Searching Connected API Subgraph via Text Phrases

Wing-Kwan Chan, Hong Cheng
Department of Systems Engineering and Engineering Management
The Chinese University of Hong Kong
{chanwk, hcheng}@se.cuhk.edu.hk

David Lo
School of Information Systems
Singapore Management University
davidlo@smu.edu.sg

ABSTRACT

Reusing APIs of existing libraries is a common practice during software development, but searching suitable APIs and their usages can be time-consuming [6]. In this paper, we study a new and more practical approach to help users find usages of APIs given only simple text phrases, when users have limited knowledge about an API library. We model API invocations as an API graph and aim to find an optimum connected subgraph that meets users' search needs.

The problem is challenging since the search space in an API graph is very huge. We start with a greedy subgraph search algorithm which returns a connected subgraph containing nodes with high textual similarity to the query phrases. Two refinement techniques are proposed to improve the quality of the returned subgraph. Furthermore, as the greedy subgraph search algorithm relies on online query of shortest path between two graph nodes, we propose a space-efficient compressed shortest path indexing scheme that can efficiently recover the exact shortest path. We conduct extensive experiments to show that the proposed subgraph search approach for API recommendation is very effective in that it boosts the average F_1-measure of the state-of-the-art approach, *Portfolio* [15], on two groups of real-life queries by 64% and 36% respectively.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Management – *Productivity*; D.2.13 [**Software Engineering**]: Reusable Software – *Reusable libraries*

Keywords

API graph, API recommendation, Subgraph searching

1. INTRODUCTION

Application Programming Interfaces (APIs) from third-party libraries provide many of the needed functions for programmers. Re-implementing these functions from scratch is obviously very costly in both time and money. Besides, these APIs are usually reliable and bug-free since they are well tested with many test cases before releasing to the public. Therefore, reusing APIs of existing libraries is a common practice during software development. If a programmer locates suitable APIs and discovers rules in using them, he/she can accomplish a task efficiently. However, this is not an easy task especially when the API library is large and complex [20]. For example, the object-oriented library - Java^{TM} Platform Standard Edition v1.6 (JSE 1.6) provides more than 3,100 classes and 28,900 methods. From the study of query logs of a search engine in [6], developers spend much time and effort in searching related APIs. It is because the learning curve of re-using APIs can be steep due to several barriers. Two of them are selection barrier and coordination barrier [10]. First, a programmer attempts to overcome selection barrier by finding suitable APIs that are appropriate for a particular task. Afterwards, the programmer attempts to overcome coordination barrier to find the usages of the selected APIs by referring to sample code or method invocations in the documentation.

There are many API (and Code) recommendation methods to facilitate re-using APIs (Code). Some approaches (Group 1) such as [16, 6, 18, 1] aim at solving selection barrier only, but they enable users to use high level natural language as the query. Other approaches (Group 2) such as [25, 14] aim at solving coordination barrier by providing method invocation context, but they start with exact signature of method(s). A user may want to pass through both barriers at the same time, and a simple way is to glue existing approaches. But this is time-consuming as a user has to decide suitable functions to be the inputs of a Group 2 approach, after issuing a series of queries in natural language to a Group 1 approach. In this paper, we consider a new attempt for helping a user to pass through both barriers with minimum human interventions. The proposed approach allows a user to use a set of short text phrases as the query, and returns a connected subgraph where its nodes represent classes or methods exhibiting high textual similarity with the query phrases, and its edges indicate the invocation relationship between the nodes. We first model classes and methods in an API library as well as their invocation relationships as an API graph. Given a query, after locating the initial API candidates (methods or classes) based on textual similarity, we try to find an optimum subgraph with the highest score from the API graph among all possible connected subgraphs. A possible connected subgraph is a graph where each phrase in the query can be similar to

SIGSOFT'12/FSE-20, November 11–16, 2012, Cary, North Carolina, USA.

at least one of its nodes (methods or classes). A subgraph scores higher if the accumulated textual similarity is higher while the total number of nodes is smaller.

Let us first illustrate the benefits and challenges of the new problem setting with a sample query Q consisting of 3 short phrases q_0 = 'database connection', q_1 = 'create sql statement' and q_2 = 'get result'. These 3 phrases in Q concisely describe that the goal of the task is *'connect to database, issue a SQL and get the result'*. Figure 1 shows the query Q together with a small API graph [1] constructed from a snapshot of 4 classes and 8 methods in java.sql package of JSE 1.6. The dashed edges link each phrase in Q to corresponding candidate classes/methods nodes, if their textual similarity is above a certain threshold. The optimum subgraph for Q only consists of nodes 0, 3, 4 and 6, illustrating a typical API usage w.r.t. the query Q, i.e., a *Connection* object invokes its member function *createStatement()*, which then returns a *Statement* object. The *Statement* object then invokes its member function *getResultSet()*.

The new problem setting has two benefits. First, forming a high-level query consisting of phrases in natural language is flexible because it allows a user to issue the query without getting familiar with the names of methods/classes. Second, a connected subgraph indicates clear relationships among methods/classes of interest. On the other hand, it has two challenges which are mainly related to finding connected subgraphs. First, an API library is usually large and complex. As a result, the constructed API graph usually forms a huge search space. Exhaustively enumerating all the possible connected subgraphs is not feasible due to exponential runtime when the number of phrases in Q increases, especially given a low textual similarity threshold. Second, existing greedy subgraph search algorithms [11, 9] rely on online query of shortest path between two graph nodes. It is necessary to precompute and store all pairwise shortest paths between graph nodes in memory to support efficient subgraph search with a small distance. However, it takes at least $O(n^2)$ space to index all pairwise shortest paths for a graph with n nodes. When an API graph contains a large number of nodes, this is prohibitively expensive.

At a first glance, we can reuse an existing code recommendation approach *Portfolio* [15] to solve the new problem (details of the approach are in Section 5). However, it does not guarantee a connected subgraph since it only outputs top-k most similar nodes to the query. For example, for the query in Figure 1, if $k = 4$, nodes 1, 3, 6 and 11 will be returned. But these isolated nodes do not reveal the invocation relationships among methods and classes, thus they are less useful. In contrast, in our problem formulation, we bear the consideration of subgraph connectivity in mind and aim to find a connected subgraph such that it has high textual similarity to the query phrases, contains very few irrelevant nodes and shows clear invocation relationships among methods and classes. Our contributions include:

1. We start from a greedy subgraph search algorithm which was originally designed for the team formation problem in an expertise network [11]. We identify two limitations in this basic solution and propose refinement techniques to improve the quality of the subgraph for API recommendation. The improved solution has the same time complexity of $O(n^2)$ as that of the original approach, where n is the number of nodes in the API graph.
2. To enable efficient online query of shortest paths between graph nodes, we propose a space-efficient domain-specific indexing scheme that only indexes pairwise shortest paths between classes. The indexing scheme can recover the exact shortest path efficiently with two supplementary indexing structures. It is applicable to any object-oriented libraries.
3. We conducted extensive experiments using two groups of real-life queries to compare the proposed approach with *Portfolio*. The results show that the average F_1-measure of the proposed approach is 64% (Group I) and 36% (Group II) higher than that of *Portfolio*.

The rest of the paper is organized as follows. Section 2 defines preliminary concepts and the problem statement. Section 3 describes the basic greedy subgraph search algorithm for finding a connected subgraph and our proposed refinement techniques. Section 4 introduces the space-efficient shortest path indexing scheme. We present our experimental results in Section 5, and discuss related work in Section 6. Finally, Section 7 concludes our study.

2. PROBLEM STATEMENT

There are two kinds of components in an object-oriented API library, namely (1) classes/interfaces, denoted as C; and (2) methods and constructors, denoted as M. The sizes of C and M are $|C|$ and $|M|$ respectively. Besides, there are four kinds of relationships between classes and methods: (1) inheritance of a child class from its parent, denoted as Inh; (2) a class to its member methods, denoted as Mem; (3) an input parameter to a method (if any), denoted as Inp; and (4) a method to its output parameter (if any), denoted as Out. We use $Rel = \{Inh, Mem, Inp, Out\}$ to denote these four kinds of relationships among C and M. An *API graph* defined below can be constructed from an object-oriented API library.

DEFINITION 1. ***(API Graph)**: An API graph $G = (V_G, E_G)$ is an unweighted and undirected graph. The node set $V_G = C \cup M$ corresponds to all classes and methods and its size is $|V_G| = |C| + |M|$. Each node $v \in V_G$ is associated with a bag of words BOW_v. There is an edge $e(u, v) \in E_G$ for $u, v \in V_G$ iff $\exists Rel(u, v) \in Rel$.*

Example 1. Figure 1 shows a sample API graph, where the 4 nodes in oval shape are classes in java.sql packages and the other 8 nodes in box shape are member methods of the 4 class nodes. Edges $e(0, 1)$ and $e(1, 4)$ represent the relationships Mem (i.e., method *createStatement()* is a member of class *Connection*) and Out (i.e., the output of method *createStatement()* is class *Statement*) respectively.

DEFINITION 2. ***(Query)**: A query Q consists of a set of required phrases $\{q_i \in Q | 1 \leq i \leq |Q|\}$, where $|Q|$ is the size of the query Q. Each phrase $q \in Q$ is associated with a bag of words BOW_q.*

For each phrase $q \in Q$, we need to pick at least one node from the API graph G with high textual similarity to q in the resultant subgraph. We use Dice's coefficient in Eq. 1 to

[1]For visualization purpose, the API graph shown here is directed. In Section 2 we formally define an API graph as **undirected**.

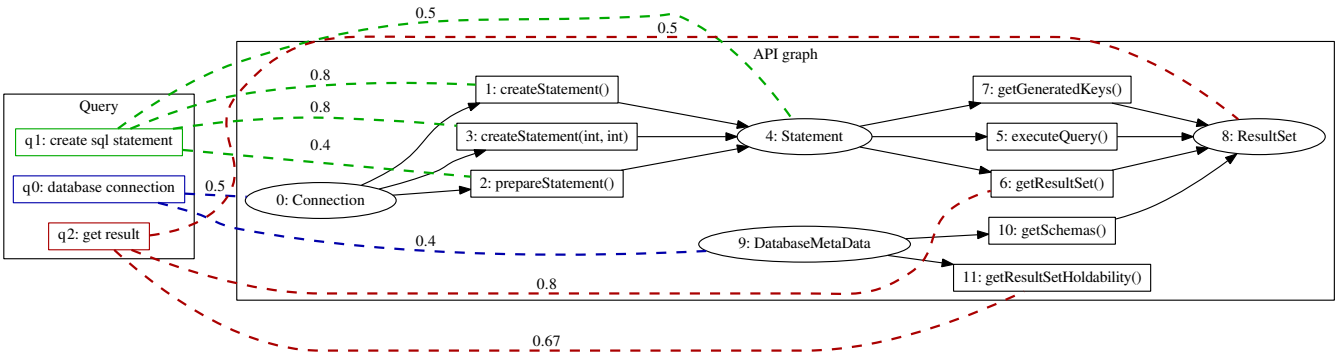


Figure 1: Sample query and API graph from a snapshot of java.sql package (For simplicity and clarity, we only show the input parameters of method nodes 1 and 3)

measure the textual similarity between a phrase $q \in Q$ and a node $v \in V_G$ based on their BOWs[2]. In addition, we set a textual similarity threshold $\alpha \in (0, 1]$. Given a query phrase q, for any $v \in V_G$, if $Sim(q, v) \geq \alpha$, v is a candidate match node for q.

$$Sim(q, v) = \frac{2|BOW_v \cap BOW_q|}{|BOW_v| + |BOW_q|} \quad (1)$$

Next, how to measure the goodness of a subgraph w.r.t. a query Q for API recommendation? Intuitively, if the accumulated textual similarity between nodes in a subgraph G' and phrases in the query is higher, G' is a better choice. For convenience, we refer to the node set in G' with highest similarity to each of the query phrases as *necessary nodes* V_o, and other nodes $V_{G'} \setminus V_o$ as *dummy nodes*. A better subgraph shall contain less dummy nodes that are not textually similar to any phrases in the query. Therefore, we use *Gain* in Definition 3 to measure the goodness of a subgraph.

DEFINITION 3. ***(Gain)**: For a subgraph $G' \subseteq G$, the Gain of G', $Gain(G', Q, \rho)$ w.r.t. a query Q is measured by Eq. 2.*

$$Gain(G', Q, \rho) = \frac{\sum_{q_i \in Q} Sim(q_i, v_i^*)}{|Q| + \rho|V_{G'} \setminus V_o|} \quad (2)$$

where $v_i^* = \arg\max_{v \in V_{G'}} Sim(q_i, v)$ is the best matching node of phrase q_i, $V_o = \{v_i^* | 1 \leq i \leq |Q|\}$ is the set of best matching (necessary) nodes to each of the query phrases, and $\rho \in (0, 1]$ is the dummy node penalty.

Example 2. Consider Figure 1. BOWs of node 1 are [create, statement] and BOWs of phrase q_1 'create sql statement' of the query Q are [create, sql, statement]. Thus $Sim(q_1, v_1) = 0.8$. Given $\alpha = 0.4$ and $\rho = 0.2$, the subgraph in Figure 2(b) has a *Gain* of 0.656 w.r.t. Q.

Now, we formally define our API recommendation in Problem 1, which contains three conditions for an optimum subgraph G' w.r.t. a query.

PROBLEM 1. ***(Searching API Subgraph)**: Given a query Q, an API graph $G = (V_G, E_G)$, a similarity threshold α and a dummy node penalty ρ, the problem of searching API subgraph is to find a subgraph $G'(V_{G'}, E_{G'}) \subseteq G$ s.t. the following three conditions hold:*

1. *G' is connected;*
2. *$\forall q \in Q, \exists v \in V_{G'}$ s.t. $Sim(q, v) \geq \alpha$;*
3. *$Gain(G', Q, \rho) = \max_{H' \subseteq G} Gain(H', Q, \rho)$.*

Condition 1 states that G' shall be connected. Condition 2 states that for every phrase in the query, we can find a node in G' that is textually similar to this phrase by passing the similarity threshold α. Condition 3 states that G' shall have the highest *Gain* among all possible subgraphs satisfying conditions 1 and 2.

3. API SUBGRAPH SEARCH

According to Problem 1, finding a subgraph with the highest *Gain* for the API recommendation task is very hard, as we need to exhaustively enumerate all the possible connected subgraphs consisting of candidate nodes whose textual similarity with the query phrases is above α. This time complexity increases exponentially with the number of phrases in a query Q, especially when α is low, i.e., the number of candidate nodes which pass the similarity threshold is large.

3.1 A Greedy Subgraph Search Algorithm

Considering the prohibitive computational cost in finding the optimum API subgraph, we resort to a greedy algorithm, called *RarestFirst* (RF), which was proposed in the work of Lappas et al. [11]. The problem studied in [11] is, given a task T which requires a set of skills, and an expertise network where each expert (node) possesses one or more skills and there is some communication cost between two experts, the goal is to find a team of experts who can jointly fulfill all required skills in T with the minimum communication cost among them. In our problem setting, *nodes* in an API graph correspond to *experts* and the *query* corresponds to the *task*. Our problem setting is a little different in that we account for textual similarity between nodes (classes or methods) and required phrases of the query. The modified RF method to suit our problem setting is in Algorithm 1.

We explain the main idea of algorithm RF. It first invokes a procedure *FindSupport* (line 1), which, for each query phrase $q \in Q$, finds all candidate match nodes $v \in V_G$ whose textual similarity with q is above α, i.e., $Sim(q, v) \geq \alpha$. The set of candidate match nodes for q is denoted as $S(q)$. Then we find the query phrase q with the smallest cardinality $|S(q)|$ and denote it as q_{rare}, i.e., the query phrase with the smallest number of match nodes. For each candidate $c \in S(q_{rare})$, it finds the closest node $v \in S(q)$ to c for all

[2]BOWs represent textual information for a node or a query phrase. In our implementation, we split the name (identifier) of a node by Camel Case to obtain BOWs for a node.

other $q \in Q$ (lines 2–7). Finally, it picks a $c^* \in S(q_{rare})$ with the smallest diameter to form a connected subgraph (lines 8–9). The subgraph is formed by connecting c^* with the closest node $v_q^* \in S(q)$ for all other $q \in Q$ by the shortest path from c^* to v_q^*. The diameter w.r.t. a center c is defined as

$$diameter(c) = \max_{q \in Q} \min_{v \in S(q)} Dist(v, c) \quad (3)$$

By 'the smallest diameter', we hope the resultant subgraph involves very few dummy nodes that are not similar to any query phrase.

Example 3. Let us reconsider the query Q consisting of 3 short phrases q_0 = 'database connection', q_1 = 'create sql statement' and q_2 = 'get result'. For the API graph in Figure 1, given similarity threshold $\alpha = 0.4$, we compute the candidate match nodes for each phrase in Q as:

$$S(q_0) = \{0, 9\},\ S(q_1) = \{1, 2, 3, 4\},\ S(q_2) = \{6, 8, 11\}$$

Therefore, q_0 is the rarest phrase (q_{rare}) with two candidate match nodes 0 and 9. For node 0, the closest node in $S(q_1)$ is any of nodes 1, 2 and 3, with the same shortest distance of 1; the closest node in $S(q_2)$ is node 6 and the shortest distance is 3. Thus $diameter(v_0) = 3$ according to Eq. 3. Similarly, if we start from node 9, the closest node in $S(q_1)$ is node 4 and the shortest distance is 4; the closest node in $S(q_2)$ is node 11 and the shortest distance is 1. Thus $diameter(v_9) = 4$. Therefore, algorithm RF will pick node 0 as c^* to form an API subgraph, as it has the smallest diameter in $S(q_0)$. Node 0 can also be regarded as the center of the subgraph.

Next, RF will arbitrarily choose any one of nodes 1, 2 and 3 for q_1 as they have the same shortest distance of 1 to node 0. Suppose node 2 is selected and connected to node 0. For q_2, RF will choose node 6, as it is the closest node in $S(q_2)$ to node 0. Note that there are *three shortest paths* from node 0 to node 6: $(0, 1, 4, 6)$, $(0, 2, 4, 6)$ and $(0, 3, 4, 6)$. Suppose RF picks $Path(0, 6) = (0, 3, 4, 6)$ as a shortest path from nodes 0 to 6. The resultant API subgraph is shown in Figure 2(a). When compared with the optimum subgraph in Figure 2(b), the solution of RF contains an extra node 2, which is redundant to node 3, as both of them match query phrase q_1 = 'create sql statement'.

(a) Solution of RF (b) Optimum

Figure 2: The solution of greedy algorithm RF versus optimum solution for the query in Figure 1

3.2 Selecting Node with High Textual Similarity

A problem of RF is that, it considers any node $v \in V_G$ as a candidate match of a query phrase q, as long as $Sim(q, v) \geq \alpha$. For all candidates in $S(q)$, RF does not differentiate them according to the textual similarity; however, some of them may have higher similarity, and some others have lower similarity. In Example 3, RF selects node 2 in the answer subgraph. But $Sim(q_1, v_2) = 0.4$, which is lower than $Sim(q_1, v_1) = Sim(q_1, v_3) = 0.8$. To address this problem, we formally define a new node selection measure, called *LocalGain*, in Definition 4.

Algorithm 1: RarestFirst(Q, G, α)

Input: query Q, API graph $G = (V_G, E_G)$, similarity threshold α
Output: subgraph $g \subseteq G$

1 $q_{rare}, S \leftarrow$ FindSupport(Q, G, α)
2 **foreach** $c \in S(q_{rare})$ **do**
3 **foreach** $q \in Q$ *and* $q \neq q_{rare}$ **do**
4 $R_{cq} \leftarrow \min_{v \in S(q)} Dist(v, c)$
5 **end**
6 $R_c \leftarrow \max_q R_{cq}$
7 **end**
8 $c^* \leftarrow \arg\min R_c$
9 $g \leftarrow c^* \cup \{Path(c^*, v_q^*) | q \in Q, v_q^* = \arg\min_{v \in S(q)} Dist(v, c^*)\}$
10 **return** g

11 **Procedure** FindSupport(Q, G, α)
12 **foreach** $q \in Q$ **do**
13 $S(q) \leftarrow \{v | Sim(q, v) \geq \alpha, \forall v \in V_G\}$
14 **end**
15 $q_{rare} \leftarrow \arg\min_{q \in Q} |S(q)|$
16 **return** $q_{rare}, \{S(q) | q \in Q\}$

Definition 4. ***(Local Gain)**: Given a center c and a query phrase $q \in Q$, for a candidate match node $v \in S(q)$, the Local Gain $Gain_L(v, q, c)$ of connecting v with c is measured by Eq. 4.*

$$Gain_L(v, q, c) = Sim(q, v) - \rho(Dist(v, c) - 1) \quad (4)$$

where $\rho \in (0, 1]$ is the dummy node penalty.

Given a fixed center c, *LocalGain* prefers a node $v \in S(q)$ having a high textual similarity with the phrase q, and having a small distance to c. We add the dummy node penalty ρ for flexibility. With a lower (or higher) value of ρ, we give a lower (or higher) penalty for the distance $Dist(v, c)$ between a node v and the center c. Accordingly, we propose an improved algorithm called *RarestGainFirst* (RGF) in Algorithm 2, which uses the *LocalGain* as the criterion to select the best node v^* from $S(q)$, $\forall q \in Q$ and $q \neq q_{rare}$ (line 5). Then, a shortest path $Path(c, v^*)$ is added to the resultant subgraph to connect v^* to c.

Example 4. Consider Figure 1, for the center node 0 and q_1='create sql statement', $Gain_L(v_1, q_1, v_0) = Gain_L(v_3, q_1, v_0) = 0.8$, $Gain_L(v_2, q_1, v_0) = 0.4$, for any value of ρ. Thus RGF will select node 1 or 3 according to *LocalGain*.

Moreover, algorithm RF finds a subgraph with the smallest diameter, according to Eq. 3. However, the subgraph with the smallest diameter does not necessarily have the highest *Gain* defined in Eq. 2. It is necessary to form a subgraph for each center node $c \in S(q_{rare})$ and return the one with the highest *Gain*. Therefore, algorithm RGF searches a subgraph L_c for each center $c \in S(q_{rare})$. Finally, the subgraph with the highest *Gain* according to Eq. 2 is returned as the answer (lines 9–10).

3.3 Handling Multiple Shortest Paths Problem

Algorithm RGF can find a better subgraph in terms of *Gain* by selecting nodes with high textual similarity for each phrase. However, RGF still faces the problem of *multiple shortest path*, i.e., the shortest path between two nodes in a

Algorithm 2: RarestGainFirst(Q, G, α, ρ)

Input: query Q, API graph $G = (V_G, E_G)$, similarity threshold α, node penalty ρ
Output: subgraph $g \subseteq G$

1 $q_{rare}, S \leftarrow \text{FindSupport}(Q, G, \alpha)$
2 **foreach** $c \in S(q_{rare})$ **do**
3 $L_c \leftarrow \emptyset$
4 **foreach** $q \in Q$ *and* $q \neq q_{rare}$ **do**
5 $v^* \leftarrow \arg\max_{v \in S(q)} Gain_L(v, q, c)$
6 $L_c \leftarrow L_c \cup \{Path(c, v^*)\}$
7 **end**
8 **end**
9 $L_c^* \leftarrow \arg\max_{L_c} Gain(L_c, Q, \rho)$
10 **return** $g = L_c^*$

graph is not unique. Consider the query Q in Figure 1. From the center node 0, for the phrase q_1, RGF may select node 1 or 3, as $Gain_L(v_1, q_1, v_0) = Gain_L(v_3, q_1, v_0) = 0.8$ are the highest. Suppose node 3 is selected, and then node 0 is connected to node 3. Next, for q_2, it will select node 6 according to *LocalGain*. However, the shortest path $Path(0, 6)$ is not unique in Figure 1. As mentioned previously, there are three shortest paths: $(0, 1, 4, 6)$, $(0, 2, 4, 6)$ and $(0, 3, 4, 6)$, all of which have the same distance of 3. Suppose RGF chooses $Path(0, 6) = (0, 1, 4, 6)$ to add to the subgraph. Figure 3 shows the resultant subgraph found by RGF. This subgraph is less optimum, as dashed node 3 is redundant to node 1. This is due to the multiple shortest path phenomenon: when there are multiple shortest paths from a source node to a destination node, an arbitrary one will be selected to add to the resultant subgraph. If the pairwise shortest paths are pre-computed and stored to support queries, only one path between a pair of nodes will be retained. It is hard to decide which shortest path should be stored among multiple choices, because at the pre-computation phase, we know nothing about a query.

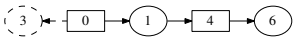


Figure 3: Less optimum subgraph from RGF

To address the multiple shortest path problem, we propose a refinement technique using *Steiner Tree*. The *Steiner Tree* problem [7] is to find a minimum-weight tree connecting a given subset of nodes of a graph.

We name this refinement technique as *LocalRegionRefine* (LRR), as shown in Algorithm 3. Initially, we start with a subgraph g' returned by Algorithm RGF (line 1). Then, from the node set $V_{g'}$, we select a subset of nodes as V_{best} by including nodes with highest textual similarity to each phrase $q \in Q$ (lines 4–6). Then, we invoke the *SteinerTree* procedure to find a refined graph g (line 9), where V_{best} is the input as the set of necessary nodes for a steiner tree. The graph can be refined as *SteinerTree* is multiple-path-aware: it always connects any covered node (in V_g) to an uncovered necessary node (in V_o) through a shortest path with the smallest distance.

Example 5. Consider the graph g' returned by RGF in Figure 3. Necessary nodes are $V_{best} = \{0, 3, 6\}$ from nodes $\{0, 1, 3, 4, 6\}$. Node 3 has a higher textual similarity than node 4 (0.8 vs. 0.5) to the phrase q_1='create sql statement', so we choose node 3 accordingly. By feeding $V_{best} = \{0, 3, 6\}$ to the *SteinerTree* procedure, we can get an optimum subgraph as shown in Figure 2(b). The refinement is possible because *SteinerTree* can tell that the shortest path $Path(3, 6) = (3, 4, 6)$ is shorter than $Path(0, 6) = (0, 1, 4, 6)$ when adding $Path(3, 6)$ from the covered node 3 to the uncovered necessary node 6 to g. Note that another possible necessary node set can be $V_{best} = \{0, 1, 6\}$, as node 1 has the same textual similarity (0.8) as node 3. In this case, the refined graph should contain nodes 0, 1, 4, 6, which is essentially the same (has the same *Gain*) as Figure 2(b).

Algorithm 3: LocalRegionRefine(Q, G, α, ρ)

Input: query Q, API graph $G = (V_G, E_G)$, similarity threshold α, node penalty ρ
Output: subgraph $g \subseteq G$

1 $g' \leftarrow RarestGainFirst(Q, G, \alpha)$
2 $V_{best} \leftarrow \emptyset$
3 **foreach** $q \in Q$ **do**
4 $V_q = \{v | v \in S(q) \wedge v \in V_{g'}\}$
5 $v^* \leftarrow \arg\max_{v \in V_q} Sim(q, v)$
6 $V_{best} \leftarrow V_{best} \cup \{v^*\}$
7 **end**
8 $v^* \leftarrow \arg\min_{v \in V_{best}, u \in V_{best}, u \neq v} Dist(u, v)$
9 $g \leftarrow SteinerTree(V_{best}, G, v^*)$
10 **return** g

11 **Procedure** SteinerTree(V_o, G, v')
12 $V_o \leftarrow V_o \setminus \{v'\}$
13 $g \leftarrow \{v'\}$
14 **while** $V_o \neq \emptyset$ **do**
15 $u^*, v^* \leftarrow \arg\min_{u \in V_g, v \in V_o} Dist(u, v)$
16 **if** $Path(u^*, v^*) \neq \emptyset$ **then**
17 $g \leftarrow g \cup \{Path(u^*, v^*)\}$
18 $V_o \leftarrow V_o \setminus \{v^*\}$
19 **end**
20 **end**
21 **return** g

4. CLASS-ONLY PATH INDEXING

Our proposed approaches rely on online query of shortest distances and paths between two graph nodes. The time complexity of single source shortest paths to all other destination nodes is $O(n + m)$ using breadth first search (BFS) for unweighted graphs and $O(n \log n + m)$ using Dijkstra's algorithm (implemented with Fibonacci heaps) for positive weighted graphs, where n and m are the number of nodes and edges of a graph respectively. In our application, an API graph constructed from Java SE 1.6 has a large $n \simeq 32,000$. Algorithm LRR needs to compute shortest paths and distances between many pairs of graph nodes before returning a resultant subgraph, so online shortest path computation is not feasible given limited query response time for a large n. On the other extreme, storing all pairs of shortest paths and distances in the main memory is not feasible given limited memory space, as the space complexity is at least $O(n^2)$ to index all pairwise shortest paths/distances.

There have been numerous studies on approximate shortest distance estimation. A representative approach is *landmark embedding*, which pre-computes and stores shortest distances (or paths) from a subset of graph nodes called

landmarks to every node in the graph [5, 19]. During on-line query, an approximate shortest distance between two nodes u and v is obtained indirectly using *triangular inequality* by

$$\widetilde{Dist}(u,v) = \min_{w \in V_L} Dist(u,w) + Dist(w,v),$$

where V_L is the landmark set. In general graphs, the approximation can have an error bound on the estimated shortest distances [19]. Although landmark embedding uses less indexing space than full path/distance indexing, this approximate distance estimation approach may affect the quality of the resultant graph in our problem, as the estimated shortest distances and paths are not precise.

To design an effective scheme to index exact shortest distances/paths instead of approximate ones, we can have some observations from the API graph G in Figure 1. G has two types of nodes: classes $C = \{0, 4, 8, 9\}$ and methods $M = \{1, 2, 3, 5, 6, 7, 10, 11\}$. Moreover, there is no direct interaction between methods, i.e., the adjacent nodes of a method node must be class nodes. Therefore, we propose to only index paths among class nodes in an API graph, which is more space-efficient and can support full recovery of exact shortest paths between any type of nodes. Note that shortest distance can be computed trivially from a shortest path.

4.1 Three Indexing Structures

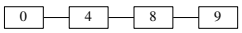


Figure 4: Class graph of Figure 1

First, we construct a *class graph* $G_C = (V_{G_C}, E_{G_C})$ from the original API graph $G = (V_G, E_G)$. The formal construction process is in Definition 5. Figure 4 shows the class graph of the API graph in Figure 1.

DEFINITION 5. ***(Class Graph)**: A Class Graph for an API graph $G = (V_G, E_G)$ is a weighted undirected graph $G_C = (V_{G_C}, E_{G_C})$, where $V_{G_C} = \{v | v \in V_G, v \in C\}$, and $\exists$edge $e(u,v) \in E_{G_C}$ for $u, v \in V_{G_C}$ with an edge weight $Dist(u,v)$ iff $1 \leq Dist(u,v) \leq 2$ in G.*

Finally, based on the class graph G_C, we construct three space efficient indexing structures:

1. **Parent**[**s**][**d**]: For every class node pair (s, d) in the class graph, $Parent[s][d]$ stores the precedent node of d on the shortest path from s to d.
2. **Adj**[**m**]: A mapping from a method node $m \in M$ to its adjacent (1-hop neighbor) class nodes.
3. **Con**[**o**][**p**]: A mapping from a 2-hop class node pair (o, p) to only one and any one of their in-between connector method nodes (if any). Here $o, p \in C$ and $Dist(o,p) = 2$ in the original API graph. As the API graph is an undirected graph, we only store $Con[o][p]$ for $o < p$ according to their node ids. $Con[p][o]$ will be the same and thus not stored.

Among the three indexing structures, $Parent[s][d]$ requires more memory space ($O(|C|^2)$), as it indexes the shortest paths between class nodes in C with size of $|C|$. As the number of classes is usually much smaller than that of methods in an API library, our class-only path indexing scheme is more space-efficient than path indexing on the original API graph (full graph indexing).

Example 6. In the class graph in Figure 4, $Path(4,9) = (4,8,9)$. So $Parent[4][9] = 8$, i.e., the precedent node of 9 is node 8 on $Path(4,9)$. Some $Adj[m]$ mappings include $Adj[1] = \{0,4\}$, $Adj[5] = \{4,8\}$ and $Adj[10] = \{8,9\}$. Some $Con[o][p]$ mappings include $Con[0][4] = 1$, $Con[4][8] = 5$ and $Con[8][9] = 10$ according to Figure 1.

4.2 Exact Path Recovery

Algorithm 4 shows how to recover the exact path between two nodes. There are three cases in path recovery: (1) between 2 classes; (2) between 1 method and 1 class; and (3) between 2 methods. Since the indexing is based on the class graph, for any case, we have to query $Con[o][p]$ to add the connector method back to the actual path between two class nodes (line 9). For case 2, if one of the two nodes is a method node, denoted as v, we use $Adj[v]$ to map v to its adjacent class nodes as references $Ref[v]$ and search a minimum shortest path among shortest paths from every reference to the other class node (line 16). This is similar for case 3, in which both source and destination nodes need to get references, i.e., adjacent class nodes, for searching shortest paths based on $Parent[s][d]$ index.

Algorithm 4 provides an efficient way to recover the exact shortest path between two nodes in an API graph. The worst case runtime for exact path recovery is case 3 and it is $O(|Adj[m]| \times |Adj[n]|)$ for two method nodes m and n. In practice, the cardinality $|Adj[m]|$ of a method is small (say 3), so the runtime to recover an exact path is only several times to that of full-graph path indexing, while we can store the class-only indexing compactly in main memory.

Example 7. Consider Figure 1. By applying path recovery in Algorithm 4, we find $Path(4,8) = (4,5,8)$, i.e., $(4, Con[4][8], 8)$; $Path(5,11) = (5,8,10,9,11)$, i.e., adding the source node 5 and destination node 11 to head and tail of another path $(8, Con[8][9], 9)$ respectively.

Algorithm 4: PathRecovery(s, d)

```
   Input: source node s, destination node d
1  if s = d then
2      return {s}
3  end
4  foreach u ∈ Ref[s] do
5      foreach v ∈ Ref[d] do
6          Path_uv ← ∅
7          n_p ← v
8          while n'_p ← Parent[u][n_p] do
9              add n_p and Con[n'_p][n_p] (if any) to Path_uv
10             n_p ← n'_p
11         end
12         add s to head of Path_uv if s ∈ M
13         add d to tail of Path_uv if d ∈ M
14     end
15 end
16 return Path_uv with min_{u∈Ref[s],v∈Ref[d]} |Path_uv|
```

5. EXPERIMENTS

The closest related work in API (Code) recommendation is *Portfolio* (PF) [15]. A live version of Portfolio can be

found at http://www.searchportfolio.net, which is targeted at searching functions in source code. It also provides context among functions when given text as the query, but does not consider the connectivity of function calling subgraph. To have a fair comparison between the proposed API recommendation approach *LocalRegionRefine* (LRR) and PF, we implemented LRR and re-implemented PF, both in Python. Then, we create queries based on sample API usage code available from three popular Java API tutorial websites: Kodejava[3], Javadb[4] and Java2s[5]. For each of the queries, we construct the underlying relevant nodes (methods or classes) and corresponding API subgraph using invocations among nodes in the sample code. Afterwards, we evaluate LRR and PF using standard evaluation metrics for information retrieval, i.e., *precision* and *recall*. All experiments were carried out on Ubuntu 11.10 (Linux OS) with a Core 2 Duo 2.26GHz CPU and 3GB DDRII RAM.

Suppose subgraph H is the optimum subgraph (ground truth) for a query, the precision and recall for a suggested subgraph G are as follows:

$$Precision(G) = \frac{|V_G \cap V_H|}{|V_G|},\ Recall(G) = \frac{|V_G \cap V_H|}{|V_H|}$$

We re-implemented Portfolio by following the linear combination of *PageRank* (PR) [2] and *Spreading Activation Network* (SAN) [4] in Eq. 5 to rank a node (method or class) in an API graph. SAN is to propagate the weight (i.e., textual similarity) of a node in the API graph to its neighbours with a decay factor δ. We follow the exact SAN used in [15], where the highest propagated weight of a node is retained when there are multiple propagation paths. Since the query of Portfolio in [15] is one text phrase and that of ours is a set of text phrases, we use nodes that pass similarity threshold α in Eq. 1 for each phrase in the query as initiating (firing) nodes for SAN. Afterwards, top-k most similar nodes are returned, together with a visualization of the subgraph (may not be connected) only containing these nodes and edges for any two adjacent nodes. The specific parameters for Portfolio in the experiments are as follows: $\lambda_{PR} = \lambda_{SAN} = 0.5$, $k = 10$, $\delta = 0.8$ and teleport probability for PageRank = 0.15.

$$Sim_{Port}(v) = \lambda_{PR}\frac{PR(v)}{\max_{u \in V_G} PR(u)} + \lambda_{SAN} SAN(v) \quad (5)$$

5.1 Query Formulation

These queries are organized in two groups:

Group I Queries. Group I consists of queries at low textual similarity threshold $\alpha = 0.6$ to tolerate a query phrase q that is not very similar to BOWs of a method/class. This illustrates a scenario that the user is not familiar with the API library. The queries in Group I are originated from 20 pieces of sample usage code in Kodejava. Each piece of sample API usage code in Kodejava is associated with a short question expressed in text. These questions are used to formulate the queries that preserve what is expressed in the questions. For example, the question for example 77[6] is *'How do I get total space and free space of my disk?'*, so the query becomes ['get total space', 'get free space']. For each of these queries, we obtain an optimum subgraph from corresponding piece of sample usage code: methods or classes (in the sample usage code) that match each phrase the best (highest textual similarity) are required nodes and invocations among these required nodes form the resultant subgraph. Other methods/classes appearing in the sample usage code but not participating in the invocations are not taken into account. All the nodes in the optimum subgraph for each query are the relevant nodes for evaluation.

To select these 20 pieces of sample code, we first crawl a total of 849 pieces of sample code from Kodejava. Then, for each piece of the sample code, we remove stop words, punctuations and the phrase 'How do I' from the associated question. Afterwards, we check if 95% of the remaining words appear as substrings in the source code. If yes, we include the corresponding piece of sample code in the candidate pool for query selection. Finally, from a total of 83 pieces of sample code in the candidate pool, we select 20 pieces of sample code by including various packages in the JSE 1.6 library.

Group II Queries. Group II consists of queries at high textual similarity threshold $\alpha = 0.8$ to indicate another common scenario [14, 25] that the user is familiar with the naming of methods/classes but not sure about the calling relationships among them. We construct the optimum subgraph for each of Group II queries by referring to invocations among methods/classes in the sample API usage code in Javadb and Java2s. We first locate some methods/classes in the sample code and use the invocations among them as the corresponding subgraph. Group II consists of a total of 20 queries, 10 from Javadb and 10 from Java2s. To select these queries, we first check if there are at least two classes that are interacting with each other via a method in the sample code. If yes, we use the BOWs of each of these classes as the BOWs of each phrase of a query. Moreover, we limit the number of queries per JSE package to 3 for sampling.

Figure 5 shows the number of Group I and Group II queries in various JSE packages. Table 1 provides a snapshot of 4 queries from Group I and 3 queries from Group II. For each of these 4 queries in Group I, we show the question and corresponding query phrases in two separate rows.

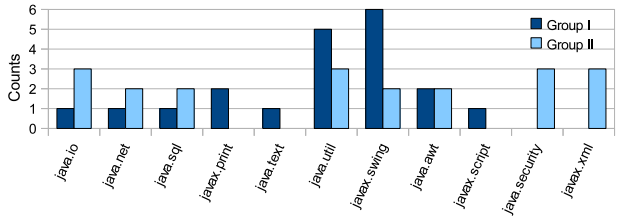


Figure 5: Number of Group I and Group II queries in various JSE packages

5.2 Results – Effectiveness

Since PF returns top-k most similar nodes w.r.t. the query phrases, and k is set to 10 in our experiments, if we measure top-10 precision, the precision could be low when the size of the underlying optimum subgraph is small. To be fair to PF, we first find the best F_1-measure among all F_1-measures for $k = 1$ to 10. Afterwards, we use the corresponding precision and recall as results. On the other hand, since our proposed approach LRR always returns a connected subgraph w.r.t. a query regardless of parameter k, we simply calcu-

[3]http://www.kodejava.org
[4]http://www.javadb.com
[5]http://www.java2s.com
[6]http://www.kodejava.org/examples/77.html

Table 1: Sample Group I and Group II queries

Group I	
Id	**Question & Query**
5	How do I get IP address of localhost?
	get ip address, local host
10	How do I set a filter on a logger handler?
	set filter, log handler, add handler
15	How do I load properties from XML file?
	load properties from xml, properties
20	How do I read entries in a zip / compressed file?
	read entries, zip file
Group II	
Id	**Query**
4	X Path Factory, X Path
8	Date, Simple Date Format
12	Properties, Input Stream

late precision and recall using all the nodes of the suggested subgraph.

Table 2: Avg. precision, recall and F_1-measure for PF and LRR

	Portfolio			LRR		
	Prec.	**Rec.**	**F_1**	**Prec.**	**Rec.**	**F_1**
Group I	0.24	0.51	0.33	0.53	0.56	0.54
Group II	0.48	0.64	0.55	0.79	0.72	0.75

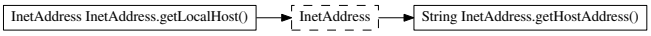
Table 2 shows the summary of the results of both Group I and II queries. Moreover, Figure 6 shows the detailed results for each of Group I and II queries. For Group I queries, PF and LRR have similar average recall at 0.51 and 0.56 respectively, but LRR improves the average precision of PF to more than twice (from 0.24 to 0.53). When the query size is 1, PF can have better precision and recall than those of LRR, such as queries 14 and 16. It is because PF outputs top-k ($k = 10$) nodes while LRR only returns 1 best node. On the other hand, when the query size is increased to 2 and 3, LRR has better precision and recall as it tries to find a connected subgraph to match the optimum subgraph. There are also some cases where both PF and LRR have zero precision and zero recall, as the textual similarity between a phrase in the query is very different from a node in the optimum subgraph, such as queries 6 and 11. For Group II queries, LRR has a better average recall of 0.72 and improves the average precision of PF by 67% to 0.79. On average, LRR improves the F_1-measure of PF by 64% in Group I and 36% in Group II respectively.

Table 3: Avg. Gain for PF, LRR and Optimum

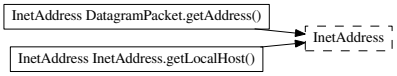
	Portfolio	**LRR**	**Optimum**
Group I	0.43	0.88	0.98
Group II	0.56	0.90	0.90

Meanwhile, Table 3 shows the average Gain (Eq. 2) of PF, LRR and the optimum subgraph constructed from sample API usage code for both Group I and II queries. While the average recall of LRR is 0.56 for Group I queries in Table 2, the average Gain of LRR is very close to that of the optimum (0.88 vs. 0.98). For Group II queries, LRR has an identical average Gain as that of the optimum. On the other hand, PF has much lower Gain since it is not connectivity aware.

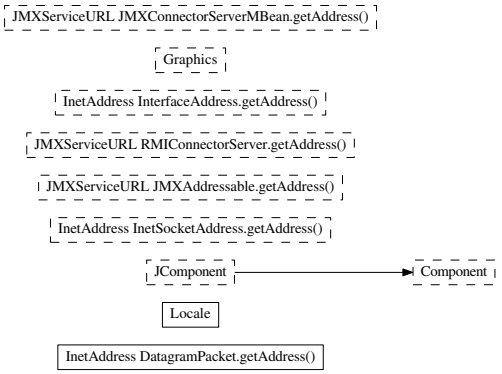
To visualize the benefits of connectivity of a subgraph, we show the resultant API subgraphs among PF, LRR and optimum subgraph for query 5 in Group I and query 8 in Group II in Figure 7 and Figure 8 respectively[7]. In the figures, necessary nodes are in solid box shape and dummy nodes are in dashed box shape. For both cases, PF fails to find a connected subgraph for necessary nodes.



(a) Optimum



(b) LRR



(c) Portfolio

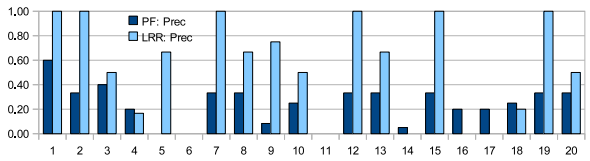
Figure 7: Solutions for query 5 in Group I

5.3 Results – Efficiency

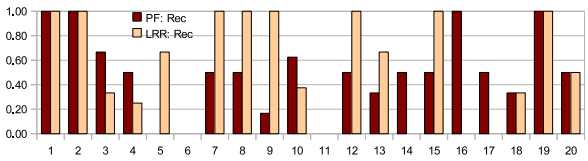
5.3.1 Runtime – using Class Graph Indexing

For a comparison of runtime among the algorithms RF, RGF, LRR and PF, we generate a set of random queries $\mathbb{Q}$ from CG (*class graph*) of JSE library. $\mathbb{Q}$ has a total of 180 queries whose sizes range from 3 to 20 and the number of queries per size is 10. To ensure there is at least one resultant connected API subgraph for each $Q \in \mathbb{Q}$, we first select a pool of candidate nodes V_c, in which each node $v \in V_c$ has at least 50 (sufficiently larger than maximum query size of 20) children nodes in the shortest path tree PT_v rooted at it. Then, from PT_v of a randomly selected node $v \in V_c$, we randomly pick a set of nodes V_Q with size equal to that of a query Q. The associated bag-of-words (BOWs) of each $v \in V_Q$ is the corresponding BOWs of each $q \in Q$. We further ensure Q has at least 5 possible resultant connected

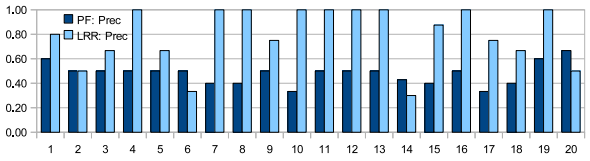
[7] *Porter2 stemming algorithm* [17] for English is applied to BOWs of nodes in the API graph and query phrases in our implementation for the similarity measure in Eq. 1



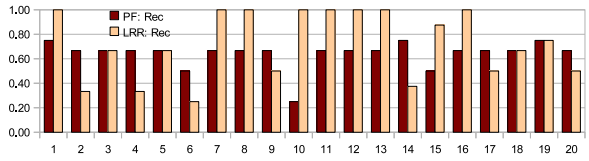
(a) Group I - Precision



(b) Group I - Recall



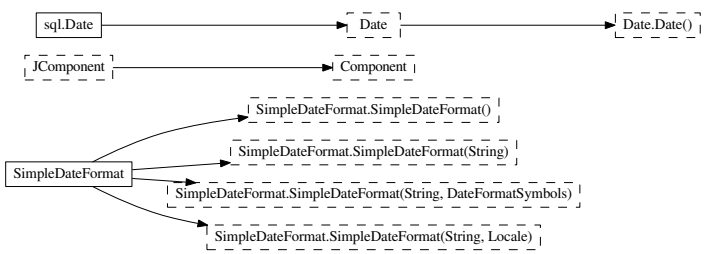
(c) Group II - Precision



(d) Group II - Recall

Figure 6: Precision and recall for Group I and II queries. (X-axis: query Id; Y-axis: precision or recall)

(a) Optimum, LRR



(b) Portfolio

Figure 8: Solutions for query 8 in Group II

subgraphs (local regions) at a similarity threshold $\alpha = 0.6$. We name this dataset for JSE as $\mathbf{DS_{JSE}}$.

Table 4: Avg. runtime (seconds) for all queries at varying α for dataset DS_{JSE}

α	RF	RGF	LRR	PF
0.5	0.269	0.282	0.293	1.774
0.6	0.033	0.038	0.052	0.519
0.7	0.012	0.013	0.024	0.203
0.8	0.012	0.012	0.025	0.176

Using dataset DS_{JSE}, Table 4 shows the comparison of average runtime over all queries for RF, RGF, LRR and PF at $\alpha = 0.5$, 0.6, 0.7 and 0.8 respectively, where RF, RGF and LRR use *class graph* indexing to store shortest paths in the main memory. Without refinement process using *SteinerTree*, RGF has comparable runtime to those of RF. With *SteinerTree* refinement, LRR has some overhead but the worst runtime is only ∼2 times to that of RF at $\alpha = 0.8$. Runtime for PF is much longer than that of LRR, ranging from 6 to 10 times by varying α.

5.3.2 Indexing: Class Graph Vs. Full Graph

How space-efficient is the proposed CG (*class graph*) indexing? A naive path indexing for LRR is to store all pairwise shortest paths in the original full API graph into a database on hard disk. We name this disk-based approach FG (*full graph*) indexing. The size of FG indexing is measured as follows. Given an API graph $G = (V_G, E_G)$, for every source node $s \in V_G$, we compute its shortest paths to every other destination node $d \in V_G$ and use $Parent[s][d]$ to store the precedent node of d in $Path(s, d)$. Afterwards, we dump the serialized parent entries ($Parent[s][d]$) with source node s as primary key to a sqlite3 database DB_{path} and measure its size as the space required for FG indexing.

For the JSE 1.6 library, FG indexing uses 10.6GB disk space, while CG indexing can fit the indexes compactly in main memory for 150MB space. In other words, CG indexing can reduce the space by ∼70 times.

Next, we study the trade-off in runtime between these two indexing schemes, i.e., the disk-based FG indexing and the main-memory-based CG indexing. Table 5 shows the corresponding results using dataset DS_{JSE} at $\alpha = 0.6$ and $\rho = 0.2$. On average, the runtime of disk-based LRR is ∼16 times to that of main-memory-based LRR using the proposed CG indexing. The runtime ranges from 9 to 24 times for various sizes of queries.

5.4 Threats to Validity

Construct Validity. Are the evaluation measures suitable? We reduce threats to construct validity by using the popular performance metrics, precision and recall for evaluation. In addition, we use a reasonable subgraph goodness measure called *Gain* as a supplement.

Internal Validity. Are there any experimental biases? The first one is query formulation and relevance judgement. We use sample code available from three websites as references to annotate the queries and relevance nodes. Con-

Table 5: Runtime (seconds) of LRR using FG indxing and LRR using CG indexing at $\alpha = 0.6$ and $\rho = 0.2$ for dataset DS_{JSE}

Size	Full Graph	Class Graph	$\text{Ratio}_{FG:CG}$
03-05	0.590	0.029	20.57
06-08	0.847	0.035	24.24
09-11	0.957	0.044	21.81
12-14	0.511	0.056	9.06
15-17	0.611	0.070	8.76
18-20	0.964	0.093	10.41
		Average $\text{Ratio}_{FG:CG}$	15.81

structing ground truths from sample code shall be a more objective evaluation approach than human judgements of relevance. As the queries are formulated by human (based on questions from Kodejava), the phrases constructed for a query may affect the results so the precision and recall can drop or increase. Since both PF and LRR receive the same set of phrases for each query, the effects will be evenly propagated to both PF and LRR but not only one of them. The final threat is the parameter settings of PF and LRR. We use an equally linear combination of *PageRank* and *Spreading Activation Network* for PF. Changing the weight of the combination may affect the behaviour in the similarity measure and hence the nodes of a suggested subgraph by PF.

External Validity. Could the results be generalized? We included various packages in JSE 1.6 library. However, we have not evaluated other Java libraries and API libraries of non-Java languages, though the idea of LRR is applicable to API of any languages. Besides, the queries may not be representative samples to common API querying habits of the majority.

6. RELATED WORK

API/Code Recommendation. Popular code search engines include Google Code Search[8], Koders[9] and Codase[10]. They allow users to search from a large collection of code repositories for common programming languages but only use simple word matching. More complex and useful API/ Code recommendation tools can be roughly divided into two groups.

The first group uses natural language as a query, such as Sourcerer [1], Gridle [18], Assieme [6], the work of McMillan et al. [16] and a cross-library API recommendation approach [24]. They can help users to overcome selection barrier to suggest related APIs. Sourcerer supports 5 types of searches using keyword(s), including components, component use, functions, function use and program structures. Gridle uses a variant of *PageRank* [2] to suggest popular classes that match the query words. Assieme groups the suggested results tidily into packages, types, members and code examples as well as inter-group hyperlinks. The work of McMillan et al. [16] suggests source code examples in 2 steps, by first locating a set of candidate APIs that are textually similar to the query and then finding code examples that cover most of these APIs. Similar to Gridle and the work of McMillan et al., we also take the structure of API calling graph into similarity measure using *Gain*.

The second group uses method/class name as a query, such as Prospector [14], MAPO [25], Altair [13] and FACG (Flow-Augmented Call Graph) [23]. Prospector suggests a list of ranked shortest paths given two methods to indicate method invocations. MAPO suggests frequent API usage patterns and associated code snippets. Both Prospector and MAPO provide context of functions of interest, which can be considered as the connected API subgraph suggested by our approach. Altair and FACG suggest similar API functions (no context) to the query function using *common functions overlapping* and *weighted API call graph* respectively.

The most related work in API/Code recommendation to our work is Portfolio [15], as briefly discussed in Section 5. We further consider to output a connected subgraph from the API graph.

Team Formation. Given a *task*, a pool of *experts* with different skills and possibly a social network that captures the proximity among them, *team formation* studies how to find a subset of experts who can jointly fulfill the task.

In the field of Operations Research, team formation problem is formulated as Integer Programming and solved approximtely using techniques such as analytical hierarchy process [3, 26] and genetic algorithms [21]. The work in [11] is the first team formation problem accounting for the structure of a graph that models communication costs among experts. We improve the *RarestFirst* algorithm in [11] to apply to API recommendation domain. Two extended work of [11] are [12, 8]. The work in [12] considers a general task that each skill requires a number of experts and solves greedily by applying an enhanced version of *SteinerTree* to a compressed group graph. The work in [8] considers the skillfulness of experts in discrete level, and linearly combines skillfulness with node distance in the expertise network. We also account for skillfulness, i.e., textual similarity of BOWs between a method/class and a query phrase. Meanwhile, the team formation problem accounting for graph structure is similar to keyword search in relational database [22, 9], but with a focus on expertise network.

7. CONCLUSIONS

We proposed a novel graph search approach to help users find usages of APIs only using simple text phrases as a query. Our solution allows software developers to pass through both selection and coordination barriers when reusing APIs of existing libraries. For a query, we suggested a high quality subgraph from the API graph based on the newly defined similarity measure called *Gain*. We improved the greedy subgraph search algorithm *RarestFirst* with a refinement process based on *SteinerTree* technique. Moreover, we design a compact index based on *Class Graph* of an API graph, which can efficiently support on-demand shortest path queries with a small index size. Experiments confirmed that our approach outperforms the state-of-the-art code recommendation approach *Portfolio* by improving average F_1-measure by 64% and average Gain by 2 times.

8. ACKNOWLEDGMENTS

This work is supported by the Hong Kong Research Grants Council (RGC) General Research Fund (GRF) Project No. CUHK 411310 and 411211.

[8]http://www.google.com/codesearch. Service is closed
[9]http://www.koders.com
[10]http://www.codase.com

9. REFERENCES

[1] 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 *OOPSLA Companion*, pages 681–682, 2006.

[2] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 30(1-7):107–117, 1998.

[3] S.-J. Chen and L. Lin. Modeling team member characteristics for the formation of a multifunctional team in concurrent engineering. *IEEE Transactions on Engineering Management*, 51(2):111 – 124, 2004.

[4] F. Crestani. Application of spreading activation techniques in information retrieval. *Artif. Intell. Rev.*, 11(6):453–482, 1997.

[5] K. Grant and D. Mould. Combining heuristic and landmark search for path planning. In *Future Play*, pages 9–16, 2008.

[6] R. Hoffmann, J. Fogarty, and D. S. Weld. Assieme: finding and leveraging implicit references in a web search interface for programmers. In *UIST*, pages 13–22, 2007.

[7] F. K. Hwang, D. S. Richards, and P. Winter. *The Steiner Tree Problem*. North-Holland, Amsterdam, Netherlands, 1992.

[8] K. Kamel, N. Tubaiz, O. AlKoky, and Z. AlAghbari. Toward forming an effective team using social network. In *IIT*, pages 308 –312, 2011.

[9] M. Kargar and A. An. Keyword search in graphs: Finding r-cliques. *PVLDB*, 4(10):681–692, 2011.

[10] A. J. Ko, B. A. Myers, and H. H. Aung. Six learning barriers in end-user programming systems. In *VL/HCC*, pages 199–206, 2004.

[11] T. Lappas, K. Liu, and E. Terzi. Finding a team of experts in social networks. In *KDD*, pages 467–476, 2009.

[12] C.-T. Li and M.-K. Shan. Team formation for generalized tasks in expertise social networks. In *SocialCom/PASSAT*, pages 9–16, 2010.

[13] F. Long, X. Wang, and Y. Cai. Api hyperlinking via structural overlap. In *ESEC/SIGSOFT FSE*, pages 203–212, 2009.

[14] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the api jungle. In *PLDI*, pages 48–61, 2005.

[15] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu. Portfolio: finding relevant functions and their usage. In *ICSE*, pages 111–120, 2011.

[16] C. McMillan, D. Poshyvanyk, and M. Grechanik. Recommending source code examples via api call usages and documentation. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, pages 21–25, 2010.

[17] M. F. Porter. Readings in information retrieval. chapter An algorithm for suffix stripping, pages 313–316. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[18] D. Puppin and F. Silvestri. The social network of java classes. In *SAC*, pages 1409–1413, 2006.

[19] M. Qiao, H. Cheng, and J. X. Yu. Querying shortest path distance with bounded errors in large graphs. In *SSDBM*, pages 255–273, 2011.

[20] C. Scaffidi. Why are apis difficult to learn and use? *ACM Crossroads*, 12(4):4, 2006.

[21] H. Wi, S. Oh, J. Mun, and M. Jung. A team formation model based on knowledge and collaboration. *Expert Syst. Appl.*, 36(5):9121–9134, 2009.

[22] J. X. Yu, L. Qin, and L. Chang. Keyword search in relational databases: A survey. *IEEE Data Eng. Bull.*, 33(1):67–78, 2010.

[23] Q. Zhang, W. Zheng, and M. R. Lyu. Flow-augmented call graph: A new foundation for taming api complexity. In *FASE*, pages 386–400, 2011.

[24] W. Zheng, Q. Zhang, and M. R. Lyu. Cross-library api recommendation using web search engines. In *SIGSOFT FSE*, pages 480–483, 2011.

[25] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. Mapo: Mining and recommending api usage patterns. In *ECOOP*, pages 318–343, 2009.

[26] A. Zzkarian and A. Kusiak. Forming teams: an analytical approach. *IIE Transactions*, 31:85–97, 1999.