Supporting Exploratory Code Search with Differencing and Visualization

Wenjian Liu[1,2,3], **Xin Peng**[1,2,3], **Zhenchang Xing**[4], **Junyi Li**[1,2,3], **Bing Xie**[5] **and Wenyun Zhao**[1,2,3]

[1]School of Computer Science, Fudan University, China
[2]Shanghai Key Laboratory of Data Science, Fudan University, China
[3]Shanghai Institute of Intelligent Electronics & Systems, China
[4]Research School of Computer Science, Australian National University, Australia
[5]School of Electronics Engineering and Computer Science, Peking University, China
{*wjliu14, pengxin, 16212010012, wyzhao*}*@fudan.edu.cn*
zhenchang.xing@anu.edu.au, *xiebing@pku.edu.cn*

***Abstract*—Searching and reusing online code has become a common practice in software development. Two important characteristics of online code have not been carefully considered in current tool support. First, many pieces of online code are largely similar but subtly different. Second, several pieces of code may form complex relations through their differences. These two characteristics make it difficult to properly rank online code to a search query and reduce the efficiency of examining search results. In this paper, we present an exploratory online code search approach that explicitly takes into account the above two characteristics of online code. Given a list of methods returned for a search query, our approach uses clone detection and code differencing techniques to analyze both commonalities and differences among the methods in the search results. It then produces an exploration graph that visualizes the method differences and the relationships of methods through their differences. The exploration graph allows developers to explore search results in a structured view of different method groups present in the search results, and turns implicit code differences into visual cues to help developers navigate the search results. We implement our approach in a web-based tool called *CodeNuance*. We conduct experiments to evaluate the effectiveness of our *CodeNuance* tool for search results examination, compared with ranked-list and code-clustering based search results examination. We also compare the performance and user behavior differences in using our tool and other exploratory code search tools.**

I. Introduction

Today, the ability to search, understand and use online knowledge affects software developers' efficiency and success in work [1], [2]. Among various sources of online knowledge, online code (e.g., open source projects on GitHub [3]) constitutes a core asset for software engineering domain. Searching for online code for reuse has become an integral part of software development [4], [5], [6], [7]. The prevalence of online code, on one hand, increases the chance that the needed code is likely out there, but on the other hand, the two characteristics of online code make it difficult to find the needed code. First, many pieces of code are largely similar, but at the same time have subtle differences. Second, several pieces of code may differ in different ways from one another, and thus may form complex relations through their differences.

Consider three pieces of code $Snippet$ 3, $Snippet$ 4 and $Snippet$ 5 in Fig. 1. $Snippet$ 3 and $Snippet$ 4 differ as $Snippet$ 4 uses *StringBuffer* to store strings when converting an input stream to a string, while $Snippet$ 3 uses *StringBuilder*. $Snippet$ 5 also uses *StringBuilder*, but it uses *try-catch* to handle *IOException* when calling *StringBuilder.append()*, while $Snippet$ 3 does not. These three pieces of code form a "gradient evolution" relation through their differences. The differences between $Snippet$ 4 and $Snippet$ 5 can be represented as those between $Snippet$ 4 and $Snippet$ 3 (i.e., replace *StringBuffer* with *StringBuilder*), and then those between $Snippet$ 3 and $Snippet$ 5 (i.e., add *try-catch* to handle *IOException*).

Typically, developers use a general search engine like Google or a code search engine like Open Hub [8] or GitHub Search [3] to search online code. The developer enters several keywords as a query to the search engine, and the search engine returns a ranked list of relevant code for the query. Consider the above-mentioned three pieces of code and many more pieces of code similar to them. As the search engine considers only the relevance between code and query, it is difficult to rank such similar code to a query "convert input stream string", unless the developer's query contains specific keywords (e.g., "StringBuffer", "IOException") that are only in some code but not others. However, developers cannot be this specific when searching for code of unfamiliar functionality, because they are unsure about the ways to achieve the desired functionality. Unfortunately, the ranked list provides no information about code commonalities, differences and relationships. When viewing a piece of code, there is no easy way to determine which other pieces of code can be skipped because they provide no new information or unwanted solutions, or which pieces of code should be examined because they enhance the currently viewed code or provide alternative solutions.

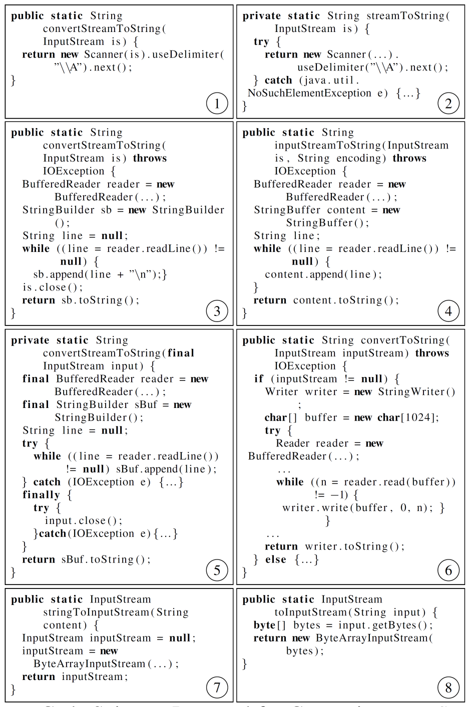
When searching for online code of unfamiliar functionality, developers are faced with the "unknown-unknown" challenge. That is, they do not know what they do not know in order to achieve their goals. Therefore, developers need an effective approach to explore the search results to a query in order to learn "unknowns" in the search results. Considering the two characteristics of online code, i.e., largely similar but subtly different and having implicit relationships in terms of code commonalities and differences, we identify two desirable properties for an effective exploratory code search approach: 1) *detect and model commonalities and differences among code in search results*; 2) *provide a structured view of search results in terms of code commonalities, differences and relationships*. These two properties turn "unknowns" in search results into explicit information, which will help to bridge the gap between the developers' needs and their unknowns.

Several code search [7] or feature location [9], [10] tools support exploratory search [11]. These tools expose multiple facets of code, such as language concepts like import, type hierarchy, data type, or frequent words in code, for filtering search results or assisting query reformulation. But they make no consideration of commonalities, differences and relationships among code in search results. Some tools assist exploratory feature location by clustering similar code in search results [9], [10] and analyzing code relationships through common attributes of code. In the context of online code search, many pieces of online code have only subtle differences. These subtle differences will be shadowed by large code commonalities.

In this paper, we present an exploratory search tool (called $CodeNuance$)[1] for online code search, with an explicit consideration of the above-mentioned two desirable exploratory code search properties. Our tool takes as input a ranked list of methods returned by a search engine for a query. It first groups methods in the ranked list by Type-I/II clones [13]. It then uses code differencing technique [14], [15] to detect code differences between method groups. Next, $CodeNuance$ models the relationships between method groups through their differences in an exploration graph. The developer can interact with this graph to explore the search results. To support effective results exploration, $CodeNuance$ analyzes the method differences to reduce unnecessary exploration edges, indicates relative differences between methods by gradient node colors, and labels the exploration edges with the method differences.

We implemented our approach in a web-based tool on top of a code base of over 9-millions Java methods extracted from 207,711 Java projects on Github. To evaluate our approach and tool support, we created 100 code search tasks from the 100 top-voted implementation-related Java questions on Stack Overflow. Our evaluation focuses on three aspects: result examination effort, comparison with keyword recommendation based exploratory code search technique, and influences of exploration graph complexity on tool usage. Our experiments show that $CodeNuance$ requires the least result examination effort to find the first ground-truth method, compared with ranked-list and code-clustering based result examination. Compared with keyword recommendation based

[1]A demo of $CodeNuance$ can be found at our tool page [12].



```
public static String
      convertStreamToString(
      InputStream is) {
  return new Scanner(is).useDelimiter(
        "\\A").next();
}
                                        ①
```

```
private static String streamToString(
      InputStream is) {
  try {
    return new Scanner(...).
          useDelimiter("\\A").next();
  } catch (java.util.
 NoSuchElementException e) {...}
}                                       ②
```

```
public static String
      convertStreamToString(
      InputStream is) throws
      IOException {
  BufferedReader reader = new
        BufferedReader(...);
  StringBuilder sb = new StringBuilder
        ();
  String line = null;
  while ((line = reader.readLine()) !=
         null) {
    sb.append(line + "\n");}
  is.close();
  return sb.toString();
}                                       ③
```

```
public static String
      inputStreamToString(InputStream
      is, String encoding) throws
      IOException {
  BufferedReader reader = new
        BufferedReader(...);
  StringBuffer content = new
        StringBuffer();
  String line;
  while ((line = reader.readLine()) !=
         null) {
    content.append(line);
  }
  return content.toString();
}                                       ④
```

```
private static String
      convertStreamToString(final
      InputStream input) {
  final BufferedReader reader = new
        BufferedReader(...);
  final StringBuilder sBuf = new
        StringBuilder();
  String line = null;
  try {
    while ((line = reader.readLine())
          != null) sBuf.append(line);
  } catch (IOException e) {...}
  finally {
    try {
      input.close();
    }catch(IOException e){...}
  }
  return sBuf.toString();
}                                       ⑤
```

```
public static String convertToString(
      InputStream inputStream) throws
      IOException {
  if (inputStream != null) {
    Writer writer = new StringWriter()
          ;
    char[] buffer = new char[1024];
    try {
      Reader reader = new
 BufferedReader(...);
      ...
      while ((n = reader.read(buffer))
            != -1) {
       writer.write(buffer, 0, n); }
             }
    ...
    return writer.toString();
  } else {...}
}                                       ⑥
```

```
public static InputStream
      stringToInputStream(String
      content) {
  InputStream inputStream = null;
  inputStream = new
    ByteArrayInputStream(...);
  return inputStream;
}                                       ⑦
```

```
public static InputStream
      toInputStream(String input) {
  byte[] bytes = input.getBytes();
  return new ByteArrayInputStream(
        bytes);
}                                       ⑧
```

Fig. 1. Some Code Snippets Returned for Converting *InputStream* to *String*

technique, $CodeNuance$ helps developers find desired code faster and more accurately, which can be attributed to the $CodeNuance$'s ability to turn implicit method differences and relationships present in the search results into explicit, explorable information. Our study also identifies several opportunities for reducing the complexity of exploration graph for more effective examination of code differences and relationships.

This paper makes the following contributions: 1) We identify two important characteristics of online code that must be explicitly handled for effective online code search; 2) We present an exploratory code search approach and tool that explicitly address these two characteristics of online code in system design; 3) We conduct a series of experiments and user studies to evaluate the effectiveness of our approach and identify improvement opportunities.

II. Motivation

A developer Jerry wants to develop a Java method to convert input stream to strings. As he does not have knowledge of how to implement this feature, he searches the Internet to find some code for reference. He issues a query "convert InputStream to String" to a code search engine, and the search engine returns hundreds of methods that are considered relevant to the query. The good thing is that there should be some code that can meet Jerry's need. The bad thing is which one is that needed code among hundreds of candidates.

Jerry opens the first-ranked method ($Snippet$ 1 in Fig. 1). It is a short method using the *Scanner* API. After reading the *Scanner*'s specification, Jerry feels that $Snippet$ 1 can partially meet his need, but it has something he is not satisfied with. First, he has to learn delimiter patterns to properly break the input into tokens. Second, $Snippet$ 1 reads only one next token, not the whole input. Jerry reads several methods down the list. Unfortunately, it is a waste of time, because these methods are code clones that differ from *Snippet* 1 only in delimiter patterns, or some enhanced versions of *Snippet* 1, such as *Snippet* 2 which handles *NoSuchElementException*.

After reading a dozen of methods, Jerry runs into two alternative solutions (*Snippet* 3 and *Snippet* 4) that use *BufferedReader* and *StringBuilder* (or *StringBuffer*) to implement the feature he wants. But *Snippet* 3 and *Snippet* 4 throw *IOException* for calling *append()*, while Jerry wants to handle the exception within the method. The search results actually contain a method (*Snippet* 5) which is exactly what Jerry wants, but he does not know this. He reads a few more methods in the search results, and runs into *Snippet* 6 which also throws *IOException* but it is for the *Writer* API. He also wastes some time on *Snippet* 7 and *Snippet* 8 which are the opposite (i.e., convert string to input stream) to what he wants. Jerry gives up, as the current search results do not seem to have what he wants. He searches with another query "convert input stream string catch IOException", and starts another round of ordeal to explore the long list of search results.

The difficulties that Jerry experiences are not simply because his query is not precise enough, the relevance ranking of the search engine is not smart enough, or he is not lucky enough to run into the method he wants. The fundamental challenge that Jerry is faced with is that he does not know what he does not know in the search results. We refer to this challenge as the "unknown-unknown" challenge in exploring code search results. Let's now see how our approach turns "unknowns" in search results into explicit information, and turns an ad-hoc exploration of search results into a much more informed and structured exploration.

Our approach groups identical or near-identical methods (i.e., Type-I and Type-II code clones [13]) in the search results, so that developers can avoid wasting time on identical or near-identical methods that provide no or little new information to what they already know. For example, the 8 methods in Fig. 1 represent 8 Type-I and Type-II clone groups in the search results to the query "convert input stream string". Methods in a group differ only in space, identifiers, literals, and/or comments. Instead of having to go through these identical or near-identical methods, Jerry can focus his investigation on the 8 methods in Fig. 1.

To assist Jerry in investigating these 8 methods, our approach uses code differencing techniques to detect differences between them. Our exploration graph provides a structured view of these 8 methods (i.e., 8 method groups) in the search results (see Fig. 2). Methods are connected based on the gradient analysis of their differences. The method with the minimum overall differences from all other methods is recommended as the starting point of exploration. The rationale is that this starting point, such as *Snippet* 3 in Fig. 2, can be transformed into the other methods with minimum changes.

The exploration graph displays the method differences on the edges. The difference labels close to a method of an edge show the APIs and programming constructs in this method, but

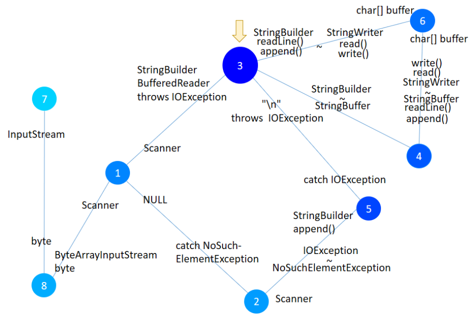


Fig. 2. An Exploration Graph for the Query "convert InputStream to String"

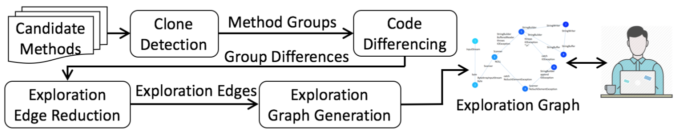


Fig. 3. Approach Overview

not in the method at the other end of the edge. The difference labels in the middle of an edge show the differences that appear in the corresponding positions of the two methods. For example, the labels on the edge connecting *Snippet* 3 and *Snippet* 4 mean that *Snippet* 3 uses *StringBuilder* but not *StringBuffer*, while *Snippet* 4 is the opposite. Looking at the code of the starting point method (*Snippet* 3 in Fig. 1) and the difference labels on the edges in Fig. 2, Jerry can quickly learn that these 8 methods provide a good range of alternatives, such as *Scanner*, *StringBuilder*, *StringBuffer*, *StringWriter*, that he may use for the wanted feature. He decides to investigate further these 8 methods.

The exploration graph colors the nodes in gradient colors depending on a method's relative difference from the starting point method. From the exploration graph in Fig. 2, Jerry quickly observes that *Snippet* 4, *Snippet* 5 and *Snippet* 6 are relatively more similar to *Snippet* 3, while *Snippet* 1, *Snippet* 2, *Snippet* 7 and *Snippet* 8 are relatively more different from *Snippet* 3.

The differences between *Snippet* 3 and *Snippet* 5 lead Jerry to *Snippet* 5, because it does not throw *IOException* but catches *IOException* in the method. *Snippet* 5 meets Jerry's need. Or if Jerry wants synchronized processing in multi-threads, he could synthesize *Snippet* 4 (using *StringBuffer* rather than *StringBuilder*) and exception handling of *Snippet* 5. Jerry has a quick check of *Snippet* 1 and *Snippet* 6, which confirms that *Snippet* 3, *Snippet* 5 or *Snippet* 4 are easier to reuse, while *Snippet* 1 is too basic and *Snippet* 6 is too complex. He skips *Snippet* 2 because it does not differ much from *Snippet* 1 (only in exception handling). He also skips *Snippet* 7 and *Snippet* 8 because they are even simpler than *Snippet* 1.

III. Approach

In this section, we first present an overview of the approach and then describe its key steps.

A. Overview

Fig. 3 shows the main steps of our approach. Our approach supports method-level exploratory code search. It takes as input a set of candidate methods in the search results to a query and generates an exploration graph (see Fig. 2) from these candidate methods. The user can interact with the exploration graph to explore the search results. The input of the approach (i.e., candidate methods) can be the results from any existing code search techniques (e.g., keyword-based search) that can return a list of methods as the search results to the query (e.g., a set of keywords) from the user.

The code search usually returns a large number of methods for a query. In the context of online code search, many of these methods are identical or have only minor differences in space, layout, literals, comments, etc. For such methods in the search results, it is usually sufficient to examine any one of them and skip the others, as the others will unlikely provide new programming information. Therefore, our approach uses clone detection to cluster the candidate methods into groups of Type I/II cloned methods (method groups in short).

Although methods in a group are identical or nearly identical, methods across groups are different structurally or semantically. Our approach randomly selects a method from each method group as its representative method, and employs a code differencing algorithm [14], [15] to detect differential token pairs between each pair of representative methods. Knowing the differences across method groups will help the user understand alternative or enhanced solutions in the search results.

Based on pairwise differences between representative methods, a fully connected exploration graph can be produced, in which each pair of methods has an edge. This graph cannot be effectively explored. Our approach attempts to simplify the exploration graph by eliminating unnecessary edges. An unnecessary edge means that the differences of the two methods connected by the edge can be synthesized by a series of differences of some other methods in between. Moreover, our approach also eliminates the edges between the two methods that have large differences.

Finally, our approach visualizes the edge-reduced exploration graph to provide a structured view of representative methods in the search results and their differences. It recommends the method with the minimum overall differences from all other methods as the starting point of exploration. The method nodes are colored based on their relative difference from the starting point method. The edges are annotated with API and programming construct differences extracted from differential token pairs between the two methods.

B. Clone Detection

Given a set of candidate methods we use a clone detector (e.g., CCFinderX [16], [17] used in our implementation) to group the methods that are Type-I or Type-II clones [13]. Each method group includes a number of candidate methods that are identical or near-identical and is treated as a basic unit of code search. Note that a method group may have only one method. For each group that has more than one method, a representative method is randomly selected for the following analysis.

C. Code Differencing

To determine the differences between two method groups, our approach compares their representative methods using code differencing algorithms proposed by Lin et al. [14], [15]. This algorithm first transforms each method into a token sequence, which consists of different kinds of tokens such as keywords, separators, operators, literals, and identifiers. It then computes a Longest Common Subsequence (LCS) between the two token sequences. After that, it identifies the corresponding differential ranges (i.e., the token subsequences that are not in the LCS) in the two token sequences, and produces a list of corresponding differential token pairs.

Let mg_1 and mg_2 be the representative method of two method groups, respectively. We denote the differences from mg_1 to mg_2 as $Diff(mg_1, mg_2)$ or simply $diff_{12}$. Note that $diff_{12}$ and $diff_{21}$ are symmetric. A differential token pair [t_1, t_2] $\in diff_{12}$ indicates that t_1 from mg_1 and t_2 from mg_2 correspond to each other but are different. For example, a difference from $Snippet$ 3 to $Snippet$ 4 in Fig. 1 is [*StringBuilder*, *StringBuffer*], which are the two different APIs that the two methods use respectively. t_1 or t_2 (but not both) can be ϵ, i.e., a placeholder token, which means that there is no corresponding token to the other token in the pair. For example, a difference from $Snippet$ 2 to $Snippet$ 1 in Fig. 1 is [*NoSuchElementException*, ϵ], which is the additional exception handling in $Snippet$ 2 but not in $Snippet$ 1.

D. Exploration Edge Reduction

Algorithm 1 identifies unnecessary edges in the fully connected graph of the representative methods of all method groups in the search results, based on the pairwise differences between the representative methods. It takes as input the set of representative methods MG and returns a set of unnecessary exploration edges between these methods $UEdges$.

For each pair of representative methods mg_1 and mg_2 in MG, the algorithm (Line 5-19) computes the synthesis of the differences from mg_1 to a transition method mg_3 (i.e., $diff_{13}$) and the differences from mg_3 to mg_2 (i.e., $diff_{32}$). This synthesis, i.e., $diff_{132}$ in the algorithm, includes all token pairs resulting from the transition of the token pairs from $diff_{13}$ to $diff_{32}$ (Line 9-18). A token pair [a, b] $\in diff_{13}$ and a token pair [b, c] $\in diff_{32}$ form a transition. The algorithm records the count of transitions (Line 11). If $a \neq c$, a token pair [a, c] is synthesized and added in $diff_{132}$. In addition, $diff_{132}$ also includes all the token pairs in $diff_{13}$ and $diff_{32}$ that are not involved in any transitions (Line 19).

Next, the algorithm computes the ratio of the count of token-pair transitions to the number of token pairs in $diff_{132}$ (i.e., $transCount/|diff_{132}|$) (Line 20). This ratio reveals the percentage of token pairs in $diff_{132}$ that are synthesized by transitions. The higher the ratio, the more symmetric the differences among mg_1, mg_2 and mg_3 are. If mg_1, mg_2 and mg_3 form a symmetric triangle (e.g., mg_1, mg_2 and mg_3 in Fig. 4), we choose to keep all the three edges between them.

Only when the transition ratio is lower than a given threshold Th_{trans}, the algorithm considers whether the edge between mg_1 and mg_2 could be replaced by the two edges going through mg_3. The algorithm computes the similarity of $diff_{132}$ and $diff_{12}$ (i.e., the differences from mg_1 to mg_2) using Eq. 1 (Line 21), i.e., the percentage of the common token pairs in $diff_{132}$ and $diff_{12}$. The higher the similarity, the more differences from mg_1 to mg_2 can be synthesized from the differences $diff_{13}$ and $diff_{32}$. If the similarity is above a given threshold Th_{sim}, the edge between mg_1 and mg_2 is marked as an unnecessary edge (Line 22). Note that the algorithm only marks the edge as unnecessary but does not eliminate it from the graph, because the edge is still needed in the decision process of other edges.

$$Sim_d(diff_{132}, diff_{12}) = \frac{|diff_{132} \bigcap diff_{12}| \times 2}{|diff_{132}| + |diff_{12}|} \tag{1}$$

Algorithm 1 Identify Unnecessary Exploration Edges

```
1: function IDENTIFYUNNECESSARYEDGES($MG$)
2:     $UEdges = \{\ \}$
3:     for each $mg_1, mg_2 \in MG \wedge mg_1 \neq mg_2$ do
4:         $diff_{12} = Diff(mg_1, mg_2)$
5:         for each $mg_3 \in MG - \{mg_1, mg_2\}$ do
6:             $diff_{13} = Diff(mg_1, mg_3)$
7:             $diff_{32} = Diff(mg_3, mg_2)$
8:             $diff_{132} = \{\ \}, transCount = 0$
9:             for each $[a, b] \in diff_{13}$ do
10:                if $\exists [b, c] \in diff_{32}$ then
11:                    $transCount = transCount + 1$
12:                    if $a \neq c$ then
13:                        $diff_{132} = diff_{132} \bigcup \{[a, c]\}$
14:                    end if
15:                    $diff_{13} = diff_{13} - \{[a, b]\}$
16:                    $diff_{32} = diff_{32} - \{[b, c]\}$
17:                end if
18:            end for
19:            $diff_{132} = diff_{132} \bigcup diff_{13} \bigcup diff_{32}$
20:            if $transCount/|diff_{132}| < Th_{trans}$ then
21:                if $Sim_d(diff_{132}, diff_{12}) > Th_{sim}$ then
22:                    $UEdges = UEdges \cup (mg_1, mg_2)$
23:                    break
24:                end if
25:            end if
26:        end for
27:    end for
28:    return $UEdges$
29: end function
```

Fig. 4 shows an example of unnecessary exploration edge identification. Initially, the fully connected exploration graph for four representative methods contains six edges. The differences between each pair of the two methods are shown on the edge. Note that $diff_{ij}$ and $diff_{ji}$ are symmetric. Take the edge between mg_4 and mg_1 as an example. Let mg_3 be the transition method. We have $diff_{43} = \{[d, \epsilon]\}$ and $diff_{31} = \{[c, a]\}$. None of the token pairs in $diff_{43}$ and $diff_{31}$ are involved in transitions. Based on Algorithm 1, the synthesis $diff_{431}$ is $\{[d, \epsilon], [c, a]\}$. As there are no token pairs resulting from transitions in $diff_{431}$ and $diff_{431}$ is the same as $diff_{41}$, the edge between mg_4 and mg_1 is marked as an unnecessary edge (dotted line). Similarly, the edge between mg_4 and mg_2 is marked as an unnecessary edge.

In contrast, the edge between mg_2 and mg_3 will not be marked as an unnecessary edge. Consider mg_1 as a transition method. The synthesis $diff_{213}$ is $\{[b, c]\}$. The ratio $transCount/|diff_{213}|$ is 1/1=1, which means that all the to-

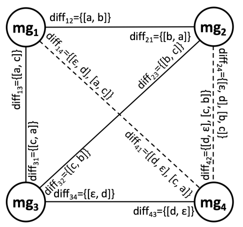


Fig. 4. An Example of Unnecessary Exploration Edge Identification

ken pairs in $diff_{213}$ are synthesized by transitions. Therefore, the edge between mg_2 and mg_3 is kept.

After eliminating unnecessary exploration edges identified by Algorithm 1, our approach further eliminates the edges between the two representative methods whose distance is larger than a threshold. A large distance between the two methods indicates that their code has little in common. So there is no need to directly connect the two methods in the exploration graph. The distance between the two methods mg_1 and mg_2 is computed as Eq. 2, where the $Length$ function returns the token length of a method or the number of differential token pairs between the two methods. The distance threshold of the set of representative methods MG is computed as Eq. 3, which first calculates the shortest distance of each method in MG to the other methods and then returns the maximum value of all the shortest distances. We use this dynamically computed threshold rather than a statically specified one in order to eliminate edges whose distances are relatively large for the set of representative methods.

$$Dist(mg_1, mg_2) = \frac{Length(Diff(mg_1, mg_2))}{Length(mg_1) + Length(mg_2)} \tag{2}$$

$$Thresh(MG) = \max_{mg \in MG} \min_{mg' \in MG \wedge mg' \neq mg} Dist(mg, mg') \tag{3}$$

Eliminating unnecessary edges and large-distance edges may break the connectivity of the exploration graph, i.e., a part of the graph may be disconnected from the other part of the graph. To ensure connectivity of the edge-reduced exploration graph, we compute a minimum connected subgraph of the initial exploration graph with the distance between each pair of methods as in Eq. 2 as the cost. If the connectivity of the exploration graph is broken, the edges in the minimum connected subgraph but have been eliminated will be added back to the edge-reduced exploration graph.

E. Exploration Graph Generation

Given the edge-reduced exploration graph of the representative methods in the search results, we determine the center point in the exploration graph based on the distances between representative methods. If the two methods are directly connected, then their distance is computed as Eq. 2. Otherwise, their distance is the sum of the distance of directly connected methods on the shortest path between the two methods. The center point is the method that has the minimum sum distance between the center point and all other methods. This center point is recommended as the starting point of the exploration.

We use a gradient coloring scheme to reflect the degree of differences between the two representative methods. We set two baseline colors (e.g., [0, 0, 255] and [0, 255, 255] in

RGB colorspace) for the center point group and the group with the largest distance to the center point method. Then the color of other nodes is proportionally calculated between the two baseline colors based on the distance between their corresponding methods and the center point method.

To generate intuitive labels for the differences between the two methods, we first merge token-based differences into syntactically complete differences as proposed in [15]. A syntactically complete difference is a differential token sequence pair. For example, some of the token differences from $Snippet$ 2 to $Snippet$ 1 in Fig. 1 can be merged as follows: [$java$, ϵ], [., ϵ], [$util$, ϵ], [., ϵ], [$NoSuchElementException$, ϵ], [e, ϵ] $\longrightarrow$ [$java.util.NoSuchElementExceptione$, ϵ]. For a syntactically complete difference [scs_1, scs_2] from a method mg_1 to another method mg_2, we generate different kinds of labels for scs_1 and scs_2 respectively by extracting specific content from them, including:

- API or local class and method call, e.g., $StringBuilder$;
- conditional statement, e.g., $if(BufferedReader.readLine() == null)$;
- loop statement, e.g., $while(BufferedReader.readLine() != null)$;
- switch statement, e.g., $switch(tag)$;
- return statement, e.g., $return\ String$;
- catch exception, e.g., $catch\ IOException$;
- others, e.g., algebraic operation.

If scs_1 (or scs_2) is ϵ, then the labels generated for scs_2 (or scs_1) are put at the end closer to mg_2 (or mg_1) on the edge between mg_1 and mg_2. Otherwise, the labels generated for scs_1 and scs_2 are put in the middle of the edge to highlight their correspondence.

IV. Tool Support

We have implemented our approach in a web-based exploratory code search tool (called $CodeNuance$). It integrates a code base of 2,786,631 Java source files from 207,711 Java projects crawled from Github [3]. These source files declare 9,085,129 methods in total. The corpus of the methods is indexed by Apache Lucene [18] for searching.

$CodeNuance$ implements keyword-based code search using Apache Lucene. After the user inputs a set of keywords and clicks the "Start" button, $CodeNuance$ starts an exploratory code search process. It analyzes the top ranked N (e.g., 200) methods in the search results. $CodeNuance$ integrates CCFinderX [16], [17] to detect Type-I/II clones in these methods. After code differencing and exploration edge reduction, it generates an exploration graph for the user to interact with the search results.The current implementation uses the following parameter settings: $Th_{trans} = 0.8, Th_{sim} = 0.8$.

When visualizing the exploration graph, we use the force-directed graph provided by Gallery [19] to ensure that the lengths of the edges are nearly equal and the crossing edges are as few as possible. In the graph, each node represents the representative method of a method group. The starting point method is highlighted in larger node size and its code is shown besides the exploration graph. The node colors indicate the relative difference of a method to the starting point method. The method differences are annotated on the corresponding edges. When clicking a method node, its code is shown in the text box besides the exploration graph.

As shown in Fig. 1 and described in Section II the user starts exploration from the recommended starting point. He/she clicks the node and checks the method code. If the current method does not satisfy the requirement, the user can explore to one neighboring node with the prompts of differences labels. This process repeats until the user finds satisfied results or decides to restart code search with reformulated query.

V. Evaluation

The aim of our approach is improving code search via the analysis and visualization of code commonalities, differences and relationships present in the search results. To evaluate our approach and tool towards this goal, we conducted a series of empirical studies to answer the following research questions:

- RQ1: *Does exploration graph reduce the result examination effort in code search?*
- RQ2: *Can CodeNuance help users find desired code faster and more accurately, compared with other exploratory code search tool?*
- RQ3: *How do the edge reduction and difference labels influence users' code search experience?*

All code search tasks with their ground-truth methods and the detailed results of our experimental studies can be found in our replication package [12].

A. Study Design

The code search tasks used in our study were created based on Stack Overflow [20] questions. We ranked all the Stack Overflow questions with tag "java" according to their votes and chose the top 100 implementation-related questions. An implementation-related question asks about the implementation of specific functionalities, e.g., *How do I convert an InputStream to a String in Java?*[2].

For each of the 100 questions, we created a code search task with the question title and question body as the task description. To identify the ground-truth methods in our code base for a code search task, i.e., the methods that can satisfy the task description, we collected the code examples in the top-5 voted answers to the corresponding question and used these code examples as queries to search our code base. Two of the authors carefully examined the returned results and determined the ground-truth methods for each code search task. The ground-truth methods in our code base may be syntactically different but must be semantically equivalent to the code examples from the question answers.

To answer RQ1, we conducted an analytical study in which we compared our $CodeNuance$'s exploration graph for search results examination with ranked-list based examination and an FCA (Formal Concept Analysis) based tree view of code clusters for result examination [9]. We approximate user effort

[2]http://stackoverflow.com/questions/1763789

in terms of the number of candidate methods that need to be examined to find the first ground-truth method.

To answer RQ2, we implemented a keyword recommendation based exploratory code search tool called $KeywordRec$ and conducted a user study in which we compared $CodeNuance$ with $KeywordRec$. 18 participants were asked to use $CodeNuance$ and $KeywordRec$ to complete 10 code search tasks selected from the 100 code search tasks. We analyzed the task completion time and success rate using different tools and the behavior differences in exploratory code search with the two tools.

To answer RQ3, we conducted a user study to investigate the influence of edge reduction and difference labels on the tool usage and user experience. Five participants were asked to complete three code search tasks with an adapted version of $CodeNuance$ that allows users to adjust edge reduction ratio. We compared the final exploration graph the participants customized for each task with the default exploration graph. We also analyzed the participants' feedbacks on the visualization and interaction design of exploration graph.

B. Analytical Study - Result Examination Effort (RQ1)

1) Method: RQ1 provides a theoretical analysis of the result examination effort using different result representations. We implemented an IR based code search method using Apache Lucene, which returns a ranked list of candidate methods for a given code search task description. We use CCFinderX [17] to group Type-I/II cloned methods in the search results. For Type-I/II cloned methods, only one of the cloned methods need to be examined, as others will provide little or no new information. So we create a clone-reduced ranked list of candidate methods. For each method group, only one method is retained in the clone-reduced list. In this study, we analyze the result examination effort for both the original search results and the clone-reduced search results.

The FCA based approach follows the concept location approach proposed in [9] to generate a tree view for a list (original or clone-reduced) of candidate methods returned by the IR based code search. This approach selects the top n ranked methods and the k most relevant attributes from these n methods to generate a concept lattice using FCA. Each folder in the tree view represents a concept node in the lattice. Based on the empirical results reported in [9], we set $n = 100$ and $k = 15$ for constructing the concept lattice. For a fair comparison, we also let $CodeNuance$ select the top 100 ranked methods to generate the exploration graph. Among our 100 code search tasks, the IR based code search does not return the ground-truth methods within the top 100 ranked candidate methods for 17 tasks. Considering our experimental setting, we ignored these 17 tasks in this analytical study.

Given the search results for a code search task, we approximate user effort in terms of the number of methods that the user needs to examine in order to find a ground-truth method. We assume an "ideal" user would examine the search results with a ranked list, a FCA-based tree view or our exploration

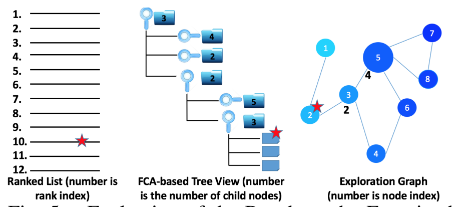

1.
2.
3.
4.
5.
6.
7.
8.
9.
10.
11.
12.
Ranked List (number is rank index)
3
4
2
2
5
3
FCA-based Tree View (number is the number of child nodes)
1
2
3
4
5
6
7
8
4
2
Exploration Graph (number is node index)

Fig. 5. Evaluation of the Results to be Examined

graph as shown in Fig. 5. The star in the figure indicates the position of a ground method in different result representation.

- For the ranked list of candidate methods, the user would examine every method in the ranked list until the first ground-truth method is reached. In Fig. 5, the ground-truth method is ranked at the 10th position, so the result examination effort is estimated as 10.
- For the FCA-based tree view, the result examination effort would be the sum of the numbers of nodes (except the root) that need to be examined at each level until the first ground-truth method is found in a node. It is assumed that the user will expand the intermediate tree node whose subtree contains the first ground-truth method. In Fig. 5, the result examination effort is estimated as 3 (2nd level) + 2 (3rd level) + 3 (4th level) = 8.
- For $CodeNuance$'s exploration graph, the result examination effort would be the sum of the numbers of nodes that need to be examined from the starting point to the first ground-truth method. It is assumed that the user will examine all neighboring nodes of a node and then choose a node on the shortest path to reach the ground-truth method. In Fig. 5, the user first examines the node 5 (the starting point) and then the four neighboring nodes of the node 5. He will choose the node 3 and examine the two neighboring nodes of the node 3 and find the node 2 (the ground-truth method). Thus, the result examination effort is $1 + 4 + 2 = 7$.

2) Results: Type-I/II clones exist in the top 100 ranked methods for all the 83 code search tasks we studied. The size of the clone groups is usually small. Most of the clone groups contain two to four cloned methods. Replacing clone groups with one of the methods in the groups helps reduce the number of candidate methods in the ranked list. The clone-reduced ranked lists contain 25 to 95 (69.95 on average) methods, which indicates that candidate methods for some tasks are more duplicated than others. The presence of method clones in the search results would affect the examination effort.

Fig. 6 presents the descriptive statistics of the result examination effort of the three approaches. For the ranked list based examination and the FCA-based tree view, we considered two treatments: one is the original search results (denoted as RList-Original and FCA-TV-Original); the other is the clone-reduced search results (denoted as RList-NoClone and FCA-TV-NoClone). It can be seen that the result examination effort can be significantly reduced by grouping Type-I/II clones and examining only one of the cloned methods. Overall, providing certain type of structured view over the search results can significantly reduce the result examination effort, with the median=4 and mean=4.49 for our exploration graph,

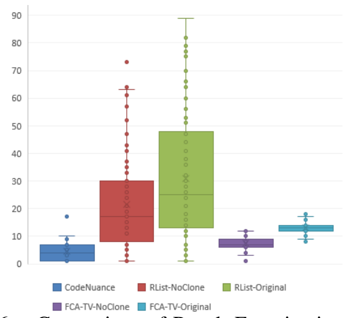


Fig. 6. Comparison of Result Examination Effort

and the median=7 and mean=7.41 for the FCA-TV-NoClone, compared with the median=17 and mean=21.62 for the RList-NoClone. Comparing the two structured views, our exploration graph requires the least result examination effort.

We manually inspected the exploration graphs and the FCA-based tree views for the 83 code search tasks. We found that for 28 tasks the starting point in the exploration graph was exactly the ground-truth method. This demonstrates the effectiveness of our strategy for determining the starting point method which is not based on the query-code relevance but the analysis of code differences between the starting point method and other methods in the search results. In other tasks, the ground-truth methods can usually be found by exploring several neighboring nodes around the starting point. This shows that the exploration edge reduction can reduce unnecessary edges to avoid meaningless exploration, and at the same time ensure the connectivity required for effective exploration.

In contrast, the FCA-based tree view usually provides several to dozens of categories at each level, each of which represents some common attributes shared by the methods in the category. Furthermore, category attributes often largely overlap, which makes the exploration of the FCA-based tree view less effective. This reflects the key difference between our $CodeNuance$ and FCA-based approaches [9]: $CodeNuance$ considers both code commonalities and differences among candidate methods and analyzes the method relationships through their differences, while FCA-based approaches consider only the commonalities among candidate methods.

C. User Study - $CodeNuance$ vs. $KeywordRec$ (RQ2)

1) Method: In the RQ2, we implemented an exploratory code search tool called $KeywordRec$ based on the query reformulation technique implemented in $CodeExchange$ [7]. $CodeExchange$ provides Web-based online service[3] and supports class-based code search. To provide a baseline tool that supports method-based code search and uses the same code base with $CodeNuance$ we chose to implement a local version of $CodeExchange$ that recommends keywords based on the most frequent variable names in the results and uses similar UI layout to $CodeExchange$.

We recruited 18 students from our school as the participants, including 13 master students, and 5 PhD students. These participants were divided into two comparable groups (PG1 and PG2) based on our pre-experiment survey about their programming experience. We randomly selected 10 tasks from the 100 code search tasks. These tasks were randomly divided into two groups (TG1 and TG2), each with 5 tasks.

[3]CodeExchange: http://codeexchange.ics.uci.edu

Before the experiment, we gave a 15-minute tutorial for each tool with live demonstration of the tool features. We randomly selected two warm-up tasks from the rest of 90 code search tasks. After the tutorial, the participants practiced the two tools using the two warm-up tasks until they find a ground-truth method for the tasks with the tools. We answered the participants' questions regarding the tool usage.

In the experiment, the participants of PG1 first used $CodeNuance$ to complete the tasks in TG1 and then used $KeywordRec$ to complete the tasks in TG2. The participants of PG2 first used $KeywordRec$ to complete the tasks in TG1 and then used $CodeNuance$ to complete the tasks in TG2. For each task the participants were asked to find a method that best matches the task. The participants were given 10 minutes for each task, which means they failed to complete a task if they cannot find a ground truth method in 10 minutes. During the whole experiment, the participants were required to run a full-screen recorder to record their code search processes.

2) Results: The performance of the 18 participants in the five tasks (T1-T5) in TG1 is shown in Table I, and that in the other five tasks (T6-T10) in TG2 is shown in Table II. The two tables provide the task completion time in seconds. They also show the success rate (Succ.) and the average task completion time (Avg.Time) of each group of participants for each task. Note that there may be multiple ground-truth methods that satisfy the task description. Identifying any of them is considered a task success. In the tables, "/" means that the participants failed to find any relevant method in 10 minutes, and "$\times$" means that the participants identified a method for the task but the method was incorrect. Both cases are considered as a task failure. If a method identified by the participants implements only a part of the task, it was regarded as partially correct (indicated by "*") and counted as 0.5 in success rate calculation.

Table I shows that the success rate and average time of PG1 (using $CodeNuance$) for T1-T5 are 77% and 124 seconds, while those of PG2 (using $KeywordRec$) are 72% and 138 seconds. Table II shows that the success rate and average time of PG2 (using $CodeNuance$) for T6-T10 are 60% and 127 seconds, while those of PG1 (using $KeywordRec$) are 39% and 175 seconds. These results suggest that compared with $KeywordRec$, the participants can find desired code faster and more accurately when using $CodeNuance$. The results show that the performance in TG2 tasks is relatively worse than that in TG1 tasks. According to the participants' feedbacks TG2 tasks are more difficult than TG1 tasks in that the right queries of desired methods can not be easily identified from the task descriptions. Thus it can be seen that the advantage of $CodeNuance$ is more significant in more exploratory tasks.

To understand the user behavior differences in using the two tools, we analyzed the participants' task completion videos and their post-experiment survey feedbacks, and interviewed

TABLE I
CODE SEARCH PERFORMANCE OF TG1 (T1-T5)

	T1	T2	T3	T4	T5	Mean
P1	217	/	327*	/	141*	-
P2	96	72×	117	95	83	-
P3	76	44*	31*	112	87	-
P4	105*	62	110	93*	40	-
P5	66	330	192	102	72	-
P6	154	146	123*	286*	57	-
P7	141	109*	161	219*	/	-
P8	140	210	112	/	45	-
P9	36	154	43	203*	34*	-
CodeNuance.Avg.Time	115	141	135	159	70	124
CodeNuance.Succ.	94%	67%	89%	56%	78%	77%
P10	63	143×	58	100	122	-
P11	205	/	81	190	61	-
P12	132	/	162	231*	/	-
P13	332×	117	138	203*	343	-
P14	305*	158	128	130	220	-
P15	223	149	65*	70*	117	-
P16	60	180*	30	120	120	-
P17	110	166*	58	180×	100	-
P18	52	163×	60×	93×	38	-
KeywordRec.Avg.Time	165	154	87	146	140	138
KeywordRec.Succ.	83%	44%	83%	61%	89%	72%

TABLE II
CODE SEARCH PERFORMANCE OF TG2 (T6-T10)

	T6	T7	T8	T9	T10	Mean
P1	/	100	47*	76	107	-
P2	/	/	75*	/	191×	-
P3	193*	172×	86*	196	/	-
P4	570×	135×	453	/	/	-
P5	294×	100	320	185	91	-
P6	311×	72	110×	105	157	-
P7	/	27×	/	280×	/	-
P8	205×	/	150	163	/	-
P9	253×	62	234×	167×	77*	-
KeywordRec.Avg.Time	304	95	184	167	125	175
KeywordRec.Succ.	6%	44%	50%	56%	39%	39%
P10	179	168	144	65*	134*	-
P11	/	118	98×	/	319	-
P12	78	83	49*	101	143	-
P13	/	91	171	102	159*	-
P14	30	73×	50	92×	123×	-
P15	357×	111	244×	250	83×	-
P16	123	258	118	95	265	-
P17	/	55	/	180	/	-
P18	20	23	92×	32×	42×	-
CodeNuance.Avg.Time	131	109	121	115	159	127
CodeNuance.Succ.	56%	89%	50%	61%	44%	60%

the participants. Overall, the participants feel that the code differences and relationships that $CodeNuance$ presents are more integral and straightforward to understand and use for search results exploration. Furthermore, $KeywordRec$ leaves the user to manually find the differences among all pieces of code. Compared with that, the participants appreciate more the explicit difference labels that $CodeNuance$ presents for similar methods, because it is usually these differences, not the commonalities that affect their decisions on whether a piece of code meets their needs or not.

D. User Study - Edge Reduction and Difference Labels (RQ3)

1) Method: Considering the importance of the method differences and relationships for search results exploration, we would like to further investigate the influence of edge reduction and difference labels on tool usage and user experience. We recruited four master students and one PhD student from our school as the participants and asked them to complete three tasks (different from the 10 tasks in RQ2) randomly selected from the 100 code search tasks. For each task the participants used an adapted version of $CodeNuance$ to adjust the edge reduction ratio between the two extremes, i.e., a minimum connected subgraph and a fully connected graph. This is implemented by dynamically tuning the thresholds used in edge reduction.

The participants were told to find a preferred edge reduction ratio by which the generated exploration graph has the most appropriate number of exploration edges and difference labels for them to obtain an overview of the candidate methods and find the desired code. After they finished the tasks, we compared the participants' preferred exploration graph with the default exploration graph $CodeNuance$ generates. We also interviewed the participants to learn their thoughts and feedbacks on edge reduction and difference labels.

2) Results: Table III shows the numbers of edges in the default exploration graph and the number of edges in the participants' preferred exploration graph for the three tasks. #Node shows that the clone-reduced search results have 83, 86 and 83 methods for the three tasks, respectively. We can see that for all the three tasks the preferred exploration graphs have few edges than the default graphs. Our interviews with the participants suggest that this is partially caused by the overlapping difference labels on the edges. The participants preferred an exploration graph in which they can clearly see difference labels. If there are many overlapping labels, they often chose to eliminate more edges to make the graph clearer. Or they may ignore difference labels and just follow the edges to examine neighboring methods of a method.

According to the participants' feedbacks, although the d-ifference labels accurately reflect the differences between candidate methods, they preferred more abstract and shorter difference labels. They suggested that the tool should allow them to filter difference labels by difference types. For ex-ample, sometimes they are only concerned with API usage or exception handling differences. In such cases, showing all differences is meaningless. They also suggested that to explore neighboring methods by exploration edges more effectively, the tool should provide a code comparison box for the two methods when they click an edge. This will reduce their dependence on edge labels for understanding the method differences.

TABLE III
PREFERRED EDGE REDUCTION RATIO

Task	#Node	#Edge	P1	P2	P3	P4	P5	Avg.
T1	83	146	109	82	114	82	82	93.8
T2	86	136	116	91	130	92	91	104
T3	83	157	95	100	147	92	92	105.2

E. Threats to Validity

A general threat to the validity of our empirical studies lies in the code search tasks and ground truth. They were manually created based on highly voted Stack Overflow questions, thus may not reflect the characteristics of every type of code search tasks. The ground-truth methods were selected based on our understanding, which may not always be correct.

Specific threats to the validity of individual studies lie in the first two studies for RQ1 and RQ2. A main threat to the validity of the analytical study lies in the queries we created for the code search tasks. The numbers of results to be examined of the approaches may be different with different queries. A main threat to the validity of the comparison study between $CodeNuance$ and $KeywordRec$ lies in our implementation of $KeywordRec$ and the limited number of tasks. $KeywordRec$ may not fully implement the capability of query reformulation based exploratory code search. And the randomly selected 10 tasks may not provide a comprehensive comparison for $CodeNuance$ and $KeywordRec$.

VI. Discussion

The current edge labels in an exploration graph are extracted from code differences, which may be not intuitive enough for the user to determine the right directions of exploration. For example, the labels shown in the exploration graph in Fig. 2 can only tell $Snippet$ 3 uses *StringBuilder* while $Snippet$ 4 uses *StringBuffer*. However, the key to understand this difference is the knowledge that *StringBuffer* is synchronized and thread-safe while *StringBuilder* is not. This knowledge can only be obtained from external resources such as API specifications and answers on Stack Overflow. To generate more intuitive labels, it is desirable that the knowledge behind the code differences can be extracted from external resources and highlighted in corresponding exploration edges. The challenge is how to process extremely large volume of unstructured text of various kinds and synthesize descriptive labels that well reflect the user's concerns about code differences.

The topology of the exploration graph can be more interactive and exploratory. Currently the topology is generated from a fixed size of search results (e.g., the top 100 ranked methods), which may limit the user's capability of exploring unknown space. Furthermore, the scale (i.e., the number of nodes and edges) of exploration graph is fixed, while the user may want to zoom in to learn more subtle differences among several methods or zoom out to learn more macro differences among different method clusters. We will improve the interaction design of exploration graph by supporting the exploration mode that has been popularly implemented in map views. First, the exploration graph can present a wide range of search results to allow the user to explore to an "unknown region" far from the starting point. Second, the tool supports the user to zoom in/out the exploration graph by dynamically tuning the number of nodes and edges shown in the graph.

VII. Related Work

Many code search tools have been developed, which exploit a wide range of software data and program information, such as test cases [21], API documentation [22], program transformation [23], and input-output constraints [24]. Shepherd et al. [25] developed a code search framework based on a general component structure underlying the existing approaches. Ge et al. [26] conducted a field study of the usage of several query recommendation techniques. All these code search tools assume a query-response paradigm, in which they take as input a user query and return a list of ranked code.

Some code search systems support query reformulation by analyzing query or search results' properties. Haiduc et al. [27] developed a tool that can detect the quality of a query and propose query reformulations. Lemos et al. [28] proposed an automatic query expansion approach that uses word relations to increase the chances of finding relevant code. Martie et al. [7] supported query reformulation by recommending frequent words in variable names, tuning code properties (e.g., length), and specifying language concepts (e.g., method calls). These approaches make no consideration of commonalities, differences and relationships among code in search results. Some feature location (or concept location) approaches [9], [10] cluster code in search results and generate category hierarchies to support interactive search results exploration. These approaches focus on the extraction of the commonalities among code rather than the subtle differences.

Some code search systems and program comprehension approaches consider the structural dependencies among code in search results. Bajracharya et al. [4] extracted fine-grained structural information from code to support results ranking. Thummalapenta and Xie [29] gathered relevant code samples from code search engine and analyzes the samples to suggest relevant method invocation sequences for a query. McMillan et al. [5] proposed an approach for finding highly relevant software projects based on the descriptions of applications, API calls used by applications and the dataflow among those API calls. Chan et al. [30] helped users find usages of APIs by analyzing an API invocation graph and finding an optimum connected subgraph that meets users' search needs. McMillan et al. [6] retrieved and visualized relevant functions and their usages based on the analysis of user navigation behaviors and program dependencies. Robillard [31] proposed a technique to automatically propose and rank program elements for program investigation based on an analysis of the topology of element dependencies. Bohnet and Döllner [32] provided an interactive multi-view visualization of function call graph to support users in locating and understanding feature implementation. Wang et al. [33] enabled developers to make queries involving dependence conditions and textual conditions on program dependence graph. These approaches consider the structural dependencies among code search results, while our approach highlights the differences and relationships among similar code in search results to support exploratory code search.

VIII. Conclusion

This paper presents a novel exploratory code search approach and tool that turn unknowns (i.e., code commonalities, differences and relationships) in search results into explicit visual cues in an exploration graph to help developers navigate and examine the search results. Our evaluation shows that our approach can turn an ad-hoc search results exploration into a much more informed and structured exploration. In the future, we will enhance our approach by incorporating external resources (e.g., API specifications and Stack Overflow answers) into the process of search results exploration, and by improving the information presentation and interaction design.

Acknowledgment

This work is supported by National Key Research and Development Program of China under Grant No. 2016YFB1000801, National Natural Science Foundation of China under Grant No. 61370079, 61402113.

References

[1] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer, "Two studies of opportunistic programming: interleaving web foraging, learning, and writing code," in *Proceedings of the 27th International Conference on Human Factors in Computing Systems, CHI 2009, Boston, MA, USA, April 4-9, 2009*, 2009, pp. 1589–1598.

[2] H. Li, Z. Xing, X. Peng, and W. Zhao, "What help do developers seek, when and how?" in *20th Working Conference on Reverse Engineering, WCRE 2013, Koblenz, Germany, October 14-17, 2013*, 2013, pp. 142–151.

[3] "Github," https://github.com.

[4] S. K. Bajracharya, T. C. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. V. Lopes, "Sourcerer: a search engine for open source code supporting structure-based search," in *Companion to the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, 2006, pp. 681–682.

[5] C. McMillan, M. Grechanik, D. Poshyvanyk, C. Fu, and Q. Xie, "Exemplar: A source code search engine for finding highly relevant applications," *IEEE Trans. Software Eng.*, vol. 38, no. 5, pp. 1069–1087, 2012.

[6] C. McMillan, D. Poshyvanyk, M. Grechanik, Q. Xie, and C. Fu, "Portfolio: Searching for relevant functions and their usages in millions of lines of code," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 4, pp. 37:1–37:30, 2013.

[7] L. Martie, T. D. LaToza, and A. van der Hoek, "CodeExchange: Supporting reformulation of internet-scale code queries in context," in *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, 2015, pp. 24–35.

[8] "Open Hub," https://www.openhub.net.

[9] D. Poshyvanyk, M. Gethers, and A. Marcus, "Concept location using formal concept analysis and information retrieval," *ACM Trans. Softw. Eng. Methodol.*, vol. 21, no. 4, pp. 23:1–23:34, 2012.

[10] J. Wang, X. Peng, Z. Xing, and W. Zhao, "Improving feature location practice with multi-faceted interactive exploration," in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, 2013, pp. 762–771.

[11] G. Marchionini, "Exploratory search: from finding to understanding," *Commun. ACM*, vol. 49, no. 4, pp. 41–46, 2006.

[12] "CodeNuance replication package," https://codenuance2017.wixsite.com/codenuance.

[13] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," *School of Computing TR 2007-541, Queens University*, 2007.

[14] Y. Lin, Z. Xing, Y. Xue, Y. Liu, X. Peng, J. Sun, and W. Zhao, "Detecting differences across multiple instances of code clones," in *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, 2014, pp. 164–174.

[15] Y. Lin, X. Peng, Z. Xing, D. Zheng, and W. Zhao, "Clone-based and interactive recommendation for modifying pasted code," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, 2015, pp. 520–531.

[16] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Trans. Software Eng.*, vol. 28, no. 7, pp. 654–670, 2002.

[17] "CCFinderX," http://www.ccfinder.net/ccfinderxos.html.

[18] "Apache Lucene," https://lucene.apache.org.

[19] "Gallery," https://github.com/d3/d3/wiki/Gallery.

[20] "Stack Overflow," http://stackoverflow.com.

[21] O. A. L. Lemos, S. K. Bajracharya, J. Ossher, R. S. Morla, P. C. Masiero, P. Baldi, and C. V. Lopes, "Codegenie: using test-cases to search and reuse source code," in *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*, 2007, pp. 525–526.

[22] S. Chatterjee, S. Juvekar, and K. Sen, "SNIFF: A search engine for java using free-form queries," in *Fundamental Approaches to Software Engineering, 12th International Conference, FASE 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, 2009, pp. 385–400.

[23] S. P. Reiss, "Semantics-based code search," in *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, 2009, pp. 243–253.

[24] K. T. Stolee, S. G. Elbaum, and D. Dobos, "Solving the search for source code," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 3, pp. 26:1–26:45, 2014.

[25] D. C. Shepherd, K. Damevski, B. Ropski, and T. Fritz, "Sando: an extensible local code search framework," in *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012*, 2012, p. 15.

[26] X. Ge, D. C. Shepherd, K. Damevski, and E. R. Murphy-Hill, "How developers use multi-recommendation system in local code search," in *IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2014, Melbourne, VIC, Australia, July 28 - August 1, 2014*, 2014, pp. 69–76.

[27] S. Haiduc, G. D. Rosa, G. Bavota, R. Oliveto, A. D. Lucia, and A. Marcus, "Query quality prediction and reformulation for source code search: the refoqus tool," in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, 2013, pp. 1307–1310.

[28] O. A. L. Lemos, A. C. de Paula, F. C. Zanichelli, and C. V. Lopes, "Thesaurus-based automatic query expansion for interface-driven code search," in *11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India*, 2014, pp. 212–221.

[29] S. Thummalapenta and T. Xie, "Parseweb: a programmer assistant for reusing open source code on the web," in *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*, 2007, pp. 204–213.

[30] W. Chan, H. Cheng, and D. Lo, "Searching connected API subgraph via text phrases," in *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012*, 2012, p. 10.

[31] M. P. Robillard, "Automatic generation of suggestions for program investigation," in *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, 2005, pp. 11–20.

[32] J. Bohnet and J. Döllner, "Visual exploration of function call graphs for feature location in complex software systems," in *Proceedings of the ACM 2006 Symposium on Software Visualization, Brighton, UK, September 4-5, 2006*, 2006, pp. 95–104.

[33] X. Wang, D. Lo, J. Cheng, L. Zhang, H. Mei, and J. X. Yu, "Matching dependence-related queries in the system dependence graph," in *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*, 2010, pp. 457–466.