This component consists of two sub-processes. First, it preprocesses the question to obtain the bag of words and generate candidate nodes via some matching rules. Second, it evaluates the text similarity between candidate nodes and question as the candidates' weight.

2.2.1 Generating Candidate Nodes. We use the bag of words model (BOW) to represent natural language questions, which ignores the sequence order and syntax information. After tokenizing the question, we remove those stopwords including common English stopwords, Java reserve words and project specific function words (e.g. Lucene).

For the code graph nodes (classes or methods), we split the name (identifier) of a node by Camel Case to obtain BOWs. For example, class node *QueryParser* is associated with the words set {query, parser}.

Query. A query Q is composed of a set of words, i.e. $BOW_Q = \{q_1, \cdots, q_n\}$, after tokenization and removing stopwords.

Nodes words set. Each node v in G is associated with a words set $BOW_v = \{s_1, \cdots, s_m\}$.

Candidate nodes set. A query Q is matched to a candidate nodes set $Candidate_Q = \{C_1, \cdots, C_n\}$, which is set family and $C_i (1 \le i \le n)$ stand for the nodes set matched to word q_i. Let $|C_i|$ be the size of set C_i.

For each word q in the query, we apply the following text matching rules to generate its candidate nodes:

1) Full name matching. The word q is a full name of class or method, e.g. q = *QueryParser*.

2) Partial name matching. The word q is contained by a node's BOW, e.g. q = *parser*.

3) Stemming matching. If the word q has the same stem as one of the words in a nodes' BOW, the node would be matched as a candidate. Stemming is a common practice in natural language processing since it can extract the same stem from different word forms. For example, *parser* and *parse* have the same stem of *pars*. We use the Snowball toolkit to perform the stemming tasks.

4) Abbreviation matching. If a node's BOW contains the abbreviation of the word q, it would be a candidate. Naming conventions of code elements are sometimes different from natural language and abbreviation is wildly used for convenience. We hand craft some abbreviation rules such as *Document* vs. *Doc*, *Number* vs. *Num*, etc.

5) Similar words matching. If a node's BOW contains similar words with word q, it would be a candidate. Similar word is still an important factor in question understanding and should be taken into consideration during the matching process. For example, if the word q in query is *remove*, but the method name in the source code is *delete*, so that the pure text matching would

fail. In this paper, we user the pre-trained glove word vector on Wikipedia to solve the problem. The similarity between word u and word v is calculated as formula 1. If the cosine similarity of two word vectors exceeds the threshold, we believe that they are similar words.

$$Similarity(u,v) = \frac{\boldsymbol{w}_u \cdot \boldsymbol{w}_v}{\| \boldsymbol{w}_u \| \| \boldsymbol{w}_v \|} \quad (1)$$

2.2.2 Measuring Candidates Weight. In order to cover adequate information of the query, we generate a lot of candidate nodes through the former step. However, not all the nodes have the same quality and some nodes would introduce the annoying noises. So we need to measure the text similarity between nodes and query to give different weight (i.e. importance) to the candidates.

We consider two evaluation indicators. First, more similar words between a node's BOW_v and a query's BOW_Q cause higher text similarity weight. Second, fewer irrelevant words introduced by the node's BOW_v is better. In other words, if two nodes have the same number of similar words with the query, then the node with smaller size of words set is preferred. The calculation formula is defined as follows:

$$score_{relevant}(v) = \frac{|BOW_v \cap BOW_Q|}{|BOW_Q|} \quad (2)$$

$$score_{irrelevant}(v) = \frac{|BOW_v \cap BOW_Q|}{|BOW_v|} \quad (3)$$

Note that $BOW_v \cap BOW_Q$ need to take similar words into accounts. If the word in BOW_v does not occur in BOW_Q, then we use the maximal vector cosine similarity among all the words in query as the contribution to the intersection. So the value $|BOW_v \cap BOW_Q|$ may not be integer but float number.

Node weight. We use the F1-score of $score_{relevant}$ and $score_{relevant}$ as the weight of the node, to measure the text similarity between the node and query.

$$w(v) = \frac{2 * score_{relevant}(v) * score_{irrelevant}(v)}{score_{relevant}(v) + score_{irrelevant}(v)} \quad (4)$$

2.3 Generating and Recommending Subgraph

Given the candidate nodes for the query, this step first selects suitable code nodes to form the subgraph which can reveal the query semantics. While there are so many candidate nodes and the search space is very large, we present a beam search based algorithm to efficiently generate the subgraph. Then it extends the selected nodes to a connected subgraph. Finally, it returns the top result as answer to the user.

2.3.1 Generating and Measuring Subgraph. We measure a subgraph from two aspects: First, the subgraph with higher accumulated text similarity between nodes and query (i.e. the node weight) is a better choice. It is intuitively reasonable because it captures more semantic information from the query. Second, the subgraph with lower distance among code elements is preferred because these code elements are more likely to be used together. In other words, we leverage the structural information of code graph to eliminate node disambiguation. For example, if one node in the subgraph is the class *Document*, and there are two candidate methods with the same name *add* to be selected, their text similarity with question is same. Then we prefer to the method in *Document* class because they have a shorter distance compared with another. Here we give some definitions to facilitate the generation of question related subgraph first:

Vertex set of the subgraph. The vertex set $V' = \{C_{i,j} \mid x_{i,j} = 1\}$ of subgraph G' is selected from $Candidate_Q = \{C_1, \cdots, C_n\}$, where $x_{i,j}$ is a binary variable. The j_{th} node in C_i is selected if $x_{i,j} = 1$, where $\sum_{j=1}^{|C_i|} x_{i,j} \geq 1$, $x_{i,j} = x_{i',j'}$ iff $C_{i,j} = C_{i',j'}$.

The first constraint guarantees that there is at least one node selected from C_i. In other words, every word in the query would have a corresponding node so that the subgraph can cover the query semantics as much as possible. The second constraint is for consistency consideration. If one node appears in multiple candidate sets and is selected in one of the candidate set, it must be selected in other candidate sets as well.

Then we define the evaluation function for a subgraph G' as formula (5). The $dist(u,v)$ is the Euclid distance of two nodes as defined in section 2.1. The generation process of the subgraph can be modeled as a optimization problem of $min\ score(G')$.

$$score(G') = \sum_{u,v \in V', u \neq v} \frac{dist(u,v)}{w(u) * w(v)} \quad (5)$$

Algorithm 1 Generate subgraph by beam search

Input: The candidate nodes set, $\boldsymbol{Candidate_Q} = \{\boldsymbol{C_1}, \cdots, \boldsymbol{C_n}\}$
Output: Top one subgraph

```
1.  C ← ∪_{i=1}^{n} C_i
2.  sort C by node weight in descending order.
3.  Beam ← Top(k, C)
4.  for i ← 2 to n do
5.    foreach V' ∈ Beam do
6.      foreach c ∈ C_i do
7.        delta ← Σ_{v∈V'} dist(c,v) / (w(c)*w(v))
8.        newV' ← V' ∪ {c}
9.        cost ← cost + delta ,update the total cost.
10.       Add(newV', newBeam) , put the results in a new beam.
11.     end
12.   end
13.   sort NewBeam by cost in ascending order.
14.   Beam ← Top(k, newBeam)
15. end
16. return Top(1, Beam)
```

We propose a beam search based algorithm. Beam search can be viewed as a kind of optimization of greedy search algorithm since it reserves top k candidates during each searching stage, where the parameter k stands for the beam size. As is shown in the pseudo code of Algorithm 1, for the input candidate nodes set, it first sorts them by node weight in the descending order and chooses the top k results as initial nodes. In line 7-10, it first calculates the delta cost after adding a new node in subgraph based on formula (5) and then generates a new subgraph and calculates the accumulated cost. Finally, it puts the new subgraph in the candidate results set and enters the next stage. In line 16, it finally returns the top one subgraph as the result.

2.3.2 Extending and Recommending Subgraph. Now we get the vertex set of target subgraph from candidate nodes set. In this process, we use graph embedding to calculate the distance between two nodes, which makes the calculation process faster and more convenient than before. On this basis, we need to we need to connect these vertexes in subgraph and provide a vivid picture for the users. There may be different edges or paths between two vertexes, which one we should select is very important. In this step, we present the approach to extending the vertex set to a connected subgraph.

We define this problem as to construct a *Minimum Spanning Tree* (MST) for the given vertex set V'. The smallest hops between any two nodes are defined as the weight of edges. Solving the MST problem means that we use as fewer edges as possible to connect all the nodes in the given vertex set. The detail is shown in Algorithm 2 as follows.

It is quite similar to the Prim algorithm. In line 5, what the FinidShortestPath function does is to find the shortest path between node set X and Y and return both the path and the end node in Y. The shortest path between every two nodes can be obtained by querying the code graph database with Cypher query language, which is supported by Neo4j. In line 6 and 7, it adds the new node to X and removes it from Y. In line 8 and 9, it adds all the nodes in the path to the vertex set and all the edges to the edge set. Finally, it generates a connected subgraph until all nodes in the set Y is contained in the set X.

Algorithm 2 Extend vertex set to connected subgraph

Input: Vertex set V'
Output: Connected subgraph G^e
1. $V^e \leftarrow V', E^e \leftarrow \emptyset$
2. $X \leftarrow$ **randomly select a node from** V'
3. $Y \leftarrow V' - X$
4. **while** $Y \neq \emptyset$ **do**
5. $v, path \leftarrow$ **FindShortestPath**(X, Y)
6. $X \leftarrow X \cup \{v\}$
7. $Y \leftarrow Y - \{v\}$
8. $V^e \leftarrow V^e \cup path.V$
9. $E^e \leftarrow E^e \cup path.E$
10. **end**
11. $G^e \leftarrow (V^e, E^e)$
12. **return** G^e

3 EXPERIMENTS

Based on above approach, we implement a graph embedding based source code search tool for software project. We also conduct some experiments to assess the effectiveness of our approach. The following two research questions are addressed in our experiments:

RQ1: How effectively can our approach do in natural language based code search? Compared with other work, our approach only uses the structural information in source code, the experiments will focus on the evaluation of our approach under this condition.

RQ2: How much can graph embedding based approach outperform existing code search approach? We think graph embedding is a good way to represent and utilize code structural information, and compare our approach with shortest path based approach on accuracy and time cost in the experiments.

3.1 Experiment Design

We select two well-known open source software projects, *Apache Lucene* and *POI*, as experimental objects. We construct code graph automatically from the source code and conduct code search tasks using natural language queries. The detailed experiment setup is shown as follows.

3.1.1 Constructing Code Graphs. We downloaded the source code of *Apache Lucene* and *POI* from their official sites and selected the wildly used versions. Then our tool constructed the code graph automatically from the source code and embedded it to vector representation. The statistical information about the code graphs is listed in Table 2, including the source code version, number of class nodes, number of method nodes, number of different relationships and total construction time. The experiments were carried out on a Windows 10 desktop with the dual core 3.4GHz CPU and 8G memory. As is shown in Table 2, the code graphs are quite large and complex, the *Lucene's* code graph has more than 39,000 nodes and 255, 880 edges, and *POI's* code graph has more than 35,679 nodes and 211, 692 edges.

Table 2: Code graphs for experiments

Project Name	Version	Class Nodes	Method Nodes	Total Nodes	Relation-ships	Time (min)
Lucene	6. 3. 0	5, 377	34, 042	39,419	255, 880	39
POI	3. 14	3, 678	32, 001	35,679	211, 692	31

3.1.2 Query Formulation. Developers may give different kinds of questions in source code search. There are various ways to generate natural language questions in our experiments, but the reality and validity of questions are important. Therefore, we use the following two objective methods to acquire our code search questions for different software projects.

For POI project, we extract questions from its user guide provided on the official site[40]. There are corresponding code snippets examples for each question in the user guide. On this basis, we label these code elements mentioned in code examples as the ground truth for each question. Considering the number of vertexes and edges, we select 20 questions about POI project as our first group query.

Table 3: Experimental questions about Lucene and their annotated code elements

NO.	Question	Annotated code elements	ENum
1	How to highlight common terms in the Lucene query?	Highlighter, CommonTermsQuery, Query, Terms, Formatter, highlightTerm	6
2	How to set document boost attribute?	Document, Attribute, BoostAttribute, AttributeImpl, BoostAttributeImp. setBoost	5
3	Get string representation of Terms using HighFreqTerms.	HighFreqTerms, Terms, Terms. iterator, HighFreqTerms. getHighFreqTerms, BytesRef. toString	5
4	How is lucene query boost affected by lengthNorm similarity?	Similarity, TFIDFSimilarity, BoostQuery, TFIDFSimilarity. lengthNorm	4
5	How to create a query using wildcards?	Query, MultiTermQuery, AutomatonQuery, WildcardQuery	4
6	How to get the score of top document in lucene?	ScoreDoc, TopDocs, Document, TopDocs.getMaxScore	4
7	How to query document have any term in a set?	TermsQuery, MultiTermQuery, Query, Document	4
8	How to query document having terms with a prefix?	PrefixQuery, Query, Document, Terms	4
9	How to get the full path of a directory?	Directory, FSTUtil.Path, DirectoryTaxonomyReader. getPath, getFullPrefixPaths	4
10	How to merge several taxonomy indexes?	TaxonomyMergeUtils, IndexWriter, TaxonomyMergeUtils. merge	3
11	Sorting on a numeric field in Lucene.	Field, SortField, SortedNumericSortField	3
12	How to escape some characters utilizing QueryParser?	QueryParser, QueryParserBase, QueryParserBase. escape	3
13	How does regexp query work on lucene?	RegexpQuery, automaton RegExp, AutomatonProvider	3
14	How to get field Term Vectors in Lucene?	Field, IndexReader, getTermVectos	3
15	How can a reader read multiple Lucene indexes?	MultiReader, IndexReader, IndexSearcher	3
16	How to implement a fuzzy search query in lucene?	FuzzyQuery, Query, MultiTermQuery	3
17	Count the number of documents in an lucene index.	IndexReader, numDocs, getDocCount	3
18	How to index an integer field in lucene	LegacyIntField, IndexableField, Filed	3
19	How to compute document length in Lucene?	Document, Similarity, computeNorm	3
20	Match all document query in Lucene.	MatchAllDocsQuery, Query, Document	3

For *Lucene* project, since there is not any tutorial or code example on the official site, we extract questions from the QA pairs on Stack Overflow instead. The detailed extraction process is described as follows. First, we attained the questions with the tag of *Lucene* from the dump of Stack Overflow in 2008-2016. Then we filtered the questions with positive votes and accepted answers and the answers contain some code snippets, so that we got a pool of 923 questions after this step. Finally, we randomly sample from this question pool and manually check whether it is a valid code search question. Each question post on Stack Overflow consists of two parts: title and content. In fact, the title of the question can represent the meaning of the whole question as most time so that we use the title to formulate the query. At last, we also selected 20 questions as our second group query. In order to label the related code elements for these queries, we organized 3 postgraduates who are familiar with Lucene project to manually read the Stack Overflow question and answer posts. We use the code snippets appeared in the answer posts as baseline and leverage the source code Javadoc and code graph database to jointly determine the final code element set for the query. Thus, the final code element set may contain other code elements that never appeared in the answer posts.

The questions used in our experiments are all real world valid natural language questions and commonly occurred in the software developing process. In Table 3, we list all the 20 questions about Lucene project that we used in our experiments. We also list the annotated code elements and the number of annotated code elements for each question. We believe that answering these questions can well evaluate our approach in practice for searching software source code.

3.1.3 Comparing Approaches. We *analyzed* the top 1 and top 3 results of our approach in the experiments, and we compared our approach with other two existing code search approaches as well.

- ✧ Our approach. It uses the top one result of our graph embedding based code search approach.
- ✧ Text similarity based approach. It only considers the text similarity between nodes and questions, which means that the measurement function in formula (5) only contains weight term but not the distance term.
- ✧ Shortest path based approach. It uses shortest path to calculate distance between nodes, which is originated from the work of W.K. Chan et al.

3.1.4 Metrics. To *assess* the effectiveness of the code search results, we apply the evaluation metrics of precision, recall and F1-score. Suppose H is the annotated subgraph for a query, and G is the suggested subgraph by the system. Following the definitions in [12], we calculate the metrics in our experiments using the following formulas:

$$P(G) = \frac{tp}{tp + fp} = \frac{|V_G \cap V_H|}{|V_G|} \quad (6)$$

$$R(G) = \frac{tp}{tp + fn} = \frac{|V_G \cap V_H|}{|V_H|} \quad (7)$$

$$F = \frac{2 \cdot P \cdot \mathrm{R}}{P + \mathrm{R}} \quad (8)$$

In the (6~8) formulas, V_G and V_H are the vertex set of G and H respectively, and their intersection stands for the true positives.

Note that we do not use the popular metrics for navigational search such as MRR (Mean Reciprocal Rank) because our search result is not a single segment of code elements or code snippets but a subgraph. Thus, using precision and recall is more appropriate in this circumstance.

In order to compare two methods, we define the gain to evaluate the improvement percentage of method 1 outperforms method 2. In our experiments, the V in formula (9) can be the precision, recall and F1-score.

$$Gain = \frac{V(1) - V(2)}{V(2)} \quad (9)$$

3.2 Experiment Results

For the two groups of queries, we evaluate the average precision, recall and F1-score of different approaches, including our approach, text similarity based approach and shortest path based approach. The experimental results are shown in Table 4 ~6. We take the 1st question in Table 3, i.e. "*how to highlight common terms in the Lucene query?*", as an example to illustrate our work process. First, we preprocess the query to get the words set {highlight, common, terms, query}. For each word in the query, it matches to the code graph nodes using the defined rules in section 2.2.1. Next, we calculate the text similarity as the weight of the nodes. For example, the weight of *CommonTermsQuery* is 0.86 and the weight of *Terms* is 0.4. After this step, we generate the subgraph and extends it, which consists of two sub-processes:

(1) Generating subgraph. It uses the graph embedding vectors to calculate the distance between two nodes, e.g. dist (*Terms, highlightTerm*) = 15.18. Then it selects candidate nodes using the beam search based Algorithm 1 and get the vertex set of the subgraph {*CommonTermsQuery, highlightTerm, Terms*}.

(2) Extending subgraph. It extends the vertex set to a connected subgraph using Algorithm 2. Finally, it returns the top result as is shown in Figure 3.

In Figure 3, the result subgraph is composed of 8 nodes and then we can calculate the corresponding precision, recall and F1-score. These 5 nodes labeled with blue boxes in Figure 3 are the annotated APIs in Table 3. Therefore, the precision equals to 5/8, and the recall equals to 5/6, and the F1-score equals to 5/7.

Here we set the dimension of graph embedding to be 200 in our experiments, and set the beam size to be 8 as a matter of experience.

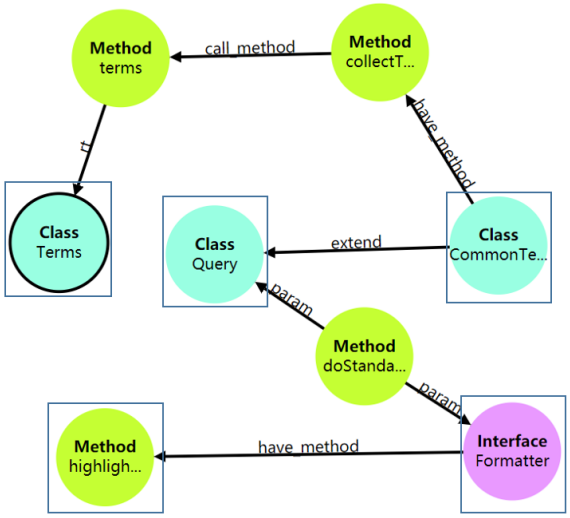

Figure 3: Search result of question "how to highlight common terms in the Lucene query?"

3.2.1 Effectiveness Comparison. The overall experimental results are shown in Table 4. It can be seen that our approach achieves the highest precision, recall and F1-score on both of the *Lucene* and *POI* projects. Text matching based approach has the lowest average precision, recall and F1-score, which reveals that solely using the text similarity between code elements and questions is not satisfying to understand the query semantics. It indicates that our graph embedding based approach can mine the structural information in the source and improve the search result significantly.

Table 4: Experimental results

Approaches	Lucene Project			POI Project		
	Precision	Recall	F1-score	Precision	Recall	F1-score
Our approach	0. 53	0. 85	0. 63	0. 70	0. 81	0. 74
Text similarity based approach	0. 41	0. 57	0. 43	0. 43	0. 62	0. 49
Shortest path based approach	0. 47	0. 62	0. 52	0. 65	0. 66	0. 64

Table 5: Gain evaluation of the experimental results

Approaches	Lucene Project			POI Project		
	Precision gain	Recall gain	F1-score gain	Precision gain	Recall gain	F1-score gain
Text similarity based code search	30.0%	46.2%	45.0%	62.0%	29.4%	51.6%
Shortest path based code search	13.1%	36.7%	22.6%	8.0%	23.1%	17.5%

Table 6: Efficiency Analysis of the experimental results

Approaches	Lucene Project		POI Project	
	Average response time	Longest response time	Average response time	Longest response time
Our approach	1. 39s	3. 42s	3.05s	4.89s
Text similarity approach	0. 13s	0. 80s	0.14s	0. 28s
Shortest path approach	86. 68s	257. 18s	164.28s	284.65s

Shortest path based approach also considers the code graph information during the search process. Its results outperform the pure text similarity matching approach but are still not as effective as out approach. Thus, using graph embedding vectors to calculate distance can capture more useful information than shortest path. We evaluated the gain of our approach with text matching based approach and shortest path based approach, the results are shown in Table 5.

To investigate the efficiency of our approach, we also investigated the average response time and longest response time of different approaches. The experimental results are shown in table 6. We can see that text matching based approach achieves the best efficiency. Its response time is very short because the computation is quite simple. Shortest path based approach's response time is very long as the average response time already exceeds 1 minutes. In practice, it is unacceptable for user interactions. It has to query the shortest path between two nodes online. If the breadth first search (BFS) is applied to find the shortest path, the worst time complexity is O(N), where N is the total number of nodes in the code graph. By contrast, graph embedding process is done offline and it only takes constant time to calculate the distance online so that our approach also improves runtime efficiency to a great extent.

The approach in [12] proposed by W.K. Chan et al. is based on shortest path. In their reported results, the precision is 0.53, recall is 0.56 and F1-score is 0.54. It is quite close to our results although we use different datasets. They extract questions about JSE coding examples from the KodeJava website and re-organize them as text phrases. While we use the domain specific software projects and directly use the natural language as queries.

3.2.2 Detailed Analysis of Precision and Recall. From the experiment results, we realize that our approach achieves high recall but relatively low precision, which means that the returned subgraph contains some nodes out of the annotated code element set. The difference between precision and recall is especially significant in Lucene project so that we have a detailed investigation of the second group query of Lucene in this section.

First, we analyzed the precision of each query and the result is shown in Figure 4. There are generally more code elements returned by our approach and the size of the most subgraphs is between 5-8. The main deviation is caused by MST based subgraph extending algorithm. Since it is difficult to figure out which type of relationship is more beneficial, we set all the edge weight to be equal. Therefore, it would generate many paths with the same length and be more likely to introduce unlabeled nodes. Although we cannot assure that these code elements are necessary, we believe that it is helpful to provide more information for users based on the fact that our recall is quite satisfying. The optimization of the subgraph extending algorithm can be a future work of this paper.

Then we analyzed the recall for each query and the result is shown in Figure 5. Our approach achieves quite high average recall and for 10 questions in the dataset the recall is 100%. The relatively lower recall question is question No. 13 and No.15. We detailed investigate the No. 15 question "How to query document have any term in a set?". The major problem for this query is the ambiguity of natural language. The word "set" in the question means the mathematical concept, while there are many method names in source code which use "set" as a verb. Thus, our approach would finally match it to the *setTerm* method using the text similarity.

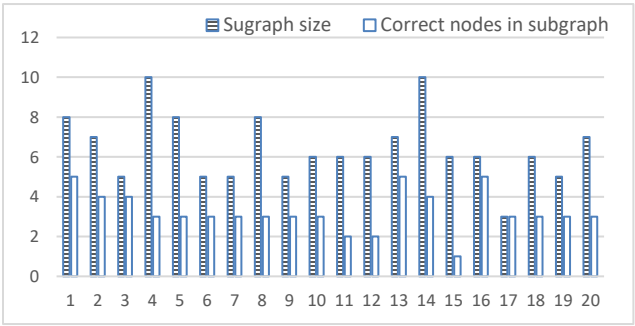

Figure 4: Precision of each question

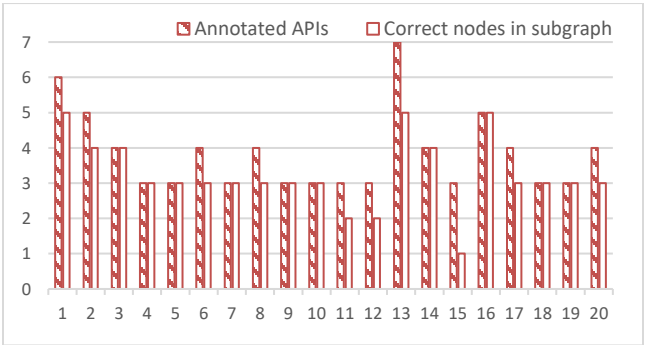

Figure 5: Recall of each question

4 REALTED WORK

In this section, we discuss the related work with our approach, which mainly consists of two parts: graph embedding techniques and code search approaches.

4.1 Source Code Search

Traditional code search mainly rely on the text similarity between code and query/question, such as Krugle [4], Ohloh [5], and Sourcerer [6]. Typically, Sourcerer[6] stores the parsed source code into their local database and use Lucene to build index on it. These methods usually support keywords based query or regex expression based query, return the relevant code snippets. However, they usually cannot achieve satisfactory results in practice for lack of semantic understanding of search question.

To "understand" search questions, many NLP techniques approaches have been applied to code search now. Typically, SNIFF [18] finds the related documents for APIs firstly, then preform the natural language based code search on the annotated code snippets. CodeHow [8] identifies some potential APIs that may be related to a question, and use the Extended Boolean Model to applied these APIs' information to the code search process. Some methods tried to add similar words to expand users' natural language question [11][19][20]. However, another research proves that the results would be much worse if some improper words were introduced[21]. Therefore, Wang et al. [22] proposed to use feedback information from users to optimize query, Refoqus [23] pointed out that we need to predict a query's quality and gave some query reformulation strategies.

As more and more data need to be processed, machine learning based code search and API recommendation become a new research hot now. ROSF [24] used supervised learning to re-rank the candidate results, DeepAPI [25] used RNN Encoder-Decoder model to translate natural language query to API invocation series, While Assistant [26] learned a translation model from a mount of code-text pairs. Although these approaches could improve the accuracy of APIs location, they need to accumulate and leverage a lot of auxiliary information. In our work, we especially focus on using the structural information contained in source code itself, users don't need to accumulate lots of auxiliary documents or query histories.

4.2 Code Graph Search

Researchers have pointed out that developers need more relevant information about the APIs, such as invoking chain and class diagram, to understand APIs quickly during code search [27][28]. Following this idea, C. MacMillan [11] and W.K. Chan [12] organized source code as directed graph and transform the code search problem to the subgraph search problem. C. MacMillan et.al.[11] implemented Portfolio, which used PageRank and SAN algorithm to return the top k related nodes in the graph as answers. While W.K. Chan et al [12] returned a connected subgraph so that it can present the relationships between these nodes clearly. RACS [29] also proposed a relationship aware code search approach for JavaScript frameworks.

The same idea is also applied in other scenarios such as documentation retrieval and feature location [30][31][39]. API2VEC [32] used the CBOW model to learn API embeddings from API sequences extracted from source code. Inspired by these work, we also build the code graph for software project and use graph embedding techniques to mine the deep structural information in source code, which can capture more contexts information in code search.

4.3 Graph Embedding

Graph embedding is a popular representation learning technique and is wildly applied to visualization, node classification and relationship prediction tasks. Existing graph embedding techniques mainly include the following categories [33]:

Factorization based embedding techniques. This kind of methods use matrix to represent the graph and use factorization techniques to obtain the embedding vectors. The typical matrix types include adjacent matrix, Laplacian matrix, Katz similarity matrix, etc. Laplacian Eigenmaps [34] is a representative of Laplacian matrix decomposing algorithm while GF [35] algorithm is conducted on the adjacent matrix. However, these methods usually have a quadratic time complexity so that they are not suitable for large scale graphs.

Random walk based embedding techniques. This kind of methods can approximately represent various properties of graph including nodes centrality and similarity. DeepWalk [36] predicts a node's embedding vector according its neighboring nodes, which is motivated by the neural language model SkipGram. It maximizes the probability of observing the last k nodes and the next k nodes in the random walk centered at current node to preserves the higher-order proximity. They are especially useful to capture the partial structure information of a large graph.

Deep learning based embedding techniques. Some of these methods use Deep Autoencoder to learn the non-linear structure in graph and generate the embedding vectors. SDNE [37] consists of two parts: unsupervised part and supervised part. It first utilizes an autoencoder to generate embedding vector for a node which can reconstruct its neighborhood, and then applies a penalty to the similar nodes which are mapped far from each other based on Laplacian Eigenmaps. Some methods utilize deep convolutional network to obtain the embedding vector, such as the semi-supervised method [38] proposed by T.N. Kipf, and the advantage of CNN is that it is relatively faster and is especially suitable for sparse graphs. Although this kind of method can better capture the non-linearity, they are usually computationally expensive.

There are other methods not strictly belong to the above categories. For example, LINE [17] explicitly defines the first-order and second-order proximity functions. It calculates two joint probability distributions for each pair of vertices, one using adjacent matrix and the other using the embedding. Then, it minimizes the Kullback-Leibler (KL) divergence of these two distributions to optimize the embedding vectors. LINE is scalable to embed large networks with a low level of complexity and is able to preserve both local and global structures.

5 CONCLUSION

In this paper, we propose an embedding based code graph search approach for software project. Graph embedding is a kind of representation learning technique, it could help us represent and utilize structural information in the source code, improving the efficiency of source code search and software reuse. At the same

time, we believe that code graph could help developers understand code element's implement logic or usage scenarios more quickly during code retrieval process. In the experiments, we select two famous open source projects, *Apache Lucene* and *POI*, as examples to build code graph and perform code search tasks. The experimental results show that our approach improves F1-score by 10% than existing shortest path based code graph search approach, while reduces average response time about 60 times. In the future, we will focus on improving code graph embedding method, and apply our approach to more projects and more code search tasks.

ACKNOWLEDGMENTS

This paper is supported by the National Key Research and Development Project of China (Grant No. 2016YFB1000801) and the National Natural Science Fund for Distinguished Young Scholars (Grant No. 61525201).

REFERENCES

[1] L. Moreno, G. Bavota, M. D. Penta, R. Oliveto, and A. Marcus. 2015. How can I use this method? In *Proceedings of the 37th International Conference on Software Engineering*. IEEE, 880-890.

[2] R. Sirres, T. F. Bissyand´e, Dongsun Kim, David Lo, J. Klein, Kisub Kim, Yves Le Traon. 2018. Augmenting and structuring user queries to support efficient free-form code search. *Empirical Software Engineering*. Springer, 2622-2654.

[3] J. Stylos and B. A. Myers. 2006. Mica: A web-search tool for finding API components and examples. In *Proceedings of the Visual Languages and Human-Centric Computing*,.195–202.

[4] Krugle code search. http://www. krugle. com/

[5] Ohloh code search. https://code. ohloh. net/

[6] E. Linstead, S. Bajracharya, T. Ngo, P. Rigor, C. Lopes, and P. Baldi. 2009. Sourcerer: mining and searching internet-scale software repositories. *Data Mining and Knowledge Discovery*, vol. 18, 300-336.

[7] R. Baeza-Yates and B. Ribeiro-Neto. 2011. *Modern Information Retrieval: The Concepts and Technology behind Search, second edition.* Addison-Wesley.

[8] Fei Lv, Hongyu Zhang, Jian-guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. 2015. Codehow: Effective code search based on API understanding and extended boolean model. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, IEEE, 260-270.

[9] E. Hill, L. L. Pollock, and K. Vijay-Shanker. 2011. Improving source code search with natural language phrasal representations of method signatures. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, IEEE, 524 – 527.

[10] S. Haiduc, G. Bavota, A. Marcus, R. Oliveto, A. De Lucia, and T. Menzies. 2013. Automatic query reformulations for text retrieval in software engineering. In *Proceedings of the 35th International Conference on Software Engineering*, IEEE, 842 – 851.

[11] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu. 2011. Portfolio: finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering*, ACM, 111–120.

[12] W. K. Chan, H. Cheng, David Lo. 2012. Searching connected API subgraph via text phrases. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 10, 1-11.

[13] http://checkstyle. sourceforge. net/

[14] http://www. csg. ci. i. u-tokyo. ac. jp/~chiba/javassist/

[15] http://www. scovetta. com/yasca. html/

[16] http://www. eclipse. org/jdt/

[17] J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, Q. Mei. 2015. Line: Largescale information network embedding. In *Proceedings of the 24th International Conference on World Wide Web*, 1067–1077.

[18] S. Chatterjee, S. Juvekar, and K. Sen. 2009. Sniff: A search engine for Java using free-form queries. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering*, 385–400.

[19] Y. Tian, D. Lo, and J. L. Lawall. 2014. Automated construction of a software specific word similarity database. In *Proceedings of CSMR-WCRE*, 44–53.

[20] J. Yang and L. Tan. 2012. Inferring semantically related words from software context. In *Proceedings of the 9th IEEE Working Conference of Mining Software Repositories*,.161–170.

[21] G. Sridhara, E. Hill, L. L. Pollock, and K. Vijay-Shanker. 2008. Identifying word relations in software: A comparative study of semantic similarity tools. In *Proceedings of the 16th IEEE International Conference on Program Comprehension*, 123–132.

[22] S. Wang, D. Lo, and L. Jiang. Active code search: Incorporating user feedback to improve code search relevance. 2014. In *Proceedings of the 29th IEEE/ACM International Conference on Automated Software Engineering*, ACM, 677-682.

[23] S. Haiduc, G. Bavota, A. Marcus, R. Oliveto, A. De Lucia, and T. Menzies. 2013. Automatic query reformulations for text retrieval in software engineering. In *Proceedings of the 2013 International Conference on Software Engineering*, 842-851.

[24] H. Jiang, L. Nie, Z. Sun, Z. Ren, W. Kong, T. Zhang, X. Luo. 2016. ROSF: Leveraging information retrieval and supervised learning for recommending code snippets. *IEEE Transactions on Services Computing*.

[25] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, Sunghun Kim. 2016. Deep API learning. In *Proceedings of the 24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*. ACM, 631–642.

[26] Kyle Richardson and Jonas Kuhn. 2017. Function Assistant: A Tool for NL Querying of APIs. In *Proceedings of EMNLP*.

[27] J. Sillito, G. C. Murphy, and K. De Volder. 2008. Asking and answering questions during a programming change task. *IEEE Transactions on Software. Engineering*, 34(4):434–451.

[28] J. Sillito, G. C. Murphy, and K. D. Volder. 2006. Questions programmers ask during software evolution tasks. In *Proceedings of the ACM SIGSOFT 14th International Symposium on the Foundations of Software Engineering*, ACM, 23–34.

[29] X. Li, Z. Wang, Q. Wang, S. Yan, T. Xie, and H. Mei. 2016. Relationship-Aware code search for JavaScript frameworks. In *Proceedings of the 24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*. ACM, 690–701.

[30] Fu K, Wu YJ, Peng X, Zhao WY. A feature location method based on call chain analysis. Computer Science, 2017, 44(4):56-59.

[31] Li Z, Niu J, Wang K, Xin YY. 2018. Optimization of Source Code Search Based on Multi-feature Weight Assignment. *Journal of Computer Applications*, 38(3):812-817.

[32] T. D. Nguyen, A. T. Nguyen, H. D. Phan and T. N. Nguyen. 2017. Exploring API Embedding for API Usages and Applications. In *Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering*, IEEE, 438-449.

[33] P. Goyal, E. Ferrara. 2017. Graph Embedding Techniques, Applications, and Performance: A Survey. arXiv:1705. 02801 [cs. SI].

[34] M. Belkin, P. Niyogi. 2001. Laplacian eigenmaps and spectral techniques for embedding and clustering. In *NIPS*, Vol. 14, 585–591.

[35] A. Ahmed, N. Shervashidze, S. Narayanamurthy, V. Josifovski, A. J. Smola. 2013. Distributed large-scale natural graph factorization. In *Proceedings of the 22nd International Conference on World Wide Web*, 37–48.

[36] B. Perozzi, R. Al-Rfou, S. Skiena. 2014. Deepwalk: Online learning of social representations. In *Proceedings 20th International Conference on Knowledge Discovery and Data Mining*, 701–710.

[37] D. Wang, P. Cui, W. Zhu. 2016. Structural deep network embedding. In *Proceedings of the 22nd International Conference on Knowledge Discovery and Data Mining*, ACM, 1225–1234.

[38] T. N. Kipf, M. Welling. 2017. Semi-supervised classification with graph convolutional networks. [online] Available: https://arXiv:1609.02907. Published as a conference paper at ICLR.

[39] Lin ZQ, Zou YZ, Zhao JF, Cao YK, Xie B. 2017. Software Text Semantic Search Approach Based on Code Structure Knowledge. *Ruan Jian Xue Bao/Journal of Software*. http://www.jos.org.cn/1000-9825/0000.htm

[40] https://poi.apache.org/spreadsheet/quick-guide.html